International Journal of Engineering Research and Science & Technology

ISSN 2319-5991 www.ijerst.com

Peer Reviewed, Referred & Indexed Journal

Research Paper

COLLEGE ERP SYSTEM: A ROLE-BASED DJANGO PLATFORM FOR INTEGRATED ACADEMIC ADMINISTRATION

SIKHA VENKATA HARIKA
PG MCA Scholar, Department of Computer Science,
Sir C R Reddy College, Eluru, Andhra Pradesh, India-534005
harikasikha8@gmail.com

Banisetti Jyothi Satya Bhavani
Assistant Professor, Department of Computer Science,
Sir C R Reddy College, Eluru, Andhra Pradesh, India-534005
jyothiganesh006@gmail.com

Musunuru Ratnakar
HOD, Department of Computer Science,
Sir C R Reddy College, Eluru, Andhra Pradesh, India-534005

Abstract*—College administration frequently depends on fragmented registers and spreadsheets that delay attendance consolidation, result publication, leave processing, and institutional communication. This paper presents a role-based College ERP System implemented with Django to unify these activities within a secure web platform. The application supports Administrator, Staff, and Student roles through a custom user model and policy-controlled access. Administrators manage courses, subjects, academic sessions, staff, students, notifications, feedback, and leave decisions. Staff members record attendance, update examination results, submit leave requests, and review assigned-subject information, while students view personal attendance, results, library information, notifications, and leave status. The system adopts Django Model-View-Template architecture, MySQL/MariaDB persistence, Bootstrap interfaces, Google reCAPTCHA, Gunicorn, and WhiteNoise. A layered architecture separates presentation, application logic, authentication, and data management. Functional validation confirms that the implemented workflows preserve role boundaries, centralize academic records, and provide responsive dashboards for institutional monitoring. The solution offers a low-cost, maintainable alternative to manual processes and heavyweight commercial ERP suites, while remaining extensible for fee management, online examinations, parent access, notifications, and mobile services.

Keywords*— College ERP, Django, role-based access control, attendance management, academic administration, MVT architecture.

I. INTRODUCTION

A) Problem Context

Colleges generate continuously changing information about students, staff, subjects, attendance, examination performance, leave requests, feedback, and institutional notices. When these records are maintained in independent registers or spreadsheets, the same information must be repeatedly copied and reconciled. This increases transcription errors, delays reporting, and prevents students and staff from obtaining timely self-service access. Enterprise systems are intended to integrate data and processes across organizational units [1]-[4], but large commercial suites can be costly and operationally complex for a single small or medium institution.

The proposed College ERP System addresses this gap with a focused web application that centralizes academic administration without attempting to reproduce every financial or campus-management module. It follows established software-engineering and database principles [7]-[10], uses a relational schema for consistency, and maps system permissions to the actual institutional hierarchy. The central design objective is not only automation, but also controlled visibility: an administrator requires institution-wide access, a staff member requires subject-scoped operational access, and a student requires access only to personal academic information.

B) Objectives and Contributions

The work contributes a complete role-oriented workflow covering master-data management, attendance, examination results, leave processing, feedback, notifications, and dashboard visualization. Its principal contributions are: (i) a custom-user and profile model supporting Administrator, Staff, and Student roles; (ii) policy-controlled routing and permissions; (iii) a unified attendance and result repository; (iv) responsive role-specific dashboards; and (v) a deployment-ready Django configuration using Gunicorn and WhiteNoise. Security is reinforced through hashed credentials, CSRF protection, authenticated sessions, and reCAPTCHA, consistent with established access-control and application-security guidance [11]-[15].

C) System Scope and Paper Organization

The implemented scope covers the academic lifecycle after student and staff accounts have been created. It includes academic sessions, courses, subjects, teaching assignments, attendance events, individual attendance reports, test and examination marks, leave requests, feedback, notifications, and library browsing. Financial accounting, payroll, hostel, transport, and biometric devices remain outside the current boundary. This focused scope supports clear evaluation because each included function maps to a concrete role, data entity, and interface.

