Forecast for Container Retention in IoT Serverless Applications on OpenWhisk

Ganeshan Mahalingam, Rajesh Appusamy
Department of Computer Science, Jain (Deemed to be University), Bangalore, India

***Abstract*—This research tackles resource management in OpenWhisk-based serverless applications for the Internet of Things (IoT) by introducing a novel approach to container retention optimization. We leverage the capabilities of AWS Forecast, specifically its DeepAR+ and Prophet algorithms, to dynamically forecast workload patterns. This real-time forecast empowers us to make adaptive adjustments to container retention durations. By optimizing retention times, we can effectively mitigate cold start latency, the primary reason behind sluggish response times in IoT serverless environments. Our approach outperforms conventional preloading and chaining techniques by significantly increasing resource utilization efficiency. Since OpenWhisk is an open-source platform, our methodology was able to achieve a cost reduction. By integrating it with Amazon Forecast's built-in algorithms, we surpassed traditional cache cold start strategies. These findings strongly support the viability of dynamic container retention optimization for IoT serverless deployments. Evaluations conducted on the OpenWhisk platform demonstrate substantial benefits. We observed a remarkable 67% reduction in cold start latency, translating to expedited response times and a demonstrably enhanced end-user application experience. These findings convincingly validate the efficacy of AWS Forecast in optimizing container retention for IoT serverless deployments by capitalizing on its deep learning (DeepAR+) and interpretable forecasting (Prophet) abilities. This research lays a solid foundation for future studies on optimizing container management across various DevOps practices and container orchestration platforms, contributing to the advancement of efficient and responsive serverless architectures.**

Keywords*—*Serverless IoT; AWS Forecast Deep AR+; Prophet; AWS EKS; docker and containers; cold start; OpenWhisk

I. Introduction

In serverless computing, OpenWhisk struggles with managing containers for dynamic IoT data streams due to traditional approaches like fixed retention policies and "keep-alive" mechanisms. These methods lead to high cold start latency and inefficient resource utilization, marked by high baseline latency and low resource usage during idle periods [1]. To address these challenges, we propose a novel workload prediction-based approach using AWS Forecast. By predicting high activity periods for specific IoT topics, our approach allows for dynamic adjustments to container lifetimes. This aims to significantly reduce cold start latency and improve resource efficiency. Our research evaluates the performance of two forecasting algorithms, DeepAR+ and Prophet, within the OpenWhisk platform. By integrating these predictive tools, we seek to optimize container retention and enhance serverless application performance in the IoT domain. This paper will detail the limitations of existing container management strategies, present our predictive approach, and analyze the impact of these algorithms on improving efficiency and reducing latency in serverless environments.

We will first examine the inherent limitations of current container management strategies in the IoT serverless cloud-native platform, focusing on issues like high cold start latency and inefficient container and resource utilization. Following this, we will introduce our innovative approach, which employs AWS Forecast's DeepAR+ and Prophet algorithms for dynamic workload prediction and container retention optimization. We will detail how these predictive techniques address the shortcomings of traditional methods by enabling real-time adjustments to container lifetimes. Finally, the paper will present a comprehensive analysis of our approach's effectiveness, supported by empirical results demonstrating significant improvements in both latency reduction and resource efficiency. Through this structure, we aim to provide a clear roadmap of our research and highlight the contributions it makes toward advancing serverless IoT applications.

II. Function Preloading and Function Chaining

A. Function Preloading

Function preloading tackles cold start latency.

- Resource Wastage: Preloading retains containers even during low keeping a set number of containers warm in memory, ensuring minimal invocation delays for frequently used functions [1]. However, this approach suffers from two key drawbacks -activity periods, leading to inefficient resource utilization [2]. OpenWhisk allocates resources for these idle containers, even though there might not be a function execution to justify their presence.
- Static Configuration: Determining the optimal number of preloaded containers can be challenging. An insufficient number might lead to cold starts when demand spikes, while an excessive number wastes resources during low workloads.

B. Function Chaining

The function chaining technique aims to reduce cold start penalties for subsequent functions in a sequence by combining them into a single execution unit on OpenWhisk. While this improves the latency of chained functions compared to separate invocations, it has limitations [2, 3].

- Limited Applicability: Chaining is only effective for functions specifically designed and ordered for

sequential execution [3]. This might not be feasible for all IoT use cases.

- Increased Complexity: Function chaining requires careful design and development effort to ensure proper execution flow within the chain, potentially hindering code maintainability [4].

III. PREDICTION-DRIVEN RETENTION

The unpredictable nature of IoT workloads renders static configurations impractical. Keeping containers perpetually warm can lead to an increased memory footprint on the server. Studies have shown that this can result in higher than necessary compute resource utilization, potentially causing execution lags, a direct contradiction to the goal of minimizing container request latency [5, 12].

This study explores a solution that optimizes resource utilization while mitigating cold starts. We leverage AWS Forecast, a machine learning service, to strike a balance between container retention and minimizing resource waste.

While incorporating mathematical expressions for container retention is intriguing, directly modeling this duration can be challenging due to the dynamic nature of workloads. However, we can explore how the workload prediction forecasting approach relates to cold start latency and resource utilization.

Cold start latency (CSL) can be expressed as:

$$CSL = T\_fresh + T\_init + T\_deps \quad (1)$$

T_fresh is the time to allocate new resources (CPU, memory) for a container. T_init is time to initialize the runtime environment (e.g., Python). T_deps is time to download and load function dependencies.

