SNULife: Student Academic Community Portal

Simplifying batch-level academic management

Suryavedha Pradhan, Vishnu Vardhan Chundu, Shreyas Achal

Abstract

This project presents a centralized academic web platform developed to streamline communication, resource access, and request resolution between the students and Class Representatives (CRs) at Shiv Nadar University. The system integrates essential academic services into a structured interface including a ticket-based **Request & Issue Tracker**, a **Community Notice Board** for announcements and deadlines, a **Course Resource Repository**, and a dynamic **Timetable & Calendar system**.

Backed by a fully normalized relational database in PostgreSQL (via **Supabase**), the portal supports real-time updates, access control, and performance monitoring. Advanced SQL features like triggers and procedures enforce data consistency and automate CR activity tracking, request handling, and calendar updates. The platform not only improves transparency and accountability in student governance but also empowers CRs to serve their batch efficiently through data-backed decisions and seamless academic planning.

Contents

1 Introduction

The University Academic Portal is a centralized, web-based platform designed to streamline class-level coordination, communication, and academic planning for undergraduate students and their Class Representatives (CRs). By bringing request channels, announcements, resource repositories, and calendar reminders into one system, the portal replaces scattered DMs and untracked emails with a structured, accountable workflow—ultimately improving transparency, efficiency, and student engagement.

2 Objectives

1. **Centralize Communication** – Provide a single "Batch Page" where CRs and students can interact, eliminating the fragmentation of group chats and personal messages.
2. **Formalize Issue Tracking** – Enable students to submit academic concerns (attendance, assignments, scheduling, etc.) via a ticketing system that logs, prioritizes, and traces each request.
3. **Empowering CRs** – Give CRs tools to manage, update, and resolve tickets; post announcements; and distribute resources, while tracking their own performance metrics.
4. **Ensure Accountability** – Automatically escalate overdue issues, log resolution remarks, and maintain timestamped histories to ensure no request falls through the cracks.
5. **Consolidate Academic Resources** – Host course-wise notes, past papers, and other materials in a structured repository along with faculty and teaching assistant contact details and office information.
6. **Promote Planning & Reminders** – Auto-populate a batch calendar with upcoming deadlines—quizzes, assignments, projects—so students never miss important dates.
7. **Track Engagement & Metrics** – Generate real-time dashboards on CR responsiveness, post upvotes, departmental ticket loads, and resource counts to drive continuous improvement.

---

3 Key Features Modules

3.1 Request and Issue Tracker

- **Ticket Submission:** Students log issues through a form that creates a unique ticket ID, captures description, category, and priority.
- **CR Management:** CRs view all open tickets for their batch, update status (Created, Acknowledged, Ongoing, Resolved), add resolution remarks, and close tickets.
- **Ticket Escalation:** If a ticket remains unresolved beyond its type-specific deadline or if its resolution is unsatisfactory to the student, it may be escalated to the designated authorities.
- **Notifications & Deadlines:** Students and CRs receive in-app alerts when a ticket nears its deadline (<24 hours) or changes status.

3.2 Community Page

- **Announcements & Posts:** CRs publish course announcements, event notices, and study tips. Posts support rich text and attachments.
- **Engagement Analytics:** Announcement reach (read counts) and votes for polls are tracked for each post. CRs use these metrics to gauge batch sentiment.

3.3 Course Repository

- **Upload & Verification:** The CR can upload notes, lab manuals, and past papers under each course code (UCC).
- **Resource Stats:** Course-level dashboards display total resources and last-updated timestamps, ensuring CRs and students see the freshest materials.

3.4 Academic Calendar & Timetable

- **Automatic Calendar Population:** Every new assignment or quiz deadline added by a CR through the "Course Deadlines" interface is instantly reflected in the batch calendar.
- **Batch-Wide View:** Students browse a sortable list of upcoming events, filter by course or event type, and can view deadline reminders.
- **Elective-Aware Scheduling:** CRs can view the course enrollment data of all students in their batch, enabling them to identify free time slots to effectively reschedule classes or accommadate extra sessions without creating clashes, especially when electives differ between students.

3.5 Performance & Activity Metrics

- **CR Performance:** Tracks total tickets handled, average resolution time, and resolved-vs-unresolved counts. Auto-updates on every ticket action.
- **CR Activity Monitor:** Logs response times and status-change counts to help CRs self-evaluate and share accountability reports with the heads of the Student Council.
- **Department & Course Statistics:** Summarizes open/resolved tickets by department, resource counts by course, and overall student participation rates.

