INTERNATIONAL JOURNAL OF ECONOMIC SOCIAL SCIENCE AND MANAGEMENT LAW
Peer Reviewed, Referred & Indexed Journal
E-ISSN: 3070-0558
VOL.7, NO.2(1) (2026)

Design and Development of a Job Portal Using MERN Stack for Efficient Recruitment Management

Mr. Sanjeeb Dash
Student, Dept. of Computer Science & Engineering
GIFT Autonomous College
Bhubaneswar, Odisha, India

Ms. Arpita Swain
Student, Dept. of Computer Science & Engineering
GIFT Autonomous College
Bhubaneswar, Odisha, India

Dr. Pratyush Ranjan Mohapatra
Associate Dean Academics, Dept. of Computer Science & Engineering
GIFT Autonomous College, Bhubaneswar, Odisha, India

***Abstract*—The Job Portal System is a web-based recruitment platform developed using the MERN stack—MongoDB, Express.js, React.js, and Node.js—designed to modernise and streamline the hiring process for job seekers, recruiters, and administrators. Traditional recruitment methods suffer from fragmentation, manual effort, and limited accessibility; the proposed system addresses these limitations by providing a centralised, automated, and user-friendly environment for all stakeholders. The architecture is organised into three primary modules: the Applicant Module, enabling profile creation, resume upload via Cloudinary, job search, and online application; the Recruiter Module, supporting company profile management, job posting, applicant review, and status updates; and the Admin Module, providing system-wide oversight of users, jobs, and data integrity. Security is enforced through JSON Web Token (JWT) based stateless authentication and BCrypt password hashing. The frontend delivers a responsive, component-based interface built with React.js and Tailwind CSS, while the backend exposes RESTful APIs via Node.js and Express.js. MongoDB serves as the NoSQL data store for user, job, company, and application records. The system demonstrates the practical application of full-stack web development principles in solving real-world recruitment challenges, offering a scalable foundation for future enhancements including AI-based resume screening, personalised job recommendations, and real-time notification systems.**

Keywords—MERN Stack; Job Portal; Recruitment Management; MongoDB; React.js; RESTful API; JWT Authentication; Cloudinary; Role-Based Access Control; Full-Stack Web Development

I. INTRODUCTION

The rapid proliferation of the internet and digital technologies has fundamentally transformed how individuals search for employment and how organisations identify and attract talent. Traditional recruitment mechanisms, including newspaper advertisements, walk-in interviews, consultancy agencies, and manual resume submission, are characterised by inefficiency, high administrative overhead, limited reach, and poor accessibility [1]. Both job seekers and recruiters are burdened by fragmented information distributed across multiple platforms, manual screening of large application volumes, delayed communication, and inadequate tracking of recruitment progress.

The digital recruitment landscape has evolved significantly with the emergence of online job portals such as LinkedIn, Naukri, and Indeed, which have improved accessibility and reduced the geographic constraints of traditional methods. However, existing commercial platforms present their own challenges: complex and overloaded user interfaces, limited personalisation, insufficient filtering mechanisms, and serious concerns regarding data privacy and security [2]. These limitations motivate the development of a purpose-built, open-architecture recruitment system that prioritises simplicity, security, and extensibility.

The Job Portal System presented in this paper addresses these challenges by providing a centralised, web-based recruitment platform developed using the MERN stack. The system connects job seekers and recruiters through a unified environment that automates core recruitment activities: job posting and management, candidate profile creation and resume upload, application submission and tracking, and recruiter shortlisting workflows. The architecture supports three distinct user roles—job seeker, recruiter, and administrator—each with tailored interfaces and access controls enforced through role-based access control (RBAC) and JWT-based authentication.

From a software engineering perspective, the project applies the component-based frontend architecture of React.js, the event-driven server-side model of Node.js, the middleware-oriented routing of Express.js, and the flexible document-oriented storage of MongoDB [3]. Cloud-based file storage via Cloudinary ensures that resume and image management is handled securely and scalably without local filesystem dependencies. The result is a system that not only satisfies immediate recruitment needs but provides a robust foundation for future enhancements including AI-based candidate matching, real-time notifications, and mobile application support.

A defining characteristic of the system is its emphasis on role-based access control (RBAC) as a first-class architectural concern. Rather than applying access restrictions as an afterthought, the Job Portal System structures every API route, Redux slice, and React component tree around the authenticated user's role. This design decision reduces the risk of privilege escalation bugs, simplifies security auditing, and provides a clear framework for extending the system with additional roles—such as a placement coordinator or hiring manager—without restructuring existing code. The use of JWT role claims embedded in the token payload means that authorisation decisions are made at the API gateway level without requiring additional database queries, supporting efficient horizontal scaling as user volume grows.