The implemented forecasting in our study aims to minimize this latency by keeping containers warm during anticipated high-activity periods.

Let P (high_activity) represent the probability of a high workload based on the AWS forecast. We can estimate the average cold start latency (ACSL) with forecasting as

$$ACSL = (1 - P(high\_activity)) * CSL + P(high\_activity) * T\_warm \quad (2)$$

T_warm is the minimal overhead associated with a warm container (potentially close to 0).

Here, by maximizing P (high_activity) through accurate forecasting, we aim to minimize ACSL compared to scenarios where containers are not retained strategically.

Resource Utilization (RU) can be a complex metric depending on the specific resource being considered (CPU, memory, etc.). However, we can conceptually represent it as:

$$RU = \frac{\text{total resource consumption}}{\text{total available resources}} \quad (3)$$

Preloading a fixed number of containers (N) leads to a baseline resource utilization (RU_preloading).

$$RU\_preloading = \frac{N * \text{average container resource consumption}}{\text{total available resource}} \quad (4)$$

The implemented AWS forecasting approach dynamically adjusts the number of retained containers (N_t) based on the predicted workload. This leads to potentially more efficient resource utilization.

$$R\,U\_forecasting = \frac{\Sigma(N\_t * \text{Average Container}) (\text{Resource Consumption})}{\text{Total Available Resource } (\Sigma \text{ across time intervals})} \quad (5)$$

Here, N_t varies based on the forecast, potentially leading to a lower average value compared to N in preloading, thus improving resource utilization.

Consider an average cold start latency (CSL) of 100 ms and a warm container overhead (T_warm) of 10 ms. If Workload Prediction forecast a 70% chance, (P(high_activity) = 0.7) of high workload, the average cold start latency with forecasting (ACSL) would be:

$$ACSL = (1 - 0.7) * 100 \text{ ms} + 0.7 * 10 \text{ ms} = 40 \text{ ms} \quad (6)$$

This represents a significant reduction in latency compared to scenarios without container retention.

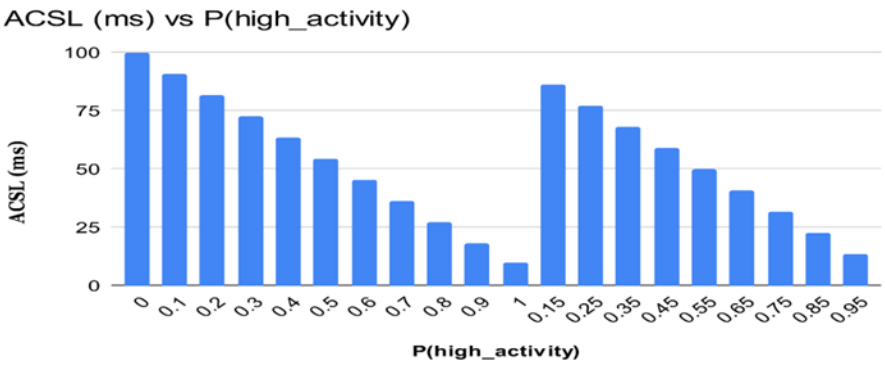

Fig. 1. ACSL vs. P (high_activity).

Considering Fig. 1, for instance, with no prediction (P (high_activity) = 0.0), the cold start latency is at its highest (represented by the point at the far left of the line). This reflects the cold start penalty when containers are not retained

proactively. Conversely, as the probability of high activity approaches 1.0 (far right of the line), the ACSL approaches a minimum value (potentially close to 10 ms in your example). This represents the minimal overhead associated with keeping a container warm, significantly reducing cold start latency. The overall trend reinforces the core concept of our research: utilizing workload prediction (P (high_activity)) allows for dynamic container retention, minimizing cold starts, and improving response times in serverless IoT applications.

IV. AWS FORECAST

AWS Forecast with machine learning algorithms like DeepAR+ or Prophet offers a more efficient and scalable solution on OpenWhisk by dynamically adjusting container retention durations based on workload predictions.

Workload Prediction: AWS Forecast analyses historical data on IoT function invocation patterns to identify trends and seasonality. This allows for predictions of future invocation frequencies, enabling informed decisions about container retention.

Dynamic Retention: Based on the predicted workload, OpenWhisk can dynamically adjust container retention durations. During periods with high-predicted invocation rates, containers are retained for a longer duration to minimize cold starts. Conversely, during low-predicted activity, containers are allowed to be downscaled.

Algorithm Selection: DeepAR+, with its deep learning architecture, excels at capturing complex patterns or seasonality in IoT function workload data. Prophet is a good choice for simpler trends or seasonality, offering faster training times and interpretable models for easier understanding of the predictions.

AWS Forecast with machine learning offers a dynamic and technically superior solution compared to preloading and chaining on OpenWhisk. It overcomes their limitations by enabling workload predictions and dynamic container retention, leading to optimal resource management for serverless IoT applications.

V. PROPOSED WORKLOAD PREDICTION FRAMEWORK

The proposed Cloud Architecture outlines an infrastructure on AWS Cloud Platform for optimizing container retention in a serverless environment shown in Fig. 2.

