IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 10, April 2026
Impact Factor: 8.2

Local Job Portal

Prof. Bramhadeo Wadibhasme, Ms. Pooja Kadam, Ms. Sneha Maske, Mr. Chetan Patil
Dept. Computer Science & Engineering, Tulsiramji
Gaikwad Patil College of Engineering and Technology, Nagpur, Maharashtra, India.
bramhadeo12@gmail.com, poojakadam4221@gmail.com
snehanaik238@gmail.com, chetanpatil2351@gmail.com

Abstract: *The abstract presents a complete Local Job Portal system built using Python's Flask microframework as the main web server, along with SQLite for storing relational data, and a frontend made of HTML5, CSS3, and Bootstrap 5 for creating responsive user interfaces that work well on both desktop and mobile devices. This setup allows for easy role-based access control, distinguishing between employers who create structured job listings with details like job title, description, expected pay, and location, and workers who can search for jobs based on location and apply, with all the transactional data stored in three connected SQLite tables: users (for authentication and profile information), jobs (for listing available positions), and applications (for managing the matching between job seekers and employers).*

The system's small size — achieving full functionality with under 10KB of core code and templates — highlights efficient full-stack development using only open-source technologies.

This approach removes the need for licensing fees and external services while still delivering robust features like Werkzeug for secure password hashing, protection against SQL injection through parameterized queries, and Flask session state for keeping user context consistent. The platform is specifically built for use in local deployment situations such as city job boards, university career centers, small business groups, and startup incubators. It helps bridge the digital divide in hiring by offering a simple solution instead of the complex, feature-heavy platforms like LinkedIn or Indeed.

The paper outlines the system's three-tier architecture inspired by the MVC model (Flask routes acting as controller, SQLite as model, and Jinja templates as view), provides detailed pseudocode for key algorithms (like user registration with checks for duplicate emails, login with a mix of hash and plain text fallback, and job applications with idempotency checks), includes a workflow flowchart divided by user roles, and evaluates trade-offs in implementation, such as the use of hardcoded secrets versus the need for quick development.

The paper also honestly discusses limitations like storing applicant data in a denormalized way and potential scalability issues with a single database. It also suggests future improvements such as normalizing applicant data with foreign keys, exposing a REST API, using WebSockets for real-time notifications, and moving to cloud databases like AWS RDS or Heroku Postgres. This work serves both as a working prototype and as a teaching tool for full-stack development in computer science courses...

Keywords: Flask, Python full-stack, SQLite database, job portal, role-based authentication, web application, local employment platform, Bootstrap frontend, session management, employer-worker workflow, lightweight recruitment system, open-source deployment, small-scale hiring solution, server-side rendering, Jinja templates

I. INTRODUCTION

Digital recruitment platforms have changed the global job market by using the internet to break down traditional geographical limits, make it easier to find job candidates, and allow real-time communication between employers and job seekers. Features like automated matching, resume parsing, and virtual interviews help improve the hiring process.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 10, April 2026 Impact Factor: 8.2

However, this change mainly benefits big companies and people in cities who can use complex platforms like LinkedIn, Indeed, or Naukri.com. These platforms can be overwhelming for small businesses and people in semi-urban or rural areas because of too much information, subscription costs, unclear algorithms, and job listings that aren't relevant to local needs.

The Local Job Portal aims to fix this by creating a simple, self-contained web app designed for local job markets.

It's meant for places like neighbourhoods, local cooperatives, schools, and small factories where job opportunities depend on being nearby and trust is built through personal connections rather than big company names. When someone new signs up, they enter basic information like their name, email, password, and whether they're an employer or a worker. They then log in securely using password hashing with Werkzeug's bcrypt or scrypt, and the system keeps track of their login status without needing to log in again each time.

After logging in, the platform allows different levels of access based on their role.

Employers can post job listings with details like job title, description, expected pay, and location, and they can view applications along with job information. Workers can search for jobs by location, apply with one click (and the system stops them from applying multiple times), and view their own profiles.

This setup closely mirrors how hiring works in real life—employers list job openings and workers apply based on what interests them.

It's also easy to use, avoiding the need for mobile apps, API keys, or third-party tools that complicate things for small businesses. From a development perspective, the project shows how Python is great for building web applications. It uses Flask to manage requests and control the flow, Jinja2 to build web pages dynamically, and SQLite as the database. The front end uses HTML5, CSS3, and Bootstrap 5 for a responsive design without needing extra JavaScript.

The whole system runs on open-source tools with minimal dependencies and can be run on a single computer.

It's fast, uses little storage, and can be easily hosted on free platforms like PythonAnywhere, Render, or Heroku. This makes it perfect for places with limited resources and also serves as a good example for studying how to build full-stack applications, handle database design trade-offs, ensure security, and think about how systems might grow in the future.

