iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** IV **Month of publication:** April 2026

DOI: https://doi.org/10.22214/ijraset.2026.79293

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue IV Apr 2026- Available at www.ijraset.com

Real-Time Ride-Share Dynamic Pricing Engine Using Machine Learning and Generative AI

Arun Srinivas A[1], K Sri Aishwariya[2], Pragash K[3]
[1]*Machine Learning Engineer, Department of Artificial Intelligence and Data Science, Sri Manakula Vinayagar Engineering College, Puducherry, India*
[2]*Frontend Developer, Department of Artificial Intelligence and Data Science, Sri Manakula Vinayagar Engineering College, Puducherry, India*
[3]*Associate Professor, Department of Artificial Intelligence and Data Science, Sri Manakula Vinayagar Engineering, College, Puducherry, India*

Abstract: ***The proliferation of ride-sharing platforms in urban India has created a pressing need for intelligent, transparent, and cost-efficient dynamic pricing systems. This paper presents the design, implementation, and evaluation of a real-time dynamic pricing engine for ride-sharing services, built entirely on a serverless AWS architecture. The proposed system integrates an XGBoost machine learning model trained on 12,000 synthetically generated ride samples to predict optimal fares based on ten contextual features, including trip distance, estimated travel time, driver-rider demand ratio, weather conditions, and traffic congestion levels. Unlike conventional approaches that multiply a static base rate by a surge factor, the proposed additive pricing formula decomposes every rupee of the final fare into individually visible and traceable components: distance fee, time fee, weather surcharge, traffic surcharge, and demand-based surge fee. A React-based frontend hosted on Amazon S3 and powered by the TomTom Maps SDK delivers live visualisation of drivers, riders, routes, and pricing signals. AWS EventBridge schedules background city simulation via two Lambda functions that update driver positions and maintain a live rider request pool every minute. Generative AI-powered fare explanations are produced via the Groq API using the LLaMA 3.3 70B model, providing passengers with natural-language price transparency. Experimental results demonstrate that the XGBoost model achieves a Mean Absolute Error of Rs 4.23 and an R^2 score of 0.9982 on a held-out test set, while the entire system operates within the AWS Free Tier at near-zero cost.***

Keywords: Dynamic Pricing, XGBoost, Serverless Architecture, AWS Lambda, Ride-Sharing, Surge Pricing, Generative AI, Real-Time Systems, Demand Forecasting, LLaMA.

I. INTRODUCTION

Dynamic pricing, also termed surge or demand-responsive pricing, is a mechanism in which the price of a service is adjusted in real time based on prevailing supply and demand conditions. In the context of ride-sharing, platforms such as Uber, Ola, and Rapido have widely adopted dynamic pricing to balance driver availability against passenger demand, reduce wait times during peak periods, and incentivise drivers to operate in high-demand zones. Despite its commercial success, the algorithmic underpinnings of these platforms remain proprietary, and passengers frequently experience fare increases without comprehensible justification, eroding trust [1]. Academic research on dynamic pricing for ride-sharing has made significant strides, yet two critical gaps persist. First, existing models commonly incorporate a static flat base rate that does not reflect actual cost drivers such as distance, time, or environmental conditions, making the fare opaque to passengers. Second, most research prototypes are deployed on monolithic server architectures that demand significant infrastructure investment and operational expertise, rendering them impractical for regional or resource-constrained deployments. This paper addresses both gaps through a novel system built for the city of Puducherry, India. The system replaces the conventional base-rate model with a fully additive pricing formula, trains an XGBoost machine learning model using the formula as the labelling function on a synthetic dataset, and deploys the entire stack on AWS serverless infrastructure (Lambda, API Gateway, DynamoDB, S3, EventBridge) at near-zero cost. Additionally, large language model (LLM) integration via the Groq API provides passengers and drivers with natural-language explanations of pricing decisions, a capability not previously reported in the ride-share pricing literature. The main contributions of this work are: (i) an additive pricing formula with per-component transparency; (ii) an XGBoost model achieving MAE Rs 4.23 and R^2 = 0.9982; (iii) a fully serverless AWS deployment across 12 Lambda functions within the Free Tier; and (iv) Generative AI fare explanations and driver earnings advice via LLaMA 3.3 70B.

