# BILKENT UNIVERSITY
# COMPUTER SCIENCE DEPARTMENT
# CS 319 OBJECT-ORIENTED SOFTWARE ENGINEERING
# DELIVERABLE 3 ITERATION 2
# SPRING 2025
# GROUP 2 SECTION 1
# 05.05.2025

| Full Name | ID |
|---|---|
| Perhat Amanlyyev | 22201007 |
| Emiralp İlgen | 22203114 |
| Anıl Keskin | 22201915 |
| İlmay Taş | 22201715 |
| Simay Uygur | 22203328 |

## Table of Contents

# 1.0 Design Goals

## 1.1 Modifiability

We explicitly selected **Modifiability** as one of our top two design goals because the system is intended to serve at least two departments that may follow different TA workflows, policies, and assignment criteria. These requirements are likely to evolve over time and may differ from one semester to another based on departmental needs or academic planning decisions. Compared to other quality attributes like performance or usability, modifiability is expected to have a greater impact on long-term adaptability and ease of reuse.

We define successful modifiability in our system by the following indicators:

- New policy implementations (e.g., workload thresholds, task types, permission changes) can be completed within 1–2 hours by editing external configuration files (YAML/JSON) or the admin interface.

- Most department-specific changes can be handled entirely through configuration, with little or no need for direct code modification.

- At least 80% of institutional policy updates can be addressed without developer involvement, by authorized admin users.

This is made possible through Spring Boot's layered architecture (controllers, services, repositories) and strict adherence to the Model–View–Controller (MVC) pattern. In this pattern:

- The Model handles the business logic and data (e.g., TA assignment rules, workload limits),

- The View is responsible for displaying information to the user (e.g., dashboards, forms),

- The Controller manages input and connects the user's actions to the system's logic.

By separating responsibilities across these three layers, we isolate business rules from infrastructure, making it easier to apply updates, add new features, or change policies without affecting unrelated components. This architectural separation is essential for maintaining flexibility and ensuring the system remains adaptable to evolving institutional needs.

## 1.2 Reliability

We chose **Reliability** as the second top design priority because our users—teaching assistants, instructors, and department administrators—depend on the system to carry out essential academic operations such as TA assignments, task approvals, and leave requests.

A single failure in these workflows can result in missed deadlines, administrative confusion, or scheduling conflicts. In academic environments, where timing and accuracy are critical, such failures are unacceptable and can negatively impact both instructional quality and administrative efficiency.

To ensure a high level of reliability, we defined the following measurable targets:

- **System uptime of 95% during business hours (08:00–17:00), measured weekly:** This limits total downtime to approximately 105 minutes per week, helping ensure system availability during key usage windows such as assignment or approval deadlines.

- **Valid responses for critical workflows within 500 ms in 95% of requests:** Fast response times are essential for smooth user interaction and timely decision-making.

- **No known data inconsistency incidents under normal operation:** All critical operations are executed as atomic database transactions. If any step fails, the operation is rolled back entirely, ensuring data consistency.

- **Automatic retry for transient faults succeeds within 3 attempts in at least 98% of cases:** Temporary disruptions such as short network delays or momentary database issues are handled internally without requiring user intervention.

These reliability goals are supported by Spring Boot's layered architecture, transactional consistency in database operations (e.g., with MySQL), structured exception handling, and standard logging and monitoring mechanisms.

Together, these strategies ensure that the system remains stable and dependable throughout each academic term, strengthening user trust and ensuring operational continuity.