The remainder of the paper follows the sample conference structure. Section II positions the work against manual, spreadsheet, commercial, and open-source alternatives. Section III explains requirements, architecture, data design, workflows, implementation, security, and deployment. Section IV presents operational outputs, editable academic formulas, quality observations, and limitations. Section V concludes the work and identifies extensions suitable for production adoption.

II. RELATED WORK

A) Manual and Spreadsheet-Based Administration

Paper registers remain easy to introduce but difficult to aggregate. Attendance totals, examination scores, and leave histories are stored in separate locations, which makes cross-semester analysis slow and audit trails incomplete. Spreadsheet systems improve calculation accuracy, yet they still create departmental silos and provide weak role separation. File-level sharing cannot naturally express that one instructor may edit attendance for assigned subjects while a student may only view their own records. These limitations motivate database-backed systems with explicit identity and

authorization models.

B) Commercial and Open-Source ERP Systems

Commercial campus ERP products integrate admissions, fees, transport, hostel, payroll, examinations, and communication, but their licensing, configuration, and training requirements can exceed the needs of a single college. Open-source systems reduce licensing cost and permit customization, although many projects remain either too broad or insufficiently documented. The proposed system adopts the practical middle ground identified in the project report: a lightweight, open-source academic ERP that concentrates on frequently used institutional workflows and can be deployed on low-cost hosting.

C) Research Gap

Research on enterprise adoption emphasizes process fit, user acceptance, data quality, and organizational readiness [1]-[6]. However, academic demonstrations often present either static interface prototypes or isolated attendance modules rather than a traceable end-to-end implementation. This work closes that gap by connecting requirements, role policies, relational models, Django views, reusable templates, deployment components, and operational outputs in one coherent architecture. The resulting system is sufficiently small for academic inspection while still demonstrating principles of modularity, maintainability, portability, and secure access.

D) Technology and Adoption Foundations

Web-based ERP adoption depends on more than feature availability. Perceived usefulness, ease of use, information quality, and organizational fit affect whether stakeholders replace familiar manual practices [4]-[6]. For this reason, the project uses a common login, consistent navigation, familiar forms, and dashboards tailored to each role. The design minimizes training by presenting only the operations relevant to the logged-in user, while Django handles repetitive infrastructure concerns such as routing, forms, sessions, validation, and database access.

The technology selection also reflects deployment constraints. Python and Django provide a mature, documented framework; MySQL/MariaDB offers structured relational storage; Bootstrap supports responsive rendering; and WSGI deployment permits the application to run on conventional Linux hosting. This combination is appropriate for an educational institution that requires low cost, source-code ownership, and the ability to extend the application without dependence on a proprietary vendor.

III. MATERIALS AND METHODS

The system was developed using an incremental module-based method. Requirements were derived from the three institutional roles, translated into relational models and access rules, implemented through Django applications and templates, and validated through role-specific workflows. The method follows the five stages below.

A) Requirement Elicitation and Feasibility

Requirements were extracted from routine interactions among the Head of Department, teaching staff, and students. Each paper or spreadsheet task was converted into a user action, a validation rule, a persistent data object, and an expected response. Feasibility was examined technically, operationally, and economically. The selected open-source stack requires no specialized hardware or paid licenses, runs in a browser, and maps directly to existing institutional responsibilities. A deployment on PythonAnywhere further demonstrates that the application can operate within modest hosting resources.

Non-functional requirements included usability, confidentiality, integrity, portability, maintainability, and acceptable response time for ordinary institutional loads. Security-sensitive functions were delegated to framework-provided mechanisms wherever possible. This decision reduces the risk of errors in password handling, session management, request validation, and database queries, and follows the principle of using well-tested components for critical controls [11]-[15].

B) Functional Scope and Role Policy

The Administrator performs CRUD operations for courses, subjects, sessions, staff, and students; reviews attendance and feedback; manages leave decisions; and broadcasts notifications. Staff members operate only on assigned academic data, including attendance and examination results, while also submitting leave and feedback. Students receive read-oriented access to personal attendance, results, library information, and notifications, together with leave and feedback submission. Role policy is evaluated after authentication using the custom user_type field, where 1 denotes Administrator, 2 denotes Staff, and 3 denotes Student.