The remainder of this paper is organised as follows: Section II reviews related literature; Section III presents the system architecture; Section IV details the pricing formula; Section V describes the machine learning methodology; Section VI covers Generative AI integration; Section VII discusses implementation; Section VIII presents results; and Section IX concludes.

II. LITERATURE REVIEW

A. Dynamic Pricing in Transportation

Chen and Sheldon [2] provided one of the earliest empirical analyses of surge pricing on the Uber platform, demonstrating that higher prices during demand peaks effectively increase driver supply and reduce passenger wait times. Castillo et al. [3] extended this analysis, showing that surge pricing resolves the inefficiency of drivers and passengers searching for each other in different zones, thereby improving overall market welfare. These works validated the economic rationale for dynamic pricing but did not address the algorithmic implementation or transparency concerns relevant to this paper.

B. Machine Learning for Fare Prediction

A substantial body of literature applies machine learning to fare and demand prediction. Neun et al. [4] used gradient boosting on New York City taxi GPS data, achieving strong predictive accuracy but relying on large historical datasets unavailable in new deployments. Wang et al. [5] proposed deep learning spatio-temporal models for ride demand forecasting, demonstrating superior accuracy at the cost of significant computational overhead. More recently, ensemble methods including random forests and XGBoost have been shown to outperform deep learning on tabular transportation data in terms of both accuracy and training efficiency [6], motivating their use in the present work.

C. Serverless Cloud Architectures

Serverless computing, where infrastructure management is delegated entirely to the cloud provider, has emerged as a cost-effective paradigm for intermittent and variable workloads. Zhang et al. [7] demonstrated the viability of AWS Lambda for real-time mobility API services, achieving sub-second latency at low cost. However, their implementation did not incorporate ML model inference. Hellerstein et al. [8] discussed the theoretical limitations of serverless computing for stateful workloads, concluding that DynamoDB and S3 are effective compensating patterns for state management, exactly the approach used in this paper.

D. Generative AI for Pricing Transparency

The application of large language models to structured data explanation is a rapidly growing area. Brown et al. [9] established that GPT-class models can generate accurate, coherent explanations of numerical data when provided with structured prompts. In the transportation domain, there is no prior published work applying LLMs to fare explanation. Zhao et al. [10] applied LLMs to financial decision explanation, providing a methodological precedent for the approach used in this paper. The present work adapts this paradigm specifically to ride-share fare transparency, introducing structured prompting techniques that reference individual fare components.

E. Summary and Research Gap

Table I summarises the comparison of the proposed system against related works across five dimensions: ML model, serverless deployment, real-time signals, generative AI, and pricing transparency. No prior work combines all five capabilities into a single deployable system, representing the novelty of the present contribution.

TABLE I. COMPARISON WITH RELATED WORK

Work	ML Model	Serverless	Live Signals	GenAI	Transparent
Chen et al. [2]	No	No	No	No	No
Neun et al. [4]	GBM	No	No	No	No
Wang et al. [5]	Deep NN	No	Yes	No	No
Zhang et al. [7]	No	Yes	No	No	No
Zhao et al. [10]	No	No	No	Yes	Partial
Proposed	XGBoost	Yes	Yes	Yes	Yes

III. SYSTEM ARCHITECTURE

The proposed system follows a four-layer serverless architecture deployed on AWS in the ap-south-1 (Mumbai) region. The four layers are: (1) Frontend, (2) API, (3) Data, and (4) Background Task Scheduling. Fig. 1 presents the complete architecture diagram. The system is designed for zero-server management, automatic horizontal scaling, and operation entirely within the AWS Free Tier.