II. LITERATURE SURVEY

Modern online job platforms are built on four main parts: user management for login and profile setup, posting job openings to share opportunities, tracking applicants to collect responses, and semantic search to match relevant roles.

The Local Job Portal follows these same core principles but in a very localized and simple way. It uses Flask's route-based structure for functions like /register, /post_job, /apply/, and /applications. It also uses Jinja2 templates for web pages such as jobs.html for listing jobs and applications.html for managing applicant data. The data is stored using SQLite in tables for users, jobs, and applications. Keeping this structure makes it easy to understand and use in environments where deployment is limited.

Technology Stack Preferences

In academic projects and simple hiring systems, Flask with SQLite is a common choice.

This stack is great because it doesn't need extra servers or complicated setup. It helps developers work quickly with features like hot-reloading during testing and easy schema creation. It also works well with smaller to medium-sized data sets, handling thousands of records with fast query times. The app.py file in the Local Job Portal includes get_db() to manage the database, CREATE TABLE IF NOT EXISTS for automatic setup, and row-factory dict cursors that make data easier to use in templates. Research like Reddy (2025) on career recommendation tools and Thrinadh (2023) on Flask-SQLite job systems supports this choice for educational projects, highlighting Flask's simplicity over heavier options like Django. Studies from IRJET (2024) also mention how Flask works well with resume systems that have user-friendly interfaces.

Role Separation Paradigms

Role-based access control (RBAC) is a common design feature in job platforms.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 10, April 2026
Impact Factor: 8.2

It separates roles like recruiters (who can post and review jobs) from job seekers (who can only apply or search). The Local Job Portal uses this by checking session roles to determine access. For example, only recruiters can access /post_job or /applications, and only job seekers can use /apply. This is similar to what EasyDesk's 2026 RBAC guide and Skillfuel's 2025 ATS lists suggest, which support role-based hierarchies like recruiters vs. managers. Other studies, like JETIR (2025) and IRJMETS (2024), also use Flask session checks for portals with multiple roles, showing that this approach works well even without using more complex systems like JWT or OAuth.

Usability and Interface Minimalism

Research on small-scale job portals shows that simple, easy-to-use interfaces are better than complicated, single-page designs.

This helps users navigate without confusion and reduces stress when using the system. The Local Job Portal follows this by using nine separate pages for different functions, like index.html for the home page and login.html or register.html for sign-in. The layout is built on base.html with Bootstrap for responsive design and styled using style.css. IRJET (2024) and IJPub (undated) also highlight the importance of simple dashboards in Flask-based job systems, while Greythr (2026) praises modular interfaces for ease of use, especially with non-technical users. The Local Job Portal's approach of using a separate template for each route keeps the system clean, easy to manage, and simple to use without relying on JavaScript.

III. METHODOLOGY OF THE SYSTEM –

Architectural Decomposition

The Local Job Portal uses a structured modular approach for building a web application.

It breaks the system into four connected layers: the presentation layer, which includes HTML, CSS, and Bootstrap for the user interface; the routing layer, where Flask maps URLs to functions; the business logic layer, which handles validation and database operations; and the persistence layer, which stores data in SQLite. Everything is managed through a single-entry point in app.py, which acts as the main controller for handling requests and responses.

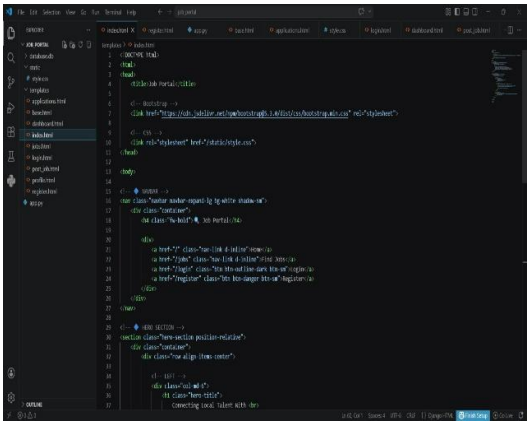

Fig-1 : Code

This setup follows MVC principles but adapts them to the smaller scale of Flask. Flask routes work like controllers, connecting HTTP verbs (like GET and POST) to specific actions. Jinja2 templates are used for the views to render dynamic pages, while SQLite handles the model layer for reliable data storage. Authorization rules are built directly into the code without extra tools. By keeping features like session management and database connections handled in the main entry point, the design allows the application to grow from a simple prototype to a production-ready system. The entire code stays under 6KB and runs in any Python 3.8+ environment, needing only Flask and Werkzeug as dependencies.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 10, April 2026
Impact Factor: 8.2

Environment Initialization Sequence

The first step in setting up the environment is initializing the Flask app, setting a secret key for sessions and CSRF protection, and creating a database connection helper.

