AI-Driven Auto-Tuning for Serverless Performance Optimization

MSc Research Project
Master of Science in Cloud Computing

Min Ko Aung
Student ID: 23340355

School of Computing
National College of Ireland

Supervisor: Prof. Shaguna Gupta

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Min Ko Aung
Student ID:	23340355
Programme:	Master of Science in Cloud Computing
Year:	2024-2025
Module:	MSc Research Project
Supervisor:	Prof. Shaguna Gupta
Submission Due Date:	15/09/2025
Project Title:	AI-Driven Auto-Tuning for Serverless Performance Optimization
Word Count:	8771
Page Count:	25

Signature:	Min Ko Aung
Date:	15th September 2025

AI-Driven Auto-Tuning for Serverless Performance Optimization

Min Ko Aung
23340355

Abstract

Serverless computing is one of the factors involved in cloud application development to help them expand dynamically and be charged under pay-per-use billing without the need for infrastructure management. Nevertheless, serverless functions continue to be challenging to optimize, especially on cloud systems like AWS Lambda, which are faced with both unpredictable loads and black-box execution patterns. This research proposes the framework for an AI-driven auto-tuning that integrates the hybrid system using machine learning (ML) and reinforcement learning (RL) to facilitate both serverless performance and cost optimization. ML is utilized for predicting the short-term invocation patterns, which are then used by RL agent for dynamically adjusting the configuration parameters, such as memory, timeout, and concurrency. The closed loop pattern allows system learning in response to the execution feedback of latency, improving the decisions, as time goes by, in terms of quality and cost. Experimental validation with synthetic workloads, including bursty, periodic, and noisy traffic, demonstrates how the framework outperforms static and single-strategy baselines. Significant improvements in response time, cost reduction, and service level agreement (SLA) compliance have resulted in and provided the simulated serverless environment.

Keywords: ***Serverless Computing, Auto-tuning, Machine Learning, Reinforcement Learning, AWS Lambda, Cloud Optimization.***

1 Introduction

1.1 Background and Motivation

Serverless computing plays an important role in modern cloud computing, as it allows developers to run applications without worrying about the server management and provisioning infrastructure. AWS Lambda, Azure Functions, and Google Cloud support automatic scaling, event-driven invocation, and pay-per-use pricing (Lynn et al.; 2017). High operational costs can be reduced by using these services; moreover, efficiency can also be enhanced within the resource availability. The motivation for undertaking this research arises from the real-world limitations of using AWS Lambda for serverless application development, as AWS Lambda needs to repeat the adjustments for memory, timeout, and concurrency parameters, and even results in the inefficiencies that spike the latency or over-provisioning. Therefore, an intelligent and automated system becomes necessary to overcome these constraints. In spite of providing abstraction from infrastructure management, serverless computing is still complex for the performance optimization as it needs more than static configurations. The research experiment motivates the investigation of the usefulness of Artificial Intelligence (AI), specifically, in the combination of predictive modeling with adaptive control, with the aim of filling the gap in the performance optimization in dynamic and real-world environments.

1.2 Problem Statement

Although serverless computing offers benefits, its performance optimization is still challenging due to cold starts, dynamic, and limited visibility into the existing infrastructure. Serverless functions are likely to experience the volatile latency, cold-start delays, and heterogeneous resource usage, especially high-concurrency loads or under bursty conditions. These limitations make it challenging to be consistent in responsiveness and reduce the cost of applications (Daraghmeh et al.; 2024). Static configuration, such as having manual configuration for memory, timeout, and concurrency values, is usually insufficient and results in either wasting cost and over-provisioning or increasing latency and under-provisioning (Mampage et al.; 2025a).

1.3 Research Question

Research Questions: How can AI, specifically the integration of machine learning and reinforcement learning, be effectively applied to optimize runtime configuration parameters in serverless environments, while improving performance and reducing cost? To expand further, this research investigates:

- How predictive ML models can facilitate proactive resource allocation to deal with variable workloads.
- How well RL agents can handle configurations under time constraints.
- The performance trade-offs of using AI-driven tuning and manual/static configurations.
- The feasibility of deploying an integrated ML-RL auto-tuning system in production-level serverless platforms.

1.4 Proposed Solution

The proposed solution in the research includes an AI-powered framework, which applies the ML predictive model and reinforcement learning for auto-tuning the parameters of serverless functions. Based on the existing pattern of historical invocation and the information of the resources consumed, the ML components are being used to predict future demand. Later, the RL agent adjusts the essential configuration parameters, such as memory allocation, timeout values, and concurrency values, in real time dynamically. This hybrid mechanism forms closed-loop feedback to predict direct adaptive adjustments and feedback from performance reinforces subsequent decisions.

1.5 Research Objectives

The main purpose of this research work is to design and implement an auto-tuning serverless framework with the integration of machine learning (ML) and reinforcement learning (RL) for the smart configuration optimization. The framework implements a closed-loop feedback system in which ML-based workload prediction informs RL agents that dynamically adjust parameters, including memory, timeout, and concurrency in real time. This technique is verified by controlled experiments based on synthetic workloads that resemble those of an AWS Lambda setup. The performance of the framework is measured by its capacity to minimize latency and to minimize the cost of operation, as well as boost responsiveness as compared to classic static frameworks of operation. In addition, the research aims to support openly accessible methods, datasets, and implementation resources for reproducibility and facilitating future enhancements in optimization of serverless performance.

1.6 Limitations and Challenges

This study has a number of limitations, which could influence the overall application and generalizability in practice. To begin with, the performance and flexibility of the suggested model can differ when implemented on various cloud providers because of the proprietary nature of the back-end infrastructure and resource allocation processes (Bisong; 2019). Whereas the framework addresses adaptive tuning strategies, it does not directly refer to cold start mitigation, although there are some indirect advantages that could be obtained by dynamic memory allocation. A further limitation is that the scope only considers synthetic and experimental workloads, since real-world production serverless-based systems are still not generally accessible. In the last place, reinforcement learning incurs a computational complexity, necessitating adequate reward shaping and care in designing an exploration strategy to promote convergence and stability in training. Another limitation experienced in practice when implementing it in the real-world scenario was the failure of loading complete pre-trained models to AWS Lambda because of the limited size of a code package and the restricted nature of an execution environment, so access to lightweight models or other serving methods is required in practice. The identified limitations are recognized to inform the future development and extension of the applicability of the framework.

1.7 Structure of the Paper

This research paper comprises of seven sections after Section 1: **Introduction**, Section 2: **Related Work**, Section 3: **Research Methodology**, Section 4: **Design Specification**, Section 5: **Implementation**, Section 6: **Evaluation** and Section 7: **Conclusion and Future Work**.

2 Related Work

Serverless computing system architecture, which is stateless and event-driven, has its advantages and disadvantages in terms of optimizing cloud performance. Although serverless functions optimize elastic scaling and prevent infrastructure costs, runtime performance and cost still remain to be resolved immediately. Existing research has shown that using ML or RL independently can improve the handling of workloads, scheduling functions, and optimizing resources. However, an end-to-end integration of ML-RL framework for proactive workload prediction with adaptive parameter adjustment at the function level is not widely available. This review critically analyses sixteen recent IEEE and peer-reviewed publications to classify their contributions among scheduling functions, predicting workloads, managing system-level cost, and dynamic configurations, and finally, identifies the significant gap to fill through the proposed study.

A. ML-Based Workload Prediction and Configuration Tuning

A foundational body of research observes how ML methods are used for workload prediction and optimizing serverless function configurations. According to Majid and Marin (2023) employ the decision trees with regression modeling for estimating resources and placing functions while enhancing SLA compliance. In a similar context, Noble et al. (2023) presents extensive ML forecasting models, including support vector machines (SVM) and k-means clustering to LSTM networks. According to the authors, they focus on how accurate demand prediction is for a wide range of scenarios.

Ponnusamy and Khoje (2024) combines AutoML and explainable AI with the prediction-based scaling frameworks with the aim of reducing cost under dynamic loads. According to Daraghmeh et al. (2024), their experiment applies multi-output regression with PCA-based dimensionality reduction in predicting the rates of invocation in various application levels, with the proof of enhanced latency predictability. On the other hand, Agarwal et al. (2024) demonstrates an ensemble-based model for memory configuration, dynamically optimizing the function for memory allocation in accordance with input structures while improving the effectiveness of performance and cost.

Although these experiments obtain high forecasting accuracy with early resource optimization as well as effective performance, they are still in a fixed setup after they are deployed. Most of the current ML-built frameworks still lack the ability for reconfiguration driven by feedback. While Agarwal et al. (2024) accomplishes dynamic memory tuning, their experiment cannot achieve other parameters, such as timeout and concurrency. Their works only focus on ML methods' prediction power but highlight an important weakness in run-time adaptiveness, so these experiments still open a space to research smarter and closed-loop systems.

