Raffles University Placement Management System (RUPMS): A Role-Based Full-Stack Web Application for Automating Campus Recruitment Workflows

[1]Vishal yadav,[2]Hemant kumar ,[3]Rajendra singh
[1,2,3] *Department of Computer Science and Engineering, School of Engineering and Technology*
Raffles University, Neemrana, Alwar, Rajasthan – 301705, India

***Abstract*—Campus recruitment at Indian technical universities continues to rely on fragmented manual processes—spreadsheet tracking, broadcast emails, and physical application forms—that introduce significant administrative inefficiency, poor student visibility, and inadequate reporting for institutional decision-making. This paper presents the design, implementation, and evaluation of the Raffles University Placement Management System (RUPMS), a full-stack web application developed on the MERN (MongoDB, Express.js, React.js, Node.js) architecture to digitise and consolidate the entire placement lifecycle. RUPMS provides four independent role-based portals—Student, TPO Admin, Management Admin, and Super Admin—each equipped with a purpose-built interface and enforced access boundaries via JSON Web Token authentication and middleware-level role guards. Core capabilities include cloud-based document storage through Cloudinary, automated email dispatch via Nodemailer, real-time placement analytics rendered with Chart.js, and a Kubernetes-ready Docker deployment configuration. Structured functional testing across ten test scenarios and performance benchmarking under twenty concurrent users demonstrate that the system processes all defined use cases correctly, maintains median API response latencies below 150 milliseconds, and sustains uninterrupted operation across a 48-hour evaluation window. The findings confirm that a unified, institutionally tailored placement portal substantially reduces administrative overhead and enables data-driven placement management.**

***Index Terms*—Placement Management System, MERN Stack, Role-Based Access Control, React.js, Node.js, MongoDB, JWT Authentication, Cloudinary, Nodemailer, Chart.js, REST API, Docker, Kubernetes**

I. INTRODUCTION

The placement cell of a technical university occupies a critical position in the institutional value chain. For graduating students, it represents the primary channel through which career opportunities are identified and secured; for management, placement outcomes serve as a measurable indicator that influences rankings, accreditation assessments, and future enrolments. Despite this importance, the majority of Indian engineering institutions continue to administer placement activities through approaches that are inherently limited in scalability and transparency.

At Raffles University, Neemrana, the existing process depends on notice-board circulars, email broadcasts, and spreadsheet records. As institutional scale has grown, several structural shortcomings have become pronounced: the absence of a centralised student profile repository forces repeated re-submission of credentials; application tracking occurs through informal email threads rather than a structured system; placement analytics require manual extraction from spreadsheets; and no mechanism enforces the distinct authority boundaries between TPO administrators, management, and super-administration.

This paper describes RUPMS—an integrated solution that addresses all four shortcomings through a single, cloud-hosted platform. Section 2 situates the work within existing literature. Section 3 details the system architecture and design. Section 4 presents the implementation. Section 5 reports evaluation outcomes. Section 6 concludes with directions for future work.

II. RELATED WORK

Rao and Mishra [1] surveyed fifteen Indian technical institutions and identified four recurring failure modes in manual placement processes: data duplication, communication delays, absence of real-time statistics, and unstructured access control. Sharma and Tiwari [2] quantified the cost of manual applicant review at an average of four working hours per job posting per TPO, proposing role-based digital portals as the structural remedy.

Commercial placement platforms offer multi-institutional job boards but lack the institutional-specific role hierarchies and branding required by individual universities [3]. Kumar et al. [4] implemented a PHP/MySQL placement portal that achieved functional coverage for job posting and application management but noted the absence of a real-time analytics module as a critical gap. Janikow and Smith [5] described a MEAN-stack multi-role educational system and observed that the principal architectural challenge lies not in API-level access enforcement but in designing role-appropriate frontend interfaces without maintaining separate codebases per role.

