Archives available at journals.mriindia.com

International Journal on Advanced Computer Theory and Engineering

ISSN: 2319-2526

Volume 15 Issue 01, 2026

QoSCollab: A Token-Governed SaaS Architecture for Real-Time QoS Testing, ML-Based Efficiency Prediction, and Service Recommendation

[1]Praveenkumar Arjun Patel, [2]Vishal Raju Honde, [3]Sanket Vijay Jadhav, [4]Saniya Salim Killedar, [5]Poonam Meghraj Mundhe

[1] *Assistant Professor, Bharati Vidyapeeth's College of Engineering, Kolhapur Maharashtra India*
[2,3,4,5] *Student, Bharati Vidyapeeth's College of Engineering, Kolhapur Maharashtra India*

Peer Review Information

Submission: 18 March 2026

Revision: 05 April 2026

Acceptance: 27 April 2026

Keywords

QoS Monitoring, Saas Architecture, Machine Learning Regression, Ensemble Methods, Edge Functions, Token-Based Billing, Real-Time Systems, Serverless Orchestration, Service Recommendation.

Abstract

Quality of Service (QoS) management for web services is usually over the place. It is spread across testing tools, analytics dashboards, predictive models, and billing systems. This paper is about QoS Collab, an integrated Software-as-a-Service (SaaS) platform that unifies real-time QoS testing, makes predictions based on machine-learning, service recommendation and comparison, and token-governed execution within a single deployable architecture. The system is implemented using a React/TypeScript frontend, a Supabase backend comprising PostgreSQL, row-level security (RLS), edge functions, and real-time channels, and a FastAPI model-serving microservice. QoS testing checks latency and uptime and throughput and failure-sensitive refund logic. Experimental evaluation demonstrates that a baseline Linear Regression pipeline achieves a holdout R^2 of 0.5350 (MAE = 1.4590, RMSE = 1.8122), whereas an optimized ensemble pipeline yields R^2 = 0.6804 (MAE = 1.0706, RMSE = 1.3324), representing a 27.2% improvement in explained variance and a 26.6% reduction in MAE. Platform-level workflows confirm the practical feasibility of combining observability, predictive analytics, and consumption-based billing governance in a unified operational design. The study also found some things to consider when making production grade Machine Learning-driven Quality of Service systems.

Introduction

Cloud services and third-party application programming interfaces (APIs) are important for modern digital products to work well [1], [2]. When these services do not work properly or unavailability directly impacts end-user experience and can impose significant operational and financial costs on dependent organisations [2]. Although a broad ecosystem of monitoring, analytics, and billing tools exists, these components are typically deployed as isolated subsystems, creating three interrelated operational problems.

First, the peoples who operate these services have to look at a lot of screens to figure out what is going on which takes a lot of time and effort. Second, the predictions that these tools make are not very accurate because they are not connected to the testing of the services. Third, when tests fail, resource consumption is often charged in full, creating billing inequity that degrades user trust.

To address these gaps, this paper introduces QoSCollab, a unified SaaS platform that integrates real-time QoS testing, ML-based efficiency prediction, service recommendation and comparison, and token-governed billing within a single runtime and data architecture.

1. Objectives

This work pursues three primary objectives: (1) The first goal is to create a system that design and implement an end-to-end architecture that tightly couples service observability, predictive inference, and logic; (2) to quantify the predictive performance gains achievable through feature engineering and ensemble optimization; and (3) to validate the practical utility of the integrated model under representative operational conditions.

2. Contributions

The principal contributions of this work are as follows:

1) A four-plane SaaS architecture (client, edge orchestration, data/security, ML inference) that enables to get the information at real-time and helps our system work properly when we are testing, prediction, and billing subsystems.
2) A failure-aware, cache-sensitive token cost model that figures partial refunds proportional to test-phase completion, ensuring billing equity under timeout, network failure, and error conditions.
3) An enhanced machine learning that combines different models and uses the best parts of each one This makes system works effieciently to gives us more accurate results.
4) An end-to-end experimental evaluation encompassing baseline and optimised pipelines with holdout and cross-validation protocols, accompanied by a discussion of threats to construct, internal, external, and reproducibility validity.
5) Identification of deployment prerequisites for production readiness, including model registry management, drift detection, billing audit trails, and CI/CD schema reproducibility.

