Career Nest-A Modern Job Portal

Chintha.Manjusha
PG Scholar
Department of Computer Science
Sir C R Reddy College, Eluru
Andhra Pradesh, India-534005
Email: manikac652@gmail.com

SandhyaRani Katragadda
Professor
Department of Computer Science
Sir C R Reddy College, Eluru
Andhra Pradesh, India-534005
Email: sandhyakiran@sircrreddycollege.ac.in

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

Abstract*—CareerNest is a full-stack web-based job portal developed to unify job discovery, application submission, status tracking, recruitment administration, and communication within one responsive platform. The system is implemented using Next.js 16, TypeScript, Prisma ORM, SQLite, Zustand, Tailwind CSS, and shadcn/ui. Job seekers can create professional profiles, search vacancies through location, job-type, and keyword filters, upload PDF or DOCX resumes, submit cover letters, and monitor application status in real time. Administrators can create, update, close, and delete job postings, inspect candidate applications, modify application status, manage users and roles, and review email communication logs. Resume content is extracted through pdf-parse and mammoth, while SMTP-based email notifications and in-application alerts communicate submission and status events. The architecture combines a role-based presentation layer, Next.js API routes, JWT-based authentication, Prisma-mediated persistence, local file storage, and external communication services. The implemented interfaces demonstrate a complete recruitment workflow from job publication and discovery to application review and status monitoring. Because the project is a transactional web application and does not train a predictive model, model accuracy, precision, recall, and F1-score are not claimed. The resulting system provides a practical, maintainable, and scalable foundation for transparent digital recruitment.

Keywords*—Job portal, Next.js, TypeScript, Prisma ORM, role-based access control, resume parsing, application tracking, SMTP notifications.

I. INTRODUCTION

Online recruitment has transformed how candidates discover opportunities and how organizations obtain applications. Nevertheless, job seekers frequently move across disconnected portals, repeat profile information, upload resumes through inconsistent interfaces, and receive limited visibility after applying. Administrators also require separate tools for publishing vacancies, reviewing candidates, communicating decisions, and maintaining records. Such fragmentation increases effort and delays hiring decisions [1]– [4].

CareerNest addresses this problem through a unified job portal that supports both job seekers and administrators. Job seekers can create accounts, maintain professional profiles, browse available positions, filter jobs by keyword, location, and type, submit resumes and cover letters, and track every application from a personalized dashboard. Administrators receive a consolidated interface for job-posting management, application review, user administration, communication logs, and dashboard statistics.

The application uses a modern full-stack architecture. Next.js 16 provides the frontend and backend API environment, TypeScript improves type safety, Prisma ORM supplies a declarative data-access layer, and SQLite offers zero-configuration persistence. Zustand coordinates client-side state, Tailwind CSS and shadcn/ui provide responsive interfaces, and Framer Motion supports interactive transitions. Resume files in PDF and DOCX format are parsed through pdf-parse and mammoth. SMTP email and in-app notifications communicate important application events.

The main contribution is the integration of recruitment functions that are often separated across multiple systems. The developed application connects vacancy management, advanced job discovery, resume-supported application submission, status tracking, role-based administration, email communication, and persistent audit records. Unlike machine-learning recruitment studies, this project does not train a recommendation or classification model; its results are therefore reported through architecture, functional implementation, and completed workflow interfaces rather than fabricated predictive metrics.

II. RELATED WORK

Early online job boards primarily digitized vacancy advertisements and resume submission. Later platforms introduced professional networking, employer branding, search filters, and candidate databases [1], [2]. Contemporary systems such as LinkedIn, Indeed, and Naukri provide extensive job inventories, but candidates may still experience fragmented application procedures and limited cross-stage visibility. Research on e-recruitment has similarly identified usability, information quality, trust, and timely communication as important determinants of adoption [3]–[7].

Web engineering research has emphasized component-based interfaces, server-side rendering, type-safe development, and API-oriented architecture for maintainable applications [8]–[12]. React and Next.js support reusable interface construction and optimized rendering, while TypeScript reduces runtime defects through static type checking. Object-relational mapping tools such as Prisma reduce repetitive database code and provide schema-driven migrations. Lightweight databases are suitable for prototypes and small deployments, while provider-independent schemas simplify later migration to PostgreSQL or MySQL.