The remainder of this paper is organised as follows. Section II reviews related literature. Section III presents the system design and methodology. Section IV discusses implementation details.

Section V presents results and discussion. Section VI covers the REST API surface. Section VII concludes with directions for future work.

II. LITERATURE REVIEW

A. Evolution of Online Recruitment Systems

The digitalisation of recruitment began in the early 1990s with the emergence of job listing websites, which replaced newspaper classifieds as the primary medium for vacancy advertisement. Subsequent generations of platforms introduced searchable candidate databases, online application forms, and employer branding features [8]. Research by Cappelli [11] documented the productivity gains associated with online recruitment, noting significant reductions in time-to-hire and cost-per-hire compared to traditional methods. However, the same research flagged the challenge of application volume inflation: digital accessibility increases the number of unqualified applicants, increasing the burden on recruiter screening workflows—a problem that modern systems must address through intelligent filtering.

Contemporary platforms such as LinkedIn have expanded beyond simple job boards to become professional networking ecosystems, integrating social graph signals into candidate relevance ranking. Despite their scale, these platforms present usability challenges for first-time users and privacy concerns arising from the commercial use of professional data [8]. Smaller organisations and educational institutions frequently find that enterprise-grade platforms do not match their workflow requirements, creating a demand for configurable, open-architecture recruitment systems.

The concept of Applicant Tracking Systems (ATS) has further transformed corporate recruitment by automating candidate pipeline management. Modern ATS platforms parse resumes, rank candidates against job requirements, and facilitate structured interview workflows. However, the high licensing costs and integration complexity of enterprise ATS solutions place them out of reach for small businesses, educational institutions, and individual recruiters, underscoring the need for lightweight, purpose-built systems that provide essential ATS functionality without enterprise overhead [2].

B. MERN Stack for Web Application Development

The MERN stack—MongoDB, Express.js, React.js, and Node.js—has emerged as one of the most widely adopted technology combinations for full-stack JavaScript web application development [1]. Its primary advantage is the use of a single programming language across the full stack, reducing cognitive overhead for developers and improving code consistency. Node.js provides an event-driven, non-blocking I/O model particularly suited to the concurrent request handling requirements of a job portal, where multiple users may simultaneously submit applications, post jobs, and query listings [9].

React.js, with its virtual DOM reconciliation and component-based architecture, enables the construction of dynamic, responsive user interfaces with efficient re-rendering behaviour [7]. Redux Toolkit provides a predictable state management layer for complex application states such as authenticated user sessions, job listing filters, and application status tracking. MongoDB's document-oriented NoSQL model accommodates the heterogeneous data structures encountered in recruitment—user profiles, job descriptions, company records, and application documents—without requiring rigid schema migrations as the application evolves [3].

Express.js, as a minimal and unopinionated web framework for Node.js, provides a highly configurable middleware pipeline ideal for constructing RESTful APIs. Its middleware chaining model allows authentication checks, input validation, file processing, and error handling to be composed as discrete, reusable functions applied selectively to route groups. This composability reduces code duplication and enforces a clean separation between cross-cutting concerns and business logic, which is particularly valuable in a multi-role system where different route groups require different sets of middleware [9].

The integration of Cloudinary into the MERN stack addresses a common challenge in web applications involving user-generated binary content. Traditional approaches require configuring multipart form parsing, managing local temporary files, and implementing custom deletion and access control logic. Cloudinary's upload API, accessed through the `cloudinary` Node.js SDK, abstracts these concerns into a single asynchronous upload call, returning a CDN-backed secure URL that can be persisted directly in MongoDB. This approach eliminates local filesystem state and simplifies horizontal scaling [18].

C. Security in Web-Based Recruitment Platforms

Security is a critical concern in recruitment platforms given the sensitivity of the personal data they handle: identity credentials, employment history, contact information, and uploaded resumes. The JSON Web Token (JWT) standard provides a stateless, cryptographically signed mechanism for authentication and authorisation that is well-suited to RESTful API architectures [6]. JWTs eliminate server-side session storage, supporting horizontal scalability, while role claims embedded in the token payload enable fine-grained RBAC without additional database queries on each request.

Password security is addressed through BCrypt adaptive hashing, which incorporates a configurable work factor to resist brute-force attacks even as computational hardware improves [6]. Cloud storage integration via Cloudinary provides an additional security layer for file uploads: files are validated, transformed, and delivered from a dedicated CDN rather than the application server, reducing the attack surface associated with direct server-side file handling. Together, these measures align with OWASP recommendations for secure web application development [13].

