© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1717082

AI Smart Attendance System Using Full Stack Java: Design, Implementation, and Evaluation

KULDEEP KHALOTIYA[1], MANISH RANA[2], NITIN KUMAR SINGH[3], SHUBHAM BAISAKH[4], SONALI KUMARI[5]

[1, 2, 3, 4, 5] *Department of Computer Science and Engineering, Parul Institute of Technology, Parul University, Vadodara, Gujarat, India*

Abstract- ***Traditional attendance management systems in educational institutions depend heavily on manual roll calls and paper-based registers. These approaches are error-prone, time-intensive, susceptible to proxy attendance, and produce data that is difficult to aggregate or analyse at scale. As institutions accelerate digital transformation, demand for reliable, role-aware, web-based attendance platforms has grown substantially. This paper presents the complete design, implementation, and empirical evaluation of an AI Smart Attendance System developed using Full Stack Java technologies — specifically Spring Boot 3.x, Hibernate ORM 6.x, MySQL 8.0, and a responsive Bootstrap 5 front-end. The system architecture follows the classic three-tier model separating the Presentation Layer (HTML5/CSS3/JavaScript), the Application Layer (Spring Boot MVC with Spring Data JPA), and the Data Layer (MySQL with Hibernate ORM). A RESTful API design pattern provides clean decoupling between client and server, enabling future integration with mobile applications or third-party academic systems without structural changes. Role-based access control (RBAC) is enforced for three user categories — Administrator, Faculty, and Student — each with dedicated dashboards and permission boundaries validated at the API level. Evaluation through functional and performance testing produced a 100% pass rate across all ten planned test cases. REST API GET endpoints averaged under 120 ms and POST endpoints under 200 ms under 50+ concurrent users. Dashboard pages loaded in under 1.8 seconds, and attendance-report database queries completed within 300 ms. The findings validate that a Spring Boot/MySQL full-stack architecture provides a robust, scalable, and maintainable foundation for enterprise-grade educational management systems.***

Index Terms- ***Attendance Management System, Bootstrap 5, Educational Technology, Full Stack Java, Hibernate ORM, MySQL, RESTful API, Role-Based Access Control, Spring Boot, Spring Data JPA, SDLC, Web Application***

I. INTRODUCTION

Attendance monitoring is a fundamental administrative task in educational institutions, directly correlating with student engagement and academic performance [1]. Research consistently demonstrates that regular class attendance is among the strongest predictors of student achievement, making its accurate documentation a critical institutional responsibility. Despite its importance, most institutions continue to rely on manual, paper-based registers that are susceptible to human error, proxy attendance, and considerable administrative overhead. Digitising this process has long been recognised as a priority, yet comprehensive, role-aware, web-based solutions remain underutilised at the institutional level.

Full Stack Java Development — encompassing back-end API design, database management, and responsive front-end construction — provides an ideal platform for such enterprise applications. Java's 'Write Once, Run Anywhere' principle, combined with the Spring Boot ecosystem and the reliability of MySQL, enables rapid development of scalable, maintainable systems. Spring Boot 3.x in particular eliminates the extensive XML-based configuration traditionally associated with the Java EE ecosystem, allowing developers to focus on business logic rather than infrastructure setup. This productivity advantage is especially significant in resource-constrained institutional settings where development timelines are short and budgets are limited.

This paper documents the end-to-end design and evaluation of an AI Smart Attendance System built on these foundations during a twelve-week industry internship at QSpiders, Ahmedabad. The project was conceived in response to real institutional pain points: faculty spending 5–10 minutes per session on roll calls, attendance registers being misplaced or damaged, and the absence of early-warning mechanisms for students approaching minimum attendance thresholds. The system addresses each of these pain points through targeted software

engineering decisions backed by empirical performance measurement.

The platform delivers the following core capabilities: (1) secure, role-based authentication for Administrators, Faculty, and Students using Spring Security; (2) session-wise attendance marking with Present/Absent/Late granularity; (3) automated report generation with date-range filtering and CSV export; (4) real-time analytics dashboards powered by Chart.js; and (5) configurable low-attendance alerts triggered at a defined threshold (default 75%). Together, these capabilities replace a fragmented manual workflow with an integrated digital system accessible from any browser-enabled device.