3. Paper Organisation

Section II surveys related work across QoS monitoring, ML-based prediction, service recommendation, and managing bills. Section III describes the QoSCollab system architecture in detail. Section IV goes into details about how tokens cost, what the cache policy is, how refunds work and the machine learning pipeline of the QoSCollab system. Section V presents the experimental setup and results. Section VI discussion about what the QoSCollab system means from a point of view what is good about it. Section VII classifies threats to validity. Section VIII concludes the paper and outlines future research directions

Related Work

1. QoS Monitoring and Performance Prediction

Early QoS research focused on things like how long it takes for something to happen and how often it is working properly monitoring of scalar metrics such as latency and uptime [3]. Zheng et al. characterized the statistical distributions of real-world web service QoS attributes and demonstrated that these metrics exhibit significant skew and motivating data-driven modelling approaches [4]. Li et al. subsequently applied collaborative filtering to QoS prediction in cloud environments, exploiting user-service interaction matrices to interpolate missing observations [5]. Wu et al. employed a two-phase K-means clustering strategy to group services by behavioral similarity prior to regression, improving holdout accuracy relative to global models [6]. A limitation shared by these approaches is their decoupling from operational testing infrastructure: predictions are generated offline from historical logs rather than from continuously refreshed, live test results.

2. Service Recommendation

Service Metadata-driven keyword matching has been used to recommend services [7], collaborative filtering on the history of user-service interactions [5], and hybrid scoring algorithms that integrate user choice signals with anticipated QoS [7]. A preference-inference framework that weights services based on past user interaction and expected reliability was proposed by Zhang et al. [7]. However, most of these systems are not responsive to short-term service degradation situations since they work against static service catalogues and do not use real-time test results in the recommendation score.

3. Usage-Based Billing and Resource Governance

As businesses look to match prices with measured value, consumption-based invoicing for cloud and API services has garnered a lot of interest. SLA-aware monitoring frameworks for cloud environments were established by Chen and Zhou, who emphasised the necessity of billing systems that represent observed rather than contracted service levels. [8]. Kumar et al. strengthened the argument for outcome-sensitive cost models by quantifying the impact of API delay on revenue and experience[9]. Despite this body of work, current billing systems usually charge per invocation regardless of the result of the test, neglecting to take timeout events or partial failures into account.

4. Serverless and Edge-Based Orchestration

For per-user, multi-tenant applications, serverless and edge-based execution paradigms have become scalable substrates. [10], [11]. A large cloud provider's serverless workload patterns were described by Shahrad et al., who showed that bursty, latency-sensitive jobs are best suited for fine-grained, event-driven function activation. [11]. Sewak and Singh emphasised the cost-effectiveness and ease of use of function-as-a-service approaches for instrumentation pipelines. [10]. Serverless edge functions are a perfect fit for QoS test orchestration because of these architectural characteristics.

5. Research Gap

No previous platform integrates QoS testing, ML-based efficiency prediction, service recommendation, and outcome-sensitive payment governance inside a single runtime and data architecture, despite advancements in each of these discrete disciplines. By offering tight operational integration across all four subsystems, QoSCollab is intended to close this gap and enable feedback loops that are not possible when these issues are handled separately.

System Architecture

The Client Layer, Edge Orchestration Layer, Data and Security Layer, and ML Inference Layer are the four interacting planes that make up the QoSCollab architecture. Fig. 1 shows the high-level interactions between these planes.

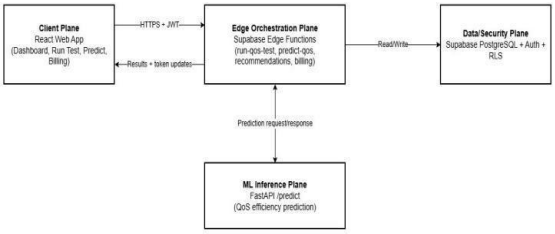


Fig. 1. High-level system architecture of QoSCollab, showing the four interacting planes: client, edge orchestration, data/security, and ML inference.

1. Client Layer

React 18 and TypeScript are used to implement the frontend as a single-page application (SPA). Dashboard, Test Execution, Prediction, Analytics, Recommendations, Comparison, and Billing/Profile are its seven functional modules. Before rendering any sensitive modules, authenticated routes impose JSON Web Token (JWT) validation. A persistent widget that subscribes to server-sent event (SSE) streams released by the edge tier presents real-time token balance and consumption-cycle statistics. Perceived delay during test submission and prediction request procedures is reduced by client-side optimistic state updates..

2. Edge Orchestration Layer

Supabase Edge Functions running top the Deno runtime host all stateful business logic. This design decision offers sub-100 ms cold-start latency, per-user isolation, and automatic scaling without requiring manual infrastructure provisioning. [10], [11]. The orchestration layer consists of seven functions: (1) run-qos-test, which manages the entire token lifecycle and performs active service checks; (2) predict-qos, which logs predictions and validates ML inference requests; (3) get-recommendations, which generates keyword and history-weighted service rankings; (4) compare-services, which aggregates multi-metric QoS comparisons across a nominated service set; (5) payments-create-order and (6) payments-verify, which manage token top-up workflows; and (7) token-stream, which transmits real-time token state to connected clients via SSE.