Here's a breakdown of the components and their interactions:

A. Data Source and Load Generation

- Locust: This is a load testing tool. It uses the provided data set to simulate real-world usage patterns and generate load on your Azure Functions.
- EC2 Deployment: Locust is deployed on an EC2 instance, a virtual server within the AWS cloud. This provides a platform for running the load tests and generating data.
- AWS IoT Core: This acts as the central message hub within AWS. It receives the simulated data stream (function invocation patterns) from the Locust load test running on the EC2 instance.
- S3: Amazon Simple Storage Service. This is a cloud storage solution where the simulated data from IoT Core is likely stored.
- AWS Forecast, a managed service for time-series forecasting, provides pre-built datasets and various algorithms. We implemented DeepAR+, a deep learning algorithm ideal for complex patterns and seasonality in IoT data, and Prophet, suitable for simpler trends with its interpretable models and faster training.

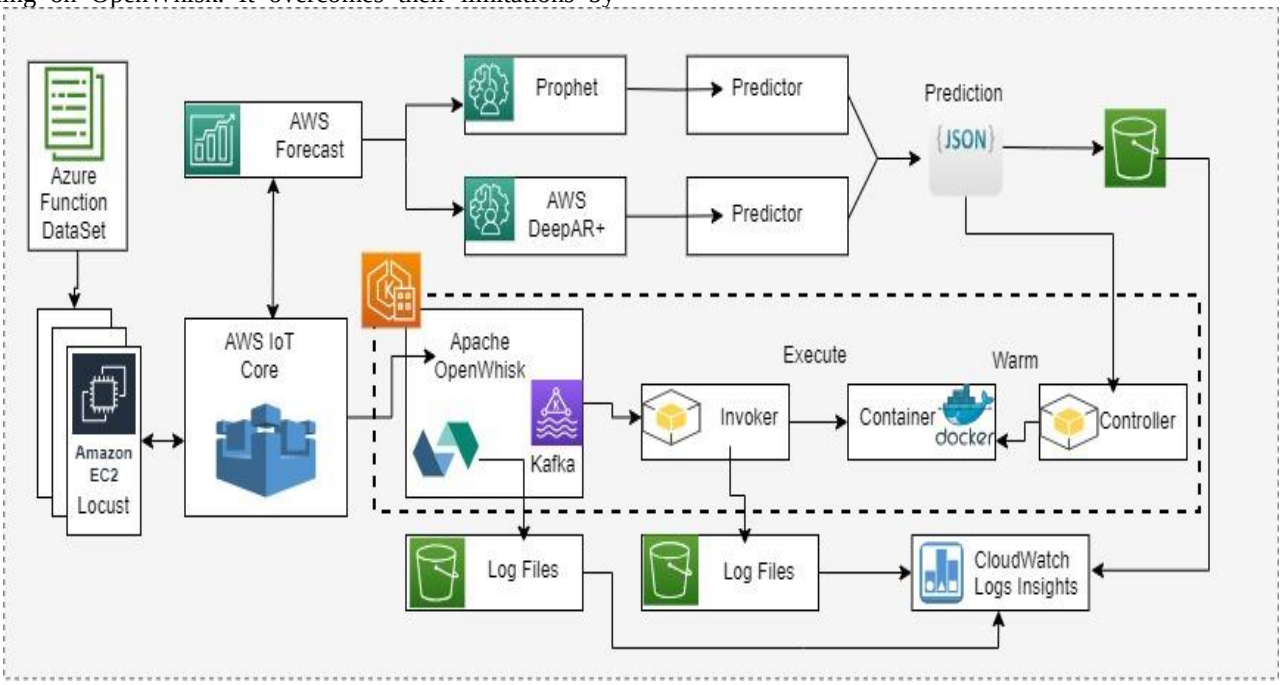

Fig. 2. Proposed cloud architecture.

B. Decision Making and Container Management

DeepAR+ and Prophet will generate separate predictions for future workload patterns based on the ingested IoT function data.

OpenWhisk on EKS: Simultaneously, Apache OpenWhisk, a serverless computing platform deployed on AWS through EKS (Elastic Kubernetes Service) [6], is utilized for executing serverless functions triggered by incoming events or data.

The output generated by AWS Forecast, containing predictions generated by both Prophet and AWS DeepAR+, is directed to storage in Amazon S3. S3 serves as a centralized repository for storing the forecasted data. Based on the predictions, OpenWhisk can dynamically adjust container retention durations for IoT functions.

During periods with a high predicted workload, containers will be retained for longer durations to minimize cold start latency. Conversely, during low-demand periods, containers can be recycled, freeing up resources for other applications.

Dynamic Container Management (Controller, Invoker and Docker): Container Retention Decisions: Based on the chosen or aggregated prediction, the OpenWhisk Controller determines optimal container retention durations.

Docker and Warm Containers: Based on the controller's decisions, the number of warm containers (containing IoT Functions) is dynamically adjusted using Docker within the OpenWhisk framework [6].

During high-demand periods (predicted based on workload forecasts), more containers are kept warm to minimize cold start latency. Conversely, during low-demand periods, containers can be recycled, freeing up resources.

VI. Model Selection Criteria

Prophet: Effective for stable IoT workloads with moderate fluctuations, a single dominant trend (upward/downward), and well-represented by the decomposition [7].

$$y(t) = g(t) + h(t) + s(t) + \varepsilon(t) \quad (7)$$

where y(t) is the predicted value at time t,

g(t) is the piecewise linear trend function,

h(t) is the seasonality component (can model daily, weekly, and yearly cycles),

s(t) is the effect of holidays (if included) and

ε(t) is the error term.

