A Deep Recurrent-Reinforcement Learning Method for Intelligent AutoScaling of Serverless Functions

Siddharth Agarwal, Maria A. Rodriguez, and Rajkumar Buyya

Abstract—Function-as-a-Service (FaaS) introduces a lightweight, function-based cloud execution model that finds its relevance in a range of applications like IoT-edge data processing and anomaly detection. While cloud service providers (CSPs) offer a near-infinite function elasticity, these applications often experience fluctuating workloads and stricter performance constraints. A typical CSP strategy is to empirically determine and adjust desired function instances or resources, known as *autoscaling*, based on monitoring-based thresholds such as CPU or memory, to cope with demand and performance. However, threshold configuration either requires expert knowledge, historical data or a complete view of the environment, making autoscaling a performance bottleneck that lacks an adaptable solution. Reinforcement learning (RL) algorithms are proven to be beneficial in analysing complex cloud environments and result in an adaptable policy that maximizes the expected objectives. Most realistic cloud environments usually involve operational interference and have limited visibility, making them partially observable. A general solution to tackle observability in highly dynamic settings is to integrate Recurrent units with model-free RL algorithms and model a decision process as a Partially Observable Markov Decision Process (POMDP). Therefore, in this paper, we investigate model-free Recurrent RL agents for function autoscaling and compare them against the model-free Proximal Policy Optimisation (PPO) algorithm. We explore the integration of a *Long-Short Term Memory* (LSTM) network with the state-of-the-art PPO algorithm to find that under our experimental and evaluation settings, recurrent policies were able to capture the environment parameters and show promising results for function autoscaling. We further compare a PPO-based autoscaling agent with commercially used threshold-based function autoscaling and posit that a LSTM-based autoscaling agent is able to improve throughput by 18%, function execution by 13% and account for 8.4% more function instances.

Index Terms—Serverless Computing, Function-as-a-Service, AutoScaling, Reinforcement learning, Constraint-awareness

✦

1 INTRODUCTION

THe growing popularity of event-driven application architectures fuel the increased adoption of serverless computing platforms. Serverless computing introduces a cloud-native execution model that offloads server governance tasks to the cloud service provider (CSP) and aims to reduce operational costs. Serverless features a variety of attributes like a microservices-inspired architecture, high elasticity, usage-based resource billing, and zero idle costs. Function-as-a-Service (FaaS) is a function-based abstraction of serverless computing that decouples an application into functions, small pieces of business logic, that execute on a lightweight virtual machine (VM) or container. These functions generally serve a single purpose, run for a very short duration, and do not maintain a state to enable faster scaling [1]. Functions can be associated with multiple event sources such as HTTP events, database or storage events, and IoT notifications that execute function handlers or business logic and respond to incoming workloads.

Serverless, often used interchangeably with FaaS, has attracted a wide range of application domains such as IoT services, REST APIs, stream processing, and predic-

tion services. These applications may have strict availability and QoS requirements, i.e., throughput and response time while having fluctuating resource requirements that uniquely affect function performance. To address performance constraints and handle complex workloads, FaaS platforms heuristically spin up a new function instance, i.e., function autoscaling, for each incoming request and shut down the instance after service [2] to free up resources. However, FaaS offerings such as AWS Lambda, Azure Functions, Google Cloud Functions, OpenFaaS [3] and Kubeless [4] may choose to re-use a function instance or keep the instance running for a limited time to serve subsequent requests [5]. A recent study [6] asserts that appropriate resource allocation, i.e., CPU and memory, is needed to guarantee QoS fulfillment and improve business value in serverless computing. *Autoscaling* is the process of adding or removing function(s) from a platform, as per the demand, and has a direct correlation with platform performance. CSPs usually employ general-purpose rule-based or threshold-based horizontal scaling mechanisms or utilize a pool of minimum running function(s) [7] [8] to handle function start-up delays while serving workload.

Autoscaling provides an opportunity for CSPs to optimally utilize their resources [9] and share unused resources in a multi-tenant environment. However, configuring thresholds involves manual tuning, expert domain knowledge, and application context that reduces devel-

- The authors are with the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, VIC 3010, Australia.
  E-mail: siddhartha@student.unimelb.edu.au,maria.read@unimelb.edu.au ,rbuyya@unimelb.edu.au

opment flexibility and increases management overhead. Since cloud workloads are highly dynamic and complex, threshold-based autoscaling solutions lead to challenges like function cold starts and hysteresis [10], failing to offer performance guarantees. A *cold start* is a non-negligible function instantiation delay that is introduced before processing the request, while hysteresis highlights the temporal dependency of environment states on the past. Therefore, providing an adaptive, flexible, and online function autoscaling solution is an opportunity to ensure efficient resource management with performance trade-offs in serverless computing. Furthermore, autoscaling approaches employed by existing FaaS frameworks are excessively dependent on monitoring solutions. Although researchers in [11] identify metric collection for thresholds as a bottleneck for autoscaling due to significant collection delay or unreliability, a self-corrective model is demanded to account for underlying variations.

Autoscaling has been actively investigated in the cloud computing domain [1] [10] [12] [13] [14], particularly for VMs, and has periodically highlighted the need for appropriate resource scaling to minimize operational costs and improve performance. Resource scaling is an NP-hard problem [9] [14] and necessitates the realization of complex environmental factors while balancing the system performance between QoS and SLAs. In the past, Reinforcement Learning (RL) algorithms have been applied in the context of VM autoscaling [9] [10] [11] [15] and have demonstrated adaptable performance over traditional methods in capturing the workload uncertainty and environment complexity. But the application of RL for function autoscaling is yet underexplored [15]. RL-based solutions are known to interact with an environment, perform an action, learn periodically through feedback, and account for the dynamics of the cloud environment.

In this work, we investigate the application of Recurrent Neural Networks (RNN), specifically Long-Short Term Memory (LSTM) in a model-free Partially Observable Markov Decision Process (POMDP) setting for function autoscaling. Earlier works [10] [16] [17] [18] employing RL-based autoscaling generally model decision making as Markov Decision Process (MDP) and fall short to discuss partial observability in realistic environments [19] [20]. Furthermore, various existing studies discussed in [10] [16] experiment with RL-based solutions in a simulated FaaS environment, with the research in [11] criticizing this methodology. Simulated FaaS frameworks generally sample factors such as cold start and execution time from profiled data and are insufficient to capture the variability in real environments. Therefore, we examine the integration of LSTM with Proximal Policy Optimization (PPO), a state-of-the-art RL algorithm, to analyze partial observability and sequential dependence of autoscaling actions and find a balance between conflicting CSP and user objectives. We perform experiments with *matrix multiplication* function and compare LSTM-PPO against Deep Recurrent Q-Network (DRQN) and PPO (clipped objective) to infer that in our experimental settings, recurrent policies capture the environment uncertainty better and showcase promising performance in comparison to PPO and commercially adopted threshold-based approaches. We make use of OpenAI Stable Baseline's [21] standard implementation of the LSTM-

PPO and PPO algorithms, and implement our compatible OpenFaaS serverless environment following Gymnasium [22] guidelines.

In summary, the key contributions of our work are:

1) We analyze the characteristics of FaaS environments to identify and model autoscaling decisions as a POMDP. We further hypothesise that scaling decisions have a sequential dependence on interaction history. We propose a POMDP model that captures function metrics such as CPU and memory utilization, function replicas, average execution time and throughput ratio, as partial observations and formulate the scaling problem.

2) We investigate how function autoscaling works, highlight the differences between contrasting approaches and investigate a Deep Recurrent RL (LSTM-PPO) autoscaling solution to capture the temporal dependency of scaling actions and workload complexity. We deploy the proposed agent to the OpenFaaS framework and utilise open-source function invocation traces [23] from a production environment to perform experiments with a matrix multiplication function.

3) We implement a Gymnasium [22] compatible OpenFaaS serverless environment to be integrated directly with the proposed RL agent.

4) We perform our experiments on Melbourne Research Cloud (MRC) and evaluate the proposed LSTM-PPO approach against the state-of-the-art PPO algorithm, commercially offered threshold-based horizontal scaling, OpenFaaS' request-per-second scaling policy, and a Deep Recurrent Q-Network i.e., DRQN, to demonstrate LSTM-PPO's ability to capture environment uncertainty for efficient scaling of serverless functions.

