**PHENIKAA UNIVERSITY**

**PHENIKAA SCHOOL OF COMPUTING**

# WEEKLY REPORT

# TOPIC: DEVELOPING A MOBILE MESSAGING APPLICATION

| | | |
|---|---|---|
| **Course Class:** | Software Architecture-1-2-25 (N01) | |
| **Instructor:** | M.Sc. Vũ Quang Dũng | |
| **Group:** | 04 | |
| **Group Members:** | Đỗ Trịnh Lệ Quyên | 23010485 |
| | Vũ Viết Huy | 23010699 |
| | Nguyễn Mạnh Cường | 23010064 |

**Hanoi, January 23, 2026**

## CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

# 1 INTRODUCTION

## 1.1 Lab Overview

This lab focuses on architecture documentation and quality attribute evaluation for the **Mobile Messaging Application**. It shifts from coding to designing the physical deployment and using a simplified **Architecture Trade-off Analysis Method (ATAM)** to assess how the architecture meets key Non-Functional Requirements **(NFRs)**, such as **Scalability** and **Availability**. Given the project's monolithic (layered/serverless) architecture using **Flutter** for the client and **Supabase** for the backend, the activities are adapted accordingly, without reliance on microservices from prior labs. This report documents the **UML Deployment Diagram** and **ATAM** analysis, integrating insights from previous labs (e.g., layered design from Lab 2 and service decomposition from Lab 4).

## 1.2 Objectives

- Create a **UML Deployment Diagram** to visualize the physical setup of the architecture **(Deployment View)**.
- Conduct a simplified **Architecture Trade-off Analysis Method (ATAM)** focusing on **Scalability** and **Availability**.
- Compare the **Monolithic** (Layered) vs. Microservices architecture concerning the chosen **Quality Attributes**.

# 2 DEPLOYMENT VIEW

## 2.1 Goal and Detailed Explanation of the Diagram

**Goal:** Model the physical allocation of software components **(nodes)** to execution environments **(hardware/containers).**

**The UML Deployment Diagram** presented in this section is tailored to the **monolithic/serverless** architecture of our **Mobile Messaging Application**, utilizing **Flutter** for the client-side and Supabase as the **Backend-as-a-Service (BaaS)** for backend operations. This diagram adheres strictly to standard **UML** conventions, employing **3D** box shapes for nodes (representing computational resources like devices or cloud instances), artifact symbols (file-like icons for deployable software units such as applications or services), database symbols for data storage, and dashed arrows for communication associations with labeled protocols. Unlike a microservices-based setup (as referenced in Lab 8's original guidelines), our monolithic approach integrates all business logic (e.g., authentication, messaging, group management, and recommendations) within a single **Flutter App** artifact on the client side, with backend support centralized in Supabase. This design draws from Lab 2's layered architecture (**Presentation Layer in Flutter** for **UI**, **Business Layer** for logic like recommendation matching, **Persistence Layer via Supabase SDK**, and **Data Layer** in the **PostgreSQL** database) and Lab 4's decomposition by business capabilities (e.g., Identity & Profile, Chat & Real-time, Group & Recommendation contracts mapped to unified artifacts and tables).

Key elements and their rationale:

- **Nodes:** The diagram features two primary nodes to reflect the client-serverless paradigm. The "Client Device (Mobile Phone)" node represents the user's physical hardware (e.g., smartphone), where the entire frontend and much of the business logic reside, aligning with the mobile-first nature of the application (as per FRs in Lab 1, such as user authentication and real-time messaging on devices). This node emphasizes mobility and offline capabilities, such as local caching for recommendations during network disruptions (supporting ASR-2: High Availability from Lab 1). The "Cloud Infrastructure (Supabase Instance)" node encapsulates the backend execution environment, which is a managed cloud service (VMs/containers abstracted by Supabase). This replaces the need for separate load balancers or Kubernetes clusters in microservices, simplifying deployment while leveraging serverless auto-scaling for NFR-03: Scalability up to 1,000 concurrent users. The choice of only two nodes highlights the monolithic simplicity, reducing complexity compared to distributed microservices nodes.
- **Artifacts:** Within the Client Device node, the "Flutter App (Monolithic Mobile Application)" artifact is the core deployable unit, encompassing all subsystems (e.g., Account Management for login/register from Lab 1's use cases, Messaging Management for send/receive messages, and Recommendation Subsystem for hobby-based matching from Lab 2). This artifact handles UI rendering, local computations (e.g., filtering recommendations to minimize server calls, as per data flow analysis in Lab 2), and direct SDK integrations with Supabase, promoting efficiency in a single codebase. In the Cloud Infrastructure node, artifacts include "Supabase Auth Service" (for secure identity contracts like login via IAuthRepository from Lab 4), "Supabase Realtime Service" (for low-latency chat and notifications via WebSockets, addressing ASR-1: Real-time Performance), and "Supabase Storage Service" (for media uploads in group messaging, as per FR-3.2 in Lab 1). These artifacts are not independent microservices but modular components within Supabase, ensuring tight integration and reduced latency, though at the cost of potential single-point failures if the cloud instance experiences issues.
- **Data Stores:** The "Supabase Database (PostgreSQL)" is depicted as a database symbol nested within the Cloud node, with a folder illustrating specific tables (e.g., auth.users for authentication, public.profiles for user data, public.conversations/messages for chats, public.groups/hobbies for recommendations). This unified database instance supports the monolithic design by centralizing data persistence, contrasting with microservices' separate databases per service (as noted in Lab 8). Connections from artifacts to the database via Supabase SDK enable efficient CRUD operations (e.g., storing messages for offline queuing, enhancing availability). The table structure is derived from Lab 4's service contracts, ensuring data isolation at the schema level (e.g., public schema for shared access) while maintaining ACID properties for reliability.
- **Associations (Communication):** Dashed arrows denote network dependencies, labeled with protocols to specify interaction details. The Client Device

