Software Architecture: Lecture 1 1

Requirements Elicitation & Modeling

Laying the Architectural Foundation for ShopSphere (An E-commerce Platform)

What is Software Architecture?

Software architecture is more than just coding; it's about making fundamental design decisions that shape a system's future. These decisions, once implemented, become difficult and costly to change, making them critical for long-term success.

Defining Structure and Components Components

It outlines the system's structural elements, their interfaces, and how they interact. This includes defining modules, services, databases, and communication protocols.

Guiding Principles and Evolution

Architecture establishes the overarching principles and guidelines that govern the system's design and ongoing evolution. It's the framework that ensures consistency and manageability.

Satisfying Quality Attributes

The primary goal for an architect is to satisfy the system's critical Quality Attributes (Non-Functional Requirements) while simultaneously meeting all functional needs. These attributes include performance, scalability, security, and maintainability.

> “Architecture is the blueprint of a building, not the interior decorating. It provides the essential structure that dictates what is possible within the system.”

The Starting Point: The ShopSphere Scenario

Our journey into requirements elicitation begins with a practical case study: ShopSphere, a hypothetical yet realistic modern e-commerce platform.

Case Study: ShopSphere

ShopSphere is envisioned as a high-traffic, secure, online retail platform designed to deliver an exceptional shopping experience to millions of users globally. It aims to compete with leading e-commerce giants by focusing on reliability and user satisfaction.

- **Business Goal:** Provide a seamless, intuitive, and highly reliable shopping experience. This includes easy product discovery, smooth checkout flows, and robust order management.
- **High Transaction Volumes:** The platform must be engineered to handle a massive number of concurrent transactions, especially during promotional events and peak seasons.
- **Reliable Payment Processing:** Integration with secure and efficient payment gateways is paramount to build customer trust and ensure financial integrity.

Key Challenge: Peak Load Management

ShopSphere must be **highly available and scalable** to gracefully manage extreme seasonal peak loads, such as Black Friday or Cyber Monday, without any degradation in performance or user experience. This resilience is a non-negotiable architectural driver.

Knowledge Base: The Four Types of Requirements

Understanding the different categories of requirements is fundamental to successful software architecture. Each type provides a unique lens through which we define and build a system.

1

Functional Requirements (FRs)

These define **what the system must do**—its features and capabilities. They describe the interactions between the system and its users, and are often captured in use cases or user stories.

ShopSphere Example: "The system must allow users to process a payment." or "The system must enable users to search the product catalog by keyword."

2

Non-Functional Requirements (NFRs)

These specify **how well the system performs a function**, focusing on quality attributes. They are critical for user satisfaction and operational success, defining aspects like performance, security, usability, and reliability.

ShopSphere Example: "The search results page must load in under 2 seconds for 95% of users."

3

Domain Requirements

These are constraints imposed by the operating environment or specific business rules. They often reflect industry standards, legal regulations, or inherent characteristics of the business domain.

ShopSphere Example: "All payment transactions must adhere to PCI DSS (Payment Card Industry Data Security Standard) compliance protocols."

4

Architecturally Significant Requirements (ASRs)

These are NFRs or FRs that critically impact the choice of a system's architecture. They are the requirements that, if not met, would render the system unusable or unacceptable, driving major architectural decisions.

ShopSphere Example: "The platform must achieve 99.9% uptime for core services." (High Availability).

Activity 1: Eliciting Functional Requirements (FRs)

Eliciting Functional Requirements involves defining the specific actions and behaviors the system must exhibit. For ShopSphere, we consider two primary user roles: Customers and Administrators.

Customer Functional Requirements

- **F1: User Authentication:** Customers must be able to securely register for a new account and log in using their credentials. This includes features like password recovery and multi-factor authentication options.
- **F2: Shopping Cart Management:** Customers must be able to add products to a persistent shopping cart, modify quantities, and remove items from it. The cart state should be maintained across sessions.
- **F3: Multi-Step Checkout Process:** Customers must be able to navigate a clear and intuitive multi-step checkout process, including shipping address entry, shipping method selection, and secure payment processing.

Administrator Functional Requirements

- **F4: Product Inventory Management:** Administrators must have full CRUD (Create, Read, Update, Delete) capabilities for managing the product catalog, including adding new products, updating product details, and managing stock levels.
- **F5: Order Status Management:** Administrators must be able to view and update the status of customer orders (e.g., from "Pending" to "Processing," "Shipped," or "Delivered"). This includes tracking and notification features.

These FRs lay the groundwork for understanding the core interactions within the ShopSphere platform.

Activity 2: Eliciting Non-Functional Requirements (NFRs)

While FRs define what the system does, NFRs define how well it does it. Eliciting NFRs is crucial for ensuring the system meets user expectations and business goals beyond basic functionality.

