COMP41720 Distributed Systems: Architectural Principles LAB 4: Building a Microservices-Style Application

1. Lab Overview

This lab provides you with **hands-on experience** in applying the fundamental architectural principles and challenges associated with building distributed systems using **microservices**. Moving beyond theoretical concepts, you will design and implement a simple application based on the microservices architectural style, focusing on **architectural decision-making** and **trade-off analysis**.

You will delve into topics such as designing **service boundaries**, implementing **inter-service communication strategies**, and considering **deployment implications**. This practical work aims to reinforce the theoretical knowledge gained in lectures and deepen your understanding of why certain architectural decisions are made in modern distributed systems.

---

2. Learning Objectives

By the end of this lab, you will be able to:

- **Design and define appropriate service boundaries** for a microservices-style application, considering factors that influence granularity.
- **Implement and compare different inter-service communication patterns**, such as synchronous (e.g., REST, gRPC) and asynchronous (e.g., messaging), and analyze their **architectural implications and trade-offs** (e.g., coupling, latency, performance, extensibility).
- **Deploy a distributed application** using modern cloud-native tools like **containerization (Docker) and orchestration (Kubernetes)**.
- **Reason architecturally about your design choices**, providing justifications for *why* specific patterns and technologies were selected, particularly in the context of desired system properties like scalability and maintainability.

---

3. Context

In Week 10, the course delves into a **"Microservices Deep Dive,"** covering designing service boundaries, inter-service communication strategies, and deployment considerations, along with an introduction to **API Gateways and Service Meshes**. This lab is your opportunity to apply these concepts, understanding that in software architecture, there are "no best practices" but rather a need to find the "least worst combination of trade-offs" for a given scenario. You are expected to critically evaluate and justify your architectural decisions.

---

4. Task Description

You are required to **design and implement a simple application comprised of at least two distinct microservices** that interact with each other.

Your application should clearly demonstrate:

- **Well-defined service boundaries**: Explain the rationale behind how you logically separated your application into distinct services. Consider using "granularity disintegrators" (e.g., code volatility, differing scalability/throughput needs, fault tolerance, extensibility) and "granularity integrators" (e.g., database transactions, shared code, tight data relationships) to justify your choices.
- **Inter-service communication**: Implement a communication mechanism between your services. You are encouraged to explore different patterns (e.g., synchronous REST/gRPC, or asynchronous messaging via a message broker) and discuss the trade-offs of your chosen approach.
- **Deployment of microservices**: Containerize your services using Docker and deploy them onto an orchestration platform like Kubernetes. This will enable you to manage and run your distributed application effectively.

(Optional but Encouraged): If time permits and it logically fits your application's design, you may consider introducing:

- A basic **API Gateway** to manage incoming requests and route them to the appropriate services.
- Simple **Service Mesh** concepts for managing operational concerns across your microservices (e.g., basic logging, or a conceptual discussion of traffic management).

---

5. Requirements

1. **Technology Choice:** Select a modern programming language and framework(s) that support building microservices (e.g., Python with Flask/Django, Node.js with Express, Java with Spring Boot, Go with Gin/Echo, C# with ASP.NET Core, etc.). You are encouraged to choose a stack you are comfortable with or interested in exploring. **Modern versions of Java are recommended, avoiding Java 8 as per module feedback**.
2. **System Design:** Implement **at least two distinct components/services** that encapsulate specific business capabilities or domains.
3. **Communication Implementation:** Clearly implement the chosen inter-service communication pattern(s).
4. **Deployment:**
   - **Containerize** each microservice using **Docker**.
   - **Deploy** your containerized services onto a local **Kubernetes** cluster (e.g., Minikube, Kind, Docker Desktop's Kubernetes) or a cloud-based one if preferred.
5. **Functionality:** Each service should implement at least one basic function demonstrating its role and interaction with other services.
6. **Architectural Justification:** Throughout your design and implementation, **think about the architectural principles and trade-offs** discussed in lectures. Be prepared to articulate the *why* behind your choices, not just the *how*.