3. Data and Security Layer

A PostgreSQL instance hosted on Supabase is responsible for managing persistent state. Tests, qos_predictions, web_services,service_recommendations, model_feedback, user profiles, token_transactions, payments, topup_records, and efficiency_logs are examples of core tables. All user-owned tables are subject to Row-Level Security (RLS) restrictions, which guarantee that each authorised session can only see and alter its own entries. Supabase's admin policy primitives further limit administrative changes to the service catalogue. This dual-layer access control approach lowers the attack surface for privilege escalation and does away with the requirement for application-level permission checks.

4. ML Inference Layer

A FastAPI microservice supported by scikit-learn loads model artefacts at startup [12]. The service offers a single /predict endpoint that takes a five-dimensional feature vector, applies the preprocessing pipeline that has been trained, and outputs a normalised efficiency score between 0 and 100. Future drift detection and longitudinal accuracy analysis are made possible by the persistence of each inference request to the qos_predictions database. Because of its low serialisation overhead and inherent support for sklearn Pipeline objects—which combine feature scaling and model

inference into a single reusable artifact—the FastAPI/scikit-learn stack was chosen

5. Real-Time Consistency

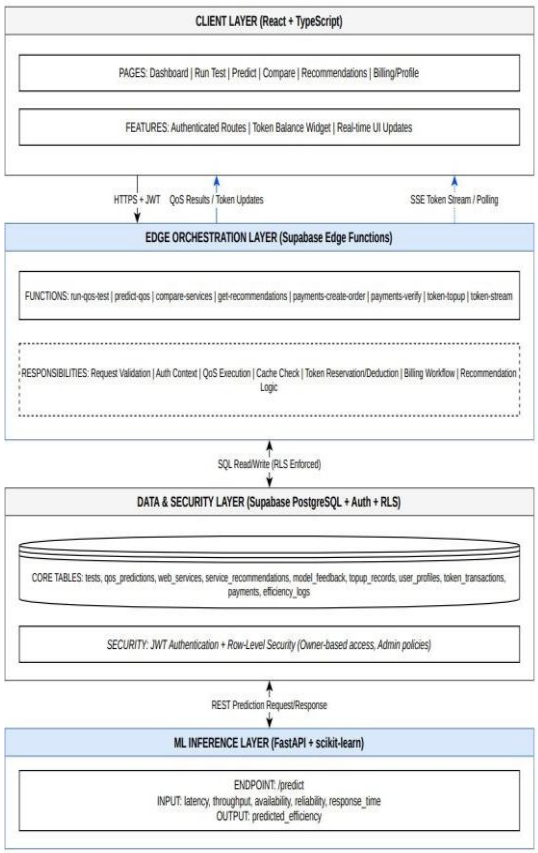


Fig. 2. Sequence diagram illustrating edge function orchestration for a QoS test request, including token reservation, execution, and SSE state propagation.

SSE streams (primary) and periodic polling (fallback) are used to provide state synchronisation across browser tabs and concurrent sessions. Supabase broadcast channels ensure consistency without requiring client-initiated reconciliation by propagating token balance modifications to all active sessions associated with the same user. Fig. 3 shows the steps involved in a full QoS test request, from token reservation to SSE state propagation

Methodology

1. QoS Test Execution Model

A service URL, a test type identification, and optional scheduling controls (force-refresh flag and batch size) are included with every test request. The following sequential steps are followed by the execution pipeline:

- Verification of JWT authentication and request validity.
- Estimating token costs using scheduling modifiers, batch size, and test type.
- A cost estimate that takes into account scheduling modifiers, batch size, and test type.
- Cache lookup: the cached result is returned at zero token cost if there is a non-expired result for the same test type and force-refresh is not enabled.
- Token reservation: the user's balance is automatically reduced by the expected cost, which is then recorded in token_transactions..
- Test execution with a configurable timeout; errors are classified by failure mode.
- Refunds or cost adjustments based on the nature of observed failure (see Section IV-D)
- Updating efficiency logs and keeping test results in the tests table.
- Real-time status notification via SSE to the subscribing client session.