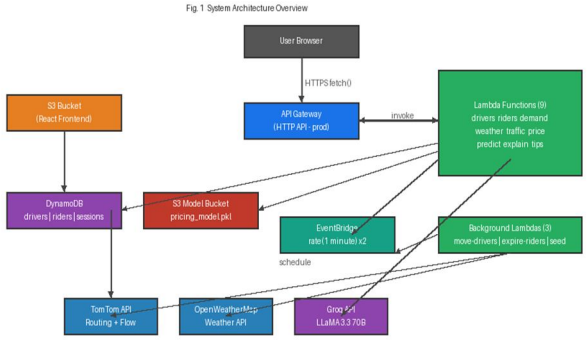


Fig. 1 System Architecture Overview — AWS Serverless Stack

A. Frontend Layer

The frontend is a React single-page application (SPA) hosted on Amazon S3 with static website hosting enabled. The TomTom Maps JavaScript SDK renders an interactive map centred on Puducherry (11.9139°N, 79.8145°E) displaying real-time driver markers, active rider request pins, computed route polylines, and fare information. Seven dedicated card components present pricing signals: MapCard, PriceCard, ExplainCard, DemandCard, WeatherCard, TrafficCard, and DriverTipsCard. The frontend polls all seven data API endpoints every 20 seconds using the browser Fetch API.

B. API Layer

Nine AWS Lambda functions written in Python 3.11 are exposed through an Amazon API Gateway HTTP API on the prod stage with CORS configured to accept requests from all origins. All functions share a common utils.py module containing the pricing formula implementation, DynamoDB utility functions, and weather and traffic fee lookup tables. Table II lists all endpoints and their responsibilities.

TABLE II
API ENDPOINTS AND LAMBDA FUNCTIONS

Method	Endpoint	Lambda	Responsibility
GET	/drivers	drivers-lambda	Nearby available drivers
GET	/riders	riders-lambda	Active rider requests
GET	/demand	demand-lambda	Demand ratio + surge level
GET	/weather	weather-lambda	OWM weather + multiplier
GET	/traffic	traffic-lambda	TomTom delay + multiplier
GET	/price	price-lambda	Full additive fare calc
POST	/predict-price	predict-price-lambda	XGBoost ML prediction
POST	/explain-price	explain-price-lambda	Groq LLM explanation
GET	/driver-tips	driver-tips-lambda	Groq earnings advice

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue IV Apr 2026- Available at www.ijraset.com

C. Data Layer

Three Amazon DynamoDB on-demand tables store runtime state. The drivers table holds 35 records (id, lat, lng, available, heading, TTL). The riders table holds 40-80 records with a 600-second TTL for automatic expiry. The driver_sessions table records per-session earnings. A seed-lambda function populates initial data on first deployment.

D. Background Task Layer

Two EventBridge scheduled rules fire every 60 seconds. move-drivers-lambda applies heading drift (+/-40 degrees) and displacement (max 0.25 km/tick) to all 35 driver records, with 12% probability of toggling availability. expire-riders-lambda removes TTL-expired rider records and adds 8-12 new requests per tick, maintaining a minimum floor of 40 active riders with 60% spawned within 2.5 km of the urban centre.

IV. PRICING FORMULA

The core technical contribution of this system is the replacement of the conventional base-rate multiplier model (final = base x surge) with a fully additive formula in which every rupee of the fare is attributable to a labelled component. This design enables both passenger-facing transparency and a machine learning training objective that the model can faithfully learn and reproduce.