The remainder of this paper is organised as follows: Section II reviews related work in digital attendance and educational management systems. Section III describes the system architecture, tools, and implementation methodology. Section IV presents functional and performance evaluation results. Section V discusses implications, comparisons with related work, and limitations. Section VI concludes with directions for future enhancement.

II. LITERATURE REVIEW

A. Traditional vs. Digital Attendance

Conventional roll-call methods impose significant burdens on faculty time and remain vulnerable to proxy attendance — a well-documented problem in higher education globally. Patil et al. [1] demonstrated that a web-based Java EE attendance system reduced administrative effort by 70% and improved accuracy to 99.2%, establishing a clear empirical case for digitisation. Their study surveyed 120 faculty members across three institutions and found that automated attendance recording freed an average of 7.4 minutes per class session, translating to over 60 hours of recovered instructional time per semester per department.

Earlier work by Goel and Singh [4] compared paper-based, spreadsheet-based, and web-based attendance systems across five key metrics: accuracy, time efficiency, data accessibility, report generation speed, and security. Web-based systems outperformed alternatives on all five metrics, yet adoption remained low due to implementation complexity and upfront development cost — gaps that the Spring Boot ecosystem substantially narrows.

B. Spring Boot and Microservices in Educational Systems

Kumar and Sharma [2] developed a Spring Boot attendance API integrated with a React front-end, highlighting the effectiveness of microservices architecture for educational management. Their work confirms that Spring Boot's auto-configuration and embedded server model significantly accelerates development without sacrificing performance. They reported an 85% reduction in boilerplate configuration compared with traditional Spring MVC projects, enabling a two-person team to build a functional prototype in under three weeks.

Mehta et al. [5] extended this line of research by embedding a Spring Boot attendance microservice within a broader Learning Management System (LMS), demonstrating that RESTful APIs enable clean integration across institutional subsystems. Their API-first design philosophy — specifying endpoints and data contracts before writing implementation code — is reflected in the present system's architecture and contributed to the near-zero integration defects observed during testing.

C. Role-Based Access Control in Web Applications

Singh et al. [3] presented a full-stack Java attendance system secured with Spring Security, achieving measurable improvements in data security and accessibility. Role-based access control (RBAC) has since become a de facto requirement for multi-stakeholder educational platforms. Their security audit showed that RBAC enforcement at the API layer — rather than only at the UI layer — was essential to preventing privilege escalation attacks, a finding directly incorporated into the present system's Spring Security configuration.

A comprehensive survey of educational system security by Verma and Tripathi [6] identified unauthorised data access as the leading security risk in institutional web applications, ahead of SQL injection and cross-site scripting. Their recommended mitigation — combining session-based authentication with method-level security annotations — is implemented in the present system through Spring Security's @PreAuthorize mechanism applied to all service-layer methods.

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1717082

D. Database Performance in High-Concurrency Educational Systems

Effective database design is critical for attendance systems, which frequently execute complex aggregation queries across large datasets. Bauer and King [8] document optimisation strategies for Hibernate ORM applications, recommending composite indices on frequently joined columns as the highest-impact single optimisation for read-heavy workloads. This guidance is directly applied in the present system through composite indexing on (student_id, course_id, date) in the attendance table, yielding a 65% query time reduction observed during performance testing.

E. Research Gap

Existing studies address individual components — API design, ORM integration, or front-end responsiveness — in isolation. Comprehensive empirical evaluations covering the full stack from database query performance to UI load time are rare in the educational software literature. Furthermore, most reported systems lack configurable business-logic features such as adjustable attendance-threshold alerts and role-differentiated report access. This paper contributes a unified account of all architectural layers together with empirical performance benchmarks and qualitative development insights, providing actionable guidance for developers undertaking similar educational platform projects.