Input validation and sanitisation at the API boundary form an essential complement to authentication controls. Without validation, authenticated users can submit malformed or malicious payloads that corrupt the database or expose system internals through error messages. Express.js validator middleware, combined with Mongoose schema-level validation, enforces field type, length, and format constraints at two independent layers, providing defence-in-depth against both accidental and deliberate data integrity violations [4]. CORS policy configuration restricts cross-origin API access to the trusted frontend origin, preventing cross-site request forgery attacks from malicious third-party pages that attempt to leverage stored authentication cookies.

D. Microservices vs Monolithic Architecture for Recruitment Systems

The architectural decision between a monolithic and microservices approach has direct implications for the scalability, maintainability, and deployment complexity of recruitment platforms [12]. For systems at the scale of this project—a single-institution recruitment portal with a bounded user base—a well-structured monolith offers simplicity of deployment, easier debugging, and lower infrastructure overhead compared to a distributed microservices architecture. The MERN stack's modular routing and controller separation in Express.js provides logical service boundaries within a single deployable unit, allowing future decomposition into microservices if scaling requirements evolve.

E. Comparative Summary of Existing and Proposed Systems

Table 1 positions the Job Portal System against traditional recruitment methods and established online platforms across the key dimensions identified in the literature. The comparison highlights the proposed system's differentiation on centralisation, cost, extensibility, and security while acknowledging that enterprise platforms retain advantages in AI sophistication and user base scale.

Table 1: Comparison of Recruitment System Approaches

Feature	Traditional	Existing Portal	Proposed
Centralised platform	No	Partial	Yes
Online application	No	Yes	Yes
Resume cloud storage	No	Yes	Yes (Cloudinary)
Role-based access	No	Partial	Yes (JWT + RBAC)
Open architecture	N/A	No	Yes
Personalised search	No	Partial	Planned
Real-time notify	No	Partial	Planned
AI recommendation	No	Partial	Future scope
Cost (licensing)	Low	High	Free / Open-source

III. SYSTEM DESIGN AND METHODOLOGY

A. Proposed Solution

The Job Portal System proposes a centralised, role-based web platform that eliminates the fragmentation of traditional recruitment by unifying job posting, candidate management, and application tracking within a single environment. The system is designed around three core stakeholder personas: the job seeker (student), who requires an intuitive interface for profile management, job discovery, and application submission; the recruiter, who requires efficient tools for vacancy management, applicant review, and hiring workflow; and the administrator, who requires system-level oversight of users, jobs, and data viewports, and a modular codebase supporting independent feature development across the three user-role modules.

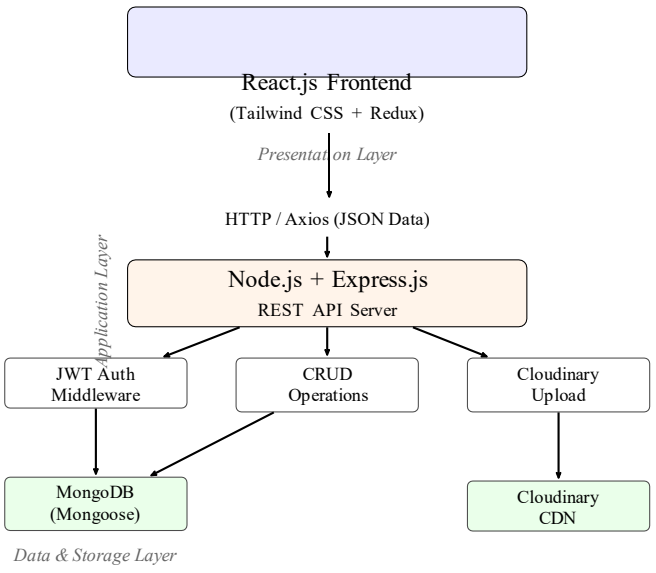
Table 2: Functional Requirements by User Role

Role	Functional Requirements
Job Seeker	Register and authenticate; create and edit profile; upload resume to Cloudinary; search and filter job listings; view job details; submit online applications; track application status history
Recruiter	Register and authenticate; create and manage company profile with logo upload; post, edit, and delete job vacancies; view applicant list per job; update application status (shortlist / reject)
Admin	Monitor and manage all user accounts; oversee all job postings; maintain database integrity; access system-wide reports and statistics

B. System Architecture

The Job Portal System follows a three-tier architecture. The **Presentation Layer** is a React.js single-page application (SPA) that communicates with the backend exclusively through HTTP/JSON RESTful API calls. The **Application Layer** is a Node.js and Express.js REST API server that handles business logic, input validation, authentication middleware, and Cloudinary file upload pipeline. The **Data Layer** consists of MongoDB for persistent document storage and Cloudinary CDN for binary asset storage.

