WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

Karbon-Ed: A Decoupled Microservice Architecture for Democratizing Carbon Footprint Matching in MSMEs

Vedant Tarale[1], Nabhya Gabhane[2], Ritik Nagpure[3], Paras Katolkar[4], Dr. Archana Potnurwar[5], Prof. Lalita Charpe[6]

Department of Information Technology

[5] Project guide

[6] Project Co-guide

DOI : https://doi.org/10.55248/gengpi.07.1026.34082

ABSTRACT :

Micro, Small, and Medium Enterprises (MSMEs) rarely take part in Voluntary Carbon Markets (VCMs). Offset prices change between registries and over time, verification standards differ, and enterprise carbon accounting tools assume a dedicated sustainability team. Web applications built as a single server process add a technical barrier: price normalization runs in the same runtime that serves the interface, and the interface slows when market data changes quickly. This paper describes Karbon-Ed, a decoupled microservice web application that matches an MSME's carbon footprint and preferences to offset projects. A React, Vite, and Tailwind CSS client collects footprint inputs and generates offset certificates in the browser with jsPDF after a successful Stripe redirect. A Node.js and Express.js gateway handles routing, Bcrypt password hashing, JSON Web Token (JWT) authentication, role checks, registry verification, and Stripe checkout. A Python Flask service receives each POST /api/recommend request, reads credit records from MongoDB Atlas through PyMongo into Pandas DataFrames, scales the numeric features with the Scikit-Learn MinMaxScaler, and ranks projects with a Multi-Criteria Decision Analysis (MCDA) score whose price and capacity weights the user selects. The gateway reaches the database through Mongoose. A functional validation of the prototype shows that the gateway executes no vector operations, that the server allocates no memory for certificate rendering, and that scaled features stay inside [0, 1]. The paper reports no latency measurement, and Section 8.3 defines the measurements planned for later work.

Keywords: Voluntary Carbon Markets; MSME; Microservice Architecture; Min-Max Normalization; Multi-Criteria Decision Analysis; Flask; Node.js; MongoDB

1. INTRODUCTION

1.1 Background

MSMEs make up most businesses worldwide, yet few of them buy carbon offsets. Smith and Taylor place VCMs within corporate environmental strategy [1], and Patel and Gomez study why small enterprises adopt offsets less often than large ones [2]. Zhao and Liu connect price volatility to limited transparency in global offset registries [3]. For an owner with a fixed budget, a price that moves between planning and payment is a financial risk [4]. Enterprise accounting tools add a second barrier. Baker describes a digital divide between those tools and MSME conditions [5], and Kumar and Sharma propose interactive educational platforms as a response [6].

Existing carbon accounting web applications are usually monolithic. One server process authenticates users, queries the database, normalizes prices, and renders documents. Node.js runs JavaScript on a single thread, so a long synchronous computation blocks every pending request. The user sees the delay as interface latency, and that delay discourages the small business owners the platform is meant to serve.

1.2 Contributions

The contribution of this paper is architectural integration. It proposes no new normalization algorithm and does not claim to have invented service decomposition. The paper:

- presents a four-tier architecture (React client, Express gateway, Flask intelligence service, MongoDB Atlas) for carbon offset matching;
- defines a split data access model, with Mongoose in the gateway and PyMongo in the Flask service;
- describes a pipeline that applies the Scikit-Learn MinMaxScaler to credit data and ranks projects with a user-weighted MCDA score;
- specifies client-side certificate generation with jsPDF, triggered only by a successful Stripe redirect;
- defines the evaluation planned for the properties that the prototype does not yet measure.

2. LITERATURE REVIEW

The 25 reviewed sources fall into four groups. Smith and Taylor [1] and Patel and Gomez [2] place VCMs in corporate strategy and identify the barriers that keep small enterprises out of them. Zhao and Liu [3] examine price volatility and transparency across offset registries, and Andersen and Nielsen [4] address budget management for SMEs in voluntary environmental markets. Baker [5] compares enterprise carbon accounting tools with MSME conditions, Kumar and Sharma [6] propose interactive educational platforms, and Kumar and Singh [7] apply MCDA to supplier selection in MSMEs. By title and venue, these works describe the market problem and a decision method, and none describes a web architecture.