B. RL-Based Adaptive Scaling and Scheduling

Majid and Marin (2023) exhibits the conceptual framing with the survey of RL architectures, and when elaborating on design considerations, they discuss how the definition of reward functions can potentially be defined and generalized over action spaces. Reinforcement learning (RL) has become well-known for automating the management of resources in serverless systems by utilizing reasoning techniques with trial-and-error. A hybrid DRL model that supports task scheduling between serverful and serverless systems is proposed by Yu et al. (2024) in order to reduce cold start issues as well as latency with latency-aware dispatching. Qi et al. (2024) presents a carbon-aware size scaling framework (CASA) to control computing policies based on the environmental and SLO target. Mampage et al. (2025b) uses DRL agents in edge computing and investigates how policies can be explored to achieve autoscaling of the latency-sensitive workloads.

Biswas and Kumar (2024) presents further hybrid research work with Q-learning using a random forest to scale dynamic resources. According to the authors, this experiment supports real-time adaptation, but the adaptation for fine-grained function level is still underexplored. Mampage et al. (2025b) also proposes a collaborative approach, while studying cost and latency-optimized scale strategies with DRL agents as well as adapting policies to trade off according to the performance and running costs.

Despite having great potential, these experiments about RL solutions are at the system-level autoscaling without forecasting. Their current models are still siloed, and these models either react to state changes or concentrate on infra policies only. Combining prognostication with real-time decision-making has not been fully discussed, which creates a substantial opportunity for hybrid AI systems to combine prediction and RL convergence.

C. System-Level Enhancements and Distributed Training

According to Gu et al. (2023), GPU-based optimization FaST-GShare is introduced to enable parallel inference to share GPU resources efficiently with spatio-temporal collaboration. Yu et al. (2023) also proposes a dynamic swap model instance, called FaaSwap, to support SLO limitations and minimize GPU idle time. These approaches significantly reduce the inference delay, but they are only based on static or rule-based policies that are difficult to predict or feedback-based tuning.

While focusing more on distributed training, GPU sharing, and infrastructure resiliency rather than function-level performance, system-level optimizations are deployed in different serverless platforms. Sarroca and Sánchez-Artigas (2024) introduces MLLess with the purpose of leveraging batch, function fusion, as well as inference-aware grouping to accomplish cost-efficiency in distributed ML training. Predoaia and García-López (2024) experiment a cloud-agnostic ML deployment framework that can migrate workloads among providers and attain the portable performance and robustness. Although their research works perform well at the system level, these solutions are not able to provide adaptive action during runtime and personalized function tuning.

Other system-level approaches are also utilized, such as auto-scaling with threshold-based or rule-driven patterns such as Biswas and Kumar (2024) demonstration for a reactive model of resource management. However, this approach still depends on hard-coded thresholds and lacks adaptive learning. All these research works focus only on the needs of infrastructure investment and cross-function optimization, as well as configuration using AI at the per-function level, and the performance bottleneck still persists.

D. Research Gap and Contribution Justification

The reviewed works bring useful improvements to predictive modeling or adaptive scheduling for serverless platforms. Still, they offer no integrated feedback system that:

- Learns demand patterns using ML forecasting.
- Tunes parameter settings such as memory, timeout, and concurrency.
- Delivers real-time response via RL-driven actions.

This research fills this gap through the development of a hybrid AI framework for facilitating smart, real-time auto-tuning of serverless functions. The system predicts invocation workloads and makes runtime parameter adjustments continually to optimize latency and cost. Compared to current strategies, this structure offers closed-loop integration of RL and ML at the level of the individual functions—demonstrating a new state-of-the-art advancement.

E. Critical Analysis

The review of the sixteen studies shows several common limitations as summarized in Table 1. ML-based solutions concentrate on the prediction of workload, but with no integration of runtime feedback. RL-based approaches are quite adaptable but reactive and detached from prediction models. System-level solutions focus on improvements in the infrastructure, e.g., distributed ML or shared GPU, but they do not provide control over function-level parameters such as memory, timeout, and concurrency at the fine-grained level (Menasce and Almeida; 2001). More to the point, none of the above reviewed works adopt a unified, closed-loop feedback mechanism that incorporates forecasting and real-time fine-tuning of serverless functions configuration. The proposed research aims to fill such a gap. The proposed framework integrates the visionary workload prediction capability of ML and the adaptive nature of the runtime tuning capability of RL to ensure intelligent scalability of serverless functions in a seamless manner. It optimizes latency and cost inefficiencies by dynamically reconfiguring in real time the parameters of functions based on the predicted demands and also based on actual performance measures. From the studied literature, the closest base paper is Biswas and Kumar (2024) paper, which combines Q-learning with a random forest from dynamic scaling. Although the study conducted by Biswas and Kumar (2024) initiates a hybrid methodology that exploits ML-RL, it lacks the ability of parameter tuning at the function-level, and it is not a closed-loop predictive-feedback loop. By adapting their experiment, this proposed research study builds a comprehensive and predictive feedback-oriented optimization system that is applicable in real-time, serverless environments.

Table 1: Summary of Related Work in ML and RL-based Serverless Optimization

Author/Year	Problem	Algorithm	Area	Tool/Technique	Platform	Application	Limitation
Daraghmeh et al. (2024)	Invocation prediction	Multi-output Regression + PCA	ML-Based	Regression, PCA	Azure Functions	Invocation Forecasting	No real-time adaptation
Noble, Dev & Joseph (2023)	Forecasting model taxonomy	SVM, LSTM, Clustering	ML-Based	ML Algorithms	Cloud (General)	Workload Forecasting	No practical implementation
Raza et al. (2023)	Function configuration	Decision Tree, Regression	ML-Based	Statistical Learning	Cloud (General)	Function Placement	Static config post-deployment
Kumar & Gupta (2023)	Memory allocation tuning	Ensemble Learning	ML-Based	Ensemble Models	AWS Lambda	Cost-efficient Tuning	No concurrency/timeout tuning
Ponnusamy & Khoje (2024)	Predictive resource allocation	AutoML, Explainable AI	ML-Based	AutoML, XAI	AWS	Cost Optimization	No runtime reconfiguration
Yu et al. (2024)	Hybrid task scheduling	Deep RL	RL-Based	DRL Scheduler	Hybrid (Serverless + VM)	Task Dispatching	No forecasting integration
Qi et al. (2024)	SLO and carbon-aware scaling	Policy-Driven RL	RL-Based	Feedback Loops	Serverless	Sustainable Scaling	No ML forecasting
Mampage et al. (2025)	Edge autoscaling	DRL	RL-Based	DRL Agent	Edge	Latency-sensitive Services	No function-level tuning
Biswas & Kumar (2024)	Adaptive scaling	Q-Learning + Random Forest	RL-Based	Hybrid RL + ML	Serverless	Function Auto-scaling	No parameter-specific tuning
Sarroca & Sánchez-Artigas (2024)	ML training optimization	Function Fusion, Batching	System-Level	Grouped Execution	Serverless	Distributed ML Training	Not general workload-focused
Predoaia & García-López (2024)	Cross-cloud ML deployment	Serverless Abstraction	System-Level	Cloud-Agnostic Architecture	Multi-Cloud	ML Portability	No AI-based optimization
Ravi et al. (2023)	Threshold-based scaling	Rule-Based Threshold	System-Level	Scaling Thresholds	Serverless	Function Scaling	No prediction or learning logic
Majid & Marin (2023)	RL review analysis	Analytical Framework	RL-Based	Framework Analysis	Serverless	Scheduler Design	No implemented system
Mampage et al. (2025)	Cost-latency tradeoff	DRL	RL-Based	Reward-based Optimization	Serverless	Cost-Aware Scaling	Infra-level only
Gu et al. (2023)	GPU sharing in inference	Spatio-temporal Scheduling	System-Level	FaST-GShare Framework	Serverless	DL Inference Optimization	No learning control
Yu et al. (2023)	SLO-aware GPU inference	Model Swapping	System-Level	FaaSwap	Serverless	Real-Time DL Inference	Static policy, no feedback

3 Research Methodology

This section describes the entire course of the research that was employed to design, apply, and analyze the proposed AI-based auto-tuning framework in the context of serverless environments. The methodology involves the rationale behind the process of research, the data sourcing plan, the process of developing the models, and the evaluation techniques. Figure 1 shows the workflow, which contains chronological phases, including the definition of the problem to reproducibility, through which the development of the system was carried out. All the stages are outlined below in the following subsections.

