SCARLET: Serverless Container Autoscaling with Reinforcement Learning Environments

Alan Song* Nikita Lazarev† Varun Gohil† Yueying Li†

***Abstract*—Serverless computing is a paradigm of cloud computing that allows users to avoid challenging server management and overprovisioning of resources. In the serverless model, users submit functions to cloud providers (e.g. Google or Amazon), who deploy and execute instances of these workloads in short-lived containers before returning the output to the user. Cloud providers are thus responsible for managing computing resources such that (1) user-provider agreements on quality of service objectives are met, and (2) resources (i.e. containers) are neither over- nor under-provisioned. Current serverless systems in production address resource management with naive autoscalers that provide heuristic solutions at best. Recent research has shown that using reinforcement learning (RL) for serverless resource management is promising; however, the implementation of RL-based autoscalers in production-grade environments like Kubernetes and the evaluation of these autoscalers using realistic serverless benchmarks have been limited. We present SCARLET, a framework for RL-based autoscaling in Kubernetes clusters. In our design, users only need to provide standard Kubernetes YAML manifests and service-level agreement (SLA) configurations for each function. SCARLET also allows developers to experiment with any RL agent implemented with adherence to the standard OpenAI Gym API. Finally, we use SCARLET to implement a Deep Q-Learning model. Our evaluation demonstrates that, through implementation via SCARLET, the model satisfies quality-of-service constraints for multiple functions running concurrently.**

1 Introduction

Motivation. Serverless computing is an emerging paradigm of cloud computing that offers two key advantages over the standard cloud execution model [2, 16]. First, the responsibilities associated with server management are delegated from the user to the cloud provider, leaving the user free to develop software without needing to deal with complex infrastructure management. Second, serverless offers a pay-per-use payment scheme where users are charged per function invocation rather than per machine, allowing users to pay only for applications that are actually running, rather than for the uptime of machines that may be idle. Large commercial cloud providers like Amazon, Microsoft, and Google are already offering FaaS platforms such as AWS Lambda, Azure Functions, and Google Cloud Functions, respectively.

Challenges. Despite its advantages, serverless comes with certain challenges and overheads. At present, one of the greatest challenges faced by serverless is fine-grained resource management to accommodate varying and unpredictable workloads, as cloud providers aim to allocate more containers to heavier workloads and to tear them down when they are no longer needed [22]. Current commercial and open-source serverless systems [8, 13, 21] use heuristic autoscalers to allocate resources; however, these automata require the manual tuning of myriad parameters and configurations, which is both complex and inexact. Recently, the use of reinforcement learning (RL) for resource management has become an active area in systems research. Unlike static rule-based autoscalers, an RL-based autoscaler dynamically adapts its policy through repeated interaction with the environment, allowing it to learn optimal scaling configurations for unfamiliar workloads. Still, the challenge of implementing and evaluating RL-based scalers in production-grade serverless environments remains.

Our work. We present SCARLET, a framework for RL-based resource management via horizontal scaling in Kubernetes-based environments. SCARLET's design incorporates API layers between the RL learner and the serverless environment to translate the RL learner's decisions into scaling actions in the environment, while also collecting real-time resource utilization and performance metrics that are used to train the learner online. SCARLET allows for running concurrent workloads and allows RL pipeline developers to easily evaluate other RL algorithms for resource management in the environment. The serverless environment is implemented as a cluster deployment of Kubernetes. The API layers are implemented using Python code and use the Kubernetes API [6], Prometheus metrics monitoring [11], and vSwarm's Invoker [14] to interact with and sample metrics from the environment. To evaluate SCARLET, we use PyTorch [12] to implement an RL agent using the Deep Q-Learning algorithm with experience replay, and we run vSwarm's open-source benchmarks concurrently and set quality-of-

*Wellesley High School
†Massachusetts Institute of Technology

service constraints for each one. Our experimental results show that the model implemented in SCARLET meets quality-of-service constraints for all concurrently running serverless benchmarks.

Contributions. In summary, our main contributions are:

- SCARLET, a production-grade framework for RL-based resource management in Kubernetes clusters.
- An implementation of SCARLET using a Deep Q-Learning model.
- An evaluation of the Deep Q-Learning model using serverless benchmarks that demonstrates model convergence and quality-of-service satisfaction when implemented via SCARLET.

Our work is organized as follows. In Section 2, we provide background on serverless computing, quality of service, and reinforcement learning, as well as an overview of related work. Section 3 describes our proposed design of SCARLET. In Section 4, we present details for SCARLET's implementation. Section 5 describes our evaluation methodology and experimental results. Section 6 discusses limitations and potential future work. Finally, we conclude this paper in Section 7.

2 Background & Motivation

2.1 Serverless

In serverless Function-as-a-Service (FaaS) architectures, myriad responsibilities associated with server management are delegated to cloud providers. Users only need to submit code for their functions, which are deployed and executed in lightweight and temporary *containers* along with all necessary dependencies. Cloud providers can allocate either more or fewer resources to each function by scaling its container instances up or down, respectively.