TABLE I. ROLE-WISE ACCESS SUMMARY

Module	Admin	Staff	Student
Master data	Create/Edit	View assigned	View own
Attendance	Monitor	Create/Update	View own
Results	Monitor	Create/Update	View own
Leave	Approve/Reject	Apply/View	Apply/View
Feedback/Notices	Reply/Broadcast	Submit/View	Submit/View

C) System Architecture

The application follows Django Model-View-Template architecture. Browser requests reach Gunicorn, pass through Django URL routing and authentication, invoke role-specific views, query models through the ORM, and render Bootstrap templates. MySQL or MariaDB stores persistent academic data, while WhiteNoise serves static resources in production. This layered arrangement isolates presentation, application rules, and persistence, and supports migration to other WSGI-compatible environments [16]-[27].

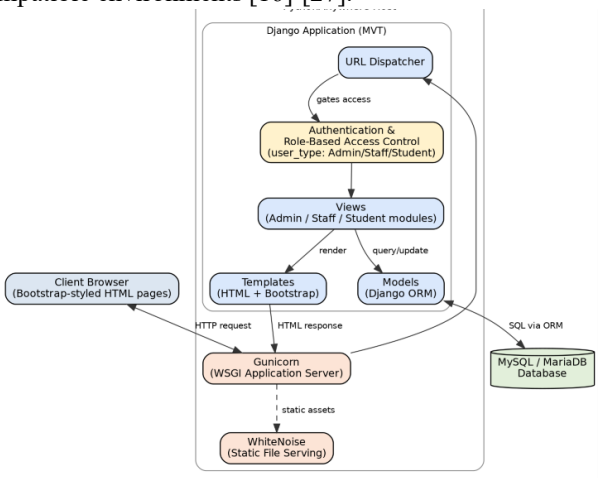

Fig. 1. System architecture of the College ERP platform.

D) Data Model and Academic Records

The central CustomUser entity is extended by AdminHOD, Staffs, and Students profiles. Courses, Subjects, and SessionYearModel represent the academic structure. Attendance identifies a subject, date, and session, while AttendanceReport records the status of each student. StudentResult stores subject-wise test and examination marks. Separate entities record staff and student leave, feedback, and administrative replies. Foreign keys maintain referential integrity and allow dashboards to aggregate information without duplicating master data. The schema follows normalized relational design principles [9], [10], [30].

E) Attendance and Result Processing Workflow

Attendance recording begins when an authenticated staff member selects a subject, academic session, and date. The view retrieves enrolled students through the course relationship and renders one status control per student. Submission creates one Attendance record and a corresponding AttendanceReport row for every learner. Validation prevents incomplete batches from being stored. This transaction-oriented design avoids partially recorded classes and gives the student dashboard a consistent source for subject-wise totals.

Result entry follows a similar authorization path. Staff can select only subjects assigned to them, enter internal and examination marks, and revise values when correction is permitted. Students receive read-only access to their own results. Leave and feedback workflows use status fields and administrative replies, providing an auditable transition from submission to review and decision.

F) Implementation Stack and Module Organization

The project is organized around Django applications, model classes, role-specific views, URL patterns, forms, templates, and static assets. The framework ORM translates model operations into parameterized database queries, while templates reuse a common navigation and theme structure. Environment variables and dj-database-url separate deployment settings from source code. The principal technologies and their roles are described through the request lifecycle and deployment discussion below.

The URL configuration separates Administrator, Staff, and Student operations while reusing common authentication and template components. Model classes represent user profiles, courses, subjects, academic sessions, attendance events, attendance reports, examination results, leave requests, feedback, and notifications. Forms validate submitted values before view logic changes the database, and templates expose only the controls appropriate to the authenticated role.

The implementation follows a traceable request lifecycle. A browser request is resolved by the URL dispatcher, checked by authentication and role-policy logic, processed by a view, translated into ORM queries, and rendered through a Bootstrap template. This organization reduces duplicated code and makes changes to academic rules easier to isolate. Database migrations preserve schema history, while environment-based connection settings support both local development and hosted deployment.