Performance

The system's responsiveness and efficiency. For ShopSphere, search results must load in < 1.5 seconds to ensure a fluid user experience and prevent customer frustration. This is a critical metric for user engagement.

Scalability

The system's ability to handle increasing workloads. ShopSphere must support 10,000 concurrent users during peak load events, such as flash sales, without any degradation in performance or availability.

Security

Protection against unauthorized access and data breaches. All financial data, including credit card information, must be encrypted end-to-end to ensure PCI DSS compliance and protect customer privacy.

Availability

The system's operational time. The payment system, being mission-critical, must guarantee 99.99% uptime to minimize revenue loss and maintain customer trust.

These NFRs will significantly influence our architectural decisions for ShopSphere.

Knowledge Base: Architecturally Significant Requirements (ASRs)

ASRs are the linchpin of architectural design. They are the requirements that hold the power to shape the entire system, acting as a crucial filter for architectural choices.

The ASR Filter: What Makes a Requirement "Significant"?

ASRs are those functional or non-functional requirements that, if not met, would lead to the unequivocal failure of the system from a business or user perspective. They are so fundamental that they directly force a choice between major architectural patterns, such as a **Monolithic** versus a **Microservices** architecture, or impact the selection of core technologies.

- **System Failure Criterion:** An ASR is a requirement whose absence or inadequate fulfillment means the system cannot serve its intended purpose or deliver value.
- **Direct Architectural Impact:** They drive fundamental decisions about structure, technology stack, deployment model, and even organizational structure. Ignoring them leads to costly rework.

Why ASRs Matter: Primary Drivers for Design

By identifying ASRs early, architects can prioritize design efforts, mitigate risks, and select appropriate architectural styles that inherently support these critical qualities. They become the primary drivers that validate and guide the architectural design process.

Focusing on ASRs ensures that the architecture is robustly aligned with the most vital business and technical imperatives.

Activity 3: Defining the ShopSphere ASRs 🔑

Based on our understanding of ShopSphere and the nature of ASRs, we now select three core requirements that will fundamentally dictate its architecture.

ASR 1: Scalability

Requirement: The system must handle 10,000 concurrent user sessions with peak latency remaining under 2 seconds. This is critical for maintaining performance during high-traffic events.

Architectural Impact: This ASR forces a Distributed Architecture. It necessitates advanced load balancing, potentially microservices for granular scaling, and careful consideration of data sharding or replication strategies.

ASR 2: Fault Isolation

Requirement: A failure in a non-critical component, such as the Notification System, must not prevent users from completing their order transactions.

Architectural Impact: This demands Decoupling and Asynchronous Communication. It requires mechanisms like message queues, event-driven architectures, and bulkhead patterns to ensure core transactional paths remain operational even if auxiliary services fail.

ASR 3: Data Security

Requirement: All customer passwords and payment details must comply with current industry standards (e.g., OAuth 2.0 for authentication, PCI DSS for payment data).

Architectural Impact: This mandates a Security Gateway Pattern. It necessitates a single, hardened entry point for authentication and authorization checks before requests are routed to internal services, often involving specialized security services.

These ASRs will serve as guiding stars for our subsequent architectural design choices.

Knowledge Base: The Four Views of Architecture

To effectively document and communicate a complex system's architecture, we use different "views," each catering to specific stakeholders and concerns. The 4+1 View Model by Philippe Kruchten is a widely recognized framework.

Logical View

Describes the system's functional structure, focusing on component abstractions (e.g., classes, modules, packages) and their relationships. It's for end-users and analysts, showing functional requirements.

+1 Use Case View

This view ties all other views together, illustrating how the system's functionalities (use cases) are realized by the components described in the other views. It's user-centric, defining external behavior.

Process View

Addresses concurrency, distribution, and system performance. It describes how processes communicate and synchronize, focusing on runtime behavior. It's for integrators and performance engineers.

Development View

Focuses on the software's static organization in the development environment. It includes modules, subsystems, and layers, aiding developers in understanding the codebase structure.

Physical (Deployment) View

Maps the software components onto physical hardware nodes, showing the system's distributed aspects and infrastructure. It's for system engineers and deployers.

Today's focus will primarily be on the **Use Case View** to model external interactions.