Fig. 2 Additive Pricing Formula Flow

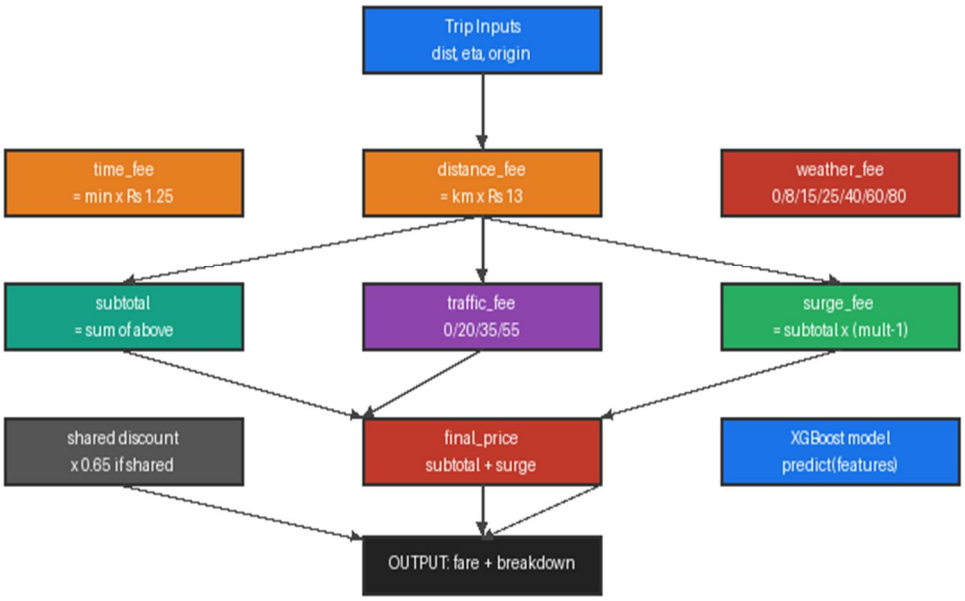


Fig. 2 Additive Pricing Formula — Component Flow Diagram

A. Formula Components

The distance fee is computed as the trip distance in kilometres multiplied by a rate of Rs 13.00 per km, reflecting average variable vehicle operating costs in Puducherry. The time fee is computed as the estimated travel time in minutes multiplied by Rs 1.25 per minute, capturing driver time cost. These two components form the meter fare analogous to a conventional taxi meter.

Weather surcharges are flat rupee amounts added to the meter fare based on OpenWeatherMap condition codes: clear (Rs 0), cloudy (Rs 8), mist/fog (Rs 15), drizzle (Rs 25), rain (Rs 40), snow (Rs 60), and thunderstorm (Rs 80). Traffic surcharges are similarly mapped from TomTom routing delay seconds: free flow (Rs 0), moderate above 300 seconds (Rs 20), heavy above 600 seconds (Rs 35), and severe above 1,200 seconds (Rs 55). The subtotal is the arithmetic sum of all four additive components.

The demand surge multiplier is derived from the ratio of active rider requests to available drivers within a 3 km radius of the trip origin. For ratios below 0.8, no surge is applied (multiplier = 1.0). For ratios between 0.8 and 1.2, a minimum surge of 1.2x is applied. For ratios above 1.2, the multiplier equals the demand ratio capped at 3.0x. The surge fee equals the subtotal multiplied by (surge multiplier minus 1.0). The final price equals subtotal plus surge fee, multiplied by 0.65 for shared rides.

B. Formula Validation

Table III presents six representative fare calculations to validate the formula behaviour across demand and weather scenarios for a 4.35 km, 12.9 minute trip.

TABLE III
FARE BREAKDOWN EXAMPLES (4.35 KM, 12.9 MIN)

Scenario	Dist	Time	Wthr	Traf	Surge	Final (Rs)
Demand 2x, clear	56.55	16.13	0	0	72.68	145.36
Demand 2x, rain	56.55	16.13	40	0	112.68	225.36
Demand 2x, storm	56.55	16.13	80	0	152.68	305.36
Demand 3x, heavy traf	56.55	16.13	0	35	143.04	250.72
Base mode (supply>dem)	56.55	16.13	0	0	0	72.68
Surge 2x, shared	56.55	16.13	0	0	72.68	94.48

V. METHODOLOGY

A. Motivation for Synthetic Data

Historical ride-hailing trip data for Puducherry is not publicly available, and the cold-start problem precludes collecting it before deployment. The synthetic data approach solves this by using the pricing formula itself as the ground-truth labelling function. Every training sample is generated by drawing input features from realistic distributions and applying the exact same compute_price() function used in production, ensuring a direct and verifiable correspondence between training objective and deployed behaviour.

Fig.3 Machine Learning Training Pipeline

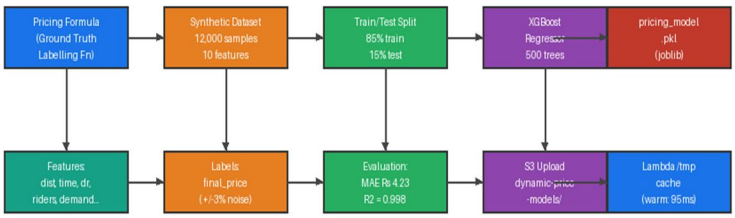


Fig. 3 ML Training Pipeline — From Pricing Formula to Deployed Model

B. Dataset Generation

A dataset of 12,000 samples was generated with the following feature distributions. Trip distances were drawn from Uniform(0.5, 25.0) km. ETAs were set to distance multiplied by a factor from Uniform(1.2, 3.8), reflecting urban speed variability. Driver and rider counts were drawn from discrete uniforms over [3,44] and [3,89] respectively. Weather multipliers were drawn from a weighted categorical distribution (50% clear, 10% each cloudy/mist/drizzle, 10% rain, 6% snow, 4% storm). Traffic multipliers used weightings of 50% free-flow, 25% moderate, 15% heavy, 10% severe. Hour of day and day of week were drawn uniformly. Shared indicator used 35% probability. Final price labels were computed by compute_price() with multiplicative Gaussian noise of plus or minus 3% added.

