National College of Ireland

Dynamic Prewarming Strategy using Reinforcement Learning and LSTM for Cold Start Mitigation in Serverless Computing

MSc Research Project
Cloud Computing

Sameer Nandkishor Kshirsagar
Student ID: 21219141

School of Computing
National College of Ireland

Supervisor: Dr. Ahmed Makki

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Sameer Nandkishor Kshirsagar
Student ID:	21219141
Programme:	Cloud Computing
Year:	2023
Module:	MSc Research Project
Supervisor:	Dr. Ahmed Makki
Submission Due Date:	14/12/2023
Project Title:	Dynamic Prewarming Strategy using Reinforcement Learning and LSTM for Cold Start Mitigation in Serverless Computing
Word Count:	7235
Page Count:	21

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Sameer Nandkishor Kshirsagar
Date:	14th December 2023

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Dynamic Prewarming Strategy using Reinforcement Learning and LSTM for Cold Start Mitigation in Serverless Computing

Sameer Nandkishor Kshirsagar
21219141

Abstract

Function-as-a-Service (FaaS) is emerging as a transformative cloud computing paradigm, offering several advantages, including pay-per-use billing, rapid application deployment, and elastic resource management. Cold start delay is a major challenge in FaaS applications, where the initial invocation of a function can take significantly longer than subsequent calls due to the need to initialize the underlying function container. These delays can significantly impact user experience and overall system performance. To mitigate coldstart delays, researchers have explored predictive prewarming techniques along with other solutions. Machine learning models like LSTM are used to predict upcoming function invocations patterns and prewarm containers accordingly. However, predictive prewarming can lead to excessive pre warming, consuming resources and increasing infrastructure costs.

To overcome this limitation, this research presents a Hybrid dynamic prewarming strategy that leverages reinforcement learning (RL) and LSTM to optimize container prewarming decisions, striking a balance between cold start latency reduction and cost minimization. The RL agent continuously learns from historical function invocation patterns and current system conditions, making real-time prewarming decisions based on predicted demand and resource availability.

Experimental evaluations demonstrate that the proposed strategy significantly improves traditional predictive prewarming approaches, achieving almost the same latency while minimizing resource costs by 30%. The dynamic nature of the RL-based approach ensures optimal prewarming decisions, effectively mitigating cold start delays while maintaining cost efficiency.

Keywords: Cold start delay, Serverless Computing, Q-learning, LSTM, Dynamic Prewarming, Reinforcement Learning.

1 Introduction

Considering the numerous perks that serverless platforms provide, this paper deep dives into the research using machine learning techniques to minimize the latency impact of cold starts on serverless platforms. Machine learning techniques focus on finding the hidden patterns in datasets, a beneficial asset for practical (real-world) applications where the function invocations are unpredictable. In the complex world of enterprise applications, minimum response time is a necessity. By reviewing the historical information

derived from old trends, this study deep dives into the potential of machine learning to warm serverless containers, leading to improved general performance. It's important to note that the intended solution currently caters to Python-driven programming projects. Making it compatible with different programming languages is the area of focus in the forthcoming section of this paper.

1.1 Research Question

Can a combination of reinforcement learning and LSTM algorithms be employed to develop a dynamic pre-warming approach for cold start mitigation in serverless computing environments?

By Design Container orchestration and their lifecycle scale up and down between 0 and N hence often faces challenges in managing the freezing and termination of containers. To prevent this, workarounds like the keep-alive method were introduced. Keep-alive involves pinging serverless functions in intervals with fake data packets that keep them active in a warm state without requiring a response. While such approaches are effective, they are not suitable for business applications with changing loads and demands. Additionally, keeping containers constantly active can lead to an increase in the server's memory footprint, potentially causing unnecessary compute resource usage. This, in turn, could lead to a slowdown in execution, creating a counterproductive problem, especially when the goal is to minimize request latency. This is an issue we need to rectify. This paper uses machine learning to investigate an optimal strategy to achieve a balance while minimizing the impact of cold starts.

Containerization makes serverless computing possible. Container orchestration is what's behind the scenes, driving the delivery of these services. Containers provide an isolated and self-sufficient environment for applications, totally independent on the type of operating system it is being run upon. This facilitates smooth deployment and gets rid of the old inflexible strategies. Orchestrating containers is nothing but automating the task of managing and provisioning them. However, this introduces the previously mentioned 'cold starts' problem. Cold start is a common issue in serverless computing and researchers have been making efforts in addressing and handling this issue. The top three platforms for serverless computing in the industry are AWS Lambda, Microsoft Azure Functions, and Google Cloud Functions.

Moreover, financial management is crucial for any business. While users will be appealed by the concept of no administration it may not be appealing for the ones responsible for managing budgets. Servelerless, like other cloud services, operates on a pay-as-you-go model. This is why, there are benefits in creating efficient and high-performing code, yet finding the correct balance between efficiency and performance can be challenging. To explain this point further, consider the keep-alive workaround for minimizing latency, keeping containers in a warm state will lead to excess resource consumption and increased cost.

1.2 Research Objectives

- Set up a serverless test infrastructure in an EC2 instance or any other cloud platform.
- Generate serverless function invocation predictions using machine learning models.

- Simulate function invocation calls in the test infrastructure using Function trace historic data.
- Analyze the function execution performance in the test infrastructure based on different metrics.

2 Related Work

The related research in this study emphasises the major key components that support serverless architecture. At the core, Function as a Service (Faas) solutions require an execution environment to host and process the business logic. Containers have emerged as the prime technology to rapidly establish such environments. Nevertheless, functions require the download of runtime libraries and dependencies before they can be operated. By breaking down these aspects, it becomes quite evident that latency may start affecting systems. Cold start latency is a huge issue in the serverless domain, with many researchers proposing solutions to address this issue. This research report is going to deep dive and thoroughly analyse such research papers.

2.1 Serverless computing and applications

In cloud computing, a serverless deployment model eliminates the burden of managing and provisioning resources from the developers which facilitates them to focus only on the application development process Baldini et al. (2017) Van Eyk et al. (2017). Serverless, refers to Function-as-a-Service (FaaS). Serverless doesn't mean there are no servers involved Instead, it focuses on the fact that the complicated job of managing resources is handed over to cloud service providers (CSP). FaaS brings in a new way of computing that reacts to events, adopts pay-as-you-go pricing, and bills according to the actual time the service is consumed. Functions are designed to scale on demand, they are stateless ,have short lifespans and operate on lightweight containers or virtual machines (VMs) when triggered by an event Vahidinia et al. (2020). Conceptually the FaaS model is created to start a new function instance whenever there's a demand request and close it down after the service. Yet, in practice commercial Faas offerings such as Google Cloud, Azure Functions and AWS Lambda Function may opt to reuse a function or keep it running for a limited period to serve more requests. Agarwal et al. (2023) Some open-source serverless frameworks, such as Kubeless and Knative, implement similar strategies that includes reusing an instance of a function instance to serve additional requests.

Using the advantages of serverless computing at the edge layer, machine learning applications are built using a serverless edge computing architecture.Bac et al. (2022) Lower latency and faster response times are the goals of this strategy. With a focus in particular on image classification using the MNIST dataset, the method makes use of federated learning in the serverless edge computing architecture to facilitate remote machine learning. The evaluation's findings show that the serverless approach can facilitate distributed machine learning, lower machine learning systems' end-to-end latency, and guarantee the best possible response and duration of operation.