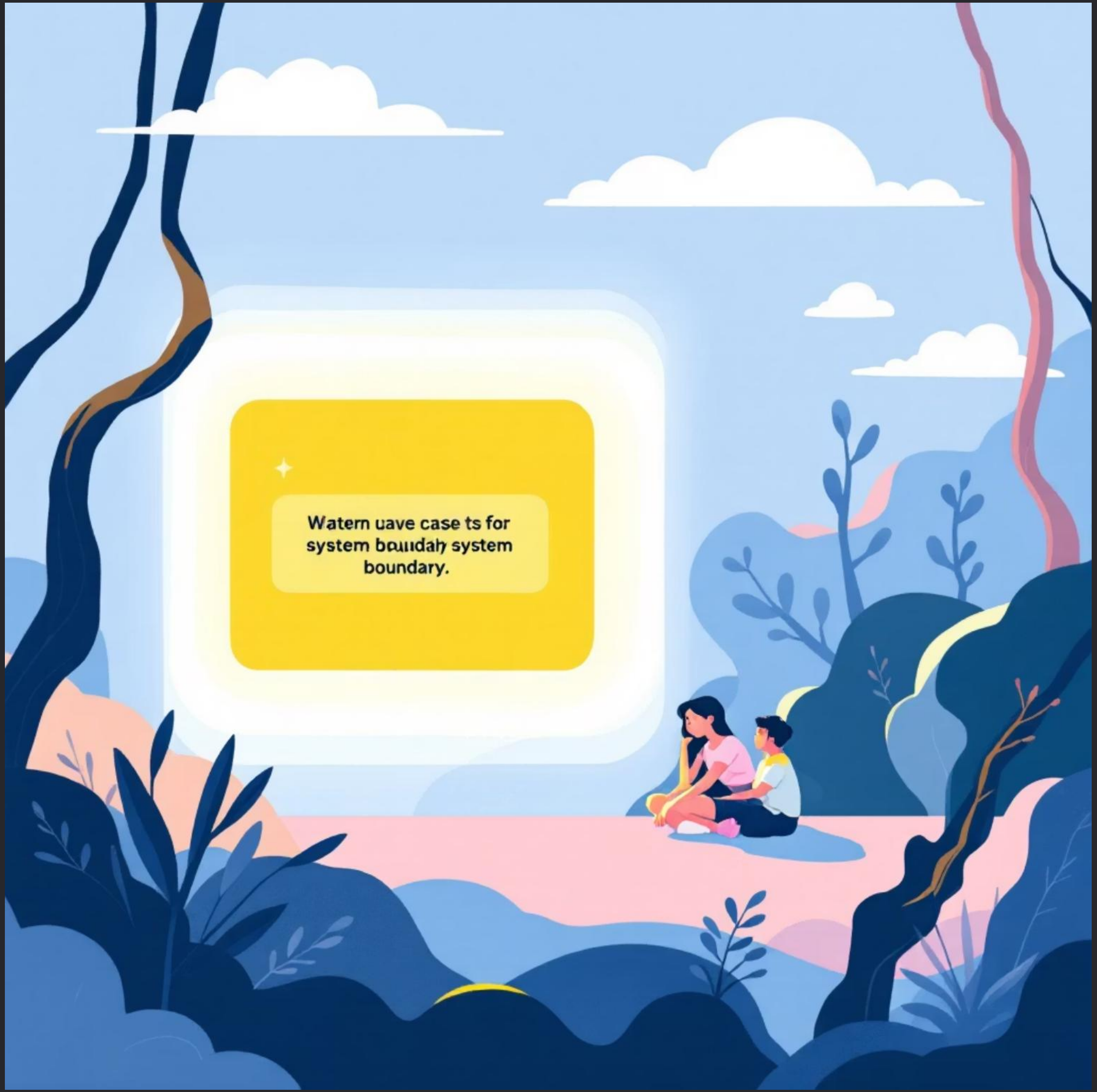
Knowledge Base: The Use Case View (UML)

The Use Case View is the simplest and most accessible architectural view, providing a high-level understanding of what the system does from an external perspective. It's part of the Unified Modeling Language (UML).

Purpose of the Use Case View

It visually represents the system's boundary and how external entities, known as **Actors**, interact with the system's core **functionalities (Use Cases)**. It defines the scope of the system and its primary interactions.

- **Actor (Stick Figure):** Represents a role played by a user or an external system that interacts with the system being modeled. They initiate or participate in use cases.
- **Use Case (Oval):** Describes a sequence of actions performed by the system, often in response to an Actor's request, to yield an observable result of value to that Actor. It focuses on **what** the system does, not **how**.
- **System Boundary (Box):** A rectangle that encloses all the use cases, explicitly defining the scope of the software system under consideration. Anything outside this box is external.



Modeling Relationships: Include and Extend

- **Include <<include>> (Required):** Denotes a mandatory sub-functionality. If Use Case A includes Use Case B, then B must be executed every time A is executed. For example, "Make Purchase" **includes** "Secure Payment."
- **Extend <<extend>> (Optional/Alternative):** Represents an optional step or an alternative flow that may be executed under specific conditions. For example, "Make Purchase" **extends** "Apply Coupon."

This foundational understanding allows us to visually model ShopSphere's interactions.