G) Authentication, Security, and Deployment

Django authentication provides salted password hashing, session management, CSRF protection, and permission-aware request handling. Google reCAPTCHA reduces automated login abuse, and role checks ensure least-privilege access. Environment-based database configuration avoids hard-coded deployment credentials. Gunicorn provides the WSGI process layer, WhiteNoise handles static assets, and PythonAnywhere offers a compact demonstration host. These controls reflect defense-in-depth, role-based access control, and secure software-design principles [11]-[15], [27]-[30].

Production hardening requires HTTPS-only cookies, secure secret storage, restricted database accounts, regular backups, centralized logs, dependency updates, and automated checks in the deployment pipeline. The current architecture permits these controls to be added incrementally because authentication, application views, static delivery, and persistence are already separated. RBAC remains the primary authorization mechanism, while audit events and object-level permissions can be introduced for sensitive operations such as result correction or bulk record deletion.

H) Data Governance and Policy Enforcement

Centralization creates value only when records remain accurate, attributable, and appropriately visible. The application therefore separates master data from transactional data. Courses, subjects, sessions, staff profiles, and student profiles are controlled by the Administrator, while attendance and results are created by authorized Staff and consumed by Students through read-only views. Leave and feedback records preserve status and reply fields so that submissions are not silently overwritten. This organization supports accountability because each change is associated with a known role and a defined workflow.

Policy enforcement occurs at several levels. Authentication establishes identity; user_type selects the allowed dashboard; decorators and view logic restrict routes; query filters scope records; templates hide unavailable actions; and relational constraints prevent orphan records. Hiding a button alone is not treated as authorization, because a user could attempt a direct URL request. The backend therefore verifies the role and object relationship before executing sensitive operations. This layered approach reflects the complete-mediation and least-privilege principles described in security literature [11], [15].

TABLE II. MODULE-TO-DATA RESPONSIBILITY MAPPING

Module	Primary Data	Authorized Role
Identity	CustomUser and profiles	Administrator / Self
Academic structure	Courses, Subjects, Sessions	Administrator
Attendance	Attendance, AttendanceReport	Staff; Admin monitor
Results	StudentResult	Staff; Student view
Leave	Staff/Student leave reports	Applicant; Admin decision
Feedback	Staff/Student feedback	Submitter; Admin reply
Notifications	Institution notices	Admin create; users view

Privacy is supported by minimizing the records shown to each user. A student query is filtered by the authenticated student profile, and a staff query is restricted to assigned subjects or personal requests. Administrative access is broader because institutional oversight requires aggregation, but production use should supplement role access with audit logs,

data-retention rules, export controls, and documented approval procedures. Backups should be encrypted and tested through restoration exercises rather than assumed to be valid.

I) Deployment, Maintenance, and Scalability

The demonstration deployment uses a conventional WSGI arrangement in which Gunicorn executes the Django application and WhiteNoise serves static resources. Database configuration is externalized so that local development and hosted execution can use different credentials without modifying application code. This separation improves portability and supports movement from a demonstration host to a virtual private server, platform service, or institutional data centre. A reverse proxy and HTTPS termination can be added without changing the MVT application structure.

Maintenance is simplified by Django migrations, reusable templates, modular views, and explicit model relationships. New academic sessions, courses, subjects, staff, and students are data additions rather than source-code changes. Future modules can reuse the same CustomUser identity and role policy. For example, fee management can link transactions to Students, online examination can link questions and attempts to Subjects, and parent access can introduce a guardian profile related to one or more learners.

Scalability has two dimensions. Data scalability concerns the growing number of users and records; appropriate indexes, query optimization, pagination, and database monitoring can address this. Organizational scalability concerns multiple departments or colleges; it requires tenant or institution identifiers, stronger object-level authorization, and separation of administrative domains. The present schema is appropriate for a single college, while a multi-institution service would require explicit tenancy design before deployment.