The rest of the paper is organised as follows. Section 2 highlights related research studies. In Section 3, we present the system architecture and formulate the problem statement. Section 4 outlines the proposed agent's workflow and describes the implementation hypothesis and assumptions. In Section 5, we evaluate our technique with the baseline approaches and highlight training results and discuss performance. Section 6 concludes the paper and highlights future research directions.

2 RELATED WORK

In this section, we summarise (see Table 1) existing work on serverless computing, autoscaling in FaaS, and the application of RL in FaaS. We compare existing work based on their key features and provide a detailed background on the Deep Recurrent RL (RPPO) algorithm used in designing our autoscaling policy.

2.1 Serverless Computing and Function-as-a-Service

Serverless computing puts forward a cloud service model wherein the server management or resource management responsibility lies with the CSP. In [2], the authors discuss the potential of this new, less complex computing model introduced by Amazon in 2014. The study briefly explains a function-based, serverless commercial offering of AWS Lambda, i.e., the Function-as-a-Service platform. It highlights three primary differences between traditional

TABLE 1: A Summary of Related Works and Their Comparison with Our Proposed Method.
H: Horizontal Scaling, V: Vertical Scaling

Work	Type	Scaling	Technique	Objective	Environment
[7]	FaaS	H	Threshold-Based	CPU Utilisation	AWS Lambda
[8]	FaaS	H	Threshold-Based	CPU Utilisation	Google Cloud Functions
[14]	Microservices	H,V,Brownout	GRU + Q-Learning	QoS	Testbed
[15]	FaaS	H	Q-Learning	QoS	OpenFaaS
[16]	FaaS	H	Q-Learning, DQN, DynaQ+	QoS + Budget	Simulation, Kubeless
[17]	FaaS	H	Q-Learning	QoS	Kubeless
[18]	FaaS	H	Q-Learning	QoS	Knative
[24]	FaaS	H	Bi-LSTM	Resource	Knative
[25]	FaaS	H, V	Q-Learning	QoS + Resource	Testbed
[26]	FaaS	H	Kneedle Algorithm	QoS + Budget	OpenFaaS
Our Method	FaaS	H	LSTM - PPO	QoS + Resource	OpenFaaS

cloud computing and serverless computing – decoupled computation and storage, code execution without resource management, and paying in proportion to the resources used. The research posits that the serverless or FaaS model promotes business growth, making the use of the cloud easier.

Baldini *et al.* [27] introduce the emerging paradigm of FaaS as an application development architecture that allows the execution of a piece of code in the cloud without control over underlying resources. The research identifies containers and the emergence of microservices architecture as the promoter of the FaaS model in serverless. The study uses FaaS and serverless interchangeably and defines it as a 'stripped down' programming model that executes stateless functions as its deployment unit.

Since the inception of serverless computing, there have been many commercial and open-source offerings such as AWS Lambda, Microsoft Azure Functions, Google Cloud Functions, Fission, and OpenWhisk. These platforms represent FaaS as an emerging technology, but Hellerstein *et al.* [28] put together gaps that furnish serverless as a bad fit for cloud innovations. The authors criticize the current developments of cloud computing and state that the potential of cloud resources is yet to be harnessed. On the contrary, the researchers in [29] argue that the serverless offerings are economical and affordable as they remove the responsibility of resource management and complexity of deployments from consumers. They discuss the opportunities offered by multiple FaaS offerings and give an overview of other existing challenges, and indicate potential approaches for future work.

In an article by Microsoft [30], Rosenbaum estimates that there will be nearly 500 million new applications in the subsequent five years, and it would be difficult for the current development models to support such large expansions. FaaS is designed to increase development agility, reduce the cost of ownership, and decrease overheads related to servers and other cloud resources. The term 'serverless' has been in the industry since the introduction of Backend-as-a-Service (BaaS). Despite the serverless benefits, FaaS experiences a few challenges, categorized as system-level, and programming and DevOps challenges [2] [27] [30]. The former identifies the cost of services, security, resource limits, and cold start while scaling, and the latter focuses on tools and IDEs, deployment, statelessness, and code granularity in the serverless model.

2.2 AutoScaling in Function-as-a-Service

Resource elasticity, analogously used with autoscaling, is a vital proposition of cloud computing that enables large-scale execution of a variety of applications. A recent survey [9] discusses the relevance of cloud resource elasticity for the Infrastructure-as-a-service (IaaS) model to express that autoscaling and pay-as-you-go billing enables infrastructure adjustments based on workload variation while complying with SLAs. On this basis, the study identifies that autoscaling addresses a set of associated challenges, namely, scaling and scheduling which are generally NP-hard problems. Additionally, the research explores the possibility of RL algorithms for autoscaling to approach the complexity and variability of cloud environments and workloads. It is emphasized that utilization of such RL algorithms for scaling purposes can help the service providers to come up with a more transparent, dynamic, and adaptable policy.

Straesser et al. [11] conduct experiments related to cloud autoscaling and assert autoscaling to be an important aspect of computing for its effects on operational costs and QoS. The authors define scaling as a task of dynamically provisioning resources under a varying load and necessitates the automation of processes for highly complex cloud workloads. They discuss that commercial solutions usually operate with user-defined rules and threshold heuristics, and state that an optimal autoscaler is expected to minimize operational cost and SLA violations.

In addition to workload variability, QoS sensitivity is also identified as an enabler for increased operational costs and resource wastage. A microservices-focused autoscaling scheme is introduced in [14] where a trade-off between horizontal, vertical, and a self-adaptable brownout technique is determined based on the infrastructure and workload conditions. The researchers exploit Gated-Recurrent Units (GRUs) for workload prediction and utilize Q-learning for making trade-off updates and scaling decisions. The study asserts that workload prediction is an important factor for autoscaling and acknowledges resource allocation to be an NP-hard problem with multi-dimensional objectives of QoS and SLAs.

In the context of FaaS autoscaling, work in [18] experiments with the concurrency-level setting of Knative, a Kubernetes-based serverless framework, and identify that function concurrency settings have varying effects on latency and throughput of function. Therefore, they utilize the Q-learning algorithm to configure functions with op-

timal concurrency levels to further improve performance. Another work [17] presents preliminary results of applying Q-learning to FaaS for predicting the optimal number of function instances to reduce the cold start problem. They utilize the function resource metrics and performance metrics and apply them to discrete state and action spaces for adding or removing the function replicas, with threshold-based rewards, to eventually improve function throughput.

Similarly, studies like [16] [24] [25] emphasize addressing the dynamicity, agility, and performance guarantees of FaaS by employing RL-based autoscaling solutions. The work in [16] follows a monitoring-based scaling pattern and explores algorithms like Q-learning, DynaQ+, and Deep QL, partially in simulation and practical settings, to reasonably utilize resources and balance between budget and QoS. They aid the agent's training process by sampling simulation data based on probability distribution and running parallel agents to speed up the learning process. The work in [24], discusses the concurrency level in the Knative framework and asserts that identifying appropriate thresholds is challenging, requires expert knowledge, and has varying effects on performance. Therefore, to efficiently use the function resources and improve performance, authors profile different concurrency levels for best performance and propose an adaptive, Bi-Long Short Term Memory (Bi-LSTM) model for workload prediction and determine the number of function replicas using identified concurrency levels. Another study [25] focuses on function response time and states that threshold-based scaling cannot devise a balance between resource efficiency and QoS. Therefore, the authors explore Q-learning to propose adaptive horizontal and vertical scaling techniques by profiling different resource allocation schemes and their corresponding performance. Their proposed state space considers resource requests and limits, along with the availability of GPU components, to model rewards as the divergence from agreed SLO levels. Taking a different approach, the researchers in [15] utilize Q-learning in the context of Kubernetes-based serverless frameworks and propose a resource-based scaling mechanism to adjust function CPU utilization threshold to reduce response time SLA violations. Taking a different approach, [26] proposes an online application profiling technique that identifies a knee point and adjusts resources until the point those changes reflect in performance gain using the Kneedle algorithm in conjunction with binary search. Further, a survey [10] summarises autoscaling techniques for serverless computing under different categories like rule-based, AI-based, analytical model, control theory-based, application profiling, and hybrid technique and envisions new directions like energy-driven and anomaly-aware serverless autoscaling.