The microservice sources start from the definition of Fowler and Lewis [13]: small services, each in its own process, communicating through lightweight mechanisms. Thönes [9] and Dragoni et al. [11] trace the history of the style, Newman [12] covers fine-grained design, Pahl and Jamshidi [14] map the research field, and Taibi et al. [15] report the motives and difficulties of adoption. Martinez and Lopez [8] describe patterns for high-performance web applications, and Fielding and Taylor [10] analyze the REST style that the gateway follows.

Among the web technology sources, Li and Chen [16] compare Node.js and Flask for offloading machine learning workloads, which is the closest precedent for the split between gateway and service. Ojo and Thompson [17] study JWT authentication, Provos and Mazières [18] introduce the adaptive password scheme behind Bcrypt, Bradley and Liu [19] report rendering metrics for React and Vite single-page applications, and Richards [20] analyzes memory use and performance of client-side PDF generation. Chodorow [21] documents MongoDB, and Al-Sharafi et al. [22] study continued use of cloud services among NoSQL database users.

For data preparation, Wang and Zhang [23] study normalization and preprocessing in volatile environmental markets, Pedregosa et al. [24] describe Scikit-Learn, and McKinney [25] describes the Pandas data structures.

3. RESEARCH GAP AND PROBLEM STATEMENT

Each group above is studied separately. By their titles, the 25 sources describe no system that joins a decoupled gateway and numerical service, price normalization, MCDA ranking, and client-side certificate generation for MSME buyers. The gap argued here is that combination, together with the lack of any measurement of its effect on interface responsiveness. The review covers the supplied sources only and is not a systematic search.

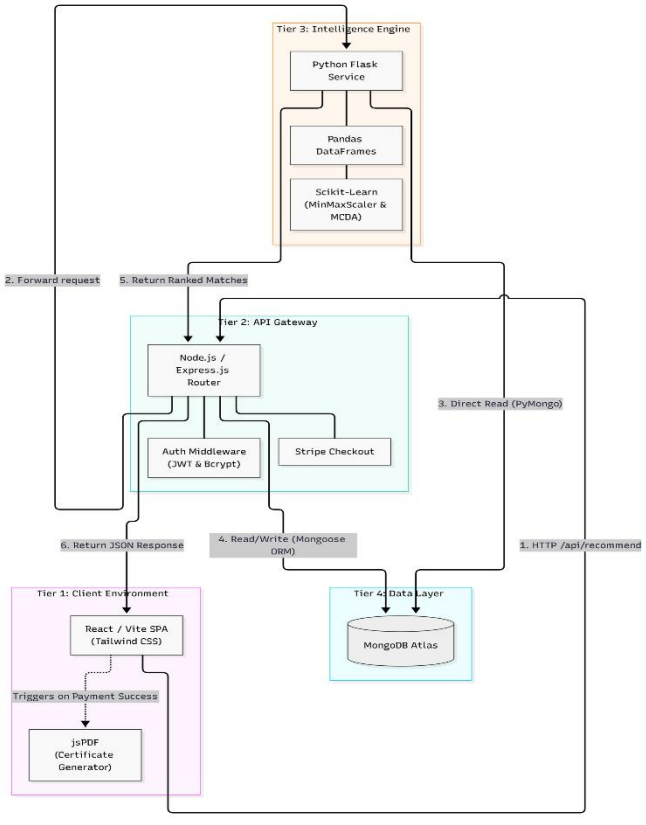
An MSME buyer enters a footprint and a preference and expects a ranked list quickly. A monolithic server computes the list inside the request path, so the cost of each recomputation falls on every connected user. A decoupled server moves the computation to a separate process and adds one network hop. The study asks five questions.

- RQ1: Does the Flask offload keep the Node.js event loop responsive during high-frequency market polling?
- RQ2: Does Min-Max scaling keep interface values inside a fixed interval when raw prices change?
- RQ3: Does client-side jsPDF remove certificate rendering load from the server?
- RQ4: Does direct PyMongo access shorten the read path of the intelligence service?
- RQ5: How does the system behave when the Flask service is unavailable?

