Software Architecture Course: 8-Lecture Proposal

Course Structure Requirements

Requirement	Description	Fulfillment in Lectures
Architectural Coverage	Must cover at least three distinct architectural patterns.	**Layered**, **Microservices**, **Event-Driven Architecture (EDA)**.
Pace/Scope	Must progress from fundamental concepts to medium-complexity implementation.	Lectures 1-3 (Beginner/Foundational), Lectures 4-8 (Medium/Advanced Patterns).
Practical Application	Every lecture must include detailed **instructional steps**, **installation guides**, and **coding steps**.	All Lectures 1-8 are designed as practical labs based on the **ShopSphere** scenario.
Documentation	Must cover essential documentation methods (UML, Requirements Elicitation, Quality Attributes).	Lectures 1, 2, 4, 8.

8-Lecture Detailed Structure

The course uses the **ShopSphere E-commerce Platform** as the core case study, progressing from a simple monolith to a distributed system.

Module 1: Foundational Architecture & Monolith Design

Lecture	Title & Architectural Focus	Description & Detailed Instructional Steps
Lecture 1	**Requirements Elicitation & Modeling (Pre-Architecture)**	**Focus:** Understanding the problem domain. **Instructions:** 1. Identify Actors (Customer, Admin). 2. Document Functional/Non-Functional Requirements (FRs/NFRs). 3. Define three **Architecturally Significant Requirements (ASRs)** (e.g., High Scalability, Security). 4. Model the system context using a **UML Use Case Diagram** (draw.io) detailing the *Make Purchase* path with $\ll\text{include}\gg$ and $\ll\text{extend}\gg$.

Lecture	Title & Architectural Focus	Description & Detailed Instructional Steps
Lecture 2	**Layered Architecture Pattern (Logical View)**	**Focus:** Designing the Monolithic structure. **Instructions:** 1. Define the four standard layers: Presentation, Business Logic, Persistence, Data. 2. Define the strict **downward dependency rule**. 3. Identify components for the *Product Catalog* feature (Controller, Service, Repository). 4. Model the dependencies using a **UML Component Diagram** (draw.io), showing provided/required interfaces (lollipop/socket notation).
Lecture 3	**Layered Monolith Implementation (CRUD)**	**Focus:** Building the designed monolith and enforcing layer separation in code. **Installation:** Python, **Flask**. **Coding Steps:** 1. Set up the project structure with separate packages (presentation, business_logic, persistence). 2. Implement the ProductRepository using an in-memory list. 3. Implement the ProductService with a business rule (e.g., price must be positive). 4. Implement the ProductController to expose a REST API (POST, GET). **Verification:** Test that the Controller successfully calls the Service, which calls the Repository.

Module 2: Transition to Distributed Systems

Lecture	Title & Architectural Focus	Description & Detailed Instructional Steps
Lecture 4	**Microservices Decomposition & Context**	**Focus:** Breaking the monolith into independent services. **Instructions:** 1. Define Microservice Boundaries based on **Business Capabilities** (Product, Order, User, Cart). 2. Document the **Service Contract (API Specification)** for the Product Service. 3. Define the **data ownership** principle (each service owns its data). 4. Model the system's external interaction using the **C4 Model (Level 1: System Context Diagram)**.
Lecture 5	**Implementing an Independent Microservice**	**Focus:** Building a *data-owning* service in isolation. **Installation:** Python, **Flask-SQLAlchemy**, SQLite. **Coding Steps:** 1. Create a **standalone** project for the **Product Service**. 2. Define the Product model using SQLAlchemy. 3. Implement the persistence

Lecture	Title & Architectural Focus	Description & Detailed Instructional Steps
		layer (ORM queries) and the API layer (Controller) within the service. 4. Expose the GET /api/products endpoint. **Verification:** Run the service on a dedicated port (**5001**) and confirm it works without any other components.
Lecture 6	**API Gateway Pattern & Security**	**Focus:** Implementing the single entry point for all client requests. **Installation:** Python, Flask, **requests**. **Coding Steps:** 1. Set up the **Gateway** application on port **5000**. 2. Implement a **Security Check Stub** (token validation) to block unauthorized requests. 3. Implement **Routing Logic** to forward requests to the Product Service (port 5001). 4. Implement **Failure Handling** (503 Service Unavailable) if the backend service is down. **Verification:** Test authorized access, unauthorized access (401), and service failure (503).

Module 3: Asynchronicity, Documentation & Quality

Lecture	Title & Architectural Focus	Description & Detailed Instructional Steps
Lecture 7	**Event-Driven Architecture (EDA) & Decoupling**	**Focus:** Implementing asynchronous communication using events. **Installation:** Docker/Local **RabbitMQ**, Python **Pika**. **Coding Steps:** 1. Implement the **Order Service** (Producer) to publish an OrderPlacedEvent to the queue. 2. Implement the **Notification Service** (Consumer) to subscribe to the queue and process the event (simulating sending an email). 3. Use time.sleep() in the Consumer to simulate latency. **Verification:** Observe that the Producer quickly publishes events while the Consumer processes them slowly, proving **non-blocking decoupling**.
Lecture 8	**Deployment View & Quality Attribute Analysis (ATAM)**	**Focus:** Documenting the final system and evaluating NFRs. **Instructions:** 1. Create a **UML Deployment Diagram** showing nodes (VMs, Load Balancer), artifacts (Microservices, Broker), and network connections. 2. Define **Scalability** and **Availability** scenarios (e.g., 10x traffic spike). 3. Conduct a simplified **ATAM**

Lecture	Title & Architectural Focus	Description & Detailed Instructional Steps
		analysis, comparing how the **Monolith** fails versus how the **Microservices** solution handles the scenarios. 4. Document the final architectural **Trade-offs** (e.g., Complexity vs. Fault Isolation).