---

4 ER & Relational Models

4.1 ER Models

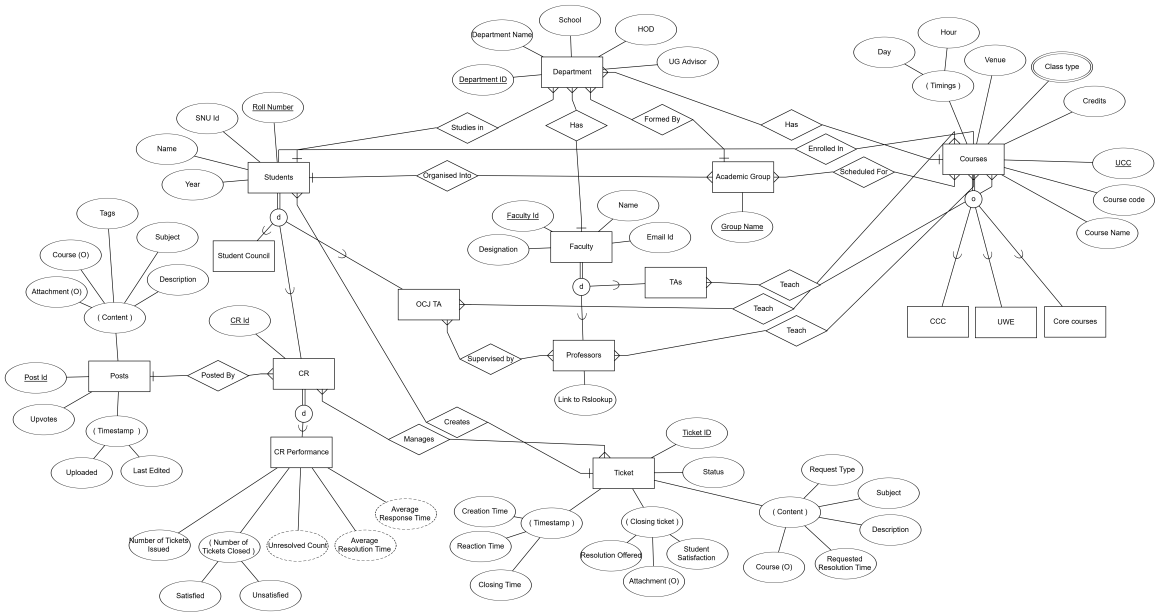

Figure 1: ER Model of Webstie

4.2 Relational Model

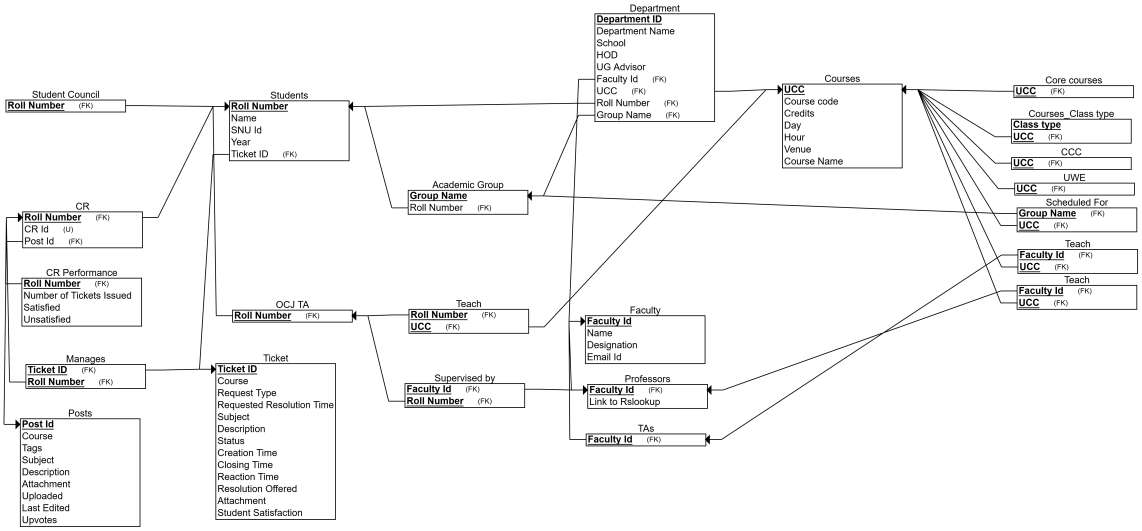

Figure 2: Relational Model

5 SQL Implementation & Database Setup

This script initializes the complete relational schema for the academic portal. It includes table creation, constraints, and foreign key references.

Execute commands in the given order to ensure referential integrity

5.1 Normalised Relations

DATABASE TABLES (NORMALIZED STRUCTURE)

Department Table

```
CREATE TABLE Department (
  Department_ID VARCHAR(10) PRIMARY KEY,
  Department_Name VARCHAR(100) UNIQUE,
  School VARCHAR(100)
);
```

HOD_Dept Table

```
CREATE TABLE HOD_Dept (
  HOD VARCHAR(20) PRIMARY KEY,
  Department_ID VARCHAR(10),
  FOREIGN KEY (Department_ID) REFERENCES Department(Department_ID)
);
```

UG_Advisor_Dept Table

```
CREATE TABLE UG_Advisor_Dept (
  UG_Advisor VARCHAR(20) PRIMARY KEY,
  Department_ID VARCHAR(10),
  FOREIGN KEY (Department_ID) REFERENCES Department(Department_ID)
);
```