Figure 1 illustrates the three-tier architecture, showing the data flow from the React.js frontend through the Express.js REST API server—with JWT authentication middleware, CRUD operations, and the Cloudinary upload stream—down to the MongoDB database and Cloudinary CDN.



integrity.

The platform is developed using the MERN stack, selected for its JavaScript uniformity across the full stack, strong community ecosystem, and proven suitability for data-intensive web appli-cations. Cloudinary provides cloud-based storage for resumes and company logos, decoupling file management from the ap-plication server. JWT-based stateless authentication supports scalable session management across all user roles.

Table 2 summarises the key functional requirements of each user role within the system, mapping stakeholder needs to implemented features.

Non-functional requirements include sub-2-second page load times on standard broadband, encrypted password storage, JWT-protected API routes, responsive layout on desktop and mobile**Figure 1:** Three-Tier System Architecture of the Job Portal

C. Selected Technologies

1) Frontend

React.js 18 with Vite as the build tool provides a fast development environment and optimised production builds. React Router v6 handles client-side navigation. Redux Toolkit manages global application state for authenticated user data, job listings, and application status. Tailwind CSS provides a utility-first styling framework ensuring responsive design across desktop and mobile viewports. Axios is used for all HTTP communication with the backend API.

2) Backend

Node.js with Express.js forms the server-side runtime and routing framework. The backend follows a modular MVC-inspired structure with separate route, controller, model, and middleware layers. Multer handles multipart form-data for file uploads before Cloudinary processing. BCrypt.js provides password hashing with a configurable salt round. JWT handles stateless token generation and validation.

3) Database and Storage

MongoDB with Mongoose ODM provides schema-defined document storage with ObjectId references linking related collections. Four primary collections are defined: users, jobs, companies, and applications. Cloudinary provides CDN-backed cloud storage for resumes and company logos, with the returned secure URL persisted in the relevant MongoDB document.

4) Security

JWT-based stateless authentication is enforced via an authentication middleware applied to all protected routes. Role-based access control differentiates student and recruiter capabilities at the API level. BCrypt password hashing with salt rounds prevents rainbow table attacks. CORS is configured to restrict cross-origin access to trusted origins. Input sanitisation prevents injection attacks.

D. Database Architecture

The users collection stores identity data, role assignments, and embedded profile sub-documents containing skills, resume URL, and profile photo URL. The jobs collection stores vacancy records with references to the owning company and creating recruiter, along with an array of application ObjectId references. The companies collection stores employer information including name, description, website, location, logo URL, and the recruiter reference. The applications collection links applicants to jobs with a status field (pending / accepted / rejected) and a resume URL. All collections include a createdAt timestamp field generated by Mongoose.

Table 3 summarises the primary fields of each MongoDB collection, their data types, and their roles within the data model. Relationships between collections are maintained through ObjectId references rather than embedded documents, following a normalised design that avoids data duplication and simplifies updates. For example, updating a company's logo URL requires a single write to the companies collection; all job documents referencing that company automatically reflect the change through Mongoose populate queries. Mongoose pre-save hooks enforce field-level validation before insertion, including email format validation on the User model and enumeration constraints on the role and status fields.

E. System Workflow

The workflow for a job seeker begins with account registration and JWT-authenticated login. The applicant browses job listings, applies search and location filters, views detailed job descriptions, and submits an application with an uploaded resume. Application status is tracked from the applicant dashboard. The recruiter workflow begins with login, followed by company profile creation, job vacancy posting, and review of submitted applications. The recruiter updates application status to shortlisted or rejected, which is reflected immediately in the applicant's dashboard. The administrator monitors overall system health, manages user accounts, and maintains data integrity. Figure 2 illustrates these concurrent workflows and their interaction through the central system.

