Software Architecture: Lecture 8

Deployment View & Quality Attribute Analysis (ATAM)

Welcome to Lecture 8 of Software Architecture. Today, we're moving beyond conceptual models to finalize the architectural blueprint of our systems. This session focuses on two critical aspects: documenting the physical structure of our software through the Deployment View and rigorously evaluating its effectiveness using the Quality Attribute Analysis Method (ATAM).

Our primary objective is to transition from the logical arrangement of components, which we've discussed in previous lectures, to a concrete understanding of how these components will be physically deployed across various execution environments. Concurrently, we will critically analyze how well this finalized architecture meets the Architectural Significant Requirements (ASRs) that were established back in Lecture 1, ensuring our design choices are robust and justifiable.

Recap: The Final Architecture

In our journey through software architecture, we've converged on a powerful and flexible pattern: the Microservices Architecture, significantly enhanced by an Event-Driven Architecture (EDA). This pattern enables us to build scalable, resilient, and independently deployable services.

Key Components of Our Architecture:

- **API Gateway (Port 5000):** The entry point for all client requests, handling routing, security, and load balancing.
- **Product Service (Port 5001):** Manages all product-related information and operations.
- **Order Service:** Handles the lifecycle of customer orders.
- **Notification Service:** Responsible for sending various notifications (e.g., order confirmations, shipping updates).
- **RabbitMQ Broker:** Our central message broker, facilitating asynchronous communication between services and enabling the EDA pattern.

Today, our goal is to bridge the gap from this logical component view to a tangible, physical view—how these services will actually run on servers and infrastructure. We will then perform a critical analysis to understand the trade-offs inherent in these deployment decisions.

Knowledge Base: The Physical View (Deployment Diagram)

The Physical View, often represented by a UML Deployment Diagram, is crucial for understanding how our software system will operate in a real-world environment. It provides a visual map of the physical architecture, illustrating where software artifacts reside and how they interact.

Purpose of a Deployment Diagram:

> A Deployment Diagram shows the physical mapping of software artifacts (components, services, executables) onto execution environments (Nodes). It's the blueprint for how your system is deployed and operationalized.

Key Elements:

Node (3D Box)

Represents a hardware or software execution environment. This can be a physical server, a virtual machine (VM), a server cluster, or a Docker container. Nodes are where your software lives and runs.

Artifact (Document Icon)

Represents a concrete, physical piece of information that is used or produced by a software development process. Examples include executable files (e.g., `product_service.py`), libraries, database schemas, scripts, or configuration files. Artifacts are deployed to Nodes.

Communication Path (Line)

Illustrates the network connections and communication protocols between Nodes. This shows how different parts of your deployed system interact, specifying protocols like HTTP/S for web traffic or AMQP for message queuing.

Activity 1: Defining the Physical Nodes (ShopSphere)

Our first practical step in building the Deployment View for our ShopSphere application is to identify and define the physical nodes that will host our services. This involves categorizing the types of environments needed and their specific purposes within the architecture.

Goal:

To precisely define the hardware and software execution environments required for deploying each component of our microservices architecture.

Edge Network	Server	External-facing infrastructure responsible for traffic ingress, load balancing, and initial request routing.	API Gateway
Application Cluster	Cluster/VMs	A scalable and elastic environment designed to host the core business logic services. This allows for horizontal scaling of individual microservices.	Product Service, Order Service, Notification Service
Messaging Server	Server	A dedicated server (or cluster) specifically for hosting our message broker, ensuring robust and performant asynchronous communication.	RabbitMQ Broker
Database Server	Server	A specialized server (or cluster) optimized for persistent data storage and retrieval, supporting various database technologies.	SQL/NoSQL Databases (e.g., PostgreSQL, MongoDB)

Practical Activity (Hands-On): Drawing Nodes

Now that we've defined our logical nodes, it's time to translate these concepts into a visual representation using a UML Deployment Diagram. For this activity, we'll use a diagramming tool like draw.io, which provides dedicated UML Deployment stencils.

Tool:

draw.io (or any other UML diagramming tool with Deployment Diagram capabilities).

Steps for Drawing Nodes:

01

Initialize Drawing Canvas

Open draw.io and select a blank diagram. Locate the "UML" stencil set, specifically the "Deployment" section.

02

Draw Main Nodes

Drag and drop three main 3D box shapes onto the canvas, representing: **Edge Network**, **Application Cluster**, and **Messaging Server**.

03

Apply Stereotypes

For each node, apply the appropriate UML stereotype to clearly indicate its nature:

- Use `<<device>>` for physical hardware nodes (e.g., a dedicated server machine).
- Use `<<execution environment>>` for software environments like a Docker container, Kubernetes pod, or a virtual machine.

04

Label Nodes Clearly

Ensure each node is clearly labeled with its descriptive name (e.g., "Edge Network", "Application Cluster").

Activity 2: Mapping Artifacts to Nodes

With our physical nodes drawn, the next critical step is to place the actual executable software artifacts—the services we've developed—onto these nodes. This mapping clarifies exactly where each piece of our application will reside and run.

Goal:

To visually associate the executable services (artifacts) with their respective deployment environments (nodes) within the UML Deployment Diagram.

Artifact Mapping Details:

- **Edge Network:** This node will host the `gateway.py` artifact, which is our API Gateway. It serves as the primary entry point and orchestrates incoming client requests, routing them to the appropriate backend services.
- **Application Cluster:** This scalable environment is designed to run our core business logic services. It will host the `product_service.py` artifact (handling product catalog operations) and the `order_service.py` artifact (managing customer orders). These services benefit from the cluster's ability to scale horizontally based on demand.
- **Messaging Server:** Dedicated to asynchronous communication, this node will host the `rabbitmq_broker` artifact. This broker is essential for our Event-Driven Architecture, enabling services to communicate without direct coupling, enhancing resilience and scalability.