These proposals are complementary yet contrasting to each other either in optimization objectives, profiled metrics, or scaling policy. Some fail to address the performance dependency on complex workloads, while few rely on pre-configured thresholds [7] [8] that require expert knowledge and application insights. Few studies focusing on workload prediction assume a fully observable environment and miss out on the temporal dependency of environment states where scaling decisions have been taken. Contradictory to these proposals, we examine a Deep Recurrent RL-based autoscaling solution, particularly LSTM-PPO, to hypothesize that FaaS environments are highly dynamic, partially observable with complex workloads, and that scaling decisions are influenced by environment uncertainty. We model function autoscaling as a partially observable Markov decision process (POMDP) and utilize monitoring metrics like average CPU and memory utilization, function resource requests, average execution time, and throughput ratio to discover an optimal scaling policy. Our proposed RL-based autoscaling agent interacts with the FaaS environment, waits for a sampling period [11] to receive delayed rewards, and feeds the observed environment state to the recurrent actor-critic model. Although a few studies [14] [24] have utilized recurrent networks like LSTM or GRU for workload prediction in serverless context but do not address the temporal relationship between scaling actions and their effect on environment state. Further, we take inspiration from [20] [31] [32] where recurrent models have been utilized to analyze the inter-dependence of environment states and retain useful information to learn optimal policies.

3 SYSTEM ARCHITECTURE AND PROBLEM FORMULATION

3.1 System Architecture

The main components of our autoscaling solution are the Prometheus monitoring service and the DRL agent, which are shown in Fig. 1. For the serverless environment, we deploy OpenFaaS [3], a Kubernetes-based FaaS framework, over a multi-node MicroK8s [33] cluster, a production Kubernetes distribution. OpenFaaS includes a Gateway deployment to expose function performance metrics and Prometheus is configured to periodically scrape function metrics such as execution time, replica count, and throughput ratio. OpenFaaS also packs an *alertmanager* that periodically watches for pre-configured request-per-second scaling threshold to provide horizontal scaling capabilities. The monitoring service further scrapes resource metrics from the Kubernetes API Server, Kubelet, and Node exporters that are utilized by our DRL agent for observation collection at every sampling window. The DRL agent utilizes the standard Stable Baseline3 (SB3) [21] implementation of LSTM-PPO [1] and models the FaaS environment following Gymnasium [22] guidelines, for the POMDP model to be directly used by SB3 algorithms. Additionally, we implement our own version of DRQN [1] using PyTorch [34] [35] for evaluation. We also deploy an HTTP-request generator tool to simulate online user behavior to train and evaluate our DRL autoscaling agent.

3.2 Problem Formulation

Existing FaaS platforms generally exercise threshold-based scaling when a monitored metric exceeds the configured maximum or minimum. Autoscaling of resources is considered a classic automatic control problem and commonly abstracted as a MAPE (map-analyse-plan-execute) control loop [1]. At every sampling interval, the monitoring control loop collects the relevant metrics and may decide to scale
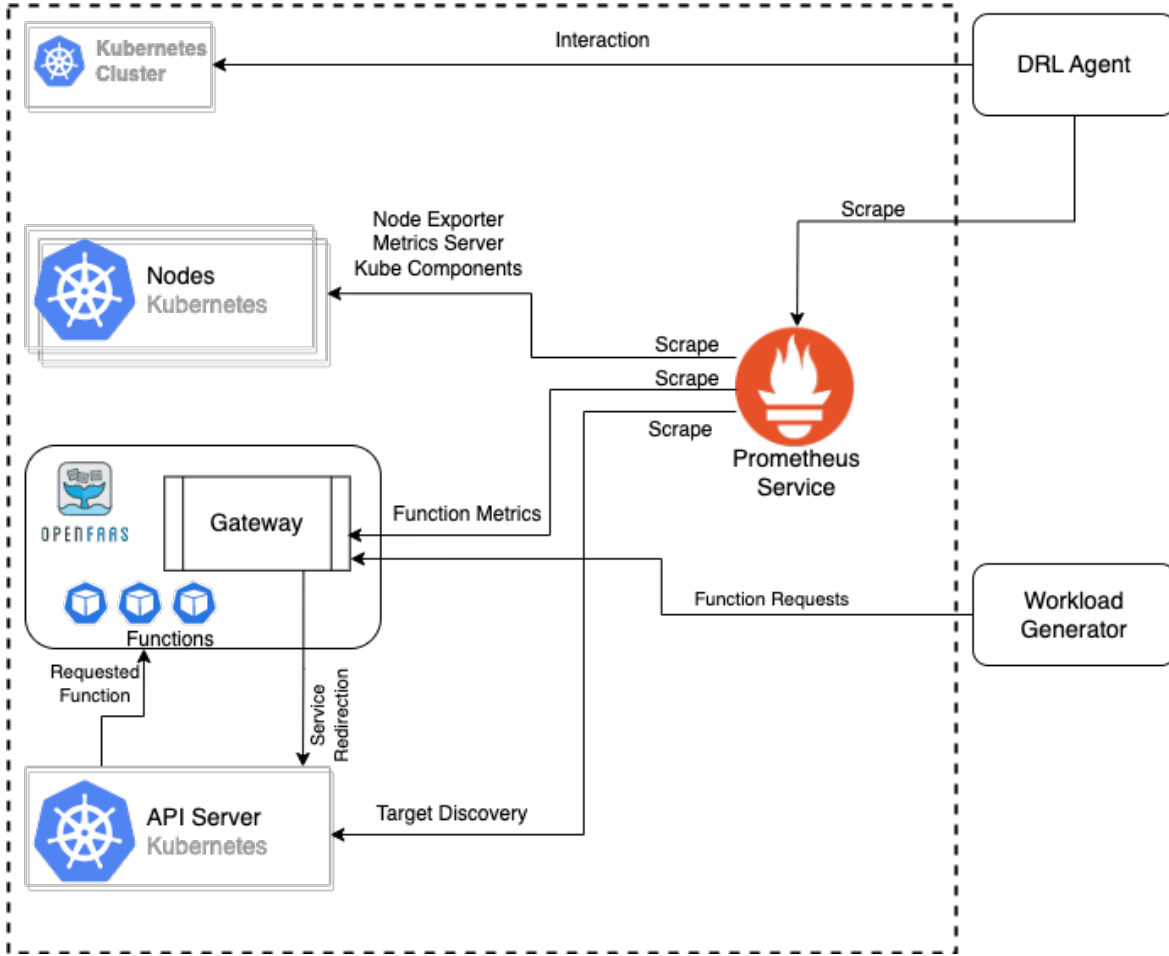
1. https://github.com/Cloudslab/DRe-SCale

Fig. 1: System Architecture

based on the analyzed observation. Autoscaling is a sequential process with non-deterministic results in a partially observable environment that is conditioned on historical interactions, therefore, we design FaaS autoscaling as a *model-free POMDP*. POMDPs are a mathematical model and an extension of Markov Decision Processes (MDP) that account for uncertainty while maximising a given objective.

3.2.1 Model-Free POMDP

In a real-world scenario, it is hard to perceive the complete state of the surrounding environment and a MDP rarely holds true [19]. Instead, a POMDP better encapsulates environmental characteristics from incomplete or partial information about said environment. Formally, a POMDP model is defined as a 6-tuple (S, A, O, T, Z, R) where: S denotes the set of all possible environment states, A denotes the set of all actions, O denotes the set of all observations that an agent can perceive, T and Z represent the transition probability function and observation probability function, respectively, and R denotes the reward function. Conceptually, the agent observes itself in some environment state s_t, hidden due to partial observability at each sampling interval t and maintains a belief b_t, an estimate of its current state, to select an action a_t and transition to a new state $\hat{s}_t$. The agent perceives the state information through observation o_t and utilises the transition and observation probability function to update the state estimates. After transitioning

to a new state $\hat{s}_t$, the agent receives reward r_t that helps in maximising the objective.