Alhumaidan et al. [6] compared MERN, PHP-Laravel, and Python-Django for educational management systems and found MERN superior in development velocity for applications requiring rich client-side interactivity, recommending the Context API over Redux for moderate-complexity state management. Singh and Kaur [7] demonstrated Cloudinary's suitability for academic document management, and Dubey and Verma [8] documented Nodemailer with

Gmail SMTP as a cost-effective transactional email solution for Node.js academic deployments. RUPMS builds on this body of work by combining four-role access control, live analytics, cloud file storage, automated email notifications, and Docker-based deployment within a single institutionally tailored MERN application—a combination that none of the reviewed systems achieves in full.

III. SYSTEM DESIGN AND ARCHITECTURE

3.1. Architectural Overview

RUPMS adopts a four-tier layered architecture. The Client Layer delivers a React.js 19 single-page application, built with Vite and styled with Tailwind CSS and Bootstrap, that presents role-specific portal interfaces to end users. The API Layer is a Node.js/Express.js REST API server that handles request routing, enforces JWT authentication, and applies role guards. The Data Layer uses MongoDB 7.0 via Mongoose ODM across nine collections. The Integration Layer connects to Cloudinary for cloud file storage and Gmail SMTP via Nodemailer for transactional email. Tiers communicate exclusively through well-defined interfaces—REST/JSON between client and API, Mongoose queries between API and data, and HTTP SDK calls between the API and external services.

CLIENT LAYER — React.js + Vite + Tailwind CSS
↕ REST API (JSON / HTTPS)
API LAYER — Node.js + Express.js | JWT Auth | Role Guard | Multer
↕ Mongoose ODM
DATA LAYER — MongoDB | Users · Jobs · Companies · Applications · Notifications
↕ HTTP SDK
INTEGRATION LAYER — Cloudinary (Files) | Gmail SMTP / Nodemailer (Email)

Figure 1: RUPMS Four-Tier Architecture

3.2. Database Design

The MongoDB schema comprises nine collections: Users, Jobs, Companies, Applications, Interviews, OfferLetters, Notifications, ActivityLogs, and Notices. The Users collection serves as a unified identity registry across all roles, using a single document structure with role-specific nested sub-documents (studentProfile, tpoProfile, managementProfile) to avoid cross-collection joins on authentication. The Jobs collection embeds an applicants array within each job document, enabling O(1) applicant count retrieval and atomic dual-write updates when a student applies. This design trades some normalisation for significant read performance gains on dashboard queries.

Collection	Key Fields	Design Note
Users	email, role, password (bcrypt), token, stu	dUenniftiPerdoifdilentity store; role sub-documents avoid joins
Jobs	jobTitle, salary, company (ref), deadline,	aEpmpbliecadndtesd[]applicants array for atomic apply operation
Companies	companyName, location, difficulty (enum	)Referenced by Jobs; cascade delete via Mongoose pre-hook
Applications	studentId (ref), jobId (ref), status (6-state	Setnautums) machine: Applied□Shortlisted□Interviewed□Selected
Notices	title, content (HTML), sender (ref), isActi	vJeodit rich-text HTML stored; filtered by isActive flag

Table 1: Key MongoDB Collection Schemas

3.3. Authentication and Role-Based Access Control

Authentication relies on the JSON Web Token (JWT) standard. At login, the server signs a token with a 256-bit HS256 secret, embedding the user's MongoDB ObjectId as the sole payload claim. The token is returned to the client and stored in localStorage, then attached as a Bearer token in the Authorization header of every protected request. The authenticateToken middleware extracts and verifies the token, queries the Users collection to confirm that the user's stored token field matches the presented value, and attaches the full user document to req.user. This server-side token binding ensures that logout—which clears the stored token field—immediately invalidates the session, regardless of token expiry.

Role enforcement on the frontend is implemented through the ProtectedRoute component, which decodes the JWT using jwt-decode, compares the embedded role against the allowedRoles prop of the route, and redirects non-matching users to the appropriate login page. This dual enforcement—middleware on the backend and ProtectedRoute on the frontend—means that neither direct URL access nor API calls can bypass the role boundary. Passwords are stored as bcrypt hashes with cost factor 10; no plaintext credential is retained at any persistence layer.

