Introduction to Software Architecture. Definition and importance, architect roles

Definition of Software Architecture

- **Common definition**:

"Software architecture is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution." (based on ISO/IEC/IEEE 42010)

- **Includes**:
    - Key components and their responsibilities
    - Interfaces and communication patterns
    - Architectural principles and rationale
    - Constraints and crosscutting concerns
- **It's about**:
    - **Structure** (what exists)
    - **Behavior** (how it interacts)
    - **Rationale** (why it exists this way)

Importance of Software Architecture

- **Manages complexity**:
  Helps you reason about large systems, especially those with multiple stakeholders, modules, or technologies.
- **Enables communication**:
  A well-documented architecture serves as a communication medium between:
    - Developers
    - Stakeholders
    - Operations/Deployment teams

- **Facilitates change**:
  A good architecture anticipates and accommodates changes in requirements, technologies, or team structures.
- **Ensures quality attributes**:
  Many non-functional requirements (e.g. performance, scalability, security) can only be met through architectural decisions.

Roles and Responsibilities of a Software Architect

- **Technical leadership**
  - Define and evolve the system structure
  - Make technology choices
  - Ensure consistent design principles
- **Mediator**
  - Align business goals with technical possibilities
  - Balance trade-offs between competing quality requirements (e.g., performance vs. maintainability)
- **Coach and communicator**
  - Guide the development team on architectural intent
  - Communicate with stakeholders and product owners
  - Ensure traceability and justification of key decisions
- **Guardian of quality**
  - Enforce crosscutting concepts (e.g., security, logging)
  - Drive architecture evaluations (e.g., ATAM-lite)
  - Own the architecture documentation and decision records

Architectural Requirements. Quality attributes, ISO 25010, ATAM-lite

Types of Requirements

- **Functional Requirements**
  Define *what* the system should do (features, use cases).
- **Non-Functional Requirements (Quality Requirements)**
  Define *how well* the system performs (also called **quality attributes**). These shape the architecture far more than functional ones.

Quality Attributes

(Also referred to as *-ilities*, or quality goals)

Examples include:

Category	**Attribute Examples**
Runtime	Performance, Availability, Scalability
Design-time	Modifiability, Testability, Maintainability
Operational	Deployability, Configurability
Crosscutting	Security, Usability, Portability

ISO/IEC 25010:2011 – Quality Model

A standard model that defines **8 quality characteristics** for software:

1. **Functional suitability**
2. **Performance efficiency**
3. **Compatibility**
4. **Usability**
5. **Reliability**

6. **Security**
7. **Maintainability**
8. **Portability**

Each characteristic is further broken into measurable sub-characteristics (e.g., *reliability* → *availability, recoverability, fault tolerance*).

👉 **Why important?**
You can *structure, assess, and communicate* quality requirements systematically.

Making Quality Requirements Concrete

Scenarios are used to make quality goals tangible and testable.
Each scenario describes:

- A **stimulus** (e.g., "a user opens 10 tabs simultaneously")
- A **response** (e.g., "within 1 second, no crash")

This supports **measurable** architecture evaluation and design trade-offs.

ATAM-lite (Architecture Tradeoff Analysis Method - Lite)

A **lightweight technique** to analyze architecture decisions based on quality goals.

- **Inputs**: Stakeholders, business drivers, quality attributes
- **Outputs**: Risks, trade-offs, sensitivity points, justifications

Basic process:

1. Identify business goals
2. Elicit quality attributes (with scenarios)
3. Analyze architectural approaches in light of those
4. Discuss tradeoffs and alternatives
5. Prioritize architectural risks

👉 **Use ATAM-lite** early in the design to ensure architectural decisions align with quality goals.

Architectural Patterns and Styles. Layered, microservices, blackboard, etc.

Architectural patterns define **recurring structural solutions** for commonly encountered design problems. Each has **strengths, trade-offs, and suitable contexts**.

Layered Architecture

Description:
System is organized into horizontal layers, each with specific responsibilities (e.g., UI, business logic, data access).