Faculty Table

```
CREATE TABLE Faculty (
  Employee_Id VARCHAR(20) PRIMARY KEY,
  Email VARCHAR(100) UNIQUE,
  Department_ID VARCHAR(10),
  FOREIGN KEY (Department_ID) REFERENCES Department(Department_ID)
);
```

FacultyNames Table

```
CREATE TABLE FacultyNames (
  Name VARCHAR(100),
  Department_ID VARCHAR(10),
  Employee_Id VARCHAR(20),
  PRIMARY KEY (Name, Department_ID),
  FOREIGN KEY (Employee_Id) REFERENCES Faculty(Employee_Id)
);
```

FacultyRoles Table

```
CREATE TABLE FacultyRoles (
  Designation VARCHAR(50),
  Department_ID VARCHAR(10),
  Employee_Id VARCHAR(20),
  PRIMARY KEY (Designation, Department_ID),
  FOREIGN KEY (Employee_Id) REFERENCES Faculty(Employee_Id)
);
```

ClassVenues Table

```
CREATE TABLE ClassVenues (
  Class_Type VARCHAR(20) PRIMARY KEY,
  Venue VARCHAR(50)
);
```

CourseDept Table

```
CREATE TABLE CourseDept (
  Course_Code VARCHAR(20),
  Department_ID VARCHAR(10),
  PRIMARY KEY (Course_Code, Department_ID),
  FOREIGN KEY (Department_ID) REFERENCES Department(Department_ID)
);
```

Courses Table

```
CREATE TABLE Courses (
  UCC VARCHAR(20) PRIMARY KEY,
  Course_Code VARCHAR(20),
  Credits INT,
  Day VARCHAR(10),
  Hour TIME,
  Class_Type VARCHAR(20),
  FOREIGN KEY (Class_Type) REFERENCES ClassVenues(Class_Type),
  FOREIGN KEY (Course_Code) REFERENCES CourseDept(Course_Code)
);
```

GroupAssignment Table

```
CREATE TABLE GroupAssignment (
  Group_Number VARCHAR(10),
  Year INT,
  Department_ID VARCHAR(10),
  PRIMARY KEY (Group_Number, Year),
  FOREIGN KEY (Department_ID) REFERENCES Department(Department_ID)
);
```

Students Table

```
CREATE TABLE Students (
  roll_no VARCHAR(20) PRIMARY KEY,
  Name VARCHAR(100),
  Student_Id VARCHAR(20) UNIQUE,
  Year INT,
  Group_Number VARCHAR(10),
  Department_ID VARCHAR(10),
  FOREIGN KEY (Department_ID) REFERENCES Department(Department_ID),
  FOREIGN KEY (Group_Number, Year) REFERENCES GroupAssignment(Group_Number,
Year)
);
```

Academic_Groups Table

```
CREATE TABLE Academic_Groups (
  Group_Number VARCHAR(10) PRIMARY KEY,
  UCC VARCHAR(20),
  FOREIGN KEY (UCC) REFERENCES Courses(UCC)
);
```

CR Table

```
CREATE TABLE CR (
  roll_no VARCHAR(20) PRIMARY KEY,
  active BOOLEAN DEFAULT TRUE,
  FOREIGN KEY (roll_no) REFERENCES Students(roll_no)
);
```

ResolutionSummary Table

```
CREATE TABLE ResolutionSummary (
  Resolved_Count INT PRIMARY KEY,
  Unresolved_Count INT
);
```

CR_Performance Table

```
CREATE TABLE CR_Performance (
  roll_no VARCHAR(20) PRIMARY KEY,
  Total_Request INT DEFAULT 0,
  Avg_Resolution_Time INT DEFAULT 0,
  Resolved_Count INT DEFAULT 0,
  Last_Updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (roll_no) REFERENCES CR(roll_no)
);
```

IssueTypeInfo Table

```
CREATE TABLE IssueTypeInfo (
  Issue_Type VARCHAR(50) PRIMARY KEY,
  Deadline INT,
  Escalated_to VARCHAR(20),
  FOREIGN KEY (Escalated_to) REFERENCES Faculty(Employee_Id)
);
```

Posts Table

```
CREATE TABLE Posts (
  Id INT AUTO_INCREMENT PRIMARY KEY,
  Title VARCHAR(200),
  Body TEXT,
  Created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  Upvotes INT DEFAULT 0,
  roll_no VARCHAR(20),
  Type VARCHAR(50) DEFAULT 'General',
  FOREIGN KEY (roll_no) REFERENCES CR(roll_no)
);
```

Tickets Table

```
CREATE TABLE Tickets (
  Ticket_ID INT AUTO_INCREMENT PRIMARY KEY,
  Description TEXT,
  Status VARCHAR(20) DEFAULT 'Created',
  Created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  Updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  Resolution_Remarks TEXT,
  Issue_Type VARCHAR(50),
  roll_no VARCHAR(20),
  Escalated_to VARCHAR(20) NULL,
  Priority VARCHAR(20) DEFAULT 'Medium',
  FOREIGN KEY (Issue_Type) REFERENCES IssueTypeInfo(Issue_Type),
  FOREIGN KEY (roll_no) REFERENCES CR(roll_no),
  FOREIGN KEY (Escalated_to) REFERENCES Faculty(Employee_Id)
);
```