3.1 Research Workflow and Stepwise Design

The outlines of the methodology are an iterative, modular development, where each step leads towards a closed-loop ML RL optimization framework of serverless functions. The Figure 1 has a brief articulation of its flow given as follows:

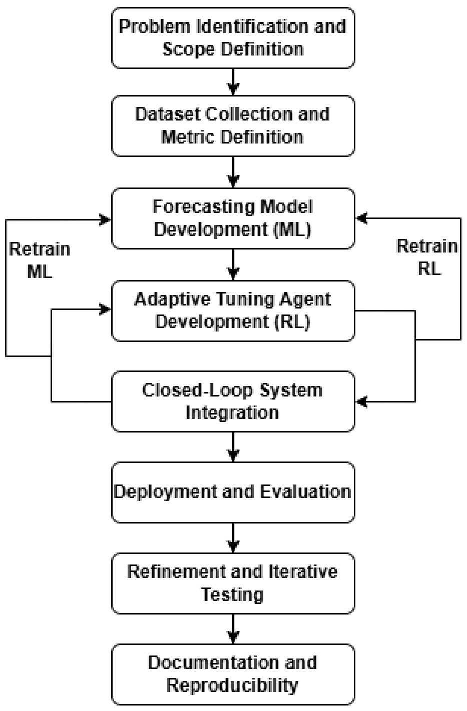

Problem Identification and Scope Definition
Dataset Collection and Metric Definition
Forecasting Model Development (ML)
Retrain ML
Retrain RL
Adaptive Tuning Agent Development (RL)
Closed-Loop System Integration
Deployment and Evaluation
Refinement and Iterative Testing
Documentation and Reproducibility

Figure 1: Research Methodology Workflow

1) Problem Identification and Scope Definition: Although AWS Lambda allows simplification of infrastructure management tasks, other settings (e.g. memory size and timeout), must be adjusted manually by the developer. Static set-ups are prone to over-provisioning (greater cost) and/or under-provisioning (greater latencies and SLA breaches) (Menasce and Almeida; 2001). This step reviewed literature and practice to establish gaps with an emphasis on AI driven approach to balance between latency and costs dynamically. Controllable parameters, workload variability, and performance targets requirements were outlined, which is the foundation of the suggested framework.

2) Dataset Collection and Metric Definition: Since no traces of public serverless workloads exist, synthesized datasets were created to be similar to AWS Lambda functions, invoke, and resource patterns under various workloads. Traffic patterns like steady rates and bursty arrivals were generated by custom scripts, and noise-injected variability was created to represent real-world dynamics. All the records were taken of the invocation rate, the execution duration, cold starts, and memory allocation. Temporal aggregates and labels of the target under ML forecasting, as well as state encodings under RL were part of the feature engineering. Using such datasets and metrics, the evaluation and training of models achieved a realistic view.

3) Forecasting Model Development (ML): The workload forecasting model employs XGBoost, which is a gradient boosted decision tree modelling that fits well for tabular time series data Chen and Guestrin (2016). The model is trained to predict the next expected invocation count, also based on engineered temporal and statistical features, and uses traces of synthesized invocations. The performance measure will be Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE), while optimizing hyperparameters will take place using grid search (Pedregosa et al.; 2011). The results of the forecast are sent back to the RL agent, which enables proactive tuning as opposed to reactive.

4) Adaptive Tuning Agent Development (RL): A reinforcement learning (RL) agent is designed to stream change serverless configurations in real-time as a response to workload forecasts. To describe each state, the custom OpenAI Gym environment uses a normalised vector, which is used to represent the predicted load, latency, and cost, whereas actions represent discrete combinations of memory settings, timeout settings, and concurrency settings. The reward function penalises the violated SLAs and high costs, as well as rewarding low-latency and cost-effective execution (Sutton and Barto; 2018). Though the tabular Q-learning agent with ϵ-greedy exploration and experience replay is programmed as a benchmark, the primary tuning strategy is an Advantage Actor Critic (A2C) policy-based agent, which is more stable and performs best in discrete action spaces. The synthetic workload is trained using multiple episodes, and convergence is measured using trends in cumulative reward.

5) Closed-Loop System Integration: The ML and RL modules are embedded into one unified closed-loop system that indeed works in successive time intervals. The ML model gives an estimate of the amount of work, and the RL agent decides the optimal configuration parameters to implement in the future. An OpenAI Gym simulation is specified to replicate AWS Lambda behaviour, simulating the configuration and reporting on metrics (latency, cost, SLA violations) of the model to give RL feedback. The ML model does not update dynamically in evaluation, but it just gives a forecast. The complete loop is coded in Python with custom serverless simulation code.

6) Deployment and Evaluation: The framework undergoes its initial test in a Python-based simulation built with the OpenAI Gym toolkit (Brockman et al.; 2016) (Van Rossum and Drake; 2009). The synthetic workloads are a stress test of elasticity to three patterns: sudden spikes (bursty traffic), periodic waves (scheduled jobs), and unpredictable jittered intervals (unpredictable usage). It was compared to several baselines, including (1) default static config, (2) ML-only forecasting with fixed tuning, and (3) RL-only adaptation without the prediction. The measures are averaged and include tail latency, execution cost (USD), and cumulative reward of latency cost trade-offs.

To go along with the simulation, a lightweight proof of concept was implemented on the AWS Lambda. The thresholds of working load categories obtained in the trained ML model with XGBoost were built into a Lambda, which will change its own memory allocation levels depending on the parameters of CloudWatch metrics. The possibility of automatic reconfiguration and accurate classification was confirmed with testing, proving effective in a production-like serverless environment. The AWS Lambda PoC employed the trained XGBoost-derived thresholds and did not stage the entire model to prevent dependency size restrictions, and kept the ML-driven logic of the decision process.

7) Refinement and Iterative Testing: Iterative testing is used after post-evaluation of the system to deal with high latency, fluctuation of costs, or the workload behavior. The RL agent iteratively learns with new experience buffers, and the ML forecaster is revalidated/refined when the patterns deviate from the training data. With the assistance of such systems as TensorBoard, training is monitored, and other runs, using different seeds, are done to increase robustness and reduce variance.

8) Documentation and Reproducibility: All the data sets, model code, tuning policies, and evaluation scripts will be published as a public GitHub repository (`https://github.com/minkoaung89/ai-serverless-tuning`) with instructions on how to use it and how it was set up as examples. This helps other researchers replicate, validate, and provide extensibility. The closed-loop architecture also makes continuous improvement possible: this deployment feedback will automatically cause retraining of the ML forecaster or upgrades of the RL policy when performance drops, giving adaptability and robustness to changeable serverless conditions.

3.2 Dataset and Data Source

Public availability of serverless traces is currently unavailable; hence, synthetic datasets are generated to simulate realistic workloads. Such repository congestion consists of bursty, periodic, and irregular traffic with sinusoidal signals, Gaussian noises, as well as random spikes, which are created using a custom Python generator (Brockman et al.; 2016), inspired by AWS Lambda Power Tuning and ServerlessBench. The data was sampled at 5-minute intervals, and the timestamps along with the number of invocations are converted into time-lagged sequences used in ML forecasting as well as RL tuning. The data is divided into training, validation, and test partitions (70/15/15) to make sure that our evaluation works well under diverse traffic environments.

3.3 Ethical Considerations and Reproducibility

This research does not relate, engage, or handle any human/personal data or identifiable information; everything is produced using synthetically generated data production of benchmark-inspired scripts; therefore, no ethical approval is required. All the code, configured files, and the experiment products will be published via a publicly viewable GitHub repository to afford transparency and reproducibility so that others can reproduce, verify, and scale the framework without any legal or ethical considerations.

4 Design Specification

In this section, the technical design of the AI-based auto-tuning framework is given, including modular components, inter-layer communication, and the closed-loop control flow required to optimize functions in real-time. The proposed architecture (Figure 2) is divided into four layers: Monitoring and Data Aggregation, Forecasting Layer (ML), Tuning Layer (RL), and the Feedback Loop.

4.1 Monitoring and Data Aggregation Layer

This layer creates and prepares performance measures that are to be predicted and tuned. The given implementation emulates AWS Lambda-like characteristics such as response time, memory utilization, and concurrency level using Python-based workload generators. A review step is used to standardize the raw synthetic data into a machine learning ready format through normalization and structuring of the data. This study is in a controlled simulation environment, although it was developed to be ultimately integrated with live telemetry (e.g., AWS CloudWatch). The forecasting module takes processed metrics and stores them, with forecasts being evaluated on the same pipeline across development and testing phases.