In serverless computing, Lee et al. (2021) function fusion is used to reduce cold start time while taking parallel execution into account. The approach involves establishing a function fusion technique that manages branch and parallel execution fusing, as well as proposing a model for workflow reaction time in sequential and parallel runs, warm start

modes, and warm start modes. The study's findings show that the suggested approach can improve the original workflow's response time by 28–86% in five different workflows Lee et al. (2021), suggesting that it may be useful in reducing the cold start issue in serverless computing. In order to overcome a major issue in serverless computing, the paper suggests a novel solution to the cold start problem. This solution has applications for enhancing the effectiveness of serverless processes. furthermore offers a thorough assessment of the suggested approach in a real cloud environment, demonstrating its applicability and efficiency in resolving the cold start latency problem.

2.2 Loading necessary Libraries when a function is invoked

The researcher investigated a method to minimize the cold start latency by loading only essential libraries on function is invocation. This approach is in line with the broader research focused on addressing and resolving cold starts in serverless environments.

Numerous studies have explored techniques to mitigate cold starts, they mainly focus on two major approaches discussed below:

2.2.1 Optimizing Environments

This approach focuses on directly minimizing the cold start latency by optimizing the execution environment for serverless functions. General techniques include:

1. Pre-warming: Actively provisioning and initializing functions in a warm state to reduce cold start overhead. This can be achieved through invoking function from time to time or utilising pinging mechanisms.

2. Containerization: Usage of containerization technologies, such as Docker, to encapsulate serverless functions and streamline the loading of dependencies and runtime environments.

3. Function Sharing: Effectively sharing the runtime environment among multiple instances of the same function to minimize cold start overhead, especially for frequently invoked functions.

2.2.2 Minimizing Invocation Frequency

This approach focuses on decreasing the occurrence of cold starts by reducing the initial invocations of functions. Methods include :

1. Caching: Caching is done by storing the outcomes of frequently invoked functions in a cache to prevent repetitive cold starts and improve response times.

2. Prediction Models: Using predictive models to predict future patterns of function invocation, allowing proactive provisioning and minimizing cold starts.

3. Burstable Instances: It is a kind of compute resource which is capable to temporarily scale up without incurring cold start penalties during periods of increased demand.

In conclusion the Oakes et al. (2017) approach discussed in this paper takes an important step ahead in resolving cold starts in serverless computing. It entirely focuses on loading required libraries and proven effectiveness, contributes meaningful insights for further research and development in this domain.

2.3 ZIP-based Deployment vs. Container-based Deployment

In this section we are going to discuss the comparison between ZIP- based deployment which is the default deployment method for AWS Lambda and Contaniner based deployment. Dantas et al. (2022) In the ZIP-based deployment, it packages the dependencies and the function's code into a ZIP file whereas the container based deployment uses container images. ZIP-based deployment offers simplicity and efficiency but it leads to longer cold start delays because of the unzipping process. While Container-based deployment, on the other hand, can notably minimize cold start delays by directly loading the container image into the Lambda execution environment.

The author highlights the effectiveness of pre-warming Kumari et al. (2022), the approach of actively invoking functions to keep their execution environment in a warm state, in reducing cold start delay. Pre-warming is an excellent technique for applications which has regular function invocation as it can minimize cold start delays by up to 90%.

2.4 Machine Learning based predictions

To reduce cold start latency and improve resource usage, a Cold Start Optimization (CSO) method was applied on a Kubernetes-based serverless computing architecture. Xu et al. (2019) In order to detect cold beginnings and dynamically change container pool capacity, an Adaptive Warm-Up (AWU) approach has been created simultaneously with the deployment of a multi-layer perception network with two hidden layers. The finding of the study demonstrated that the CSO method outperformed traditional strategies like Moving Average (MA) and Auto-Regressive Moving Average (ARMA) models in terms of cold start latency and resource waste. This unique method contributed to a thorough assessment of the suggested technique by addressing cold start delay in serverless computing and using actual application data for validation. Predicting serverless function execution has also been done using similar LSTM deep learning prediction models, with at least 1.8 $\times$ improvements over baseline Apache OpenWhisk installations Govindan (2020).

2.5 Reinforcement Learning

Reinforcement Learning (RL) is a category of AI algorithms that is made up of an agents, environment, and rewards . The agent's responsibilities invovlve taking actions that changes both its own state and the state of the environment with the aim of achieving a goal. Depending on the effect of each action on achieving the goal, the agent either gets rewarded or a penalized. By analyzing the accumulated feedback from a set of actions, the agent develops policies to maximize rewards.

RL learning algorithms have the ability to learn best policies for decision -making in complex environments, adjusting to changes without depending on explicit rules or heuristics. Various studies have dived into the utilisation of RL to autoscaling in serverless platforms.

For instance, Zafeiropoulos et al. (2022) introduced an RL-based autoscaling mechanism that uses a deep Q-learning (DQN) algorithm to dynamically allocate and deallocate serverless functions. Compared to conventional mechanisms their approach accomplished notable reductions in resource provisioning latency and cold start instances compared to traditional mechanisms.

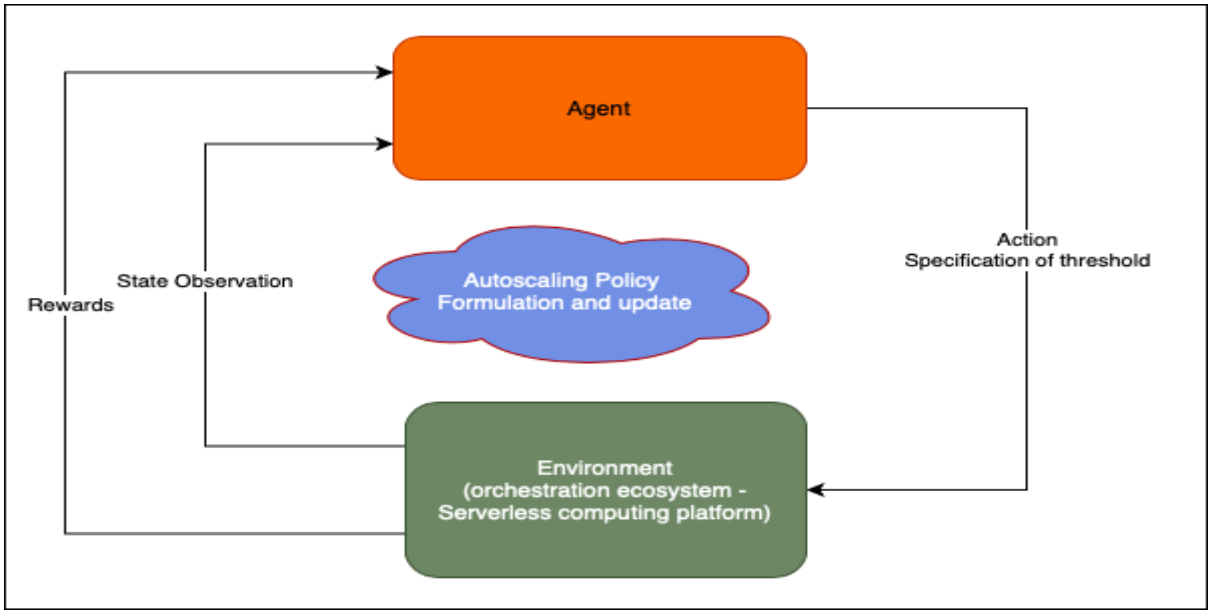

Figure 1: Reinforcement Learning Mechanism for Autoscaling Zafeiropoulos et al. (2022).