CourseInstructor Table

```
CREATE TABLE CourseInstructor (
  UCC VARCHAR(20) PRIMARY KEY,
  Employee_Id VARCHAR(20),
  FOREIGN KEY (UCC) REFERENCES Courses(UCC),
  FOREIGN KEY (Employee_Id) REFERENCES Faculty(Employee_Id)
);
```

TicketCR Table

```
CREATE TABLE TicketCR (
  Ticket_ID INT PRIMARY KEY,
  roll_no VARCHAR(20),
  FOREIGN KEY (Ticket_ID) REFERENCES Tickets(Ticket_ID),
  FOREIGN KEY (roll_no) REFERENCES CR(roll_no)
);
```

5.2 Additional Tables

ADDITIONAL TABLES FOR EXTENDED FUNCTIONALITY

DepartmentStats Table

```
CREATE TABLE DepartmentStats (
  Department_ID VARCHAR(10) PRIMARY KEY,
  Open_Tickets INT DEFAULT 0,
  Resolved_Tickets INT DEFAULT 0,
  Last_Updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (Department_ID) REFERENCES Department(Department_ID)
);
```

Notifications Table

```
CREATE TABLE Notifications (
  Notification_ID INT AUTO_INCREMENT PRIMARY KEY,
  roll_no VARCHAR(20) NULL,
  Employee_Id VARCHAR(20) NULL,
  Message TEXT NOT NULL,
  Type VARCHAR(50) NOT NULL,
  Related_ID VARCHAR(50) NULL,
  Created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  Read_Status BOOLEAN DEFAULT FALSE,
  FOREIGN KEY (roll_no) REFERENCES Students(roll_no),
  FOREIGN KEY (Employee_Id) REFERENCES Faculty(Employee_Id)
);
```

CourseResources Table

```
CREATE TABLE CourseResources (
  Resource_ID INT AUTO_INCREMENT PRIMARY KEY,
  UCC VARCHAR(20) NOT NULL,
  Title VARCHAR(200) NOT NULL,
  Description TEXT,
  File_Path VARCHAR(255) NOT NULL,
  Resource_Type VARCHAR(50) NOT NULL,
  Uploaded_By VARCHAR(20) NOT NULL,
  Upload_Date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  Verification_Status VARCHAR(20) DEFAULT 'Pending',
  Verifier_ID VARCHAR(20) NULL,
  Verification_Deadline DATE NULL,
  FOREIGN KEY (UCC) REFERENCES Courses(UCC),
  FOREIGN KEY (Uploaded_By) REFERENCES Students(roll_no),
  FOREIGN KEY (Verifier_ID) REFERENCES Faculty(Employee_Id)
);
```

CourseStats Table

```
CREATE TABLE CourseStats (
  UCC VARCHAR(20) PRIMARY KEY,
  Resource_Count INT DEFAULT 0,
  Last_Updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (UCC) REFERENCES Courses(UCC)
);
```

CourseDeadlines Table

```
CREATE TABLE CourseDeadlines (
  Deadline_ID INT AUTO_INCREMENT PRIMARY KEY,
  UCC VARCHAR(20) NOT NULL,
  Deadline_Type VARCHAR(50) NOT NULL,
  Deadline_Date DATE NOT NULL,
  Description TEXT,
  Created_By VARCHAR(20) NOT NULL,
  FOREIGN KEY (UCC) REFERENCES Courses(UCC),
  FOREIGN KEY (Created_By) REFERENCES CR(roll_no)
);
```

BatchCalendar Table

```
CREATE TABLE BatchCalendar (
  Event_ID INT AUTO_INCREMENT PRIMARY KEY,
  Event_Date DATE NOT NULL,
  Event_Name VARCHAR(200) NOT NULL,
  Event_Type VARCHAR(50) NOT NULL,
  Course_Code VARCHAR(20) NULL,
  Description TEXT,
  UNIQUE KEY (Event_Date, Event_Name)
);
```

CRActivityMetrics Table

```
CREATE TABLE CRActivityMetrics (
  roll_no VARCHAR(20) PRIMARY KEY,
  Last_Active TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  Avg_Response_Time INT DEFAULT 0,
  Status_Changes INT DEFAULT 0,
  FOREIGN KEY (roll_no) REFERENCES CR(roll_no)
);
```

PostEngagementStats Table

```
CREATE TABLE PostEngagementStats (
  Id INT PRIMARY KEY,
  Total_Upvotes INT DEFAULT 0,
  Engagement_Score DECIMAL(10,2) DEFAULT 0.0,
  Last_Updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (Id) REFERENCES Posts(Id)
);
```