Typical layers:

- **Presentation Layer**
- **Application Layer**
- **Domain Layer**
- **Infrastructure/Data Access Layer**

Pros:

- Separation of concerns
- Easy to understand and maintain
- Promotes modifiability

Cons:

- Can be inefficient due to layer-by-layer communication
- May lead to layer leakage if not enforced

Use When:
Business logic is cleanly separable from UI/data and modifiability is important.

Client-Server

Description:
Separates concerns between clients (requesters of services) and servers (providers).

Examples:

- Web apps (browser ↔ server)

- Mobile apps with backend

Pros:

- Clear role separation
- Scales horizontally on the server side

Cons:

- Requires robust communication handling
- Single point of failure unless mitigated

Microservices

Description:
System is composed of **small, independent services**, each handling a specific business capability and communicating via APIs.

Pros:

- Independent deployment and scaling
- Supports domain-driven design
- Technology diversity possible

Cons:

- High complexity (network, data consistency)
- Requires DevOps and automation maturity

Use When:
Scalability, team autonomy, or deployment independence is required.

Pipes and Filters

Description:
Data flows through a series of processing elements (filters), connected by pipes.

Pros:

- Good for streaming or data-processing tasks
- High reusability of filters

Cons:

- Hard to manage shared state
- Can be inefficient with large intermediate data

Use When:
Transformations on data need to be composed (e.g., compilers, data ingestion pipelines).

Blackboard Architecture

Description:
Multiple independent components (knowledge sources) contribute solutions to a central **blackboard** (shared state), monitored by a **control component**.

Pros:

- Excellent for exploratory or AI tasks (e.g., speech, image recognition)
- Promotes loose coupling

Cons:

- Complex coordination
- Hard to test/debug

Use When:
You need a collaborative problem-solving approach under uncertainty.

Event-Driven Architecture (EDA)

Description:
Components communicate by emitting and reacting to **events** asynchronously.

Variants:

- **Publish/Subscribe**
- **Event Streaming**
- **Event Sourcing**

Pros:

- Loose coupling
- High responsiveness and scalability

Cons:

- Harder to trace flow
- Event ordering/consistency can be complex

Use When:
You need responsive, scalable systems (e.g., IoT, UIs, reactive systems).

Domain-Driven Design (DDD)

- **Strategic Patterns**: Bounded Contexts, Context Maps
- **Tactical Patterns**: Entities, Value Objects, Aggregates, Repositories

DDD encourages building architectures around the **business domain**, not technical infrastructure.

Use When:
You have a complex business domain that requires close alignment between architecture and domain experts.

Summary Table

Pattern/Style	Key Benefit	Best For
Layered	Modifiability	Classic business apps
Microservices	Scalability, autonomy	Large teams, independent deployments
Pipes & Filters	Composability	Data transformation pipelines
Blackboard	Collaborative reasoning	AI, speech/image recognition
Event-Driven	Reactivity & scalability	IoT, GUIs, distributed async systems
DDD	Domain alignment	Business-critical logic-heavy systems

Design Principles. SoC, SRP, DIP, DRY

Separation of Concerns (SoC)

Definition:
Divide the system into distinct sections, each addressing a **separate concern** (e.g., UI, business logic, persistence).

Goal:

- Reduce complexity
- Improve modularity
- Enhance readability and maintainability

📌 *Example*: MVC (Model-View-Controller) separates data (Model), UI (View), and logic (Controller).

Single Responsibility Principle (SRP)

Definition:
Each building block (class/module/component) should have **only one reason to change**.

Goal:

- Encourage clarity and simplicity
- Minimize side effects
- Align with modular architecture

📌 *Example*: A UserManager class that handles both persistence and email notifications violates SRP — those responsibilities should be split.

Dependency Inversion Principle (DIP)

Definition:
High-level modules should not depend on low-level modules. Both should depend on **abstractions**.