IV. RESULTS AND DISCUSSION

A) Functional Outcomes

The completed system provides distinct Administrator, Staff, and Student dashboards. The administrator view combines summary counters, attendance charts, enrolment information, and shortcuts for managing institutional records. Staff interfaces expose assigned-subject actions and attendance/result workflows. Student dashboards present personal attendance and marks using compact visual summaries. The documented validation confirms successful registration, authenticated redirection, CRUD operations, attendance submission, result entry, leave processing, feedback, and notification display. The system therefore satisfies the core functional requirements identified during analysis without exposing one role's operations to another.

B) Editable Academic Measures

Attendance is derived from the number of classes in which a student is marked present relative to the total classes recorded for the subject. The editable formula is:

$$Attendance\ (\%) = \frac{Present\ Classes}{Total\ Classes} \times 100 \quad (1)$$

For a subject containing internal test and examination components, a normalized result percentage can be represented as:

$$Result\ (\%) = \frac{Obtained\ Marks}{Maximum\ Marks} \times 100 \quad (2)$$

Functional validation can be summarized by the proportion of successful scenarios among all executed scenarios:

$$Validation\ (\%) = \frac{Passed}{Executed} \times 100$$

The project report states that the documented functional scenarios produced the expected outcomes. Consequently, the calculated validation rate for those documented scenarios is 100%, although this should not be interpreted as a substitute for load, penetration, accessibility, or long-duration reliability testing.

C) Interface Outputs

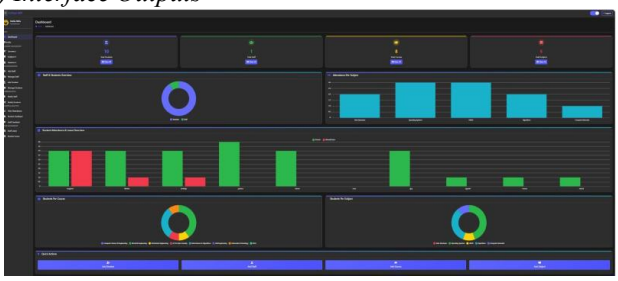

Fig. 2. Administrator dashboard with institutional summaries and analytics.

Figure 2 shows the Administrator view, where counters, attendance summaries, enrolment charts, and management shortcuts are presented together. This layout supports rapid inspection of institutional status before the administrator enters detailed CRUD or approval screens. Visual analytics do not replace underlying records; rather, they provide an aggregated navigation layer linked to the relational data.

D) Staff and Student Outputs

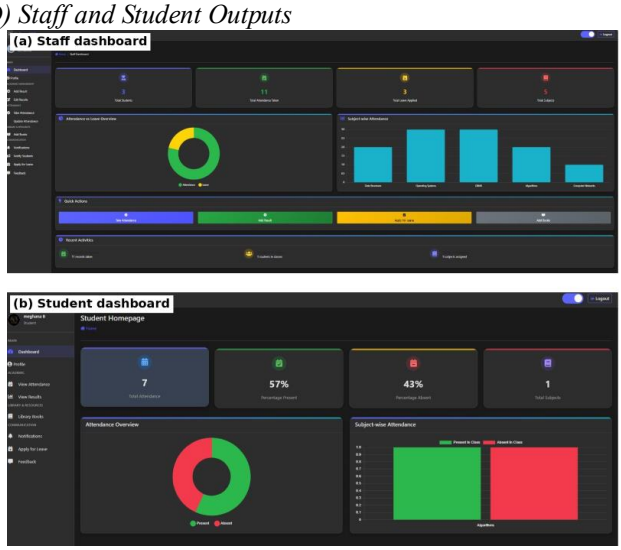

Fig. 3. Staff and Student dashboards in the deployed application.

The Staff dashboard prioritizes assigned-subject tasks, attendance actions, result entry, leave status, and quick navigation. The Student dashboard removes editing controls and emphasizes personal attendance percentages, subject-level marks, notices, and self-service requests. The contrast between the two outputs demonstrates that authorization is reflected not only in backend route checks but also in the visible interface, reducing accidental access attempts and simplifying user training.