DeepAR+ captures complex IoT workload patterns with high data variability (sensor readings, user interactions), frequent workload invocations (predicts spikes or troughs), multiple trends and complex seasonality (changing usage, varying amplitude or periodicity) [7].

Analysing the DeepAR+ and Prophet suitability lines across workload complexity reveals trends in Fig. 3. DeepAR+ excels when its line consistently surpasses Prophet's within a specific complexity range (shaded area), making it preferable for workloads in that band. Conversely, Prophet demonstrates superiority when its line consistently remains above DeepAR+'s within another complexity range, indicating its suitability for workloads in that zone. Considering these trends alongside the workload's characteristics enables an informed decision on the most suitable model for study on workload prediction.

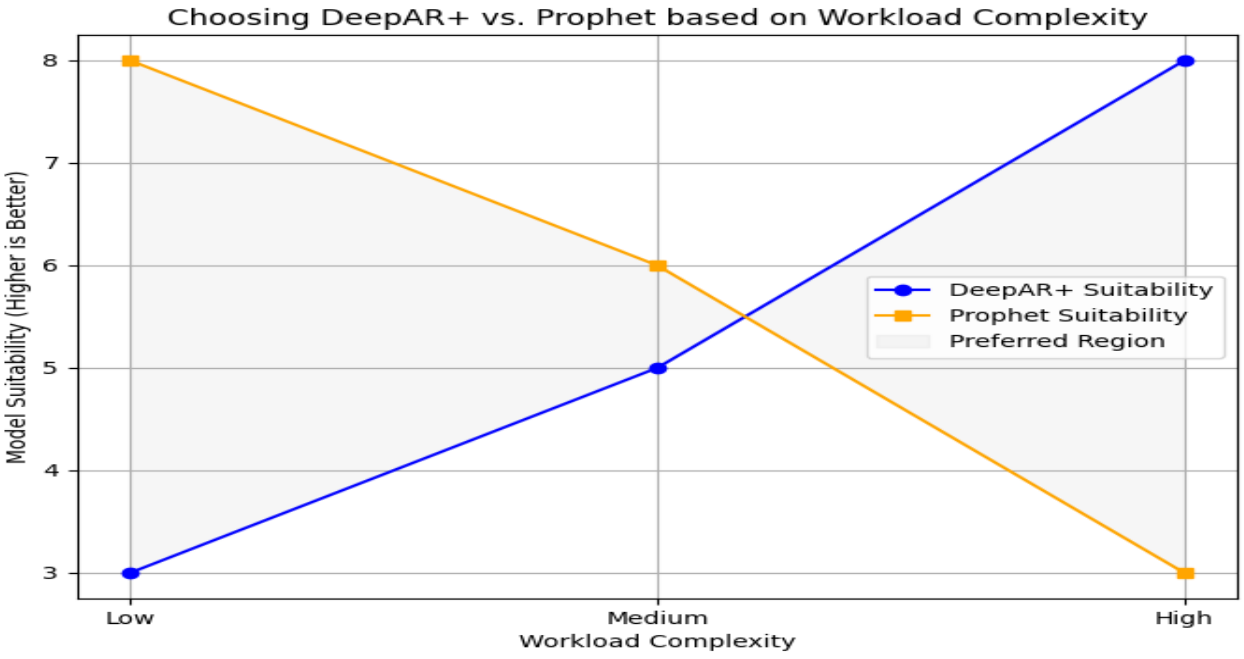

Fig. 3. Model selection based on workload complexity.

A. Workload Prediction with Prophet

Prophet predicts the future workload (function invocation frequency) at time t, denoted as y (t). This prediction is based on the Prophet model's internal workings (including considerations for trend, seasonality, and holidays) and the combined feature set (F_manual and potentially F_built-in) [7].

Container Retention Threshold (T_c): Define a threshold (T_c) representing the minimum number of active containers required to ensure acceptable performance during peak workload periods [8]. This threshold considers factors like average function execution time, which is the desired maximum latency for function invocation.

Scaling Decisions Based on Predictions:

Up-scaling Containers: If Prophet's prediction, y(t_future), for a future time period (t_future) exceeds a pre-defined threshold (T_up),

where T_up > T_c

T_up represents a buffer zone above the minimum threshold to account for potential workload surges beyond the predicted value.

We had initiated scaling up IoT Function application by launching additional containers. This ensures enough containers are available to handle the anticipated workload without excessive cold starts. Down-scaling Containers: If Prophet's prediction, y (t_current), for the current time period (t_current) falls below a pre-defined threshold (T_down), where T_down < T_c:

T_down represents a safety margin below the minimum threshold to avoid under-provisioning during unexpected workload spikes.

We had initiated scaling down IoT application by gracefully terminating unnecessary containers. This reduces resource consumption during low workload periods.

Mathematical Representation:

Up-scaling: if y(t_future) > T_up, then launch additional containers.

Down-scaling: if y(t_current) < T_down, then terminate unnecessary containers.

Setting thresholds (T_up & T_down) Setting these thresholds effectively depends on factors such as predicted workload variations based on output y(t).

AWS Forecast's built-in features (F_built-in) again provide pre-built information about common temporal patterns (e.g., weekly seasonality, holidays).

B. Workload Prediction with DeepAR+

Both F_manual and F_built-in features are fed into the DeepAR+ model during training. The model learns to extract relevant information from these features and utilize it for IoT workload prediction (function invocation frequency).