Recruitment portals must also protect credentials, uploaded resumes, and application records. Password hashing, token-based sessions, protected routes, server-side validation, and least-privilege authorization are commonly recommended [13]–[17]. Notification studies indicate that immediate acknowledgements and status updates reduce uncertainty during digital transactions [18], [19]. Resume-processing systems frequently support PDF and DOCX extraction so that employers can access both the original file and machine-readable content [20]–[22].

Existing academic prototypes often focus on one function, such as job posting, resume upload, or candidate search. CareerNest combines these functions with application-state management, in-app notifications, automated email, user-role administration, email logs, and responsive dashboards. The system therefore emphasizes operational completeness and traceability rather than a narrow algorithmic component.

III. MATERIALS AND METHODS

A) System Architecture

CareerNest follows a layered architecture in which the Next.js presentation layer communicates with API routes that implement authentication, authorization, business rules, file processing, and communication. Prisma ORM mediates access to SQLite, while uploaded resumes are stored in the local file system and referenced by database records. Nodemailer and SMTP deliver event-driven emails. The original architecture from the project document is shown in Fig. 1.

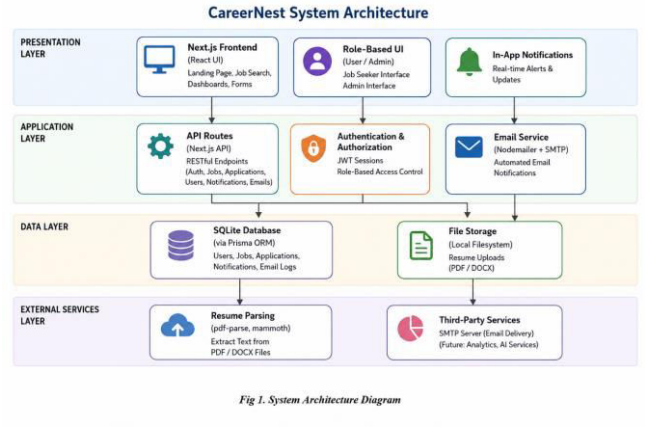

Fig. 1. System Architecture

The presentation layer contains public landing pages, job-seeker dashboards, administrator views, forms, and the notification panel. The application layer exposes REST-style routes for authentication, jobs, applications, users, notifications, and email logs. The data layer stores users, jobs, applications, notifications, and communication records. This separation reduces coupling and allows the database or file-storage implementation to be changed without redesigning the complete interface.

B) Role-Based Access and Authentication

The system distinguishes ADMIN and USER roles. Registration and login create authenticated sessions protected by JWT tokens stored in HTTP-only cookies. Server-side route checks verify both identity and role before protected operations are executed. Job seekers can update their profiles, browse vacancies, apply, and view only their own records. Administrators can create and manage jobs, review all applications, update application status, manage accounts, and inspect communication logs.

C) Job and Application Management

A job record stores title, company, location, employment type, salary, description, requirements, experience, education, skills, status, and the administrator who published it. Status values such as ACTIVE, CLOSED, and DRAFT control visibility. Search and filtering are performed over title, location, type, and keywords so that candidates can narrow the vacancy list without navigating multiple pages.

An application links one user with one job and stores the cover letter, resume path, creation time, and current status. A unique constraint prevents duplicate applications for the same user-job pair. The application lifecycle includes PENDING, REVIEWED, SHORTLISTED, REJECTED, and HIRED. Every status transition is persisted and reflected in the job-seeker dashboard, providing continuous visibility after submission.

D) Resume Processing and Notifications

The upload component accepts PDF and DOCX resumes. PDF text is extracted using pdf-parse, while DOCX content is processed using mammoth. File type, size, and ownership are validated before storage. The original resume remains available for administrator review, and the extracted content supports later search or extension toward skill matching without requiring a change to the submission workflow.

Two communication channels are integrated. Nodemailer sends SMTP messages for application submission and status changes, and the in-app notification model stores alerts with read/unread state and optional navigation links. EmailLog records recipient, subject, message type, delivery status, and any error, creating a useful audit trail for recruitment communication.

E) Database Design and Technology Stack

The Prisma schema defines five central models: User, Job, Application, Notification, and EmailLog. User maintains one-to-many relations with posted jobs, submitted applications, and notifications. Job maintains applications and references its creator. Application forms the candidate-vacancy relationship, while Notification and EmailLog support communication and auditability. SQLite is used for straightforward deployment, and Prisma permits later migration to a server database.