E) Authentication Output

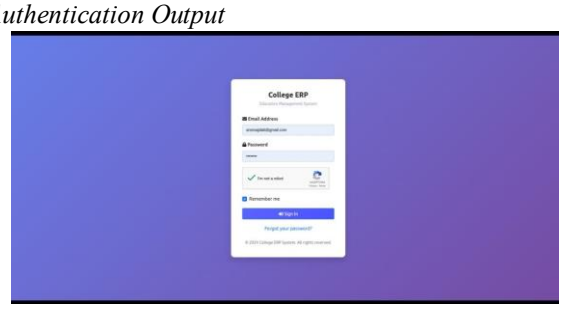

Fig. 4. Login interface with reCAPTCHA verification.

F) Dashboard Analytics and Quality Evaluation

The Administrator analytics view converts operational records into visual summaries that are easier to interpret than raw database rows. The dashboard combines a Staff-and-Student overview, subject-wise attendance bars, and student attendance-versus-leave comparisons. These outputs help the Head of Department identify weak attendance patterns, verify whether subjects are receiving regular attendance entries, and obtain a quick institutional overview before opening detailed management screens.

Fig. 5. Administrator analytics for role distribution and attendance monitoring.

Figure 5 presents the light-theme analytics output recorded in the project report. The donut chart summarizes the relative distribution of Staff and Student accounts, while the subject chart indicates where attendance records are available. The lower grouped bars distinguish present records from absence or leave records for individual students. Because these values are generated from the same relational records used by role dashboards, the visualization supports administrative review without creating a separate reporting database.

The output also demonstrates consistency across presentation modes. The project provides light and dark interfaces without altering role permissions or data semantics. Responsive Bootstrap components keep counters, charts, tables, and action controls readable on different screen sizes, while the dashboard remains connected to the central Django ORM layer. This separation allows visual refinements to be introduced without changing attendance, result, leave, or authorization rules.

TABLE III. IMPLEMENTATION QUALITY SUMMARY

Quality	Observed Design Evidence
Security	Hashed passwords, CSRF protection, reCAPTCHA, role checks
Usability	Role dashboards, responsive Bootstrap UI, theme support
Maintainability	Modular Django apps, ORM models, reusable templates
Portability	Python/WSGI deployment with environment-based database configuration
Scalability	Relational design supports additional users, courses, subjects, and sessions

The present implementation remains bounded to one institution and does not include fees, payroll, hostel, transport, parent accounts, native mobile clients, or online examinations. Dashboard results depend on the completeness of staff-entered records, and production deployment requires stronger secret management, HTTPS enforcement, automated backups, logging, and systematic security testing. Nevertheless, the architecture is modular enough to incorporate these capabilities without redesigning the core identity and academic-data model.

G) Operational Scenario Analysis

A typical attendance scenario begins with a Staff login, followed by subject, session, and date selection. The system retrieves enrolled students, requires a status for every learner, and persists the batch. The Administrator can later inspect aggregated attendance, while each Student sees only the personal subject-wise result. This scenario demonstrates identity verification, relationship-based filtering, transactional record creation, aggregation, and role-specific presentation in one end-to-end flow.

A typical leave scenario begins with a Student or Staff submission containing a date and message. The record is stored with a pending status and becomes visible to the Administrator. Approval or rejection updates the status while preserving the original request. The applicant dashboard then displays the decision. Unlike informal messages, this workflow creates a traceable state transition and prevents requests from being lost across email, paper, or verbal communication.

Feedback and notification scenarios provide two-way communication. Users submit feedback that can receive an administrative reply, while the Administrator broadcasts notices to one or more role groups. Because these functions use authenticated profiles, the system can associate communication with institutional users and display it in context. Production extensions may add delivery timestamps, read receipts, email or SMS gateways, and escalation policies without changing the core feedback entities.

CRUD scenarios for courses, subjects, sessions, staff, and students demonstrate referential dependencies. A subject belongs to a course and may be assigned to a Staff profile; a Student belongs to a course and academic session. Validation should therefore prevent deletion of master records that are referenced by attendance or result data, or should require an explicit archival policy. Soft deletion and academic-year closure procedures are recommended for long-term institutional use.