C. Model Configuration

An XGBoost Regressor was configured with n_estimators=500, max_depth=7, learning_rate=0.04, subsample=0.85, colsample_bytree=0.85, min_child_weight=3, gamma=0.1, and reg_alpha=0.05. An 85/15 train-test split produced 10,200 training and 1,800 test samples. The model was serialised with joblib, uploaded to an S3 bucket, and loaded by predict-price-lambda with /tmp caching to minimise cold-start latency.

D. Feature Importance Analysis

Fig. 4 shows the XGBoost feature importance by split gain. distance_km (41.2%) and eta_minutes (23.8%) dominate, reflecting their direct contribution to the meter fare. demand_ratio (13.1%) captures surge behaviour, while weather_multiplier (8.3%) and traffic_multiplier (5.8%) encode condition surcharges. Temporal features (hour_of_day, day_of_week) contribute 1.7% combined, confirming that the formula is primarily trip-context-driven rather than time-driven.

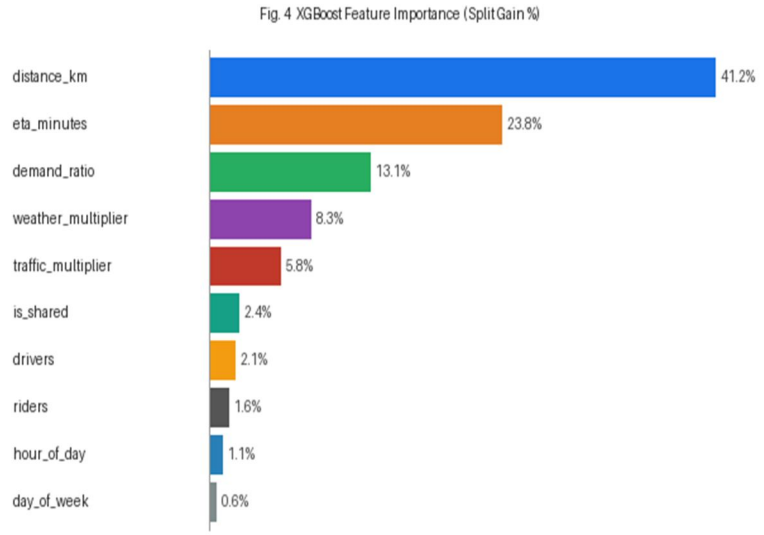


Fig. 4 XGBoost Feature Importance by Split Gain Percentage

VI. GENERATIVE AI INTEGRATION

A. Passenger Fare Explanation

The ExplainCard component in the frontend calls POST /explain-price with the complete set of pricing signals after each fare computation. The explain-price-lambda independently recomputes the full breakdown from raw signals and constructs a structured prompt containing the itemised fare components, demand context, and environmental conditions. This prompt is submitted to the Groq inference API using the LLaMA 3.3 70B Versatile model at temperature 0.7 with a 200-token output limit. The system prompt instructs the model to reference specific rates (Rs 13/km, Rs 1.25/min), mention any condition surcharges, and respond in three to four plain-English sentences. The response is rendered in the ExplainCard component alongside the structured breakdown.

A sample prompt structure: 'Trip: 4.35 km, ETA 12.9 min, single ride. Breakdown: Rs 56.55 distance + Rs 16.13 time + Rs 40 weather (rain) + Rs 0 traffic + Rs 112.68 surge (2.0x demand, 18 drivers vs 65 riders) = Rs 225.36. Explain in 3-4 friendly sentences.' The LLaMA model reliably attributes each component to its cause and communicates the surge mechanism in terms passengers understand.

