**What is Software architecture?**

Software architecture is the high-level structure of a software system, the way its components are organized, how they interact, and the guiding principles that shape its design and evolution.

It defines how the system works as a whole, not just what it does.

It’s about making the big decisions early:

* + How components communicate
  + How data flows
  + How to ensure scalability, maintainability, and performance

In simple words:

Software architecture is the blueprint that turns code into a coherent, reliable system.

The first thing architecture needs to understand is system requirement.

system requirement consists of 3 parts:

* Functional Requirement / Features of the system

Functional requirements describe **what the system should do** — the specific behaviors, actions, or features that fulfill user needs.

You can think of them like this, “When a user takes an action, how should the system respond?” or “When the system performs an action, how will it impact users or other components?”

The Functional requirements it’s what define our System.

EX: The user can order food for delivery/pick-up therefore our system is food ordering system.

* Non-Functional Requirement / Quality attributes

If *functional requirements* define **what** the system should do,  
then ***non-functional requirements* describe how the system should do it**.

They define the **qualities and behaviors** that make the system reliable, efficient, and pleasant to use.

**It’s not about specific features — it’s about *how well* those features work together.**

You can think of it like this:

“How should the system react under pressure?”  
“How should components communicate, recover, or scale when things grow or fail?”

Common examples include:

* **Performance** – how fast the system responds
* **Scalability** – how it handles growth
* **Security** – how data and users are protected
* **Availability** – how often it stays up and running
* **Maintainability** – how easily it can be updated or improved
* System Constraints / Limitations and Boundaries

Every system operates within a set of constraints — rules or conditions that define **what it can and cannot do**.

These **constraints help shape realistic expectations and guide architectural decisions**.

You can think of them in two parts:

Limitations: In a food ordering system, deliveries are limited to **within Saudi Arabia (KSA)** only.

Boundaries: **Users** can interact only through the *mobile app*, while **administrators** access the system exclusively via the *web portal*.

Now we understand the system requirements concepts, but how do we identify them? Let’s start with how we can identify functional requirements.

* Identify Functional Requirements (Requirement Gathering):

1. Identify all the actors/users in our system

listing everyone who will interact with the system — direct users, admins, external systems, or third-party services.

*Example: Customer, Delivery Driver, Restaurant Admin, Payment Gateway.*

1. Capture and describe all the possible use-cases/ scenario

Define what each actor needs to achieve. Each use case should describe a specific goal or interaction.

Example: “Place Order,” “Track Delivery,” “Manage Menu,” “Process Payment.*”*

1. User Flow

Visualize how each use case is triggered and what sequence of events it produces — from the first user action to the final system response.

Example: A customer adds items to the cart → places an order → payment is processed → restaurant receives the order → driver gets notified.

* Non-Functional Requirement (Quality Attributes):

Non-functional requirements can have multiple **dimensions**, such as:

* + **Performance**
  + **Scalability**
  + **Security**
  + **Availability**
  + **Maintainability**

A key principle is that non-functional requirements **must be** **measurable and testable**

For example,

* + - Performance Dimension: When the user submits a food delivery order the **system should response at most within 100 Milliseconds**.
    - Availability Dimension: The application should be **available** to users at least **99.9% of time**.

As we can see, each quality attribute has a **quantifiable target** — whether it’s response time, uptime, or resource usage.

Key Points to Remember:

* + - **No single system can achieve all quality attributes perfectly.**
    - Some attributes may **conflict with others** (e.g., security vs. performance).
    - Certain combinations **can** **be hard or impossible to achieve simultaneously**.
    - Architecture often involves **making trade-offs — choosing which qualities matter most for your system’s goals**.
    - Feasibility, Architecture needs to make sure the required quality attributes are possible, **no system can have zero latency or 100% up time**

* System Constraints
  + Technical Constraints: The System can’t be implemented using the existing technology or it is too expensive.
  + Business Constraints: Limitations driven by business goals, policies, or operational rules.
  + Legal Constraints: some systems or functionalities are prohibited globally or in specific countries.

Now after covering the main parts of functional requirements and non-functional requirements, we can start by answering the following question:

How do system clients or components interact?

First let’s break this question apart,

Clients: Clients can be users or external systems that interact with our application.

Components: Component is a building block of the overall system; each building block is responsible for a specific part of the system.

Depending on the architecture, they can take different forms:

* In a monolith, a component might be a module or package (for example, “User Management,” “Payments,” or “Notifications”).
* In a microservices architecture, a component is usually a service (for example, “Order Service,” “Inventory Service”).
* In a front-end system, a component might be a UI widget or React component.

Once we understand the two parts (clients and components) the answer becomes clear:

**Clients and components interact through APIs (Application Programming Interfaces).**

APIs define how data is exchanged, how requests are made, and what responses are returned.

Now that we understand what an API is and why it’s important, the next step is to learn how to design a good one.

At the architectural level, a good API is not just about URLs or payloads — it’s about clear contracts, predictable behavior, and reliable communication between different parts of a system.

Whether it’s REST, GraphQL, gRPC, or even asynchronous messaging (RabbitMQ, Kafka, etc.), the same design principles apply.

Let’s start with the three **main categories of APIs**:

1. Public API: Exposed to targeted users (even if it requires authentication)
2. Private/Internal API: Exposed internally within an organization, usually administrators, account managers etc...
3. Partner API: Exposed only to users/companies that have a business relationship with system owners.

Now let’s deep dive into API best practices (regardless of protocol or style):

1. Complete encapsulation of internal implementation.

APIs should hide the internal complexity of the system.  
API consumers should only know **what** the API does not **how** it does it.

1. Easy to use.

Use the same terms, formats, and patterns across all endpoints or message types whether it’s URLs, topics, or payload fields.

1. API Pagination.

When an API returns large amounts of data, it’s best to **split results into pages**.

1. Asynchronous Operations (If needed)

Some operations take time, like file uploads, report generation, or large data processing.

Instead of blocking the client, provide asynchronous APIs that allow the client to check progress later.

1. API Versioning

API versioning gives you the ability to update and improve existing APIs while still supporting backward compatibility for older consumers.

1. API Documentation

Always document your APIs using helper tools like Swagger (OpenAPI), Postman Collections, or AsyncAPI (for event-driven systems)