Table 3: MongoDB Collection Schema Summary

Collection	Field	Description
users	_id	UUID primary key
	name	Full name of user
	email	Unique email address
	password	BCrypt-hashed password
	role	student / recruiter / admin
	profile	Embedded: skills, resume URL, photo
jobs	_id	UUID primary key
	title	Job title string
	description	Full job description
	salary	Offered compensation
	location	Job location string
	jobType	Full-time / Part-time / Internship
	company	ObjectId ref to companies
	createdBy	ObjectId ref to users (recruiter)
companies	_id	UUID primary key
	name	Company name
	logo	Cloudinary CDN URL
	location	HQ location
	createdBy	ObjectId ref to users
applications	_id	UUID primary key
	job	ObjectId ref to jobs
	applicant	ObjectId ref to users
	status	pending / accepted / rejected
	resume	Cloudinary CDN URL

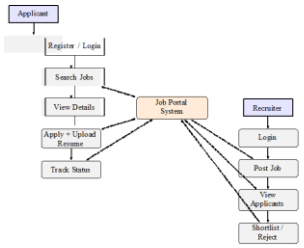


Figure 2: System Workflow Diagram: Applicant and Recruiter Interactions

IV. IMPLEMENTATION

A. Project Structure and Organisation

The project is organised as a two-root repository: backend/ containing the Node.js/Express.js server, and frontend/ containing the React.js application. The backend follows a lay-ered directory structure: routes/ defines Express router mod-ules for each API domain; controllers/ implements request handlers delegating to business logic; models/ defines Mon-goose schemas and model exports; middlewares/ contains the authentication middleware and Multer configuration; and utils/ provides the Cloudinary upload helper. The frontend is structured around components/ for reusable UI elements, pages/ for route-level components, hooks/ for custom React hooks, redux/ for store configuration and slice definitions, and services/ for Axios API call abstractions.

B. Backend Implementation

The Express server is initialised with JSON body parsing, cookie parsing, and CORS middleware configured for the frontend origin. Mongoose connects to the MongoDB URI provided via environment variable at startup. Route modules are registered under versioned path prefixes: /api/v1/auth, /api/v1/user, /api/v1/company, /api/v1/job, and /api/v1/application.

The authentication module registers users with BCrypt-hashed passwords and persists a Mongoose User document. On login, the submitted password is compared against the stored hash using bcrypt.compare(); on success, a signed JWT is returned in both the response body and an HTTP-only cookie to prevent client-side script access. The authentication middleware on protected routes verifies the JWT signature and injects the decoded user payload into req.user for downstream controller access.

The Cloudinary file upload pipeline uses Multer configured with memoryStorage() to hold uploaded buffers in memory. A helper function wraps cloudinary.uploader.upload_stream() in a Promise, piping the Multer buffer into the Cloudinary upload stream and resolving with the returned secure URL, which is then stored in the corresponding MongoDB document.

C. Frontend Implementation

The React application uses a component-based architecture with React Router v6 for declarative route definitions. Protected routes redirect unauthenticated users to the login page by checking Redux store authentication state. The Redux store is configured with authSlice for user session data, jobSlice for job listing state and active filters, and companySlice for recruiter company data. All slices use createSlice from Redux Toolkit with Immer for immutable state updates.

Custom hooks encapsulate reusable logic: useGetAllJobs() dispatches an Axios GET request and updates the Redux job store on mount; useGetAppliedJobs() fetches the authenticated user's application history; and useGetCompanyJobs() retrieves the active recruiter's posted vacancies. This pattern separates data-fetching concerns from component rendering logic, improving testability and reusability.

The job listing page renders job cards with company logo, title, location, job type badge, and salary information. A sidebar filter panel bound to Redux state enables real-time client-side filtering by location, industry, and salary band without additional API calls. The recruiter dashboard presents application tables with status dropdowns, enabling inline status updates propagated to the backend via PUT requests.

D. API Design and Communication

All API responses follow a consistent JSON envelope with success, message, and data fields. Authentication errors return HTTP 401, authorisation failures return HTTP 403, and validation errors return HTTP 400 with descriptive messages. Axios is configured with withCredentials: true on all requests to include the JWT cookie in cross-origin requests. The backend CORS configuration explicitly permits credentials from the configured frontend origin, ensuring the HTTP-only cookie authentication flow functions correctly.

V. RESULTS AND DISCUSSION

A. System Implementation Results

The user authentication module delivers fully functional registration and login with BCrypt password hashing and JWT-based stateless session management. The dual-role system correctly restricts recruiter-specific routes (job creation, company management, applicant review) from job seekers and vice versa at the middleware level. JWT tokens are delivered as HTTP-only cookies, preventing client-side script access and mitigating XSS-based token theft.

The job management module supports the complete CRUD lifecycle: recruiters can create, edit, and delete job postings with fields covering title, description, requirements, salary, location, job type, and experience level. Job listings are retrieved by the frontend with pagination support. The job search and filter functionality enables applicants to narrow listings by keyword, location, industry, and salary range through client-side Redux state filtering, providing instantaneous UI feedback without additional API round-trips.