The interface is implemented with React components, Next.js 16, TypeScript, Zustand, Tailwind CSS 4, shadcn/ui, and Framer Motion. API handlers execute input validation and database operations. Bcrypt-based password hashing, protected routes, and HTTP-only session cookies support security. The modular component structure separates user, administrator, shared, and public-facing interfaces.

F) API Design and Data Validation

The backend is organized around resource-oriented API routes for authentication, jobs, applications, users, notifications, and email logs. Each route performs request parsing, validation, authorization, database access, and response generation. Create and update operations reject incomplete or inconsistent payloads before they reach the database. Job creation verifies mandatory fields and accepted status values, while application submission confirms that the vacancy exists, remains active, and has not already received an application from the same user. These controls prevent duplicate or orphaned records and provide predictable error messages to the interface.

Prisma transactions and relation constraints preserve consistency among user, job, and application records. The application route stores the file path only after upload validation succeeds, reducing the possibility of database entries that refer to missing files. Administrative status changes are checked against the supported lifecycle states, and every successful update refreshes the client-side store so that the interface reflects the persisted value. This request-response discipline is important because the portal performs several related operations across database, file, and notification services.

G) Interface and State Management

Zustand provides a centralized store for authentication state, job lists, filters, application records, notifications, and current view selection. Actions encapsulate asynchronous Fetch API calls and update the store after successful responses. This structure reduces prop drilling and allows shared components, including the navigation bar and notification panel, to access consistent state. Client-side

rendering is combined with server-side API processing, producing a responsive single-page experience without exposing database operations in the browser.

The interface adopts reusable cards, forms, tables, badges, dialogs, and dashboard widgets. Tailwind CSS utilities provide responsive layouts and consistent spacing, while shadcn/ui supplies accessible component foundations. Framer Motion is used selectively for transitions and interaction feedback. Validation messages appear close to the affected fields, status badges use distinct labels, and primary actions remain visible across common screen sizes. These design decisions support rapid task completion for both candidates and administrators.

H) Security, Privacy, and Deployment

Passwords are hashed before persistence, and authenticated sessions use signed tokens delivered through HTTP-only cookies. Protected API routes verify the session on every sensitive request rather than relying only on client-side navigation. Role checks prevent job seekers from accessing administrative resources, and object-level checks restrict candidates to their own applications, profile, and notifications. Input sanitization, accepted-file controls, and server-side validation reduce exposure to injection, unauthorized upload, and parameter-tampering attacks.

Resume files and personal profiles contain sensitive employment information. Production deployment should therefore use encrypted transport, randomized file names, access-controlled object storage, backup policies, retention rules, and auditable administrator actions. SMTP credentials and token secrets must be stored in environment variables rather than source code. The SQLite prototype is appropriate for demonstration; a multi-user deployment should migrate to PostgreSQL or MySQL, use cloud file storage, and place the application behind a reverse proxy with rate limiting and monitoring.

IV. IMPLEMENTATION AND RESULTS

A) Job Discovery and Filtering

The implemented job-discovery interface presents active vacancies as structured cards and provides filters for search text, location, and job type. Fig. 2 shows the central job-seeker workflow. Each card exposes essential vacancy information and opens the corresponding details and application action. The interface updates results without requiring a full-page reload, consistent with the single-page interaction design.

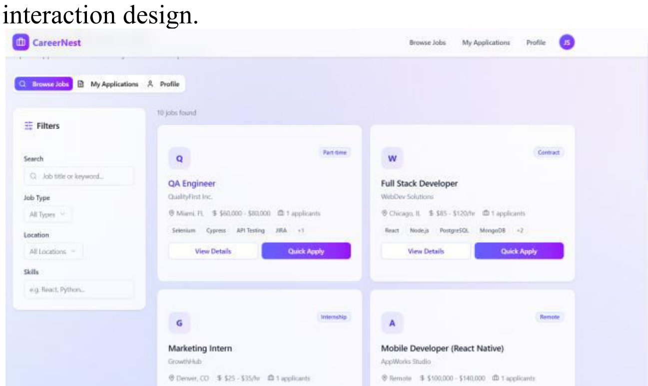

Fig. 2. Job discovery and filtering interface.

B) Application Submission and Tracking

After selecting a position, the user submits a cover letter and an accepted resume file. The API verifies authentication, vacancy status, file format, and duplicate-application constraints before creating the record. A successful submission generates an in-app notification and attempts an SMTP acknowledgement. The My Applications page in Fig. 3 consolidates every submission with company, location, applied date, and current status.