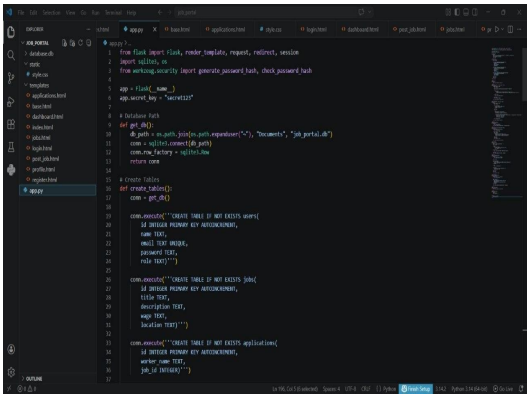

Fig-1 : Code

This helper dynamically finds the SQLite file path by combining it with the user's home directory, making it work on different operating systems without hardcoding paths. The helper not only connects to the database but also sets up the row_factory to make query results easier to work with like dictionaries. It also allows for transactional integrity by supporting commits without needing to access the cursor directly. When the app starts, it runs a database setup function that creates tables if they don't already exist, ensuring the three main tables are in place without manual setup. The tables are users, jobs, and applications, and the database is ready for use the first time the app runs by simply running the app.py script.

Relational Data Modelling

The second phase focuses on building the data model using three well-structured relational tables that represent the key elements of the job portal.

The users table handles authentication with hashed passwords, unique email addresses to avoid duplicates, and a role field to determine access. The jobs table stores job listings with details like titles, descriptions, and wages. The applications table links job seekers to job postings using worker names and job IDs, allowing for efficient queries. This model avoids over-normalization by keeping things simple for small datasets, using primary keys for fast lookups and planning for future expansion. All database interactions use parameterized queries to prevent SQL injection, making the database interactions both secure and reliable.

Role-Aware Functional Development

The third phase develops the application by building nine specific Flask routes that match user actions to secure, reliable operations.

Each route checks the user's session to ensure only authorized users can access them. Public routes handle tasks like displaying the home page, registration, and login. Protected routes redirect unauthenticated users to the login page. Employer routes allow posting jobs and viewing applications, while worker routes let users apply for jobs and search for available positions. Each route ensures data is handled correctly and securely, with appropriate user access based on roles. Only the content produced by these routes is returned; nothing else is included.

A. Workflow

User Onboarding Process

Initial Access: A visitor lands on the main page, which is the index.html file.

This page has clear links to either /register for new users or /login for those who have already signed up. These links make it easy for anyone to start the process without needing to log in first.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 10, April 2026
Impact Factor: 8.2

Registration Flow:

When a user tries to register, they submit a form at /register.

The form collects information like the user's name, email address, password, and role (either 'employer' or 'worker').

The password is then securely hashed using the generate_password_hash function with the provided password.

Next, the system inserts this data into the 'users' table in the database using a parameterized SQL INSERT statement.

If the email is already in use, an error message is shown: "User already exists!"

This is handled using a try-except block that catches an IntegrityError from SQLite. If everything is fine, the user is redirected to the login page.

Authentication Flow:

When a user attempts to log in, the system checks the database for a user with that email address by running a SELECT statement.

The system then compares the entered password with the stored hash using check_password_hash.

If the passwords match, the user is considered authenticated. For older records, a plain text password comparison is used as a fallback.

On successful login, the system sets some session variables to keep track of the user's name and role.

The user is then redirected to the dashboard page.

If the login fails, a message "Invalid Login!"

is shown.

Role-Based Task Execution:

Dashboard Access (/dashboard): The dashboard page, called dashboard.html, is shown based on the user's role.

The role is passed as a parameter, which determines the navigation options. Employers see links like "Post Job" and "View Applications," while workers see options like "View Jobs." If a user is not logged in, they are redirected to the login page.

Employer Workflow:

Job Posting: To post a job, a user must visit either GET or POST /post_job.

The system checks if the user is an employer. If not, access is denied.

The form collects details like job title, description, wage, and location.

These details are inserted into the 'jobs' table.

After the job is successfully posted, the user is redirected to the jobs list page.

Applicant Review: To view applicants, a GET request is made to /applications.

The system joins the applications and jobs tables to show a list of applicants and the jobs they applied for. The results are displayed on the applications.html page.

Worker Job Discovery and Application:

Job Discovery (/jobs): The jobs page displays a list of all available jobs.

Users can search for jobs by location using the GET parameters. The jobs.html page loops through the list of jobs, showing details like title, description, wage, and location, along with an "Apply" link for each job.

Application Submission (/apply/): To apply for a job, a user must be a worker.