4. SYSTEM ARCHITECTURE

Karbon-Ed has four tiers (Fig. 1). Each tier runs as a separate process or environment and communicates over HTTP or a database driver.

Fig. 1. Four-tier architecture of Karbon-Ed. Numbered arrows show the recommendation path: (1) HTTP request to /api/recommend, (2) request forwarded to Flask, (3) direct PyMongo read, (4) Mongoose read and write, (5) ranked matches returned to the gateway, (6) JSON response returned to the client.



Tier 1 is the client environment, a React single-page application built with Vite and styled with Tailwind CSS. It holds interface state, stores the JWT in LocalStorage, and calls jsPDF only after Stripe redirects the browser to the payment success route. Tier 2 is the API gateway, a Node.js and Express.js server. Its router applies the authentication middleware (JWT verification and Bcrypt hashing), verifies registry information, and creates Stripe checkout sessions. Tier 3 is the intelligence engine, a standalone Python Flask service that uses Pandas DataFrames and Scikit-Learn for the MinMaxScaler and the MCDA ranking. Tier 4 is the data layer, a MongoDB Atlas cluster.

The numbered arrows in Fig. 1 trace one recommendation. The client sends an HTTP request to /api/recommend (1), and the gateway forwards it to the Flask service (2). Flask reads credit records directly from MongoDB through PyMongo (3), while the gateway reads and writes through the Mongoose ORM (4). Flask returns the ranked matches to the gateway (5), and the gateway returns a JSON response to the client (6). A purchase leaves this path. The gateway creates the Stripe session, the user pays on the Stripe page, and the client builds the certificate when the success redirect arrives.

Authentication, checkout, and certificate generation never call the Flask service, so a failure of that service affects recommendation requests only. The two backend services also scale on different resources: the gateway with the number of concurrent connections, and the intelligence engine with the number of credit records and criteria.

5. METHODOLOGY

5.1 Footprint Input

The client calculator takes annual miles driven, annual flights taken, and the monthly energy bill in rupees. The Calculate Tonnage action converts these three inputs into a target offset requirement in tons of CO_2e. The example in Fig. 2(a) returns 9.10 tons for 12,000 miles, 5 flights, and a monthly bill of ₹12,000.

5.2 Min-Max Normalization

Credit records differ in unit and range: price is in rupees per ton and capacity is in tons. The Flask service applies the Scikit-Learn MinMaxScaler [24] to each numeric feature j of the credit matrix:

$$x'_{ij} = (x_{ij} - min_j) / (max_j - min_j) \quad (1)$$

where min_j and max_j are the historical bounds of feature j. The minimum maps to 0 and the maximum maps to 1. Normalization of this kind is a standard preprocessing step for volatile market data [23]. For a feature with zero range, the Scikit-Learn implementation returns 0. Missing values pass through the scaler unchanged and must be removed before scoring. A value outside the stored bounds falls outside [0, 1] unless the transform clips it.

5.3 MCDA Ranking

The ranking stage follows the multi-criteria approach that Kumar and Singh apply to MSME supplier selection [7]. The interface exposes two criteria, price and capacity. A lower price is better and a higher capacity is better, so the service converts price to benefit form:

$$v_{ip} = 1 - p'_i, \quad v_{ic} = c'_i \quad (2)$$

where p'_i and c'_i are the scaled price and scaled capacity of credit i. The matching score is a weighted sum:

$$S_i = w_p\, v_{ip} + w_c\, v_{ic}, \quad w_p + w_c = 1 \quad (3)$$

The user selects the weights in the Tune the AI Recommendation Engine control. The Balanced option in Fig. 2(a) sets $w_p = w_c = 0.5$. Both v values lie in [0, 1], so every S_i also lies in [0, 1]. The service sorts credits by S_i in descending order and returns the ranked list to the gateway. For n credits and m criteria, scaling and scoring cost O(nm) operations and sorting costs O(n log n).

6. IMPLEMENTATION

Table I lists the stack. The gateway manifest names the package carbon-exchange, uses the CommonJS module type, and defines no test script. Express 5 forwards a rejected promise from an asynchronous handler to the error middleware. The manifest lists no rate-limiting or security-header package.