H) Threats to Validity

The reported results are functional rather than statistical. Screenshots and scenario outcomes show that the implemented interfaces and workflows operate as intended, but they do not establish performance under thousands of concurrent users or resistance to determined attackers. The demonstration data may not represent every departmental rule, grading scheme, attendance policy, or accessibility need. In addition, user satisfaction and reduction in administrative effort were not measured longitudinally.

These limitations can be addressed through staged evaluation. Unit and integration tests should cover models, permissions, forms, and status transitions; load tests should evaluate peak attendance and result-publication periods;

security assessment should include dependency scanning and penetration testing; accessibility testing should verify keyboard navigation and contrast; and pilot deployment should measure task time, error rate, and stakeholder satisfaction. Such evidence would complement the successful functional demonstrations reported here.

Compared with paper and spreadsheet administration, the implemented platform provides centralized records, stronger access separation, immediate dashboards, and consistent history. Compared with a commercial ERP suite, it offers narrower functionality but substantially lower complexity and greater source-level customizability. Its effectiveness therefore depends on the institution's actual scope: it is well suited to focused academic administration, but it should be integrated with specialized finance, payroll, hostel, or transport systems when those processes are required.

No machine-learning classifier is used in this project, so Accuracy, Precision, Recall, and F1-Score are not applicable evaluation measures. Appropriate evidence consists of successful role workflows, correct data relationships, immediate dashboard updates, and enforcement of authentication and authorization policies. Future evaluations should quantify response time, concurrent-user capacity, task completion time, accessibility, security findings, and user satisfaction under realistic institutional workloads.

V. CONCLUSION AND FUTURE WORK

A) Conclusion

This paper presented a complete College ERP System for role-based academic administration. The Django platform centralizes master data, attendance, results, leave, feedback, notices, and library information while preserving distinct Administrator, Staff, and Student permissions. Its MVT architecture, relational database, reusable templates, authentication controls, and lightweight deployment stack provide a maintainable alternative to paper records, disconnected spreadsheets, and oversized commercial suites. The project outputs demonstrate that a focused ERP can improve transparency, reduce repeated manual work, and offer timely self-service information to institutional users.

B) Future Work

Future development can add fee and scholarship management, online examinations, parent access, timetable generation, document workflows, email/SMS/push notifications, audit-log export, REST APIs, and mobile applications. Additional evaluation should include concurrent-user load tests, penetration testing, accessibility assessment, disaster-recovery exercises, and longitudinal user studies. Containerized deployment, continuous integration, centralized monitoring, and multi-tenant database separation can further prepare the platform for use across multiple colleges.

C) Deployment Recommendations

Before institutional deployment, the application should be configured with separate development, testing, and production environments. Production settings should disable debugging, load the secret key and database credentials from protected environment variables, enforce HTTPS, enable secure and HTTP-only cookies, restrict allowed hosts, and maintain scheduled database and media backups. A reverse proxy can provide TLS termination, request-size controls, and access logging, while application logs should be retained long enough to support troubleshooting and audit requirements. Database accounts should receive only the permissions required by the application, and administrative access should use stronger authentication than ordinary student accounts.

Operational procedures are equally important. Each academic session should have a documented opening and closure process, subject assignments should be reviewed before attendance begins, and result corrections should require authorization and an audit trail. Backup restoration should be tested periodically, inactive accounts should be disabled, and software dependencies should be reviewed for security updates. Training material should explain the responsibilities of each role and the correct handling of personal academic data. These measures convert the demonstrated software into a governed institutional service rather than an isolated web application.

D) Governance and Change Management

Long-term success also depends on governance. The institution should assign ownership for user provisioning, academic-session setup, subject allocation, attendance correction, result approval, and data retention. Proposed changes should be reviewed against role policy and database relationships before release, and production updates should follow version control, backup, migration, and rollback procedures. A small change advisory group can prioritize enhancements, document decisions, and prevent local customizations from producing inconsistent rules across departments. Periodic review of inactive accounts, permissions, exception reports, and audit records will help the system remain aligned with institutional policy as staff, courses, and academic regulations change.