communicates with Cloud Infrastructure via HTTPS for general REST calls (e.g., auth/login/register, ensuring secure data transmission). The Flutter App artifact connects to Supabase Auth via WSS (WebSockets) for real-time features like chat updates and live recommendations, minimizing polling overhead and achieving

This diagram effectively visualizes how the architecture allocates resources to meet NFRs, such as fault tolerance through cloud redundancy, while highlighting monolithic trade-offs like reduced modifiability (ASR-3 from Lab 1) compared to microservices.

## 2.2 UML Deployment Diagram for Monolithic Architecture

The diagram is designed using standard UML elements: 3D box nodes, artifact symbols, database symbols, and dashed arrows with labels for communication. It reflects the monolithic structure with Flutter + Supabase, incorporating layered responsibilities (Presentation in client, Business/Persistence in app logic, Data in Supabase) from Lab 2 and business capabilities (Identity, Chat, Group, Recommendation) from Lab 4.

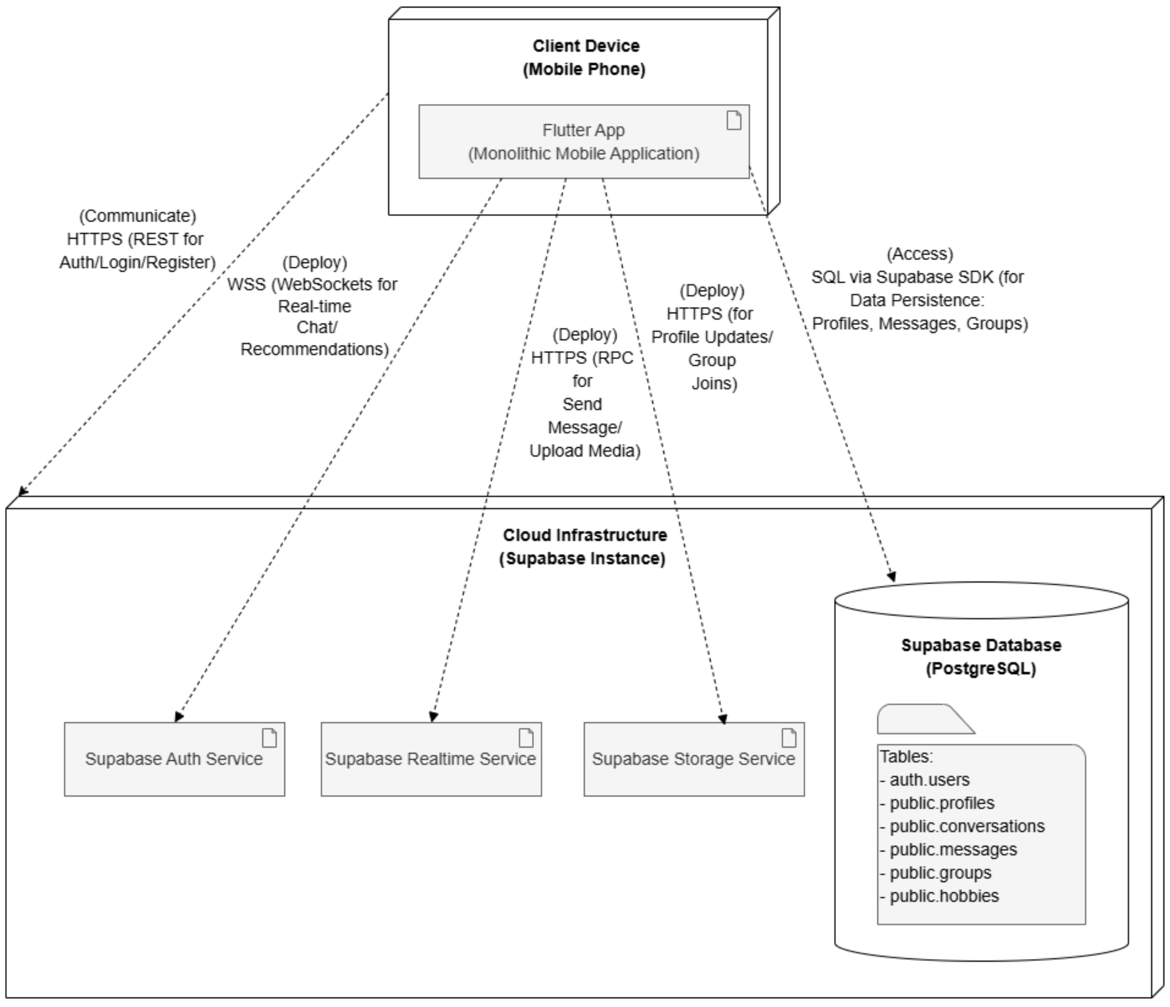

*Figure 1: UML Deployment Diagram*

Description: The Client Device node hosts the Flutter App, which communicates via HTTPS/WSS to the Cloud Infrastructure. The Supabase artifacts handle auth, real-time, and storage, with data persisted in a unified PostgreSQL database. This setup supports NFRs like performance (low latency via WebSockets) from Lab 1.

## 3 QUALITY ATTRIBUTE ANALYSIS (ATAM)

### 3.1 Define Scenarios

Based on NFRs (e.g., NFR-03: Scalability for 1,000 concurrent users) and ASRs (e.g., ASR-1: Real-time Performance, ASR-2: High Availability) from Lab 1:

- **Scalability Scenario (SS1):** "During peak hours (e.g., a viral event in a public channel), the system must handle a sudden 10x spike in concurrent users (from 1,000 to 10,000) sending messages, joining groups, and receiving recommendations without service interruption or latency exceeding 1 second."
- **Availability Scenario (AS1):** "The real-time messaging component (e.g., Supabase Realtime) fails completely for 1 hour due to a cloud outage. The system must still allow users to send messages (queued for later delivery), update profiles, and process offline recommendations without total system downtime."

### 3.2 Evaluate Architectures against Scenarios

| Quality Attribute | Scenario | Monolithic (Layered) Approach | Microservices Approach |
|---|---|---|---|
| 1. Scalability | SS1 (10x User Spike) | Response: Must scale the entire application (Flutter app logic + Supabase instance) even if only messaging/recommendations need capacity. Supabase serverless auto-scales database and realtime connections, but client-side (Flutter) may bottleneck with heavy local logic (e.g., recommendation filtering). Medium effectiveness due to serverless, but less flexible without isolated components. May exceed latency if unoptimized (based on NFR-03). | Response: Can scale independent services (e.g., Chat Service and Recommendation Service). Database sharded/replicated for high-read operations like group joins. Higher effectiveness with fault isolation and horizontal scaling (e.g., Kubernetes), |