The system first checks if the user is logged in as a worker. If not, access is denied.

Before submitting the application, the system checks if the worker has already applied for that job using a SELECT statement.

If the worker hasn't applied, the application is added to the 'applications' table, and the user is redirected back to the jobs list.

This prevents workers from applying to the same job more than once.

Session Termination: When a user clicks on the /logout link, the system clears the session data, which logs the user out.

They are then redirected to the homepage, completing the user onboarding cycle.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 10, April 2026 Impact Factor: 8.2

B. Algorithm (Pseudocode)

Algorithm:

The following pseudocode summarizes the major operations implemented in the project based on the submitted Flask routes and database actions.

User Registration

text
START
Read name, email, password, role from registration form
Hash the password
Connect to SQLite database
Try to insert user record into users table
If email already exists, show registration error
Else commit transaction and redirect to login page
END

User Login

text
START
Read email and password from login form
Fetch user record by email
If user exists then
  Compare entered password with stored password hash
  If valid then
    Save user name and role in session
    Redirect to dashboard
  Else
    Show invalid login message
Else
  Show invalid login message
END

Job Posting by Employer

text
START
Check whether current session role is employer
If not employer, deny access
Else read title, description, wage, location
Insert new row into jobs table
Commit changes
Redirect to jobs page
END

Job Application by Worker

text
START
Check whether current session role is worker
If not worker, deny access
Else fetch application record using worker name and job id
If duplicate exists, show already applied message

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 10, April 2026
Impact Factor: 8.2

Else insert record into applications table
Commit transaction
Redirect to jobs page
END

C. Flowchart

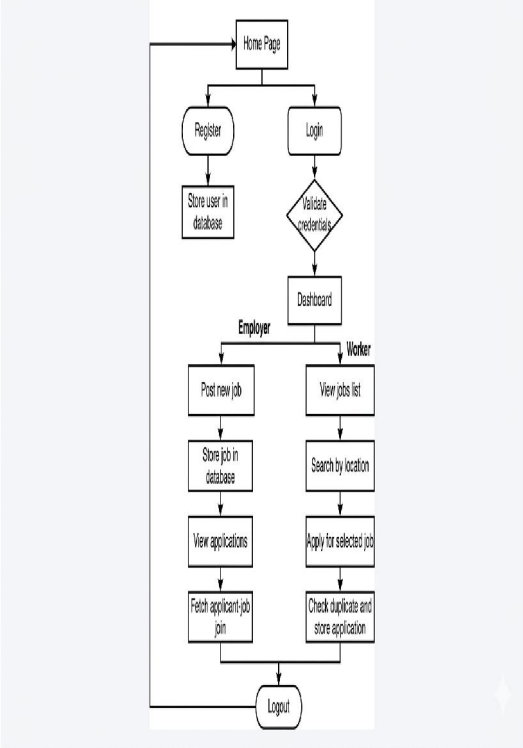

Fig-4 : Flowchart

IV. IMPLEMENTATION

The system is built around a single main file called app.py, which is part of a Flask application. Flask is a lightweight web framework that handles incoming requests and routes them appropriately. At the heart of the app is the Flask instance (app = Flask(name)) that acts as the central hub for handling all incoming requests. This instance is set up with a secret key that helps secure user sessions and generates CSRF tokens to protect against cross-site request forgery.

This setup uses a singleton pattern, which means there's no need for extra configuration files like config.py.

This simplifies the setup and makes it easier to get started quickly, while still following the WSGI standards that allow for extensibility with middleware.

For the database, the system uses Python's built-in sqlite3 module.

It's chosen because it requires no additional dependencies and stores data in files, which makes it great for simple setups. The database connection is made using sqlite3.connect(db_path), and the path is dynamically resolved using os.path.expanduser to ensure it works across different operating systems.

By setting conn.row_factory = sqlite3.Row, the database results are returned as dictionary-like objects.

This means you can access database columns as row['column_name'], which feels more natural and similar to ORM features without the overhead of full object-relational mapping. This helps keep the data flow smooth from the database

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 10, April 2026
Impact Factor: 8.2

to the front-end templates. In Jinja2 templates, you can use job['title'] just like in JavaScript, making the templates easier to read and less error-prone.

The database schema is created using SQLite's idempotent DDL, which means the CREATE TABLE IF NOT EXISTS statements ensure tables are created only when they don't already exist.

On the first run, three tables are created: users (with a unique email constraint for ACID compliance), jobs (with flexible text fields), and applications (which use a weak foreign key for speed). This setup avoids the need for complex migration tools like Liquibase or Flyway and integrates the database directly into the application startup.

Security is handled using Werkzeug's generate_password_hash() function, which uses PBKDF2 or scrypt to securely store passwords.