StudentCourses Table

```
CREATE TABLE StudentCourses (
  roll_no VARCHAR(20),
  UCC VARCHAR(20),
  Registration_Date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  Status VARCHAR(20) DEFAULT 'Active',
  PRIMARY KEY (roll_no, UCC),
  FOREIGN KEY (roll_no) REFERENCES Students(roll_no),
  FOREIGN KEY (UCC) REFERENCES Courses(UCC)
);
```

StudentParticipation Table

```
CREATE TABLE StudentParticipation (
  roll_no VARCHAR(20) PRIMARY KEY,
  Issues_Raised INT DEFAULT 0,
  Last_Activity TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (roll_no) REFERENCES Students(roll_no)
);
```

CRRatings Table

```
CREATE TABLE CRRatings (
  roll_no VARCHAR(20) PRIMARY KEY,
  Efficiency_Score DECIMAL(5,2) DEFAULT 0.0,
  Response_Rate DECIMAL(5,2) DEFAULT 0.0,
  Last_Updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (roll_no) REFERENCES CR(roll_no)
);
```

AnnouncementReach Table

```
CREATE TABLE AnnouncementReach (
  Post_Id INT PRIMARY KEY,
  Student_Count INT DEFAULT 0,
  Read_Count INT DEFAULT 0,
  Last_Updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (Post_Id) REFERENCES Posts(Id)
);
```

5.3 INDEXES

CREATE INDEXES FOR PERFORMANCE OPTIMIZATION

SQL Index Creation

```
-- Indexes for frequently joined columns
CREATE INDEX idx_student_dept_year ON Students(Department_ID, Year);
CREATE INDEX idx_ticket_status ON Tickets(Status);
CREATE INDEX idx_ticket_roll_no ON Tickets(roll_no);
CREATE INDEX idx_ticket_issue_type ON Tickets(Issue_Type);
CREATE INDEX idx_course_day_hour ON Courses(Day, Hour);
CREATE INDEX idx_course_code ON Courses(Course_Code);
CREATE INDEX idx_faculty_dept ON Faculty(Department_ID);
CREATE INDEX idx_post_created_at ON Posts(Created_at);
CREATE INDEX idx_group_dept ON GroupAssignment(Department_ID);
```

6 Triggers

TRIGGERS FOR DATA INTEGRITY & AUTOMATION

1. Update CR Performance Metrics

When: AFTER INSERT OR UPDATE on Tickets
Why: Keeps the `CR_Performance` table up to date by recalculating, for the affected CR, their total number of tickets, number of resolved tickets, average resolution time, and timestamp of last update. If no performance record exists yet for that CR, it creates one.

update_cr_performance Trigger

```
DELIMITER //
CREATE TRIGGER update_cr_performance
AFTER INSERT OR UPDATE ON Tickets
FOR EACH ROW
BEGIN
  -- Update total request count
  UPDATE CR_Performance
    SET Total_Request = (
      SELECT COUNT(*) FROM Tickets WHERE roll_no = NEW.roll_no
    ),
    Resolved_Count = (
      SELECT COUNT(*) FROM Tickets
      WHERE roll_no = NEW.roll_no AND Status = 'Resolved'
    ),
    Avg_Resolution_Time = (
      SELECT IFNULL(AVG(TIMESTAMPDIFF(HOUR, Created_at, Updated_at)), 0)
      FROM Tickets
      WHERE roll_no = NEW.roll_no AND Status = 'Resolved'
    ),
    Last_Updated = NOW()
    WHERE roll_no = NEW.roll_no;

  -- If CR_Performance record doesn't exist yet, create it
  IF ROW_COUNT() = 0 THEN
    INSERT INTO CR_Performance (roll_no, Total_Request, Resolved_Count,
Avg_Resolution_Time, Last_Updated)
    VALUES (
      NEW.roll_no,
      (SELECT COUNT(*) FROM Tickets WHERE roll_no = NEW.roll_no),
      (SELECT COUNT(*) FROM Tickets WHERE roll_no = NEW.roll_no AND Status
= 'Resolved'),
      (SELECT IFNULL(AVG(TIMESTAMPDIFF(HOUR, Created_at, Updated_at)), 0)
        FROM Tickets
        WHERE roll_no = NEW.roll_no AND Status = 'Resolved'),
      NOW()
    );
  END IF;
END//
DELIMITER ;
```

2. Synchronize ResolutionSummary Table

When: AFTER UPDATE on CR_Performance
Why: Synchronizes the global `ResolutionSummary` table so it always reflects the latest total of resolved versus unresolved tickets across all CRs.

update_resolution_summary Trigger

```
DELIMITER //
CREATE TRIGGER update_resolution_summary
AFTER UPDATE ON CR_Performance
FOR EACH ROW
BEGIN
  -- Calculate unresolved count based on total and resolved
  DECLARE unresolved INT;
  SET unresolved = NEW.Total_Request - NEW.Resolved_Count;

  -- Insert or update in ResolutionSummary
  INSERT INTO ResolutionSummary (Resolved_Count, Unresolved_Count)
    VALUES (NEW.Resolved_Count, unresolved)
    ON DUPLICATE KEY UPDATE Unresolved_Count = unresolved;
END//
DELIMITER ;
```