Zafeiropoulos et al. (2022) Along with the RL-based autoscaling, researchers have investigated different approaches to improve resource management in serverless computing platforms. For example, introduced a hierarchical autoscaling framework that uses RL algorithms and prediction models to optimize resource allocation.The goal of their approach is to achieve balance between energy consumption, latency and resource utilization. Figure 1 represents the reinforcement Learning Mechanism for Autoscaling Zafeiropoulos et al. (2022).

Despite the promising result observed in RL-based autoscaling, numerous challenges still left to be tackled. One such challenge is the shortage of real-world datasets with enough data for training RL agents. The complex task of modeling the complex interactions between workloads, resource availability, and application performance in serverless environments, is another hurdle. Moreover, RL algorithms can be computationally intensive, requiring the development of efficient implementation strategies to manage the high-dimensional state spaces and action spaces encountered in serverless autoscaling scenarios.

Even with these challenges, RL remains an effective and promising approach for enhancing autoscaling in serverless computing platforms. With increasing accessibility of datasets with real-world autoscaling and ongoing improvement of RL algorithms, RL-based autoscaling is expected to play a major role in optimizing resource usage, reducing costs, and enhancing performance of applications in serverless environments.

3 Methodology

3.1 Data Collection

Data collecting acted as an accelerator for the proposed solution's deployment and evaluation. A dataset provided by Microsoft online using the GitHub platform Azure (n.d.a) became known as having recorded practical serverless execution patterns. The dataset has been made available with log records captured on Microsoft Azure Functions for 2

weeks beginning on January 31, 2021, at 00:00. Microsoft Azure has published and made this dataset accessible to individuals with a Creative Commons Attribution 4.0 License Bannon (2022) Azure (n.d.b). This 1.98 million-row data file was legally copied, edited, and shared for academic purposes Bannon (2022).

Along with the Microsoft dataset, large amount of data were produced in log files during the project's development and project's execution. In order to enable smooth integration with machine learning models and expect future invocations based on noticeable patterns from historical trends, these logs were created carefully. It is important to focus that no private or sensitive data was added to any dataset related to this research at any point during the whole procedure. This well planned method highlights the commitment to upholding the research's moral principles while utilizing data-driven insights to advance the study. The result of these efforts improves in the field of research and opens the door to new developments in understanding and making sense of serverless computing paradigms.

3.2 Data Preparation

A well-known data mining technique called Knowledge Discovery in Databases (KDD) was thoroughly used to prepare the Microsoft dataset. Bannon (2022) The schema that gave important data for each function request was shaped by an early study that indicated the need for additional attributes. 'App' and 'func,' which stand for encrypted serverless application and function IDs, respectively, were two of these properties. The "end timestamp," which shows the exact instant the function execution was completed, and the "duration," which shows the entire execution time in seconds, were also included in the dataset. Applications are the deployment unit in the Azure Functions environment, and the dataset contained 120 applications that had 420 functions. Remarkably, nine functions showed significant activity, each providing more than 40,000 entries; these functions together account for 70% of the collection, or 1.3 million records Bannon (2022).

The investigation into the temporal dynamics showed that the events' inter-arrival times did not fit the Poisson distribution model. Rather, it showed a discrete but positive and continuous distribution within the sample set. The formula $p(A \cap B) \neq p(A)p(B)$ raised concerns about the possibility of variable independence because of daily and weekly temporal patterns. High levels of data variability were suggested by the coefficient of variance ($CV = \frac{\sigma}{\mu}$). Before the dataset was made available, trace timestamps were adjusted, beginning at zero, to improve its usefulness. During the implementation phase, actual start timestamps were computed, with the baseline being set at $A =$ January 31, 2021, at 0:00. The careful preparation of the data guaranteed a strong basis for the subsequent analysis and the creation of efficient machine learning models.

$$\text{start timestamp}_i = A + \Delta t \cdot \text{end timestamp}_i - \Delta t \cdot \text{duration}_i$$

Following a thorough examination of the dataset, the records were logically placed in order along their natural timeline, resulting in a well-organized presentation of the events. The computation of the time difference along the designated axis was done for the start timestamp of row n+1 and row n and so on. With regard to the temporal dynamics between subsequent function requests, this time difference turned out to be an important characteristic. The computed values functioned as vital inputs for the simulation script, which was essential to its capacity to coordinate the intervals between function calls. The well-crafted script made use of this feature to include millisecond-long pauses between

functions, accurately mimicking real-world time intervals between function executions. Because it reflected the complexities of real-world scenarios including function requests and their temporal dependencies, the simulation script's refined approach to temporal control enhanced the simulated environment's authenticity and accuracy.

$$\text{delay}_i = \text{start\_timestamp}_{i+1} - \text{start\_timestamp}_i$$

3.3 Machine Learning Techniques

When it comes to optimizing serverless systems, one carefully thought-out tactic is to use machine learning techniques to forecast the best times for functions to execute. This strategy's main component is the discovery of timestamps linked to function execution, which are vital inputs for a background process that is currently operating on the server. This background operation makes sure that functions are prepared to run smoothly at the scheduled times by installing required dependencies and preemptively warming Docker containers. The predicted time period, which is a 5-hour window, is closely linked to the complete orchestration. The 10 days used in load testing trials were not included in the training data used to refine the prediction models, highlighting a robust and diversified dataset.

This study used Q-learning and the Long Short Term Memory (LSTM) algorithm, two well-known machine learning techniques. Based on past data, the earlier method—supervised learning—performed exceptionally well in predicting execution times. Whereas Q-learning shown remarkable efficiency in enhancing function execution schedules, resulting in superior overall outcomes throughout the studies. Although linear regression, was also investigated, it was not used in the trials since LSTM performed better.

Multiple research investigations have demonstrated the effectiveness of neural networks, particularly when processing time series data. It was found that unsupervised learning strategies, such clustering, were more appropriate for specific activities, whereas supervised learning strategies, like linear regression, were suitable for tasks involving the prediction of historical data. Previous research on cold start mitigation had extensively explored reinforcement learning, which led to its consideration and eventual removal in favor of neural networks. Essential Python packages, such as Pandas for data processing, Numpy for numerical operations, and Tensorflow for neural network construction and training, made it easier to apply this machine learning-driven approach. This thorough method highlights the importance of careful algorithm selection and a well-curated dataset in attaining effective and responsive serverless computing, in addition to showcasing the versatility of machine learning techniques in optimizing serverless operations.

3.3.1 Linear Regression

A fundamental component of statistical methods, regression analysis is a key instrument for determining the complex relationships between dependent and independent variables. It was possible to get insight into the geometric meaning and originality of the actual risk minimizer by showing the estimator's distribution's progress to a multivariate normal distribution and obtaining the normal equations. Su et al. (2012) Determining and estimating the parameters of a function that offers the best fit for a given dataset is the main objective of regression analysis. To understand relationships between variables and explaining how changes in one variable translate in another is the fundamental compon-

ent of this analytical method. Within regression analysis, there is a range of linear types: basic regression models, in which a single independent variable predicts a dependent variable, to more intricate multiple regression models that take into account many predictors. Because it makes it easier to forecast outcomes based on changing independent factors, this adaptable statistical tool is quite useful, particularly when working with continuous dependent variables. Korkmaz (2021) Regression analysis is a widely used technique in many different domains. It is useful for quantifying and forecasting the impact of changes in individual variables on the continuous dependent variable, as well as for understanding the nature of relationships within datasets. For researchers and analysts wishing to gather significant insights from their data—whether they are studying scientific phenomena, identifying economic trends, or assessing the success of programs—regression analysis is a necessary instrument.

$$Y = b_0 + b_1 X + \varepsilon$$

Where:

- Y is the dependent variable.
- X is the independent variable.
- b0 is the y-intercept (the value of Y when X is 0).
- b1 is the slope of the regression line (the change in Y for a one-unit change in X).
- ε represents the error term, which accounts for the variability in Y that cannot be explained by the linear relationship with X.