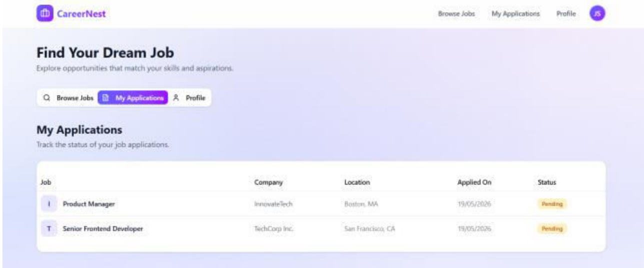

Fig. 3. Real-time job application tracking interface.

Status labels allow candidates to distinguish pending, reviewed, shortlisted, rejected, and hired applications. Because the state is read from the persistent application record, the dashboard remains synchronized with administrator updates. This directly addresses the limited post-submission visibility identified in conventional recruitment workflows.

C) Administrative Monitoring

The administrator dashboard summarizes users, jobs, applications, status distribution, notifications, and communication activity. Fig. 4 shows the management overview. The dashboard provides direct access to recent records and reduces the need to inspect individual tables before identifying pending recruitment work.

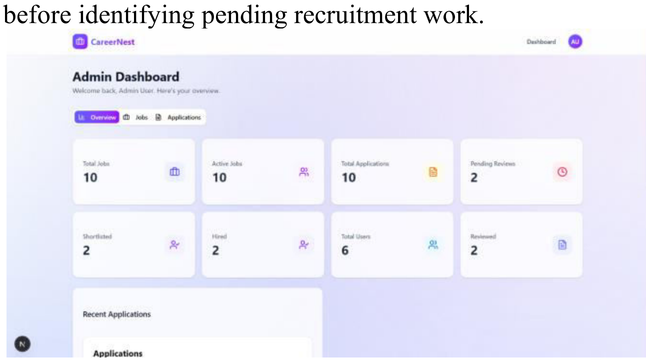

Fig. 4. CareerNest administrator dashboard.

D) Application Review and Status Updating

Administrators review candidate details, job information, uploaded resumes, and current status through the application-management interface shown in Fig. 5. Status controls update the persistent record and initiate candidate notification. The page supports filtering and centralized review across multiple vacancies, enabling a consistent workflow from initial submission to final hiring decision.

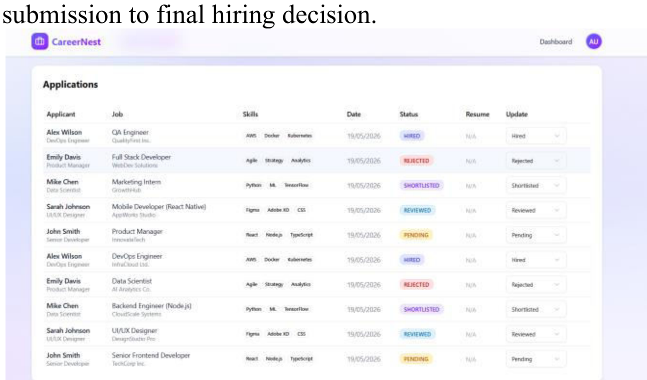

Fig. 5. Administrative application-review and status-management interface.

E) Operational Recruitment Indicators

The administrator dashboard stores the numbers of published jobs, submitted applications, shortlisted candidates, and hired candidates. These records support operational indicators without implying predictive-model performance. Application density is defined as:

$$D_a = \frac{N_{app}}{N_{jobs}} \quad (1)$$

where N_app is the total number of applications and N_jobs is the number of published jobs. The shortlist rate is calculated as:

$$R_s = \frac{N_{shortlisted} \times 100\%}{N_{app}} \quad (2)$$

The hiring rate represents the proportion of submitted applications that reach the hired state:

$$R_h = \frac{N_{hired} \times 100\%}{N_{app}} \quad (3)$$

These measures are obtained directly from the persistent job and application records displayed in the administrative dashboard. They summarize workflow activity and are not accuracy, precision, recall, or F1-score measurements.

F) Result Analysis

The screenshots demonstrate the implemented end-to-end sequence: administrators publish active jobs; job seekers discover and filter vacancies; candidates submit resumes and cover letters; applications are stored with an initial status; administrators inspect submissions and update decisions; and candidates observe the updated state through their dashboards. Email and in-app notifications provide communication at important transitions, while database and communication logs preserve traceability.