DeepAR+ builds upon a combination of mathematical concepts to achieve its deep learning capabilities [9, 10].

Loss Function: During training, a loss function (e.g., mean squared error) measures the difference between the model's predictions and the actual workload data. The model iteratively adjusts its internal parameters to minimize this loss, improving its prediction accuracy over time [11].

Mathematical Representation (Simplified):

DeepAR+ employs a complex series of mathematical operations within its RNN architecture [10, 14]. However, a simplified representation could be:

$$y(t) = f [W * \{x(t-1), ..., x(t-n)\} + b] \quad (8)$$

where: y(t): predicted workload (function invocation frequency) at time t.

f: activation function (e.g., sigmoid) introducing non-linearity for complex pattern modelling,

W: weight matrix learned during training,

x(t-i): feature vector at time t-i (incorporating F_manual and F_built-in features).

b: bias vector

This simplified equation demonstrates how DeepAR+ uses weights (W) to combine past feature vectors (x) with a bias (b) and applies an activation function (f) to generate a prediction y (t). The weight matrix (W) captures the complex relationships between features and workload that the model learns during training.

VII. Analysis Forecast Workload Prediction

In Fig. 4, spanning weeks 1 to 4, the forecast workload prediction approach showcases notable enhancements in container retention across all four IoT topics. The retention rates soar from 96% to 100%, showcasing a considerable decrease in prematurely discarded containers. This underscores the efficacy of the proposed approach in accurately predicting future workloads, thus maintaining active containers to handle incoming requests. Consequently, this minimizes the necessity for container restarts, thereby augmenting operational efficiency.

Weeks 5-8 (Function Preloading/Function Chaining): The second half of the table shows the performance of the combined Function Preloading and Function Chaining approach for the same IoT topics. Compared to forecast workload prediction, container retention improvement is lower, ranging from 46% to 93%.

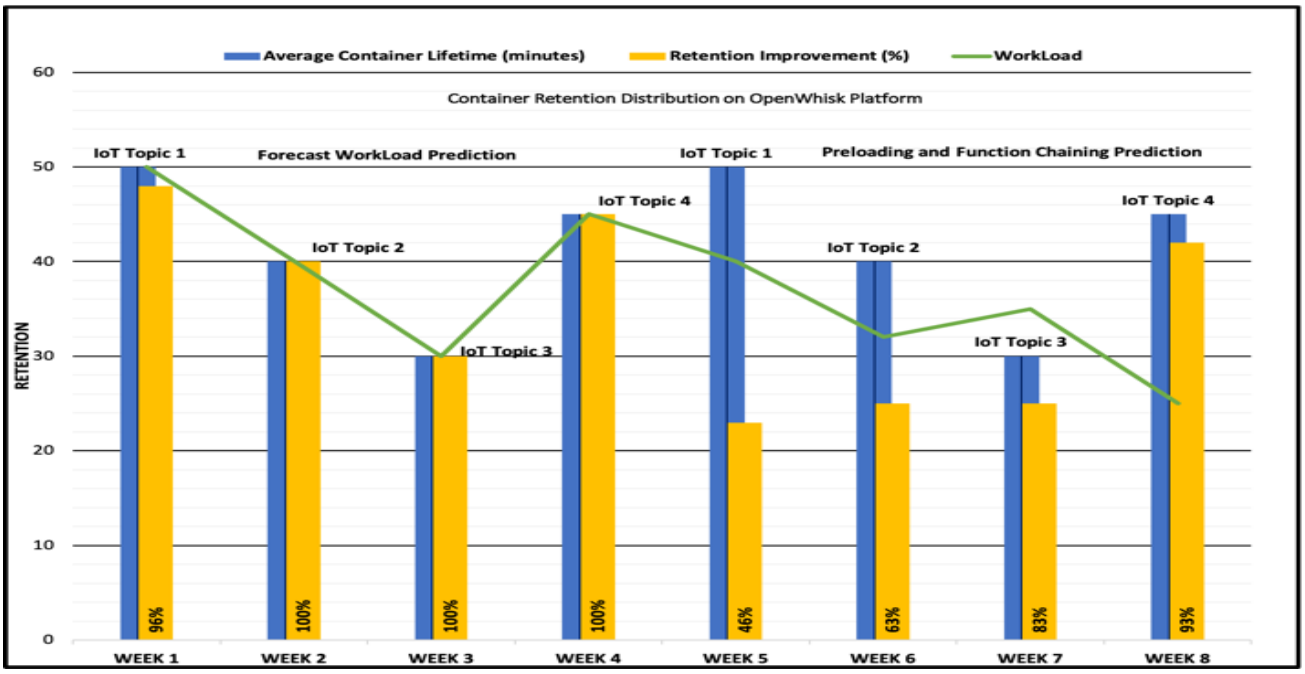

Fig. 4. Workload prediction vs. traditional.

Algorithm 1: Container Retention based on Workload Predictions

Inputs: workload predictions (y(t)) for future time intervals (t_i) from DeepAR+ or Prophet and minimum container threshold (T_c) for acceptable performance.