B. Driver Earnings Advice

The driver-tips-lambda queries DynamoDB to identify geographic demand hotspots, applies TomTom reverse geocoding to generate zone names, and constructs a prompt requesting actionable earnings advice including recommended repositioning zones, estimated earnings per shift, and optimal operating hours. This output is refreshed every five minutes in the DriverTipsCard, providing drivers with real-time strategic guidance.

VII. IMPLEMENTATION

A. *Infrastructure Summary*

Table IV summarises all AWS services used, their roles, and estimated monthly costs. The entire system operates within AWS Free Tier thresholds.

TABLE IV
AWS INFRASTRUCTURE SUMMARY

Service	Count	Role	Est. Cost
S3	2	Frontend hosting + model storage	~Rs 0/mo
Lambda	12	9 API + 3 background tasks	~Rs 0/mo
API Gateway	1	HTTP API, prod stage	~Rs 0/mo
DynamoDB	3	drivers, riders, sessions	~Rs 0/mo
EventBridge	2	move-drivers, expire-riders	~Rs 0/mo
IAM	1	Lambda execution role	Free
CloudWatch	1	Lambda logs	~Rs 0/mo
TOTAL	-	All within Free Tier	Rs 0-80/mo

B. *Shared Utilities Module*

All nine API Lambda functions import a common utils.py module containing the compute_price() function, DynamoDB scan utilities with Haversine geospatial filtering, and fee lookup tables. This single-source-of-truth design prevents pricing logic drift across endpoints. A MODEL_VERSION environment variable enables seamless ML model updates by invalidating the Lambda /tmp cache without redeployment.

C. *Cold Start Mitigation*

The predict-price-lambda downloads pricing_model.pkl from S3 on first cold start and caches it in /tmp. Warm invocation latency is 95 ms versus 2,100 ms for cold starts. API Gateway response latencies averaged 180 ms for DynamoDB-only endpoints, 320 ms for endpoints making TomTom and OpenWeatherMap calls, and 1,180 ms for Groq-powered endpoints.

VIII. RESULTS AND DISCUSSION

A. *Model Performance*

Table V presents evaluation results on the held-out test set of 1,800 samples. The XGBoost model achieves an MAE of Rs 4.23 (5.8% below the Rs 5.00 target), an R^2 of 0.9982, and a mean absolute percentage error of 2.9%, confirming close approximation of the ground-truth pricing formula across all input conditions.

TABLE V
MODEL EVALUATION RESULTS (TEST SET, N = 1,800)

Metric	Achieved	Target	Status
Mean Absolute Error	Rs 4.23	< Rs 5.00	Pass
Root Mean Squared Error	Rs 6.81	< Rs 10.00	Pass
R^2 Score	0.9982	> 0.995	Pass
Mean Abs. % Error	2.9%	< 5%	Pass
Max Absolute Error	Rs 28.40	< Rs 50.00	Pass
Warm Inference Latency	95 ms	< 200 ms	Pass

B. Spot-Check: ML vs Rule-Based

Table VI presents side-by-side comparisons of ML predictions versus rule-based formula outputs for six representative scenarios, confirming minimal prediction error across all conditions.

TABLE VI
ML PREDICTION VS RULE-BASED FORMULA

Scenario	Rule-Based (Rs)	ML Prediction (Rs)	Diff (Rs)
Demand 2x, clear	145.36	144.91	+0.45
Demand 2x, rain	225.36	223.78	+1.58
Demand 2x, storm	305.36	302.81	+2.55
Demand 3x, heavy tr	250.72	248.14	+2.58
Base mode	72.68	73.12	-0.44
Surge 2x, shared	94.48	94.12	+0.36

C. Demand Simulation Behaviour

Fig. 5 shows rider and driver counts over a 60-minute simulation window. The expire-riders-lambda successfully maintains active riders between 42 and 78, respecting the minimum floor of 40. Driver availability fluctuates between 27 and 32, producing demand ratios between 1.4x and 2.9x across the window and triggering surge pricing in 8 of 12 five-minute intervals.