3.3.2 Long Short-Term Memory:

A neural network architecture with specialized elements, such as a cell state, gates, and activation functions, is created in order to achieve Long Short-Term Memory (LSTM). Together, these parts allow the network to store and update information selectively across time, which enables it to identify long-term dependencies in sequential data. LSTMs are well-suited for tasks like time series forecasting and natural language processing because they can successfully model and make predictions based on sequential data by precisely directing the flow of information through the gates and updating the cell state. Nguyen et al. (2022) The LSTM model is trained on the previous patterns of function invocation to capture the dependencies between the function request patterns and the number of future invocation requests. The quantity of pre-warmed containers needed to minimize cold-start delays is then estimated in advance using this data Kumari et al. (2022).

In order to forecast function invocation times and keep an efficiently warm function container queue, LSTM networks are essential. The implementation of an LSTM network is an advanced technique that facilitates the deliberate prediction of invocation timings, which helps to create a container pool that is responsive. This approach also includes a container pool technique, indicating a dedication to resource optimization and improved scalability. Agarwal et al. (2023) The empirical results demonstrate the effectiveness of the suggested solution and exhibit better results than baseline methods. Remarkable results include an important increase in response time and a significant decrease in the

frequency of cold starts. The paper's strong points are its realistic implementation of reinforcement learning in the context of serverless computing and its thorough assessment, which compares the suggested solution to accepted baseline methods. Nonetheless, it is critical to recognize some constraints, including the controlled nature of the laboratory setting and the possibility of increased data and time expenses related to Reinforcement Learning (RL) methods.

3.3.3 QLearning (Reinforcement learning)

With reinforcement learning, an agent gains decision-making skills by acting and getting feedback in the form of rewards or penalties. It's particularly effective for solving problems modeled as Markov Decision Processes, where the outcome of an action depends only on the current state and not on the sequence of events that preceded it .Q-learning is a subset of reinforcement learning, characterized by its use of a value-based method to iteratively improve an agent's policy by learning the optimal actions to take in various states through a system of rewards and penalties.

The Learning algorithm works in environments with a discrete, finite number of states and actions, making it suitable for scenarios where decisions are based on uncertain outcomes, such as prewarming decision optimization .Kostrikov et al. (2021) The Q-learning agent always aims to learn optimal policy by estimating the value of state-action pairs, which is captured in a Q-table. This is particularly relevant to our focus on optimizing decisions about when to prewarm containers in advance. The LSTM model was used to make predictions about future states in the system, focusing on time-series forecasting due to its capability to remember long-term dependencies. These predictions provided a basis for the Q-learning agent to make informed decisions on when to pre warm a container. Kumar et al. (2020) The agent uses a rewards function that takes into account the costs involved in preparing resources early and tries to avoid unnecessary preparation if the expected delay is short.

The learning implementation uses an epsilon-greedy strategy for action selection, which alternates between exploration (with probability ϵ) and exploitation (with probability $1-\epsilon$) Kumar et al. (2020). This method allowed our agent to balance learning about the environment (exploration) and making the best decision based on present knowledge (exploitation).

The update rule for Q-learning is expressed as:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[R(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

where:

- $Q(s,a)$ is the Q-value for state s and action a.
- α is the learning rate,
- $R(s,a)$ is the reward for taking action a in state s,
- γ is the discount factor,
- $\max_{a'} Q(s',a')$ is the maximum estimated Q-value for the next state s'.

Type	Tool/Platform
Operating System	Linux Ubuntu Server 18.04 LTS (HVM)
Virtual Machines	Amazon Web Services (AWS) EC2
Machine Learning	Keras Tensorflow 2.15.0
Serverless Platform	Apache OpenWhisk 1.0.0 (open-source)
Container Technology	Docker 20.10.17 CE
Programming Language	Python 3.11.5
Performance/Load Testing	Java (openjdk-11) & Apache JMeter 5.6.2

Table 1: Used Platforms and Tools

3.4 Platforms and Tools

Table 1 is the table where all platforms and tools is mentioned.

4 Design Specification

4.1 Experimental Setup

Table 1 provides a representation of the setup and configuration for the experimental portion of this investigation. Provisioning two 't2.large' AWS EC2 instances, each with two virtual CPUs, eight gigabytes of RAM, and sixteen gigabytes of SSD volume storage, comprised the infrastructure used for this experiment. The Ubuntu Server 18.04 LTS operating system was chosen as a key component of this configuration due to its excellent interoperability with the Apache Serverless Framework. Notably, it was discovered that the more recent operating system versions were less compatible, particularly with regard to OpenWhisk's installer techniques. Following difficulties and several unsuccessful installations on more recent versions, Ubuntu Server 18.04 LTS was determined to be the best option for the testing setup.

Using core components like Docker for containerization, Kafka for real-time streaming, Djemame et al. (2020) Nginx as the web server, Consul for networking, and CouchDB as the NoSQL document database, OpenWhisk, a serverless platform, was essential to this configuration. Although these software components were used locally for the project, it's important to note that OpenWhisk is the foundation for IBM Bluemix, a web serverless platform. Due to its effectiveness, the installation method of choice was the Docker Compose plugin, version 1.21.2. The main software requirements for this method were Python, Java, and Docker. The meticulously set up test setup established the foundation for the later stages of the research, guaranteeing a stable and harmonious arena for the exploration of serverless computing frameworks and paradigms.

4.2 Architecture Diagram

Figure 2 represents the architecture diagram of project.

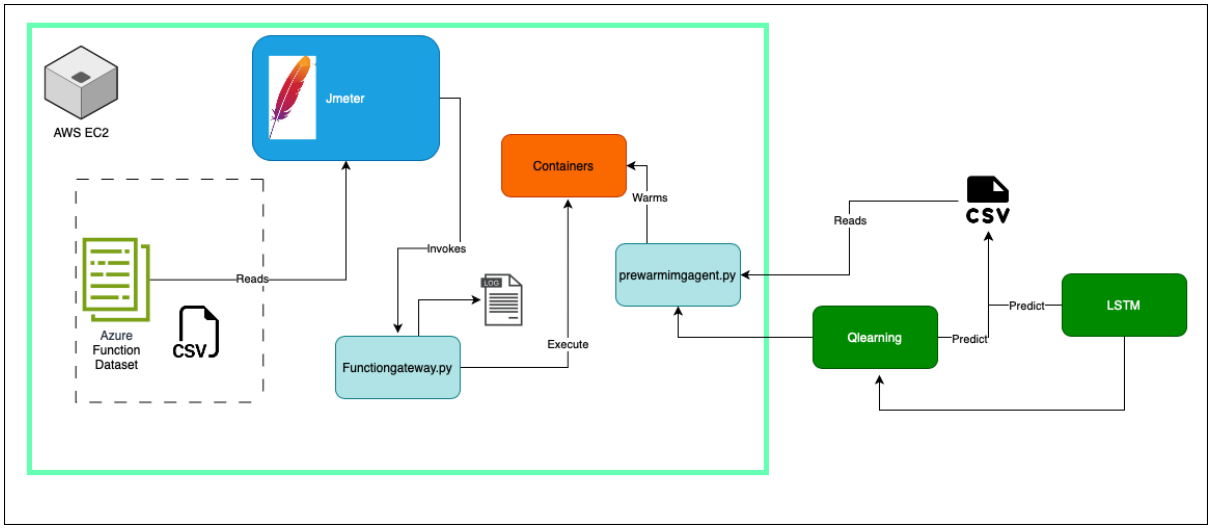

Figure 2: Architecture Diagram

4.3 Algorithm

Please refer **??** for Algorithm used in this project.