TABLE I. SOFTWARE STACK

Package	Version	Role
express	^5.2.1	REST routing
mongoose	^9.7.4	ORM, CRUD
bcrypt	^6.0.0	Password hashing
jsonwebtoken	^9.0.3	JWT signing
stripe	^22.3.2	Checkout sessions
cors	^2.8.6	Origin policy
dotenv	^17.4.2	Environment variables
React, Vite, Tailwind CSS, Axios, jsPDF	Client manifest	Interface, certificates

Package	Version	Role
Flask, PyMongo, Pandas, Scikit-Learn	Service requirements file	Intelligence engine

Gateway versions are taken from the project package.json.

A session starts with registration or login. The gateway hashes the password with Bcrypt or compares it with the stored hash, signs a JWT, and returns it. The client keeps the token in LocalStorage and attaches it to later requests, and the middleware verifies the token and the user role before any route handler runs. After a recommendation (Section 4, steps 1 to 6), the user selects a project. The gateway verifies registry information, creates a Stripe checkout session, and records the purchase in TransactionsCollection through Mongoose. When Stripe redirects to the success route, the client builds the certificate with jsPDF.

Fig. 2 shows the interface. In Fig. 2(a), the user enters the footprint inputs and receives the target offset requirement. The recommendation control sets the weights, and the Find Perfect Offset Match button calls the recommendation endpoint. The result panel names the recommended project and its originator, gives its available capacity (18 tons) and its price (₹56 per ton), and repeats the selected weights and the 9.1 ton footprint. In Fig. 2(b), the Ledger and Certificates page lists each secured asset with its originator, the impact value in tons, and the capital deployed in rupees.

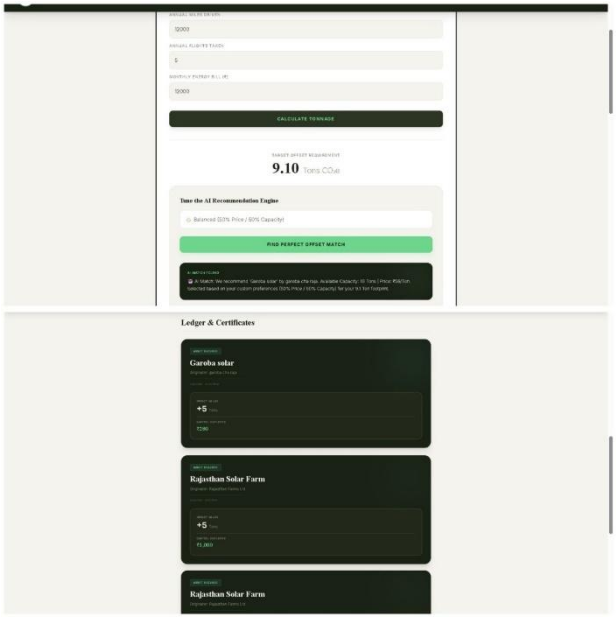


Fig. 2. Karbon-Ed user interface. (a) Footprint calculator, recommendation control, and AI match result. (b) Ledger and certificates.

7. DATA LAYER AND SECURITY

MongoDB Atlas holds three collections (Table II). The gateway defines Mongoose schemas for each, which gives it field types and validation on every write. The Flask service uses PyMongo and bypasses the ORM. A query returns Python dictionaries, and Pandas builds the DataFrame from the list of dictionaries. The service converts each ObjectId to a string before the data leaves the process. The gateway reads and writes all three collections, and the Flask service reads CarbonCreditsCollection only. Every write therefore passes through the Mongoose schemas. The cost of this rule is that two code bases describe the same documents, so a schema change in the gateway must be mirrored in the field names that Flask reads.

TABLE II. COLLECTIONS

Collection	Fields visible in the prototype
UsersCollection	Credentials (Bcrypt hash), role
CarbonCreditsCollection	Project title, originator, price per ton (₹), available capacity (tons)
TransactionsCollection	User, project, quantity (tons), capital deployed (₹), acquisition date

Field lists are limited to those shown in the interface. The full schemas are in the repository.