3. Enforce Department Consistency in GroupAssignment

When: BEFORE INSERT OR UPDATE on GroupAssignment
Why: Ensures that a given group number remains tied to the same department across different years—prevents assigning the same group number to different departments.

enforce_group_department Trigger

```
DELIMITER //
CREATE TRIGGER enforce_group_department
BEFORE INSERT OR UPDATE ON GroupAssignment
FOR EACH ROW
BEGIN
  -- Check if this Group_Number already exists with a different
Department_ID
  DECLARE existing_dept VARCHAR(10);

  SELECT Department_ID INTO existing_dept
    FROM GroupAssignment
    WHERE Group_Number = NEW.Group_Number AND Year != NEW.Year
    LIMIT 1;

  -- If exists with different dept, prevent the operation
  IF existing_dept IS NOT NULL AND existing_dept != NEW.Department_ID THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Group Number must be consistent with Department
across years';
  END IF;
END//
DELIMITER ;
```

4. Synchronize TicketCR Table with Tickets

When: AFTER INSERT on Tickets
Why: Auto-inserts a corresponding row into `TicketCR` so that every ticket is linked in that lookup table without requiring manual insertion.

sync_ticket_cr Trigger

```
DELIMITER //
CREATE TRIGGER sync_ticket_cr
AFTER INSERT ON Tickets
FOR EACH ROW
BEGIN
  -- Create corresponding entry in TicketCR
  INSERT INTO TicketCR (Ticket_ID, roll_no)
    VALUES (NEW.Ticket_ID, NEW.roll_no);
END//
DELIMITER ;
```

5. Enforce Course Department Consistency

When: BEFORE INSERT OR UPDATE on CourseDept
Why: Validates that the department prefix in the course code (e.g., "CSD" in "CSD317") matches the `Department_ID`, preventing mismatches.

enforce_course_department Trigger

```
DELIMITER //
CREATE TRIGGER enforce_course_department
BEFORE INSERT OR UPDATE ON CourseDept
FOR EACH ROW
BEGIN
  -- Extract department prefix from course code (assuming format like
"CS317")
  DECLARE dept_prefix VARCHAR(10);
  SET dept_prefix = SUBSTRING(NEW.Course_Code, 1, 2);

  -- Check if department prefix matches department ID (assuming
department IDs like "CS")
  IF dept_prefix != NEW.Department_ID THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Course code prefix must match department ID';
  END IF;
END//
DELIMITER ;
```

6. Verify Faculty Department Before Course Assignment

When: BEFORE INSERT OR UPDATE on CourseInstructor
Why: Confirms the assigned faculty member belongs to the same department as the course they're being assigned to.

verify_faculty_department Trigger

```
DELIMITER //
CREATE TRIGGER verify_faculty_department
BEFORE INSERT OR UPDATE ON CourseInstructor
FOR EACH ROW
BEGIN
  DECLARE course_dept VARCHAR(10);
  DECLARE faculty_dept VARCHAR(10);

  -- Get course department
  SELECT cd.Department_ID INTO course_dept
    FROM CourseDept cd
    JOIN Courses c ON cd.Course_Code = c.Course_Code
    WHERE c.UCC = NEW.UCC
    LIMIT 1;

  -- Get faculty department
  SELECT Department_ID INTO faculty_dept
    FROM Faculty
    WHERE Employee_Id = NEW.Employee_Id;

  -- Check match
  IF course_dept != faculty_dept THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Faculty must belong to the course department';
  END IF;
END//
DELIMITER ;
```

7. Validate CR Role Before Ticket Assignment

When: BEFORE INSERT on Tickets
Why: Enforces that only active Class Representatives can be assigned as the roll_no on a ticket—other students cannot create or own tickets.

validate_cr_role Trigger

```
DELIMITER //
CREATE TRIGGER validate_cr_role
BEFORE INSERT ON Tickets
FOR EACH ROW
BEGIN
  -- Ensure only CRs can create tickets
  IF NOT EXISTS (SELECT 1 FROM CR WHERE roll_no = NEW.roll_no) THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Only Class Representatives can handle tickets';
  END IF;
END//
DELIMITER ;
```

8. Update Ticket Timestamps

When: BEFORE UPDATE on Tickets
Why: Ensures that any status change to "Resolved" is accompanied by non-null resolution remarks; otherwise it aborts the update.

update_ticket_timestamps Trigger

```
DELIMITER //
CREATE TRIGGER update_ticket_timestamps
BEFORE UPDATE ON Tickets
FOR EACH ROW
BEGIN
  -- Require resolution remarks if marking as resolved
  IF NEW.Status = 'Resolved' AND OLD.Status != 'Resolved' AND
NEW.Resolution_Remarks IS NULL THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Resolution remarks required when resolving a
ticket';
  END IF;
END//
DELIMITER ;
```