Algorithm 1 functiongateway

```
0: function FUNCTIONGATEWAY(execution_version)
0:     beginning_time ← datetime.now()
0:     formatted_start_time ← beginning_time.strftime("%Y-%m-%d %H:%M:%S.%f")
0:     exists, docker_list ← does_container_exist(CONTAINER_NAME)
0:     container_status ← "warm" if exists else "cold"
0:     docker_name ← docker_list[0].strip() if exists else initialize_docker_container()
0:     execute_function_in_container(docker_name)
0:     ending_time ← datetime.now()
0:     formatted_end_time ← ending_time.strftime("%Y-%m-%d %H:%M:%S.%f")
0:     execution_duration ← int((ending_time - beginning_time).total_seconds() * 1000)
0:     timestamp ← datetime.now().strftime("%Y%m%d_%H%M%S")
0:     log_file_name ← f"ML_Run_Log_model_name_execution_version_timestamp.csv"
0:     result_file ← os.path.join(LOG_FILE_PATH, log_file_name)
0:     with open(result_file, 'a', newline='') as result_file:
0:         result_writer ← csv.writer(result_file)
0:         result_writer.writerow([
0:             formatted_start_time, formatted_end_time,
0:             execution_duration, docker_name, PYTHON_SCRIPT, container_status])
0: end function=0
```

Section **??** contains documentation that goes into further detail on the algorithms mentioned above.

5 Implementation

We implemented a serverless test infrastructure using a custom framework built in Python. This system was set up on an Amazon EC2 server, running Ubuntu. Initially,

we planned to use Apache OpenWhisk for the test infrastructure, but we faced some limitations with it, especially regarding container configuration. Due to this, we had to create our own container orchestration module in Python to address these issues. For load testing, we used Apache JMeter, which tested the infrastructure by using function invocation data from the Azure Git repository. The machine learning part of our project involved training and testing models in Jupyter notebooks, using the Keras library with TensorFlow. We should mention that integrating the machine learning part directly into our test infrastructure was not covered in this research. Therefore, we manually added the machine learning predictions to the test infrastructure

5.1 Machine learning models

In this research, we employed three machine learning techniques: Linear Regression, LSTM, and Q-learning. Our initial approach involved a comprehensive analysis of Azure function invocation logs data, spanning a two-week period. We conducted an extensive visualization of the data to discern patterns, focusing on frequently invoked functions, daily and hourly invocation trends. Subsequent to the preliminary analysis, a specific two-week dataset exhibiting the most representative patterns was selected. We then applied the Interquartile Range (IQR) method to this dataset to filter out anomalies, ensuring the data's integrity for further testing and evaluation. Linear regression was deployed first, serving as a baseline model due to its computational efficiency and simplicity. The model's predictions informed our understanding of the basic trends within the data. Following this, an LSTM model was trained on the cleansed dataset to predict wait times. The LSTM's ability to capture temporal dependencies made it suitable for forecasting the server load and response times. Finally, we optimized a Q-learning algorithm through an iterative process of trial and error. Key parameters of the learning agent—learning rate, discount factor, and exploration probability—were meticulously adjusted to optimal levels after multiple experimental runs, enhancing the agent's decision-making process for container prewarming strategies. This fine-tuning was crucial to balance the trade-off between exploration of new strategies and exploitation of known efficient actions.

5.2 Container Management and Simulation Module

The initial test infrastructure was implemented using Apache OpenWhisk, which provides static container pooling capabilities. However, this research requires dynamic prewarming based on predictions, which OpenWhisk does not natively support. To address this requirement, a custom Python-based container orchestration module was developed.

1. Function Gateway (`Function_gateway.py`): This module serves as the entry point for all incoming serverless function requests to the test system. It handles JMeter test plans and receives the machine learning model name and execution version as arguments. It checks for existing containers and creates new ones if necessary before function execution. It also defines a test function that implements Monte Carlo simulation to generate realistic workloads. Finally, it generates logs based on the traffic to track the system's performance.

2. Prewarming Agent (`prewarming_agent.py`): This module is responsible for proactive warming up containers based on prediction CSV files generated by the machine

learning model. Each prediction represents a function invocation and the subsequent wait time. The prewarming agent creates a new container for each line in the prediction CSV file and sets a thread to sleep for the duration of the wait time specified in the input line. This simulates the intervals predicted between successive function calls, ensuring that containers are ready to handle incoming requests when they arrive.

3. Q-Learning Agent (`qlearning_agent.py`): Ideally, the Q-learning agent would be integrated directly with the testing infrastructure to optimize its action-reward feedback and enable truly dynamic prewarming. However, due to current limitations, the Q-learning agent module is used to run Q-learning simulations on the output of the LSTM prediction model. The script generates an improved prediction CSV file that can be used for subsequent testing.

This approach provides a more dynamic and responsive prewarming strategy compared to static container pooling. By proactively warming up containers based on predictions, the system is better equipped to handle incoming requests promptly, minimizing cold start delays and improving overall system performance.

5.3 Apache Jmeter

Apache Jmeter test plan was generated through Jemeter GUI tool to evaluate the effectiveness of the machine learning results, simulating a realistic workload of function invocations based on prediction data. It uses the filtered test data generated during data preprocessing which contains function invocation timestamps and wait times to guide the test execution. The test plan employs a ThreadGroup with a single thread and 2452 loops. Each iteration utilizes a SystemSampler to invoke the `function_gateway` module that triggers a test function call. Additionally, UserParameters are employed to set the start time variable before each iteration, enabling accurate measurement of the overall wait time. Each line in the input test data corresponds to one function invocation, a user parameter START is defined to indicate the start time of test execution. This variable is updated to the start time corresponding to the input line. Jmeter threads are set to sleep for a duration defined as follows, this is implemented to make sure the interval between consecutive function calls are similar to the input test data.

$$\text{sleep duration} = \text{START\_TIME} + \text{WAIT} - \text{CURRENT\_TIME}$$

A detailed log capturing the start time, wait time, and other relevant metrics is also generated after each run, providing valuable insights into the performance of the execution.

6 Evaluation

The research evaluation is divided into two parts, similar to the implementation process. The first part explores the details of the machine learning implementation and the predictions generated. These predictions are then used for excessive testing in a serverless test infrastructure. Analyzing the collected logs in section 2 **??**, we assess the model's performance and identify key findings related to the research question.

6.1 Machine Learning Model Predictions

6.1.1 Long Short-Term Memory(LSTM)

The Long Short-Term Memory (LSTM) network was used for modeling function invocation patterns throughout the week. The LSTM's ability for capturing temporal dependencies makes it particularly suitable for forecasting invocation patterns of serverless computing environments. For the LSTM model, initial data preparation involved the removal of outliers and the conversion of time stamps into a numerical format suitable for sequential analysis. This preprocessing step included the encoding of time features such as hour and day into their sine and cosine components to maintain their cyclical nature, enhancing the model's ability to interpret time-based patterns. The constructed model consists of an initial input layer shaped to the training data, an LSTM layer with 64 neurons featuring ReLU activation for temporal processing, and two dense layers—the first with 8 ReLU neurons for non-linearity, and a final single neuron with linear activation for outputting the predicted wait time. Normalization of the data was done to ensure consistency in the input scale, which is essential for the model's performance. This process involved adjusting the 'wait' feature based on the training data's mean and standard deviation, setting a standardized baseline for the LSTM network to learn from.This architecture was compiled to focus on the key task of time series forecasting within the given dataset. After model training, the LSTM model's performance was visually evaluated by plotting its predictions against the actual wait times for the test dataset. Fig shows comparison between the model prediction and actual wait times, with a few exceptions at peak intervals indicating areas for potential model improvement. In terms of learning dynamics, the model demonstrated a quicker adjustment to fluctuations in server demand during periods of high request rates. Despite initial discrepancies, the LSTM model adapted over successive epochs, suggesting a capacity to optimize threshold adjustments to prevent SLA violations. Under lower request rates, the LSTM model exhibited a steadier learning curve with fewer deviations from the actual wait times.

