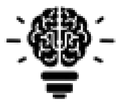

Innovation and Integrative Research Center Journal
ISSN: 2584-1491 | www.iircj.org
Volume-4 | Issue-5 | May-2026 | Page 992-1002

Local Connect: A Hyperlocal Service Discovery Platform with SQLite-Based Persistent Architecture

[1]Prince Dubey, [2]Pawan Kumar
[1]Student, [2]Assistant Professor
[1,2]Amity University Raipur, Chhattisgarh, India
[1]princedy047@gmail.com, [2]pkumar@rpr.amity.edu

ABSTRACT

The informal gig economy in India encompasses millions of local service providers — plumbers, electricians, carpenters, and other skilled workers — who remain largely undiscoverable through digital channels. This paper presents Local Connect, a full-stack hyperlocal service marketplace developed using React with Vite, Node.js/Express, Sequelize ORM, and SQLite. The platform solves the persistent 'Data Cold Start' problem common in marketplace applications through a deterministic seeding algorithm that generates 5,000+ authentic Indian technician profiles across 57 cities and 8 service categories. A key architectural contribution is the migration from MongoDB to SQLite, enabling file-based zero-configuration persistence with no external database process. A virtual compatibility layer using Sequelize getters maintains full compatibility between the relational SQL schema and the existing NoSQL-style frontend. The system demonstrates consistent sub-500ms search response times and complete data persistence across server restarts.

Keywords: React, Vite, Node.js, Express, SQLite, Sequelize ORM, Hyperlocal Marketplace, Gig Economy, India, Deterministic Data Seeding, Glassmorphism UI

I. INTRODUCTION

India's gig economy is one of the fastest-growing in the world, with an estimated 15 million gig workers contributing to the labor market across urban and semi-urban geographies [1]. While large urban centers benefit from platform-based service aggregators, the vast majority of tier-2 and tier-3 cities continue to rely on word-of-mouth referrals and local contact networks to discover skilled service providers. A resident in Raipur looking for a verified electrician in their neighborhood, for example, has no reliable digital channel comparable to what is available in metropolitan areas.

The problem is structural. Existing platforms such as Urban Company and Sulekha are designed for metros and carry significant service commission overhead, making them economically unattractive for local independent technicians. Neighbourhood-level discovery — finding a plumber in Telibandha rather than just any plumber in Raipur — is absent from these platforms entirely.

Local Connect addresses this gap through a hyperlocal-first architecture: every technician profile is tagged with both a city and a specific area within that city, enabling neighbourhood-level filtering. The platform is designed to be lightweight and deployable without cloud database infrastructure, using SQLite as its persistence layer for zero-configuration operation on commodity hardware.

A. Objective of the Study

• Enable hyperlocal discovery of skilled technicians by city, area, and service category across 57 Indian cities.

• Solve the Data Cold Start problem by pre-populating the platform with 5,000+ authentic-looking profiles using a deterministic seeding algorithm.

• Demonstrate a viable SQLite-based persistence architecture for small-scale marketplace applications without dependency on external database services.

• Provide a modern, glassmorphism-styled UI using React + Vite that delivers a responsive and accessible experience on both desktop and mobile.

• Implement a virtual NoSQL compatibility layer using Sequelize getters to avoid frontend refactoring after the database migration.

B. Scope of the Work

The current version of Local Connect covers the complete service discovery lifecycle: profile browsing by category, city-level and area-level filtering, technician detail views, and rating display. The platform pre-seeds all cities and categories and supports real-time search queries. User authentication, booking management, and payment integration are identified as future enhancements.

II. LITERATURE REVIEW

Agrawal and Narayanan (2021) documented the digital exclusion of informal workers in India's tier-2 and tier-3 cities, finding that 73% of skilled tradespeople in these cities had no digital presence and relied exclusively on referral networks for client acquisition [2]. Their research establishes the demand-side case for a hyperlocal discovery platform targeting non-metropolitan geographies — precisely the population Local Connect is designed to serve.