Goal:

- Promote flexibility and testability
- Reduce coupling between layers

- Enable plug-in behavior

📌 *Example*:

// BAD: High-level directly uses low-level class

OrderService uses MySQLDatabase;

// GOOD: Both depend on an interface

OrderService depends on DatabaseInterface

MySQLDatabase implements DatabaseInterface

Result: Implementation details can change without modifying high-level logic.

Don't Repeat Yourself (DRY)

Definition:
Every piece of knowledge or logic should exist **in a single place**.

Goal:

- Avoid redundancy
- Prevent inconsistencies
- Reduce maintenance effort

📌 *Example*: If tax calculation logic is duplicated in two services, changes must be synchronized — a clear violation of DRY.

Relationship to Architecture

These principles are not just coding rules — they affect **architecture-level** decisions:

Principle	Supports Quality Attributes	Applies To
SoC	Modifiability, Reusability	Component & system level
SRP	Maintainability, Simplicity	Module/class level
DIP	Testability, Modifiability, Reuse	Interfaces & layering
DRY	Maintainability, Consistency	Code, configuration, models

Architectural Views and Notations. 4+1, arc42, UML, C4

Software systems are complex. **Architectural views** are different perspectives that help manage and communicate this complexity. **Notations** (like UML or C4) are used to express these views clearly.

- **View-Based Architecture**

You do not document "the architecture" — you document **multiple views** of it, each serving different stakeholders.

4+1 View Model (by Philippe Kruchten)

A widely accepted model using **five interrelated views**:

View	Purpose	Audience
Logical	Describes functional decomposition (e.g., modules, services)	Developers, designers
Development (Implementation)	Focuses on software organization (e.g., components, packages)	Developers
Process	Describes runtime behavior and concurrency	Architects, system integrators
Physical	Maps software to hardware (deployment)	DevOps, infrastructure teams
Scenarios (+1)	Validates views using use cases or quality scenarios	Stakeholders, testers

arc42 Template

A practical and widely used template for **documenting software architecture** (especially in iSAQB).

Core sections include:

- **Context and Scope**
- **Solution Strategy**
- **Building Block View**

- **Runtime View**
- **Deployment View**
- **Crosscutting Concepts**
- **Design Decisions**
- **Quality Requirements**
- **Risks and Technical Debt**

📌 **Goal**: Structure the documentation for clarity, traceability, and completeness.

🔗 Strongly recommended by iSAQB for consistent documentation.

UML (Unified Modeling Language)

UML is a standard modeling language for visualizing software systems. Key diagrams useful for architecture:

Diagram	Purpose
Component Diagram	Structure of modules and interfaces
Class Diagram	Domain models, object relationships
Sequence Diagram	Message exchange between components
Deployment Diagram	Mapping software to infrastructure
Activity Diagram	Workflows, runtime behavior

📌 Used heavily in 4+1 and arc42 views.

C4 Model (by Simon Brown)

A **modern and simplified approach** for visualizing software architecture at multiple levels.

Level	View	Description
C1	**Context Diagram**	System and external actors
C2	**Container Diagram**	Applications/services in the system
C3	**Component Diagram**	Breakdown of a container

Level	View	Description
C4	**Code (optional)**	Class-level design (if needed)

📌 Great for communicating with **technical and non-technical stakeholders** alike.

How These Relate

Model / Notation	Strengths	Typical Use
4+1	Covers stakeholders via multiple views	Formal architectural documentation
arc42	Template for structured architecture docs	iSAQB, regulated domains
UML	Standardized, tool-supported	Formal system modeling
C4	Simple, modern, hierarchical	Lightweight communication, Agile teams

Summary Table

View Type	Typical Notation	Example Diagram
Logical View	UML Class, Component	Layered component model
Development View	UML Package	Source code breakdown
Runtime View	UML Sequence, Activity	Login flow, user request flow
Deployment View	UML Deployment	Cloud vs. on-prem server map
Context View (C4)	Custom or UML	System ↔ external actors

Crosscutting Concepts. Logging, error handling, security, etc.

What Are Crosscutting Concepts?

Crosscutting concepts (also called *crosscutting concerns*) are **aspects of a system that affect multiple components** or layers — they "cut across" functional boundaries.