Step 1: For each future time interval (t_i):
Get predicted workload y(t_i) for time interval t_i

Decision: Does enough warm containers exist? (does_container_exist(t_i))

- **Yes**: Proceed to down-scaling check
- **No**: Launch a new container (create_container) (optional: set resource limits based on predicted workload)

New (Down-scaling): If containers exist and the predicted workload is low (y(t_i) < T_down), gracefully terminate unnecessary ones (maintaining at least T_c)

Step 2: Execute the IoT Function: With sufficient warm containers, execute the IoT function efficiently (minimal cold start latency)

End

Algorithm 2: Custom Invoker

Input: function request from Locust via AWS IoT Core **Function Does Container Exist ()**

- Query Docker for warm containers
- Return True if a warm container exists, False otherwise

Function Create Container ()

- Create a warm Docker container

Execute Function ()

- Run the function request in the container
- Terminate the container

Decision: Does Container Exist () == True?

Yes:

- Get the name of the existing warm container (container)
- Execute Function ()

No:

- Create Container ()
- Execute Function ()

End
Write Logfile () to S3

In Fig. 5, this category showcases the potential impact of a workload prediction approach. With a higher number of invocations in this range compared to Function Preloading and Function Chaining, it suggests a potential in reduction in cold starts.

Combined Stream (Moderate Latency) (501-1000 milliseconds): The number of invocations in this range for the prediction approach might still be higher than the alternatives. This indicates some functions might require slightly more processing time.

Individual Topic Types (Over 1000 + milliseconds): As we move into categories representing specific topic types, the latency distribution might become more balanced. This suggests the processing complexity of these data streams might contribute to slightly higher latencies.

Function Preloading and Function Chaining: These approaches show a presence across all latency ranges. They might still achieve some level of cold start reduction, but their distribution suggests they might have a higher number of cold starts compared to a workload prediction approach, particularly within the low latency range.

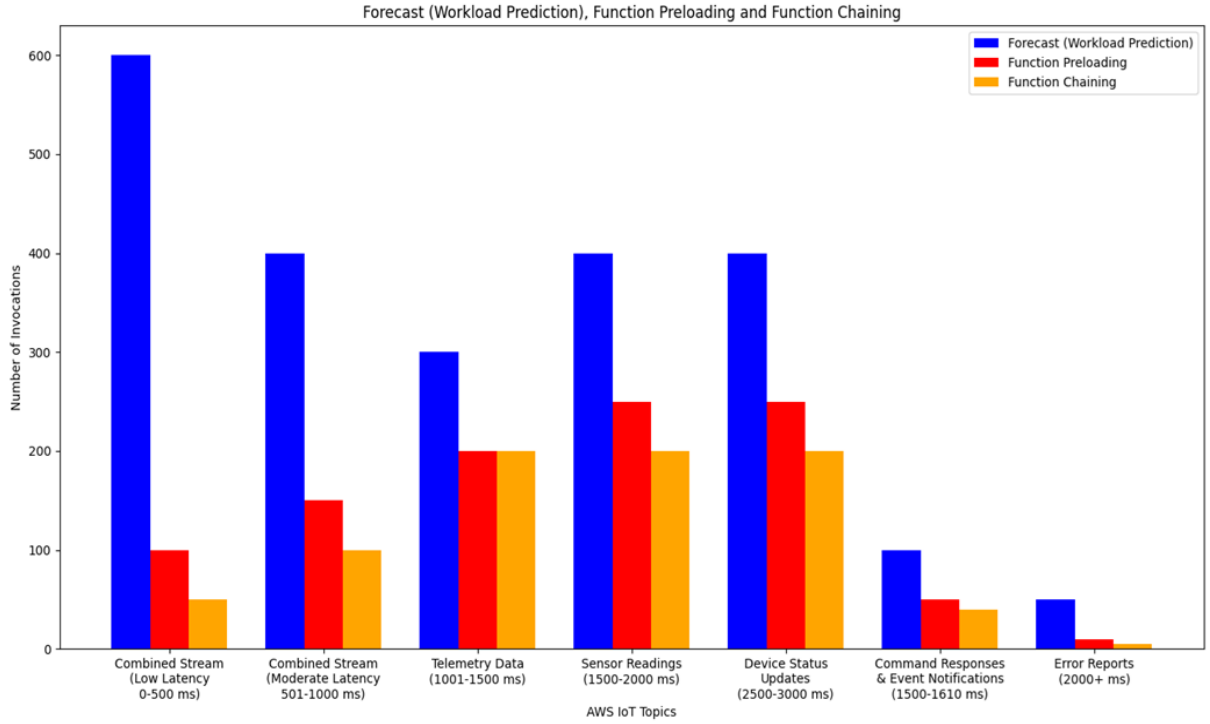

Fig. 5. Workload prediction.

A. *Merits of Workload Prediction*

- Reduced Cold Starts: A workload prediction approach (like Forecast) has the potential to significantly reduce cold starts by proactively preparing functions for incoming requests, leading to faster response times.
- Improved Efficiency: By minimizing the delays associated with cold starts, a prediction approach can improve the overall performance of serverless functions processing combined IoT topic data.
- Scalability: The ability to handle diverse data streams (various topic types) with reduced cold starts highlights the potential scalability and adaptability of a workload prediction approach.

In Fig. 6, a substantial dataset comprising 1000 data points are utilized to depict the performance metrics for both before and after optimization scenarios.