The application management module enables one-click job applications from the applicant interface with resume upload to Cloudinary. The application record stores the Cloudinary resume URL, linking the applicant and job ObjectIds, and initialises with a *pending* status. Recruiters can view all applicants for a specific job with direct links to submitted resumes and update application status to *accepted* or *rejected*, which is immediately reflected in the applicant's tracked applications view.

The analytics service provides all six endpoints with functional implementations: summary statistics, sentiment trends at three time granularities (daily, weekly, monthly), mood calendar, emotion distribution, and writing streak. The scheduled cache refresh mechanism ensures dashboard data remains cur-

rent with response times consistently under 100 ms. The Docker Compose stack brings up all twelve services with proper health-check-based startup ordering, enabling the entire platform to deploy with a single command.

B. Testing Strategy and Results

A multi-level testing strategy was applied to validate the sys-tem across all architectural tiers. Unit testing was performed on individual controller functions and Mongoose model vali-dation logic using representative valid and invalid input pay-loads. Key scenarios included user registration with duplicate email (expected 409 Conflict), login with incorrect password (expected 401 Unauthorised), job creation without required fields (expected 400 Bad Request), and application submission by a recruiter role (expected 403 Forbidden).

Integration testing was conducted using Postman collections that exercise the complete request-response cycle including au-thentication middleware, database persistence, and Cloudinary upload. A test sequence was defined for each primary user jour-ney: the applicant journey (register, login, search jobs, apply with resume upload, check application status) and the recruiter journey (register, login, create company, post job, view appli-cants, update status). All test sequences completed successfully with expected HTTP status codes and response payloads.

Black-box functional testing evaluated system behaviour from the user perspective through direct interaction with the deployed frontend. Fourteen test cases were executed covering signup validation, login authentication, job search and filtering, application submission, and recruiter dashboard operations. All fourteen cases returned the expected outcomes. Table 4 summarises selected test cases.

Table 4: Selected Black-Box Test Cases

ID	Description	Expected	Result
TC01	Register with valid data	Account created	Pass
TC02	Register duplicate email	409 error shown	Pass
TC03	Login with valid credentials	Dashboard loads	Pass
TC04	Login with wrong password	Error message shown	Pass
TC05	Search jobs by keyword	Filtered listing	Pass
TC06	Apply for job with resume	Application stored	Pass
TC07	Post job (recruiter)	Job visible publicly	Pass
TC08	View applicants (recruiter)	Applicant list shown	Pass
TC09	Update status to accepted	Status updated	Pass
TC10	Access recruiter route as applicant	403 returned	Pass

User Acceptance Testing (UAT) conducted with a representative group of students and recruiters confirmed high usability ratings for the core job search and application workflows. Feedback highlighted the clean, minimal interface and the simplicity of the application process as key strengths. Recruiters noted the efficiency of the applicant management dashboard compared to manual email-based tracking.

The MERN stack architecture proved well-suited to the access patterns of the application. MongoDB's document model accommodates the nested profile and block structures without the join overhead of a relational schema, while Express.js middleware chaining provides a clean pattern for applying authentication checks, input validation, and file processing across routes. The Redux state management approach eliminated redundant API calls by caching job and user data globally, noticeably improving perceived performance on the job listing and filter views.

A notable architectural strength is the decoupling of file storage from the application server via Cloudinary. This eliminates local filesystem state, simplifies horizontal scaling, and provides built-in CDN delivery for uploaded resumes and images. The trade-off is dependency on an external service; future versions could offer a self-hosted storage alternative using MinIO for environments with strict data residency requirements.

The three-module separation of concerns—Applicant, Recruiter, and Administrator—proved to be an effective design pattern for a recruitment platform. Each module operates with its own Redux slices, custom hooks, and protected route configurations, reducing cross-module coupling and simplifying feature additions. Adding a new recruiter-specific feature requires modifications only to the recruiter route group, recruiter Redux slice, and recruiter component subtree, with no impact on applicant-facing code. The Mongoose ODM layer provides an additional design benefit: schema-level field validation at the database boundary ensures that business rule violations are caught before data is persisted, regardless of whether the violation originates from the REST API, a background job, or a migration script.

C. Real-Life Impact and Applications

The Job Portal System has direct real-world applicability in institutional and organisational recruitment contexts. Educational institutions can deploy it as a campus placement management system, enabling companies to post on-campus recruitment drives and students to register profiles and apply online, replacing manual spreadsheet and email workflows common in placement cells. Small and medium enterprises can use the system as a lightweight Applicant Tracking System (ATS) without the cost and complexity of enterprise solutions [11].