They typically deal with **non-functional requirements (quality attributes)** and **influence consistency, maintainability, and reliability**.

Examples of Crosscutting Concepts

Concept	Description
Logging	Standardized collection of runtime events and errors
Error Handling	How the system detects, reports, and recovers from failures
Security	Authentication, authorization, data protection, and compliance
Configuration	Externalization and management of system behavior without code changes
Monitoring	Collecting metrics and health indicators for observability
Transaction Management	Ensuring data consistency across distributed systems or components
Caching	Reuse of expensive results for performance optimization
Internationalization (i18n)	Making the system adaptable to various languages and regions
Validation	Enforcing business or technical constraints on inputs and states

Why Are They Important?

Crosscutting concepts:

- Ensure **system-wide consistency**
- Influence **multiple views**: logical, runtime, deployment
- Strongly affect **quality attributes** such as:
    - **Security**
    - **Maintainability**
    - **Performance**
    - **Scalability**
    - **Robustness**

How to Handle Crosscutting Concepts

Technique	Explanation
Define them explicitly	Document each concept just like components (often in a separate section, e.g., in arc42)
Standardize early	Align with stakeholders and development teams during architectural design
Apply consistently	Use common libraries, patterns (e.g., AOP, middleware), and frameworks
Document rationale and impact	Explain trade-offs and affected quality goals

📘 **In arc42:**

Crosscutting concepts are documented in **Section 8 – Crosscutting Concepts**.

🧠 **iSAQB Tip:**

You may be asked:

- To **identify** crosscutting concerns
- To explain **how they influence quality attributes**
- To describe how to **document and standardize** them

Building Block View. Component modeling, black/white box

The Building Block View describes the **static structure** of your system by breaking it down into **hierarchically organized components (building blocks)**.

🔹 **Purpose**

- Clarifies **responsibilities and structure**
- Helps developers understand **how the system is composed**
- Basis for **interface contracts**, **responsibility distribution**, and **modular design**

Two Key Concepts

White-Box Description

Describes the **internal structure** of a building block:

- Subcomponents (child blocks)
- Their relationships
- Internal data flow and control flow

📌 Example: A white-box view of a PaymentService might show it includes a CardProcessor, BankGateway, and a Logger.

Black-Box Description

Describes **external view** of a building block:

- **Name** and **responsibility**
- **Interface** (inputs/outputs)
- **Dependencies** (what it uses or needs)
- **Constraints** (e.g., performance, reliability)

📌 Example: A black-box view of a LoginService describes:

- What it does: Authenticates users
- What it exposes: login(), logout(), isSessionActive()
- What it requires: User repository, encryption module

Hierarchical Decomposition

- Architecture is documented **top-down** in multiple levels of detail:
    1. Level 0: Entire system
    2. Level 1: Major subsystems or components
    3. Level 2: Subcomponents of each level 1 block, etc.

📌 Each **white-box at level N** becomes a **black-box at level N+1**.

Modeling Techniques

UML Component Diagrams – for static structure

UML Package Diagrams – to show logical groupings

C4 Component View (C3) – for a lightweight alternative

Attributes of a Good Building Block

Attribute	**Purpose**
Cohesive	Has a well-defined, focused responsibility
Loosely Coupled	Minimizes dependencies to others
Replaceable	Can be re-implemented independently
Testable	Supports isolated testing

🧠 **iSAQB Tip:**

- Be able to model and explain **black-box and white-box** structures.
- Know how the building block view **relates to runtime, deployment, and crosscutting concerns**.
- Understand how decomposition **supports maintainability, scalability, and team collaboration**.