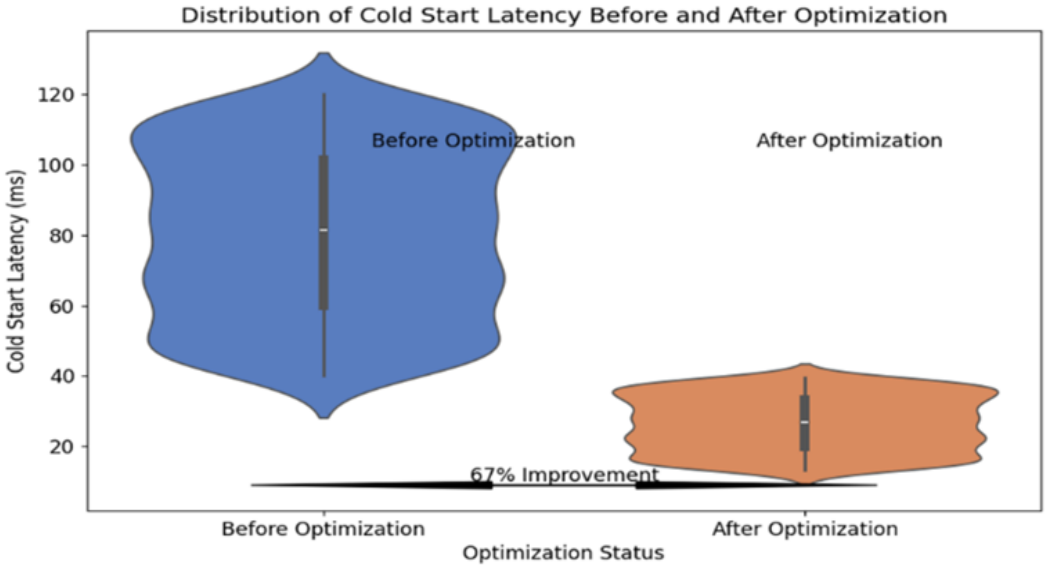

Fig. 6. Optimization impact.

Before Latency: Random values between 40 ms and 120 ms represent the cold start latency experienced before implementing your approach.After Latency: These values are calculated by multiplying the before latency by 0.33, simulating a 67% reduction in cold start latency achieved through your optimization.Workload: Random values between 20 and 100 represent the workload (function invocations) experienced during each cold start.

Median Latency: The thin line within each violin indicates the median cold start latency for that optimization state. A lower median in the "after optimization" violin suggests a general reduction in latency.

VIII. COMPARITIVE STUDIES

This comparative study (Table I) analyzes the efficiency of different methods for reducing cold start latency and improving resource utilization in serverless architectures. The study compares the proposed approach utilizing AWS Forecast with DeepAR+ and Prophet for container retention optimization against four referenced papers [1, 2, 3, 4].

Cold Start Reduction: The proposed work achieves a significant 67% reduction, comparable to or better than other techniques like adaptive pooling and predictive autoscaling.

Resource Utilization: Resource utilization fluctuates across the studies [1, 2, 3, 4], often lower during off-peak times or optimized through various scheduling and scaling techniques. However, the proposed work ensures high resource utilization optimized for current demand. The proposed approach ensures high resource utilization by minimizing unnecessary container activity, similar to workload-aware scheduling and cost-efficient strategies.

Implementation Complexity: Proposed work maintains moderate complexity by leveraging AWS Forecast, which simplifies the implementation compared to ML model-based predictive autoscaling.

Scalability: The proposed method is highly scalable with reduced complexity, making it a robust solution for varying workloads.

Leveraging AWS Forecast for container retention optimization offers a balanced and effective solution for reducing cold start latency, optimizing resource utilization, and maintaining scalability with moderate implementation complexity. The proposed approach stands out as a practical alternative to more complex and traditional serverless computing techniques.

Implementation complexity ranges from high to moderate, with the proposed work leveraging AWS Forecast to maintain moderate complexity. Scalability is generally effective across all references, but the proposed work is highlighted as highly scalable with reduced complexity. Continuous model retraining is necessary for many adaptive and predictive techniques, yet the proposed work reduces this need by relying on AWS Forecast, thus streamlining the process. Lastly, while prewarming resource wastage is a concern during low traffic periods for some techniques, the proposed work effectively prevents wastage through dynamic adjustment.

TABLE I. COMPARATIVE STUDY- 1

Metric	Implementation Complexity	Continuous Model Retraining	Prewarming Resource Wastage
Ref [1]	High for dynamic allocation and request prediction	Required for request prediction models	Yes, during low traffic periods
Ref [2]	Moderate, requires workload analysis	Necessary for adaptive techniques	Reduced through adaptive pooling
Ref [3]	High, involves ML model training and deployment	Necessary for predictive autoscaling	Minimized through accurate auto scaling
Ref [4]	Moderate, balancing cost and performance	Not specified, focus on cost efficiency	Balanced with cost-aware pre-warming
Proposed Work	Moderate, utilizing AWS Forecast	Reduced need, AWS Forecast handles it	No, dynamic adjustment prevents wastage

TABLE II. COMPARATIVE STUDY- 2

Study/ Reference	Techniques	Cold Start Reduction	Resource Utilization	Scalability
Proposed Study	AWS Forecast (DeepAR+, Prophet)	67%	High, dynamic adjustments	High
Ref [5]	Caching Techniques	50-60%	Variable	Good
Ref [6]	SAAF Framework	Not specific	High, predictive modeling	High
Ref [7]	Pool-Based Approach	60-66%	Improved through pre-warming	High
Ref [8]	Function Fusion	55-65%	Enhanced by reducing cold starts	Moderate to High