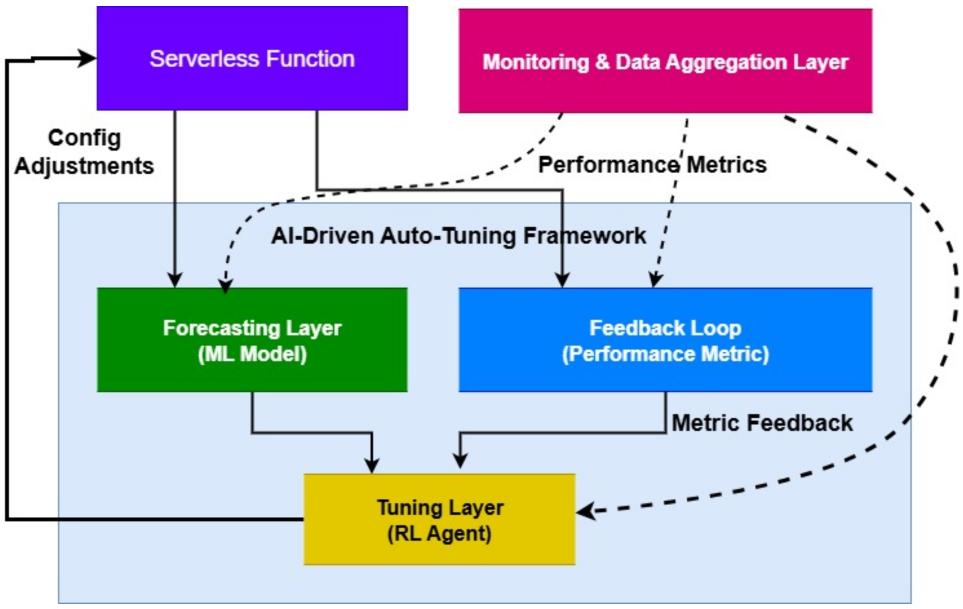

Serverless Function
Monitoring & Data Aggregation Layer
Config Adjustments
Performance Metrics
AI-Driven Auto-Tuning Framework
Forecasting Layer (ML Model)
Feedback Loop (Performance Metric)
Metric Feedback
Tuning Layer (RL Agent)

Figure 2: Architecture Diagram

4.2 Forecasting Layer - Workload Prediction Module (ML)

Short-term predictions of the invocation of functions are generated by the Forecasting Layer to allow the system to proactively adapt to future demand. In this implementation, a trained model built using XGBoost- optimised through grid search was loaded and used on synthetic workload data. The Monitoring Layer historical telemetry is converted into machine learning ready form through feature engineering, such as including temporal features (hour, day, weekend flag), lagged invocation counts, and other forms of rolling statistics. The model is then used to predict the number of times a certain invocation will happen next, and those estimates are put into the state representation of the RL agent. This enables the tuning agent to achieve configuration decisions that respond to future load as opposed to simply being responsive to current load.

4.3 Tuning Layer - Adaptive Tuning Controller (RL Agent)

The Tuning Layer applies projected load and recency of performance data to the selection of ideal serverless configurations. A reinforcement learning agent (Q Learning or A2C) adjusts dynamic settings such as memory, timeout, and concurrency. The state representation is a combination of forecasted demand in the Forecasting Layer with latency, cost, and SLA violation indicators. The reward function penalizes a high cost or SLA violation and rewards low latency, cost-efficient execution (Sutton and Barto; 2018). A replay buffer (in the case of Q Learning) holds experience with tuples (state, action, reward, next state) in order to facilitate sampled training (Watkins and Dayan; 1992). The discrete action-space ranges over 128 to 3008 MB memory allocations, 1 to 15s timeouts, and concurrency levels of 1 to 10, allowing fine-grained adaptation to workload variations.

4.4 Feedback Loop - Function Deployment and Execution Environment

The Feedback Loop emulates the use of serverless functions based on selected configurations by the RL agent and completes the optimization cycle within a controlled environment. Rather than deploy live to the cloud, an emulator that is specific to Python-based imitates core AWS-Lambda behaviors and is run with the selected parameters for memory, timeout, and concurrency, to calculate latency, cost, and invocation success rate (Brockman et al.; 2016). These outputs are recorded and input to the RL agent to come up with improved policy decisions. Although the ML forecasting model is independent within the runtime, it still provides predictions to the RL. This real-time measure-based exchange allows continuous adaptation and optimization without necessarily deploying to production code directly. The four layers combine in a feedback-driven architecture that makes building dynamic serverless functions modular (Bisong; 2019).

4.5 ML and RL Component Details

The framework employs an XGBoost model to predict short-term serverless workloads, which is trained on engineered temporal and statistical features and utilizes a 5-minute prediction horizon. The accuracy of the forecast is measured through MAE, RMSE, and R2 (Chai and Draxler; 2014). The states are all represented as a combination of the predicted load, current configuration, latency, and cost in the RL module. Actions are specified by the combination of memory (128 - 3008 MB), timeout (1 - 15 s), and concurrency (1 - 10). An individualized reward function penalizes the violation of the SLA, as well as high costs, and rewards low-latency, cost-efficient execution. Two RL algorithms are used, Q Learning with ϵ-greedy exploration and the Advantage Actor Critic (A2C) algorithm, to compare the performance of tuning and the convergence behavior.

4.6 Closed-Loop Integration Logic

The RL agent and ML forecaster are synchronized through the closed-loop integration to facilitate the real-time tuning. XGBoost is trained to predict the next interval workload based on the past history of workload invocations, with this prediction then given to the RL agent. The agent is provided with an optimal configuration tuple (memory, timeout, concurrency), which is used in a simulated serverless environment. Latency and costs are the execution performance metrics that are logged and fed back to the RL agent as experience tuples to optimize its policy. Whereas an ML model is fixed during evaluation, the RL agent can relentlessly adapt and respond to changing workloads.

5 Implementation

Proposed AI-based auto-tuning system is implemented as a closed loop solution with ML-based prediction, RL-based decision making, and simulated deployment of serverless function. It is fully written in Python utilizing open-source libraries such as scikit learn, XGBoost, and stable baselines3 (Chen and Guestrin; 2016) (Pedregosa et al.; 2011). An OpenAI Gym compatible custom environment simulates the behavior of AWS Lambda, allowing repeatable testing without having to deploy in the live cloud (Brockman et al.;

2016). The framework of this simulation couples ML forecasting, RL tuning agents, and workload replay to assess optimisation strategies in a variety of conditions.

5.1 Tools and Libraries

The framework is built in Python 3.10 and includes XGBoost as the workload forecasting method, scikit learn to do the preprocessing and evaluation (Pedregosa et al.; 2011), and stable baselines3 and OpenAI Gym/Gymnasium in a custom AWS Lambda simulator to do the RL-based tuning (Brockman et al.; 2016). Data handling will be done through NumPy and Pandas (Harris et al.; 2020), and result visualization will be provided by Matplotlib (Hunter; 2007), and Boto3 will be used to implement the proof of concept using AWS Lambda (Amazon Web Services; n.d.). The list of all dependencies is included in the file requirements.txt to increase reproducibility.

5.2 Platform and Environment Setup

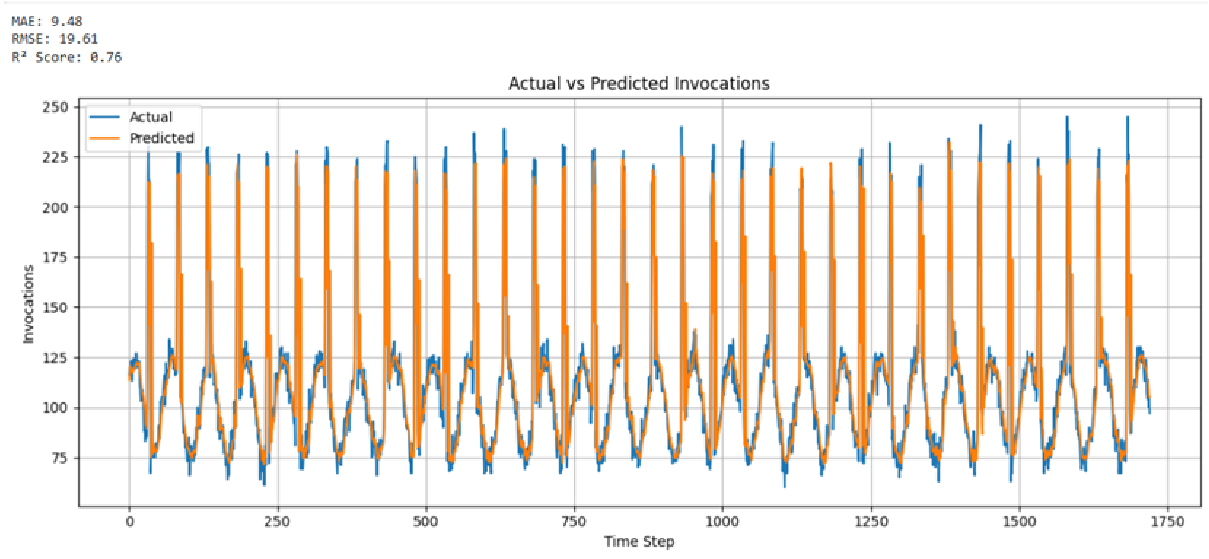
Development and testing of the system is performed in a local, cloud-inspired simulation within a custom OpenAI Gym compatible environment that simulates the behaviour of AWS Lambda (Brockman et al.; 2016), including invocation latency, memory usage, and cost estimates. This enables efficient and repeatable reinforcement learning agent training without cloud execution cost. The main experiments are run on Google Colab and Jupyter Notebooks, having both CPU and GPU support. Although the prototype does not depend on live AWS integration, it can be extended (such as Boto3 and CloudWatch) can be used in the future. Such a simulation-based setup allows rapid prototyping, reproducibility, and experimentation in different workload patterns.

5.3 Dataset Integration and Feature Engineering

The dataset is synthetically created to simulate AWS Lambda call patterns, inclusive of bursty, periodic, and high-irregular workloads. The traffic variations that are generated by a custom Python script contain the timestamps, the counts of invocations, memory allocations, and time spent on executions. Based on this, lagged characteristics and rolling statistics (mean, standard deviation) are calculated to reveal temporal trends. The engineered features are consumed in both ML-based workload forecasting, as well as directly as RL-based configuration tuning state inputs, with data flowing consistently through the framework (McKinney; 2010).