9. Auto-Update Ticket Statistics by Department

When: AFTER INSERT OR UPDATE on Tickets
Why: Keeps each department's row in `DepartmentStats` current by counting its open vs. resolved tickets and updating the timestamp.

update_department_ticket_stats Trigger

```
DELIMITER //
CREATE TRIGGER update_department_ticket_stats
AFTER INSERT OR UPDATE ON Tickets
FOR EACH ROW
BEGIN
  DECLARE dept_id VARCHAR(10);

  -- Find department
  SELECT Department_ID INTO dept_id
    FROM Students
    WHERE roll_no = NEW.roll_no;

  -- Update statistics
  INSERT INTO DepartmentStats (Department_ID, Open_Tickets,
Resolved_Tickets, Last_Updated)
    VALUES (dept_id,
      (SELECT COUNT(*) FROM Tickets t JOIN Students s ON t.roll_no =
s.roll_no
        WHERE s.Department_ID = dept_id AND t.Status != 'Resolved'),
      (SELECT COUNT(*) FROM Tickets t JOIN Students s ON t.roll_no =
s.roll_no
        WHERE s.Department_ID = dept_id AND t.Status = 'Resolved'),
      NOW())
    ON DUPLICATE KEY UPDATE
      Open_Tickets = (SELECT COUNT(*) FROM Tickets t JOIN Students s ON
t.roll_no = s.roll_no
        WHERE s.Department_ID = dept_id AND t.Status != 'Resolved'),
      Resolved_Tickets = (SELECT COUNT(*) FROM Tickets t JOIN Students s
ON t.roll_no = s.roll_no
        WHERE s.Department_ID = dept_id AND t.Status = 'Resolved'),
      Last_Updated = NOW();
END//
DELIMITER ;
```

10. Academic Resource Count Tracker

When: AFTER INSERT OR DELETE on CourseResources
Why: Maintains `CourseStats.Resource_Count` automatically, incrementing on uploads and decrementing on deletions (never going below zero).

update_course_resource_count Trigger

```
DELIMITER //
CREATE TRIGGER update_course_resource_count
AFTER INSERT OR DELETE ON CourseResources
FOR EACH ROW
BEGIN
  IF (TG_OP = 'INSERT') THEN
    UPDATE CourseStats
      SET Resource_Count = Resource_Count + 1,
        Last_Updated = NOW()
      WHERE UCC = NEW.UCC;

    IF ROW_COUNT() = 0 THEN
      INSERT INTO CourseStats (UCC, Resource_Count, Last_Updated)
        VALUES (NEW.UCC, 1, NOW());
    END IF;

  ELSEIF (TG_OP = 'DELETE') THEN
    UPDATE CourseStats
      SET Resource_Count = GREATEST(Resource_Count - 1, 0),
        Last_Updated = NOW()
      WHERE UCC = OLD.UCC;
  END IF;
END//
DELIMITER ;
```

11. Academic Calendar Sync

When: AFTER INSERT OR UPDATE on CourseDeadlines
Why: Syncs course deadlines with the batch calendar, auto-creating or updating relevant entries to reflect upcoming evaluations.

update_academic_calendar Trigger

```
DELIMITER //
CREATE TRIGGER update_academic_calendar
AFTER INSERT OR UPDATE ON CourseDeadlines
FOR EACH ROW
BEGIN
  DECLARE course_code VARCHAR(20);

  SELECT Course_Code INTO course_code
    FROM Courses
    WHERE UCC = NEW.UCC;

  INSERT INTO BatchCalendar (Event_Date, Event_Name, Event_Type,
Course_Code, Description)
    VALUES (NEW.Deadline_Date,
      CONCAT(NEW.Deadline_Type, ' - ', course_code),
      NEW.Deadline_Type,
      course_code,
      NEW.Description)
    ON DUPLICATE KEY UPDATE
      Event_Name = CONCAT(NEW.Deadline_Type, ' - ', course_code),
      Description = NEW.Description;
END//
DELIMITER ;
```

12. CR Responsiveness Tracker

When: AFTER UPDATE on Tickets
Why: Tracks how promptly CRs respond to tickets by logging status changes, average response time, and last activity timestamp.

monitor_cr_activity Trigger

```
DELIMITER //
CREATE TRIGGER monitor_cr_activity
AFTER UPDATE ON Tickets
FOR EACH ROW
BEGIN
  IF NEW.Status != OLD.Status THEN
    DECLARE response_time INT;
    SET response_time = TIMESTAMPDIFF(HOUR, NEW.Created_at, NOW());

    INSERT INTO CRActivityMetrics (roll_no, Last_Active,
Avg_Response_Time, Status_Changes)
      VALUES (NEW.roll_no, NOW(), response_time, 1)
      ON DUPLICATE KEY UPDATE
        Last_Active = NOW(),
        Avg_Response_Time = ((Avg_Response_Time * Status_Changes) +
response_time) / (Status_Changes + 1),
        Status_Changes = Status_Changes + 1;
  END IF;
END//
DELIMITER ;
```