This analysis of the LSTM model's predictions provided valuable insights into its predictive accuracy. The trained model has a Mean Squared Error (MSE) of 7976.886869300415(ms) and Mean Absolute Error (MAE) of 69.03350519778282(ms). Through iterative training and evaluation, the LSTM network illustrated a reliable capacity for forecasting server wait times. These predictions were exported to a csv file for validation in the test infrastructure.

6.1.2 Q-learning

Reinforcement learning agents interact with its environment directly and the decision policy is optimized based on the action reward feedback. In this research, we have simulated the QLearning executing rather than manipulating the test infrastructure directly. This was done due to the limitation of the test infrastructure and complexity. Q-learning agent was programmed to interact with LSTM wait time predictions and 2 other optimization parameters. Prediction CSV was preprocessed to aggregate average predictions per minute and the used in the Q Learning implementation forming the states within which the Q-learning agent would operate. Each state represents a possible prewarming opportunity and the agent can choose one of the 2 actions based on the reward. Action can be either 0 or 1, which indicates if the container needs to be created or not. The Q-learning algorithm was designed to reduce the number of containers created, thus

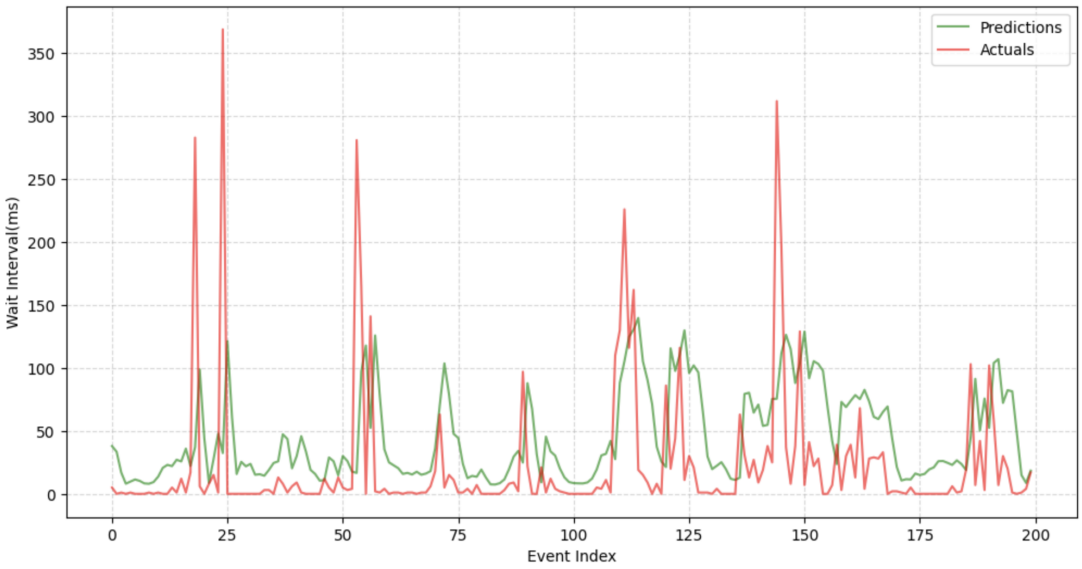

Figure 3: Long Short-Term Memory(LSTM) actual vs prediction output.

controlling costs while maintaining server response efficiency. The reward mechanism penalized both the wait time and the cost of launching new containers, guiding the agent toward a strategy that would selectively instantiate containers based on demand and cost constraints. Throughout the simulation, the Q-learning agent continuously updated its Q-table, which represented the accumulated knowledge of the value of taking certain actions in specific states. This update process was governed by the learning rate, which determined how new information affected existing knowledge, and the discount factor, which quantified the importance of future rewards. A key feature of this implementation was the cost-per-minute limitation, which acted as a threshold to prevent excessive container creation. This constraint was crucial for ensuring the agent did not exceed the budgetary limits set for container creation within each minute.When the agent decided to create a container, it updated the running cost for that minute, and the wait time for that state was reset, indicating the deployment of a new container to manage the request load. If the wait time was under two seconds, the agent added it to the next state's wait time, considering it too short to warrant a new container. The results from the Q-learning agent's training were recorded in a new CSV file, reflecting the agent's evolving policy and the decisions made at each state. The resulting data from this experiment provide a granular look at how reinforcement learning can be applied to make more cost-effective resource allocation decisions in cloud computing environments. This approach has the potential to significantly reduce operational costs and increase efficiency in serverless architectures. Fig shows the changes in wait time after q learning decision. Here the drop to zero implies removal of a prediction based on the agent decision Fig 4.

. . .

6.2 Model Testing in Infrastructure

The validation of the machine learning predictions was conducted using a Python-based test infrastructure. Initially, an Apache OpenWhisk-based test system was utilized to evaluate the performance without any prewarming helpers. However, we later transitioned

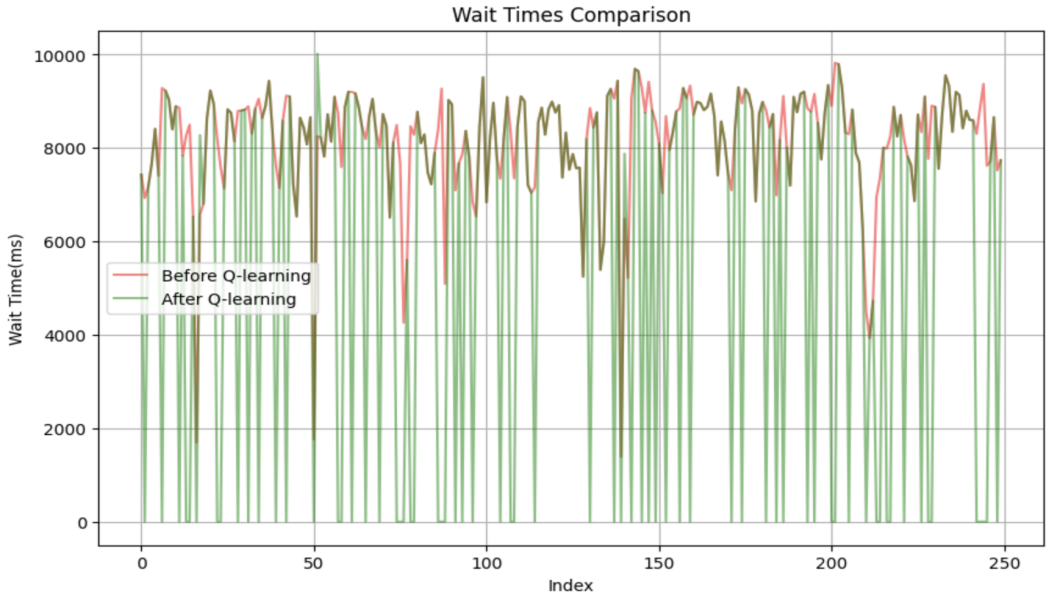

Figure 4: Q-learning output.

to a Python-based container orchestration module to enable implementation of predictions-based prewarming. Logs from the OpenWhisk-based system will not be incorporated in the result analysis. Apache JMeter was employed to simulate a subset of real-world Function trace calls from the Azure dataset to validate each machine learning prediction. A custom Python method implementing Monte Carlo simulation was executed for each test run. The Prewarming Agent module within the system executed the container prewarming logic based on the predictions generated by the machine learning model.