2.2 Quality of Service

Quality of service (QoS) measures the system's performance from users' perspectives. QoS for cloud applications consists of two metrics: *latency* and *throughput*.

Latency refers to the response time of requests sent to the provider, typically measured in milliseconds or microseconds. In particular, QoS is typically determined using *tail latency*, which is the slowest 90^{th}, 95^{th}, or 99^{th} percentile of requests. This is because even if the majority of requests are executed quickly, large lag spikes caused by the occasional slow request disrupt QoS for the user.

Throughput is the rate at which requests are completed, typically measured in Requests Per Second (RPS). By meeting users' throughput targets, cloud providers ensure that users' requests are executed quickly enough.

To guarantee that QoS constraints are met, users make Service Level Agreements (SLAs) with providers that establish specific objectives for latency and throughput: for instance, serverless users and a cloud provider may agree that a particular function's requests must be executed at a throughput of 500 RPS with the 95^{th} percentile latency remaining below 100 ms. Cloud providers must therefore allocate resources efficiently such that both resource usage is minimized and SLAs are met.

2.3 Reinforcement Learning

Overview. Reinforcement learning (RL) is an unsupervised learning method in which the model "learns" via experience acquired through repeated interactions in a trial-and-error manner. In RL models, an *agent* observes a *state* from an *environment* and, based on this observation, takes *actions* to maximize a numerical *reward*. The agent relies on experience and feedback in the form of the reward to deduce which actions are optimal in a given environment state.

Formally, at each timestep t, the agent observes state $s_t \in S$ representing the environment and takes action $a_t \in A$, where S and A are the set of all possible states and the set of all possible actions, respectively. The agent receives an instantaneous reward r_t as feedback reflecting the desirability of the action, and at timestep $t+1$, the environment transitions to a new state s_{t+1}. The training loop is continued until some termination condition, such as a time constraint, is met.

Q-Learning. Q-Learning is a model-free RL algorithm that aims to learn a function $Q(s,a)$ to estimate expected cumulative reward, called the Q-value, given a (state, action) pair. By learning this function, the agent can always select action a in state s such that $Q(s,a)$ is maximized. The most basic Q-Learning algorithm uses a structure called a Q-table to keep track of expected cumulative reward for (state, action) pairs. At each timestep t, each entry in the Q-table corresponding to the pair (s_t, a_t) is updated according to the rule:

$$Q(s_t,a_t) \leftarrow (1-\alpha)Q(s_t,a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1},a)],$$

where r_t is the instantaneous reward computed using the reward function, and α and γ are hyperparameters representing the learning rate and the discount factor, respectively. Q-Learning, however, struggles when the state or action space is continuous, since the Q-table is only effective when the agent repeatedly observes a limited number of states. In the serverless resource management problem, the state space includes continuous variables such as resource utilization and target throughput values, so we employ a variant of Q-Learning called Deep Q-Learning to address this.

Deep Q-Learning. In Deep Q-Learning, the Q-table is approximated by a deep neural network called a Deep Q-Network (DQN) that effectively mimics the functionality

of the Q-table: mapping (state, action) pairs to a Q-value. During each timestep t, the DQN takes the current state s_t as input and outputs a Q-value for each action in the action space. The learner can then select the action corresponding to the highest Q-value. Over many training iterations, the DQN adjusts its weights to form a mapping that approximates the function $Q(s,a)$.

In order to do this, two copies of the DQN are used: a *target network* and a *policy network*. The target network is updated with a soft update (i.e. less dynamically) and outputs Q-values that are used as targets for the policy network: at each timestep, the weights of the policy network are adjusted to minimize the Temporal Difference (TD) error, i.e. the difference between its Q-values and the target network's Q-values. As TD error is minimized, the model's training will stabilize and converge, since both dynamic and soft updates to the DQN will have minimal impact on the model's learned behavior.

Experience Replay. Even though the use of neural networks allows for learning in a continuous state space, neural networks are data-hungry: many experiences (i.e. state, action, reward tuples) are needed to effectively train them. Large quantities of open-source serverless data from production are not widely available, so the networks must be trained using collected experience; however, each of the agent's interactions with the environment incurs a significant temporal cost—at each timestep, time is spent creating and terminating containers, running functions, and, occasionally, cleaning up crashed containers.

To address these challenges, we use experience replay, a technique in which the model trains using experiences randomly sampled from past experiences collected in a *replay buffer*. By storing experiences and reusing them, data efficiency is significantly improved. In addition, the model is trained with randomly sampled batches of prior experience instead of relying on the model's most recent experience at each timestep, removing temporal correlations in consecutive experiences that can lead to unstable training and slow convergence.

2.4 Related Work

To the best of our knowledge, there are no open-source frameworks for reinforcement learning-based autoscaling in production-grade serverless environments. As an improvement over prior work, SCARLET (1) is integrated and evaluated with a production-grade system and real serverless benchmarks, (2) allows for a continuous state space to more precisely capture differences in resource utilization, and (3) follows the standard RL API advocated by OpenAI, allowing for adaptability to a variety of RL models. These differences are summarized in Table 1.