3.4. API Design

The backend exposes a RESTful API organised into six route namespaces: /student, /tpo, /management,

/admin, /user, and /company. Each namespace maps to a role domain or shared resource. Table 2 summarises representative endpoints from each namespace.

Endpoint	Method	Auth	Description
POST /student/login	POST	No	Authenticate student; return signed JWT
PUT /student/job/:jobId/:studentId	PUT	No	Atomic dual-write job application

POST /tpo/post-job	POST	Yes (TPO)	Create new job posting with company ref
PUT /tpo/post-job/:jobId	PUT	Yes (TPO)	Update job details or applicant status
GET /management/reports	GET	Yes (Mgmt)	Aggregate analytics across all collections
POST /admin/create-user	POST	Yes (Super)	Create any-role user; dispatch credential email
GET /user/all-jobs	GET	Yes	Fetch all active postings with company info
PUT /user/update-password	PUT	Yes	Re-hash and store updated password
POST /company/add	POST	Yes (TPO)	Register new company in master directory
DELETE /company/:id	DELETE	Yes (TPO)	Delete company; cascade to jobs and applicants

Table 2: Representative REST API Endpoints

3.5. Frontend Architecture

The React.js 19 frontend is structured around five domain areas: public routes (landing page, login forms), and four authenticated portal areas—Student, TPO, Management, and Super Admin—each prefixed with a distinct URL namespace. The App.jsx root component defines the complete routing tree using React Router v7, with each authenticated domain wrapped in a ProtectedRoute instance. Application-level authentication state is distributed through a UserContext provider using the Context API, avoiding prop-drilling while maintaining a single source of truth.

Code splitting is implemented via React.lazy and Suspense on all page-level imports, ensuring that unauthenticated visitors receive a bundle that excludes dashboard and admin-portal code. The management analytics dashboard renders four Chart.js visualisations in a responsive two-by-two grid: a bar chart of monthly placements, a doughnut chart of department-wise distribution, a horizontal bar chart of company-wise hires, and a line chart of salary package trends across the past six cycles.

3.6. File Upload and Email Integration

File uploads—student resumes (PDF), profile photographs, and TPO-issued offer letters—are handled by Multer middleware configured with the multer-storage-cloudinary adapter. Uploaded files are streamed directly to the Cloudinary CDN without touching the local filesystem, and the resulting public URL is persisted to the appropriate MongoDB document field. MIME-type validation at the middleware level restricts accepted file types to application/pdf for documents and image/* for photographs.

Transactional email notifications are dispatched through a Nodemailer transporter connected to Gmail SMTP via an App Password credential. A sendMail utility function wraps the transporter,

accepting recipient address, subject, and an HTML body string. The emailTemplates.js module generates styled HTML credential-notification emails for newly created administrator accounts, embedding the assigned role, login email, and auto-generated initial password.

IV. IMPLEMENTATION

4.1. Technology Stack

Layer	Technology	Version
Runtime	Node.js	v20.x LTS
Web Framework	Express.js	v5.2.1
Database	MongoDB + Mongoose ODM	7.0 / 9.1.5
Frontend	React.js + Vite	19.2.3 / 7.3.1
Styling	Tailwind CSS + Bootstrap	3.4.19 + 5.3.8
Auth	jsonwebtoken + bcrypt	9.0.3 / 6.0.0
File Upload	Multer + multer-storage-cloudinary	2.0.2 / 4.0.0
Email	Nodemailer	7.0.12
Charts	Chart.js via react-chartjs-2	4.x
Containerisation	Docker + Kubernetes (kind)	24.x / 1.29
HTTP Client	Axios	1.13.2

Table 3: Technology Stack

4.1. Key Implementation Details

Dual-Write Job Application: When a student submits a job application, the apply-job controller executes two MongoDB updates atomically: a $push to the Job document's applicants array and a simultaneous $push to the student's studentProfile.appliedJobs array. This bidirectional reference ensures that both job-centric queries (how many applicants does job X have?) and student-centric queries (which jobs has student Y applied to?) resolve in a single collection scan without cross-collection joins.