7. **Code Quality:** Write clean, well-structured, and readable code.
8. **Environment Clarity:** Ensure your project includes clear instructions on how to set up, build, and run your code, including any specific dependencies or required language versions (e.g., "Requires Python 3.9", "Built with Node.js v16"). This addresses feedback from previous years regarding clarity on environment setup.

---

6. Deliverables

1. **Source Code Repository Link:** Provide a link to your code repository (e.g., GitHub, GitLab) containing all application code, Dockerfiles, and Kubernetes manifests used for your application.
2. **Lab Report (PDF format):** A comprehensive document (e.g., 3-5 pages) explaining your architectural decisions and implementation.
   - **Introduction:** Briefly state the lab's purpose, your chosen technologies, and the application's basic functionality.
   - **System Design & Setup:**
     - Provide a **clear diagram of your microservices architecture**, illustrating service boundaries and communication flows.
     - Detail your **Docker containerization** (e.g., Dockerfiles) and **Kubernetes deployment** (including YAML manifests).
     - Include clear **setup and run instructions** for the entire application.
   - **Architectural Analysis & Justification:**
     - **Microservice Granularity:** Explain how you defined your service boundaries. Discuss the **granularity disintegrators and integrators** you considered, and how they influenced your decision-making.
     - **Inter-service Communication:** Describe the communication patterns you implemented (e.g., REST, gRPC, messaging). Provide a **detailed analysis of the architectural trade-offs** associated with your chosen communication strategy (e.g., coupling, latency, error handling, performance vs. data consistency, extensibility vs. agility). Justify *why* this approach was appropriate for your application.
     - **(Optional) API Gateway/Service Mesh:** If implemented, explain their role and architectural benefits in your system.
     - **Key Architectural Decisions (ADRs):** Document at least 2-3 significant architectural decisions using a simplified Architectural Decision Record (ADR) format. For each decision, include:
       - **Context:** The problem or situation prompting the decision.
       - **Decision:** The specific architectural choice made.
       - **Consequences (Trade-offs):** Both the positive and negative impacts, and the trade-offs considered.
   - **Conclusion:** Summarize your key learnings about designing microservices, any challenges encountered, and insights gained from this hands-on experience.

---

7. Assessment Criteria

Your lab submission will be assessed based on the following criteria:

1. **Correctness and Functionality (40%):**
    - Successful setup, deployment, and operation of the microservices application on Kubernetes.
    - Correct implementation of defined service functionalities and inter-service communication.
2. **Depth of Architectural Design and Justification (40%):**
    - **Clear and insightful analysis of microservice granularity and the rationale behind your chosen service boundaries**.
    - **Strong justifications for your inter-service communication patterns**, linking observations to distributed systems principles and architectural characteristics (e.g., scalability, availability, fault tolerance, maintainability).
    - Effective use of trade-off analysis and clear articulation of the *why* behind your architectural decisions.
    - Quality of documented Architectural Decision Records (ADRs), if included.
3. **Clarity and Organization of Report (10%):**
    - Well-structured, concise, and easy-to-understand report.
    - Effective use of diagrams and visual aids to explain the architecture.
4. **Code Quality and Readability (10%):**
    - Clean, well-structured, and readable code.
    - Easy-to-follow setup and running instructions for your application.

---

8. Important Notes & Support

- **Start Early!** Setting up distributed systems environments, especially with containerization and orchestration tools like Docker and Kubernetes, can have a learning curve. Give yourself ample time for setup and troubleshooting.
- **Read Documentation:** Familiarize yourself with the documentation for your chosen programming languages, frameworks, resilience libraries, and deployment tools (Docker, Kubernetes, etc.).
- **Focus on the "Why":** A core theme of this course is understanding the *why* behind architectural decisions. Beyond getting the code to work, ensure you deeply understand *why* certain architectural patterns are chosen, *why* certain behaviors occur, and *why* different choices yield different trade-offs.
- **Incremental Builds:** Implement and test each part of your microservices application incrementally to simplify debugging.
- **Utilize Support:** Teaching Assistants (TAs) and instructors are available for questions during lab sessions and office hours. **Do not hesitate to ask for clarification on vague instructions or technical challenges**. We are working to provide clearer instructions and better support based on past feedback.
- Your chosen technology stack is flexible, but ensure you can get help if needed (e.g., choose popular frameworks).