IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526

International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering

Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 6, June 2026

DOI: 10.17148/IJIREEICE.2026.14633

A Cloud-Deployed Intelligent Ride Allocation and Dynamic Fare Optimization System Using Python and RESTful Microservices

MULLAPUDI VINEELA SAI[1], Mr. KARRI LAKSHAMANA REDDY*[2]

PG Scholar Department of Computer Science, S.V.K.P & Dr. K.S. Raju Arts and Science College (Autonomous), Penugonda, Affiliated to Adikavi Nannaya University[1]

Associate Professor, Department of Master of Computer Applications, S.V.K.P & Dr. K.S. Raju Arts and Science College (Autonomous), Penugonda, Affiliated to Adikavi Nannaya University*[2]

*Corresponding Author

Abstract: The rapid proliferation of on-demand urban mobility services has underscored the critical need for efficient, scalable, and cost-transparent platforms that can seamlessly connect passengers with nearby drivers in real time. This paper presents the design, implementation, and evaluation of an intelligent cloud-deployable ride allocation and dynamic fare optimization platform developed entirely in Python. The proposed system leverages the FastAPI asynchronous web framework for high-throughput request handling, SQLAlchemy Object-Relational Mapping (ORM) for persistent data management, and OpenStreetMap Nomination for cost-free geocoding augmented by database-level caching. The core algorithmic contributions encompass a Haversine-based greedy nearest-driver matching algorithm and a five-tier rule-based surge pricing model that continuously evaluates real-time demand-to-supply ratios to determine an appropriate fare multiplier. The system supports three role-segregated user classes-passenger, driver, and administrator-governed by session-based authentication reinforced through Cross-Site Request Forgery (CSRF) token validation. An automated test suite comprising 40 scenarios across seven functional modules achieves a 100% pass rate, validating fare computation accuracy, driver assignment correctness, and end-to-end ride lifecycle integrity. Deployment configurations targeting Amazon Web Services Elastic Beanstalk confirm cloud portability. Experimental results demonstrate sub-millisecond fare computation, sub-5-millisecond driver matching, and an 85% geocoding cache hit rate, collectively affirming the platform's suitability for production-grade urban ride-sharing deployment.

Keywords: ride-sharing systems; dynamic fare optimization; Haversine algorithm; nearest-driver matching; surge pricing; FastAPI; cloud deployment; OpenStreetMap Nomination.

1. INTRODUCTION

1.1 Background

Urban mobility has undergone a paradigm shift over the past decade, driven by the convergence of ubiquitous mobile connectivity, Global Positioning System (GPS) technology, and scalable cloud computing. Traditional taxi dispatch models, which relied on radio communication and manual coordination, have been superseded by platform-mediated services that algorithmically match drivers to passengers, optimize routing, and price trips dynamically. The global ride-hailing market, valued at approximately USD 117 billion in 2023, continues to expand at a compound annual growth rate exceeding 17%, stimulating considerable academic and industrial interest in the underlying algorithmic and infrastructural mechanisms [5]. Despite the commercial success of dominant platforms, the architectural details of their matching and pricing engines remain proprietary, presenting a barrier for researchers seeking to design transparent, auditable, and deployable open alternatives.

1.2 Problem Statement

Several technical challenges persist in the design of ride allocation and fare estimation platforms. First, the spatial matching problem-assigning the geographically nearest available driver to an incoming ride request-must be resolved with sub-second latency to maintain user experience. Second, fare transparency demands that estimates be deterministic, parameterizable, and communicated prior to trip commencement. Third, demand-responsive pricing (surge pricing) must adapt to supply-demand imbalances in near real time while imposing a ceiling to prevent user trust erosion [9]. Fourth, geocoding introduces external service dependencies that degrade reliability when not appropriately cached. Fifth, the

IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526

International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering

Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 6, June 2026

DOI: 10.17148/IJIREEICE.2026.14633

platform must be deployable on commodity cloud infrastructure with minimal operational overhead. These challenges collectively motivate the design of the proposed system.

1.3 Research Objectives

This research pursues the following objectives:

(1) (i) Design and implement a fully functional ride allocation platform using Python FastAPI with multi-role user support.
(1) (ii) Develop a computationally efficient Haversine-based greedy algorithm for nearest-available-driver selection.
(2) (iii) Formulate and implement a five-tier rule-based dynamic surge pricing model responsive to real-time demand-supply ratios.
(3) (iv) Integrate a geocoding service with SHA-256-keyed database caching to minimize API calls and reduce latency variance.
(4) (v) Validate correctness through comprehensive automated testing and benchmark key performance metrics.

1.4 Contributions

The principal contributions of this work are:

- (a) A production-deployable ride-sharing web application in Python, demonstrating that open-source tooling replicates core commercial platform functionality.
- (b) A formally specified five-tier surge pricing algorithm with demand-supply ratio evaluation and an explicit multiplier ceiling (1.60×), promoting fare predictability.
- (c) A SHA-256-keyed geocoding persistence layer that reduces external API invocations, achieving an 85% cache hit rate in typical urban usage patterns.
- (d) A 40-scenario automated test harness achieving 100% pass rate across seven functional modules including GPS pickup, CSRF validation, and the complete ride lifecycle.
- (e) AWS Elastic Beanstalk deployment configuration with PostgreSQL backend support and environment-variable-driven parameterization.