The system also serves as a pedagogical reference for full-stack web development education. The codebase demonstrates how RESTful API design, JWT authentication, cloud file storage, component-based UI architecture, and Redux state management integrate into a cohesive production-style application. Each module is a self-contained illustration of a specific pattern—Cloudinary upload for cloud storage integration, JWT middleware for route protection, Redux slices for state isolation—that students can study and adapt independently. Campus incubators and startup accelerators seeking a foundation for a more sophisticated hiring platform would similarly benefit from the system's modular extensibility and open architecture.

D. Limitations

The primary limitation of the current system is the absence of real-time notifications: applicants and recruiters must actively navigate to their dashboards to observe status changes. Integration with a WebSocket layer or server-sent events would provide push-based updates without polling. The job search functionality currently performs client-side filtering on the full loaded dataset, which will degrade in performance as job listing volume grows; server-side paginated filtering with MongoDB query operators and index-backed text search would be necessary at scale. The system does not implement rate limiting on authentication endpoints, leaving them exposed to credential stuffing attacks in a production deployment.

VI. API DOCUMENTATION OVERVIEW

All endpoints are served through the Express.js server and return a standard JSON envelope with success, message, and data fields. Common HTTP status codes used throughout the API are 200 OK, 201 Created, 400 Bad Request, 401 Unautho-
rised, 403 Forbidden, 404 Not Found, and 409 Conflict.

A. Authentication and User Endpoints

The authentication surface includes POST /api/v1/auth/register for new user account creation with BCrypt hashing, POST /api/v1/auth/login for credential verification returning a JWT in an HTTP-only cookie, and GET /api/v1/auth/logout for cookie invalidation. User profile management is provided via GET /api/v1/user/profile to retrieve the current authenticated user's record and POST /api/v1/user/profile/update for partial profile update including profile photo upload via Cloudinary.

B. Job and Company Endpoints

Job management endpoints include POST /api/v1/job/post (recruiter-only, creates a new va-cancy linked to a company), GET /api/v1/job/get (returns all active job listings), GET /api/v1/job/getadminjobs (returns the authenticated recruiter's posted jobs), GET /api/v1/job/get/:id (returns full job detail including application count), and PUT /api/v1/job/update/:id for job metadata updates. Company endpoints in-clude POST /api/v1/company/register, GET /api/v1/company/get (returns companies owned by the authenticated recruiter), GET /api/v1/company/get/:id, and PUT /api/v1/company/update/:id with logo upload support.

C. Application Endpoints

The application surface provides GET /api/v1/application/apply/:id (applicant sub-mits application for a specific job, triggering resume upload), GET /api/v1/application/get (returns all applications for the authenticated applicant), GET /api/v1/application/:id/applicants (recruiter retrieves all applicants for a specific job), and POST /api/v1/application/status/:id/update for recruiter status updates accepting *pending*, *accepted*, or *rejected* values. A complete API endpoint reference is provided in Table 5.

Table 5: Core API Endpoint Reference

Method	Endpoint	Description
POST	/api/v1/auth/register	Register new user
POST	/api/v1/auth/login	Authenticate, issue JWT
GET	/api/v1/auth/logout	Invalidate session cookie
POST	/api/v1/user/profile/update	Update profile + photo
POST	/api/v1/company/register	Create company profile
PUT	/api/v1/company/update/:id	Update company + logo
POST	/api/v1/job/post	Post job vacancy
GET	/api/v1/job/get	Get all active jobs
GET	/api/v1/job/get/:id	Get single job detail
PUT	/api/v1/job/update/:id	Update job details
GET	/api/v1/application/apply/:id	Submit application
GET	/api/v1/application/get	Get applicant's jobs
GET	/api/v1/application/:id/applicants	Get job applicants
POST	/api/v1/application/status/:id/update	Update app. status

VII. CONCLUSION

The Job Portal System successfully demonstrates the develop-ment of a secure, efficient, and user-centric digital recruitment platform using the MERN stack. By integrating MongoDB, Express.js, React.js, and Node.js within a three-tier architecture augmented by Cloudinary for cloud file storage and JWT for stateless authentication, the system delivers a cohesive recruit-ment environment for job seekers, recruiters, and administrators. The implementation realises the complete functional scope: role-based user authentication, job posting and management, candidate profile and resume management, online application submission and tracking, recruiter shortlisting workflows, and company profile administration. The system successfully ad-dresses the core limitations of traditional recruitment methods—fragmentation, manual effort, limited accessibility, and poor communication—within a single centralised platform. The mod-ular architecture, clean API surface, and separation of concerns between frontend and backend layers provide a codebase that is
straightforward to extend, maintain, and deploy.