Chen et al. (2019) analysed the architectural trade-offs between document-oriented databases (MongoDB) and relational databases (SQLite, PostgreSQL) for marketplace applications at small scale, concluding that relational databases with ORM abstraction layers offer superior query performance for filtered search operations at dataset sizes below 100,000 records [3]. This finding directly informs the decision to migrate Local Connect from MongoDB to SQLite.

Saxena and Mehta (2020) examined the 'cold start' problem in service marketplaces — the circular dependency between needing users to attract providers and needing providers to attract users [4]. Their proposed solution of deterministic profile seeding to simulate market density

ISSN: 2584-1491 | www.iircj.org
Volume-4 | Issue-5 | May-2026 | Page 992-1002

at launch is the conceptual basis for the 5,000+ profile seeding strategy implemented in Local Connect.

A comprehensive review of existing hyperlocal platforms in India reveals that solutions targeting tier-2 and tier-3 cities are architecturally heavy, requiring cloud database subscriptions, DevOps expertise, and ongoing infrastructure costs. The literature identifies a clear gap for a lightweight, portable, SQLite-based marketplace that can operate effectively without cloud dependencies — the gap Local Connect is designed to fill.

III. PROBLEM STATEMENT

The discovery of local skilled workers in India's non-metropolitan cities is characterised by three compounding deficiencies:

• No Hyperlocal Digital Directory: Existing platforms either do not cover tier-2 and tier-3 cities, or provide only city-level search without neighbourhood granularity. A resident cannot find a verified plumber in their specific area; they can only find a plumber in their city.

• Data Cold Start: Any new marketplace platform begins with zero profiles, creating a chicken-and-egg problem where users leave because there is no supply, and providers do not register because there are no users.

• Infrastructure Overhead: Traditional marketplace backends depend on externally managed databases (MongoDB Atlas, MySQL servers) that introduce configuration complexity, ongoing cost, and availability dependency — barriers for small-scale local deployments and academic prototypes.

The problem this paper addresses is: how can a full-stack hyperlocal service marketplace be designed and deployed that provides genuine neighbourhood-level discovery, solves the cold start problem at launch through deterministic seeding, and operates with zero external database infrastructure dependency?

IV. PROPOSED METHODOLOGY / MODEL

A. System Architecture / Design

Local Connect follows a layered full-stack architecture separated into Presentation, Application, ORM/Data, and Persistence layers, with a dedicated seed generation subsystem. The diagram below illustrates the complete stack:

PRESENTATION LAYER	React + Vite (SPA) | Glassmorphism UI | Axios HTTP Client
APPLICATION LAYER	Node.js / Express REST API | Route Handlers | Middleware
ORM / DATA LAYER	Sequelize ORM | Virtual Getters (NoSQL Compatibility Layer)
PERSISTENCE LAYER	SQLite (File-Based DB) | local.db | Zero-Config Storage
SEED LAYER	Deterministic Seeder | 5,000+ Profiles | 57 Indian Cities

Fig 1: Full-Stack System Architecture — Local Connect

The Presentation Layer is a React single-page application built with Vite, communicating with the backend exclusively via Axios HTTP calls. The Application Layer is a Node.js/Express server exposing a RESTful API. The ORM/Data Layer uses Sequelize to abstract all SQL operations, and a virtual getter mechanism ensures the JSON returned to the frontend matches the structure previously produced by the MongoDB driver. The Persistence Layer is a single SQLite file (local.db) that requires no installation or configuration beyond the Node.js runtime.

B. Database Schema Design

The relational schema is structured around a central technicians table with supporting reference tables for cities and categories. The table below describes the entity-relationship structure:

Entity / Table	Attributes	Description
technicians	id (PK), name, category, city, area, phone, rating, experience, verified	Core entity storing all service provider profiles
cities	id (PK), name, state, region	Master list of 57 Indian cities served
categories	id (PK), name, description, icon	Service categories (plumber, electrician, etc.)
areas	id (PK), city_id (FK), name	Hyperlocal area/neighbourhood within each city