Visually, these artifacts should be drawn inside their respective 3D node boxes on the deployment diagram, typically represented by a document icon with the artifact's name.

Activity 3: Modeling Communication Paths

After mapping artifacts to nodes, the deployment diagram is incomplete without illustrating how these physically separated components communicate. Defining communication paths is vital for understanding network dependencies, potential bottlenecks, and security considerations.

Goal:

To clearly depict the communication channels and protocols between the nodes in our ShopSphere deployment, highlighting both external and internal interactions.

Key Communication Paths:

Client $\leftrightarrow$ Edge Network

This path represents the interaction between end-users (clients) and our application's entry point. Communication here is primarily over **HTTP/S**, ensuring secure web traffic over the Internet. The Edge Network acts as the first line of defense and request routing.

Edge Network $\rightarrow$ Application Cluster Cluster

Once requests pass through the Edge Network, they are routed internally to the Application Cluster. This communication typically uses **HTTP** (or internal HTTPS) for efficient internal routing and service invocation. The API Gateway intelligently directs traffic to the correct microservice instances within the cluster.

Application Cluster $\leftrightarrow$ Messaging Messaging Server

This critical path enables asynchronous event-driven communication. Services within the Application Cluster (e.g., Order Service) publish events to, and consume events from, the Messaging Server (RabbitMQ Broker) using the **AMQP** protocol. This decoupling enhances fault tolerance and scalability.

On your diagram, these paths should be drawn as lines connecting the respective nodes, with clear labels indicating the protocol used. This detail helps convey the runtime dependencies and data flow across the deployed system.

Visualization: The UML Deployment Diagram

After completing the previous activities, your UML Deployment Diagram for ShopSphere should now be a comprehensive visual representation of our physical architecture. This diagram serves as a crucial document for developers, operations teams, and stakeholders.

A Well-Formed Deployment Diagram for ShopSphere Will Show:

- **External Client:** Representing the end-users interacting with the system.
- **Edge Network Node:** Hosting the API Gateway artifact.
- **Application Cluster Node:** Hosting Product Service, Order Service, and Notification Service artifacts.
- **Messaging Server Node:** Hosting the RabbitMQ Broker artifact.
- **Database Server Node:** Hosting the actual database instances (e.g., ProductDB, OrderDB).
- **Communication Paths:** Clearly labeled lines showing HTTP/S, HTTP, and AMQP connections between these nodes.

> This diagram is more than just a picture; it's a living document that guides deployment strategies, identifies infrastructure requirements, and aids in troubleshooting. It provides a shared understanding of how the logical components come together to form a resilient and scalable physical system.

Ensure all stereotypes (like <> and <>) are correctly applied and that artifact names reflect the actual files or deployable units.

Knowledge Base: Quality Attribute Analysis (ATAM)

Once an architectural design is established, it's imperative to evaluate its efficacy, particularly in satisfying crucial quality attributes. This is where the Architecture Trade-off Analysis Method (ATAM) comes into play.

What is ATAM?

ATAM is a systematic, scenario-based approach to evaluate how well a software architecture satisfies its quality attributes (also known as Architectural Significant Requirements or ASRs). Developed by the Software Engineering Institute (SEI), it helps stakeholders understand the architectural risks, sensitivities, and trade-offs.

Core Idea: Trade-offs are Inevitable

A fundamental principle of ATAM is that you cannot maximize all quality attributes simultaneously. For instance, striving for extremely high security often introduces overhead that can negatively impact performance or usability. ATAM helps explicitly identify and document these trade-offs, making them transparent to all stakeholders.

"Maximizing security often reduces performance, just as maximizing performance might compromise maintainability." ATAM's strength lies in highlighting these intrinsic compromises.

Tool: Using Scenarios to Stress-Test the Architecture Mentally

ATAM relies heavily on architectural scenarios—short, concise stories that describe how the system is expected to behave under specific conditions related to a quality attribute. By "walking through" these scenarios with the architecture, we can identify potential weaknesses, areas of risk, and the impact of architectural decisions on performance, security, reliability, and other crucial qualities.

The Scenario Structure: ASR Validation

To effectively conduct Quality Attribute Analysis using ATAM, scenarios must be well-structured and detailed. A robust scenario explicitly defines the conditions and expected outcomes, allowing for a clear evaluation of the architecture against specific ASRs.

Elements of a Good Scenario:

Source

Identifies who or what initiates the action. This could be a specific user role, an external system, or a quantifiable load. **Example:** 10,000 Concurrent Users.

Stimulus

Describes the event or condition that occurs. This is the action that the source performs or the environmental change that impacts the system. **Example:** A spike in traffic (e.g., during a flash sale).

Artifact

Specifies the part of the system that is affected by the stimulus. This could be a specific component, service, or data store. **Example:** The Product Catalog service.

Environment

Defines the conditions under which the stimulus occurs. This provides crucial context for the scenario. **Example:** During Black Friday promotional period.

Response

States the measurable and expected result if the architecture successfully handles the stimulus under the given environment. **Example:** System latency remains below 2 seconds and no data loss occurs.

By articulating scenarios with this level of detail, we create a common language for discussing and analyzing architectural implications, moving beyond abstract quality attributes to concrete, testable expectations.