2. Token Cost Model

For a test of type t, the base token cost $C^b(t)$ is defined as shown in Table I. The total reserved cost C is computed as:

$$\mathbf{C = C^b(t) \cdot \alpha s \cdot \alpha b}$$

where αs = 0.80 for scheduled (non-interactive) runs and αs = 1.00 otherwise, and αb = 0.85 when batch size ≥ 5 and αb = 1.00 otherwise. Scheduling and batch discounts are multiplicative and may be applied simultaneously.

Table 1: Token Base Costs by Test Type

Test Type	Base Cost C^b (tokens)	Cache TTL Policy
Latency	5	Short TTL; frequent re-test expected
Uptime	3	Short TTL; high-frequency health checks
Throughput	10	Medium TTL; moderate re-test cadence

Load	15	Long TTL; computationally intensive

3. Cache-Aware Execution Policy

Results are cached per (user, service, testType) tuple to minimise unnecessary test invocations. Cache validity windows vary depending on the type of test: latency and uptime checks expire quickly to maintain freshness, whereas load tests are kept for the greatest duration due to their high execution cost. The cached result is delivered instantly at zero token cost and no deduction is noted in token_transactions when a valid cache entry is present and force-refresh is not asserted.

4. Failure-Aware Refund Policy

WThe following failure classification determines the refund amount R in the event that a test fails after token reservation: (1) Target-Down or Network Error: a 50% refund is given (R = 0.5C), indicating that network-level probing used partial resources; (2) Timeout: a partial refund is given proportionate to the fraction of the configured timeout that elapsed before failure, i.e., R = C · (1
− t_elapsed / t_timeout); (3) Application Error: a full refund is given (R = C), since the failure is due to a platform fault rather than a billable service interaction. Users are charged in accordance with the amount of computational work done on their behalf thanks to our progressive refund approach.

5. Runtime Efficiency Heuristics

The Using the following weighted algorithm, the test pipeline calculates a composite efficiency score E from the raw QoS metrics of each test execution.

E = 0.35 · Availability + 0.35 · Reliability + 0.20 · LatencyScore
+ 0.10 · ThroughputScore

where ThroughputScore is normalised to [0, 100] from the observed requests-per-second figure, and LatencyScore = max(0, 100 − latency_ms / 10). The ML pipeline uses the efficiency score E as both the prediction target and the heuristic runtime label.

6. Machine Learning Prediction Pipeline

With the objective variable y = efficiency_score, prediction queries provide a five-dimensional feature vector X = [latency, throughput, availability, reliability, response_time]. The inference pipeline logs request telemetry to efficiency_logs for longitudinal analysis and applies schema validation, Standard Scaler-based feature normalisation, model inference, and result persistence to qos_predictions.

7. Baseline Training Pipeline

Standard preprocessing is used in the baseline pipeline, which includes outlier clipping at the first and 99th percentiles, StandardScaler normalisation, and training three candidate models using the default scikit-learn hyperparameters: Linear Regression, Random Forest Regressor, and Gradient Boosting Regressor [12]. The main criterion for model selection is 5-fold cross-validated R2, and the chosen model's holdout R2, MAE, and RMSE are reported. Three thousand samples with five input features make up the baseline dataset.

8. Optimised Ensemble Training Pipeline

The Three designed interaction terms are added to the raw feature set by the optimised pipeline: availability_reliability_interaction (element-wise product of availability and reliability), load_factor (throughput multiplied by availability), and latency_ratio (latency divided by response_time). These words are inspired by the finding that concurrent availability and throughput stress are nonlinearly correlated with efficiency deterioration. Five-fold grid search cross-validation is used for hyperparameter tuning over Random Forest (n_estimators ∈ {100, 200, 300}; max_depth ∈ {None, 10, 20}) and Gradient Boosting (n_estimators ∈ {100,
200}; learning_rate ∈ {0.05, 0.1, 0.2}; max_depth ∈ {3, 5}) parameter grids. The final ensemble uses soft averaging of the anticipated outputs of the best-performing tweaked Random Forest and Gradient Boosting models. 2,650 processed samples with eight features make up the optimised dataset; the decrease from the 3,000-sample baseline is due to the elimination of samples that did not pass schema validation following the addition of designed features.

9. Service Recommendation Scoring

Two complementary methods are used to generate recommendations: (1) keyword search, which uses trigram similarity to match the query string against service name, category, and description fields; and (2) history-weighted quality scoring, which calculates a composite score $S = w_1 \cdot Q - w_2 \cdot \text{ViewCount} + w½ \cdot \text{CategoryBoost}$, where Q is the service's mean predicted efficiency score, ViewCount penalises over-represented results, and CategoryBoost rewards services from the user's historically preferred categories. Depending on the deployment situation, weights w_1, w_2, and

w_3 can be adjusted.