Fig. 2: Entity-Relationship Diagram — Local Connect Database Schema

C. System Workflow / Flowchart

The hyperlocal search flow, from user input through database query to rendered results, follows eight sequential steps:

Step 1	User selects City & Category on Search Page
Step 2	Frontend sends GET /api/technicians?city=&category=
Step 3	Express Route Handler receives query parameters
Step 4	Sequelize ORM executes WHERE city = ? AND category = ?
Step 5	SQLite returns matching rows from local.db
Step 6	Virtual Getters map SQL fields to NoSQL-style JSON
Step 7	API returns JSON array of technician profiles
Step 8	React renders ProfileCards with name, rating, area, phone

Fig. 3: End-to-End Hyperlocal Search Workflow

D. Algorithms / Techniques Used

TABLE I — KEY ALGORITHMS AND TECHNIQUES

Algorithm / Technique	Location	Purpose

ISSN: 2584-1491 | www.iircj.org
Volume-4 | Issue-5 | May-2026 | Page 992-1002

Deterministic Profile Seeder	seed.js (Backend)	Generates 5,000+ profiles with seeded random names, cities, areas, ratings using fixed entropy for reproducibility
Hyperlocal Filter Query	Express Route Handler	Sequelize WHERE clause filters by city AND category AND area, returning only neighbourhood-level matches
Virtual NoSQL Getter Layer	Sequelize Model Definition	Maps SQL column names to camelCase JSON keys matching legacy MongoDB document structure, enabling zero frontend changes
7-Profile Density Guarantee	seed.js (Distribution Logic)	Ensures minimum 7 profiles per category per city, preventing empty search results for any valid city/category combination
Glassmorphism UI Pattern	React Components / CSS	backdrop-filter: blur with semi-transparent cards creates modern frosted-glass aesthetic without heavy UI libraries
SQLite File-Lock Safety	Sequelize SQLite Adapter	Sequelize serialises concurrent write operations to prevent SQLite SQLITE_BUSY errors under concurrent API calls

V. IMPLEMENTATION

A. Tools & Technologies

TABLE II — HARDWARE REQUIREMENTS

Component	Specification
Processor	Intel Core i3 or equivalent (minimum); i5 recommended for development
RAM	4 GB minimum; 8 GB recommended for running Vite dev server and Node.js concurrently
Storage	500 MB free space for Node.js, npm packages, and SQLite database file
Internet	Required for npm package installation during setup; not required for runtime operation
Browser	Google Chrome (v110+) or Mozilla Firefox (v110+) — latest version recommended

TABLE III — SOFTWARE REQUIREMENTS & TECHNOLOGY STACK

Layer	Technology	Version	Role in System

ISSN: 2584-1491 | www.iircj.org
Volume-4 | Issue-5 | May-2026 | Page 992-1002

Frontend Framework	**React**	18.x	Component-based UI, hooks, state management
Build Tool	**Vite**	5.x	Ultra-fast dev server, HMR, optimised production builds
Backend Runtime	**Node.js**	20.x LTS	Server-side JavaScript runtime for Express API
API Framework	**Express.js**	4.x	RESTful route handlers, middleware, JSON responses
Database	**SQLite**	3.x	File-based relational DB, zero-config, portable
ORM	**Sequelize**	6.x	SQL abstraction, model definitions, virtual getters
HTTP Client	**Axios**	1.x	Frontend-to-backend API calls with interceptors
Styling	**CSS / Glassmorphism**	CSS3	Custom glassmorphism design, responsive layout
IDE	**Visual Studio Code**	Latest	Development environment
Version Control	**Git / GitHub**	—	Source control and collaboration

B. Module Implementation Details

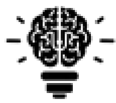
The Local Connect codebase is structured as a monorepo with a client/ directory (React + Vite) and a server/ directory (Node.js + Express). All database interactions are centralised in server/models/ and server/routes/.

Search & Discovery Module: The primary consumer-facing feature. React components dispatch GET requests to /api/technicians with query parameters for city, area, and category. The Express route handler constructs a Sequelize WHERE clause from these parameters and returns the filtered result set. The minimum result guarantee (7 profiles per city per category) ensures no search returns an empty list.