Cascade Deletion via Mongoose Pre-Hooks: Company deletion triggers a pre-deleteOne Mongoose middleware that queries and removes all associated Job documents. Each Job deletion in turn fires its own pre-hook, which removes the corresponding jobId entries from the appliedJobs arrays of all students who had applied. This middleware chain propagates referential integrity through the document graph without requiring multi-step logic in the controller.

Code Splitting for Bundle Optimisation: All page-level React components are imported using React.lazy wrapped in Suspense boundaries. This defers the loading of dashboard and admin-panel code until the user navigates to the relevant route, reducing the initial JavaScript bundle delivered to unauthenticated users to 187 KB (gzip-compressed) and improving First Contentful Paint on constrained network connections.

Docker and Kubernetes Deployment: The system is containerised via a docker-compose.yaml

that orchestrates the Node.js API container and a MongoDB container with a persistent volume. Kubernetes manifests are provided for cloud deployment, enabling horizontal pod autoscaling of the API layer independently of the database tier. This separation of concerns allows the system to handle traffic spikes without database disruption.

V. EVALUATION AND RESULTS

5.1.Functional Testing

Ten structured test scenarios were designed to verify system behaviour across authentication, job management, application lifecycle, file handling, email dispatch, analytics rendering, protected route enforcement, and cascade deletion. Each scenario was executed on the deployed system and validated by inspecting MongoDB document state, the Cloudinary media library, the test Gmail inbox, and the frontend UI. All ten scenarios produced the expected outcome.

ID	Scenario	Result
TC-01	Student login with valid credentials returns JWT and loads dashboard	PASS
TC-02	Student login with incorrect password returns 400 error	PASS
TC-03	TPO creates a new job posting; listing updates immediately	PASS
TC-04	Student applies to job; both Job and User documents updated	PASS
TC-05	TPO updates status to Shortlisted; student dashboard reflects change	PASS
TC-06	Student uploads resume PDF; Cloudinary URL stored in user record	PASS
TC-07	Management admin navigates to analytics; all four Chart.js charts render	PASS
TC-08	Super admin creates TPO account; credential email delivered in under 10s	PASS
TC-09	Non-student attempts direct access to /tpo/dashboard; redirected to login	PASS
TC-10	TPO deletes company; cascades to 3 associated jobs and student records	PASS

Table 4: Functional Test Scenario Results (All 10/10 PASS)

5.2. Performance Benchmarking

Performance was assessed on an AWS EC2 t2.micro instance (1 vCPU, 1 GB RAM) under a simulated load of

20 concurrent users. API response latencies were recorded over a 500-request sample for the ten most frequently called endpoints. Frontend metrics were captured using the Lighthouse audit tool.

Metric	Measurement
Median response time — GET /user/detail	58 ms
Median response time — GET /student/job-listings	83 ms
Median response time — GET /management/reports (aggregation)	142 ms

95th percentile API response time	310 ms
Landing page — First Contentful Paint (simulated 4G)	1.2 s
Landing page — Time to Interactive (simulated 4G)	2.1 s
Initial JS bundle size (gzip)	187 KB
Average EC2 CPU utilisation (48-hour window)	2.8 %
Peak EC2 CPU utilisation (20 concurrent users)	31.4 %
Node.js process memory (stable state)	248 MB
System uptime across 48-hour evaluation period	100 %

Table 5: Performance Benchmark Results

The results demonstrate that RUPMS operates comfortably within the resource constraints of a free-tier cloud instance while maintaining sub-150ms median latency for all dashboard-critical endpoints. The 100% uptime across the 48-hour evaluation window confirms operational stability for continuous institutional deployment.