# 2.0 Design Trade-Offs

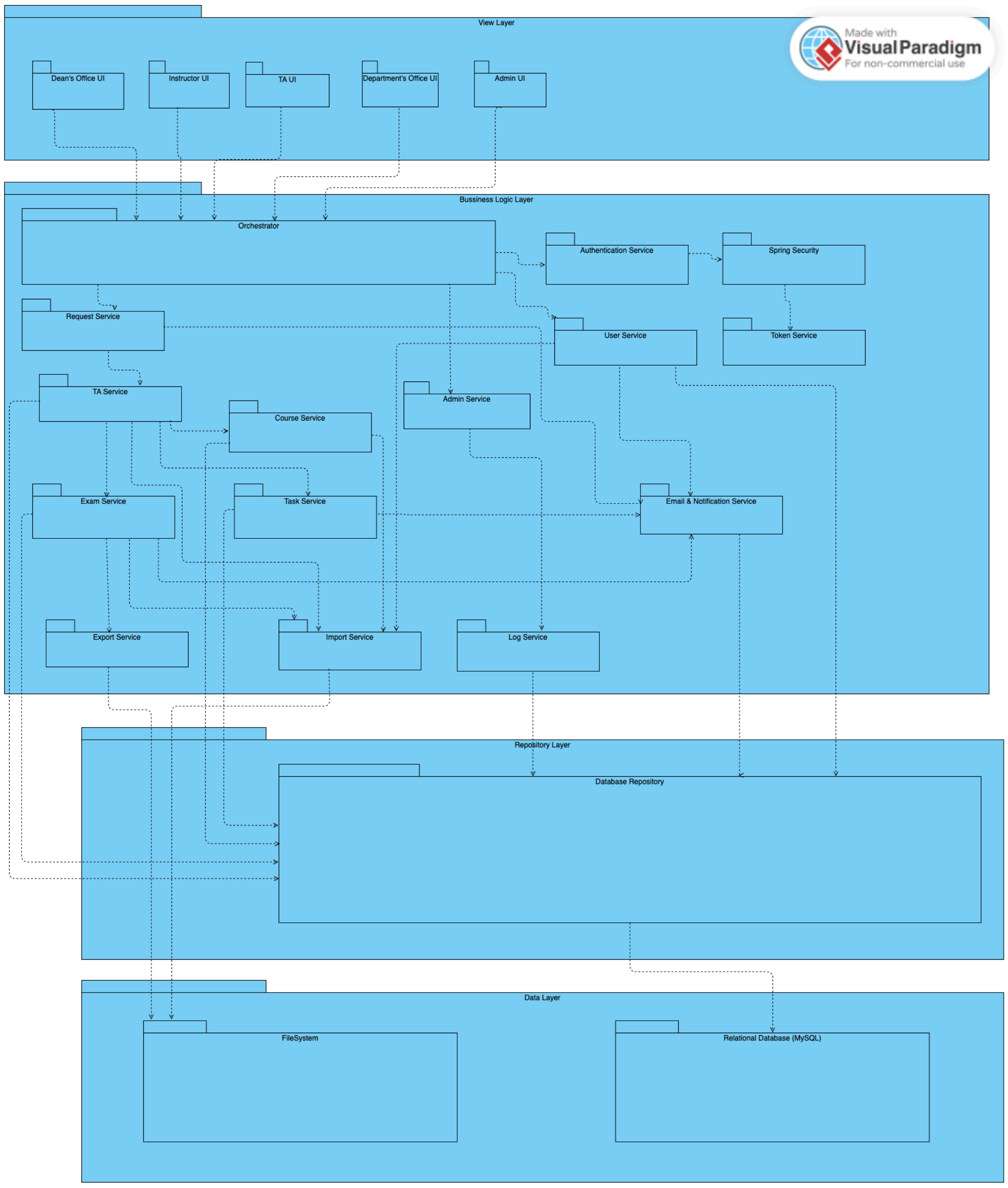
## 2.1 Modifiability vs. Performance

The modular structure that supports modifiability, such as the use of controllers, services, and repositories, introduces extra abstraction layers and well-defined interfaces. While this makes the system easier to change and extend, it also results in additional method calls and indirection, meaning components do not communicate directly but instead go through intermediary layers, for instance, a controller calling a service, which then calls a repository. This added flexibility can slightly degrade performance, especially in frequently accessed or time-sensitive parts of the system, due to the overhead of these layered interactions.

## 2.2 Reliability vs. Cost

Ensuring reliability requires robust validation, structured error handling, and conflict prevention mechanisms, such as enforcing that a task intended for a single TA is not accidentally assigned to multiple users. Achieving this level of trustworthiness often demands defensive coding, edge-case handling, and comprehensive testing, all of which increase development and maintenance effort. Additionally, if the system were to evolve toward high-availability architectures (e.g., redundant clusters, real-time replication, or long-term log retention), these safeguards would significantly raise hosting, storage, and operational costs. The overall trade-off is that while reliability enhances system robustness, it often comes at the expense of increased complexity, slower development cycles, and higher long-term costs.

## 3.0 System Decomposition Diagram



The link is: diagramLink

We follow a 4-layer architecture consisting of:

- View Layer: Handles all user-facing interfaces. It receives user input and displays data, delegating actions to the Business Logic Layer without containing any core business rules or data access code.

- Business Logic Layer: Implements the core application workflows and rules. It orchestrates services (e.g., authentication, TA assignments, scheduling) and enforces transactional integrity before passing data to repositories.

- Repository Layer: Abstracts persistence operations. It exposes data-access interfaces (e.g. Spring Data JPA repositories) for CRUD (create- read- update- delete) and custom queries, shielding the Business Logic Layer from database details.

- Data Layer: Manages physical storage of information. It comprises the relational database (MySQL) for structured entities and the filesystem for logs, import/export files, and other non-relational assets.

Each layer is responsible for a specific concern and communicates only with adjacent layers to ensure separation of responsibilities.