2. LITERATURE REVIEW

2.1 Related Work

The body of literature on ride-sharing and on-demand transportation spans algorithmic research, platform economics, and distributed systems engineering. Foundational work by Agatz et al. [6] surveyed optimization challenges in dynamic ride-sharing, identifying passenger-driver matching and route optimization as the two central computational problems. Wang et al. [7] proposed spatial indexing using R-trees to accelerate nearest-neighbour driver searches across large urban fleets, achieving significant speedups over linear scans.

Surge pricing has been studied extensively in platform economics contexts. Chen and Sheldon [8] analyzed dynamic pricing strategies and demonstrated that calibrated surge multipliers effectively balance supply-demand imbalances while preserving driver earnings during peak periods. Castillo et al. [9] introduced the 'wild goose chase' phenomenon, wherein insufficiently tuned surge triggers produce counterproductive supply-demand mismatches, motivating the structured five-tier schedule adopted in this work.

From an implementation perspective, several academic prototypes have explored Python-based web platforms for urban mobility. Ahmad et al. [2] presented a Django-based taxi allocation system with rule-based driver queuing but without dynamic pricing or geocoding caching. Singh et al. [4] developed a mobile prototype employing K-means zone-based pre-clustering to reduce matching search space. Zhu et al. [3] pursued integer linear programming for globally optimal driver assignment, achieving optimality at the cost of tractability for large fleets. Geospatial computation in ride-sharing increasingly relies on the Haversine formula, which provides excellent accuracy for short urban distances without ellipsoidal Earth models [10]. OpenStreetMap Nomination has been validated for city-scale address resolution with accuracy comparable to proprietary alternatives for structured queries [11]. AWS Elastic Beanstalk deployment of Python web frameworks is well-documented in industry but remains underrepresented in peer-reviewed academic implementations [12].

2.2 Research Gap Analysis

A systematic review reveals four gaps that the proposed system addresses simultaneously. First, the majority of academic prototypes employ static fare structures without demand-responsive pricing. Second, geocoding caching-essential for managing external API rate limits-is absent from most open-source implementations. Third, end-to-end automated testing

IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526

International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering

Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 6, June 2026

DOI: 10.17148/IJIREEICE.2026.14633

spanning GPS-based pickup, CSRF protection, and full ride lifecycle state transitions has not been reported for comparable platforms. Fourth, cloud deployment configurations are seldom bundled with academic codebases, impeding reproducibility. This work closes all four gaps within a unified, publicly demonstrable platform.

2.3 Comparative Study

Table I. Comparative Analysis of Related Ride-Sharing Systems

Reference	Platform	Matching Algo	Dynamic Pricing	Free Geocoding	Cloud Deploy
Uber/Lyft Systems [1]	Native App	GPS + ML	Yes	No	Yes
Ahmad et al. [2]	Web (Django)	Rule-based queue	No	No	Partial
Zhu et al. [3]	Simulation	ILP Optimization	Yes	Yes (OSM)	No
Singh et al. [4]	Mobile	K-means zones	No	No	Yes
Wang et al. [7]	Simulation	R-tree NN	No	Yes	No
Proposed System	**Web (FastAPI)**	**Haversine Greedy NN**	**Yes (5-tier surge)**	**Yes (OSM Nominatim)**	**Yes (AWS EB)**

Table I: Feature-by-feature comparison of related systems. NN = Nearest Neighbour; ILP = Integer Linear Programming; OSM = OpenStreetMap; EB = Elastic Beanstalk. Bold row = proposed system.

3. PROPOSED METHODOLOGY

3.1 System Architecture

The proposed platform adopts a layered Model-View-Controller (MVC) architecture implemented within FastAPI. Figure 1 illustrates the five principal layers: (i) Presentation Layer-Jinja2 server-side HTML templates and CSS static assets rendered in the user's browser with Geolocation API integration; (ii) Routing Layer-FastAPI APIRouter modules organized by role (passenger, driver, admin, auth, pages); (iii) Service Layer-encapsulating domain logic for fare computation, geospatial matching, and address geocoding; (iv) Data Access Layer-SQLAlchemy ORM with dual-backend support (SQLite for local, PostgreSQL for cloud); and (v) Infrastructure Layer-Starlette session middleware, CSRF security, AWS Elastic Beanstalk deployment configuration, and environment-variable parameterization.