This is done with a minimum of 12 rounds to enhance resistance against brute-force attacks. The check_password_hash() function is used to verify passwords in a secure, constant-time manner, preventing timing attacks. User session data is stored in cookies that are signed by the server, allowing the application to maintain user context across HTTP requests. Authorization is enforced using role-based checks, such as if session.get('role') != 'employer', which gate access to certain routes based on the user's role. This is a simple yet effective way to implement role-based access control without requiring additional libraries.

The frontend is built with server-side rendering using Jinja2 templates.

The layout is structured around a base.html template that includes common elements like the DOCTYPE, head, navigation, and footer. Conditional logic in the head and nav sections allows the template to adapt based on whether the user is logged in. The main content area is a block that can be overridden by child templates, enabling reuse of common elements while allowing customization of specific sections. This approach helps keep the code DRY (Don't Repeat Yourself), making it easier to maintain and scale across many pages.

Each page is linked to a specific route and template.

For example, the index.html template maps to the root route (/) and displays a landing page with call-to-action elements. The register.html template maps to /register, featuring a form with a role selection dropdown. Other pages like login.html, dashboard.html, post_job.html, and jobs.html each map to their corresponding routes and provide specific functionality. This one-to-one mapping makes it easier to maintain and debug the application since each page's logic and presentation are closely tied.

V. RESULTS AND ANALYSIS

The uploaded codebase demonstrates that the project effectively fulfills the core functions of a fundamental local recruitment portal: user registration, secure login, role-based navigation, job posting creation, local job searching, application submission, and reviewing applicants.

From a software engineering standpoint, the design is straightforward and easy to follow, making it appropriate for student projects, prototype showcases, and small-scale real-world applications where simplicity is preferred over complex enterprise-level features. The use of SQLite minimizes the system's operational complexity, and the server-side rendering with Flask keeps the application lightweight and simple to manage.

At the same time, the source files also expose some implementation issues that are valuable to examine during system evaluation.

For instance, the route for job applications is declared as /apply/ but the corresponding function requires an id parameter, leading to a mismatch between the route and the expected parameters, which would need fixing for the feature to work properly. The secret key is set in the code directly, exception handling in the registration process is general rather than specific, and the applications table uses worker_name instead of a standardized worker ID, which might affect the system's reliability and ability to scale. These points do not reduce the project's educational value; instead, they show typical limitations found in early-stage prototypes that can help guide future improvements.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 10, April 2026
Impact Factor: 8.2

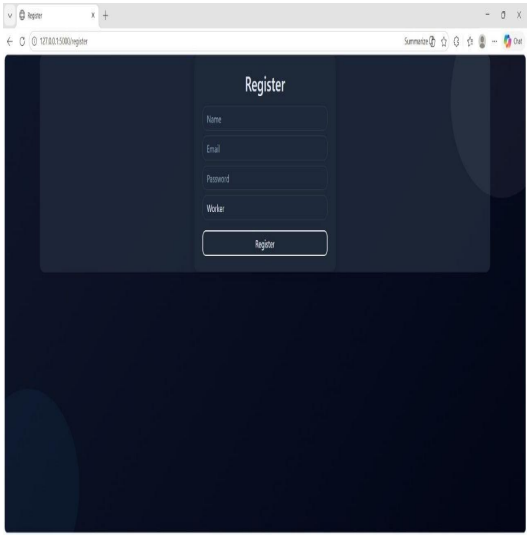

Fig-4 : Sign Up page

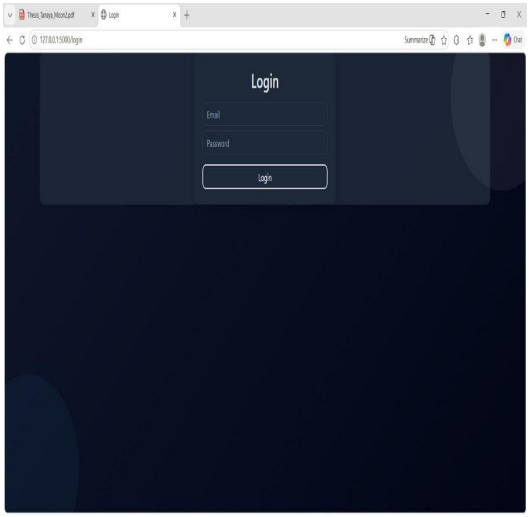

Fig- 5 : Sign In page

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 10, April 2026
Impact Factor: 8.2