Runtime and Deployment Views. Runtime interaction and deployment strategies

These two views describe:

- **How components collaborate at runtime** (Runtime View)
- **How software is mapped to infrastructure** (Deployment View)

Runtime View

✅ **Purpose:**

- Shows **how building blocks interact dynamically**
- Supports **understanding of control flow, error paths, performance bottlenecks**
- Critical for:
    - Use case realization
    - Scenario evaluation
    - Explaining behavior to stakeholders

🛠 **Tools & Notations:**

Technique	Used for
UML Sequence Diagrams	Ordered interaction between components
UML Activity Diagrams	Workflow and parallel behavior
State Diagrams	Finite-state behavior modeling
Message Flows	Event-based or asynchronous systems

📌 Example: A login scenario may show:

- User → LoginService → UserDB → TokenService
- Error handling paths
- Timeout retries

Deployment View

✅ **Purpose:**

- Describes **where software components run**
- Captures **physical infrastructure**:
  - Servers
  - Network topology
  - Cloud vs on-premise
- Shows **distribution strategies**, **replication**, **failover**

🛠 **Tools & Notations:**

Diagram Type	Description
UML Deployment Diagram	Nodes, artifacts, network links
Infrastructure Diagrams	AWS/Azure architecture visuals, containers
Container Diagrams (C4-C2)	Where services/apps run

📌 Example:

- AuthService container runs in Kubernetes pod on AWS EC2
- Connected to PostgreSQL on another host
- Deployed behind a load balancer

Relation Between Views

View Type	Key Concern	Question Answered
Runtime	How components collaborate	"What happens when user X performs action Y?"
Deployment	Where and how components are hosted	"Which node hosts this service?"

🧠 **iSAQB Tip:**

You should be able to:

- Identify when each view is useful
- Model component interactions clearly (sequence diagrams, activity diagrams)
- Map software components to hardware nodes (especially under constraints like latency, security, and scalability)
- Understand trade-offs between **centralized vs distributed**, **synchronous vs asynchronous** deployment strategies

Architecture Documentation. Practices and tooling for documentation

Purpose

Good documentation enables:

- **Understanding** of the system's structure and decisions
- **Communication** across teams and stakeholders
- **Onboarding**, maintenance, and handover
- **Validation and review** of architecture decisions

📣 *"If it's not documented, it doesn't exist."*

What Should Be Documented?

Aspect	**Description**
Building Blocks	Their interfaces, responsibilities, structure
Architectural Decisions	Rationale, trade-offs, constraints
Quality Requirements	And how they're achieved (crosscutting concepts, patterns)
Views	Logical, runtime, deployment, crosscutting
Terminology	Glossary, ubiquitous language, domain concepts

Templates and Frameworks

arc42 Template (highly recommended)

Covers:

- System scope & context
- Solution strategy
- Building block view
- Runtime view
- Deployment view
- Crosscutting concepts
- Architectural decisions
- Risks and technical debt

🔗 https://arc42.org

4+1 View Model

- Logical View
- Process View
- Development View
- Physical View
- Use-Case View

Best Practices

Principle	**Application**
Consistency	Use the same structure, naming, and format across documents
Traceability	Link decisions to requirements and components
Versioning	Maintain historical context for decisions
Collaborative Authoring	Involve developers, testers, DevOps — not just architects

Principle	Application
Living Documentation	Keep it updated as the architecture evolves
Tailoring	Only document what is necessary for your stakeholders

Tooling Support

Tool	Purpose
PlantUML, Structurizr	Diagram generation and automation
Confluence, Docusaurus	Living documentation portals
Markdown with Git	Lightweight, version-controlled docs
ADR tools	Manage architecture decision records

📌 Many teams now embed architecture docs **close to code**, often using .adr files and Markdown.

Common Pitfalls

Outdated documents

Overly abstract or too detailed content

Not addressing the *why* behind architectural decisions

Failing to consider the audience (business vs devs)

💡 **iSAQB Tip:**

Expect questions on:

- Why documentation is important
- What should go into a black-box or white-box description
- How documentation supports **reviews**, **evaluation**, **testing**, and **change**

Architecture Evaluation and Validation. Qualitative and quantitative evaluations

Purpose

To assess whether the architecture:

- Meets **functional and quality requirements**
- Is **feasible**, **maintainable**, and **scalable**
- Is aligned with **business goals** and **technical constraints**

Evaluation helps reduce **technical debt**, avoid costly redesign, and support **risk management**.