REFERENCES

[1] T. H. Davenport, "Putting the enterprise into the enterprise system," Harvard Business Review, vol. 76, no. 4, pp. 121–131, 1998.
[2] F. F.-H. Nah, J. L.-S. Lau, and J. Kuang, "Critical factors for successful implementation of enterprise systems," Business Process Management Journal, vol. 7, no. 3, pp. 285–296, 2001.
[3] M. Al-Mashari, "Enterprise resource planning systems: A research agenda," European Journal of Operational Research, vol. 146, no. 1, pp. 1–4, 2003.
[4] W. H. DeLone and E. R. McLean, "The DeLone and McLean model of information systems success: A ten-year update," Journal of Management Information Systems, vol. 19, no. 4, pp. 9–30, 2003.
[5] F. D. Davis, "Perceived usefulness, perceived ease of use, and user acceptance of information technology," MIS Quarterly, vol. 13, no. 3, pp. 319–340, 1989.
[6] E. M. Rogers, Diffusion of Innovations, 5th ed. New York, NY, USA: Free Press, 2003.
[7] I. Sommerville, Software Engineering, 10th ed. Boston, MA, USA: Pearson, 2015.
[8] R. S. Pressman and B. R. Maxim, Software Engineering: A Practitioner's Approach, 9th ed. New York, NY, USA: McGraw-Hill, 2019.
[9] R. Elmasri and S. B. Navathe, Fundamentals of Database Systems, 7th ed. Boston, MA, USA: Pearson, 2015.
[10] A. Silberschatz, H. F. Korth, and S. Sudarshan, Database System Concepts, 7th ed. New York, NY, USA: McGraw-Hill, 2019.
[11] W. Stallings, Cryptography and Network Security: Principles and Practice, 7th ed. Boston, MA, USA: Pearson, 2017.
[12] R. Anderson, Security Engineering: A Guide to Building Dependable Distributed Systems, 3rd ed. Indianapolis, IN, USA: Wiley, 2020.
[13] OWASP Foundation, "Application Security Verification Standard 4.0.3," 2021.
[14] OWASP Foundation, "OWASP Top 10: The ten most critical web application security risks," 2021.
[15] National Institute of Standards and Technology, "Digital identity guidelines: Authentication and lifecycle management," NIST SP 800-63B, 2017.
[16] Python Software Foundation, "Python 3 documentation." [Online]. Available: https://docs.python.org/3/
[17] Django Software Foundation, "Django documentation." [Online].

Available: https://docs.djangoproject.com/
[18] Django Software Foundation, "Security in Django." [Online]. Available: https://docs.djangoproject.com/en/stable/topics/security/
[19] Oracle Corporation, "MySQL 8.0 reference manual." [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/
[20] Bootstrap Team, "Bootstrap documentation." [Online]. Available: https://getbootstrap.com/docs/
[21] Chart.js Contributors, "Chart.js documentation." [Online]. Available: https://www.chartjs.org/docs/
[22] Gunicorn Project, "Gunicorn documentation." [Online]. Available: https://docs.gunicorn.org/
[23] D. C. Lawrence, "WhiteNoise documentation." [Online]. Available: https://whitenoise.readthedocs.io/
[24] Google, "reCAPTCHA developer documentation." [Online]. Available: https://developers.google.com/recaptcha/
[25] PythonAnywhere, "PythonAnywhere help and deployment documentation." [Online]. Available: https://help.pythonanywhere.com/
[26] P. Eby, "Python Web Server Gateway Interface v1.0.1," PEP 3333, 2010.
[27] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.
[28] M. Fowler, Patterns of Enterprise Application Architecture. Boston, MA, USA: Addison-Wesley, 2002.
[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Boston, MA, USA: Addison-Wesley, 1994.
[30] E. F. Codd, "A relational model of data for large shared data banks," Communications of the ACM, vol. 13, no. 6, pp. 377–387, 1970.