## 3.1 Subsystems

### 3.1.1 View Layer

This layer handles user interaction. Each interface is designed for a specific user role:

- **Instructor UI:** Instructors use this dashboard to submit preferred TA lists for their courses for any task, view assigned workloads, and approve or reject workloads.
- **Dean's Office UI:** Responsible for handling proctoring and override requests within the faculty. It provides a unified dashboard for quick decision-making, especially for urgent approvals, without exposing the underlying scheduling complexity.
- **Department Office UI:** It is to give department chairs fine-grained control over TA assignments: manual override of the restrictions, leave request handling, and weekly scheduling views. It presents conflict alerts in real-time and links directly to the scheduling workflow, so chairs can adjust assignments without navigating through unrelated menus.
- **Admin UI:** The global administration console lets system administrators manage user roles, configure system-wide settings (e.g., notification thresholds), and review audit logs. All data-management screens here enforce strict role-based access control (RBAC) checks, so only admins see sensitive operational metrics or can perform destructive actions.
- **TA UI:** Designed for teaching assistants, this view shows personal schedules, pending leave or swap requests, and proctoring assignments. It blends calendar widgets with inline forms so TAs can submit requests, such as swap, transfer requests, in just a few clicks, without having to navigate away from their dashboard.

### 3.1.2 Business Logic Layer

This layer implements all core application logic, orchestrates workflows, and enforces business rules. It exposes a clean API to the View Layer and relies on the Repository Layer for persistence.

- **Orchestrator**
  Acts as a high-level façade for complex, cross-cutting workflows (e.g. assignAllTAsAutomatically()). It sequences calls to multiple services, ensures transactional integrity, and centralizes multi-step processes so that individual services remain focused on their single responsibilities.

- **Authentication Service + Token Service + Spring Security**
  Together, these components manage the entire login and authorization lifecycle:

  - **Authentication Service** validates user credentials and applies password policies.

  - **Spring Security** hooks enforce role-based access control (e.g. TA vs. Instructor vs. Admin) across all endpoints.

  - **Token Service** issues, refreshes, and revokes JWTs, securing all subsequent requests.

- **User Service**
  Handles CRUD operations for user accounts and profiles. It validates input data, manages role assignments, and triggers notifications on key events (e.g. account creation, password change).

- **TA Service**
  Manages all TA-specific logic:

  - **Availability & Leave Tracking:** TAs set their working hours and submit leave requests, which are validated and routed through the Request Service.

  - **Eligibility Checks:** Ensures each TA meets department criteria (e.g. minimum GPA, prerequisite trainings) before assignment.

  - **Notification Triggers:** Sends alerts when schedules or assignments change.

- **Admin Service**
  Provides system-wide administrative capabilities:

  - Manage user roles and permissions via RBAC policies.

    - Configure global settings (e.g. notification thresholds, exam time-window rules).
    - Access audit logs and perform data-cleanup operations.

- **Task Service**
  Oversees task-assignment workflows:
    - **Assignment & Swap Handling:** Validates time-slots, prevents double-bookings, and coordinates swap requests. If a task is locked against swaps, the service logs TA requests to unlock it and makes it available once approved.
    - **Override Approvals:** Applies business rules for special cases (e.g. emergency reassignment).
    - Throws domain-specific exceptions on rule violations (e.g. invalid time range).
- **Exam Service**
  Handles exam-related workflows:
    - Creates and updates exam time slots.
    - Assigns proctors while enforcing non-overlap constraints.
    - Integrates with Schedule Service to visualize room and time conflicts.
- **Course Service**
  Maintains course metadata and instructor preferences:
    - Stores TA-preference lists.
    - Publishes domain events when course parameters change.
- **Request Service**
  Routes and tracks all multi-level requests (swap, override, leave):
    - Implements approval chains (TA → Dept. Chair → Dean).
    - Applies business rules at each stage and publishes notifications to involved parties.
- **Email & Notification Service**
    - Constructs and sends transactional emails (e.g. approval notifications, system alerts).

    - Uses a shared template engine.
    - Implements retry logic for transient failures.
    - Drives in-app, real-time updates via WebSockets or Server-Sent Events.
    - Delivers alerts for task changes, approvals, and system announcements.
    - Provides filtering by category and marks unread items.
- **Log Service**
  Captures every significant user or system action (e.g. task approvals, bulk imports, request submissions):
    - Writes immutable records to a centralized audit store.
    - Supports compliance reporting and forensic analysis.
- **Import Service & Export Service**
  Manage bulk data operations with robust feedback:
    - **Parsing & Validation:** Reads Excel files, validates against schema rules, and highlights inline errors.
    - **Transformation & Persistence:** Applies data mappings, writes valid records in transactions, and rolls back on partial failures.
    - **Progress Events:** Emits real-time status updates so the UI can display success/failure counts and detailed error messages.

### 3.1.3 Repository Layer

- **Database Repository:**Contains Spring Data JPA interfaces (e.g. TARepository, ScheduleRepository, TaskRepository) with support for paging, sorting, and custom queries.

### 3.1.4 Data Layer

- **Filesystem**
  Stores non-relational data such as logs, import/export files, and backups.
- **Relational Database (MySQL)**
  The primary data store for all structured entities including users, schedules, tasks, exams, and requests.