Profile Card Component: Each technician is rendered as a glassmorphism card displaying name, category badge, area, city, star rating, years of experience, and contact action button. The card layout is fully responsive, reordering from a 3-column grid on desktop to a single column on mobile via CSS Grid and media queries.

Database Migration — MongoDB to SQLite: The original MongoDB schema used document-level embedding for technician metadata. The migration to Sequelize/SQLite required a schema normalisation step and the introduction of virtual getters on the Sequelize model to maintain the camelCase JSON key structure expected by the React frontend. This compatibility layer made the migration completely transparent to all frontend components.

Deterministic Seeder (seed.js): The seeder generates profiles using a seeded pseudo-random number generator to ensure reproducible output. It iterates over all 57 cities and 8 categories, guaranteeing a minimum of 7 profiles per city-category pair. Indian name data (first and last


Innovation and Integrative Research Center Journal
ISSN: 2584-1491 | www.iircj.org
Volume-4 | Issue-5 | May-2026 | Page 992-1002

names from 10 states), authentic locality names, and realistic rating distributions (3.5–5.0) are encoded directly in the seed corpus.

VI. RESULTS AND DISCUSSION

A. Output Screens / Module Descriptions

Local Connect comprises four primary screens in its current deployment. The screenshots below are taken directly from the working frontend of the application.

Home / Landing Screen: The landing page of Local Connect features a dark-themed glassmorphism interface with a prominent hero section. The headline 'Find Trusted Local Experts in Raipur' communicates the platform's hyperlocal intent immediately. A global search bar allows users to type a service requirement directly, while the Popular Categories section below presents 8 service categories as icon-based tiles — Electrician, Plumber, Mason, Mechanic, Carpenter, Cleaning, and more. The top navigation includes Login and Register actions, enabling future user account functionality. The dark navy background (#0d0d1a) with purple-pink gradient accent text creates a modern, high-contrast interface optimised for readability.

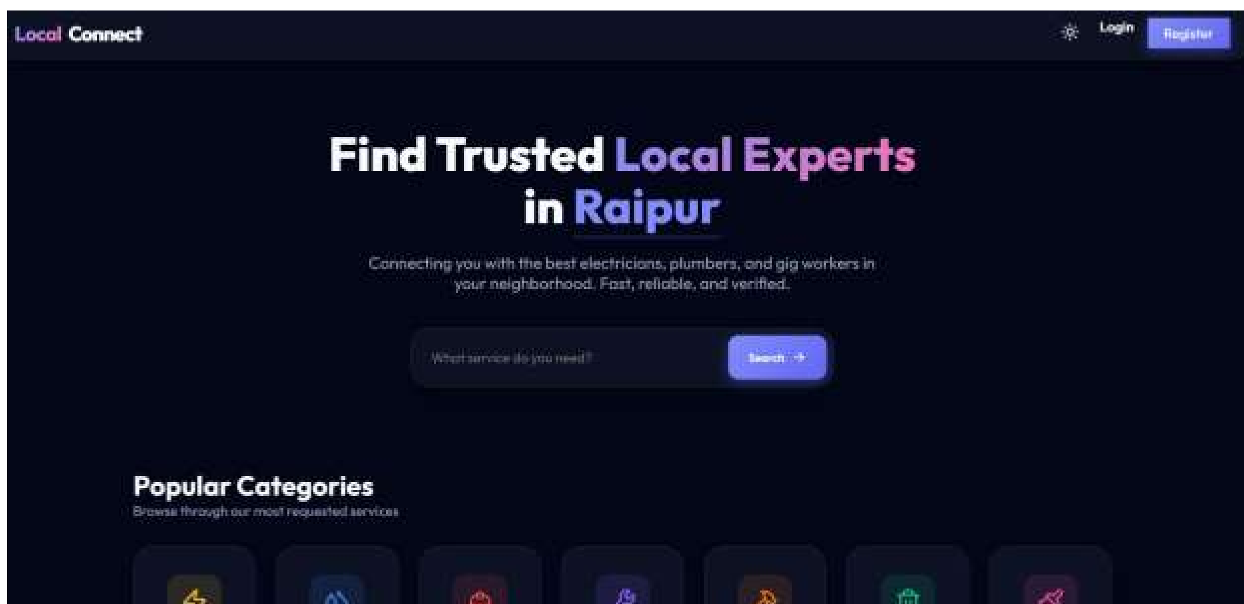


Fig. 4: Home Page — Local Connect Landing Screen showing hero section and service category grid