III. METHODOLOGY

A. System Architecture

The system follows a classic Three-Tier Architecture that enforces a clean separation of concerns across the Presentation, Application, and Data layers (Fig. 1). This architectural pattern was selected for its proven scalability and testability characteristics in enterprise Java applications. Each tier communicates only with its adjacent tier: the Presentation Layer submits HTTP requests to the Application Layer's REST API, which in turn executes queries against the Data Layer through Hibernate ORM. This isolation enables independent scaling — for example, the database tier can be migrated to a replicated MySQL cluster without modifying application or front-end code.

Table I: System Architecture Overview

Layer	Component	Technology Stack
Presentation	Web Browser UI	HTML5, CSS3, JavaScript ES6, Bootstrap 5
Application	Spring Boot App	Java 17, Spring Boot 3.x, Spring MVC, Spring Security
Business Logic	Service Classes	Java Service Layer, Hibernate 6.x, Spring Data JPA
Data	Relational DB	MySQL 8.0 with Hibernate ORM, Composite Indices
Integration	REST API Layer	Spring Boot RESTful endpoints, JSON, Fetch API
Build & Deploy	DevOps	Maven 3.9, Apache Tomcat 10 (embedded), JAR packaging

Java 17 was selected for its long-term support (LTS) status, ensuring security patches through 2029. Spring Boot 3.x provided the framework backbone, with its opinionated defaults reducing project setup time from an estimated two days to under four hours compared with a bare Spring MVC project. MySQL 8.0 served as the relational data store, accessed through Hibernate 6.x via Spring Data JPA. The front-end was built with HTML5, CSS3, Bootstrap 5, and JavaScript ES6, communicating with the back-end exclusively via the Fetch API to maintain a strict API contract.

B. Software Development Life Cycle

Development followed an Agile-inspired SDLC with seven iterative phases spanning the twelve-week internship. Each phase produced deliverable artefacts reviewed before the next phase commenced, enabling early defect detection and requirements refinement. Table II summarises the phases, durations, and key deliverables.

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1717082

Table II: SDLC Phase Summary

Phase	Duration	Key Deliverables
1. Requirement Analysis	Week 1	User stories, use-case diagram, feature backlog
2. System Design	Week 2	Architecture diagram, ER diagram, API contract, wireframes
3. Database Design	Week 3	MySQL schema, DDL scripts, seed data
4. Back-End Development	Weeks 4–7	Spring Boot REST API, Spring Security config, unit tests
5. Front-End Development	Weeks 6–8	Bootstrap 5 pages, Chart.js dashboards, AJAX integration
6. Integration & Testing	Weeks 9–10	Postman collections, UAT, bug fixes
7. Deployment & Review	Weeks 11–12	JAR deployment, load testing, documentation

C. Database Design

The MySQL schema comprises seven normalised tables linked by foreign key constraints. The users table stores authentication credentials (bcrypt-hashed passwords), roles, and account status. The students and faculty tables extend users with domain-specific attributes. The courses table captures module metadata. The enrollments table implements the many-to-many relationship between students and courses. The attendance table records each session event with Present/Absent/Late status and a timestamp. The attendance_report table stores pre-computed aggregates (total classes, present count, attendance percentage) per student–course pair, updated via a triggered stored procedure after each marking session to avoid expensive runtime aggregation.

Referential integrity is enforced through foreign key constraints with ON DELETE CASCADE semantics where appropriate, ensuring that removing a course automatically purges associated enrollment and attendance records. A composite index on attendance(student_id, course_id, date) — the most frequently queried combination — reduced report-generation query time by 65% compared with the unindexed baseline during performance testing. A separate index on users(email) accelerates login lookups.

D. Back-End: Spring Boot REST API

The back-end is structured in four horizontal layers following the Repository pattern. The Controller Layer exposes RESTful endpoints returning JSON payloads via @RestController annotations. All endpoints are versioned under the /api/v1/ prefix to support future non-breaking API evolution. The Service Layer encapsulates all business logic and validation, including attendance threshold calculation and alert triggering, ensuring that controllers remain thin and testable in isolation. The Repository Layer extends Spring Data JPA's JpaRepository interface, receiving automatic CRUD implementations and enabling custom queries through the @Query annotation with JPQL. The Entity Layer maps Java POJOs to MySQL tables using JPA annotations (@Entity, @Table, @Column, @ManyToOne, @OneToMany).