5.3. Comparative Analysis

Approach	4-Role RBAC	Live Analytics	Cloud Files	Email Notif.	Mobile-Ready	Custom Brand
Manual Spreadsheet	No	No	No	No	No	N/A
Commercial Portal	Partial	Limited	Yes	Yes	Partial	No
PHP/MySQL (Kumar 2020)	Yes	No	No	No	No	Yes
MEAN Stack (Janikow 2018)	Yes	Partial	No	No	No	Yes
RUPMS (This Work)	Yes (4 roles)	Yes	Yes	Yes	Yes	Yes

Table 6: Comparative Analysis Against Related Approaches

VI. CONCLUSION AND FUTURE WORK

This paper has presented RUPMS, a comprehensive, role-stratified placement management platform developed on the MERN stack for Raffles University, Neemrana. The system consolidates every phase of the campus recruitment lifecycle—student profile management, job posting, application tracking, offer distribution, and institutional analytics—within a single, secure, cloud-hosted application. Structured evaluation confirms 100% test-scenario pass rate, sub-150ms median API latency, a 187 KB initial bundle, and uninterrupted availability over a 48-hour window on a free-tier cloud instance.

The primary contribution of this work is the demonstration that a custom, institutionally tailored MERN application can fully replace fragmented manual placement workflows while remaining

deployable on minimal infrastructure. The four-tier architecture, Mongoose pre-hook cascade deletion, dual-write application model, and Context API state management collectively represent a reusable pattern for multi-role educational web applications.

Future enhancements identified include: (1) real-time push notifications via Socket.io WebSockets; (2) a companion React Native mobile application with Firebase Cloud Messaging support; (3) NLP-driven resume parsing for automated candidate-job matching; (4) integration with the university's Student Information System for automatic record synchronisation; (5) a machine learning layer for predictive placement probability scoring; and (6) multi-campus schema extensions for organisation-wide deployment.

REFERENCES

[1] S. Rao and P. Mishra, "Campus placement management practices in Indian technical universities: A comparative study," *International Journal of Educational Technology*, vol. 4, no. 2, pp. 45-62, 2016.

[2] A. Sharma and R. Tiwari, "Challenges in manual placement management: A case study of two engineering colleges in Maharashtra," *Journal of Engineering Education Research*, vol. 7, no. 1, pp. 28-41, 2018.

[3] M. Gupta and N. Mehta, "Student adoption barriers in commercial placement software: A multi-college survey," *Procedia Computer Science*, vol. 152, pp. 178-185, 2019.

[4] A. Kumar, P. Singh, and R. Joshi, "Design and implementation of a PHP/MySQL college placement portal," *International Journal of Advanced Computer Science and Applications*, vol. 11, no. 5, pp. 203-212, 2020.

[5] C. Janikow and T. Smith, "Multi-role educational management system on the MEAN stack: Architecture and lessons learned," in *Proceedings of the 2018 ACM SIGCSE Symposium*, pp. 341-346, ACM, 2018.

[6] H. Alhumaidan, K. Salah, and A. AlJarallah, "Comparative evaluation of MERN, PHP-Laravel, and Python-Django for educational management systems," *IEEE Access*, vol. 10, pp. 58341-58357, 2022.

[7] V. Singh and H. Kaur, "Cloudinary-based document management in university web applications," *International Journal of Computer Applications*, vol. 175, no. 22, pp. 30-36, 2020.

[8] S. Dubey and P. Verma, "Nodemailer with Gmail SMTP for transactional email in Node.js: Implementation patterns and security considerations," *Indian Journal of Computer Science*, vol. 6, no. 4, pp. 55-62, 2021.

[9] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *IEEE Computer*, vol. 29, no. 2, pp. 38-47, 1996.

[10] R. Patel and K. Joshi, "MERN stack adoption in final-year B.Tech projects: A three-university survey," *Journal of Engineering Education Technology*, vol. 9, no. 3, pp. 88-102, 2021.

[11] Mongoose, *Mongoose 9.x Documentation*. [Online]. Available: https://mongoosejs.com/docs/
[12] React, *React 19 Documentation*. [Online]. Available: https://react.dev/
[13] Docker, *Docker Documentation*. [Online]. Available: https://docs.docker.com/
[14] Kubernetes, *Kubernetes Documentation*. [Online]. Available: https://kubernetes.io/docs/
[15] Chart.js, *Chart.js Documentation*. [Online]. Available: https://www.chartjs.org/