Since probability functions are difficult to model in complex FaaS environments and states cannot be perfectly represented to capture the estimates of belief or hidden states [20], we define the autoscaling problem as model-free POMDP. Model-free POMDP attempts to maximise the cumulative reward without explicitly modelling the transition or observation probabilities. Further, it needs function approximation techniques like neural networks, specifically recurrent neural networks (RNN), to capture the uncertainty and temporal dependency. Therefore, we define the POMDP observations as a tuple of (O, A, R) and utilise recurrency to model and infer transition probabilities, observation probabilities and hidden states to fulfil the conflicting objectives of resource utilisation, operational cost and QoS objectives.

3.2.2 Deep Recurrent-Reinforcement Learning

A possible solution to learning effective policies in a model-free POMDP is the application of model-free RL algorithms. Here, the agent directly interacts with the environment and does not explicitly model the transition or observation probabilities. Vanilla RL algorithms like Q-learning and DQN have no mechanism to determine underlying state [20] and speculates that fed observation is a complete representation of the environment. To capture sequential or temporal dependencies, often recurrent units are integrated with vanilla RL approaches, known as Recurrent Reinforcement

TABLE 2: Notations

Symbol	Definition
f_l	Function for training {matmul}.
S	State space for POMDP agent.
A	Action space for POMDP agent.
O	Observation space for POMDP agent.
T, Z	Transition and Observation probability functions.
R	Reward function for POMDP agent.
N	Maximum number of function replicas possible (function quota).
n_{min}	Minimum number of function replicas.
Q	Maximum requests possible in a sampling window.
t	sampling window.
n_t	All available functions during t.
τ_t	Average execution time of n_t functions.
c	Average CPU utilisation of n_t functions.
c_{max}	Maximum CPU utilisation of a function.
m_t	Average memory utilisation of n_t functions.
m_{max}	Maximum memory utilisation of a function.
ϕ_t	Throughput of function.
q_t	Requests during t.
k	Scaling limits.
s_t	Environment state at t.
b_t	Belief state for POMDP agent.
o_t	Environment observation tuple $(\tau_t, \phi_t, q_t, n_t, c_t, m_t) \in O$ at t.
a_t	Agent action $\in \{-k, \dots, +k\}$ at t.
r_t	Reward for action $a_t \in R$ at t.
r_{min}	Negative immediate reward (-100).
α, β, γ	Objective weight parameters.

Learning (RRL) [36]. Prior studies [20], [31] [32], [37] [38], have introduced and applied RRL approaches to a variety of application domains such as T-maze task, financial trading, network resource allocation and Atari games, to address sequential nature and partial observability of environment, i.e., a non-Markovian or POMDP setting. In RRL, an agent follows the basic principle of performing an action in the environment, establishing its state and receiving feedback to improve the policy, but, additionally employs RNN units/cells to model uncertainty. Theoretically, POMDP has an underlying dynamics of MDP with an additional constraint of state uncertainty or observability that makes the process non-Markovian. Therefore, we define the core RL components as observation O, action A, reward R (guiding signals) and FaaS environment.

We model the observation space as $o_t = (\tau_t, \phi_t, q_t, n_t, c_t, m_t) \in O$ where τ_t is average execution time of n_t available function replicas with c_t average CPU and m_t average memory utilisation, while successfully serving ϕ_t proportion of q_t requests in the sampling window t. The agent adjusts the number of function instances in the upcoming sampling window $t + 1$ using suitable actions in an attempt to maximise the reward. Therefore, we define scaling action a_t as the number of function instances, k, to add or remove and represent it as $a_t \in A = \{-k, \dots + k\}$ such that $n_{min} \le (n_{t-1} + a_t) \le N$, where N is function quota. This estimate helps the agent to control the degree of exploration by maintaining replication within quota N.

The objective of the DRL agent is to learn an optimal scaling policy, and therefore, we structure the rewards $r_t \in R$ over monitored metrics - c_t average CPU utilisation, m_t average memory utilisation, ϕ_t successful proportion of total requests and number of available function replicas n_t. Our proposed agent does not work towards achieving a

specific threshold. Instead, it learns to maximise the returns, i.e., improve resource utilisation, throughput and economically scaling function replicas. After performing an action a_t, the agent receives a delayed reward r_t at every sampling window t and updates its network parameters.

RL application for model-free POMDP does not explicitly estimate the probabilities, instead, RNNs are incorporated to analyse environment uncertainties and model time-varying patterns [36] [38]. The structure of RNNs is made-up of highly-dimensional hidden states that act as network memory and enables it to remember complex sequential data. These networks map an input sequence to output and consist of three units - input, recurrent and output unit, serving towards memory goal.

4 LSTM-PPO BASED AUTOSCALING APPROACH

As discussed in section 3.2, we introduce recurrency to handle system dynamics, complex workloads, and hidden correlation of components based on POMDP model in autoscaling tasks. We select Proximal Policy Optimisation (PPO), a popular state-of-the-art on-policy RL algorithm for autoscaling agents. While model-free off-policy algorithms such as Deep Q-Network (DQN), Deep Deterministic Policy Gradient (DDPG) have been studied with recurrent units [20] [32], we explore a model-free on-policy PPO in our setting due to its ease of implementation, greater stability during learning, better performance across different environments [39] and support for discrete actions while providing better convergence [21]. Although on-policy methods are known to be sample inefficient and computationally expensive, our agent continuously collects samples for timely policy updates. Also, off-policy algorithms tend to be harder to tune than on-policy because of significant bias from old data and Schulman at el. [40] suggests that PPO is less sensitive to hyperparameters than other algorithms. PPO has found its application in domains like robotics, finance and autonomous vehicles, and takes advantage of the Actor-Critic method to learn optimal policy estimations. However, for partial observability or temporal dependence, general RL algorithms struggle to capture underlying correlations and patterns effectively. Therefore, we utilise RNN units, specifically LSTM, to address partial observability in the FaaS environment and improve the agent's decision-making capabilities. This integration is expected to enhance PPO's ability to capture historical data and make informed decisions while improving its policy via new and previous experiences.

The core component of the proposed autoscaling solution is the integration of recurrent units with a fully-connected multi-layer perceptron (MLP) that takes into environment observation and maintains a hidden internal state to retain relevant information. The LSTM layer is incorporated into both actor and critic networks to retain information i.e., the output of the LSTM layer is fed into fully-connected MLP layers, where the actor (policy network) is responsible for learning an action selection policy and the critic network serves as a guiding measure to improve actor's decision. The network parameters are updated as per PPO clipped surrogate objective function [41] (Eq. 1) which helps the agent balance its degree of exploration
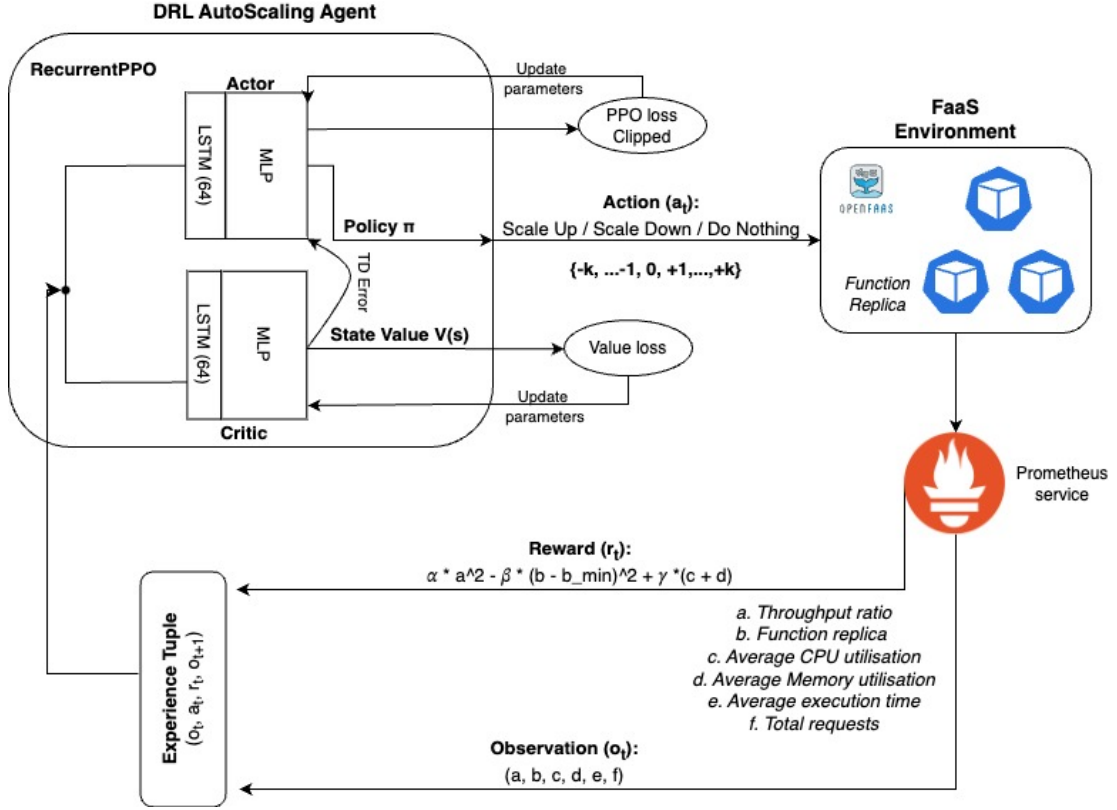
Fig. 2: DRL agent structure for Autoscaling