5.4 Machine Learning Forecasting Module

XGBoost has also been selected due to its efficiency and outstanding performance on characterized time-series data to furnish the workload forecasting module (Chen and Guestrin; 2016). It is trained with synthetic traces of invocations that are engineered to contain features such as hour, day, weekend flag, lagged invocation counts, and rolling statistics. The tuning of hyperparameters is done through grid search cross-validation (Pedregosa et al.; 2011). This trained model produced an MAE of 9.48, RMSE of 19.61, and 0.76 R^2 score, reflecting good predictive ability. The results of forecasting are then applied to specify the thresholds of the workload classification according to adaptation tuning. Figure 3 presents the difference between the actual and predicted invocation counts.



```
MAE: 9.48
RMSE: 19.61
R² Score: 0.76
```

Figure 3: Comparison of actual and predicted serverless function invocations

5.5 Reinforcement Learning Tuning Agent

Reinforcement learning (RL) component learnt optimum serverless configurations through contact with an OpenAI Gym simulated environment. It uses a tabular Q learning algorithm, which is simple, interpretable, and appropriate when applied to numbers of discrete states and actions (Watkins and Dayan; 1992). The state space takes the form of a normalised vector of predicted workload, latency, and execution cost, whilst the action space relates to discrete combinations of memory size (128 – 3008 MB), timeout (1 – 15 s), and the concurrency level (1 - 10). High cost and latency are penalised in the reward function equation1:

$$R = -\left(\frac{\text{latency}}{300} + \frac{\text{cost}}{0.1}\right) \tag{1}$$

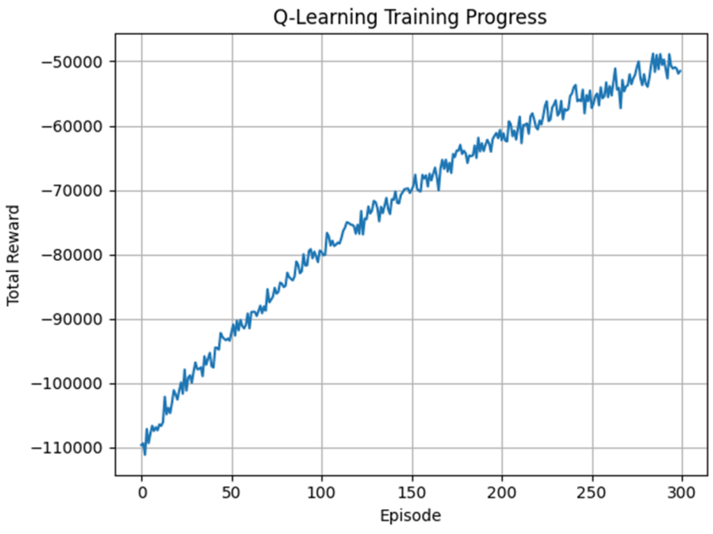


Figure 4: Q-Learning training curve with cumulative reward improvement

The Bellman equation, using learning rate α, a discount factor γ and a decaying ϵ-greedy policy to balance exploration/exploitation, are used to update Q-values. To promote stability in non-sequential order, experience replay is employed in order to have a learning procedure based on past transitions. The agent is trained in over 300 episodes, at which point the convergence is checked through the trend of the cumulative reward criterion. Figure 4 demonstrates the training curve used in Q learning, which implies the continuous but stable improvement of the policy as well as successful learning used to tune optimally the adaptive configuration.

5.6 Closed-Loop Integration and Execution

The system operates in a closed loop cycle with an ML-driven workload forecasting integrated with RL-driven configuration tuning. Each interval, the XGBoost forecaster performs a prediction for the next invocation load, which is included in the state of the RL agent (Chen and Guestrin; 2016). The agent will choose an optimal configuration tuple, such as memory, timeout, and concurrency, and subsequently use this in the local simulator, like AWS Lambda. Their execution statistics, such as latency and cost, are recorded and input into the RL rewarding equation as a continuous policy optimisation process. Forecasting, tuning, deployment, and logging are managed by modular Python scripts to provide a flexible testing and real-time performance adjustments within workload changes (Van Rossum and Drake; 2009).

5.7 Evaluation Outcomes and Logging

The A2C agent seems to converge stably on bursty, periodic, and irregular traffic across synthetic loads, as well as consistent improvements in the latency vs cost trade-offs. The Q-learning was tested with a single workload to act as a baseline comparison. Any run can be reproducible and analysable because it records actions, forecasts, actual load, latency, cost, and reward in CSV format. Reward trend charts prove the adaptability of A2C, as they also prove the integrability of ML-RL in optimization of the serverless solution.

6 Evaluation

The framework is evaluated in a Python-based testbed with XGBoost as the workload predictor, A2C as a reinforcement learning mechanism to tune configuration, and a custom implementation of the OpenAI Gym-compatible simulator that resembles the behaviour of the AWS lambda (Brockman et al.; 2016). The proposed design compares three baselines, which are static default settings, ML-only forecasting, and RL-only tuning, against the ML-RL hybrid approach. Three categories of traffic are patterned in synthesized workloads, including bursty spikes, periodic waves, and irregular jitter. The main performance indicators are mean and 95th percentile latency, execution cost (USD), SLA violation rate, and cumulative RL reward. The configuration enables precision, repeatable experiments to measure accuracy, productivity, and flexibility across a variety of workload conditions without necessarily suffering cloud execution costs.

6.1 Evaluation Metrics

The metrics that are used to assess the framework capture the level of forecasting accuracy and the tuning effectiveness. Performance forecasting is measured by various formulas and equations such as Mean Absolute Error (MAE), Root Mean Square Error (RMSE), SLA Violation Rate, Function Execution Cost, and Cumulative Reward (RL Agent).

6.1.1 Mean Absolute Error (MAE)

Mean Absolute Error (2) calculates the average difference between the real/calculated in invocation counts and the forecast/anticipated in invocation counts with a simple and robust measure of accuracy without being sensitively affected by the outliers (Willmott and Matsuura; 2005).

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{2}$$

where y_i is the actual value and $\hat{y}_i$ is the predicted value.

6.1.2 Root Mean Square Error (RMSE)

Root Mean Square Error (3) estimates the magnitude of prediction error and is more penalizing with large deviations than MAE, thus providing an idea of how well the prediction works in extreme variation (Chai and Draxler; 2014).

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2} \tag{3}$$

6.1.3 SLA Violation Rate

The SLA Violation Rate (4) calculates the percentage of function invocations that experience a latency beyond a specific value (200 ms in this experiment). The lower the value, the better the service level objectives (SLOs) are met, and the more responsiveness to different workloads. This measurement plays a critical role in evaluating the capacity of the tuning agent to sustain latency-sensitive performance in serverless environments.

$$\text{SLA Violation Rate} = \frac{\text{Number of responses exceeding threshold}}{\text{Total number of invocations}} \tag{4}$$

6.1.4 Function Execution Cost

Function Execution Cost (5) measures the cost in monetary terms of the serverless function executed on the AWS Lambda's pricing model. The cost is determined per invocation as a function of memory allocated (in gigabytes), execution time (in seconds), and total number of invocations.

$$\text{Cost (USD)} = \text{Memory (GB)} \times \text{Duration (s)} \times \text{Request Count} \times \text{Unit Price} \tag{5}$$

As of 2025, AWS charges \$0.00001667 per GB-second for function execution.

6.1.5 Cumulative Reward (RL Agent)

Cumulative Reward (6) indicates how much reward was gained by the RL agent in total during training episodes, which reflects its learning progress and, as a result, whether the RL agent successfully makes configuration decisions (Sutton and Barto; 2018). An increasing curve, which goes steadily upward, indicates an effective result in the policy optimization and adaptation in the environment.

$$R = \sum_{t=1}^{T} r_t \tag{6}$$

6.2 Experimental Scenarios

In order to evaluate the adaptability and efficiency of the ML-RL auto-tuning system, five primary experiment scenarios have been developed. These experiments include a variety of tuning strategies, patterns of workloads, algorithm versions, specifications of hyperparameters, and trade-offs between cost and latency. The description of each experiment is briefly presented below, whereas the detailed analysis and interpretation are presented in the Section 6.3.

1. *Strategy Comparison* - Compares the 4 strategies, Static (AWS default), ML-only, RL-only, and a hybrid combination of those to see which is optimal to exclusively run on prediction, exclusively run on adaptation, and combinations of both.

2. *Workload Variability* - Exercises the robustness of the system under three types of traffic that include periodic, bursty, and irregular workloads.

3. *Algorithm Substitution* - Assessment and comparison of the key Advantage Actor Critic (A2C) algorithm with Q learning algorithm and the contrast of the RL methods on the resilience of tuning and performance.

