PHENIKAA UNIVERSITY

PHENIKAA SCHOOL OF COMPUTING

Deployment View & ATAM

Course: Software Architecture

Course Class: CSE703110-1-2-25(N02)

Group 7:

1. Le Thi Kieu Trang **ID: 23010502**

2. Quach Huu Nam **ID: 23012358**

3. Trieu Tien Quynh **ID: 23010648**

Hanoi, January 7, 2026

Lab 8 Deployment View & ATAM

1. Overview

Having completed the source code implementation (Coding) from Lab 1 through Lab 7, Lab 8 transitions into the documentation and architectural evaluation phase.

Objectives:

Construct a UML Deployment Diagram to visualize how software components are allocated onto the hardware/virtualization infrastructure.

Apply the ATAM (Architecture Trade-off Analysis Method) to compare the current Microservices architecture against the traditional Monolithic architecture, based on quality attributes (NFRs).

2. Activity Practice 1: UML Deployment Diagram

Goal: Model the physical allocation of software components (artifacts) to execution environments (nodes).

2.1. Identify Nodes (Physical Environments)

Based on the current project structure, the system is designed for a typical cloud-native deployment:

Client Device: The user's web browser running the modern frontend from frontend.

Load Balancer (e.g., Nginx): Acts as the entry point, distributing incoming traffic to available instances.

Application Cluster (Docker/Kubernetes): The execution environment for the Python-based microservices.

2.2. Place Artifacts (Services)

The following artifacts (from your microservices directory) are deployed within the Application Cluster as independent containers:

API Gateway Artifact (api_gateway)

Enrollment Service Artifact (enrollment_service)

Course Service Artifact (course_service)

Student Service Artifact (student_service)

Identity Service Artifact (identity_service) - Handles Auth/JWT.

Notification Service Artifact (notification_service)

Message Broker Artifact (RabbitMQ/Redis): Used for asynchronous communication between services.

2.3. Place Data Stores

Each microservice maintains its own dedicated database (Database-per-Service pattern):

enrollment_db, course_db, student_db, etc., connected to their respective service repositories.

2.4. Draw Associations (Communication)

Client → API Gateway: HTTP/REST (Port 5000/8000).

API Gateway → Internal Services: Internal REST calls (Course, Student, etc.).

Enrollment Service → Message Broker: Publishes "Enrollment Success" events.

Message Broker → Notification Service: Consumes events to trigger student emails.

3. Activity Practice 2: Quality Attribute Analysis (Simplified ATAM)

3.1. Define Scenarios (Test Cases)

Scalability Scenario (SS1): "During the first 15 minutes of the course registration period, the system must handle a 20x spike in concurrent students (10,000+) attempting to register for classes simultaneously."

Availability Scenario (AS1): "The notification_service fails due to an SMTP error. The system must still allow students to complete their course registration without interruption."

3.2. Architecture Evaluation Matrix

Quality Attribute	Scenario	Monolithic (Legacy) Approach	Microservices Approach
Scalability	SS1 (Registration Spike)	Response: Must scale the entire monolithic process (including Grade, Faculty, etc.) even if only Enrollment needs	Response: Can independently scale-out enrollment_service and course_service containers. The enrollment_db can be optimized specifically for high writes. Efficient scaling.

		capacity. This is resource-heavy and inefficient.	
Availability	AS1 (Notification Failure)	Response: If notification logic is tightly coupled in enrollment_service.py, a failure in email sending may crash the transaction or block the user.	Response: Thanks to the Event-Driven architecture, enrollment_service sends a message to the Broker and finishes. The notification failure has zero impact on the registration success.

3.3. Identify Trade-offs

Trade-off: The Microservices architecture provides Superior Scalability and Fault Isolation (essential for high-traffic registration periods) but introduces increased complexity in deployment (Docker, API Gateway, Message Broker) and data consistency management compared to the simpler Monolith.

4. Architectural Trade-offs Summary

Based on the ATAM analysis above, the team has drawn the following conclusions regarding the trade-offs involved in transitioning from Monolithic to Microservices architecture:

4.1. Advantages

Scalability: The system is flexible, allowing for the independent scaling of high-load modules (such as Course Registration) without impacting the performance of the entire system.

Fault Tolerance: A failure in a single sub-service (such as Email) does not crash the entire system, thanks to fault isolation mechanisms and asynchronous processing.

Technology Heterogeneity: Enables the use of diverse technologies stack (e.g., Python for Backend, Node.js for real-time services if needed) without conflict.

4.2. Disadvantages

Complexity: Operating an infrastructure of 10 Microservices, an API Gateway, and message brokers (RabbitMQ) is significantly more complex than managing a single Monolithic app, requiring specialized knowledge in Docker and DevOps.

Data Consistency: Decoupling databases makes operations requiring high data integrity (Transactions) more challenging, as simple JOINs or ACID transactions across services are no longer applicable.

Network Latency: Inter-service communication via HTTP/AMQP is inherently slower than direct in-memory function calls used in monolithic architectures..

5. Final Conclusion

Through the series of practical exercises from Lab 1 to Lab 8, the team has successfully constructed a modern Academic Management System:

Successful Transition: Shifted from a Monolithic mindset to Microservices Architecture.

Comprehensive Implementation: Fully applied key design patterns including Layered Architecture, Database-per-Service, API Gateway, and Event-Driven Architecture.

Proven Efficiency: Demonstrated the advantages of the new architecture through load testing and fault tolerance scenarios, while clearly recognizing the operational complexity challenges.

The system is now architecturally ready for further expansion and real-world deployment.