and knowledge exploitation. It further improves network sample efficiency and conserves large policy updates.

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_{\approx} \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}\left(\hat{r}_t(\theta), 1 - \epsilon, 1 + \epsilon\right)\hat{A}_t \right) \right] \tag{1}$$

$$\hat{r}_t(\theta) = \frac{\pi_\theta(a_t|o_t)}{\pi_{\theta_{\text{old}}}(a_t|_t)} \tag{2}$$

$$r_t = \begin{cases} \alpha.\phi_t^2 - \beta.(n_t - n_{min})^2 + \gamma.(c_t + m_t) & ; 1 \le a_t + n_{t-1} \\ r_{min} & ; \text{otherwise} \end{cases} \tag{3}$$

The proposed autoscaling technique has two phases: an agent training phase and a testing phase. Fig. 2 demonstrates the agent training workflow. The environment setup process precedes the agent training, where the agent interacts with the environment and obtains information. After initial setup, the agent is trained for multiple episodes of sampling windows, where it assesses the function demand q_t over individual sampling window t and ascertains appropriate scaling action. During a sampling window t, the agent collects the environment observation o_t and samples an action a_t according to LSTM-PPO policy. If the agent performs an invalid action, it is awarded an immediate negative reward r_{min}, else the agent obtains a delayed reward r_t (Eq. 3), for sampling window t, calculated using the relevant monitored metrics (3.2). This reward helps the agent in action quality assessment, transition to a new state and has significant effects on the function's performance. These rewards are essential for improving the agent's decision-making capability. The critic network estimates the agent state and helps update the network parameters. The agent

continues to analyse the demand over multiple sampling windows, repeating the interaction process and accumulating the relevant information in recurrent cells for learning. Once the agent is trained for sufficient episodes and rewards appear to converge, we evaluate the agent in the testing phase.

In the testing phase, the agent is evaluated for its learnt policies. It collects current environment observation, samples the action through actor policy and scales the functions accordingly. We hypothesised the relationship between QoS and resource utilisation and deduce that appropriately scaling the functions improve throughput, resource utilisation and reduce operational costs (number of function replicas used).

5 PERFORMANCE EVALUATION

In this section, we provide the experimental setup and parameters, and perform an analysis of our agent compared to other complementary solutions.

5.1 System Setup

We set up our experimental multi-node cluster, as discussed in Section 3, using NeCTAR (Australian National Research Cloud Infrastructure) services on the Melbourne Research Cloud. It includes a combination of 2 nodes with 12/48, 1 node with 16/64, 1 node with 8/32 and 1 node with 4/16 vCPU/GB-RAM configurations. We deploy OpenFaaS along with Prometheus service on MicroK8s (v1.27.2), however, we used Gateway v0.26.3 due to scaling limitations in the latest version and remove its alert manager component

TABLE 3: Parameters for System Setup

Parameter Name	Value
MicroK8s version	v1.27.2
OpenFaaS Gateway version	v0.26.3
Nodes	5
OS	Ubuntu 18.04 LTS
vCPU	4,8,12,16
RAM	16,32,64,48 GB
Workload	Matrix Multiplication ($m \times m$)
m	10(small), 100(medium), 1000(large)
CPU, memory, timeout	150 millicore, 256 MB, 10 seconds

to disable rps-based scaling. The system setup parameters are listed in Table 3.

As FaaS is beneficial for short-running, single-purpose functions that require few resources, we consider *matrix multiplication* function with three different input sizes $small, medium, large$-$(10, 100, 1000)$ and configure it with 150/256 millicore/MB resources approximately as AWS Lambda offering and a maximum timeout of 10 seconds. Additionally, we generate the user workload using the Hey [42] load generator tool, a lightweight load generator written in Go language. For the workload we leverage an open-sourced, 14-day function trace [23] by Azure functions, Fig. 3, that largely represents an invocation behaviour of a production-ready application function running on a serverless platform. Although it appears stationary due to its repetitive nature, it is representative of real cloud invocation patterns with relevant variations for scaling decisions. Since the Poisson distribution has been shown to approximately sample online user behaviour, request inter-arrival times are sampled from it. Prometheus service is configured with relevant discovery and target points to regularly scrape metrics from OpenFaaS gateway, function instances and Kubernetes API server.

As discussed in Sec. 4, the agent assesses the function demand during a sampling window of 30 seconds for a single episode of 5 minutes. Based on the deployed infrastructure capacity, we fix the maximum function instances as 24 in isolation, to reduce the performance interference. Since frequent scaling can result in resource thrashing, we explore scaling actions within a range of 2 instances, i.e., $a_t \in \{-2, -1, 0, 1, 2\}$, avoiding resource wastage during acquisition and release of function instances. Further, the observation space is composed of the throughput ratio $\phi_t \in [0, 100]\%$, number of function instances $n_t \in [1, 24]$ and resource utilisation (CPU, c_t and memory, m_t) $\in [0, 2]*100\%$ that contributes towards over-burdened CPU and out-of-memory scenarios. The LSTM-based PPO agent takes advantage of a single LSTM layer of 256 units and is integrated with both Actor and Critic networks with identical network architectures having 2 fully connected MLP layers of 64 neurons each, i.e., in[64,64] and out[64,64].

5.2 Experiments

Function autoscaling is a continuous and non-episodic process, however, we set an episode based on the default scaling window of 5 minutes by Kubernetes' horizontal scaling mechanism. To demonstrate the effectiveness of recurrency in autoscaling tasks, we chose a workload with varying resource requirements at different sampling windows. After careful consideration of network parameters and sensitivity analysis, listed in Table 4, the DRL agent is trained for
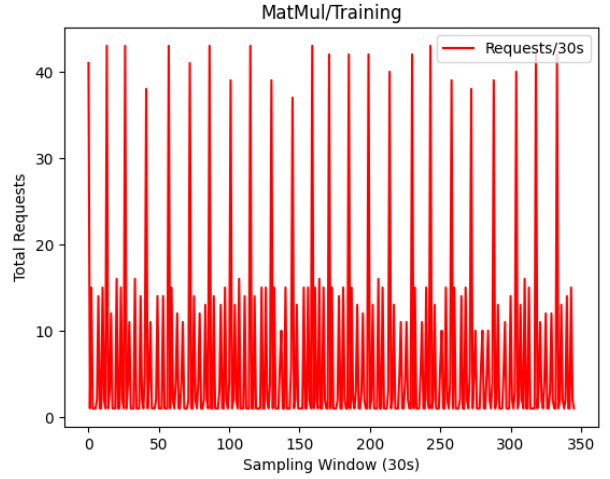