Table III: Key REST API Endpoints

Method	Endpoint	Role	Description
POST	/api/v1/auth/login	All	Authenticate user; return session token and role
GET	/api/v1/students	Admin	Retrieve paginated student list
POST	/api/v1/students	Admin	Create new student record
PUT	/api/v1/students/{id}	Admin	Update student details

DELETE	/api/v1/students/{id}	Admin	Remove student and cascade attendance
POST	/api/v1/attendance/mark	Faculty	Record bulk session attendance
GET	/api/v1/attendance/report/{studentId}	Faculty/Student	Personalised attendance summary
GET	/api/v1/attendance/export?course={id}	Faculty	Download CSV report for course
GET	/api/v1/dashboard/stats	Admin	Aggregated institutional statistics
GET	/api/v1/alerts/low-attendance	Admin/Faculty	Students below threshold

Spring Security is configured with an in-memory session store and method-level security (@PreAuthorize) on all service methods. HTTP CSRF protection is enabled for form-submission endpoints, while the REST API endpoints use SameSite cookie attributes to mitigate cross-site request forgery. All password storage uses BCryptPasswordEncoder with a strength factor of 12, balancing security with login latency.

E. Front-End Development

The front-end comprises six primary views, each tailored to a specific user role and workflow. The Login Page performs role-based redirection upon successful authentication, sending Administrators to the system dashboard, Faculty to the attendance marking interface, and Students to their personal portal. The Admin Dashboard presents real-time institutional statistics — total students, total courses, overall attendance rate, and active alerts — rendered as animated Chart.js doughnut and bar charts refreshed via periodic AJAX polling every 60 seconds.

The Student Management view provides full CRUD functionality with server-side pagination (20 records per page), client-side search filtering, and inline form validation before submission. The Attendance Marking view presents a faculty-facing bulk-checkbox interface that lists all enrolled students for a selected course and session date, with a single-click 'Mark All Present' shortcut for efficiency. The Attendance Report view accepts date-range inputs and course selection, rendering a filterable HTML table and providing one-click CSV export via a Spring Boot endpoint that streams a comma-separated file using Spring's StreamingResponseBody. The Student Portal displays a personal attendance summary table with colour-coded percentage indicators (green ≥ 75%, amber 60–74%, red < 60%) aligned with common institutional regulations. All views are fully responsive, tested across Chrome 120, Firefox 121, and Edge 120 on desktop (1920×1080), tablet (768×1024), and mobile (390×844) viewports.

IV. RESULTS AND ANALYSIS

A. Functional Validation

A structured test plan comprising ten functional test cases was executed upon completion of integration. All ten cases passed on first execution, yielding a 100% pass rate with zero defects requiring rework (Table IV). Test coverage spanned the complete user journey for each of the three roles: Administrator (student and course management, dashboard statistics, user account creation), Faculty (attendance marking, report generation, CSV export), and Student (portal access, personal attendance view). Every REST endpoint returned the correct HTTP status code — 200 OK for successful retrievals, 201 Created for resource creation, 400 Bad Request for malformed payloads, 401 Unauthorized for unauthenticated access, and 403 Forbidden for role violations.

Table IV: Functional Test Results

TC#	Feature	Test Case	Expected Result	Status
TC-01	User Login	Valid credentials submitted	Role-specific dashboard redirect	PASS
TC-02	User Login	Invalid password submitted	Error message displayed; no redirect	PASS

TC-03	Student CRUD	Admin adds new student record	Record persisted; list refreshes	PASS
TC-04	Attendance Marking	Faculty marks bulk session attendance	All records saved; confirmation shown	PASS
TC-05	Attendance Report	Faculty filters by course and date range	Filtered data displayed correctly	PASS
TC-06	CSV Export	Faculty exports attendance as CSV	File downloaded with correct data	PASS
TC-07	REST GET /students	Admin sends authenticated GET request	JSON array returned; 200 OK	PASS
TC-08	REST POST /attendance	Faculty POSTs valid attendance DTO	201 Created; DB record verified	PASS
TC-09	Role-Based Access	Student attempts to access /admin/**	403 Forbidden returned	PASS
TC-10	Dashboard Charts	Admin loads dashboard page	Chart.js charts rendered; data accurate	PASS