4. *Hyperparameter Sensitivity* - Measures the effect on the stability of training or convergence of the policy induced by altering the most significant RL hyperparameters (e.g. α, γ, ϵ).

5. *Cost vs Latency Trade-off* - This case study analyses the performance of the system when cost is the primary reward instead of the latency of the system, emulating cost-constrained deployment.

6.3 Experimental Results

In this section, results for the 5 scenarios in Section 6.2 are presented with respective charts. Each performs an assessment of a respective feature of the ML-RL auto-tuning framework with latency, cost, SLA violations, and rewards trends as performance indicators.

1. *Strategy Comparison* - According to Figure 5, the hybrid ML+RL (A2C) outperforms the best average latency (118.6 ms) when compared to Static (181.9 ms), ML-only (208.4 ms), and RL-only (155.1 ms). The lowest cost ($0.04586) was attained by ML-only, but it had the highest SLA violations (71.7%), whereas the hybrid model possessed low violations (5.5%) and a moderate cost ($0.09064).

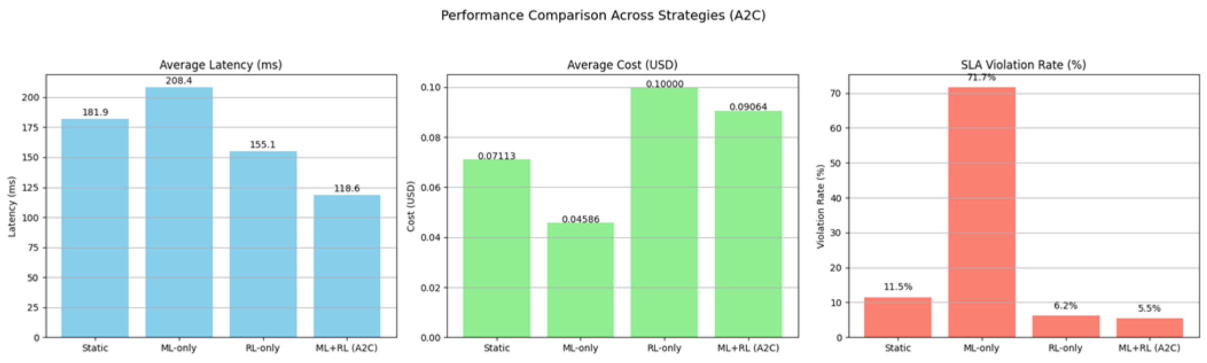


Figure 5: Performance Comparison of Configuration Strategies

2. *Workload Variability* - Figure 6 illustrates that the A2C-based implementation achieved low latency across periodic (161.16 ms), bursty (163.95 ms) and irregular (170.71 ms) workloads with a minor increase. The costs varied from \$0.23019(irregular) to \$0.27342(bursty). The level of SLA violation was least in periodic (5.07%) and bursty (6.80%) and higher in the case of irregular traffic (27.12%).

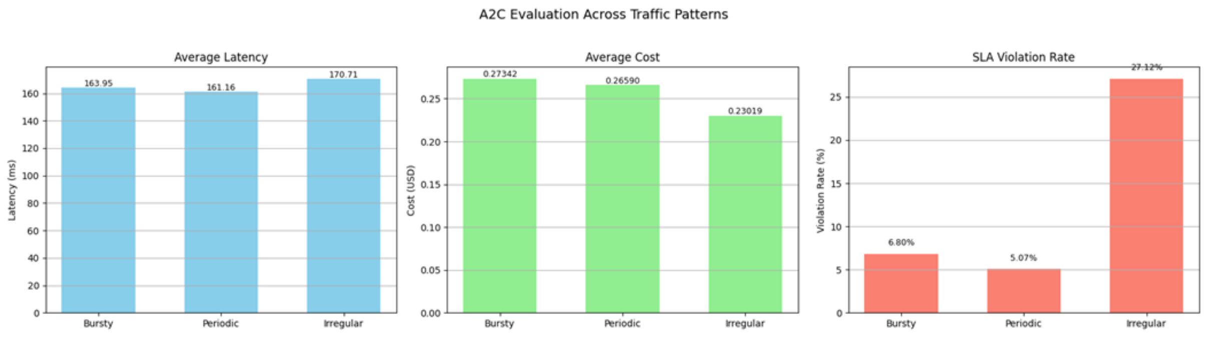


Figure 6: Performance Metrics Across Traffic Patterns (A2C Strategy)

3. *Algorithm Substitution* - In Figure 7, A2C performed better than Q-learning at a lower latency (118.6 ms vs. 159.6 ms), SLA violations (5.5% vs. 16.4%), and at a slightly higher cost (\$0.09064 vs. \$0.06633).

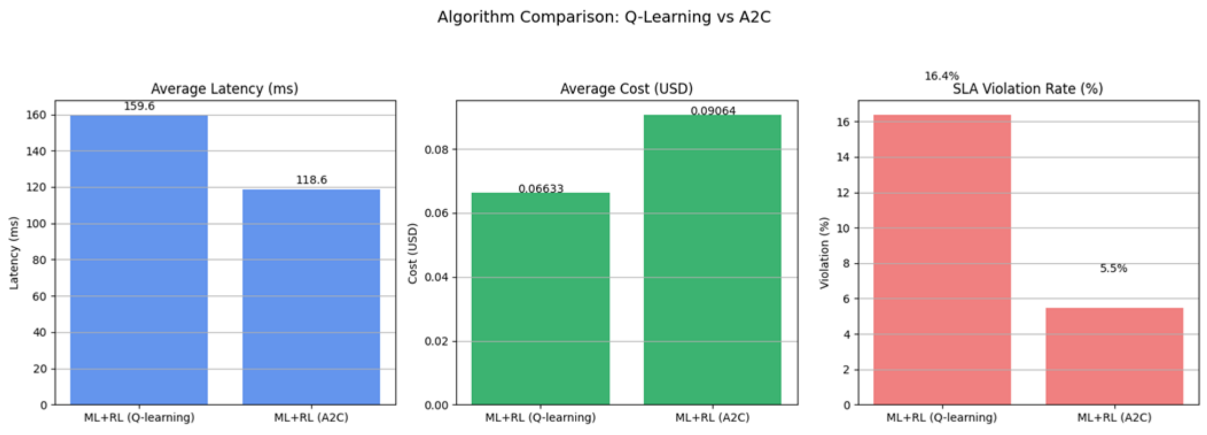


Figure 7: Q-Learning vs A2C: Comparison of Latency, Cost, and SLA Violation

4. *Hyperparameter Sensitivity* - Figure 8 demonstrates that the high learning rates (α=0.1) and zero entropy (Ent = 0.0) produce unstable rewards, whereas moderate (α=0.01) with (γ=0.999) made the most stable learning.

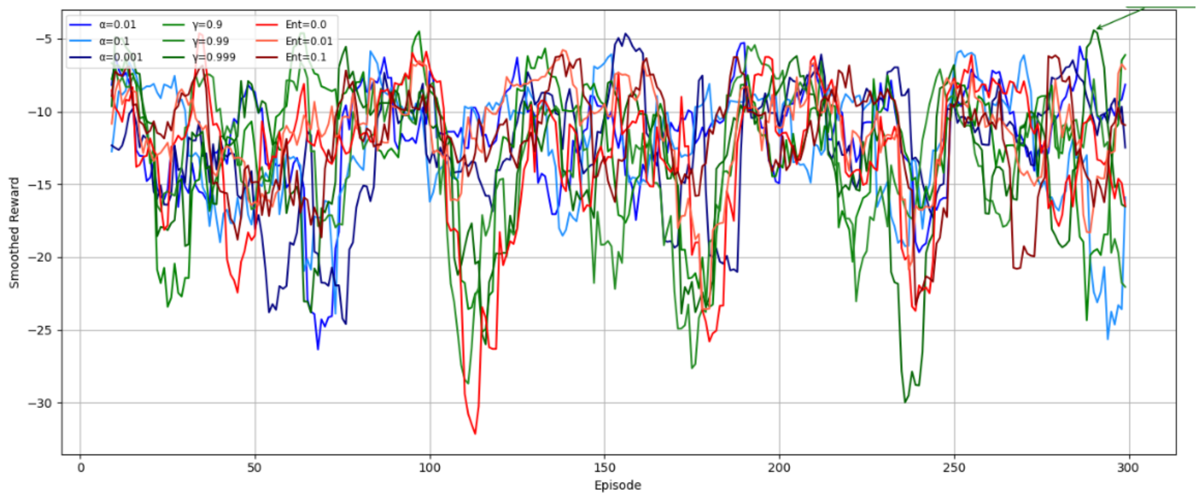


Figure 8: A2C Reward Trends Across Hyperparameter Variations

5. *Cost vs Latency Trade-off* - Figure 9 demonstrates that focusing cost in the reward function minimized the cost by more than 80 percent, but increased the latency to well beyond 200 ms and SLA violations exceeding 99 percent, the importance of balanced reward design was realized.