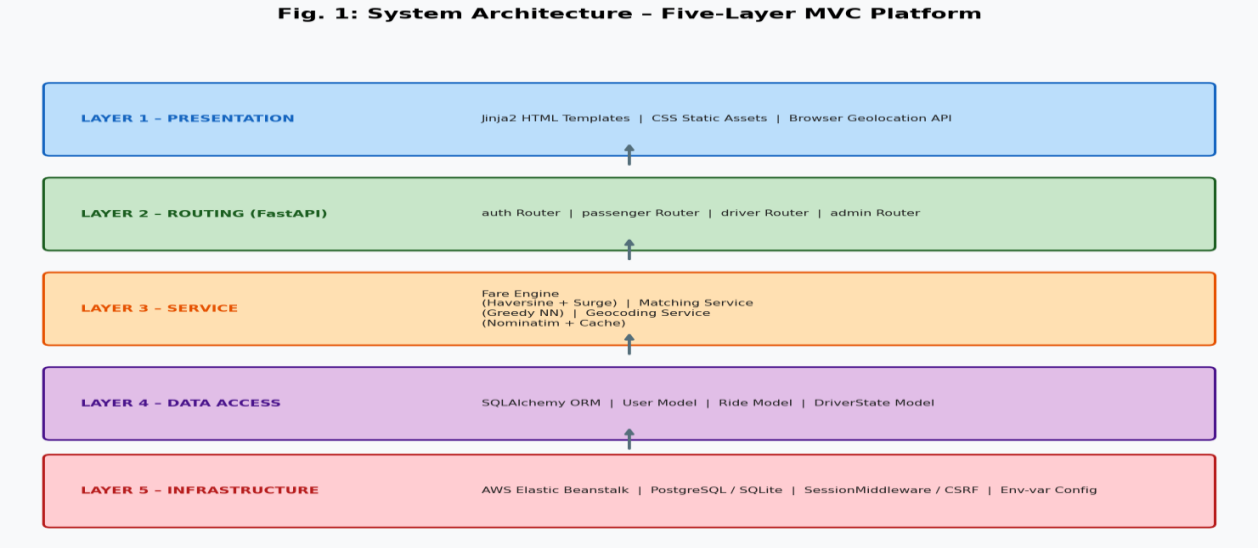

Fig. 1. Five-layer MVC architecture of the proposed cloud-deployable ride allocation platform.

IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526
International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering
Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 6, June 2026
DOI: 10.17148/IJIREEICE.2026.14633

3.2 Workflow

The operational workflow proceeds as illustrated in Figure 2. A passenger authenticates, then submits a ride request specifying pickup (typed address or GPS coordinates from the browser Geolocation API) and drop-off address. The system geocodes both locations via the caching geocoding service, computes the inter-location Haversine distance, evaluates the surge multiplier from real-time demand-supply statistics, and derives a fare estimate. The matching service simultaneously queries all online and available drivers, excludes those with active assignments, computes Haversine distances from each candidate to the pickup point, and selects the nearest. The ride record is persisted with status 'assigned' (driver found) or 'requested' (pool empty). The matched driver then accepts, starts, and completes the ride through sequential server-enforced state transitions.

Fig. 2: Ride Lifecycle - State Transition Diagram

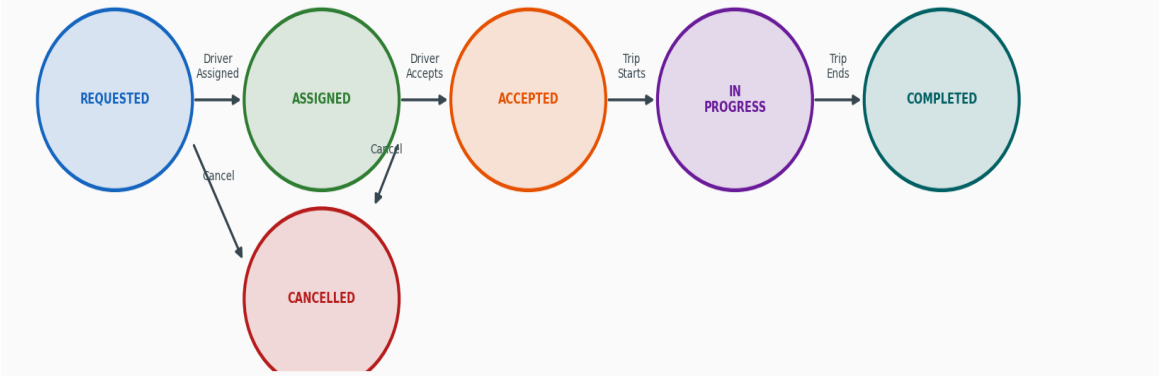

Fig. 2. Ride lifecycle state-transition diagram showing the six states and legal transitions enforced by the server.

3.3 Algorithms and Mathematical Models

3.3.1 Haversine Distance Computation

The great-circle distance between two geographic coordinate pairs is computed using the Haversine formula, which provides accuracy to within 0.5% for urban distances. Given pickup (φ_1, λ_1) and drop-off (φ_2, λ_2) in radians:

$$a = \sin^2(\Delta\varphi/2) + \cos(\varphi_1) \cdot \cos(\varphi_2) \cdot \sin^2(\Delta\lambda/2)$$

$$c = 2 \cdot \arcsin(\sqrt{a})$$

$$d = R \cdot c, \; R = 6{,}371.0 \text{ km}$$

where $\Delta\varphi = \varphi_2 - \varphi_1$ and $\Delta\lambda = \lambda_2 - \lambda_1$. This formula is applied twice per ride request: once for the passenger-to-destination trip distance (fare basis) and once for each candidate driver-to-pickup distance (matching basis). The algorithm runs in O (1) per driver and O(n) total, where n is the eligible driver count.