Fig. 3: Workload for Matrix Multiplication function

TABLE 4: RL Environment and Network Parameters

Parameter	Value
N	24
t	30 seconds
τ_t	(0 - 10) seconds
ϕ_t	(0 - 100) %
q_t	{0,…,Q} requests
n_t	{1,…,N} functions
c_t	(0 - 2) *100%
m_t	(0 - 2) *100%
a_t	{-2, -1, 0, +1, +2 }
LSTM Network	Layer 1(256 cells)
Actor Network	Layer 1(64 cells), Layer 2(64 cells)
Critic Network	Layer 1(64 cells), Layer 2(64 cells)

more than 500 episodes to determine a scaling policy to maximise the throughput while using minimal resources. The agent is expected to retain workload information and perform in accordance with the received feedback. Further, the agent is evaluated against a state-of-the-art, PPO-based autoscaling agent, with the same the Actor/Critic network architecture, (Table 4) as the RPPO agent, i.e., having 2 MLP layers with 64 neurons each. In addition to it, we evaluate a DRQN agent that integrates a LSTM layer (256 cells) with regular off-policy Deep Q-Network (DQN), and has 2 MLP layers with 128 neurons, each for target network and q-network. Fig. 4 shows the training results of these competing approaches in terms of mean episodic rewards. The rewards are given as per Eq. 3, and it is evident that the mean episodic reward for PPO (60190) begins to diminish after 400 episodes as compared to LSTM-PPO(RPPO) (60540) agent. Additionally, a similar pattern is visible for the throughput of RPPO and PPO approaches, Fig. 4(b) where PPO struggles to keep a higher success rate by provisioning more functions. Also, we observe that mean episodic reward for the DRQN (59564) approaches that of RPPO while exploring the search space and gradually serves more workload successfully, Fig. 4(e), but closely tailing the trend of other approaches, Fig. 4(b). As mentioned in section 5, matrix multiplication is performed for three input sizes - $small, medium, large$ and similar input randomness is followed for competing approaches that are evident in execution time (3.7 and 4 seconds) of successful requests in Fig. 4(c), (d) and (e).

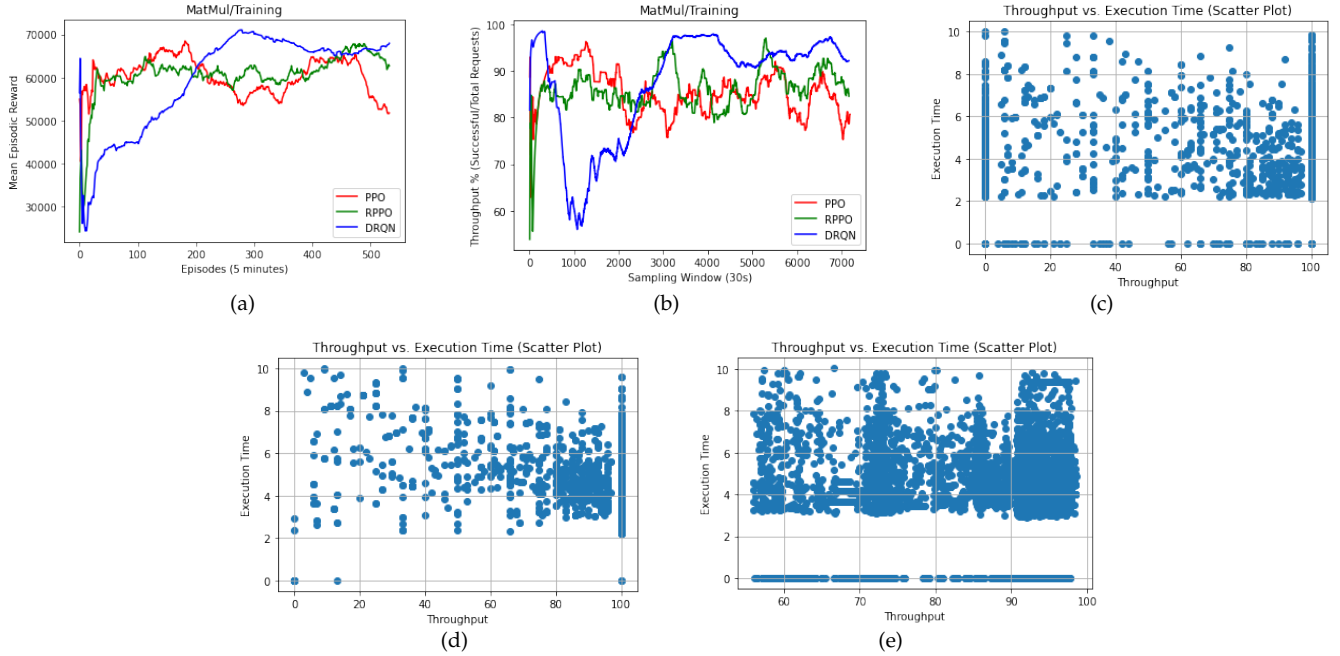We evaluate the agents for 200 sampling windows and

Fig. 4: (a) Mean episodic reward, (b) Throughput, (c) Throughput vs Execution time - PPO, (d) Throughput vs Execution time - RPPO and (e) Throughput vs Execution time - DRQN

present the results in Fig. 5. Out of the 200 sampling windows, RPPO based autoscaling agent performed 18% better in terms of throughput, while having an average of 85% mean success ratio as compared to 67% of the PPO agent. On the other hand, the DRQN agent fell short to serve 22% of the workload with a mean success rate of 66% as compared to RPPO agent. In serving the evaluated workload, the RPPO agent utilised at least 8.4% more resources than the PPO agent and improved average execution time (seconds) by 13%, while it utilised at least 8% more resources than DRQN and slightly improved the average execution time (seconds) by 2.6%. Although the DRQN agent tries to capture sequential dependency of the workload, we suspect it fails to explore the search space and only exploits minimal replica count. Hence, as evident in Fig. 5(d), the DRQN agent keeps utilising lesser function resources. This agent behaviour is in-line with training results where it could serve better with less requests, thus receiving higher reward for that sampling window and eventually, accumulating higher episodic reward and throughput percentage.

We also assess the effectiveness of our approach against a default commercial scaling policy, CPU threshold-based horizontal scaling. Kubernetes-based serverless platforms like OpenFaaS [3] and Kubeless [4] can leverage underlying resource-based scaling, known as horizontal pod autoscaling (HPA) implemented as a control loop that checks for target metrics to adjust the function replicas. HPA has a pre-configured query period of 15 seconds to control deployment based on target metrics like average CPU utilization. Therefore, the HPA controller fetches the specific metrics from the underlying API and empirically calculates the number of desired functions. However, the controller is unaware of workload demand and only scales after a 15-second metric collection window. The expected threshold for function average CPU utilisation is set to be 75% with maximum scaling up to 24 instances. Therefore, whenever

the average CPU utilisation of a function exceeds the fixed threshold, new function instances are provisioned. Also, HPA has a 5-minute down-scaling window during which resources are bound to function irrespective of incoming demand, representing potential resource wastage. Similarly, we compare our scaling methods with another metric-based autoscaling supported by OpenFaaS based on request-per-second processed. It is also implemented as a control loop and watches for processed requests per second (rps) and raises an alert if rps is above 5 for 10 seconds (default). Therefore, it is worthwhile to analyse the performance of the DRL-based agent against HPA that reserves enough resources for either idle time or low resource utilisation.

The results for both threshold-based scaling are presented in Fig. 6, and both approaches struggle to keep up with the incoming workload. The rps could only manage to serve 50% of incoming load at any sampling window while only using a single instance. This happens as a single request takes approximately 4 seconds to process, and rps never goes beyond the set threshold, failing the majority of requests. On the other hand, HPA could serve 80% of incoming load on average, but fluctuates due to its set cooldown period. Although HPA tries to scale its resources to 5 replicas, its performance is degraded by 35% against RPPO and similarly, rps degrades throughput performance by 58%.