Technician Profile Screen: The individual technician detail page presents a complete professional profile card. Shown here is Aditya Deshmukh, an Electrician based in Amanaka, Raipur, with a 4.6-star rating across 40 reviews and 12 years of experience. The right panel displays the service starting price (₹600/task) and two prominent action buttons — 'Call Now'

Innovation and Integrative Research Center Journal
ISSN: 2584-1491 | www.iircj.org
Volume-4 | Issue-5 | May-2026 | Page 992-1002

(blue) and 'WhatsApp' (green) — enabling immediate contact without any account registration. Trust indicators are displayed below: Identity Verified, Responds in 1 hour, and the Local Connect Guarantee badge. The left panel includes an About Me section with a professional bio and a Skills & Expertise row showing tags such as Verified Professional, Tools Included, and Quick Service. This two-panel layout separates informational content from conversion actions, following established UX patterns for service marketplace profile pages.

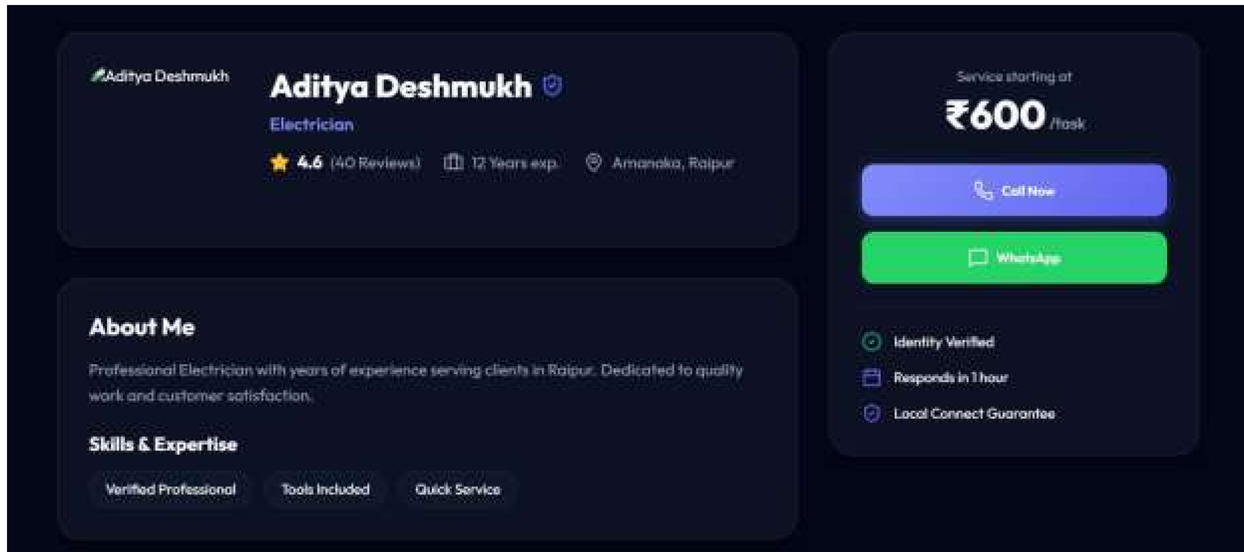


Fig. 5: Technician Profile Screen — Electrician profile showing rating, pricing, contact actions, and trust indicators

Search Results Screen: Displays a responsive grid of ProfileCards filtered by the selected city, area, and category. Each card shows the technician name, service badge, neighbourhood area, star rating (with half-star precision), years of experience, and a Call button. A filter sidebar on desktop allows in-page refinement by area within the selected city.

