***

Student Support and grievance system

Chapter 1: Introduction

Project Title and Abstract

The **Student Support and Grievances System** is a web-based platform designed to help students raise issues, track their complaint status, and receive timely responses from the administration. Traditional reporting methods (paper forms, informal emails) cause delays and lost records; this centralized portal improves **transparency and speed** for grievance handling.

Background and Problem Statement

Students often face academic or administrative problems but currently have no unified way to file complaints. As noted, “there is no centralized, trackable, and transparent system for students to submit grievances and receive timely resolutions.” The lack of a formal system leads to mismanagement of complaints, slow responses, and student frustration. By **digitalizing grievance reporting**, we aim to streamline communication between students and authorities.

Objectives

Key objectives include designing and implementing an online grievance management platform. The primary objective is to develop an **intuitive system for submitting and managing student complaints**. Secondary objectives (from the project report) are:

Secure student login and profile management (role-based authentication).
* Submit grievances with detailed information and attach evidence (e.g., screenshots).
*Track grievance status** through its lifecycle (submitted, assigned, resolved).
* Provide admin and staff dashboards for reviewing and assigning complaints.
*Generate reports and analytics** on grievances for decision-making (e.g., by category, department).
*Send notifications** (email or SMS) to keep students updated on their cases.

Scope and Limitations

In-scope features include student registration/login, grievance submission and tracking, and admin/staff updates with reporting features. Out-of-scope for the current version are: developing a mobile app, voice/chat-based complaint intake, or integrating with external

portals (these could be future enhancements). Limitations include reliance on institutional email for alerts and use of **SQLite** for the prototype (which may limit concurrent connections).

Chapter 2: Requirements

Functional Requirements

The system shall support the following core functions (from the project report):

*User Management:** Students can register and log in; Admins and staff have role-based accounts.
*Grievance Submission:** Logged-in students can submit a new grievance with details (subject, description, category).
*Assignment:** Admin users can view submitted grievances and assign them to appropriate staff for resolution.
*Status Updates:** Staff users can update the status of an assigned grievance (e.g., “In Progress”, “Resolved”) and add remarks.
*Tracking:** Students can track the status of their grievances through their profile (view history of actions).
*Reporting:** Admins can generate reports summarizing grievances (e.g., by type, date, department) for management review.

Non-Functional Requirements

The system should meet the following quality attributes:

*Usability: The interface must be intuitive and user-friendly** (easy navigation, clear forms).
*Performance:** Pages and APIs should load quickly (target $<2s$ response), even with many concurrent users.
*Security: Data must be stored securely; use HTTPS and hashed passwords**. Role-based access control ensures students, staff, and admin see only appropriate data.
*Compatibility:** The web app should function correctly on modern browsers and devices.
*Scalability (future):** While prototype uses SQLite, design should allow migration to a scalable database if user load grows.

Constraints and Assumptions

We assume **reliable internet access** on campus and that users have **unique institutional email addresses**. The system will be deployed on cloud infrastructure (e.g., AWS or PythonAnywhere) for high availability. A technical constraint is the use of **SQLite** for development (as specified in the tech stack), which supports small to medium user volumes.

We assume availability of an SMTP/email service for notifications.

Use Cases / User Stories

Example user stories illustrate interactions:

*Student Submits Complaint:** As a student, I want to log in and file a new grievance so that my issue is formally recorded. (Fulfills functional requirements of login and submission.)
*Student Tracks Complaint:** As a student, I want to view the status of my grievances so that I know when they are resolved.
*Admin Reviews Complaint:** As an admin, I want to view all incoming complaints and assign each to an appropriate staff member so that issues get addressed by the right person.
*Staff Updates Status:** As a staff member, I want to update the grievance status and add resolution notes so that the student and admin are informed.
*Report Generation:** As an admin, I want to generate a report of all grievances by category and date range so that I can analyze common issues.