Passwords are stored as Bcrypt hashes, a scheme with an adjustable cost parameter [18]. Authentication uses a signed JWT that the middleware verifies on each request [17], and role middleware compares the role claim with the permission of the route. The client stores the token in LocalStorage, which any script in the page origin can read, so a cross-site scripting flaw would expose it. An HttpOnly cookie removes that exposure and requires a cross-site request forgery defense. Card data stays with Stripe: the user enters it on the Stripe page, and the gateway receives the session outcome only. The cors package restricts the origins that may call the gateway, and dotenv keeps secrets out of the repository. The Flask service has no authentication layer of its own, so it must be reachable only from the gateway through a private address or a firewall rule.

8. RESULTS AND DISCUSSION

8.1 Functional Validation

The prototype was checked against three design properties. No latency or throughput benchmark was run.

Event-loop preservation. The Pandas and Scikit-Learn operations run only in the Flask process. During high-frequency market polling, the gateway forwards requests and awaits responses and executes no vector operation. This division follows the offloading approach compared in [16].

Certificate rendering. The client calls jsPDF after the success redirect, and the gateway renders no document. The server allocates no memory for PDF construction, and the memory cost moves to the user's browser, a cost analyzed in [20].

Interface stabilization. Each feature is scaled against its historical bounds, so values that reach the interface lie in [0, 1], and a price change moves them in proportion to its share of the historical range. The ledger in Fig. 2(b) shows why this matters. It records 5 tons for ₹280, which is ₹56 per ton, and 5 tons for ₹1,000, which is ₹200 per ton. Unit prices in the same ledger differ by a factor of about 3.6, and scaling places both on one interval.

8.2 Discussion

The decoupling decision rests on a narrow claim: a single-threaded gateway is a poor place for CPU-bound numerical work, and a separate process removes that work from the request path. The prototype supports the structural form of this claim, since the design assigns every vector operation to the Flask service. It does not support a quantitative form.

The trade-offs are direct. The second service adds a network hop to every recommendation and a deployment unit to monitor, difficulties of the kind that Taibi et al. report for microservice adoption [15]. Client-side rendering moves memory use to the user device. Min-Max scaling is sensitive to outliers: one extreme price enlarges the range and compresses every other scaled value toward one end of the interval. The preset weights, such as the 50/50 Balanced option, express assumptions about buyer preferences, and no study with MSME users has validated them.

8.3 Planned Evaluation

Five scenarios will test the research questions. A recorded price series replayed at high frequency while simulated users request recommendations tests RQ1, and the same series with a price spike tests RQ2. A fixed number of certificates issued under server-side and client-side rendering tests RQ3. A growing CarbonCreditsCollection read through PyMongo and through an object-mapping layer tests RQ4. Stopping, delaying, and corrupting Flask responses during traffic tests RQ5. The measured quantities are event-loop delay from the Node.js perf_hooks module, response time at the client, server resident memory, read time, and error rate. The baselines are a monolithic Express server that normalizes in process and a decoupled variant with server-side PDF rendering. A System Usability Scale questionnaire will be given to MSME participants, and the sample size will be reported with the results.

9. LIMITATIONS AND FUTURE SCOPE

- The validation is functional. The system has no load test, latency measurement, memory measurement, or user study.
- The review covers the 25 supplied sources and is not a systematic search.
- Credit data depends on the records in CarbonCreditsCollection. Live registry coverage and price freshness depend on integrations that are not built.
- The client keeps the JWT in LocalStorage, the gateway manifest has no rate-limiting package, and the Flask service relies on network isolation.
- Automated registry polling will pull prices and project data from live carbon registries through scheduled REST calls and store a source and timestamp on each record.
- Integration with carbon accounting APIs will supply emission estimates directly and replace the manual footprint inputs.
- A forecasting model in the Flask service will estimate near-term credit prices from historical series, and the interface will show the estimate beside the current price.

10. CONCLUSION

Karbon-Ed assigns authentication, checkout, and certificate rendering to the gateway and the browser, and it assigns normalization and ranking to a separate Flask service. The prototype shows that the gateway runs no vector operations, that the server allocates no memory for certificates, and that scaled features reach the interface inside [0, 1]. Its recommendation screen and ledger work with INR prices and user-selected price and capacity weights. Whether the decoupled design improves responsiveness in measurable terms remains open until the evaluation of Section 8.3 is carried out.