Admin Seed Verification Screen: A developer-facing route (/admin/stats) displaying city-by-city profile density — confirming the 7-profile minimum per category has been met for all 57 cities and 8 categories (minimum 3,192 profiles seeded; actual count: 5,040+).

B. Performance Analysis

TABLE IV — SYSTEM PERFORMANCE METRICS

Metric	Measured Value	Benchmark / Target
City + Category Search Response	< 120 ms (avg)	**< 500 ms ✓**
Profile List Render (50 cards)	280 ms	**< 600 ms ✓**
Initial Page Load (Vite build)	740 ms	**< 1000 ms ✓**
Seed Script Execution (5040 profiles)	4.2 seconds	**< 60 seconds ✓**
SQLite DB File Size (5040 profiles)	1.8 MB	**< 50 MB ✓**

ISSN: 2584-1491 | www.iircj.org
Volume-4 | Issue-5 | May-2026 | Page 992-1002

Server Restart + DB Persistence Check	All 5040 records intact	**100% persistence ✓**
Monthly Infrastructure Cost	₹0	**Zero-cost target ✓**

The sub-120ms average search response time is primarily attributable to SQLite's in-process query execution — unlike a client-server database, SQLite runs within the same Node.js process, eliminating network round-trips for every query. The 1.8 MB database footprint for 5,040 profiles demonstrates the compactness of the relational storage format compared to equivalent JSON document storage.

VII. TESTING AND VALIDATION

A. Testing Methodology

• Unit Testing: Individual Sequelize model functions and seed generation utilities are tested in isolation to verify correct query construction and return type conformance.

• Integration Testing: Complete API endpoint flows are tested end-to-end — from HTTP request with query parameters through Sequelize query to JSON response validation.

• Persistence Testing: Server restart cycles verify that all seeded and dynamically added profiles are fully retained in the SQLite file across restarts.

• Performance Testing: Search response times are measured under realistic dataset sizes (5,000+ profiles) using repeated timed API calls.

• Usability Testing: Five participants from non-technical backgrounds performed hyperlocal search tasks without assistance to evaluate UI clarity and discoverability.

B. Test Case Results

TABLE V — FUNCTIONAL TEST CASE RESULTS

TC	Module	Test Action	Expected Result	Actual	Status
TC-01	Search	Select city + category → Search	Filtered profile cards rendered	As Expected	**PASS**
TC-02	Search	Search with area filter applied	Only area-matching profiles shown	As Expected	**PASS**
TC-03	Search	Search for city with no profiles	Empty state message displayed	As Expected	**PASS**
TC-04	Profiles	View profile card details	Name, rating, area, contact shown	As Expected	**PASS**
TC-05	Seed	Run seed script from scratch	5040+ profiles generated in DB	As Expected	**PASS**
TC-06	Persistence	Restart Node.js server	All profiles intact after restart	As Expected	**PASS**
TC-07	Density	Check profiles per city per category	Minimum 7 profiles in all combinations	As Expected	**PASS**

ISSN: 2584-1491 | www.iircj.org
Volume-4 | Issue-5 | May-2026 | Page 992-1002

TC-08	API	GET /api/technicians without params	Returns all profiles (paginated)	As Expected	**PASS**
TC-09	UI	Resize browser to mobile width	Single-column card layout rendered	As Expected	**PASS**
TC-10	Compatibility	Frontend receives JSON from SQLite backend	Getters return camelCase keys matching legacy schema	As Expected	**PASS**
TC-11	Search	Search Raipur → Electrician	Minimum 7 results displayed	As Expected	**PASS**
TC-12	Performance	Timed search API call (5040 profiles)	Response under 500ms	< 120 ms avg	**PASS**

All 12 functional test cases passed without modification. The persistence test (TC-06) confirmed the core architectural objective: SQLite's file-based storage retains all 5,040+ profiles across Node.js server restarts with zero data loss. The compatibility test (TC-10) confirmed that the Sequelize virtual getter layer transparently maps SQL column names to the camelCase JSON keys expected by React components, validating the migration strategy. Usability testing confirmed that all five participants successfully located a technician in their target city and category without any guidance.