Requirements Traceability

Each requirement will be linked to design components and test cases. For example, the "Grievance submission" requirement maps to the Django view/controller for filing a complaint and to a unit test verifying form validation. A **traceability matrix** will ensure full coverage of requirements in design and QA.

Chapter 3: Architecture

This section describes the high-level architecture (C4 System Context and Container) and design diagrams. The system follows a **3-tier web architecture**: a presentation layer (HTML/CSS/JS client), an application server layer (**Python/Django**), and a database layer (SQLite). The core container is the Django web application, which exposes views and REST endpoints. It handles authentication, business logic (grievance workflow), and interfaces with the database. A typical deployment could use AWS EC2 (or PythonAnywhere) to host the app, with a managed SQLite or PostgreSQL database. External actors (Students, Staff, Admin) interact through a browser UI over HTTPS. (Optionally, integrations might include an email/SMS gateway for notifications.)

This context (C4 Level 1) implies components as in the container diagram:

*Web Client:** HTML/CSS/JavaScript (Bootstrap) frontend running in user's browser.
*Application Server (Django):** Handles requests, business logic, and session management.

*Database:** Relational store for users, grievances, etc. (SQLite for prototype; could be PostgreSQL in production).
*Notification Service:** A component (could be part of Django or separate) that sends emails/SMS on events (e.g., grievance update).

As shown below, the Use Case, Class, and Sequence diagrams detail the key interactions and data structures.

Figure: Use case diagram illustrating system interactions. Actors include Student, Admin, and Staff. Students can log in, submit new grievances, and track existing ones. Administrators can review and assign grievances, and staff can update resolution status. Each use case corresponds to a key requirement of the system.

Figure: Class diagram (UML) showing main entities. A Student (with name, ID) can submit multiple Grievance instances (with title, description, status, timestamps). Each grievance is handled by a User (Admin or Staff). The diagram identifies attributes (e.g., $status$, $description$) and associations (1-to-many links) between classes.

Figure: Sequence diagram for the “Submit Grievance” workflow. The Student (actor) sends a POST request via the web UI (browser) to the Django Controller. The Controller validates input and saves a new grievance record in the Database. The system then returns a confirmation to the user. This sequence ensures data consistency and provides immediate feedback.

Justification of Design Choices

We chose **Django** (a monolithic MVC framework) for **rapid development** and built-in security features. A monolithic Django app is easier to implement and deploy for this project’s scale, while still modular via apps (e.g., a grievances app, a users app). **SQLite** is used for simplicity and quick setup; it can be switched to PostgreSQL later if needed. The **3-tier architecture** separates concerns and allows us to scale each tier independently if necessary. Continuous integration/deployment will use Git (GitHub) and automated pipelines to test and deploy new code. Overall, this design balances ease of development with robustness.

Chapter 4: Detailed Design

Data Models (ER Diagram)

The core data entities and relationships are captured in the ER diagram below. Key entities include **Student**, **Grievance**, and **User** (Staff/Admin). Each student (with attributes like $studentID$, $name$, etc.) may file many grievances, so there is a one-to-many relationship from Student to Grievance. Each Grievance has fields such as $grievanceID$, $category$, $description$, $submissionDate$, and $status$, and it is linked to the submitting Student and (once assigned) to a Staff/Admin user. Lookup tables like **Category** (grievance type) and **Status** (e.g., “Pending”, “Resolved”) can also be modeled for normalization. This ER model ensures all complaint data is normalized and relational integrity is maintained.

Figure: ER diagram of core data entities. Each Student can have multiple Grievances. A Grievance record holds details (description, date, status) and is linked to the submitting Student. Staff/Admin users (of type User) are assigned to grievances. Lookup entities (not shown) include Category and Status for possible values. Primary keys (e.g., $StudentID$, $GrievanceID$) and relationships are illustrated.

Data Flow Diagrams (DFDs)

We model the system’s processes using DFD levels. At the highest context level (Level-0), external entities (Student, Admin) interact with a single “Grievance System” process. Level-1 DFD decomposes this into subprocesses like **Submit Grievance**, **Review/Assign**, **Update Status**, and **Generate Report**. Data flows connect to data stores (StudentDB, GrievanceDB). For example, when a student submits a complaint, data flows from the Student to Submit Grievance process and is stored in GrievanceDB; updates then flow back to the Student as status info. An example DFD is shown below.