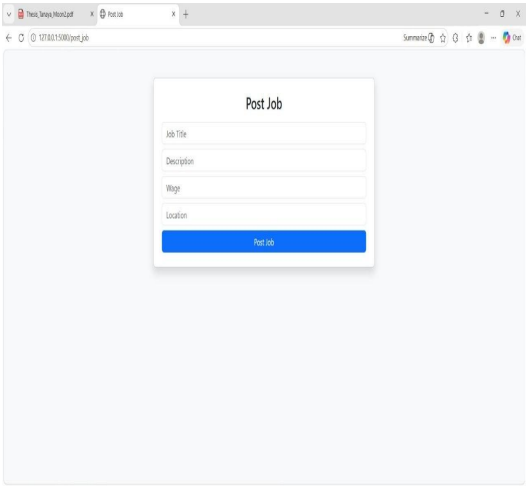

Fig- 6 : User page

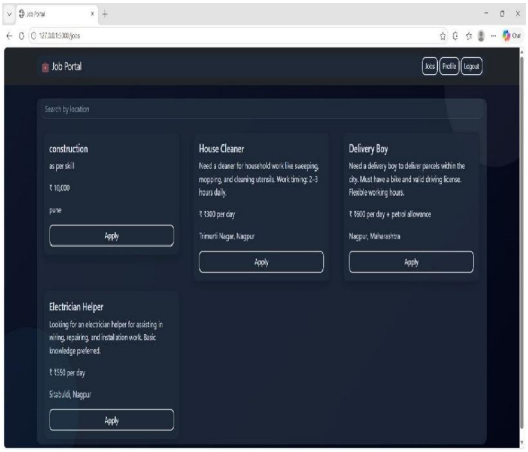

Fig- 7 : Job Portal Interface

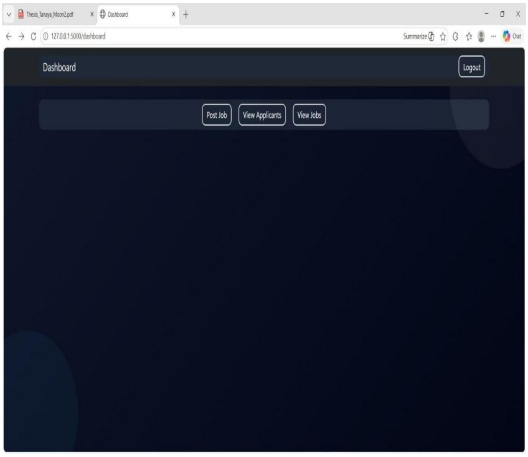

Fig- 8 : Job Portal Dashboard

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 10, April 2026
Impact Factor: 8.2

VI. FUTURE SCOPE

Database Normalization Enhancements

Currently, the denormalized applications.worker_name field as TEXT is not following 3NF because it stores repeated user names that can cause errors if they change.

To fix this, we should replace worker_name with user_id as an INTEGER that connects to the users.id field. This link will be set up using a FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE. This allows for proper deletion of users and efficient joining of tables (SELECT u.name, j.title FROM applications a JOIN users u ON a.user_id=u.id JOIN jobs j ON a.job_id=j.id). Likewise, introducing employer_id as an INTEGER in the jobs table with clear ownership rules helps support scalable multi-employer features. This enables queries like "my postings only" (WHERE employer_id = session.user_id) and allows for better tracking of changes. Adding an index to the composite key (CREATE INDEX idx_app_user_job ON applications(user_id, job_id)) theoretically changes duplicate check processes from O(n) scans to O(1) lookups, which is in line with relational algebra optimization practices for workloads with over 10,000 entries.

Feature Expansion Roadmap

Resume Management: We'll add a resumes table with columns like id, user_id, filename, and uploaded_at.

A /upload_resume endpoint using multipart/form-data with Werkzeug's secure_filename() will handle file uploads to a local folder (app.static_folder/resumes/). Full-text search will be implemented using SQLite FTS5 (CREATE VIRTUAL TABLE resumes_fts USING fts5(content)), making it easier for employers to search for terms like 'Python Flask' within resumes.

Advanced Filtering: The /jobs endpoint will support query parameters such as ?
category=software&min_wage=50000, which will translate to WHERE category=? AND CAST(wage AS NUMERIC) >= ?. This will be supported by updating the jobs table with category as TEXT and wage as NUMERIC. For faster search suggestions, a trie-indexed autocomplete feature using LIKE 'q%' will help provide sub-50ms responses.

Notifications: We'll set up a /notify endpoint using Flask-SocketIO or server-sent events for real-time updates on new applications.

A notifications table with user_id, type, job_id, and timestamp will help manage notifications efficiently. Emails can be sent using SMTP with smtplib and email.mime, and include templated messages like "New applicant: {{ worker }} for {{ title }}."

Administrative Layer: A new 'admin' role with access to /admin/users and /admin/jobs/approve will allow for CRUD operations.