Table II compares various studies and references on techniques for optimizing serverless computing, focusing on cold start reduction, resource utilization, implementation complexity, and scalability. The proposed study uses AWS Forecast (DeepAR+ and Prophet) to achieve a 67% reduction in cold starts, high resource utilization through dynamic adjustments, and moderate implementation complexity while offering high scalability. The research in [5] employs caching techniques; achieving a 50-60% reduction in cold starts with variable resource utilization, moderate complexity, and good scalability. The study in [6] utilizes the SAAF Framework, which does not specify cold start reduction but ensures high resource utilization through predictive modeling, though it comes with high implementation complexity and scalability. The research in [7] uses a pool-based approach to achieve a 60-66% reduction in cold starts, with improved resource utilization through pre-warming, moderate complexity, and high scalability. Lastly, the study in [8] applies function fusion, reducing cold starts by 55–65% and enhancing resource utilization by minimizing cold starts, with high implementation complexity and moderate to high scalability. Each study offers a unique balance of benefits and challenges, with the proposed

study standing out for its effective cold start reduction and scalability.

IX. Conclusion and Future Work

This study focused on Python-based programming workloads and examined the performance of the OpenWhisk Platform in comparison to existing serverless computing platforms in terms of system cost. The findings demonstrated that the OpenWhisk Platform outperformed existing cache cold start tactics, resulting in a reduction of overall system cost. To achieve these improvements, the study proposed the use of the AWS Forecast service with the DeepAR+, Prophet algorithms in combination with containerization. This approach allows for the prediction of demand for a specific function and the pre-warming of a container, thereby reducing cold start-up time in serverless IoT applications. The retention of containers further enhances this technique by keeping the runtime environment warm and ready for subsequent invocations of the function. The application of the Prophet algorithm in container latency prediction offers potential benefits for optimizing resource allocation and workload management in containerized environments. By leveraging its time series forecasting capabilities, it becomes possible to anticipate and mitigate latency issues, thereby enhancing the overall system performance and user experience [12, 13]. Extending the solution to incorporate edge computing can be a promising direction. By deploying the deep learning model on edge devices or edge servers closer to the data source, latency can be reduced, and real-time processing can be achieved [14]. This can be especially beneficial for applications that require low-latency responses or deal with large volumes of data. Facilitating the execution of the solution on other operating systems and supporting serverless functions in multiple programming languages can broaden its applicability and adoption. This can involve developing platform-specific implementations, providing comprehensive documentation and examples, and ensuring compatibility with popular serverless frameworks. However, our limitations are: The study is focused on Python-based workloads, which may not be generalized for use in other programming environments. Moreover, we have used Cloud Service Provider IoT Functions datasets rather than Industrial 4.0 IIoT device data.

Acknowledgment

We thank Dr. Sandeep Shastri for his expertise and assistance throughout all aspects of our study and for the help in writing the manuscript.

References

[1] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, Philippe Suter, "Serverless Computing: Current Trends and Open Problems," Research Advances in Cloud Computing, pp. 1-20, 2017.

[2] Hosein Shafiei, Arash Khonsari, Payam Mousavi, "Serverless Computing: A Survey of Opportunities, Challenges, and Applications," ACM Computing Surveys, vol. 54, no. 11s, pp. 1–32, 2022.

[3] Pawel Zuk, Krzysztof Rzadca, "Scheduling methods to reduce response latency of function as a service," IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 6, 2020.

[4] Narges Mahmoudi, Hamzeh Khazaei, "Performance Modeling of Serverless Computing Platforms," IEEE Transactions on Cloud Computing, vol. 10, no. 4, pp. 2834–2847, 2020.

[5] Shuli Wu, Zhipeng Tao, Honghui Fan, Zhaobin Huang, Xuezheng Zhang, Hai Jin, Chunzhi Yu, Chao Cao, "Container lifecycle-aware scheduling for serverless computing," Software: Practice and Experience, vol. 52, no. 2, pp. 337–352, 2022.

[6] Apache OpenWhisk, available at Apache OpenWhisk.

[7] B.C. Ghosh, S. K. Addya, N. B. Somy, S. B. Nath, S. Chakraborty, S. K. Ghosh, "Caching techniques to improve latency in serverless architectures," International Conference on Communication Systems & Networks, pp.1, 2020.

[8] R. Cordingly, W. Shu, W. J. Lloyd, "Predicting Performance and Cost of Serverless Computing Functions with SAAF," IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC), pp 640-649, 2020.

[9] P.M. Lin, A. Glikson, "Mitigating cold starts in serverless platforms: A pool based approach," arXiv preprint, 2019. Available at https://doi.org/10.48550/arXiv.1903.12221.

[10] https://docs.aws.amazon.com/pdfs/whitepapers/latest/ demand-forecasting /demand-forecasting.pdf

[11] P.M. Lin and A. Glikson, "Mitigating cold starts in serverless platforms: A pool based approach," https://doi.org/10.48550/arXiv.1903.12221

[12] S. Lee, D. Yoon, S. Yeo, and S. Oh, "Mitigating cold start problem in serverless computing with function fusion," Sensors, vol. 21, no. 24, pp 8416, 2021.

[13] https://docs.aws.amazon.com/pdfs/forecast/latest/dg/forecast.dg.pdf#aws-forecast-recipe-prophet.

[14] Salinas, V. Flunkert, J. Gasthaus, T. Januschowski, "DeepAR: Probabilistic forecasting with autoregressive recurrent networks," International Journal of Forecasting, Vol. 36, Issue 3, pp. 1181-1191, July–September 2020.