6.2.1 Testing with LSTM

Fig 5 shows the serverless function execution times with the LSTM based predictions compared to the default executions. Improved Latency up to 30% is observed when containers are prewarmed based on predictions made by th eLSTM model. There are spikes in execution time which was observed even with the predictive prewarming. This is caused by the high number of concurrent requests in that timeframe. As the current implementation only prewarms 1 container based on the prediction, this is expected. Further improvements in latency is expected if prewarming pool size is increased to handle concurrent requests.

6.2.2 Testing with Q Learning

In the evaluation of the Q-learning model, a focus was placed on cost efficiency improvements in container management within a serverless computing framework. The implementation of Q-learning aimed to reduce unnecessary container initializations, a key factor contributing to operational costs. The Q-learning algorithm was trained on server log data, where it learned to identify and skip wait times under two seconds. This decision criterion was established to avoid the instantiation of containers when the expected wait time did not justify the resource allocation. Additionally, a cost cap per minute was set for container prewarming, ensuring that the algorithm adhered to budget constraints while maintaining service availability and performance.

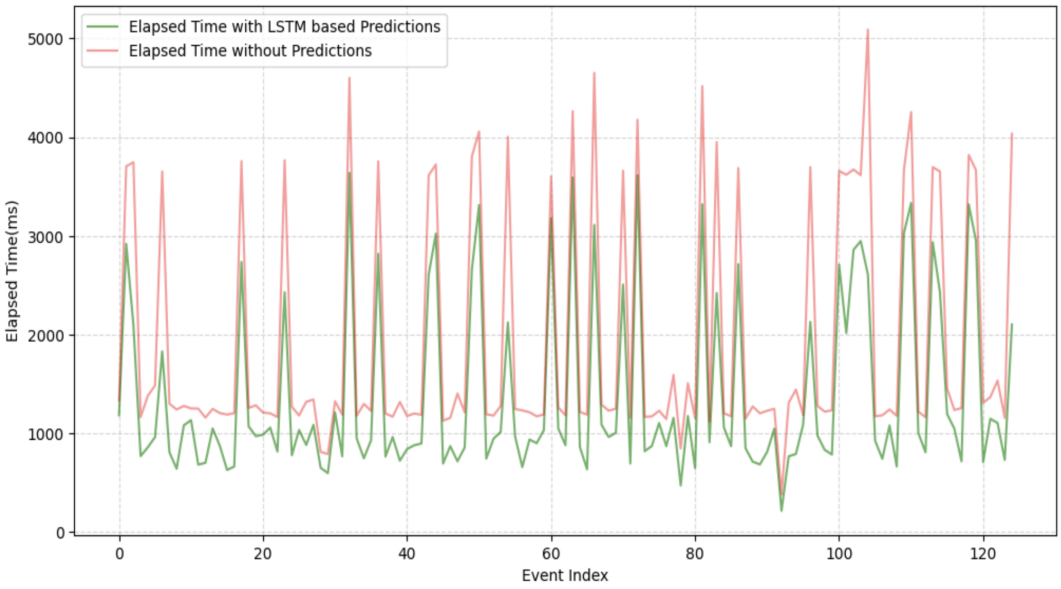

Figure 5: Testing with LSTM output.

Fig 6 The bar chart illustrates the tangible cost savings achieved through the Q-learning approach. The 'Before Q-learning' bar represents the total cost incurred without the Q-learning model's intervention, characterized by a higher frequency of container initializations. In contrast, the 'After Q-learning' bar shows a significant reduction in costs, demonstrating the financial benefits of the Q-learning policy.. 35This cost reduction did not come at the expense of execution times, which remained consistent with those observed when employing LSTM-based predictions. The Q-learning policy successfully optimized container usage, dynamically adjusting the threshold for container creation and thereby reducing the total number of containers used. Fig 7 shows the box plot comparing executing time for LSTM prediction and Q Learning optimization. The interquartile range and mean execution time remained almost the same in both implementations. The strategic avoidance of unnecessary container initializations, coupled with the cost cap strategy, underscores the effectiveness of Q-learning in resource management. By implementing a more selective container instantiation policy, the Q-learning model managed to reduce operational costs without compromising on the quality of service, as indicated by the steady execution times. The Q-learning model's ability to learn and adapt to usage patterns, all the while considering cost implications, positions it as a valuable tool for cloud resource optimization. Moving forward, the continued refinement of the Q-learning algorithm could further enhance cost savings and resource utilization in serverless environments.

. . .

7 Discussion and Conclusion

Long Short-Term Memory (LSTM) model based predictive prewarming was a promising solution for the cold start delay problem. The analysis conducted within the test infrastructure showcased a notable enhancement in execution times, with improvements in the range of 10-35%. Performance improvements observed in this study matched the prior research, validating the efficacy of predictive prewarming in reducing latency.

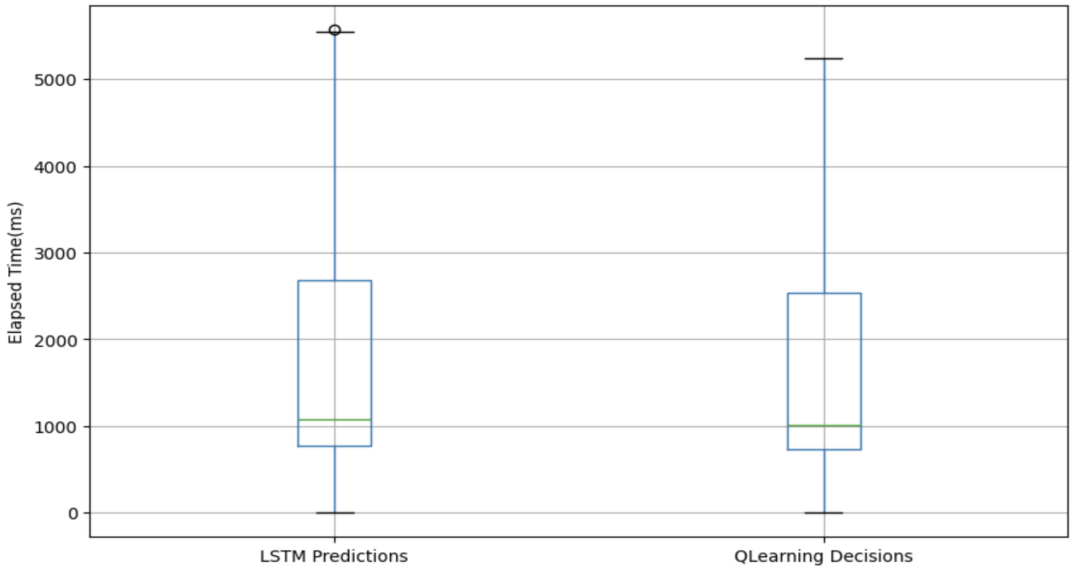

Figure 6: Testing with Q-learning.