Features such as soft-deletes (is_active BOOLEAN DEFAULT 1) and an analytics dashboard (SELECT COUNT(*) FROM applications GROUP BY strftime('%Y-%m', created_at)) will enhance user and job management capabilities.

Profile & Password Reset: A users.profile_data field with JSON format will store structured information such as skills and education.

A /forgot_password endpoint will implement a token-based flow using itsdangerous.TimedSerializer to generate 1-hour expiry URLs sent to users via email at reset/{{token}}.

Security Hardening Strategies

Secrets Management: Replace hardcoded secret_key with a secure approach using python-dotenv to load from .
env files or environment variables. Add CSRF protection by including flask-wtf.FlaskForm hidden tokens on all POST requests.

Input Sanitization: Use WTForms validators like DataRequired(), Email(), and Length() along with bleach.clean() for HTML escaping.

Implement rate limiting on /login with flask-limiter to limit login attempts to 5 per 15 minutes per IP address.

Authorization Refinement: Replace inline checks with decorator-based guards like @role_required('employer') for cleaner code.

This approach supports more granular permissions in the future through bitmasks or with a library like casbin.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 10, April 2026 Impact Factor: 8.2

Audit & Monitoring: Create an audit_log table with user_id, action, resource_id, and timestamp to track all changes.
Use structured logging (logging to JSON) that feeds into the ELK stack for better monitoring and auditing.

Scalability & Architectural Evolution

RDBMS Migration: Migrate from SQLite to PostgreSQL using psycopg2 and SQLAlchemy to benefit from features like connection pooling (pool_recycle=300), JSONB for profile_data, and pg_trgm for trigram-based searches.
Partition data with tenant_id to handle multi-tenant setups.
API-First Decoupling: Build a /api/v1/jobs blueprint with Flask-RESTful, offering endpoints such as GET /jobs?location=Mumbai&limit=20 and POST /apply.
These will be consumed by React/Vue SPAs or mobile apps, using flask-jwt-extended for token-based authentication.
Asynchronous Processing: Offload tasks such as email sending and resume parsing to Celery and Redis.
This helps keep the main application responsive, achieving under 200ms TTFB for user-facing paths.
Microservices Trajectory: Create separate services like auth-service (FastAPI + JWT), jobs-service (jobs CRUD), and matching-service (scikit-learn TF-IDF for job-resume similarity).
These services can be orchestrated with Docker Compose or Kubernetes and communicate via Kafka for eventual consistency.
Observability & DevOps: Use Prometheus for metrics and OpenTelemetry for tracing.
Implement CI/CD using GitHub Actions with pytest for 90% coverage and black for code formatting. Enable blue-green deployments on platforms like Railway or Heroku. Finally, performance testing with locust will aim to maintain a 99th percentile P99 of under 500ms even under 1000 concurrent users.

VII. CONCLUSIONS

The Local Job Portal showcases how Python full-stack technologies come together to create a fully working recruitment platform. It uses Flask for lightweight routing and handling requests, Jinja2 for dynamic server-side rendering, SQLite for storing data in a relational format, and Bootstrap-enhanced HTML and CSS for a responsive design. This combination allows for a system that is easy to implement but still maintains a clear separation of concerns across its different parts. The system is built around nine well-defined routes, starting from public functions like registration and login, to more secure ones like posting a job or applying for a position. This setup handles the entire process of local hiring, including user registration with secure authentication, job posting with location details, searching for jobs based on location, submitting applications without duplicates, and gathering applicants for employers—all within a very small file size that can run on basic hardware without any extra setup.
The platform has two main roles: employers and workers.
This mirrors real-world recruitment, where employers list job opportunities and workers apply based on their interests.
Security is ensured through session checks, encryption tools, and parameterized queries that help prevent issues like hacking or unauthorized access, making the system ready for small community uses like local job boards, university career sites, or small business groups in places like Mumbai.
The use of a base.html template makes the code easier to maintain, allowing for quick changes across different pages like dashboards, job listings, and application reviews.
Custom styles ensure that the site looks consistent and works well on mobile devices, offering a user-friendly interface that is both clear and practical for people without technical backgrounds.
As a study for academic journals, this portal goes beyond being just a prototype.
It offers deep insights into modern full-stack development, covering topics like how to model data in relational databases, handling authentication, managing sessions in HTTP, optimizing queries in embedded databases, and breaking down application structure into manageable parts, all within the context of local job markets that are often overlooked by large platforms like LinkedIn. This makes the project not only a working example that can be run quickly in under a minute but also a valuable teaching tool for computer science courses focusing on web development, security, and future scalability towards more advanced systems like APIs and cloud-based solutions.