13. Course Registration Validator

When: BEFORE INSERT on StudentCourses
Why: Ensures students can only register for courses within their academic group and checks for schedule clashes in enrolled courses.

validate_course_registration Trigger

```
DELIMITER //
CREATE TRIGGER validate_course_registration
BEFORE INSERT ON StudentCourses
FOR EACH ROW
BEGIN
  DECLARE student_group VARCHAR(10);
  DECLARE course_group VARCHAR(10);

  SELECT Group_Number INTO student_group
    FROM Students
    WHERE roll_no = NEW.roll_no;

  SELECT Group_Number INTO course_group
    FROM Academic_Groups
    WHERE UCC = NEW.UCC;

  IF student_group != course_group AND course_group IS NOT NULL THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Student not in correct academic group for this
course';
  END IF;

  IF EXISTS (
    SELECT 1
    FROM StudentCourses sc
      JOIN Courses c1 ON sc.UCC = c1.UCC
      JOIN Courses c2 ON NEW.UCC = c2.UCC
    WHERE sc.roll_no = NEW.roll_no
      AND c1.Day = c2.Day
      AND c1.Hour = c2.Hour
  ) THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Schedule conflict detected';
  END IF;
END//
DELIMITER ;
```

14. Student Participation Logger

When: AFTER INSERT on Tickets
Why: Tracks how often students participate in the system by raising tickets, updating their issue count and latest activity time.

update_student_participation Trigger

```
DELIMITER //
CREATE TRIGGER update_student_participation
AFTER INSERT ON Tickets
FOR EACH ROW
BEGIN
  INSERT INTO StudentParticipation (roll_no, Issues_Raised, Last_Activity)
    VALUES (NEW.roll_no, 1, NOW())
    ON DUPLICATE KEY UPDATE
      Issues_Raised = Issues_Raised + 1,
      Last_Activity = NOW();
END//
DELIMITER ;
```

15. CR Handover Manager

When: AFTER UPDATE on CR
Why: When a CR is deactivated, this finds a replacement CR and reassigns all open tickets and CR-tied records for smooth transition.

cr_handover_process Trigger

```
DELIMITER //
CREATE TRIGGER cr_handover_process
AFTER UPDATE ON CR
FOR EACH ROW
BEGIN
  IF OLD.active = TRUE AND NEW.active = FALSE THEN
    DECLARE new_cr VARCHAR(20);

    SELECT cr.roll_no INTO new_cr
      FROM CR cr
      JOIN Students s1 ON cr.roll_no = s1.roll_no
      JOIN Students s2 ON s2.roll_no = OLD.roll_no
      WHERE cr.active = TRUE
        AND s1.Department_ID = s2.Department_ID
        AND s1.Year = s2.Year
        AND cr.roll_no != OLD.roll_no
      LIMIT 1;

    IF new_cr IS NOT NULL THEN
      UPDATE Tickets
        SET roll_no = new_cr
        WHERE roll_no = OLD.roll_no AND Status != 'Resolved';

      UPDATE TicketCR
        SET roll_no = new_cr
        WHERE roll_no = OLD.roll_no;
    END IF;
  END IF;
END//
DELIMITER ;
```

7 Conclusion

7.1 Limitations

- **Elective Enrollment:** The system does not currently integrate with the official ERP or academic records. As electives are often chosen midway through a semester, students must manually input their elective course registrations into the portal for access to relevant course resources and deadlines.
- **Cross-Batch Course Access:** Students from other batches or departments who are enrolled in a course (as an elective or otherwise) will not have access to its resources if they are not part of the owning batch's portal. To address this, a future implementation can include an elective enrollment verification module that authenticates users and grants access based on course registration rather than batch affiliation.

7.2 Future Scope

While the current implementation of SNULife focuses solely on academic coordination and utility, the ideation behind the platform has been developed with a much broader ecosystem in mind. The core concept — providing structured access, centralized communication, and visibility across student groups — is scalable to other non-academic domains within the university.

The following modules represent the natural future expansion of this project:

- **Academic Repository:** A centralized library of academic documents, policy guides, and process manuals accessible through a structured and searchable portal. This will include keyword-tagged content and a categorized "Common Topics" index for student ease.
- **Clubs and Organizations:** Dedicated pages for university clubs to present their team structure, member roles and contact information, past projects, upcoming events, and a visual gallery — making it easier for students to discover and engage with various communities.
- **Sports and SL League Integration:** A module for university sports teams and internal leagues like the SL League to showcase achievements, player information, tryout requirements, and updates. This also gives visibility to lesser-known teams and helps in institutional memory and recognition.

Each of these additions empowers student-led groups, academic or extracurricular, with digital visibility and operational autonomy, enabling them to shape and share their own vision through the portal.

References

[1] ERDPlus – Tool for ER Diagrams & Relational Schemas. https://erdplus.com/

[2] Lovable AI – Used for designing the user interface and building frontend layout code. https://www.lovable.ai/

[3] MySQL Documentation – Official reference for SQL commands, triggers, and performance tuning. https://dev.mysql.com/doc/