Notwithstanding these advancements, the feasibility of deploying aggressive prewarming strategies based on predictive models in real-world scenarios is not cost effective. Aggressive pre-warming could lead to an increase in operational costs, making it a less viable option for cost-conscious serverless computing deployments. It is in this context that the integration of a reinforcement learning (RL) layer becomes important, offering a dynamic approach to prewarming that balances latency improvements against cost implications and other dynamic runtime parameters.While the integration of the Q-learning layer did not result in significant latency reductions, it maintained execution times comparable to the LSTM-based prewarming strategy. This outcome was anticipated as the RL model was designed not to prioritize latency but to conserve costs, achieving a considerable 35% reduction in prewarming expenses.

The findings of this research suggest that the Q-learning model can effectively sustain service latency at acceptable levels while significantly curtailing the costs associated with prewarming. The cost efficiency part of the solution also indicated how Reinforcement learning agent based decisions can be used for enhancing the sustainability of serverless architectures operation which is relevant now. Looking ahead, the integration of the Q-learning agent into the test infrastructure will enable incorporating additional parameters into the reward function which will further improve the prewarming decision making. This would allow a more granular approach to decision-making, taking into account a wider array of operational metrics such as container utilization rates, frequency of function invocations, recent actual invocation history and historical cost data.
The extension of the Q-learning model to consider these additional parameters could enable a more sophisticated approach to resource management. By leveraging a broader dataset and a more complex reward structure, the Q-learning agent can be trained to make even more informed decisions, enhancing its ability to dynamically optimize for both cost and performance. Such an approach will help in making the serverless computing infrastructures not only more responsive to real-time demands but also more economically

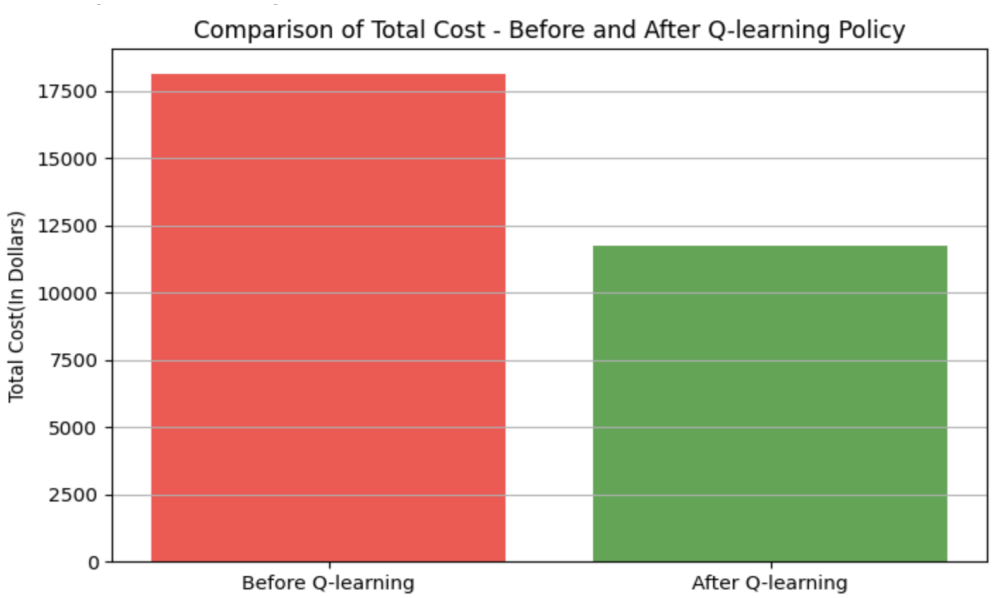

Figure 7: Final results with Q-learning.

efficient, thereby addressing one of the critical challenges faced by the industry today.

References

Agarwal, S., Rodriguez, M. A. and Buyya, R. (2023). Reinforcement learning (rl) augmented cold start frequency reduction in serverless computing, *arXiv preprint arXiv:2308.07541* .

Azure (n.d.a). Azurepublicdataset/azurefunctionsinvocationtrace2021.md, `https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsInvocationTrace2021.md`. GitHub.

Azure (n.d.b). Azurepublicdataset/license, `https://github.com/Azure/AzurePublicDataset/blob/master/LICENSE`. GitHub.

Bac, T. P., Tran, M. N. and Kim, Y. (2022). Serverless computing approach for deploying machine learning applications in edge layer, *2022 International Conference on Information Networking (ICOIN)*, IEEE, pp. 396–401.

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A. et al. (2017). Serverless computing: Current trends and open problems, *Research advances in cloud computing* pp. 1–20.

Bannon, R. (2022). *Leveraging Machine Learning to Reduce Cold Start Latency of Containers in Serverless Computing*, PhD thesis, Dublin, National College of Ireland.

Dantas, J., Khazaei, H. and Litoiu, M. (2022). Application deployment strategies for reducing the cold start delay of aws lambda, *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, IEEE, pp. 1–10.

Djemame, K., Parker, M. and Datsev, D. (2020). Open-source serverless architectures: an evaluation of apache openwhisk, *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, IEEE, pp. 329–335.

Govindan, S. K. (2020). *A Deep Learning Based Framework to Initialize New Containers and Reduce Cold Start Latency in Serverless Platforms*, PhD thesis, Dublin, National College of Ireland.

Korkmaz, M. (2021). A study over the general formula of regression sum of squares in multiple linear regression, *Numerical Methods for Partial Differential Equations* **37**(1): 406–421.

Kostrikov, I., Nair, A. and Levine, S. (2021). Offline reinforcement learning with implicit q-learning, *arXiv preprint arXiv:2110.06169* .

Kumar, A., Zhou, A., Tucker, G. and Levine, S. (2020). Conservative q-learning for offline reinforcement learning, *Advances in Neural Information Processing Systems* **33**: 1179–1191.

Kumari, A., Sahoo, B. and Behera, R. K. (2022). Mitigating cold-start delay using warm-start containers in serverless platform, *2022 IEEE 19th India Council International Conference (INDICON)*, IEEE, pp. 1–6.

Lee, S., Yoon, D., Yeo, S. and Oh, S. (2021). Mitigating cold start problem in serverless computing with function fusion, *Sensors* **21**(24): 8416.

Nguyen, T., Do, T., Le, K., Go, S., Na, S., Kim, D. and Tran, D. (2022). An lstm-based approach for predicting resource utilization in cloud computing, *Proceedings of the 11th International Symposium on Information and Communication Technology*, pp. 173–179.

Oakes, E., Yang, L., Houck, K., Harter, T., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H. (2017). Pipsqueak: Lean lambdas with large libraries, *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, IEEE, pp. 395–400.

Su, X., Yan, X. and Tsai, C.-L. (2012). Linear regression, *Wiley Interdisciplinary Reviews: Computational Statistics* **4**(3): 275–294.

Vahidinia, P., Farahani, B. and Aliee, F. S. (2020). Cold start in serverless computing: Current trends and mitigation strategies, *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, IEEE, pp. 1–7.

Van Eyk, E., Iosup, A., Seif, S. and Thömmes, M. (2017). The spec cloud group's research vision on faas and serverless architectures, *Proceedings of the 2nd international workshop on serverless computing*, pp. 1–4.

Xu, Z., Zhang, H., Geng, X., Wu, Q. and Ma, H. (2019). Adaptive function launching acceleration in serverless computing platforms, *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, IEEE, pp. 9–16.

Zafeiropoulos, A., Fotopoulou, E., Filinis, N. and Papavassiliou, S. (2022). Reinforcement learning-assisted autoscaling mechanisms for serverless computing platforms, *Simulation Modelling Practice and Theory* **116**: 102461.