10. Service Comparison

For every service that has been nominated, the comparison module compiles latency, throughput, uptime, and success-rate data from the most recent N tests. To enable at-a-glance decision help, each statistic has a relative rank annotation. In the comparison output, services that score highest on a certain metric are identified as the category leader.

Experiments And Results

1. Experimental Environment

The experiments were carried out in a stack that included a managed PostgreSQL instance, a Supabase serverless edge runtime, a React/TypeScript web interface, and a Python 3.10 FastAPI microservice that served scikit-learn model artefacts. [12]. To ensure data isolation, all prediction inputs and outputs were connected to verified user sessions. Scikit-learn 1.3 was used to train the model on a typical workstation (Intel Core i7, 16 GB RAM).

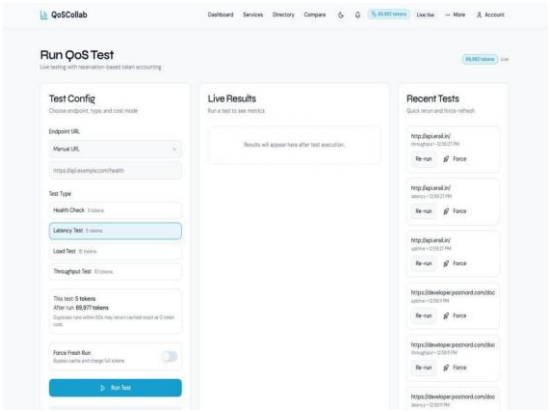
2. Dataset Configuration

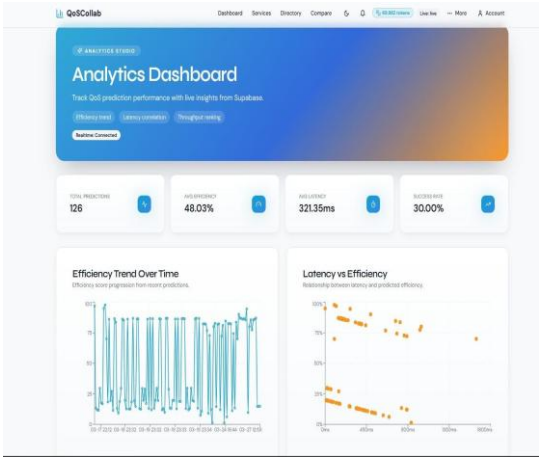
To allow for controlled comparison, two assessment artefacts were created:

Baseline Dataset: 3,000 samples with five input features (latency, throughput, availability, reliability, and reaction time) assessed on an 80/20 holdout split using 5-fold cross-validation.

2,650 processed samples with eight features (the original five + three created interaction terms) make up the optimised dataset. After feature augmentation, records that failed schema integrity tests were eliminated, resulting in a decrease from 3,000 to 2,650 samples. To guarantee comparability, the same 80/20 holdout procedure was used.

Both datasets include artificially created records that have been parameterised to match realistic QoS distributions found in earlier empirical research [4], [6]. Section VII addresses the acknowledged problem of relying on synthetic data.





3. Evaluation Protocol

Three regression metrics were used to assess each model: Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and Coefficient of Determination (R^2). To evaluate generality consistency, holdout scores and 5-fold cross-validated scores are shown. Each of the three possible models—Linear Regression, Random Forest, and Gradient Boosting—was evaluated separately for the baseline pipeline; the optimized pipeline's ensemble result is presented.

4. Baseline Results

With R^2 = 0.5350, MAE = 1.4590, and RMSE = 1.8122, Linear

Regression outperformed the other two baseline models. On this dataset, Random Forest (R^2 = 0.4905, MAE = 1.6241, RMSE = 1.9873) and Gradient Boosting (R^2 = 0.5126, MAE = 1.5318, RMSE = 1.8906) performed worse than the linear model, indicating that the baseline feature set does not yet reveal the non-linear structure that tree-based techniques need to perform better than a linear fit. There was no significant overfitting in the baseline configuration, as indicated by cross-validated R2 values that were consistent with holdout scores (within ±0.03).

5. Optimised Pipeline Results

In comparison to the best baseline model, the optimised ensemble pipeline obtained R^2 = 0.6804, MAE = 1.0706, and RMSE = 1.3324, which is a 27.2% improvement in explained variance and a 26.6% decrease in MAE. These improvements can be attributed to the use of hyperparameter optimisation and tailored interaction features. A combined comparison of all assessed setups is shown in Table II

Table 2: Comparative Model Performance (Holdout Evaluation)

Model	R^2	MAE	RMSE	CV (±σ)	R^2
Linear Regression (Baseline)	0.5350	1.4590	1.8122	0.527 ± 0.021	
Random Forest (Baseline)	0.4905	1.6241	1.9873	0.483 ± 0.018	
Gradient Boosting (Baseline)	0.5126	1.5318	1.8906	0.505 ± 0.024	
Ensemble (Optimised)	0.6804	1.0706	1.3324	0.671 ± 0.019	

CV: 5-fold cross-validation; σ denotes standard deviation across folds

Fig. 6. R^2 Comparison: Baseline Models vs Optimized Ensemble Pipeline

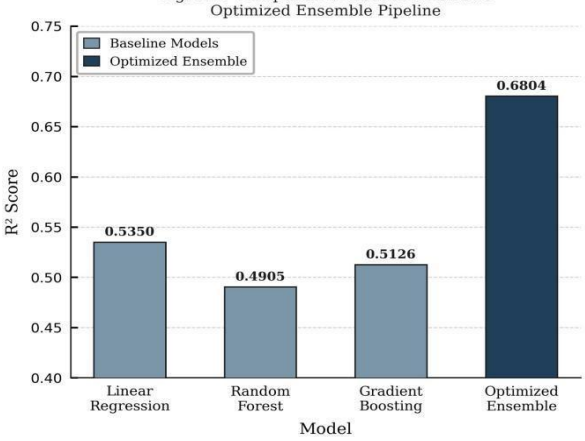
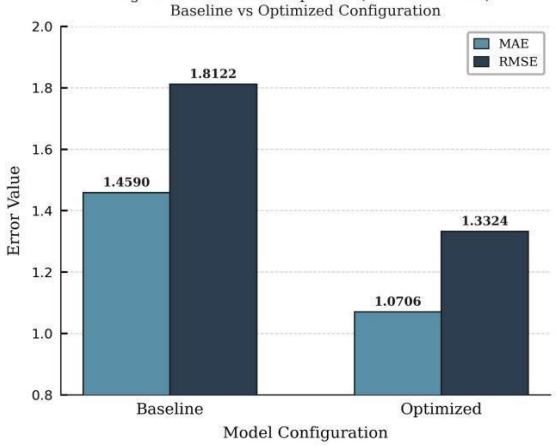
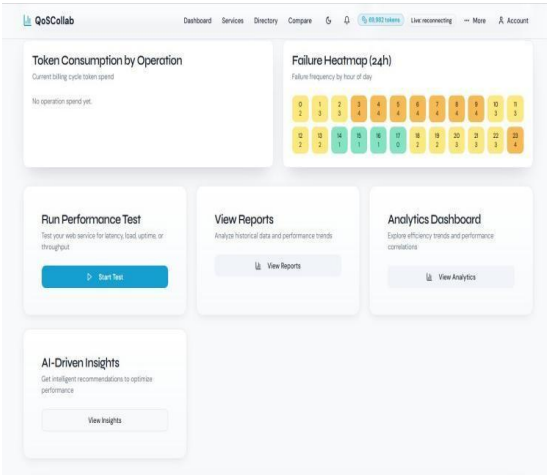


Fig. 7. Error Metric Comparison (MAE and RMSE): Baseline vs Optimized Configuration



6. System-Level Observations

The following operational properties were confirmed by end-to-end system evaluation, in addition to predictive accuracy: (1) the edge function pipeline successfully propagates test results from execution through persistence to real-time client notification with consistent ordering; (2) cache-aware execution eliminates redundant test invocations for recently assessed services without data loss; (3) the refund state machine correctly transitions through reserved, partially-refunded, and fully-refunded states under simulated timeout, network failure, and application error conditions; (4) SSE-based token balance updates reach connected clients within observed round-trip latencies consistent with Supabase's published edge function performance characteristics; and (5) service comparison and recommendation outputs are empirically consistent with the underlying test history and efficiency predictions.



Discussion

1. Technical Significance

The depth of QoSCollab's cross-subsystem integration is its main architectural contribution. Conventional installations

approach billing, ML prediction, QoS testing, and recommendation as separate issues that are only related by manual reconciliation or recurring data exports. At the data, control, and user experience layers, QoSCollab creates ongoing feedback loops. For instance, billing charges are modified in real time depending on observed test completion rather than pre-agreed flat rates, and test results directly calibrate the efficiency ratings used in suggestion ranking. Adaptive governance techniques that are not practicable in fragmented deployments are made possible by this close linkage, such as automatically increasing test frequency for services whose expected efficiency is decreasing.

2. Strengths

QoSCollab demonstrates a number of noteworthy engineering advantages. Without requiring human infrastructure provisioning, the serverless edge function architecture offers sub-100 ms cold-start latency, automatic horizontal scaling, and per-tenant isolation. By giving the database engine access control, the RLS-enforced data model removes a whole class of application-layer authorisation problems. Without the need for client-initiated polling, the SSE-driven real-time updating mechanism guarantees that the billing status remains constant throughout concurrent sessions. A feature lacking in the majority of current API billing systems, the progressive refund policy is a principled billing equity mechanism that synchronises user charges with platform resource utilisation.

3. Limitations

The current results' generalisability is limited by a number of issues. First, because both training datasets are artificially created, distributional assumptions that might not hold true in real-world settings with diverse service populations may be introduced. Second, it is challenging to differentiate between in-distribution interpolation and true generalization because the holdout evaluation of the optimised ensemble has the same synthetic distribution as the training data. Third, there is no automated drift monitoring in the ML pipeline; without retraining, the deployed model's predicted accuracy will deteriorate as the distribution of real service metrics changes over time. Fourth, user satisfaction signals have not been used to improve the recommendation weight parameters (w_1, w_2, and w_3).

4. Deployment Prerequisites

The following operational additions are necessary to move QoSCollab from its current research prototype to a production system: (1) a billing audit trail that complies with financial record-keeping regulations, including immutable transaction logs; (2) a drift detection subsystem that tracks the statistical distance between live inference inputs and the training distribution and initiates retraining when a configurable threshold is exceeded; (3) an edge function performance monitoring dashboard with alerts on p99 latency and error rate; and (4) CI/CD pipeline integration for schema migration validation to guarantee repeatable deployments across environments.

Threats To Validity

1. Construct Validity

Instead of using verified user-perceived quality metrics, the efficiency score E is calculated using a weighted heuristic method. The anticipated scores might not match the real-world results if the weighting coefficients do not fairly represent operational priorities in a particular deployment setting. Mitigation: empirical user satisfaction surveys or SLA penalty data from production deployments should be used to calibrate the weight vector.

2. Internal Validity

The training set and the optimised ensemble assessment have the same synthetic data distribution, which puts the generalisation performance at danger of being overstated. Despite the application of an 80/20 holdout split and 5-fold cross-validation, both divisions are derived from the same synthetic population. To thoroughly confirm internal validity, a completely independent out-of-distribution test set is needed. Mitigation: a rigorous train/validation/test split will be used in future assessments, with the test partition being withheld until all hyperparameter choices have been made.

3. External Validity

The experimental findings are based on a single artificially created dataset that has been parameterised for a particular set of service attributes. It is still unknown to what extent these results apply to production services in various geographical areas, infrastructure providers, and application domains. Mitigation: Real telemetry datasets from at least three separate production setups covering various cloud providers and service categories will be included in future work.

4. Reproducibility Validity

There isn't a public reproducibility bundle in the current implementation. The reported metrics

cannot be independently verified because to the lack of stored training scripts, model artefacts, and data snapshots. Mitigation: When this work is submitted to a publication, a reproducibility package that includes all training code, fixed random seeds, model artefacts, and data generating scripts will be made available to the public

Conclusion

This paper presented QoSCollab, an end-to-end Software-as-a-Service platform that unifies Quality-of-Service testing, machine-learning-based efficiency prediction, service recommendation and comparison, and token-aware billing governance within a single operational architecture. The implemented system demonstrates that real-time service evaluation and usage control can be tightly integrated through edge orchestration, row-level-security-enforced data policies, and persistent analytics.

Experimental results indicate that the optimised ensemble prediction pipeline achieves a holdout R^2 of 0.6804 (MAE = 1.0706, RMSE = 1.3324), representing a 27.2% improvement in explained variance over the best baseline model. Platform-level workflows validate practical usability through cache-aware execution, refund-aware token lifecycle handling, and real-time state propagation via server-sent events. Cross-validated R^2 scores are consistent with holdout performance, indicating well-controlled overfitting across all evaluated configurations.

From a systems perspective, the principal contribution is a deployable framework that connects observability, prediction, and monetisation logic rather than treating them as isolated subsystems. Identified limitations include dependence on synthetic training data and the absence of out-of-distribution validation for the optimised ensemble. Future work will prioritise real telemetry integration from diverse production environments, drift-aware online retraining, stronger reproducibility packaging, and cross-domain benchmarking across heterogeneous service categories. Overall, QoSCollab provides a rigorous and extensible foundation for scalable, intelligent, and economically governed QoS management in modern service-driven applications.