Two Types of Evaluation

Type	Focus	Examples
Qualitative	Architectural soundness, design decisions	Reviews, scenarios, ATAM
Quantitative	Measurable attributes	Metrics, benchmarks, load tests

Qualitative Evaluation

🔹 **Techniques:**

- **Scenario-based reviews** (e.g. ATAM-lite)
- **Architecture reviews** (internal/external stakeholders)
- **Checklists** (consistency, quality attributes)
- **Risk/stress workshops**: What could go wrong?

🛠 **Key Concepts:**

- **Quality Attribute Scenarios**: Structured way to evaluate maintainability, performance, security, etc.
- **ATAM (Architecture Tradeoff Analysis Method)**:
    - Identify quality goals
    - Describe architecture

- Analyze trade-offs
- Document risks/sensitivity points

Quantitative Evaluation

- **What can be measured?**

Metric	Quality Attribute
Cyclomatic complexity	Maintainability
Response time, latency	Performance
Code churn, commit frequency	Stability, maintainability
Coupling/cohesion metrics	Modularity
Log frequency / error rates	Robustness, fault tolerance

- **Techniques:**
  - Static code analysis (e.g., SonarQube)
  - Profiling and load testing
  - Monitoring logs and KPIs in production
  - Automated quality gates in CI/CD

Validation Outcomes

Outcome	Description
Risk Identification	Points where quality might degrade
Trade-off Visibility	Awareness of what's been sacrificed or prioritized
Decision Justification	Whether architectural decisions were valid
Improvement Actions	Refactoring, additional documentation, design shifts

🧠 **iSAQB Tip**

You may be asked to:

- Distinguish qualitative vs quantitative evaluations
- Understand and apply **ATAM-lite** or **quality scenarios**
- Interpret common architectural metrics
- Recognize when lightweight evaluations are sufficient

Tools and Techniques. UML, SysML, ADRs, versioning

This topic emphasizes **how architects can practically support, communicate, and maintain architecture work** using standard tools and lightweight practices.

Modeling Tools

These tools help express architectural structure, views, and behavior clearly and consistently.

Tool / Notation	Purpose
UML (Unified Modeling Language)	Industry-standard visual language for modeling structure (e.g., component diagrams), behavior (e.g., sequence diagrams), and deployment
SysML (Systems Modeling Language)	Extension of UML for system engineering; used where hardware + software interact (e.g., embedded systems)
C4 Model	Lightweight alternative focusing on hierarchy: Context → Container → Component → Code
PlantUML / Structurizr	Text-based diagram tools to generate UML/C4 visuals versioned in Git
ArchiMate	High-level enterprise modeling (often used with TOGAF)

Architecture Decision Records (ADRs)

✅ **What they are:**

- **Concise text documents** capturing architectural decisions
- Typically written in **Markdown** and versioned alongside code
- Follow the format: **"In the context of X, we decided Y, because Z."**

📘 **Benefits:**

- Traceability of **why** things were designed a certain way
- Communication of **trade-offs**, alternatives, and rationale
- Supports **onboarding**, **governance**, and **audits**

Versioning and Traceability

Practice	Purpose
Version control of architecture artifacts (e.g., Git)	Ensures historical tracking of diagrams, decisions, models
Traceability matrices	Link requirements → decisions → components
Automated change tracking	Prevents undocumented architectural drift

📌 *Architecture should evolve like code: iteratively and transparently.*

Toolchain Examples

Function	Tools
Modeling	Visual Paradigm, draw.io, Archi, Papyrus
Diagram-as-code	PlantUML, Mermaid, Structurizr DSL
ADRs	Markdown files, adr-tools, log4brains
Documentation	Confluence, Docusaurus, MkDocs
Versioning	Git, GitHub/GitLab, Git-based wikis

🧠 **iSAQB Tip**

You should understand:

- When to use **heavyweight** vs **lightweight** tools

- How to **integrate architectural artifacts** into the dev workflow
- That tools **support**, not replace, architecture thinking
- That **traceability** and **collaboration** are more important than fancy diagrams