| | | | |
|---|---|---|---|
| | | | supporting 10,000 users with |
| 2. Availability | AS1 (Real-time Component Fails) | Response: If realtime fails, messaging is impacted due to tight coupling in Supabase. However, built-in replication and failover (from ASR-2) allow queuing in database and later sync. Profile updates and offline recommendations (local in Flutter) continue, but overall availability drops without separate load balancers. | Response: Event-Driven (e.g., Kafka) ensures failure in Notification/Chat Service doesn't affect Profile/Group Services. Messages queue in broker and process on recovery, with zero core function impact (from ASR-2). High fault isolation supports modifiability (ASR-3). |

*Table 1: Evaluate Architectures*

### 3.3 Identify Trade-offs

**Trade-off:** The Monolithic **(Layered/Serverless)** architecture offers simplicity and low operational overhead (no need to manage **Docker/Kubernetes**, **API Gateway**, or **separate Message Brokers**), suitable for the small-scale messaging app with **Flutter + Supabase**, where scalability and availability are handled automatically by the cloud provider (supporting **NFR-03** and **ASR-2**). However, it sacrifices resilience under high load (e.g., 10x spike may cause system-wide bottlenecks) and poorer fault isolation, increasing downtime risks if one component fails. In contrast, **Microservices** provide superior **Scalability** and **Availability** through independent scaling and event-driven design, but increase deployment and maintenance complexity, which may be unnecessary for the initial app (based on Lab 4's business capability decomposition but monolithic implementation).

## 4 COMPARISON OF ARCHITECTURES

**The Monolithic** (**Layered**) architecture excels in ease of development and maintenance for the Mobile Messaging Application, aligning with the project's use of Flutter for client-side logic and Supabase for backend services. It effectively meets basic NFRs through serverless auto-scaling but struggles with extreme scalability spikes and availability during partial failures, as components are tightly coupled (e.g., real-time chat impacts overall if failed).

**Microservices**, while not implemented, would offer better isolation (e.g., separate services for Chat, Recommendation, and Profile from Lab 4's decomposition), enabling targeted scaling and higher fault tolerance via tools like Kubernetes and message brokers. This comes at the cost of increased complexity, potentially overkill for the app's scope. Overall, Monolithic is preferable for rapid prototyping, while Microservices suit growth to enterprise levels.

## 5 CONCLUSION

This lab report completes the documentation for Lab 8 by presenting the **UML Deployment Diagram** and simplified **ATAM** analysis for the **Mobile Messaging Application**. The monolithic architecture demonstrates adequate support for **Scalability** and **Availability** through serverless features, with clear trade-offs compared to **Microservices**. Future enhancements could explore hybrid approaches to address identified limitations.