No predictive model is implemented in the documented project. Consequently, Accuracy, Precision, Recall, F1-score, ROC curves, and model-comparison graphs are not applicable and are not invented. The result is instead evaluated as a completed information-system workflow. The application demonstrates role separation, protected operations, multi-format resume processing, duplicate-application prevention, job filtering, persistent status tracking, and administrator control.

The implementation is suitable for academic demonstrations, departmental placement portals, and small recruitment environments. SQLite and local file storage simplify installation, while the Prisma-based schema and layered architecture permit migration to PostgreSQL, cloud object storage, and distributed deployment. The most important limitation is that recruitment scale, concurrent-user load, email-service reliability, and accessibility were not reported through a controlled benchmark. These areas require formal testing before production adoption.

G) Functional Coverage and Practical Evaluation

The project report documents the major functional requirements and corresponding interfaces rather than a machine-learning experiment. Functional coverage includes account registration, login, role enforcement, profile maintenance, job creation and editing, job browsing, search filters, application submission, PDF and DOCX resume handling, duplicate-application prevention, application-state management, user administration, notifications, email delivery, and communication logging. The implemented screens provide direct evidence that the core user and administrator journeys are connected through persistent records.

The result workflow can be interpreted through four operational qualities. Completeness concerns whether every recruitment stage is available in one system. Consistency concerns whether the same job, user, and status information appears across dashboards and management pages. Traceability concerns whether an administrator can follow an application from submission through later decisions and communication. Usability concerns whether common actions are exposed through clear filters, cards, forms, and status labels. The documented implementation addresses each quality, although formal user-study scores are not reported.

The portal also demonstrates maintainability through modular components and schema-driven persistence. User-facing pages, administrator pages, shared navigation, API handlers, database models, resume parsers, and email utilities are separated by responsibility. This organization supports isolated correction and future extension. For example, a cloud storage adapter can replace local file storage, or a job-matching service can be introduced, without changing the candidate-facing application sequence.

H) Limitations

The current project does not report controlled response-time, load, accessibility, or security-penetration measurements. Local file storage and SQLite may limit concurrent institutional deployment, while SMTP delivery depends on external configuration and network availability. Resume extraction may vary with complex PDF layouts, scanned documents, tables, or uncommon DOCX formatting. The application currently provides search and filtering but does not use semantic recommendation, candidate ranking, or automated skill matching.

These limitations do not affect the documented demonstration workflow, but they define the boundary between a complete academic prototype and a production recruitment service. A deployment study should measure page-load time, API latency, file-upload reliability, notification delivery, concurrent-user behavior, database growth, and task-completion success. Accessibility testing should include keyboard navigation, screen-reader labels, contrast, focus order, and responsive behavior on mobile devices.

I*) Recruitment Workflow Traceability*

Traceability is implemented by preserving identifiers and relationships across every stage. The user record identifies the candidate, the job record identifies the vacancy and publishing administrator, and the application record binds both entities with the submitted resume, cover letter, date, and status. Notifications and email logs refer to the relevant event, allowing an administrator to reconstruct how and when communication occurred. This is more reliable than maintaining independent spreadsheets or email threads because the decision history remains associated with the original application.

The same traceability improves candidate experience. A job seeker does not need to contact the administrator repeatedly to ask whether an application was received or reviewed. Submission confirmation and dashboard status provide immediate evidence, while later state changes are communicated through the notification mechanisms. The system therefore reduces uncertainty without disclosing private information about other applicants or administrative decisions.

J) Scalability and Extensibility

The prototype architecture provides a direct path toward larger deployment. Prisma can change database providers while retaining the conceptual schema and most application code. Resume files can move from local storage to an object-storage service, and SMTP operations can be executed through a background queue to prevent slow external communication from delaying page responses. Server-side caching can reduce repeated reads of active job listings, while pagination can control large application tables.

Extension points are also available for intelligent recruitment functions. Extracted resume text and structured

job requirements could support skill matching, semantic search, recommendation, duplicate-resume detection, and candidate shortlisting. Such functions should be added as decision-support components with transparent criteria and human review. They would require representative data, fairness analysis, privacy controls, and quantitative evaluation; none of these predictive capabilities is claimed in the present implementation.