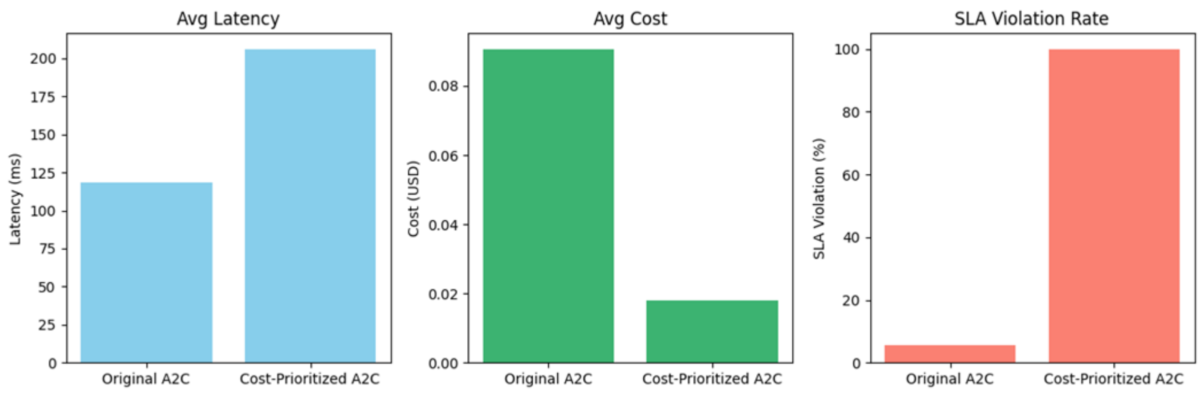


Figure 9: Impact of Cost-Prioritized A2C on Latency, Cost, and SLA Violation

6.4 Discussion

The evaluation outcomes prove that the proposed hybrid ML+RL solution provides flexible and efficient optimization of serverless performance with full compliance with the architecture depicted in Section 4. Using ML-based forecasting combined with A2C-based reinforcement learning, the system exploits predictive workload information and provides flexibility to change configuration settings dynamically in real time. This closed-loop implementation enabled the hybrid method to steadily beat the static, ML-only, and RL-only baselines as well as stay cost-efficient in terms of reducing latency and meeting its SLAs.

The workload forecasting module was essential in predicting the peak demand so that the RL agent could make proactive and not just reactive decisions. Such predictive ability proved useful, especially in the presence of periodic and bursty traffic, where latency remained low and the SLA violations were minimal. Nevertheless, under the conditions of irregular traffic, SLA violations increased, affecting the performance of the agent, which demonstrates the difficulty of the tuning the agent under the unpredictable workloads. This implies that resilience performance may be enhanced under high-variance workloads by adaptive exploration schemes, i.e., by dynamically adjusting the entropy coefficient or learning rate to the volatility of the workload.

In comparison of A2C with Q-learning, the findings indicated that A2C performed better with a lower mean latency and fewer SLA breaches but at an increased cost of 36 %. This trade-off is in line with the policy-gradient nature of A2C, which is prone to over-provisioning of resources to guard the performance. The Q-learning, on the other hand, would have given greater control over the cost but at a reduced level of responsiveness. These results suggest the A2C can be adopted in applications where the performance of a serverless deployment matters more, and Q-learning can be more effective in cost-sensitive applications.

The experiments based on the hyperparameter sensitivity emphasized the need to closely tune RL-based systems. With a high learning rate ($\alpha = 0.1$) or a low entropy (Ent $= 0.0$), training stability deteriorated, but with a medium ($\alpha = 0.01$) was much smoother to converge with ($\gamma = 0.999$). These findings indicate that meta-optimization might be an effective method that can be used in the search process to attain stable learning in various deployment specifications.

Finally, the cost-latency trade-off experiment demonstrated an important architectural consideration: when deciding how aggressively to focus on cost savings, can compromise the quality of service dramatically. The cost-prioritized A2C lowered execution expenses by more than 80 percent; however, latency was more than twice as high, and SLA violations were close to 100 percent. This emphasizes the importance of multi-objective reward shaping that would strike the trade-off between cost-efficiency and performance assurance. Such a balanced goal would be needed in real-world cloud systems to avoid over-optimization with respect to a specific metric without regard to service reliability.

Comprehensively, the experiments confirm the hybrid design to perform as a scaling and intelligent serverless tuning architecture that can respond to various workload contexts. They also show some aspects to be improved, especially regarding dynamic exploration, hyperparameter automation, and multi-objective optimization, and this is the foundation on which future research directions are outlined in Section 7. Further, the practicality of AWS Lambda deployment was shown with a proof of concept in Section 6.5, where an ML-generated threshold classifier effectively varied the memory allocation of the active functions based on how the workload was classified, closing the gap between the simulation results and practice.

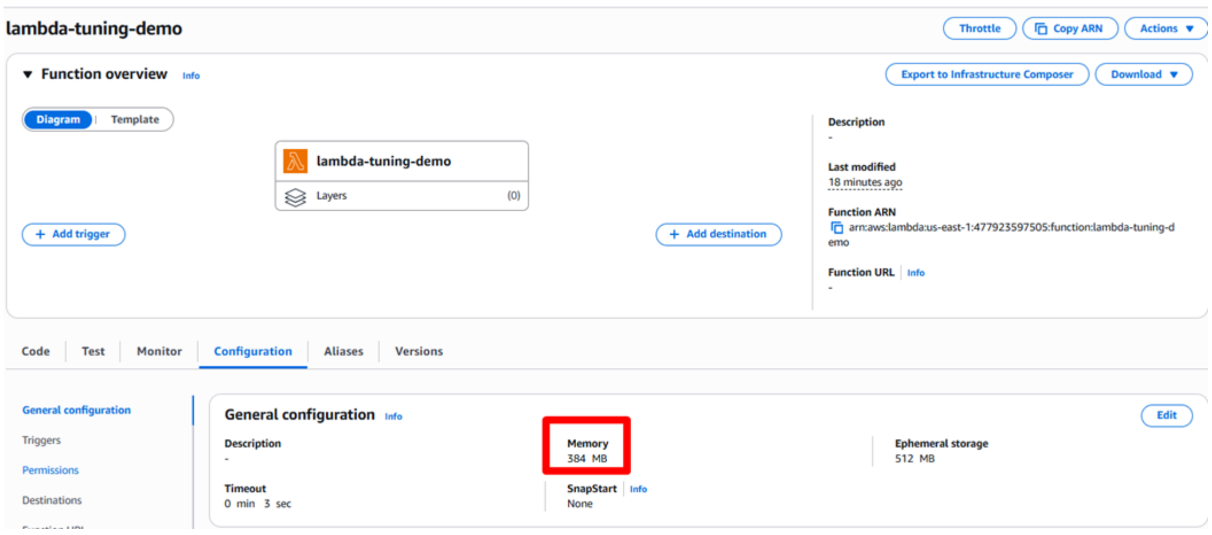
6.5 Proof-of-Concept AWS Lambda Deployment

It was proven that the framework is feasible in the real world on the lightweight AWS Lambda PoC. Retrieving the metrics by querying CloudWatch based on the classification of workload at the threshold given by the trained XGBoost model (avoiding large ML dependencies), and then converting the load to low/medium/high load tags and scaling the memory using the AWS SDK with the function. During the test, it was rated as

Medium load and used **326.32 ms** in 15 invocations and allocated **256 MB** to **384 MB** memory, which verified the feasibility of ML-driven tuning on AWS Lambda.

The following *'output.json'* file created by an invocation of a Lambda function summarizes the decision of the tuning agent with reference to recent performance measures, and a screenshot captured in Figure 10 proves that the ML model performed the memory up-size tuning on AWS Lambda.

```
  cat output.json
{
  "avg_duration": 326.32,
  "invocations": 15.0,
  "decision": "Medium load",
  "new_memory": 384,
  "update_status": "Updated memory to 384 MB"
}
```



Description
-

Last modified
18 minutes ago

Function ARN
arn:aws:lambda:us-east-1:477923597505:function:lambda-tuning-demo

Function URL
-

General configuration

Description
-

Memory
384 MB

Ephemeral storage
512 MB

Timeout
0 min 3 sec

SnapStart
None

Figure 10: Proof of concept Lambda execution using ML derived thresholds

7 Conclusion and Future Work

The research explored how AI (specifically the combination of machine learning (ML) and reinforcement learning (RL)) can be used to optimize the runtime config in a serverless environment while minimizing latency or cost. To experiment with this, a hybrid ML+RL architecture was created, where workloads are predicted with an XGBoost-driven ML module and where memory, timeout, and concurrency are adaptively fine-tuned by an A2C-driven RL agent. The closed-loop architecture integrates predictive intelligence and adaptive tuning, enabling proactive and feedback optimization. Controlled experiments with synthetic workloads demonstrated a performance improvement over a static, ML-only, and RL-only baseline. The hybrid approach displays its resiliency over various traffic patterns (bursty, periodic, and irregular traffic) and consistently reduces SLA violations and average latency while maintaining moderate cost. The results justify the capability of the framework to provide a scalable, performance-driven approach to the serverless optimization of the architecture outlined in Section 4.