B. Performance Metrics

Performance testing was conducted using Apache JMeter 5.6 with a simulated load of 50 concurrent virtual users executing a representative mix of API calls (40% GET /students, 30% POST /attendance/mark, 20% GET /attendance/report, 10% GET /dashboard/stats). Tests were run over a five-minute sustained load period on a development server with 8 GB RAM and an Intel Core i5-1135G7 processor — hardware representative of low-cost institutional server deployments. Results are summarised in Table V.

Table V: Performance Test Results (50 Concurrent Users, 5 Minutes)

Metric	Target	Result	Status
API GET Average Response Time	< 200 ms	112 ms	PASS
API POST Average Response Time	< 300 ms	183 ms	PASS
API GET 95th Percentile Response	< 350 ms	198 ms	PASS
API POST 95th Percentile Response	< 500 ms	312 ms	PASS
Dashboard Page Load Time	< 2.0 s	1.74 s	PASS
Attendance Report Query Time (DB)	< 500 ms	287 ms	PASS
CSV Export (500 records)	< 1.0 s	0.63 s	PASS
Error Rate (all endpoints)	< 1%	0.0%	PASS
Functional Test Cases Passed	10/10	10/10 (100%)	PASS
Browser Compatibility	3 browsers	Chrome, Firefox, Edge	PASS

C. Development Observations

Several qualitative insights emerged during the twelve-week implementation. Spring Boot's auto-configuration reduced initial project setup from an estimated two days to under four hours, eliminating the need for manual dispatcher servlet, view resolver, and data source configuration. Spring Data JPA with Hibernate eliminated repetitive JDBC boilerplate — an estimated 1,200 lines of DAO code replaced by 7 repository interfaces — increasing development

velocity by an estimated 60%. Bootstrap 5's grid system and pre-built form components accelerated front-end delivery, with complete responsive layouts achievable in two to three hours per view rather than the one to two days required for hand-crafted CSS.

The most impactful single optimisation was the addition of the composite index on attendance(student_id, course_id, date), which reduced attendance report query time from 820 ms to 287 ms — a 65% improvement — under the standard test dataset of 1,000 students, 20 courses, and 18 weeks of daily session records (approximately 180,000 attendance rows). This finding corroborates Bauer and King's [8] guidance on composite index design for Hibernate-backed applications and underscores the importance of profiling query execution plans before declaring a system production-ready.

D. Security Validation

Security testing confirmed that Spring Security's method-level access control correctly rejected all unauthorised access attempts. A total of 30 cross-role access tests were executed — ten per non-admin role attempting to reach endpoints restricted to higher-privilege roles. All 30 returned 403 Forbidden with no information leakage in the response body. BCrypt password verification added an average of 42 ms to the login endpoint response time — an intentional and acceptable overhead given that login is a low-frequency, high-security operation. SQL injection attempts using common payloads (UNION SELECT, OR 1=1, comment sequences) against all string-typed input fields were entirely neutralised by Hibernate's parameterised query mechanism.

E. Business Impact Assessment

Based on a qualitative assessment conducted at the end of the internship, the system eliminates the entire manual attendance pipeline — recording, aggregation, and reporting — replacing it with a real-time digital workflow. Faculty attendance marking time was reduced from an average of seven minutes per session to under ninety seconds using the bulk-checkbox interface. The automated low-attendance alert mechanism identified students approaching the 75% threshold in real time, enabling proactive academic counselling that was previously feasible only after manual end-of-term report compilation. The clean RESTful API design positions the system for future integration with the institution's existing Student Information System without requiring architectural changes.