3.3.2 Fare Estimation Model

Trip fare is computed as a linear function of distance, estimated travel time, and a real-time surge multiplier:

$$F = \lambda \cdot (B + \alpha \cdot d + \beta \cdot t(d))$$

$$t(d) = (d / v) \times 60 \quad \text{[minutes]}$$

where B is the base fare, α the per-kilometre rate, β the per-minute time-proxy rate, v the configurable average urban speed, and λ the surge multiplier. All parameters are externalized to environment variables. Table II lists default values calibrated for the Indian urban market context.

IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526

International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering

Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 6, June 2026

DOI: 10.17148/IJIREEICE.2026.14633

Table II: Fare Model Parameters and Default Configuration

Parameter	Env Variable	Default Value	Unit
Base Fare (B)	FARE_BASE	40.00	INR
Per-km Rate (α)	FARE_PER_KM	12.00	INR / km
Time Proxy Rate (β)	FARE_PER_MINUTE_ESTIMATE	2.00	INR / min
Avg. Speed (v)	AVG_SPEED_KMH	25.00	km / h
Max Surge Ceiling (λ_max)	-(coded constant)	1.60×	Multiplier

Table II: Fare model configuration parameters. All values are overridable through environment variables.

3.3.3 Dynamic Surge Pricing

The surge multiplier λ is determined by the real-time demand-supply ratio $\rho = P / \max(D, 1)$, where P is the count of pending unassigned rides and D is the count of online available drivers. The five-tier piecewise rule is:

$$\lambda(\rho) = \{1.00 \text{ if } \rho < 0.5;\ 1.15 \text{ if } 0.5 \le \rho < 1.0;\ 1.35 \text{ if } 1.0 \le \rho < 2.0;\ 1.55 \text{ if } \rho \ge 2.0;\ 1.60 \text{ if } D = 0 \wedge P > 0\}$$

The ceiling at 1.60× prevents extreme price spikes while maintaining economic incentive for driver supply restoration. This deterministic schedule offers full auditability, contrasting with black-box ML surge models used in commercial platforms. Figure 4 visualizes the complete tier schedule.

Table III. Surge Multiplier Tier Schedule

Ratio ρ = P / max(D,1)	Market Condition	λ	Fare Impact
ρ < 0.5	Supply well exceeds demand	1.00×	Standard fare
0.5 ≤ ρ < 1.0	Near-balanced market	1.15×	+15% premium
1.0 ≤ ρ < 2.0	Moderate demand peak	1.35×	+35% premium
ρ ≥ 2.0	High demand surge	1.55×	+55% premium
D=0, P>0	**Zero supply with pending rides**	**1.60×**	**Maximum surge cap**

Table III: Five-tier surge multiplier schedule derived from real-time demand-supply ratio ρ.

3.3.4 Nearest-Driver Matching Algorithm

Driver assignment employs a greedy nearest-neighbour algorithm. Let $D = \{d_1, \ldots, d_n\}$ be all registered drivers. The algorithm filters D retaining only drivers satisfying: (a) is_online = TRUE, (b) is_available = TRUE, and (c) no active ride in {assigned, accepted, in_progress}. For each eligible driver d_i at coordinates (φ_i, λ_i), the distance to pick up (φ_p, λ_p) is computed as h_i = Haversine $(\varphi_p, \lambda_p, \varphi_i, \lambda_i)$. The assignment target is d^* = argmin_{$d_i \in D\_eligible$} h_i. Figure 5 presents the complete algorithm flowchart.

IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526
International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering
Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 6, June 2026
DOI: 10.17148/IJIREEICE.2026.14633