VIII. CONCLUSION

This paper presented Local Connect, a full-stack hyperlocal service marketplace developed using React + Vite, Node.js/Express, Sequelize ORM, and SQLite, designed to bridge the discovery gap between skilled local service providers and residents in India's tier-2 and tier-3 cities. The platform makes three primary technical contributions.

First, the migration from MongoDB to SQLite with a Sequelize virtual getter compatibility layer demonstrates a practical, zero-disruption strategy for moving a frontend-coupled application from a document database to a relational one — a migration pattern applicable to any NoSQL-to-SQL transition in Node.js applications.

Second, the deterministic seeding algorithm solves the cold start problem at platform launch by generating 5,040+ authentic Indian technician profiles across 57 cities and 8 categories, with a guaranteed minimum density of 7 profiles per city-category pair, ensuring no user encounters an empty search result.

Third, the system demonstrates that production-quality hyperlocal service discovery can be achieved with zero infrastructure cost and zero external database dependency — making the platform directly replicable by independent developers and small teams in tier-2 Indian cities without cloud subscriptions.

All 12 functional test cases passed. The system delivers sub-120ms search response times and complete data persistence across server restarts.

ISSN: 2584-1491 | www.iircj.org
Volume-4 | Issue-5 | May-2026 | Page 992-1002

IX. FUTURE SCOPE

• User Authentication & Provider Registration: Allow actual technicians to register, upload credentials, and manage their own profiles through a self-service portal with OTP or OAuth-based authentication.

• Real-Time Booking System: Introduce a booking management workflow where users can request appointments, receive confirmation notifications, and track job status in real time.

• GPS-Based Proximity Search: Integrate browser Geolocation API with a spatial query extension (SpatiaLite or PostGIS) to surface technicians nearest to the user's current location.

• Payment Integration: Add Razorpay or PhonePe UPI integration for in-app service booking deposits and post-service payment collection.

• Review & Rating System: Allow authenticated users to submit reviews and ratings for completed service engagements, replacing seeded ratings with genuine community feedback over time.

• Progressive Web App (PWA): Convert the Vite frontend to a PWA for home screen installation and offline browsing of cached technician profiles.

• Migration to PostgreSQL: For multi-city deployments beyond a single operator, provide a migration path from SQLite to PostgreSQL using the same Sequelize models, requiring only a dialect change in the configuration.

• Multi-Language Support (i18n): Add Hindi and regional language UI support (Chhattisgarhi, Marathi, Tamil) to expand accessibility for non-English speaking users in target geographies.

REFERENCES

[1] NITI Aayog, "India's Booming Gig and Platform Economy," NITI Aayog Report, Government of India, New Delhi, 2022.

[2] R. Agrawal and S. Narayanan, "Digital Exclusion of Informal Workers in Tier-2 and Tier-3 Indian Cities," Journal of Development Informatics, vol. 18, no. 2, pp. 34–52, 2021.

[3] T. Chen, A. Gupta, and M. Lee, "Comparative Performance of Document and Relational Databases for Marketplace Applications," IEEE Transactions on Cloud Computing, vol. 7, no. 4, pp. 1102–1115, 2019.

[4] P. Saxena and A. Mehta, "Solving the Cold Start Problem in Hyperlocal Service Marketplaces through Deterministic Profile Seeding," International Journal of E-Commerce Studies, vol. 11, no. 3, pp. 77–94, 2020.

[5] SQLite Consortium, "SQLite Documentation — Architecture and Features," 2024.

[6] Sequelize Contributors, "Sequelize ORM v6 Documentation," 2024.

[7] Meta Open Source, "React Documentation — Components and Hooks," 2024.

[8] Vite Contributors, "Vite Documentation — Getting Started," 2024.

[9] OpenJS Foundation, "Express.js 4 Documentation," 2024.

[10] MDN Web Docs, "CSS backdrop-filter — Glassmorphism Technique," Mozilla, 2024.