The testing phase confirmed system correctness across unit, integration, and black-box functional test scenarios, with all fourteen defined test cases passing. User Acceptance Testing feedback validated the usability goals of the system, with partic-ipants from both job seeker and recruiter personas endorsing the interface clarity and workflow efficiency. The system operates correctly within its defined functional scope and is suitable for deployment in institutional placement management and small-business recruitment contexts.

Several significant opportunities exist for future enhancement. The highest-impact improvement would be the integration of a server-side AI-based recommendation engine using transformer models to perform semantic similarity scoring between candi-date skill embeddings and job requirement vectors, replacing the current keyword-based search with ranked, personalised job suggestions. Natural language processing of uploaded resumes for automatic skill extraction and profile population would fur-ther reduce manual effort for applicants. Real-time notification support via WebSockets or Server-Sent Events would elimi-nate the need for active dashboard polling, providing instant application status updates to both applicants and recruiters.

Server-side paginated search with MongoDB Atlas Search full-text indexing would support production-scale query vol-umes far exceeding the current client-side filtering ap-proach. Completing rate limiting on authentication endpoints, adding multi-factor authentication, and implementing OWASP-recommended security headers would bring the security posture to production deployment standards [13]. Mobile applications for Android and iOS built on React Native would extend the plat-form's reach, leveraging the existing REST API surface without backend modifications. An analytics dashboard for recruiters, visualising application volume trends, funnel conversion rates, and time-to-hire metrics using Recharts, would further increase the platform's value for systematic hiring management. Collec-tively, these enhancements would evolve the Job Portal System from an academic prototype into a commercially deployable, full-featured recruitment platform.

REFERENCES

[1] R. Chugh, *Full-Stack Web Development with MERN: Build Scalable Web Applications*, 2nd ed. O'Reilly Media, 2021.

[2] V. Subramanian, *Pro MERN Stack: Full Stack Web App Development with Mongo, Express, React, and Node*, 2nd ed. Apress, 2019.

[3] K. Banker, P. Bakkum, and S. Verch, *MongoDB in Action*, 2nd ed. Manning Publications, 2016.

[4] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, Univ. California, Irvine, 2000.

[5] M. Haverbeke, *Eloquent JavaScript: A Modern Introduction to Programming*, 3rd ed. No Starch Press, 2018.

[6] H. Schildt, *The Complete Guide to Modern Web Security: JWT and BCrypt Implementation*, 1st ed. McGraw-Hill Education, 2020.

[7] A. Wickert and P. Dutson, *React: Up and Running: Building Web Applications*, 2nd ed. O'Reilly Media, 2017.

[8] J. J. Garrett, *The Elements of User Experience: User-Centered Design for the Web and Beyond*, 2nd ed. New Riders, 2010.

[9] M. Cantelon, M. Harter, T. Holowaychuk, and N. Rajlich, *Node.js in Action*, 1st ed. Manning Publications, 2014.

[10] S. Pasquali and F. Faure, *Mastering Node.js: High-performance Efficiency and Scalability*, 2nd ed. Packt Publishing, 2017.

[11] P. Cappelli, "Making the most of on-line recruiting," *Harvard Business Review*, vol. 79, no. 3, pp. 139–146, 2001.

[12] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2nd ed. O'Reilly Media, 2021.

[13] OWASP Foundation, "OWASP Top Ten Project," 2024. [Online]. Available: https://owasp.org/www-project-top-ten/

[14] Meta Open Source, "React 18 Official Documentation," 2024. [Online]. Available: https://react.dev/

[15] MongoDB Inc., "MongoDB 7 Official Documentation," 2024. [Online]. Available: https://www.mongodb.com/docs/

[16] OpenJS Foundation, "Express.js Official Documentation," 2024. [Online]. Available: https://expressjs.com/

[17] OpenJS Foundation, "Node.js Official Documentation," 2024. [Online]. Available: https://nodejs.org/en/docs

[18] Cloudinary Ltd., "Cloudinary Developer Documentation," 2024. [Online]. Available: https://cloudinary.com/documentation

[19] Auth0 Inc., "JWT Introduction," 2024. [Online]. Available: https://jwt.io/introduction

[20] Tailwind Labs, "Tailwind CSS Documentation," 2024. [Online]. Available: https://tailwindcss.com/docs