K) Comparison with Conventional Recruitment Practice

In a conventional manual workflow, vacancy information may be distributed through notices or multiple websites, resumes arrive through separate email threads, candidate status is maintained in spreadsheets, and applicants receive updates only when staff members send individual messages. CareerNest consolidates these operations in a single information system. Job records provide a consistent vacancy source, applications are linked directly to users and jobs, status changes are controlled through defined values, and communication is triggered from the same administrative interface.

This consolidation reduces duplicated data entry and improves record consistency. It also supports accountability because administrator actions operate on persistent entities instead of informal documents. The benefit is organizational rather than predictive: the system does not decide which candidate is best, but it gives authorized users a structured environment in which applications can be collected, reviewed, updated, and communicated efficiently.

V. CONCLUSION

This paper presented CareerNest, a modern full-stack job portal that unifies vacancy management, job discovery, resume-supported applications, real-time status tracking, role-based administration, in-app notifications, SMTP email, and communication logging. Next.js 16 and TypeScript implement the web platform, Prisma ORM and SQLite maintain persistent records, Zustand coordinates client state, and Tailwind CSS with shadcn/ui provides a responsive user experience. PDF and DOCX resume handling extends the application workflow beyond simple form submission.

The implemented interfaces confirm a complete operational sequence for job seekers and administrators. Candidates can locate relevant jobs, submit applications, and observe progress, while administrators can manage postings, inspect candidate records, update decisions, administer users, and monitor communication. The system does not claim machine-learning performance because no predictive model is trained in the project. Its contribution is the integration, usability, and traceability of the recruitment workflow.

Future work can add employer-specific accounts, organization verification, semantic resume-to-job matching, interview scheduling, analytics, recommendation services, accessibility auditing, cloud file storage, PostgreSQL deployment, background email queues, and automated testing. Controlled usability studies and load testing can quantify response time, concurrency, notification delivery, and user satisfaction before institutional or commercial deployment.

A) Future Development Directions

The next development stage should separate employer accounts from platform administrators. At present, the administrator role performs vacancy publication and candidate review. A production portal should allow verified organizations to manage only their own jobs and applications, while the platform administrator supervises organizations, policies, moderation, and system health. Organization verification, recruiter invitations, and team-level permissions would make the access-control model more suitable for real recruitment environments.

Search can be extended beyond literal filters by indexing structured skills, education, experience, and location preferences. Resume text already extracted during upload provides a foundation for candidate-controlled profile completion and improved search. Any semantic matching or ranking component should present reasons for its recommendations and should not automatically reject candidates. Training data, protected attributes, and ranking behavior would require fairness, privacy, and bias evaluation before such features are used operationally.

Interview scheduling is another practical extension. Administrators or employers could propose time slots, candidates could confirm availability, and both parties could receive calendar and email reminders. The application status model could then include interview stages and offer management. Document generation for interview letters, offer letters, and rejection communication would further reduce manual work while keeping every event connected to the original application record.

System quality can be improved through automated unit, integration, and end-to-end tests. API tests should verify authorization, validation, duplicate prevention, and status transitions. Interface tests should cover registration, search, resume upload, application tracking, and administration. Load testing should measure concurrent browsing and application submission, while security testing should inspect authentication, file upload, access control, and session handling. These evaluations would provide quantitative evidence suitable for future versions of the system.

A cloud-oriented deployment may use PostgreSQL, object storage, a managed email provider, background jobs, centralized logging, and containerized services. Monitoring should track API errors, database latency, failed email delivery, storage use, and unusual authentication activity. Regular backups and recovery drills are essential because resumes and application decisions are business records. With these improvements, CareerNest can evolve from a self-contained academic implementation into a scalable recruitment platform while preserving its current modular architecture and transparent role-based workflow.

B) Evaluation Plan for Future Versions

Future evaluation should combine technical and user-centred measures. Technical testing can record median and percentile API response time for job search, application creation, status updates, and dashboard loading. Upload tests should cover valid PDF and DOCX files, unsupported formats, oversized files, interrupted transfers, and resumes containing complex layouts. Email evaluation should record successful delivery, retry count, and failure reasons, while database tests should verify consistency after concurrent application and status-update requests.

A usability study can assign realistic tasks to job seekers and administrators. Candidate tasks may include registration, profile completion, filtered job search, resume upload, application submission, and status verification. Administrator tasks may include creating a vacancy, locating a candidate, changing application status, inspecting email logs, and closing a job. Completion rate, task time, error count, and participant satisfaction can then provide objective evidence of interface quality.