V. DISCUSSION

A. Implications for Educational Software Development

The results support three broad implications for developers and institutions undertaking similar digital transformation projects. First, a layered Spring Boot architecture naturally enforces separation of concerns, making systems easier to maintain and extend as requirements evolve. The ability to replace the front-end technology (for example, migrating from server-rendered Bootstrap to a React SPA) without modifying the API or database layers was validated during a mid-project scope change when Chart.js was substituted for Highcharts due to licensing constraints — a change that required only front-end modifications.

Second, Spring Data JPA repositories dramatically reduce the code surface area vulnerable to data-access bugs, improving reliability without sacrificing the flexibility provided by @Query annotations for complex aggregations. The seven JpaRepository interfaces in this system replace an estimated 1,200 lines of hand-written DAO code, proportionally reducing the lines-of-code-per-defect metric and simplifying onboarding for new developers.

Third, Bootstrap 5's responsive grid and pre-built component library eliminate the need for separate mobile codebases, reducing long-term maintenance cost. In an institutional context where front-end development expertise is scarce, the ability to deliver a professional, responsive UI using primarily utility classes — rather than bespoke CSS — is a significant practical advantage.

B. Comparison with Related Work

Compared with Patil et al. [1], who reported a 70% reduction in administrative effort using Java EE, the Spring Boot approach achieves comparable functional outcomes with significantly less configuration overhead and a more modern dependency injection model. The Spring Boot application was deployable as a self-contained JAR with an embedded Tomcat server, eliminating the need for a separate application server installation — a meaningful operational simplification in

institutional IT environments with limited systems administration capacity.

Unlike the React-based front-end of Kumar and Sharma [2], this system uses server-rendered HTML with AJAX enhancement, reducing client-side complexity while maintaining responsiveness. This architectural choice also improves search-engine indexibility and accessibility for screen readers — considerations relevant to institutions with accessibility compliance obligations. The trade-off is that richer single-page application interactions (such as offline capability) are not achievable in the current architecture; this is acknowledged as a limitation and addressed in the future directions below.

The Spring Security integration echoes Singh et al. [3] and extends it with two features absent from their reported system: configurable attendance-threshold alerts and CSV export functionality. These additions were identified as high-priority requirements during the internship's requirement analysis phase, reflecting real institutional workflows that prior academic prototypes had not addressed.

C. Limitations

Several limitations were identified during development and testing. First, the authentication layer currently uses session-based login stored in server memory; stateless JWT-based authentication would improve scalability for distributed or cloud deployments where multiple application instances may run behind a load balancer. Second, the system requires stable internet connectivity and has no offline mode, limiting its utility in institutions with unreliable network infrastructure — a common constraint in developing regions where digital transformation in education is most urgently needed.

Third, face-recognition or biometric verification — implied by the 'AI' designation — is not yet implemented in the current release. The system is 'AI-ready' in that its REST API exposes a POST /api/v1/attendance/biometric endpoint designed to accept future biometric payloads from an external recognition service, but the recognition service itself is outside the present implementation scope. Fourth, load testing was limited to 50 concurrent users; institutions with enrolments exceeding several thousand students accessing the system simultaneously during a high-stakes period (such as end-of-term report release) would require horizontal scaling, a Redis caching layer for session management, and potentially read-replica databases.

Fifth, the system currently lacks an audit trail for attendance amendments — if a faculty member corrects a mistaken absence marking, the original record is overwritten without logging the change. For regulatory environments where attendance data carries legal significance, an append-only audit log table with amendment records would be required. This represents a straightforward schema addition planned for the next development sprint.

VI. CONCLUSION

This paper has presented the complete design, implementation, and empirical evaluation of an AI Smart Attendance System built on Full Stack Java technologies. The platform successfully automates the complete attendance lifecycle — from session-wise marking to report generation and low-attendance alerting — within a secure, role-aware, responsive web application developed over twelve weeks. The technology stack — Java 17, Spring Boot 3.x, Spring Data JPA, Hibernate 6.x, MySQL 8.0, and Bootstrap 5 — proved well-suited to the requirements of an institutional-grade educational management system, delivering the performance, security, and usability characteristics demanded by multi-stakeholder academic environments.

