

Introduction to Software Architecture

Introduction to Software Architecture

Key concepts, principles, and common styles – concise, readable notes

1. What Is Software Architecture?

Aspect	Description
Definition	The high-level structure of a software system, describing its components, their responsibilities, and how they interact.
Purpose	Guides design decisions, ensures quality attributes (performance, security, etc.), and provides a shared vision for the development team.
Stakeholders	Architects, developers, testers, project managers, operations, customers, and sometimes regulators.
Artifacts	Architectural diagrams, documentation (views, rationale), style guides, and decision records.

2. Core Architectural Concepts

Concept	Meaning	Typical Questions
Components / Modules	Independent units of functionality (e.g., services, libraries, microservices).	What does each component do?
Connectors	Mechanisms for communication (APIs, message queues, RPC, event streams).	How do components talk to each other?
Configurations	Arrangement of components and connectors in a particular deployment.	Where are components deployed?
Views	Different perspectives (logical, development, process, physical) to address stakeholder concerns.	Which view is relevant for me?
Architecture Decisions	Recorded rationale for choosing a style, technology, or pattern.	Why was this choice made?

3. Quality Attributes (Non-Functional Requirements)

Introduction to Software Architecture

Attribute	What It Affects	Typical Architectural Levers
Performance	Response time, throughput	Caching, async processing, load balancing
Scalability	Ability to handle growth	Horizontal scaling, stateless services
Reliability / Availability	Fault tolerance, uptime	Redundancy, circuit breakers, graceful degradation
Security	Confidentiality, integrity, authentication	Encryption, zero-trust networking, role-based access
Maintainability	Ease of change, readability	Modularization, clear interfaces, coding standards
Portability	Running on different platforms	Containerization, platform-agnostic libraries
Testability	Ability to verify behavior	Dependency injection, mockable interfaces

Tip: Use the *ATAM* (Architecture Tradeoff Analysis Method) to evaluate how well a design meets these attributes.

4. Architectural Styles & Patterns

Style / Pattern	Core Idea	Typical Use Cases
Layered (N-tier)	Organize code into horizontal layers (presentation ? business ? data).	Enterprise apps, CRUD systems
Client-Server	Separate requestors (clients) from providers (servers).	Web apps, mobile backends
Microservices	Small, independently deployable services communicating over lightweight protocols.	Large, evolving systems needing independent scaling
Service-Oriented Architecture (SOA)	Reusable services with a shared bus (often SOAP/WS*).	Enterprise integration, legacy systems
Event-Driven	Components react to events via pub/sub or message queues.	Real-time analytics, IoT, decoupled workflows

Introduction to Software Architecture

Style / Pattern	Core Idea	Typical Use Cases
Domain-Driven Design (DDD) – Hexagonal / Clean Architecture	Center business logic, isolate external concerns.	Complex domains, high maintainability
Pipe-Filter	Data flows through a sequence of processing stages (filters).	Compilers, ETL pipelines
Broker	Mediator (broker) routes requests to appropriate components.	Middleware, message brokers
**MVC /		