Figure: Example Data Flow Diagram (DFD). This illustration (from generic sources) shows how data flows among external actors and processes. In a Level-0 context, Students and Admins exchange data with the Grievance system. In Level-1 (not shown here), processes such as “Submit Grievance” and “Update Status” handle data moving to/from the database stores.

Component Responsibilities

The system’s components have clear roles:

*Django Controllers/Views:** Handle HTTP requests, enforce business rules, and render HTML pages or JSON responses.
*Models/Repositories:** Encapsulate database access. For instance, a $GrievanceRepository$ provides methods to create, read, and update grievance records in SQLite. Django's ORM serves as this repository layer.
*Frontend (Templates):** Use Django templates and Bootstrap for UI. They present forms for complaint submission and tables/lists for tracking. Frontend scripts (JavaScript) handle dynamic elements.
*Notification Service:** A module (or Django signal handler) listens for changes and sends email/SMS alerts to students.
*Utilities:** Miscellaneous helpers (e.g., for generating PDF reports or charts) that are invoked by the application layer.

Design Patterns Used

We apply standard patterns:

*Model-View-Controller (MVC):** Django follows the MVC (or Model-Template-View) pattern, separating data models, business logic (controllers), and UI templates.
*Singleton (Configuration):** The database connection/config is a singleton within the Django app context.
*Observer/Signal: Django's signal framework (e.g., $post\_save$) can implement the Observer pattern** for notifications: when a Grievance's status changes, the signal notifies the Notification Service to send an update.
*Repository Pattern:** Our models act like repositories to abstract data access, allowing easy swapping of SQLite with another database.

Chapter 5: Implementation Plan

Agile Sprint Breakdown

The project follows an **Agile methodology** with iterative sprints. Planned sprints are:

*Sprint 1:** Requirements gathering and UI prototyping (login/register forms, grievance submission page).
*Sprint 2:** Student module (implement student dashboard, submission workflow, database models).
*Sprint 3:** Admin and Staff modules (review/assignment interface, status update, role-based access).
*Sprint 4:** Reporting and Testing (create summary reports, finalize validation, conduct

system/integration tests).
*Sprint 5:** Deployment (set up production environment, finalize documentation).

These sprints align with the timeline: first week on requirements, then iterative development and testing, leading to a deployment-ready system by week 9 (as drafted in the project schedule).

CI/CD Pipeline Description

We will use **version control (Git/GitHub)** to manage the codebase. Each commit triggers automated builds and tests via GitHub Actions (Unit tests, code linting). On the main branch, successful builds automatically deploy to a staging server (e.g., AWS EC2 or Heroku). Merged code on release branches triggers deployment to production.

The CI/CD process includes:

*Build:** Install dependencies and run static code checks.
*Test:** Execute automated tests (unit and integration).
*Deploy:** On success, the pipeline packages the Django app and pushes it to the web host.
*Rollback:** If post-deployment monitoring detects errors, the pipeline can roll back to the last stable version.

Infrastructure as Code (IaC) Plan

We will use **IaC tools (Terraform or AWS CloudFormation)** to provision infrastructure. The IaC configuration will define:

* A VPC and subnets (if using AWS).
* EC2 instance(s) or Elastic Beanstalk environment for the Django app.
* An RDS instance if migrating from SQLite to PostgreSQL.
* Security groups (firewall rules for HTTP/HTTPS and SSH).
* S3 bucket for static/media files.

This ensures that the entire deployment (servers, networks, database) is **version-controlled and reproducible**, enabling automated provisioning for test and production environments.