5.3 Discussion

Autoscaling is an essential feature of cloud computing and has been identified as a potential research gap in serverless computing models. As compared to service-oriented architectures where the services are always running, FaaS functions run for shorter duration and release resources, if unwanted. Hence, an adaptive scaling solution is critical in handling complex workloads for these small and ephemeral functions. Thus, we investigate a DRL-based autoscaling
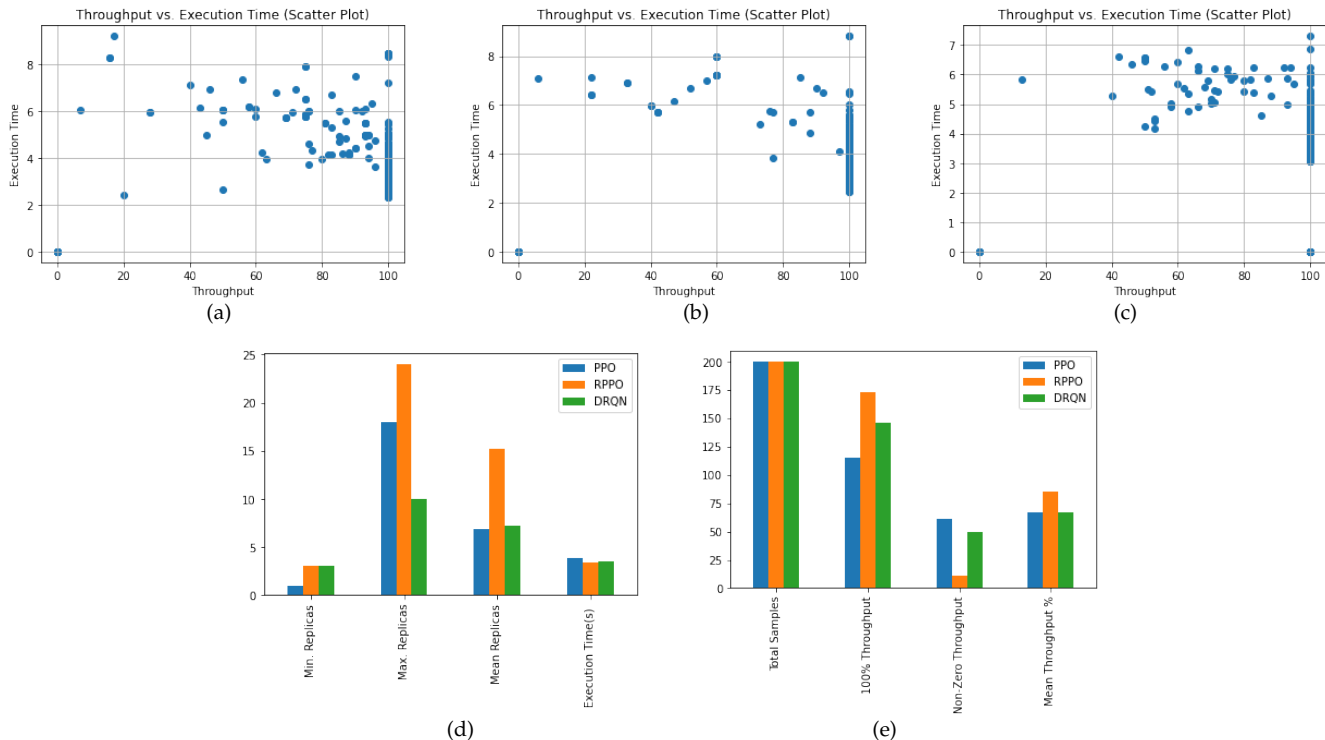
Fig. 5: (a) Throughput vs Execution time - PPO, (b) Throughput vs Execution time - RPPO, (c) Throughput vs Execution time - DRQN, (d) Function Replicas and Execution time, and (e) Throughput Comparison
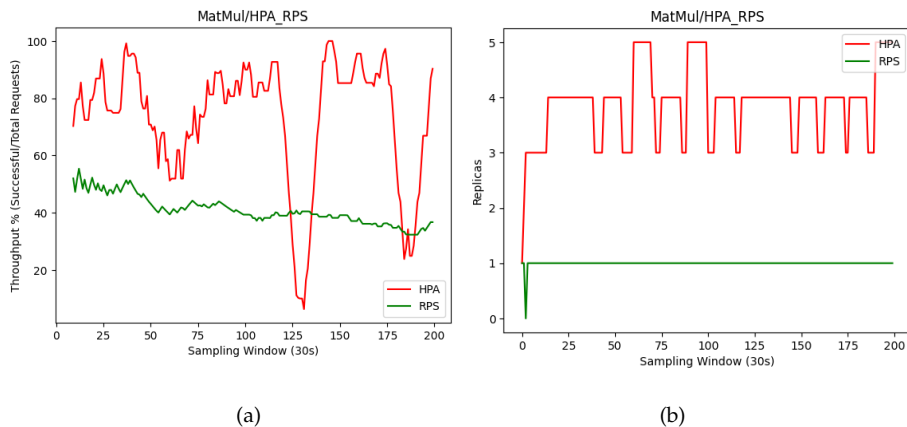

Fig. 6: (a) Throughput - HPA vs RPS, (b) Replica use - HPA vs RPS

agent, LSTM-PPO, to work in complex FaaS settings and utilise relevant environmental information to learn optimal scaling policies. We train and evaluate the proposed solution against a state-of-the-art on-policy PPO approach, alongside commercial default, and infer that LSTM-PPO is able to capture environment dynamics better and shows promising results. Although, we argue that real-time systems are hard to model and transparent to a certain degree and that RL approaches can analyse these uncertainties better. There are certain points to remember associated with the appropriateness and application of RL methods to real systems.

We model function autoscaling in FaaS as a model-free POMDP and leverage monitoring tools, like Prometheus, to collect the function-related metrics and apply model-free RL methods to learn the scaling policy. In general, RL algorithms are expensive in terms of data, resource and time, where an agent interacts with the modelled environment

and acquire relevant information over multiple episodes that signify a higher degree of exploration. Although, as showcased through results, the proposed RL approach took more than 500 episodes (>6000 sampling windows) to slightly improve the performance over baselines, RL methods in real-time systems are considerably expensive following stringent optimization goals.

The current training time has an episode of 5 minutes that consists of 10 iteration windows or epochs of 30 seconds each, where a decision to scale is taken by the agent and feedback is calculated for learning. This duration of an episode is chosen keeping in mind the minimum resource scaling and cooldown time of Kubernetes-based serverless platforms and an industry insight [11] for taking scaling actions in production environments. In addition to these settings, an agent training could further be affected by the invocation pattern and set of actions to be explored. In

the current work, the agent explores 5 discrete actions that follow a conservative approach to avoid resource thrashing while scaling function instances. In a particular state, an agent could take all the possible actions from the action space and would be penalised for an infeasible action. This static behaviour of action modelling elongates the training process since the agent explores infeasible actions in a state and only learns from negative experience. To overcome this, an action masking technique could be integrated that prevents the agent to take certain infeasible actions in particular states, based on defined rules like the total number of function instances to remain within function limits. Therefore, different functions do not necessarily show similar behaviour for training and realised quality of results under similar settings.

The proposed DRL method is a composition of two different neural network techniques, recurrent and fully-connected layers, and these models are known to be sensitive to respective hyper-parameters or application/workload context. Therefore, configuring hyper-parameters can also be an intensive task in real-world settings. Additionally, the proposed agent analyzes individual workload demand for a particular function, the learning cannot be generalized to other functions with different resource requirement profiles and therefore requires individual training models to be commissioned. However, techniques like transfer learning or categorising functions with similar resource and workload profile to use a trained agent as a starting point could be explored. Moreover, these agents could be deployed in similar fashion to tools like AWS Compute Optimiser [43] to gradually obtain experiences and build models with high confidence, from real-time data before making any recommendation/autonomous decision.

Furthermore, the agent is trained for a limited number of episodes, approximately 500 episodes and evaluated, but the chances of exploring are limited. Therefore, the agent expects to be guided by its actor-critic network policies in making informed decisions. Additionally, the agent utilizes resource-based metrics that affect the cold starts, so the availability of relevant tools and techniques to collect instantaneous metrics is essential [11] in reducing the observation uncertainty. Also, the respective platform implementation, such as metric collection frequency, function concurrency policy, and request queuing, can extend support to the analyses. Hence, based on performance evaluation results and discussion, we can adequately conclude that the proposed LSTM-PPO agent successfully performs at par with competing policies for given workload and experimental settings.

6 Conclusions and Future Work

The FaaS model executes the piece of code inside a container, known as a function and prepares new function containers on demand. FaaS platforms usually support threshold-based autoscaling mechanisms like CPU utilisation to cope with incoming demand and heuristically create more functions. These methods do not consider any system complexity or workload characteristics for scaling and therefore result in sub-optimal scaling policies. Therefore, an adaptive autoscaling mechanism is required to analyse the workload and system uncertainty to optimally scale resources while improving system throughput.