As much as the system achieves its objectives, some prospective changes that may be incorporated into the system in the long-term are the redesigning of the system to be compatible with expansion to live cloud environments such as AWS Lambda, the provision of dynamic exploration algorithms to cope with high-dispersion traffic, the automatization of hyperparameter optimization, the utilization of multi-objective reward functions to resolve the trade-off between cost and performance, and the scaling to multi-objective optimization or multi-tenant optimization scenarios. Moreover, the incorporation of real-time streaming of telemetry and a self-correcting feedback loop utilizing services such as AWS CloudWatch and EventBridge may allow ongoing self-learning streamlining, which would seal the difference between academic simulation and an enterprise-ready serverless AI-integrated solution.

References

Agarwal, S., Rodriguez, M. A. and Buyya, R. (2024). Input-based ensemble-learning method for dynamic memory configuration of serverless computing functions, *2024 IEEE/ACM 17th International Conference on Utility and Cloud Computing (UCC)*, IEEE, pp. 346–355. Available at: `https://doi.org/10.48550/arXiv.2411.07444` [Accessed 12 June 2025].

Amazon Web Services (n.d.). Boto3 documentation, Available at: `https://boto3.amazonaws.com/v1/documentation/api/latest/index.html`. [Accessed 24 Jun. 2025].

Bisong, E. (2019). *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*, Apress, Berkeley, CA. Available at: `https://doi.org/10.1007/978-1-4842-4470-8` [Accessed 12 July 2025].

Biswas, T. and Kumar, P. (2024). Optimizing resource management in serverless computing: A dynamic adaptive scaling approach, *2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, IEEE, pp. 1–7. Available at: `https://doi.org/10.1109/ICCCNT61001.2024.10724128` [Accessed 15 June 2025].

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. and Zaremba, W. (2016). Openai gym, arXiv preprint arXiv:1606.01540. Available at: `https://arxiv.org/abs/1606.01540` [Accessed 20 June 2025].

Chai, T. and Draxler, R. (2014). Root mean square error (rmse) or mean absolute error (mae)? – arguments against avoiding rmse in the literature, *Geoscientific Model Development* **7**(3): 1247–1250. Available at: `https://doi.org/10.5194/gmd-7-1247-2014` [Accessed 19 June 2025].

Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*, ACM, San Francisco, CA, USA, 13–17 August 2016, pp. 785–794. Available at: `https://doi.org/10.1145/2939672.2939785` [Accessed 22 June 2025].

Daraghmeh, M., Agarwal, A. and Jararweh, Y. (2024). Optimizing serverless computing: A comparative analysis of multi-output regression models for predictive function invocations, *Simulation Modelling Practice and Theory* **134**: 102925. Available at: `https://doi.org/10.1016/j.simpat.2024.102925` [Accessed 24 June 2025].

Gu, J., Zhu, Y., Wang, P., Chadha, M. and Gerndt, M. (2023). Fast-gshare: Enabling efficient spatio-temporal gpu sharing in serverless computing for deep learning inference, *Proceedings of the 52nd International Conference on Parallel Processing (ICPP)*, ACM, pp. 635–644. Available at: `https://doi.org/10.1145/3605573.3605638` [Accessed 24 June 2025].

Harris, C., Millman, K., van der Walt, S., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. and Kern, R. (2020). Array programming with numpy, *Nature* **585**(7825): 357–362. Available at: `https://doi.org/10.1038/s41586-020-2649-2` [Accessed 26 June 2025].

Hunter, J. (2007). Matplotlib: A 2d graphics environment, *Computing in Science & Engineering* **9**(3): 90–95. Available at: `https://doi.org/10.1109/MCSE.2007.55` [Accessed 27 June 2025].

Lynn, T., Rosati, P., Lejeune, A. and Emeakaroha, V. (2017). A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms, *Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, Hong Kong, China, 11–14 December 2017, pp. 162–169. Available at: `https://doi.org/10.1109/CloudCom.2017.15` [Accessed 30 June 2025].

Majid, A. Y. and Marin, E. (2023). A review of deep reinforcement learning in serverless computing: Function scheduling and resource auto-scaling, *arXiv preprint arXiv:2311.12839* . Available at: `https://arxiv.org/abs/2311.12839` [Accessed 15 June 2025].

Mampage, A., Karunasekera, S. and Buyya, R. (2025a). A deep reinforcement learning based algorithm for time and cost optimized scaling of serverless applications, *Future Generation Computer Systems* p. 107873. Available at: `https://doi.org/10.1016/j.future.2025.107873` [Accessed 18 June 2025].

Mampage, A., Karunasekera, S. and Buyya, R. (2025b). Deep reinforcement learning for scheduling applications in serverless and serverful hybrid computing environments, *IEEE Transactions on Services Computing* **18**: 718–728. Available at: `https://doi.org/10.1109/TSC.2024.3520864` [Accessed 18 June 2025].

McKinney, W. (2010). Data structures for statistical computing in python, *in* S. van der Walt and J. Millman (eds), *Proceedings of the 9th Python in Science Conference (SciPy 2010)*, Austin, TX, USA, 28 June–3 July 2010, pp. 51–56. Available at: `https://doi.org/10.25080/Majora-92bf1922-00a` [Accessed 25 June 2025].

Menasce, D. A. and Almeida, V. A. (2001). *Capacity Planning for Web Services: Metrics, Models, and Methods*, Prentice Hall PTR, Upper Saddle River, NJ, United States. Available at: `https://dl.acm.org/doi/book/10.5555/560806` [Accessed 28 June 2025].

Noble, N., Dev, Y. and Joseph, C. (2023). Machine learning based techniques for workload prediction in serverless environments, *2023 International Conference on Electrical, Electronics, Communication and Computers (ELEXCOM)*, IEEE, pp. 1–6. Available at: `https://doi.org/10.1109/ELEXCOM59242.2023.10370421` [Accessed 30 June 2025].

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V. and Vanderplas, J. (2011). Scikit-learn: Machine learning in python, *Journal of Machine Learning Research* **12**: 2825–2830. Available at: `https://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf` [Accessed 01 July 2025].

Ponnusamy, S. and Khoje, M. (2024). Optimizing cloud costs with machine learning: Predictive resource scaling strategies, *2024 5th International Conference on Innovative Trends in Information Technology (ICITIIT)*, IEEE, pp. 1–8. Available at: `https://doi.org/10.1109/ICITIIT60125.2024.10580717` [Accessed 01 July 2025].

Predoaia, I. and García-López, P. (2024). A cloud-agnostic serverless architecture for distributed machine learning, *Proceedings of the 2024 IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (BDCAT)*, IEEE, pp. 131–140. Available at: `https://doi.org/10.1109/BDCAT63179.2024.00032` [Accessed 02 July 2025].

Qi, S., Moore, H., Hogade, N., Milojicic, D., Bash, C. and Pasricha, S. (2024). Casa: A framework for slo- and carbon-aware autoscaling and scheduling in serverless cloud computing, *Proceedings of the 2024 IEEE 15th International Green and Sustainable Computing Conference (IGSC)*, IEEE, pp. 1–6. Available at: `https://doi.org/10.1109/IGSC64514.2024.00010` [Accessed 19 June 2025].

Sarroca, P. G. and Sánchez-Artigas, M. (2024). Mlless: Achieving cost efficiency in serverless machine learning training, *Journal of Parallel and Distributed Computing* **183**: 104764. Available at: `https://doi.org/10.1016/j.jpdc.2023.104764` [Accessed 20 June 2025].

Sutton, R. and Barto, A. (2018). *Reinforcement Learning: An Introduction*, Vol. 9, 2nd edn, MIT Press, Cambridge, MA. Available at: `https://doi.org/10.1109/TNN.1998.712192` [Accessed 02 June 2025].

Van Rossum, G. and Drake, F. (2009). *Python 3 Reference Manual*, CreateSpace, Scotts Valley, CA. Available at: `https://dl.acm.org/doi/book/10.5555/1593511` [Accessed 04 June 2025].

Watkins, C. and Dayan, P. (1992). Q-learning, *Machine Learning* **8**(3–4): 279–292. Available at: `https://doi.org/10.1007/BF00992698` [Accessed 08 June 2025].

Willmott, C. and Matsuura, K. (2005). Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance, *Climate Research* **30**(1): 79–82. Available at: `https://doi.org/10.3354/cr030079` [Accessed 24 June 2025].

Yu, L., Fu, L. and Sun, C. (2024). Serverless cold start performance optimization based on multi-request processing and adaptive hierarchical scaling, *IEEE Access* . Available at: `https://doi.org/10.1109/ACCESS.2024.3462389` [Accessed 13 June 2025].

Yu, M., Wang, A., Chen, D., Yu, H., Luo, X., Li, Z., Wang, W., Chen, R., Nie, D. and Yang, H. (2023). Faaswap: Slo-aware, gpu-efficient serverless inference via model swapping, arXiv preprint. Available at: `https://arxiv.org/pdf/2306.03622v1` [Accessed 15 June 2025].