C) Responsible Recruitment Support

CareerNest is designed to organize recruitment rather than automate employment decisions. The system records and communicates administrator-selected statuses but does not infer candidate suitability. This boundary preserves human

accountability and avoids presenting an unexplained algorithmic ranking as an objective decision. If recommendation or matching is introduced later, candidates and recruiters should be informed about the attributes used, and the interface should allow users to inspect and correct profile information that influences results.

Together, the layered architecture, persistent data model, role-based workflow, communication services, and extension plan make CareerNest a useful foundation for further research and development. The project demonstrates that a modern web stack can provide an integrated recruitment experience without relying on proprietary enterprise systems. Its value lies in workflow completeness, transparency, and maintainable implementation, while future quantitative testing can establish performance and usability under real operational conditions.

The completed system demonstrates that a modern recruitment portal can integrate public job discovery, secure role-based access, resume-supported application submission, persistent status tracking, automated email communication, in-application notifications, and administrative review within one consistent workflow. The documented implementation is therefore evaluated through functional completeness, traceability, responsiveness of the interface, and continuity of recruitment records rather than through predictive-model accuracy. This distinction keeps the reported results aligned with the actual capabilities of the CareerNest project.

D) Deployment Readiness

CareerNest is currently suitable for departmental placement support, academic demonstrations, and small recruitment environments in which a limited number of administrators manage vacancies and applications. The use of SQLite and local file storage reduces configuration effort and allows the portal to run without a separate database server. For institutional deployment, the same Prisma schema can be migrated to PostgreSQL or MySQL, while uploaded resumes can be moved to protected object storage without changing the role-based workflow presented in this paper.

Production readiness also requires monitoring and recovery controls. API latency, failed authentication attempts, resume-upload errors, email-delivery failures, and unusual status changes should be logged centrally. Scheduled database backup, file-storage backup, and recovery testing are important because application histories and uploaded resumes form part of the recruitment record. Background queues can be introduced for email delivery and document processing so that slow external services do not block the user interface.

E) Operational Significance

The portal consolidates information that is often distributed among vacancy notices, email attachments, spreadsheets, and personal communication. A candidate can identify an active vacancy, submit an accepted resume format, receive confirmation, and inspect later status changes from the same account. An administrator can create and close jobs, review candidate details, update application status, manage user roles, inspect email logs, and monitor dashboard counts without transferring information between unrelated tools.

This continuity improves traceability because every application remains linked to the originating user and vacancy. Status transitions such as PENDING, REVIEWED, SHORTLISTED, REJECTED, and HIRED are stored as persistent values rather than temporary messages. The notification and email mechanisms communicate these changes, while the administrative dashboard provides aggregate visibility into job and application activity. These characteristics are the principal operational results of the implemented system.

F) Privacy and Responsible Use

Resume files and professional profiles contain personal information and must be handled with restricted access. Candidates should be able to view and update their own records, whereas administrators should access candidate information only for legitimate recruitment operations. Secure cookies, password hashing, input validation, controlled file paths, encrypted transport, retention rules, and audit logs are therefore necessary for production use. Email notifications should avoid exposing sensitive details beyond what is required to communicate an application event.

CareerNest organizes recruitment decisions but does not automatically infer candidate suitability. The administrator selects application states after reviewing the available records, and the portal communicates those decisions transparently. Future recommendation or resume-matching modules should remain decision-support features, provide understandable matching criteria, and be evaluated for privacy, fairness, and bias before influencing employment outcomes.

G) Accessibility and User Experience

The responsive interface is designed to support desktop and mobile access through reusable Next.js components, Tailwind CSS utilities, and accessible component primitives. Job cards, search controls, application forms, status badges, notification indicators, and administrative tables use consistent layouts so that users can move through the recruitment process without learning a different interaction pattern on every page. Client-side state management prevents unnecessary reloads and preserves the current search or dashboard context after common actions.

Accessibility improvements should include complete keyboard navigation, visible focus indicators, descriptive labels for form controls, sufficient colour contrast, status messages that are not communicated by colour alone, and clear validation feedback near the relevant field. Resume-upload controls should state accepted formats and size limits before submission. Administrative tables should remain usable at smaller screen sizes through responsive column selection or card-based alternatives rather than forcing unreadable horizontal layouts.