In this work, we investigated a recurrent RL approach for function autoscaling and presented results against a state-of-the-art PPO algorithm and commercially applied threshold-based autoscaling. We perform our analyses for matrix multiplication function and utilise an open-source function trace by Azure [23]. The experimental multi-node cluster was set up on the MicroK8s distribution and took advantage of the OpenFaaS serverless framework. We presented evidence of modelling real-time FaaS environments as partially observable and application of recurrent networks to model-free RL algorithms to maximise the objective. We evaluate our proposed technique after training of more than 500 episodes and successfully validate our hypothesis that recurrent techniques capture the system dynamicity and uncertainty to give better autoscaling policies. In our evaluation setting, experiments show that RPPO improved system throughput by 18%, 22%, 35% and 58% in comparison to PPO, DRQN, HPA and rps scaling policy, respectively.

As part of future work, we will extend our analysis to different functions and workload types to examine the effect of POMDP modelling. We further plan to experiment with other on-policy and off-policy RL methods like TD3, to expedite the learning process due to their sample efficiency. The proposed methods are dependent on the metric collection process for observing system states which can act as bottlenecks and single points of failure [11]. Therefore, we plan to investigate distributed metric collection and agent learning to avoid single-point-failure and improve learning and sample efficiency for estimating optimal function autoscaling policies.

Software Availability: Our environment setup code and algorithms we implemented for OpenFaaS can be accessed from: https://github.com/Cloudslab/DRe-SCale

References

[1] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–33, 2018.

[2] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[3] OpenFaaS, "Openfaas-serverless function, made simple," 2016. last accessed on 09/09/2022.

[4] Kubeless, "Kubeless-kubernetes native serverless," 2019. last accessed on 12/12/2021.

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

[6] J. Manner and G. Wirtz, "Resource scaling strategies for open-source faas platforms compared to commercial cloud offerings," in *Proceedings of the 15th IEEE International Conference on Cloud Computing (CLOUD)*, pp. 40–48, IEEE, 2022.

[7] AWS, "Lambda function scaling - aws lambda," 2023. last accessed on 01/02/2023.

[8] G. C. Functions, "Configure minimum instances - google cloud functions," 2023. last accessed on 01/02/2023.

[9] Y. Garí, D. A. Monge, E. Pacini, C. Mateos, and C. G. Garino, "Reinforcement learning-based application autoscaling in the cloud: A survey," *Engineering Applications of Artificial Intelligence*, vol. 102, p. 104288, 2021.

[10] S. N. A. Jawaddi and A. Ismail, "Autoscaling in serverless computing: Taxonomy and openchallenges." pre-print on webpage https://doi.org/10.21203/rs.3.rs-2897886/v1, 2023.

[11] M. Straesser, J. Grohmann, J. von Kistowski, S. Eismann, A. Bauer, and S. Kounev, "Why is it not solved yet? challenges for production-ready autoscaling," in *Proceedings of the ACM/SPEC on International Conference on Performance Engineering*, pp. 105–115, 2022.

[12] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, "Horizontal pod autoscaling in kubernetes for elastic container orchestration," *Sensors*, vol. 20, no. 16, p. 4621, 2020.

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

[14] M. Xu, C. Song, S. Ilager, S. S. Gill, J. Zhao, K. Ye, and C. Xu, "Coscal: Multifaceted scaling of microservices with reinforcement learning," *IEEE Transactions on Network and Service Management*, vol. 19, no. 4, pp. 3995–4009, 2022.

[15] P. Benedetti, M. Femminella, G. Reali, and K. Steenhaut, "Reinforcement learning applicability for resource-based auto-scaling in serverless edge applications," in *Proceedings of the IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pp. 674–679, IEEE, 2022.

[16] A. Zafeiropoulos, E. Fotopoulou, N. Filinis, and S. Papavassiliou, "Reinforcement learning-assisted autoscaling mechanisms for serverless computing platforms," *Simulation Modelling Practice and Theory*, vol. 116, p. 102461, 2022.

[17] S. Agarwal, M. A. Rodriguez, and R. Buyya, "A reinforcement learning approach to reduce serverless function cold start frequency," in *Proceedings of the 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 797–803, IEEE, 2021.

[18] L. Schuler, S. Jamil, and N. Kühl, "Ai-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments," in *Proceedings of the 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 804–811, IEEE, 2021.

[19] T. Ni, B. Eysenbach, and R. Salakhutdinov, "Recurrent model-free rl can be a strong baseline for many pomdps," *arXiv preprint arXiv:2110.05038*, 2021.

[20] M. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," in *Proceedings of the aaai fall symposium series*, 2015.

[21] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021.

[22] M. Towers, J. K. Terry, A. Kwiatkowski, J. U. Balis, G. d. Cola, T. Deleu, M. Goulão, A. Kallinteris, A. KG, M. Krimmel, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, A. T. J. Shen, and O. G. Younis, "Gymnasium," Mar. 2023.

[23] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 205–218, 2020.

[24] H.-D. Phung and Y. Kim, "A prediction based autoscaling in serverless computing," in *Proceedings of the 13th International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 763–766, IEEE, 2022.

[25] Z. Zhang, T. Wang, A. Li, and W. Zhang, "Adaptive auto-scaling of delay-sensitive serverless services with reinforcement learning," in *Proceedings of the 46th IEEE Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 866–871, IEEE, 2022.

[26] X. Li, P. Kang, J. Molone, W. Wang, and P. Lama, "Kneescale: Efficient resource scaling for serverless computing at the edge," in *Proceedings of the 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 180–189, IEEE, 2022.

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

[28] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," *arXiv preprint arXiv:1812.03651*, 2018.

[29] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges, and applications," *ACM Computing Surveys (CSUR)*, 2019.

[30] E. Rosenbaum, "Next frontier in microsoft, google, amazon cloud battle is over a world without code." https://www.cnbc.com/2020/04/01/new-microsoft-google-amazon-cloud-battle-over-world-without-code.html, April, 2020. last accessed on 01/09/2021.

[31] C. Gold, "Fx trading via recurrent reinforcement learning," in *Proceedings of the IEEE International Conference on Computational Intelligence for Financial Engineering, 2003. Proceedings.*, pp. 363–370, IEEE, 2003.

[32] P. Hu, L. Pan, Y. Chen, Z. Fang, and L. Huang, "Effective multi-user delay-constrained scheduling with deep recurrent reinforcement learning," in *Proceedings of the 23rd International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing*, pp. 1–10, 2022.

[33] MicroK8s, "Microk8s-the lightweight kubernetes," 2023. last accessed on 01/03/2023.

[34] PyTorch, "Recurrent dqn: Training recurrent policies," 2023. last accessed on 30/10/2023.

[35] A. Juliani, "Deeprl-agents: A set of deep reinforcement learning agents implemented in tensorflow.." https://github.com/awjuliani/DeepRL-Agents/tree/master, 2023. last accessed on 30/10/2023.

[36] X. Li, L. Li, J. Gao, X. He, J. Chen, L. Deng, and J. He, "Recurrent reinforcement learning: a hybrid approach," *arXiv preprint arXiv:1509.03044*, 2015.

[37] B. Bakker, "Reinforcement learning with long short-term memory," *Advances in neural information processing systems*, vol. 14, 2001.

[38] Q. Cong and W. Lang, "Double deep recurrent reinforcement learning for centralized dynamic multichannel access," *Wireless Communications and Mobile Computing*, vol. 2021, pp. 1–10, 2021.

[39] R. Kozlica, S. Wegenkittl, and S. Hiränder, "Deep q-learning versus proximal policy optimization: Performance comparison in a material sorting task," in *Proceednigs of the 32nd International Symposium on Industrial Electronics (ISIE)*, pp. 1–6, IEEE, 2023.

[40] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017.

[41] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[42] J. Dogan, "Hey: Http load generator, apachebench (ab) replacement." https://github.com/rakyll/hey, 2023. last accessed on 01/02/2023.

[43] AWS, "Aws compute optimizer." https://aws.amazon.com/compute-optimizer/, 2023. last accessed on 30/10/2023.