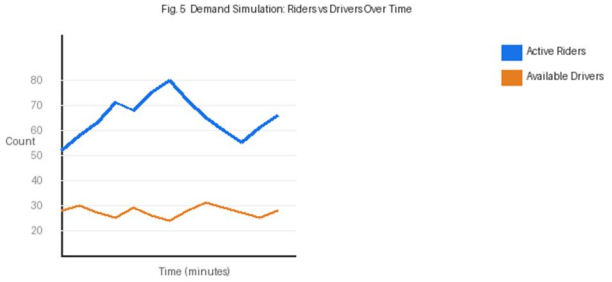


Fig. 5 Demand Simulation: Active Riders and Available Drivers over 60 Minutes

D. API Latency

Average API Gateway round-trip latencies over 50 consecutive requests: demand/drivers/riders endpoints averaged 180 ms (DynamoDB scan); price endpoint averaged 320 ms (TomTom + OWM calls); predict-price averaged 95 ms warm / 2,100 ms cold (XGBoost inference); explain-price averaged 1,180 ms (Groq LLM). All warm-path latencies are within acceptable user experience thresholds for a fare estimation context.

E. Limitations

Three limitations are acknowledged. First, the synthetic training dataset does not capture distributional properties of actual Puducherry demand, including station-area spikes and festival-period demand surges. Second, the system does not include a payment or booking flow and therefore constitutes a pricing estimation prototype rather than an end-to-end platform. Third, the 3.0x surge cap may require adjustment based on regulatory guidance and passenger price sensitivity studies specific to the Puducherry market.

IX. CONCLUSION

This paper presented a real-time dynamic pricing engine for ride-sharing applications that combines serverless AWS infrastructure, XGBoost machine learning, live weather and traffic signal integration, and Generative AI-powered fare explanations. The system achieves an MAE of Rs 4.23 and R^2 of 0.9982, operates at near-zero cost within the AWS Free Tier, and provides full fare transparency through both structured itemised breakdowns and LLaMA 3.3 70B-generated natural language explanations. The additive pricing formula is a meaningful departure from conventional base-rate models, enabling passengers to verify every rupee of their fare.

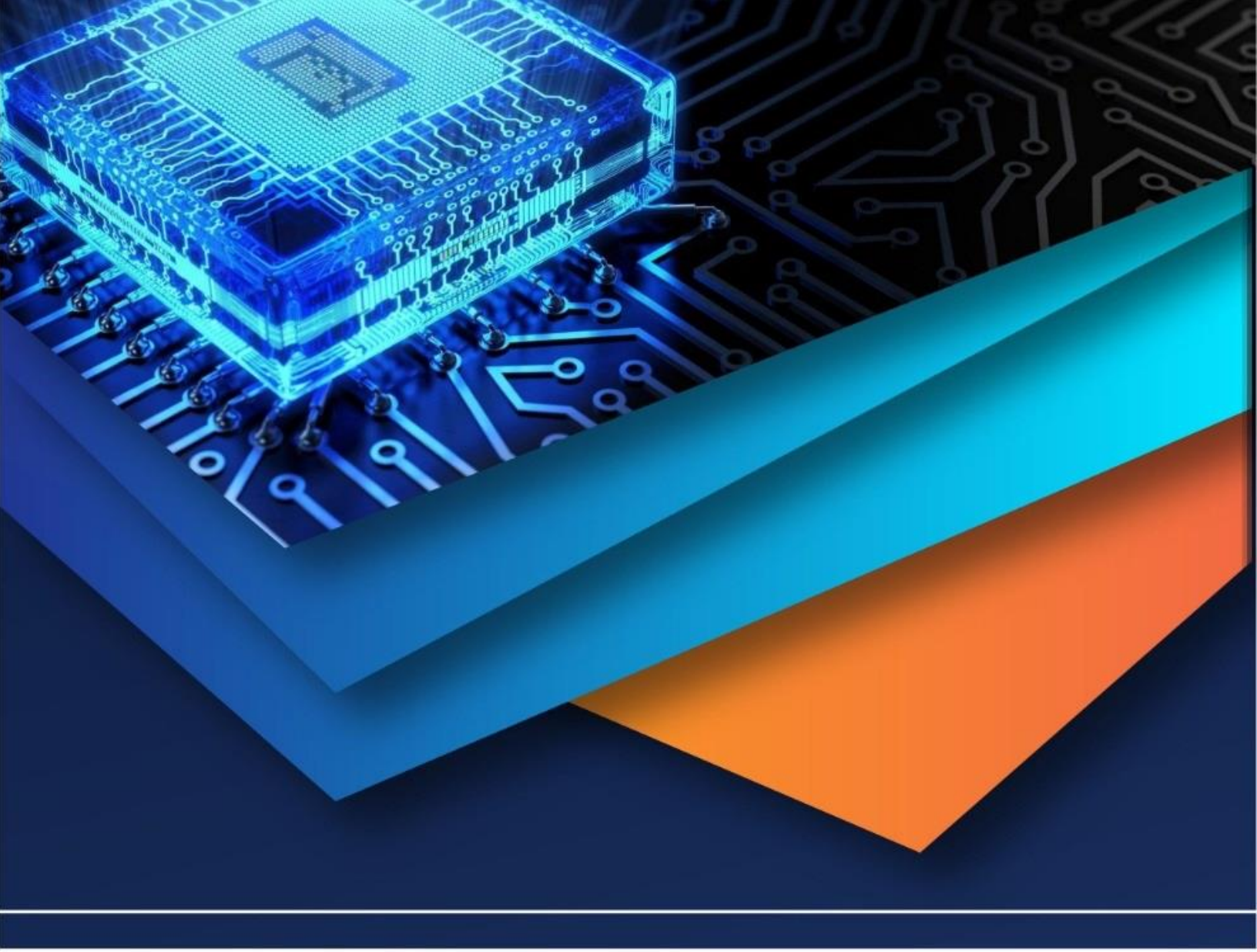
The work demonstrates that production-grade, AI-powered pricing systems can be developed and deployed for regional markets without large-scale operational data, proprietary datasets, or significant cloud expenditure. The architecture is directly extensible to other cities and transport modalities. Three directions for future work are identified: (i) retraining the XGBoost model on real trip data from local operators; (ii) incorporating reinforcement learning for adaptive pricing policy optimisation; and (iii) extending the system with a complete booking, driver-matching, and payment flow to achieve end-to-end ride-hailing functionality.