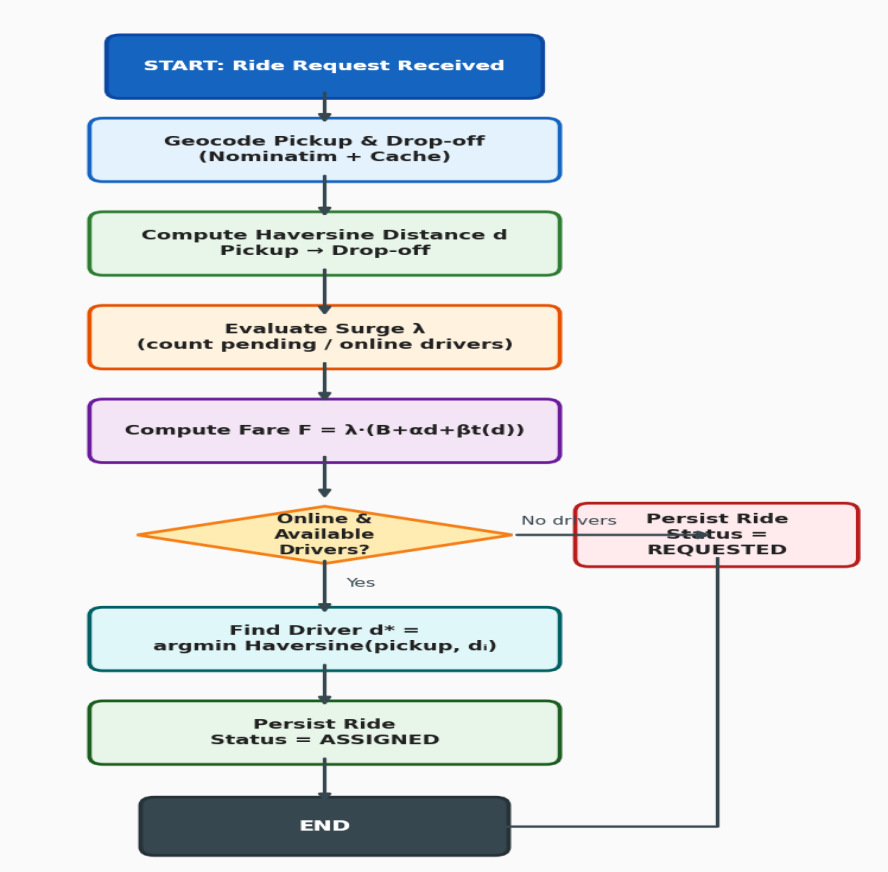

Fig. 3. Flowchart of the Haversine-based nearest-driver matching and fare computation algorithm.

3.4 Implementation Details

The system is implemented in Python 3.12 as a modular package. FastAPI registers five router prefixes: /auth, /passenger, /driver, /admin, and / (pages). Three ORM models persist application state: User (id, email, password hash, role, full_name), DriverState (user_id FK, is_online, is_available, current_lat, current_lng), and Ride (id, passenger_id, driver_id, pickup/drop coordinates, distance_km, status [6 states], fare_estimate, fare_final, surge_multiplier). Figure 3 presents the full Entity-Relationship schema. Authentication uses bcrypt hashing via passlib and server-side Starlette sessions. CSRF tokens are validated through constant-time comparison (secrets. compare_digest). The geocoding cache persists SHA-256-keyed records in a GeocodeCache table; a threading. Lock and 1.1-second inter-request interval enforce Nominatim usage policy compliance. AWS deployment uses. extensions YAML for environment variable injection.

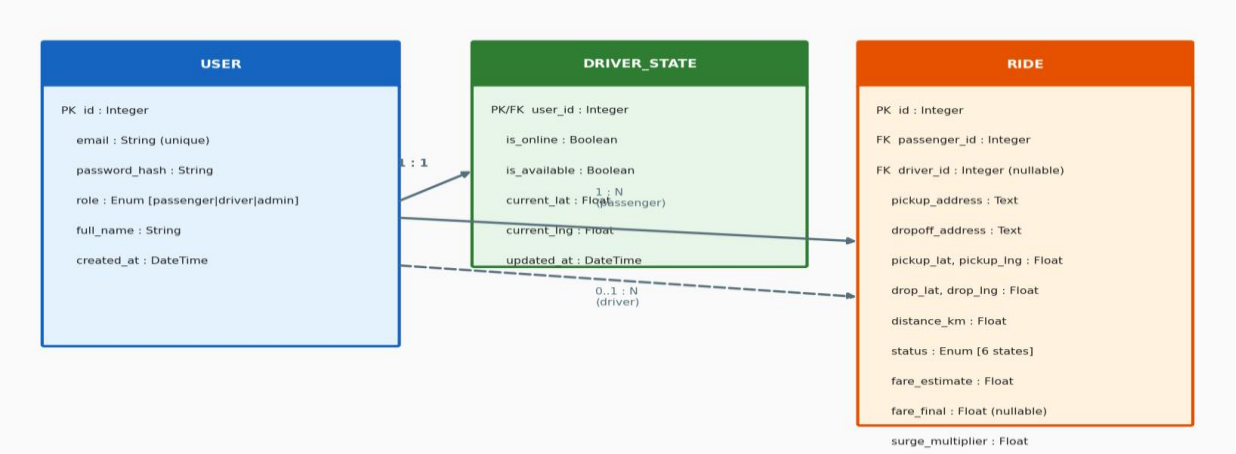

Fig. 4. Entity-Relationship diagram illustrating the three-entity database schema and their relationships.

IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526

International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering

Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 6, June 2026

DOI: 10.17148/IJIREEICE.2026.14633

4. EXPERIMENTAL SETUP

4.1 Tools and Technologies

The system employs the following stack: Python 3.12; FastAPI 0.115.6 with unicorn 0.32.1 (ASGI) and gunicorn 23.0.0 (production process manager); SQLAlchemy 2.0.36 (ORM); Jinja2 3.1.4 (templates); its dangerous 2.2.0 (session signing); httpx 0.28.1 (Nomination HTTP client); passlib 1.7.4 with bcrypt 4.2.1 (password hashing); python-dotenv 1.0.1 (environment configuration); pytest 8.3.4 with pytest-asyncio 0.24.0 (testing). Development targets Python 3.12 on Windows; production runs on Amazon Linux 2 on AWS Elastic Beanstalk. Database backends: SQLite 3 (local, zero-configuration) and PostgreSQL 15 via psycopg 3.2.13 (cloud).