All ten planned functional test cases passed with 100% success, and REST API response times remained well below established web-performance thresholds under simulated concurrent load. Dashboard pages loaded in under two seconds, and database query optimisation through composite indexing delivered a 65% reduction in report generation latency. These empirical results provide concrete benchmarks against which future implementations can be measured, addressing a gap in the existing educational software literature where performance data is rarely reported with methodological rigour.

The three primary contributions of this work are: (1) a reusable, documented three-tier Spring Boot architecture with RESTful API contracts suitable for adoption by other institutions; (2) empirical performance and functional benchmarks establishing realistic expectations for comparable deployments; and (3) a modular, API-first design that

accommodates future enhancement — biometric verification, native mobile clients, and machine learning-driven at-risk student prediction — without requiring structural changes to the existing codebase.

Future development will proceed along five fronts: migration to JWT-based stateless authentication for improved horizontal scalability; integration of a face-recognition microservice exposed as a REST endpoint consumed by the Spring Boot backend; development of native Android and iOS applications leveraging the existing API; an advanced analytics module providing cohort-level and longitudinal attendance trend analysis; and a machine learning pipeline predicting at-risk students from attendance patterns using scikit-learn models served via a Python Flask microservice. Together, these enhancements will realise the full vision of an AI-driven attendance ecosystem capable of transforming institutional data into actionable academic intelligence.

ACKNOWLEDGMENT

The authors gratefully acknowledge the guidance and infrastructure support provided by QSpiders Institute, Ahmedabad, during the twelve-week internship project. Special thanks are due to the faculty coordinators at Parul Institute of Technology, Parul University, for their continuous mentorship, and to the peer reviewers whose constructive feedback substantially improved the quality of this manuscript.

REFERENCES

[1] A. Patil, R. Sharma, and K. Singh, "Web-Based Attendance Management System Using Java EE," International Journal of Computer Applications, vol. 180, no. 15, pp. 12–18, 2020.
[2] V. Kumar and P. Sharma, "Full Stack Attendance System Using Spring Boot and React," Journal of Software Engineering and Applications, vol. 14, no. 6, pp. 221–234, 2021.
[3] R. Singh, A. Gupta, and S. Verma, "Role-Based Attendance System with Spring Security," Proc. National Conference on Emerging Technologies, Pune, 2022.
[4] S. Goel and M. Singh, "Comparative Study of Attendance Tracking Systems in Higher Education," International Journal of Educational Technology, vol. 9, no. 2, pp. 45–58, 2019.
[5] R. Mehta, P. Joshi, and A. Patel, "Integrating Attendance Microservices into Learning Management Systems," Proc. International Conference on Smart Education, Mumbai, pp. 112–119, 2022.
[6] P. Verma and S. Tripathi, "Security Risk Assessment of Educational Web Applications," Journal of Information Security Research, vol. 11, no. 4, pp. 233–247, 2020.
[7] J. Gosling, B. Joy, and G. Steele, The Java Language Specification, Java SE 8 Edition. Oracle/Addison-Wesley, 2014.
[8] C. Bauer and G. King, Java Persistence with Hibernate, 3rd ed. Manning Publications, 2020.
[9] C. Walls, Spring Boot in Action, 2nd ed. Manning Publications, 2022.
[10] J. Bloch, Effective Java, 3rd ed. Addison-Wesley Professional, 2018.
[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
[12] J. Duckett, HTML and CSS: Design and Build Websites. Wiley, 2011.
[13] Oracle Corp., "Java SE 17 Documentation," 2024. [Online]. Available: https://docs.oracle.com/en/java/javase/17/
[14] Spring.io, "Spring Boot 3.x Reference Documentation," 2024. [Online]. Available: https://docs.spring.io/spring-boot/
[15] MySQL, "MySQL 8.0 Reference Manual," 2024. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/
[16] Bootstrap Team, "Bootstrap 5 Documentation," 2024. [Online]. Available: https://getbootstrap.com/docs/5.3/
[17] Baeldung, "Spring Boot REST API Tutorials," 2024. [Online]. Available: https://www.baeldung.com/spring-boot