H) Data Integrity and Auditability

Prisma ORM provides a schema-driven data-access layer that links users, jobs, applications, notifications, and email logs through explicit identifiers and relationships. Unique constraints prevent the same candidate from applying repeatedly to the same vacancy, while enumerated values restrict job and application status to supported states. Server-side validation is performed before database operations so that invalid role changes, unsupported files, incomplete job records, or unauthorized status updates are rejected consistently.

Auditability is supported by preserving the original application relationship and by recording email communication outcomes. An administrator can identify which user submitted an application, which job received it, the current recruitment state, and whether a related email was successfully delivered. Production versions can strengthen this record by adding immutable status-history entries containing the previous state, new state, acting administrator, timestamp, and optional reason. Such a history would make later review possible without relying on temporary interface messages.

I) Maintainability and Extension Strategy

The separation of presentation components, Zustand actions, API routes, Prisma data access, file processing, and communication services reduces the effect of local changes. A redesigned dashboard does not require a change to the database schema, and migration from SQLite to another supported provider does not require the job-discovery interface to be rewritten. Shared validation utilities and common response structures can further reduce duplicated logic across authentication, job, application, user, and notification endpoints.

Future recruitment features can be introduced as independent modules connected to the existing entities. Interview scheduling can extend an application with proposed time slots and confirmation state; employer-specific accounts can link jobs to verified organizations; and recommendation services can use structured skills and extracted resume text without changing the basic application record. These extensions should preserve the existing rule that automated functions support authorized users and do not silently replace accountable recruitment decisions.

REFERENCES

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

[2] J. F. Reeve and M. A. Schultz, "Electronic recruitment and the evolution of online job boards," Human Resource Management Review, vol. 14, no. 3, pp. 395–408, 2004.

[3] A. Parry and S. Tyson, "An analysis of the use and success of online recruitment methods in the UK," Human Resource Management Journal, vol. 18, no. 3, pp. 257–274, 2008.

[4] A. K. Melanthiou, F. Pavlou, and E. Constantinou, "The use of social network sites as an e-recruitment tool," Journal of Transnational Management, vol. 20, no. 1, pp. 31–49, 2015.

[5] D. Chapman and J. Webster, "The use of technologies in the recruiting, screening, and selection processes for job candidates," International Journal of Selection and Assessment, vol. 11, no. 2–3, pp. 113–120, 2003.

[6] S. Laumer, A. Eckhardt, and T. Weitzel, "Online recruitment: The role of information quality and applicant reactions," Information Systems Journal, vol. 22, no. 3, pp. 179–208, 2012.

[7] A. Nikolaou, "Social networking web sites in job search and employee recruitment," International Journal of Selection and Assessment, vol. 22, no. 2, pp. 179–189, 2014.

[8] Vercel, "Next.js Documentation," 2025.

[9] Microsoft, "TypeScript Language Documentation," 2025.

[10] Meta Platforms, "React Documentation," 2025.

[11] Prisma Data, "Prisma ORM Documentation," 2025.

[12] P. LePage, "Progressive web application architecture and responsive design," Web Engineering Notes, 2020.

[13] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," RFC 7519, IETF, 2015.

[14] OWASP Foundation, "OWASP Application Security Verification Standard 4.0.3," 2021.

[15] OWASP Foundation, "Password Storage Cheat Sheet," 2024.

[16] A. Barth, "HTTP State Management Mechanism," RFC 6265, IETF, 2011.

[17] W. Stallings and L. Brown, Computer Security: Principles and Practice, 4th ed. Pearson, 2018.

[18] J. Nielsen, Usability Engineering. Morgan Kaufmann, 1993.

[19] J. Lazar, J. H. Feng, and H. Hochheiser, Research Methods in Human-Computer Interaction, 2nd ed. Morgan Kaufmann, 2017.

[20] Mozilla Foundation, "PDF.js and Web Document Processing," 2024.

[21] Mammoth Project, "Mammoth.js: Convert Word Documents to HTML," 2024.

[22] Node.js Foundation, "Node.js Documentation," 2025.

[23] Nodemailer Project, "Nodemailer Documentation," 2025.

[24] SQLite Consortium, "SQLite Documentation," 2025.

[25] Tailwind Labs, "Tailwind CSS Documentation," 2025.

[26] Shadcn, "shadcn/ui Component Documentation," 2025.

[27] Zustand Contributors, "Zustand State Management Documentation," 2025.

[28] Framer, "Motion for React Documentation," 2025.

[29] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, Univ. California, Irvine, 2000.

[30] I. Sommerville, Software Engineering, 10th ed. Pearson, 2016.