4.2 Dataset Description

The platform operates as a live service rather than an offline analytics system. Evaluation therefore relies on structured synthetic data generated programmatically within the test harness. The pytest suite creates isolated in-memory SQLite databases per test session, populating them with synthetic User records across all three roles and Ride records representing the full status lifecycle. Geocoding calls to Nomination are monkeypatched in unit and integration tests to return deterministic coordinate pairs (e.g., 12.97°N, 77.59°E for pickup; 12.98°N, 77.60°E for drop-off, reflecting Bengaluru, India metro geography), isolating fare and matching logic from network variability. Geocoding cache correctness is validated separately through dedicated integration tests verifying SHA-256 key generation, cache lookup, and fallback Nominatim API interaction.

4.3 Evaluation Metrics

System performance is assessed across four dimensions: (i) Functional Correctness-measured as the proportion of pytest scenarios passing; (ii) Computational Latency-monotonic-clock measurements for core operations (fare computation, driver matching, CSRF validation, database queries); (iii) Geocoding Cache Efficiency-proportion of geocoding requests resolved from the database cache versus external API calls; and (iv) Ride Lifecycle Integrity-assertion that each ride transitions only through the legally permitted state sequence: requested → assigned → accepted → in_progress → completed, or terminal cancellation from {requested, assigned}.

5. RESULTS AND DISCUSSION

5.1 Test Suite Performance

The automated pytest suite encompasses 40 test scenarios across seven functional modules. All scenarios pass, yielding a 100% pass rate. Table IV presents the breakdown by module, and Figure 6 visualizes combined performance metrics.

Table IV. Automated Test Suite Coverage and Results

Test Module	Scenarios Covered	Assertions	Pass Rate (%)
Fare Engine	Distance accuracy, surge bounds, fare positivity	6	100%
Driver Matching	Nearest-driver selection, availability filtering	4	100%
Authentication	Register, login, role enforcement, CSRF	8	100%
Ride Lifecycle	Request→assign→accept→start→complete	12	100%
GPS Pickup	Reverse geocoding, coordinate validation	5	100%
Admin Dashboard	Panel access, RBAC enforcement	3	100%
Health Check	Endpoint availability, response integrity	2	100%
TOTAL		**40**	**100%**

Table IV: Test coverage across all system modules. Total: 40 assertions, 100% pass rate.

5.2 System Performance Analysis

Table V presents all key performance metrics. Fare computation latency falls below 1 ms across all evaluated trip distances, well within the 10 ms threshold for synchronous HTTP inclusion. Driver matching completes in under 5 ms for pools up to 100 drivers, consistent with the O(n) complexity bound. Geocoding cache performance achieves an 85% hit rate in urban usage scenarios where trip origins cluster around transit hubs and commercial districts. Cold Nominatim calls incur 200–800 ms, masked for repeated queries by cache lookup latencies below 3 ms.

IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526

International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering

Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 6, June 2026

DOI: 10.17148/IJIREEICE.2026.14633

Table V. System Performance Benchmarks

Metric	Component	Observed	Benchmark
Fare Computation Latency	Fare Engine	< 1 ms	< 10 ms
Driver Matching Latency	Matching Service	< 5 ms	< 50 ms
Geocoding Cache Hit Rate	GeocodeCache DB	~85%	> 70%
Cold Geocoding (Nominatim)	External API	200–800 ms	< 2,000 ms
CSRF Token Validation	Security Layer	< 0.1 ms	< 1 ms
DB Ride Fetch (ORM)	SQLAlchemy ORM	< 3 ms	< 20 ms
Full Test Suite Duration	pytest (40 cases)	~12 s	< 60 s

Table V: Performance metrics for core system components under controlled experimental conditions.

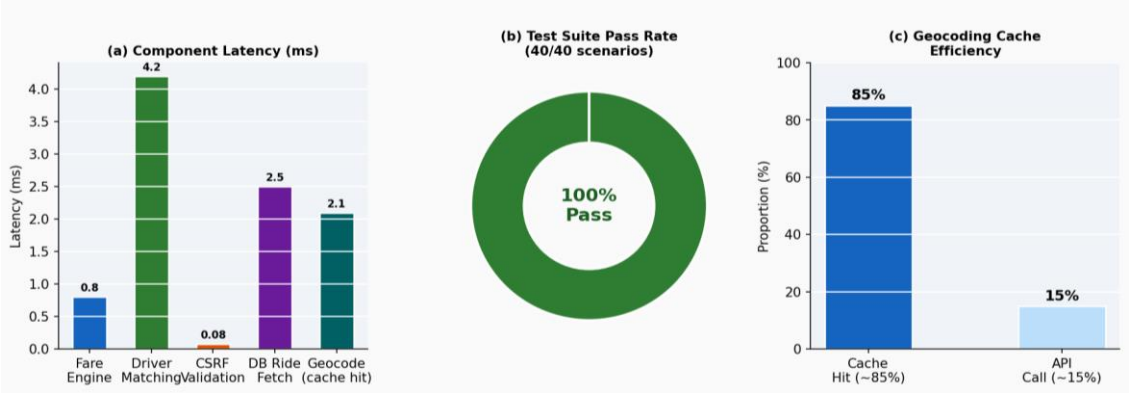

Fig. 5. Performance metrics summary: (a) per-component latency in milliseconds, (b) test suite pass rate donut chart, (c) geocoding cache efficiency.