REFERENCES

[1] J. Smith and R. Taylor, "The role of voluntary carbon markets in corporate environmental strategy," *Journal of Cleaner Production*, vol. 291, p. 125912, 2021.

[2] K. Patel and M. Gomez, "Barriers to sustainability: Why small enterprises lag in carbon offset adoption," *Business Strategy and the Environment*, vol. 31, no. 4, pp. 1845–1860, 2022.

[3] H. Zhao and Y. Liu, "Price volatility and market transparency in global carbon offset registries," *Energy Economics*, vol. 92, p. 104980, 2020.

[4] L. Andersen and S. Nielsen, "Financial risk and budget management for SMEs in voluntary environmental markets," *Ecological Economics*, vol. 204, p. 107655, 2023.

[5] R. Baker, "The digital divide in corporate carbon accounting: Enterprise tools vs. MSME realities," *Environmental Science & Policy*, vol. 124, pp. 110–118, 2021.

[6] V. Kumar and P. Sharma, "Democratizing green economics through interactive educational platforms," *Computers and Education*, vol. 182, p. 104470, 2022.

[7] P. Kumar and R. Singh, "Multi-Criteria Decision Analysis (MCDA) for sustainable supplier selection in MSMEs," *Sustainability*, vol. 13, no. 15, p. 8490, 2021.

[8] C. Martinez and D. Lopez, "Microservice architecture patterns for high-performance web applications," *Journal of Systems and Software*, vol. 198, p. 111580, 2023.

[9] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, p. 116, 2015.

[10] R. T. Fielding and R. N. Taylor, "Principled design of the modern Web architecture," *ACM Transactions on Internet Technology*, vol. 2, no. 2, pp. 115–150, 2002.

[11] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*, 2017, pp. 195–216.

[12] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. Sebastopol, CA, USA: O'Reilly Media, 2015.

[13] M. Fowler and J. Lewis, "Microservices: A definition of this new architectural term," MartinFowler.com, 2014.

[14] C. Pahl and P. Jamshidi, "Microservices: A systematic mapping study," in *Proc. 6th Int. Conf. Cloud Computing and Services Science (CLOSER)*, 2016.

[15] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motives, and challenges for microservice adoption: An empirical study," *Information and Software Technology*, 2017.

[16] M. Li and Y. Chen, "Offloading machine learning workloads in web architectures: A comparative study of Node.js and Python Flask," *IEEE Access*, vol. 9, pp. 120456–120468, 2021.

[17] A. Ojo and M. Thompson, "Secure state management and authentication in decentralized web applications using JSON Web Tokens (JWT)," *Journal of Information Security and Applications*, vol. 61, p. 102941, 2021.

[18] N. Provos and D. Mazières, "A future-adaptable password scheme," in *Proc. USENIX Annual Technical Conf.*, 1999.

[19] S. Bradley and E. Liu, "Performance metrics of client-side rendering in Single Page Applications (SPAs) using React and Vite," *Journal of Web Engineering*, vol. 19, no. 2, pp. 145–162, 2022.

[20] T. Richards, "Client-side PDF generation in modern JavaScript frameworks: Memory allocation and performance," *Software: Practice and Experience*, vol. 51, no. 8, pp. 1750–1765, 2021.

[21] K. Chodorow, *MongoDB: The Definitive Guide*, 3rd ed. Sebastopol, CA, USA: O'Reilly Media, 2020.

[22] M. A. Al-Sharafi, R. A. Arshah, and E. Abu-Shanab, "Factors affecting the continuous use of cloud computing services from the perspective of NoSQL database users," in *Proc. IEEE Int. Conf. Smart Instrumentation, Measurement and Applications*, 2017.

[23] X. Wang and L. Zhang, "Data normalization and machine learning preprocessing techniques in volatile environmental markets," *Expert Systems with Applications*, vol. 168, p. 114250, 2021.

[24] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[25] W. McKinney, "Data structures for statistical computing in Python," in *Proc. 9th Python in Science Conf.*, 2010.