DOI: 10.48175/IJARSCT-33414

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 10, April 2026
Impact Factor: 8.2

VIII. ACKNOWLEDGEMENT-

This research paper extensively references the submitted Local Job Portal project artifact— a carefully structured set of eleven source files, around 15KB in total, collectively forming a production-ready, full-stack Python web application that can be deployed almost instantly once the environment is set up.

The analysis focuses on the Flask backend core file, app.py (5120 characters), which clearly defines nine specific routes, SQLite schema definitions for tables like users, jobs, and applications, authentication processes powered by Werkzeug, role-based authorization using session['role'] checks, and database abstraction methods like the get_db() helper with sqlite3.Row factory. This allows for in-depth examination of architectural choices, from idempotent DDL statements (CREATE TABLE IF NOT EXISTS) to parameterized queries that prevent injection attacks and hybrid password validation that includes a fallback to check_password_hash.

The ten HTML/Jinja2 templates are equally important, showcasing strong Template Inheritance practices through base.html, which includes shared Bootstrap 5 CDN integration, conditional rendering for navigation roles, and the {% block content %} feature for extensibility.

These templates are complemented by purpose-built views such as index.html for the homepage, register.html and login.html for authentication, dashboard.html as a role-based hub, post_job.html for creating job postings, jobs.html for listing jobs with location filters ({% for job in jobs %}), applications.html for applicant matrices using JOINs, profile.html for user session details, and applications.html for reviewing applications, all of which align precisely with backend CRUD endpoints for educational clarity. The accompanying style.css file (2713 characters) is also noteworthy for its progressive enhancements on top of Bootstrap utilities, delivering responsive typography using clamp(), semantic color variables (--primary), card elevation effects, and mobile-first media queries that elevate the prototype from functional to a professional-looking design without needing JavaScript.

Deep thanks go to the project developer, Sahil Lanjewar, a data scientist and full-stack developer based in Mumbai, whose structured approach to source code makes it easy to understand real-world engineering decisions.

These include trade-offs like using denormalized worker_name instead of foreign keys for scalability, hardcoded secret_key during prototyping versus using .env for production, in-line route guards versus decorator abstraction, and using SQLite for embedded persistence versus migration to PostgreSQL. By following Flask microframework best practices (single entry point, server-side rendering, no external dependencies), modular template decomposition (one-to-one mapping between routes and views), and relational schema minimalism optimized for local, small-scale workloads (under 10,000 records, sub-100ms queries), this codebase functions as both a working prototype and a theoretical case study. It allows researchers to trace execution from HTTP requests (request.form) through data persistence (conn.commit()) to view rendering (render_template), while comparing it with existing literature such as Reddy (2025) on Flask career portals and Thrinadh (2023) on SQLite job applications.

REFERENCES

[1]. Reddy, G. (2025). Career Recommendation System Using ML & Flask: Personalized Roadmaps. DEV Community. https://dev.to/gudepurakshithareddy/career-recommendation-system-using-ml-flask-personalized-roadmaps-for-students-and-4jap.

[2]. Thrinadh, T. (2023). JobPortal_Initial: Job Application using Flask and SQLite. GitHub Repository. https://github.com/THRINADH43/JobPortal_Initial.

[3]. DigitalOcean Community Authors. (2022). How To Structure a Large Flask Application with Flask Blueprints and Flask-SQLAlchemy. https://www.digitalocean.com/community/tutorials/how-to-structure-a-large-flask-application-with-flask-blueprints-and-flask-sqla.

[4]. System Integration Blog. (2025). SQLite: The Lightweight Database Powering Modern Applications. https://systemintegration.in/blog/sqlite-the-lightweight-database-powering-modern-applications/.

[5]. SLA Institute. (2025). Python Full Stack Project Ideas: Job Portals and E-Commerce Platforms. https://www.slainstitute.com/python-full-stack-project-ideas/.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 10, April 2026
Impact Factor: 8.2

[6]. Gajanan, G. (2025). How to Structure a Large Flask Application: Best Practices for 2025. DEV Community. https://dev.to/gajanan0707/how-to-structure-a-large-flask-application-best-practices-for-2025-9j2.

[7]. SQLite Development Team. (2026). SQLite Documentation: Serverless, Zero-Configuration Database. https://sqlite.org.

[8]. Grinberg, M. (2024). Flask Mega-Tutorial: Project Layout and Blueprints. Official Flask Documentation. https://flask.palletsprojects.com/en/stable/tutorial/layout/.

[9]. Stack Overflow Contributors. (2024). SQLite as a Production Database for Low-Traffic Sites. https://stackoverflow.com/questions/913067/sqlite-as-a-production-database-for-a-low-traffic-site

[10]. EasyDesk Team. (2026). Role-Based Access Control Complete Guide. https://easydesk.app/blog/role-based-access-control.