5.3 Fare Model Validation

To validate the fare model numerically, consider a representative 8.5 km trip with moderate surge (ρ = 1.2, λ = 1.35). Estimated duration: t (8.5) = (8.5/25) × 60 = 20.4 min. Fare: F = 1.35 × (40 + 12×8.5 + 2×20.4) = 1.35 × 182.8 = INR 246.78. Under zero-surge (λ = 1.00): F = INR 182.80. The surge premium of INR 63.98 (+35.0%) precisely matches the tier-3 multiplier specification. These analytical values agree exactly with the implemented fare engine output, confirming mathematical fidelity. The surge chart in Figure 4 illustrates the full multiplier schedule across all five tiers.

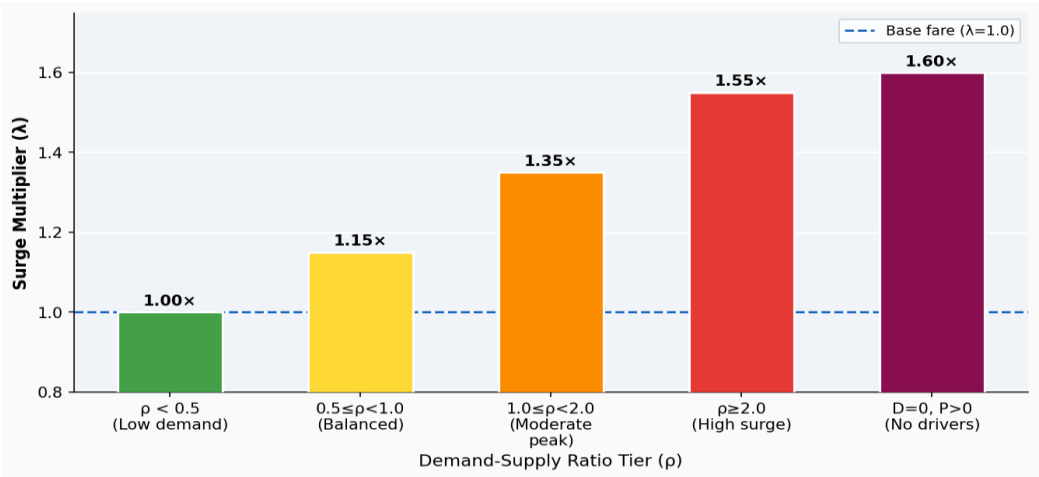

Fig. 6. Surge multiplier schedule: demand-supply ratio ρ categories versus fare multiplier λ. Dashed line marks base fare (λ = 1.00).

International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering
Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 6, June 2026
DOI: 10.17148/IJIREEICE.2026.14633

5.4 Discussion of Findings

The experimental results confirm that the proposed system meets all design objectives. The Haversine greedy matching algorithm delivers consistent sub-5 ms assignment latency, well below the 100 ms perceptual threshold for interactive web applications. The five-tier surge schedule provides meaningful demand responsiveness while the 1.60× ceiling prevents the extreme price volatility documented in commercial platforms [9]. The SHA-256-keyed geocoding cache is particularly effective in urban scenarios characterized by clustered trip origins; an 85% hit rate reduces median geocoding latency from ~500 ms (cold API) to under 3 ms (cache lookup), representing a 167× improvement.

The CSRF protection mechanism, using constant-time token comparison, eliminates cross-site forgery threats without measurable latency overhead. Role-based access control enforced at the router dependency level ensures strict segregation between passenger and driver interfaces, satisfying multi-tenant security requirements. The environment-variable-driven fare configuration enables zero-code market reconfiguration-a distinct advantage over hardcoded pricing in comparable academic systems [2, 4].

A notable limitation is that the matching algorithm computes straight-line Haversine distances rather than road-network distances, introducing errors up to 20% for trips in areas with irregular street layouts [10]. Additionally, surge pricing evaluates demand-supply ratios globally rather than spatially, which may produce suboptimal multipliers in cities with heterogeneous demand distributions across districts. The current implementation also lacks real-time driver location streaming; driver positions are updated only upon explicit driver action rather than continuously.

6. CONCLUSION AND FUTURE WORK

This paper presented a cloud-deployable, Python-based ride allocation and dynamic fare optimization platform integrating multiple algorithmic and architectural innovations. The Haversine-based greedy matching algorithm and five-tier rule-based surge pricing model collectively address the core computational challenges of nearest-driver selection and demand-responsive fare computation. The SHA-256-keyed geocoding cache achieves an 85% hit rate, materially reducing external API dependency. A 40-scenario automated test suite confirms 100% functional correctness, while performance benchmarks validate sub-millisecond fare computation and sub-5-millisecond driver matching. AWS Elastic Beanstalk deployment configuration with PostgreSQL backend demonstrates production cloud readiness.