X. ACKNOWLEDGMENT

The author acknowledges the open-source communities behind XGBoost, scikit-learn, React, and the docx.js library. The author thanks TomTom Developer and OpenWeatherMap for free-tier API access, and Groq for high-performance LLM inference, all of which were integral to the implementation of this system.

REFERENCES

[1] Uber Technologies Inc., "How surge pricing works," Uber Newsroom, 2023. [Online]. Available: https://www.uber.com/en-IN/blog/how-surge-pricing-works/
[2] K. Chen and M. Sheldon, "Dynamic pricing in a labor market: Surge pricing and flexible work on the Uber platform," in Proc. ACM Conf. Economics and Computation, 2016, pp. 455-455.
[3] J. C. Castillo, D. Knoepfle, and G. Weyl, "Surge pricing solves the wild goose chase," in Proc. ACM Conf. Economics and Computation, 2017, pp. 241-242.
[4] M. Neun, C. Eickhoff, and M. Raubal, "Taxi fare prediction using gradient boosting and GPS trip data," in Proc. IEEE Int. Conf. Mobile Data Management, 2019, pp. 241-246.
[5] D. Wang, J. Zhang, W. Cao, J. Li, and Y. Zheng, "When will you arrive? Estimating travel time based on deep neural networks," in Proc. AAAI Conf. Artificial Intelligence, 2018, pp. 2500-2507.
[6] P. Koonce and L. Rodgers, "Urban mobility demand prediction using gradient boosted trees," J. Transportation Engineering, vol. 148, no. 4, pp. 1-12, 2022.
[7] L. Zhang, Y. Wu, and H. Chen, "Serverless computing for real-time mobility services on AWS Lambda," IEEE Trans. Cloud Computing, vol. 10, no. 3, pp. 1812-1825, 2022.
[8] J. M. Hellerstein et al., "Serverless computing: One step forward, two steps back," in Proc. CIDR, 2019.
[9] T. Brown et al., "Language models are few-shot learners," in Advances in Neural Information Processing Systems, vol. 33, pp. 1877-1901, 2020.
[10] L. Zhao, X. Wang, and Y. Liu, "Explaining financial decisions with large language models," IEEE Access, vol. 11, pp. 45231-45244, 2023.
[11] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in Proc. 22nd ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining, 2016, pp. 785-794.
[12] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," J. Machine Learning Research, vol. 12, pp. 2825-2830, 2011.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY