Predicting FaaS Runtime with the Orama Framework Using Machine Learning

Leonardo Rebouças de Carvalho[1], **Geraldo Pereira Rocha Filho**[1,2], **Aleteia Araujo**[1]

[1]Computer Science Department - University of Brasilia
Campus Darcy Ribeiro – Brasilia – DF – Brazil

[2]Department of Exact and Technological Sciences, State University of Southwest Bahia
Vitória da Conquista – BA – Brazil.

leouesb@gmail.com, geraldo.rocha@uesb.edu.br, aleteia@unb.br

Abstract. *One of the significant challenges in Function-as-a-Service (FaaS) is the unpredictability of function runtimes, which can lead to cost overruns and performance degradation when deploying applications across multiple cloud providers. This paper presents a Machine Learning (ML)-based predictor integrated into the Orama Framework, which combines static code metrics (Halstead complexity measures) with empirical performance data to estimate function runtimes directly from the source code. Three neural network architectures (Dense, LSTM, and BLSTM) were evaluated, with the BLSTM achieving the highest accuracy (R^2 = 0.91) and a 20% lower MSE compared to the baselines. The predictor is available via API and through a graphical interface within the Orama Framework, supporting deployment planning, multi-cloud comparisons, and cost-performance optimization for serverless applications.*

1. Introduction

Serverless computing has emerged as one of the leading paradigms for developing and deploying microservice-based applications. In this model, Function-as-a-Service (FaaS) [Schleier-Smith et al. 2021] enables the execution of functions on demand with automatic scalability, eliminating the need for infrastructure management. However, the unpredictability of function runtime remains a major challenge. In multi-cloud environments, unexpected delays can compromise SLAs, degrade user experience, and lead to significant additional costs. Recent studies indicate that variations of only a few milliseconds per execution can result in substantial financial losses when accumulated over time.

The runtime of serverless functions is influenced by multiple factors, including code complexity, runtime environment, memory configuration, scaling policies, and provider-specific characteristics such as those of AWS Lambda, Google Cloud Functions, Azure Functions, and Alibaba Function Compute. Despite its importance, developers often lack tools that accurately estimate runtime before deployment or that allow informed comparisons among providers. Traditional benchmarking approaches face limitations, including low reproducibility, limited support for multiple providers, and difficulty capturing contextual variations, which can lead to inefficient resource allocation and suboptimal deployment decisions. Predicting runtime in serverless systems is challenging because many factors interact in unpredictable ways.This complexity makes it difficult to design models that capture consistent and reliable. patterns.

To address these limitations, this work extends the Orama Framework [de Carvalho et al. 2023, Carvalho et al. 2024] with a runtime predictor based on Machine Learning (ML) techniques. The solution leverages static code metrics, particularly Halstead complexity measures, combined with empirical benchmarking data previously collected by Orama, allowing runtime estimation directly from the function's source code. This integration enables comparative analysis among multiple providers without requiring prior execution, supporting decision-making in multi-cloud environments and reducing the risk of cost overruns. This solution is aimed at solution engineers, providing them with insights to select the most suitable cloud provider to support a given application.

In addition to the methodology for integration and dataset generation, we present the results of experiments with three neural network architectures (Dense, LSTM, and BLSTM), evaluated in terms of generalization ability and prediction accuracy. The BLSTM architecture stood out, achieving $R^2 = 0.91$ and up to 20% lower MSE compared to the other models. Finally, the predictor was incorporated into the Orama Framework through unified APIs and an interactive graphical interface, allowing users to submit FaaS functions, define load parameters, and obtain real-time runtime estimates by provider.

The remainder of this paper is organized as follows. Section 2 presents the main concepts behind Orama's prediction tool. Section 3 describes the methodology used for dataset generation and model training. Section 4 discusses the related work. Section 5 presents the experimental results, and Section 6 concludes the paper and outlines future work.

2. Background

Runtime evaluation in FaaS environments poses challenges requiring instrumentation, automated testing, metric collection, and statistical analysis across cloud providers, configurations, and usage patterns. Traditional benchmarking tools often fail due to limited multi-cloud support, insufficient metric granularity, and lack of reproducibility. To overcome these issues, specialized infrastructures such as the Orama Framework [de Carvalho et al. 2023] were developed to enable controlled experimentation and systematic knowledge extraction in serverless computing.

2.1. The Orama Framework

Orama [Carvalho et al. 2024][1] is a modular and extensible framework designed to facilitate benchmarking experiments in FaaS environments, enabling the evaluation of function performance across multiple cloud providers in a controlled, reproducible, and scalable manner.

The architecture of Orama comprises several modules responsible for various stages of the experimental process, including function provisioning, execution orchestration, metric collection, statistical analysis, and report generation. The tool supports primary FaaS services, including AWS Lambda, GCF, AZF, and AFC. It allows configuration of variables such as allocated memory, invocation frequency, concurrency level, source region, and function input parameters. The data collected through Orama served as the foundation for training the runtime prediction models presented in this work. Orama's

[1]`https://github.com/unb-faas/orama`

flexibility and extensibility also made it an ideal platform for integrating ML techniques, as discussed in the following subsection.

2.2. Machine Learning

ML [Mitchell 1997] is fundamental to modern computational systems, with applications spanning finance, healthcare, manufacturing, and cloud computing. Its strength lies in identifying patterns in large datasets and making predictions without explicit programming. In FaaS platforms, ML is increasingly relevant for optimizing performance, forecasting costs, and enabling automated deployment strategies across multiple clouds. Among ML approaches, supervised, unsupervised, and reinforcement learning stand out [Almuqati et al. 2024, Dai et al. 2024]. Supervised learning, especially regression, is beneficial for predicting runtimes, as seen in the Orama framework, which estimates function runtime based on source code features.

To build Orama's predictor, three neural network architectures specialized in regression on structured, high-dimensional data were evaluated: Dense (Fully Connected) [Jiang et al. 2022], LSTM (Long Short-Term Memory) [Tan 2021], and BLSTM (Bidirectional LSTM) [Radman and Suandi 2021]. Simpler models such as Decision Trees and Random Forest Regressors were evaluated in preliminary experiments; however, they did not yield satisfactory results, which motivated the adoption of models capable of capturing more complex relationships among the features. Dense networks handle non-sequential inputs and serve as a strong baseline. LSTM models capture long-term dependencies in sequential data. BLSTM extends this by processing sequences bidirectionally for a richer context. Although effective in regression tasks, these models rely on high-quality input data, necessitating rigorous preprocessing to ensure consistency and proper formatting. The following section covers these refinement techniques essential for learning meaningful patterns and achieving accurate predictions.

2.3. Data Preprocessing

Effective data preprocessing is crucial in ML, as it impacts model accuracy and generalization. Real-world data often contains missing values due to gaps in data collection or errors in data collection. These can be handled by deletion, imputation using the mean or median, or advanced methods such as K-nearest neighbors (KNN). Categorical variables must be converted into a numerical form using techniques such as one-hot encoding for nominal data or ordinal encoding when categories have an inherent order.

Normalization and outlier treatment are also key steps. Features on different scales can hinder learning, so methods such as min-max scaling [Bishop and Nasrabadi 2006] and z-score standardization [García et al. 2015] are employed. Outliers are detected by the interquartile range (IQR) method [Barbato et al. 2011], and can be removed or adjusted to prevent bias. Lastly, correlation analysis and dimensionality reduction improve efficiency. Highly correlated features may cause overfitting, so Pearson's correlation helps identify them [Franzese et al. 2018].

2.4. Model Evaluation

Evaluating a ML model is critical to ensure accurate and reliable performance in real-world scenarios. This process measures the model's ability to capture underlying data patterns and helps identify issues such as bias, variance, or data leakage. Using training, validation, and test datasets with appropriate performance metrics allows systematic model selection, tuning, and deployment based on empirical evidence.

Performance metrics are crucial for evaluating model effectiveness, particularly in regression tasks. Standard metrics include Mean Absolute Error (MAE) [Chai et al. 2014], which measures the average absolute difference between predictions and actual values; Mean Squared Error (MSE) [Akmal et al. 2024] and Root Mean Squared Error (RMSE) [Chai et al. 2014], which penalize larger errors; Mean Absolute Percentage Error (MAPE) [Akmal et al. 2024], which expresses errors as percentages; and the coefficient of determination (R^2) [Akmal et al. 2024], which measures explained variance. Each metric provides distinct insights, and together they form a comprehensive evaluation framework. Visualizations complement these metrics by revealing performance variability and alignment. Boxplots show distribution characteristics, while predicted-versus-observed plots help detect underfitting, overfitting, or data issues. Combined, these analyses guide further optimization through hyperparameter tuning, regularization, and architectural adjustments to enhance accuracy and generalization.

2.5. Model Optimization

Model optimization is a crucial phase in ML development, focused on enhancing predictive accuracy and generalization by fine-tuning various model components. This process primarily involves hyperparameter tuning, where configuration settings not learned during training, such as learning rate, network architecture, batch size, and regularization coefficients, are systematically selected through external search methods. While traditional techniques, such as grid and random search, exhaustively or randomly explore the hyperparameter space, they can be computationally prohibitive in high-dimensional scenarios. More sophisticated approaches, including Bayesian optimization and tools like Hyperopt [Kurniawan et al. 2024], leverage probabilistic models to efficiently identify hyperparameter configurations that minimize validation loss and improve generalization with fewer evaluations.

Beyond hyperparameter tuning, model optimization encompasses architectural modifications such as adjusting network depth and width, incorporating dropout to mitigate overfitting, and applying regularization methods like L1 and L2 penalties to control model complexity. These combined strategies aim to balance bias and variance, yielding models that generalize robustly across diverse datasets. Building upon prior stages of the ML pipeline, data preprocessing, model evaluation, and optimization, it is evident that the quality and relevance of input features are foundational to success. In software engineering contexts, exemplified by the Orama Framework, models rely on metrics extracted directly from source code, capturing structural and complexity attributes essential for predicting performance or fault proneness. The following section outlines techniques for extracting code metrics, which are crucial for enabling effective, context-aware ML within the Orama pipeline.

2.6. Source Code Metrics Generation

Source code metric extraction enables the transformation of software artifacts into structured, quantifiable data that can be used as input for ML models. Among various existing techniques, lexical-based metrics have gained significant traction due to their simplicity and language-agnostic nature. These metrics rely on analyzing the source code statically, without requiring execution or dynamic instrumentation, which makes them efficient and scalable. By capturing properties such as code length, vocabulary size, and syntactic richness, lexical metrics provide a meaningful abstraction of code complexity that can be leveraged for predictive tasks in software engineering.

One of the most widely recognized lexical metric systems is the Halstead [Halstead 1977, Khan and Nadeem 2023] complexity measures, introduced by Maurice Halstead. This technique involves counting distinct and total operators (n_1, N_1) and operands (n_2, N_2) in a given code fragment. From these basic counts, several derived metrics are calculated, including program vocabulary, length, volume, difficulty, and effort. For instance, the *volume* reflects the amount of cognitive information needed to implement the code and is calculated as shown in Equation (1):

$$V = (N_1 + N_2) \cdot \log_2(n_1 + n_2) \tag{1}$$

These derived measures encompass not only structural properties but also the potential cognitive effort required for understanding or modifying the code.

Compared to other techniques, such as cyclomatic complexity [McCabe 1976] or line-of-code counts [Barry et al. 1981], Halstead metrics offer a more nuanced and expressive view of the code's intrinsic complexity. While cyclomatic complexity focuses solely on control flow and branching structures, Halstead's approach captures a broader range of syntactic elements, making it more suitable for ML pipelines that benefit from diverse and fine-grained features. Additionally, their static nature enables fast computation and applicability across various languages and programming styles, making them an ideal choice for scalable and automated metric extraction within frameworks like Orama.

This background section has presented key concepts underlying the Orama Framework, including its reliance on ML techniques and the extraction of source code metrics to serve as informative features. Understanding the interplay between data preprocessing, model evaluation, optimization, and the effective generation of lexical code metrics, such as those of Halstead, establishes a solid foundation for building accurate and robust performance predictors. Within this context, the following section outlines the methodology employed to construct the performance predictor for FaaS environments within the Orama Framework, detailing the use cases and scenarios, data collection, model training, validation processes, and feature engineering that drive its predictive capabilities.

3. Methodology

The construction of the runtime predictor for FaaS functions within the Orama Framework began with the definition and normalization of a set of cross-platform use cases. Table 1 presents these use cases, along with the mapping of their equivalent services across each cloud provider evaluated (AWS, Google Cloud, Microsoft Azure, and Alibaba Cloud). This standardization aims to ensure comparability across heterogeneous platforms, reduce bias introduced by differences in naming conventions or native resources, and enable the automated reuse of experiments. Three types of use cases were defined: Calculator, API for Object Storage, and API for DBaaS, as shown in Table 1. All cloud providers had the corresponding use cases implemented in Orama, with the exception of Alibaba Cloud, which was unable to automate the API for DBaaS use case despite repeated efforts.

Table 1. Use Cases used to train the Orama's predictor.

Use Case	Description	AWS	Google Cloud	Azure	Alibaba Cloud
Calculator	A simple calculator that allows basic arithmetic operations and, in doing so, triggers the FaaS technology stack of the provider.	Lambda	GCF	AZF	AFC
API for Object Storage	An API with 3 endpoints to List, Insert, and Delete JSON objects. The data is stored in the provider's respective Object Storage service.	Lambda + S3	GCF + Cloud Storage	AZF + Blob Storage	AFC + Object Storage Service
API for DBaaS	An API with 3 endpoints to List, Insert, and Delete JSON objects. The data is stored in a Database as a Service (DBaaS).	Lambda + DynamoDB	GCF + Firestore	AZF + CosmosDB	not implemented

For each use case implementation on each provider, code complexity metrics were extracted based on the Halstead metrics family. This process was automated via an internal service named Halsteader, integrated into the Orama stack. The extracted metrics (e.g., length, vocabulary, difficulty, volume, and estimated effort) were synthesized and organized in Table 2. It is possible to verify that significant differences exist in the function's source code, as Azure functions, for example, exhibit structural variations compared to functions developed for other providers, which explains the discrepancies observed in the metric values.

Code metrics alone do not capture variations induced by infrastructure, cold starts, scaling policies, or provider-specific resource limits. To incorporate such factors, empirical results from previously conducted benchmark campaigns using the Orama Framework, published in prior studies, were reused. These results include runtimes observed under multiple concurrency levels, workload sizes, and regional configurations.

Table 2. Use Cases' Halstead Metrics Extracted from The Source Codes.

Provider	Use case	Length	Vocabulary	Difficulty	Volume	Effort	Bugs	Time	Distinct Ops	Total Ops	Distinct Opds	Total Opds
AFC	calculator	9	7.5	1.50	28.22	52.43	0.01	2.91	3.00	4.50	4.50	4.50
AZF	calculator	212	52	44.02	1208.49	53194.54	0.40	2955.25	23.00	101.00	29.00	111.00
GCF	calculator	5	5	1.00	11.61	11.61	0.00	0.64	2.00	2.00	3.00	3.00
Lambda	calculator	9	7.5	1.50	28.22	52.43	0.01	2.91	3.00	4.50	4.50	4.50
AFC	api4os	53	21	3.19	232.93	740.13	0.08	41.12	4.00	26.00	17.00	27.00
AZF	api4os	302	90	23.62	1960.54	46303.33	0.65	2572.41	22.00	156.00	68.00	146.00
GCF	api4os	26	16	2.10	104.95	221.64	0.03	12.31	4.00	13.00	12.33	13.00
Lambda	api4os	44	24	4.21	234.73	1758.85	0.08	97.71	6.67	21.83	17.33	22.33
AFC	api4dbaas	—	—	—	—	—	—	—	—	—	—	—
AZF	api4dbaas	96	34	9.05	518.82	7550.96	0.17	419.50	9.33	48.33	24.67	47.67
GCF	api4dbaas	21	14	2.50	79.95	199.89	0.03	11.10	5.00	12.00	9.00	9.00
Lambda	api4dbaas	29	14	2.81	131.44	661.34	0.04	36.74	4.33	14.83	10.50	14.17

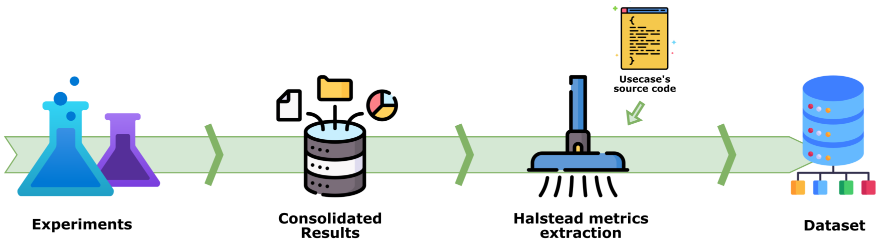

Figure 1. Dataset composition process.

Figure 1 illustrates the workflow used to compose the dataset for training the predictor. The process consists of four sequential macro phases: Experiments → Consolidated Results (for all three use cases) → Halstead Metric Extraction → Final Dataset with 332,191 entries. In the Experiments phase, controlled executions are orchestrated in a reproducible manner across the four cloud providers, using the Orama Framework. The Consolidated Results phase aggregates contextual metadata (region, payload size, number of invocations) and the elapsed time of each execution. Subsequently, the Halstead Metric Extraction step injects static code attributes generated by Halsteader. Finally, all attributes are integrated and harmonized into an analytical tabular dataset, aligned by use case and provider keys, and prepared for preprocessing and modeling steps.

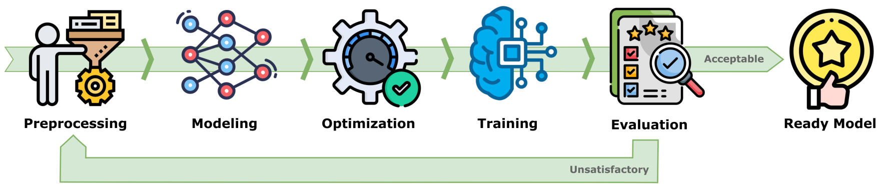

Figure 2. Training process.

The ML pipeline used to build the predictor is depicted in Figure 2. This pipeline comprises the stages: Preprocessing → Modeling → Optimization → Training → Evaluation → Decision. During Preprocessing, outlier removal, missing value imputation, categorical encoding of providers, and scaling of numerical attributes (including Halstead metrics) are performed. The Modeling stage evaluates various families of regression models (e.g., Dense, LSTM, and BLSTM) using all features in the dataset. The Optimization stage applies a multi-objective hyperparameter search, considering both mean squared error and cross-provider robustness. The Training stage produces candidate models (using 70% of dataset for training and 30% for validation), which are then assessed in the Decision stage. If the predefined performance criteria are met, the model is frozen and versioned; otherwise, a feedback loop returns to the Preprocessing stage for adjustments in feature engineering, data filtering, or sample balancing.

Figure 3 presents the updated architecture of the Orama Framework, which incorporates two new services: (i) Halsteader, responsible for static code analysis and generation of Halstead metrics; and (ii) Predictor, which consumes the trained model to estimate expected runtimes per provider and concurrency level. Although the backend provides unified APIs for automating experiments and querying predictions, the need for enhancements to the graphical interface has been identified. The proposed improvement involves

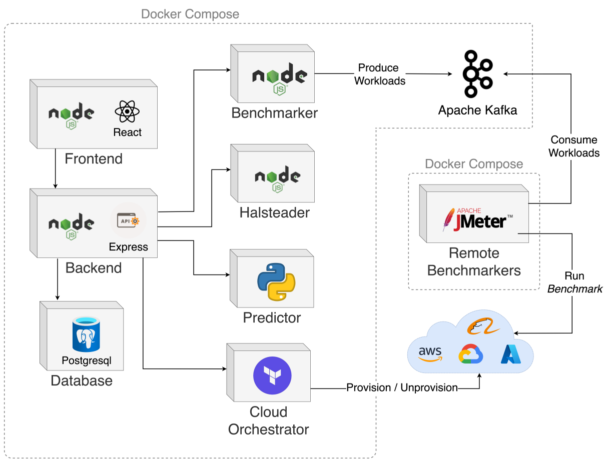

Figure 3. The new architecture of the Orama Framework, composed of the previously defined components [de Carvalho et al. 2023], with the addition of Halstead and Predictor components.

the development of an interactive interface that enables users to submit the source code of a FaaS function, define load parameters, and obtain comparative runtime estimates across AWS, Google Cloud, Azure, and Alibaba Cloud. This functionality is considered essential for enabling studies in portability, cost-performance analysis, and multi-cloud planning. To better situate our methodology within the existing body of research, the following section discusses related works on runtime prediction in FaaS, emphasizing how our approach builds upon or diverges from previous efforts.

4. Related Works

Several studies in the literature address runtime prediction in cloud computing, particularly in Serverless environments such as FaaS. In [Tomaras et al. 2023], the SLOPE framework uses neural networks to estimate the number of instances needed based on past executions. When no prior data is available, it applies a graph-edit-distance method to find similarities among applications. Although SLOPE reduced operational costs by up to 66.25%, its scope is limited due to reliance on container-based infrastructures, which not all providers adopt. Unlike SLOPE, the Orama framework uses client-side experimental data, making its predictions more reflective of real-world conditions.

FaaStest [Horovitz et al. 2019] optimizes cost and runtime in FaaS through a hybrid ML approach that learns service behavior and selects the best platform dynamically. It also reduces cold start latency, achieving over 50% cost savings and 90% improvement in response time. However, it lacks runtime prediction based on source code, a key feature offered by Orama for supporting deployment decisions.

TrIMS [Dakkak et al. 2019] enhances neural network inference in FaaS by reducing data transfer overhead using a distributed persistent memory system across GPU, CPU, and cloud. It achieves up to 24 times lower latency and 8 times higher throughput, but does not address source code–based runtime prediction, as Orama does.

FuncMem [Pandey and Kwon 2024], implemented in OpenWhisk, manages memory by prioritizing non-blocking asynchronous requests and dynamically rescheduling functions. It significantly reduces cold start latency (63.48%) and runtime (54.93%), and improves throughput. However, it is limited to OpenWhisk environments, whereas Orama supports major cloud providers, including AWS, GCP, Azure, and Alibaba.

The ML-FaaS framework [Filippini et al. 2025] utilizes ML to predict resource usage and system overload in FaaS through function profiling. It achieves 97% accuracy in overload prediction but does not include a predictor based on source code complexity, which is central to Orama's approach.

Existing approaches have demonstrated effective strategies for cost optimization, reducing cold starts, and predicting resources in FaaS. However, most solutions either rely on infrastructure-level metrics, are restricted to specific platforms, or do not incorporate application source code analysis. The Orama Framework addresses these gaps by leveraging client-side execution data and source code complexity to predict runtime across multiple cloud providers, offering a more generalizable and deployment-oriented solution for performance prediction in real-world serverless scenarios.

5. Results

This section presents the evaluation results of the runtime prediction models integrated into the Orama Framework. The performance of three neural network architectures (Dense, LSTM, and BLSTM) was compared using MSE and R^2. These metrics were calculated using cross-validation in the dataset's validation set (30% of the dataset). Since the data used for validation are contained within the dataset, use cases different from those in training may yield estimates with low accuracy. The inclusion of new use cases requires retraining the model, taking the new context into account, in order to improve the tool's accuracy.

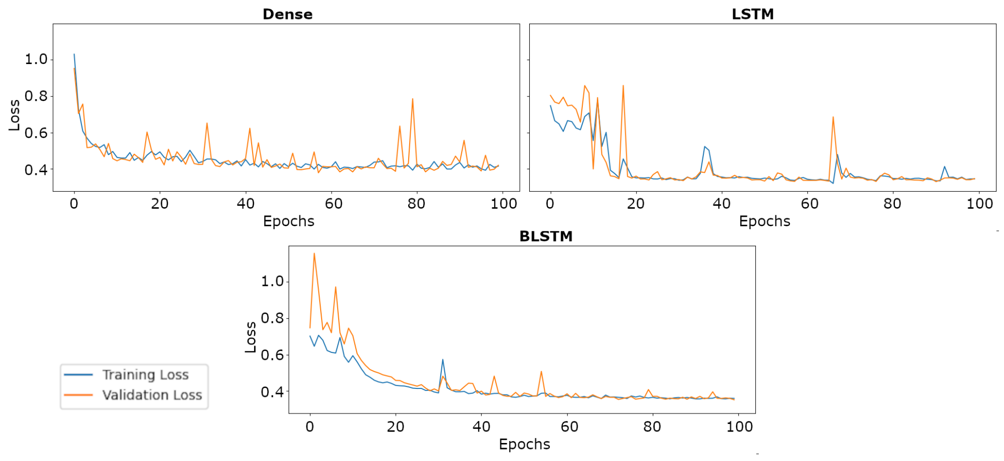

Figure 4. Loss in Dense, LSTM, and BLSTM models.

As illustrated in Figure 4, all three models showed convergence during training, with the Dense and BLSTM architectures demonstrating more stable and rapid reductions in loss. While LSTM models initially struggled with higher variance, their performance improved after several epochs due to their capacity to capture long-term dependencies in the extracted metrics.

Figures 5 and 6 present boxplots of the MSE and R^2 scores for each model. The BLSTM model achieved the lowest median MSE and the highest R^2, indicating superior

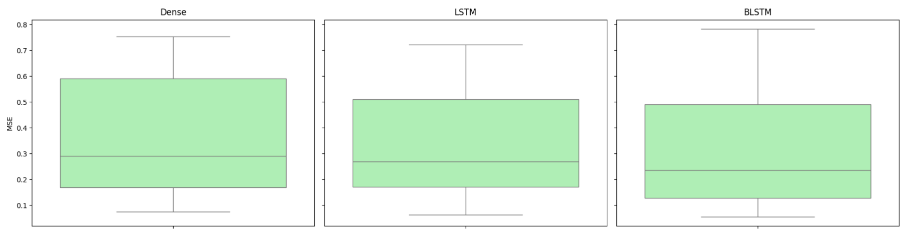

Figure 5. MSE Boxplot for Dense, LSTM, and BLSTM models.

accuracy and generalization across use cases and cloud platforms. The Dense model followed closely, whereas the LSTM model exhibited slightly higher prediction errors and greater variability, likely due to its sensitivity to input sequence length and complexity.

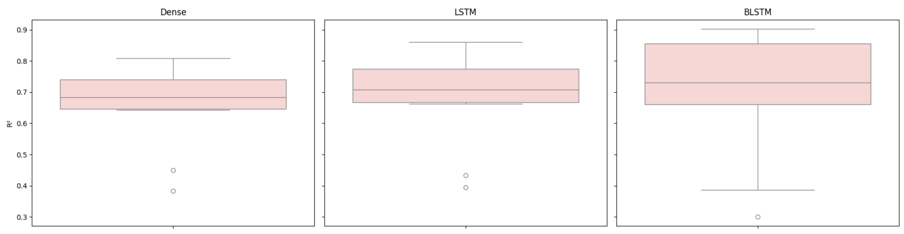

Figure 6. R^2 **Boxplot for Dense, LSTM, and BLSTM models.**

Figure 7 plots the observed versus predicted runtimes for the best-performing model (BLSTM). The data points closely align with the diagonal, confirming strong predictive agreement. Minor deviations are observable in edge cases with high complexity or uncommon combinations of Halstead metrics and provider behaviors; however, the predictor maintains robustness across the dataset overall.

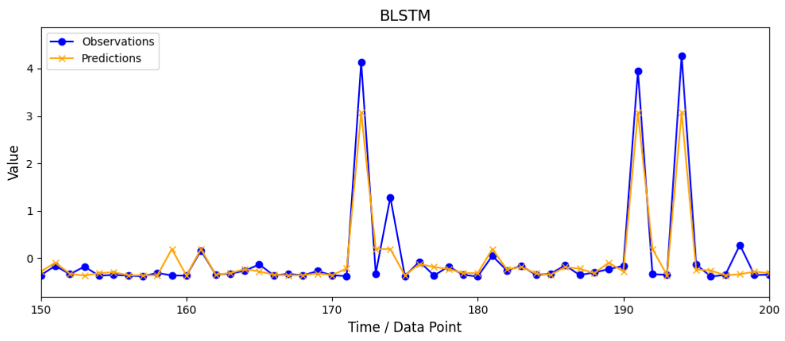

Figure 7. Observations vs. Predictions in BLSTM model.

The results confirm the effectiveness of using lexical code metrics to predict FaaS runtimes. Among the evaluated models, the BLSTM architecture stood out for its superior accuracy and generalization. Its ability to process code sequences bidirectionally allows it to capture richer contextual relationships, which are essential for modeling performance-related patterns in source code. Due to its consistent performance across scenarios, the BLSTM model was adopted as the foundation for the new Predictor component in the Orama Framework.

6. Conclusion and Future Works

This work presented a ML–based predictor integrated into the Orama Framework to estimate the runtime of serverless functions across major FaaS platforms. By combining Halstead code metrics with empirical benchmarking data, the solution enables accurate runtime predictions directly from source code, improving deployment planning in multi-cloud environments. Among the evaluated models, the BLSTM architecture achieved the best performance ($R^2 = 0.91$ and 20% lower MSE than baselines). The predictor, now available through unified APIs and a graphical interface, allows users to upload FaaS code and obtain real-time runtime estimates per provider.

Future work includes extending predictions to execution costs, supporting additional languages and providers, and evaluating advanced neural architectures such as BERT and LLaMA to enhance the semantic understanding of code and further improve accuracy. Expanding the dataset with new use cases and refining the user interface are also planned to broaden the applicability of the predictor.

References

Akmal, F., Nurcahya, A., Alexandra, A., Yulita, I. N., Kristanto, D., and Dharmawan, I. A. (2024). Application of machine learning for estimating the physical parameters of three-dimensional fractures. *Applied Sciences*, 14(24):12037.

Almuqati, M. T., Sidi, F., Mohd Rum, S. N., Zolkepli, M., and Ishak, I. (2024). Challenges in supervised and unsupervised learning: A comprehensive overview. *International Journal on Advanced Science, Engineering & Information Technology*, 14(4).

Barbato, G., Barini, E., Genta, G., and Levi, R. (2011). Features and performance of some outlier detection methods. *Journal of Applied Statistics*, 38(10):2133–2149.

Barry, B. et al. (1981). Software engineering economics. *New York*, 197(1981):40.

Bishop, C. M. and Nasrabadi, N. M. (2006). *Pattern recognition and machine learning*, volume 4. Springer.

Carvalho, L. R. d., Kamienski, B., and Araujo, A. (2024). Main FaaS Providers Behavior Under High Concurrency: An Evaluation with Orama Framework Distributed Architecture. *SN Computer Science*, 5(5):541.

Chai, T., Draxler, R. R., et al. (2014). Root mean square error (rmse) or mean absolute error (mae). *Geoscientific model development discussions*, 7(1):1525–1534.

Dai, X., Wei, T.-C., Yoo, S., and Chen, S. Y.-C. (2024). Quantum machine learning architecture search via deep reinforcement learning. In *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 1, pages 1525–1534. IEEE.

Dakkak, A., Li, C., Garcia de Gonzalo, S., Xiong, J., and Hwu, W.-m. (2019). Trims: Transparent and isolated model sharing for low latency deep learning inference in function-as-a-service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 372–382.

de Carvalho, L. R., Kamienski, B., and Araújo, A. P. (2023). Faas benchmarking over orama framework's distributed architecture. In *CLOSER*, pages 67–77.

Filippini, F., Cavenaghi, L., Calmi, N., Savi, M., and Ciavotta, M. (2025). Ml-based performance modeling in edge faas systems. In *European Conference on Service-Oriented and Cloud Computing*, pages 112–127. Springer.

Franzese, M., Iuliano, A., et al. (2018). Correlation analysis. In *Encyclopedia of bioinformatics and computational biology: ABC of bioinformatics*, volume 1, pages 706–721. Elsevier.

García, S., Luengo, J., Herrera, F., et al. (2015). *Data preprocessing in data mining*, volume 72. Springer.

Halstead, M. H. (1977). *Elements of Software Science (Operating and programming systems series).* Elsevier Science Inc.

Horovitz, S., Amos, R., Baruch, O., Cohen, T., Oyar, T., and Deri, A. (2019). Faastest - machine learning based cost and performance faas optimization. In Coppola, M., Carlini, E., D'Agostino, D., Altmann, J., and Bañares, J. Á., editors, *Economics of Grids, Clouds, Systems, and Services*, pages 171–186, Cham. Springer International Publishing.

Jiang, C., Jiang, C., Chen, D., and Hu, F. (2022). Densely connected neural networks for nonlinear regression. *Entropy*, 24(7):876.

Khan, B. and Nadeem, A. (2023). Evaluating the effectiveness of decomposed halstead metrics in software fault prediction. *PeerJ Computer Science*, 9:e1647.

Kurniawan, O., Alkhalifi, Y., Fitriana, L. A., Firdaus, M. R., Rais, A. N., and Hadi, S. W. (2024). Hyperparameter tuning optimization on machine learning models to predict software defects. In *2024 International Conference on Advanced Information Scientific Development (ICAISD)*, pages 35–40. IEEE.

McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.

Mitchell, T. M. (1997). Does machine learning really work? *AI magazine*, 18(3):11–11.

Pandey, M. and Kwon, Y.-W. (2024). Funcmem: reducing cold start latency in serverless computing through memory prediction and adaptive task execution. In *Proceedings of the 39th ACM/SIGAPP symposium on applied computing*, pages 131–138.

Radman, A. and Suandi, S. A. (2021). Bilstm regression model for face sketch synthesis using sequential patterns. *Neural Computing and Applications*, 33(19):12689–12702.

Schleier-Smith, J., Sreekanti, V., Khandelwal, A., Carreira, J., Yadwadkar, N. J., Popa, R. A., Gonzalez, J. E., Stoica, I., and Patterson, D. A. (2021). What serverless computing is and should become: The next phase of cloud computing. *ACM*, 64(5):76–84.

Tan, F. (2021). Regression analysis and prediction using lstm model and machine learning methods. In *Journal of Physics: Conference Series*, volume 1982, page 012013. IOP Publishing.

Tomaras, D., Tsenos, M., and Kalogeraki, V. (2023). Prediction-driven resource provisioning for serverless container runtimes. In *2023 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 1–6.