Chapter 6: Quality Assurance Plan

Testing Strategy

A multi-level testing approach will be used:

Unit Testing: Individual components (models, forms, utilities) will have unit tests (e.g., using Django's $TestCase$) to verify core logic.
Integration Testing: We test interactions between components (e.g., submitting a grievance creates a database record and triggers a notification).
System Testing: The entire system is tested end-to-end (e.g., student login, submit $\rightarrow$ admin assigns $\rightarrow$ student sees update) in a staging environment.
User Acceptance Testing (UAT): Representative users (students and administrators) test the system to ensure it meets requirements.

Tests will cover key scenarios: login, grievance submission, updating status, and report generation. For example, we will verify that an unregistered student cannot submit a complaint, and that a status change correctly notifies the submitter.

Monitoring and Observability Plan

In production, the system will use **logging and metrics** to ensure health. Django's logging framework will record errors and important events. We will integrate an application monitoring tool (e.g., AWS CloudWatch or Grafana) to track server CPU/RAM, request latency, and error rates. Critical alerts will trigger email/SMS alarms to the DevOps team.

Chaos Engineering (Optional)

To improve resilience, we may adopt light **chaos testing**. For instance, we can simulate a database failure to verify that the system fails gracefully and recovers with minimal data loss. Such practices (inspired by Netflix's Simian Army) help ensure high availability. However, for this small-scale system, chaos tests will be limited to scheduled drills.

Chapter 7: Sustainability and Emerging Technologies

Serverless and Modern Tech

Future enhancements include using **serverless components** where feasible. For example, the notification service could be reimplemented as AWS Lambda functions triggered by database events to send emails/SMS without managing a full server. We could also explore a **Single Page Application (SPA)** frontend (React or Vue) calling Django REST APIs for a more responsive UI.

Green IT Considerations

We will host in a **green-friendly data center region**. The system will minimize resource use

by scaling down servers during off-peak hours. Efficient code and **caching** (e.g., using Redis for frequent queries) will reduce CPU load. Using Infrastructure as Code allows easy deployment to new cloud regions or providers if more eco-friendly options arise.

Future Enhancements

Planned features include developing a **mobile app** or progressive web app, integrating a **chatbot** for quick complaint intake, and using **AI/ML** to analyze complaint trends. As the report draft notes, possible enhancements are “Mobile app, chatbot, AI-based suggestions, push notifications”. An intelligent assistant could categorize complaints automatically or suggest solutions from a knowledge base. In summary, the design is made extensible so these emerging technologies can be incorporated.

Chapter 8: Conclusion and Future Work

Summary of Achievements

This report detailed the design of a comprehensive Student Support and Grievances System. We achieved all core objectives: students can register/log in, submit complaints, and track their status; administrators can manage and assign grievances; staff can update resolutions; and automated reports provide administrative insights. The chosen technology stack (Django, SQLite) and 3-tier architecture were justified by the requirements. We also laid out a clear QA and deployment plan, ensuring the solution is robust and maintainable.

Future Roadmap

Immediate next steps include finalizing the deployment (using IaC) and conducting UAT. Longer-term, we will implement the enhancements mentioned above (mobile app, advanced notifications, etc.) and transition the database to a more scalable system (PostgreSQL). We also plan to integrate analytics dashboards (e.g., with Grafana) for real-time monitoring of complaint metrics.

Lessons Learned

*Technical:** Using Django’s rapid development features (ORM, admin interface) greatly sped up prototyping. Ensuring a clean data model (via the ER diagram) helped avoid redesign later. We learned to balance simplicity (monolithic design) with scalability needs. Proper testing (unit and integration) early on prevented bugs from propagating.
*Process: The Agile approach** allowed quick adjustments. Clear requirement specifications were invaluable; any ambiguities there would have caused rework. Regular code reviews and automated testing in CI improved code quality. We also saw the importance of good

documentation (like this report) to communicate design decisions.

Each lesson will guide future phases of development and maintenance, ensuring the system remains effective and sustainable as it evolves.

Sources

The above design leverages requirements and initial analyses from the Student Support and Grievances System project report, along with established software architecture practices. All diagrams are conceptual examples corresponding to these designs.