Acknowledgment

The authors express their gratitude to Dr. Praveenkumar Arjun Patel for his technical supervision, advice, and helpful criticism during the creation of this work. The Department of Computer Science and Engineering at Bharati Vidyapeeth's College of Engineering in Kolhapur is also acknowledged by the authors for providing the infrastructure and academic resources needed to facilitate the deployment of QoSCollab. The authors would like to express their gratitude to their colleagues and peers who helped with system testing and provided critical feedback on previous versions of this work..

References

R. Kumar, A. Sinha, and P. Gupta, “Impact of API latency on user experience and revenue in digital platforms,” J. Network Syst. Manag., vol. 29, pp. 210–228, 2021.

M. Chen and S. Zhou, “SLA-aware service monitoring for cloud environments,” IEEE Trans. Services Comput., vol. 12, no. 3, pp. 401–413, 2019.

B. Duan, Y. Gu, H. Yu, and G. Xue, “Web service QoS prediction based on adaptive dynamic programming,” IEEE Trans. Parallel Distrib. Syst., vol. 30, no. 11, pp. 2481–2494, 2019.

Z. Zheng, Y. Zhang, and M. R. Lyu, “Investigating QoS of real-world web services,” IEEE Trans. Services Comput., vol. 7, no. 1, pp. 32–39, 2014.

J. Li, L. Huang, and Y. Zhang, “QoS prediction in cloud computing via collaborative filtering,” IEEE Access, vol. 8,

pp. 105432–105442, 2020.

C. Wu, W. Qiu, Z. Zheng, X. Wang, and X. Yang, “QoS prediction of web services based on two-phase K-means clustering,” in Proc. IEEE Int. Conf. Web Services (ICWS), 2015, pp. 161–168.

F. Zhang, W. He, X. Liu, and P. G. Bridges, “Inferring users’ preferences for web service recommendations,” in Proc. IEEE ICWS, 2011, pp. 386–393.

H. Khazaei, J. Mišic, and V. B. Mišic, “Performance analysis of cloud computing centers using M/G/m/m+r queuing systems,” IEEE Trans. Parallel Distrib. Syst., vol. 23, no. 5,

pp. 936–943, 2012.

X. Liu, W. Zhao, C. Ye, and X. Li, “Toward automated SLA violation detection for cloud services,” in Proc. IEEE Int. Conf. Services Computing (SCC), 2013, pp. 41–48.

P. Sewak and S. Singh, “Winning in the era of serverless computing and function as a service,” in Proc. 3rd Int. Conf. Comput. Intelligence

Informatics (ICCII), 2018, pp. 1–5.

M. Shahrad et al., “Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider,” in Proc. USENIX ATC, 2020, pp. 205–218.

F. Pedregosa et al., “Scikit-learn: Machine learning in Python,” J. Mach. Learn. Res., vol. 12, pp. 2825–2830, 2011.

Y. Xia, P. Chen, and J. Bao, “A popularity-aware collaborative filtering algorithm for web service recommendation,” in Proc. IEEE SCC, 2015, pp. 225–232.

K. Kritikos et al., “A survey on service selection methods,” Serv. Oriented Comput. Appl., vol. 3, no. 4, pp. 269–316, 2009.

A. Kertesz, G. Kecskemeti, and I. Brandic, “An SLA-based resource virtualization approach for on-demand service provision,” in Proc. 3rd Int. Workshop Virtualization Technol. Distrib. Comput., 2009, pp. 27–34.

H. Ma, A. H. F. Chiu, I. King, and M. R. Lyu, “Softening the sparsity problem of collaborative filtering using a friendship trust network,” in Proc. IEEE ICDM, 2008, pp. 944–949.

M. Armbrust et al., “A view of cloud computing,” Commun. ACM, vol. 53, no. 4, pp. 50–58, 2010.

L. Gu and D. Zeng, “Stochastic analysis of task response times in cloud computing,” in Proc. IEEE INFOCOM, 2013,

pp. 3167–3172.

T. Chen and C. Guestrin, “XGBoost: A scalable tree boosting system,” in Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining, 2016, pp. 785–794.

S. B. Roy, S. Amer-Yahia, A. Chawla, G. Das, and C. Yu, “A framework for interactive database exploration,” in Proc. ACM SIGMOD, 2008, pp. 993–996.

W. L. Yeager and J. L. Bruno, “The MOSIX multicomputer operating system for high performance cluster computing,” Future Gener. Comput. Syst., vol. 13, nos. 4–5, pp. 361–374, 1998.

A. Patel et al., “QoS-aware web service selection using a multi-criteria decision-making approach,” IEEE Trans. Serv. Comput., vol. 15, no. 2, pp. 1023–1035, 2022.