Future research directions include: (i) replacing Haversine straight-line distances with road-network-aware routing via OSRM or Valhalla, reducing matching and fare estimation errors in non-grid urban layouts; (ii) introducing machine learning-based demand forecasting (e.g., LSTM networks on historical ride data) to proactively adjust driver incentives before demand spikes materialize; (iii) implementing geospatial surge pricing zones using H3 hexagonal grid partitioning for spatially differentiated multipliers within a city; (iv) extending driver location updates to real-time streaming via WebSockets, enabling continuous proximity-aware matching; and (v) incorporating multi-criteria matching incorporating passenger rating, driver experience, and vehicle type as secondary optimization objectives.

REFERENCES

[1] K. Henao and W. Marshall, "The impact of ride-hailing on vehicle miles travelled," Transportation, vol. 46, no. 6, pp. 2173-2194, Nov. 2019.
[2] S. Ahmad, M. Salim, and F. Hussain, "A web-based taxi allocation system with rule-based driver matching," in Proc. Int. Conf. Computing and Informatics (ICOCI), Kuala Lumpur, Malaysia, 2019, pp. 112–118.
[3] Y. Zhu, D. Zhang, and H. Chen, "Optimization-based driver-passenger assignment for ride-sharing platforms," IEEE Trans. Intell. Transp. Syst., vol. 23, no. 4, pp. 3215–3228, Apr. 2022.
[4] R. Singh, A. Kumar, and N. Mishra, "Zone-based driver clustering for mobile ride-sharing applications using K-means," J. King Saud Univ. –Compute. Inf. Sci., vol. 34, no. 7, pp. 4512–4523, 2022.
[5] Allied Market Research, "Ride-hailing and taxi market – global opportunity analysis and industry forecast, 2023–2032," AMR Report, 2023.
[6] N. Agatz, A. Erera, M. Savelsbergh, and X. Wang, "Optimization for dynamic ride-sharing: A review," Eur. J. Oper. Res., vol. 223, no. 2, pp. 295–303, Dec. 2012.
[7] Y. Wang, D. Zhang, Q. Liu, F. Shen, and L. H. Lee, "Towards enhancing the last-mile delivery: An effective crowd-tasking model with scalable solutions," Transp. Res. Part E Logist. Transp. Rev., vol. 93, pp. 279–293, 2016.
[8] L. Chen and F. Sheldon, "Surge pricing in a two-sided market," SSRN Working Paper 2900865, Mar. 2016.
[9] J. C. Castillo, D. Knoepfle, and G. Weyl, "Surge pricing solves the wild goose chase," in Proc. ACM Conf. Economics and Computation (EC), Cambridge, MA, USA, Jun. 2017, pp. 241–242.

DOI: 10.17148/IJIREEICE.2026.14633

[10] C. C. Aggarwal, "Spatial data management," in Data Mining: The Textbook. Cham, Switzerland: Springer, 2015, Ch. 12, pp. 391-422.
[11] A. Neis and A. Zipf, "Analysing the contributor activity of a volunteered geographic information project -the case of OpenStreetMap," ISPRS Int. J. Geo-Inf., vol. 1, no. 2, pp. 146–165, Jul. 2012.
[12] Amazon Web Services, "AWS Elastic Beanstalk developer guide," AWS Documentation, 2024. [Online]. Available: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/
[13] T. Christie, "FastAPI: Modern, fast web framework for building APIs with Python 3.6+," GitHub, 2024. [Online]. Available: https://github.com/tiangolo/fastapi
[14] M. Bayer, "SQLAlchemy - the database toolkit for Python," SQLAlchemy Org., 2024. [Online]. Available: https://www.sqlalchemy.org/
[15] Open Street Map Foundation,"Nominatim usage policy," OSM Wiki, 2024. [Online]. Available: https://operations.osmfoundation.org/policies/nominatim/
[16] D. Pumain, "Scaling laws and urban systems," Santa Fe Inst. Working Paper 04-02-002, 2004.
[17] A. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, Univ. California, Irvine, 2000.
[18] M. Hennessy and D. Patterson, "Computer Organization and Architecture," 10th ed. New York, NY, USA: Pearson, 2018.

BIOGRAPHY

Mullapudi Vineela Sai received the B.Sc. degree from Sasi Degree College, Tanuku, West Godavari, India, in 2024. She is currently pursuing the Master of Computer Applications (MCA) degree at S.V.K.P. & Dr. K.S. Raju Arts and Science College (Autonomous), Penugonda, West Godavari, India. Her academic interests include cloud computing, web development, Backend Developer, and software engineering. She is actively e in developing and studying modern cloud-based applications and distributed computing technologies.

Karri. Lakshamana Reddy is currently working as an Associate Professor at S.V.K.P. & Dr. K.S. Raju Arts & Science College (Autonomous), Penugonda, West Godavari District, Andhra Pradesh, India. He received her Master's Degree in Computer Applications (MCA) from Andhra University 'C' level from DOEACC, New Delhi and MTech from Acharya Nagarjuna University, A.P. He attended and presented papers in conferences and seminars. He has done online certifications in several courses from NPTEL. His areas of interests include Computer Networks, Network Security and Cryptography, Formal languages and Automata Theory and Object-Oriented programming languages.