Rule-based Autoscaling. Serverless platforms currently in production manage resources using rule-based autoscalers [1, 4, 7]. These automata read metrics such as CPU and memory utilization and scale up or down to keep resource utilization within pre-defined, fixed limits. Since workloads vary greatly with respect to expected resource usage, different limits must be manually defined for different serverless functions. For instance, if a rule-based autoscaler encounters an unfamiliar workload, resources will be either overprovisioned or underprovisioned to the new workload. Defining accurate rules for individual workloads, however, can be both time-consuming and challenging, often demanding significant technical expertise.

RL-driven Autoscaling. To address the inflexibility of rule-based autoscalers, many recent works in resource management have used reinforcement learning to automatically allocate resources.

Schuler *et al.* [20] use Q-Learning for horizontal scaling in the Kubernetes-based Knative serverless platform. They propose a model in which the agent observes an environment state consisting of three features: the system's concurrency limit, the average CPU utilization per user-container, and the average memory utilization per user-container. The agent then chooses between the actions of increasing, decreasing, or maintaining the concurrency limit, after which it receives an immediate reward based on the ratio between the measured throughput resulting from the action and an artificial reference value defined by the best throughput value obtained to date. Although they demonstrate the viability of an RL-based approach to optimizing Knative configurations, they discretize the continuous state variables using binning, which limits the precision of the agent's observations. Also, they evaluate their model using a synthetic workload, rather than with production-grade serverless benchmarks.

Agarwal *et al.* [15] explore the possibility of using model free Q-Learning to reduce cold starts by learning function invocation patterns for a specific workload, thus determining in advance the optimal number of function instances. The researchers in this work define the state of the agent's environment as a combination of the number of function instances available and per-instance CPU utilization of the function. The agent's action space consists of either scaling up or scaling down the number of instances without scaling to 0 or exceeding a given upper bound. The immediate reward is determined by a weighted sum of CPU utilization, function instance count, and request success or failure rate during the observed time span. Like in [20], the authors use binning, and their evaluation is limited to the use of a simulated workload rather than of real serverless benchmarks.

Variations on RL-driven Autoscaling. Qiu *et al.* [18] propose SIMPPO, an online learning framework that uses multi-agent reinforcement learning to manage resources in multi-tenant serverless FaaS platforms, i.e. platforms in which resources are shared by multiple users. In their work, the researchers suggest using multiple agents in a shared environment, each with its own Proximal Policy Optimization (PPO)-trained policy, and each managing its own function. During training, other agents are treated as

Category	[4]	[20]	[15]	[18]	[19]	SCARLET
Serverless Benchmarks	✓	✓	✗	✓	✗	✓
Kubernetes	✓	✗	✓	✗	✓	✓
RL	✗	✓	✓	✓	✓	✓
Continuous State Space	-	✗	✗	✓	✓	✓
Follows RL API	-	✗	✗	✗	✓	✓

Table 1: Comparison of Related Works

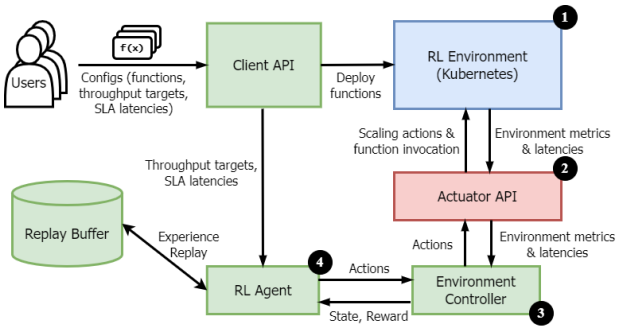

Figure 1: Overview of SCARLET system design.

State Space S
Number of Containers, Resource Utilization (CPU, Memory, Network), Target Throughputs
Action Space A
For each function, increase the number of containers by -1, 0, or 1
Reward Function r_t
if SLA violation, $r_t = -1$, **else** $r_t = 1 - \left(Mean\left(\frac{Tail\ Latency}{SLA}\right)\right)^{0.4}$

Table 2: RL-based autoscaling characterization.

part of the environment, and reward is calculated as the average of all agents' rewards, which the authors justify using mean-field theory. The work achieves a significant improvement over single-agent reinforcement learning in multi-tenant cases with reasonable resource overhead.

Finally, AWARE integrates RL in a Kubernetes environment for microservice resource management [19]. It is compatible with different types of RL models thanks to its following of the OpenAI standard API; however, it is designed for and evaluated with microservices rather than with serverless benchmarks. In contrast, we design SCARLET specifically for serverless workloads.

3 System Design

3.1 Overview

We now discuss the design of SCARLET. First, we present our choices for the state space, action space, and reward function. Then, we describe the four core components of our design: the *RL Environment*, the *Actuator API*, the *Environment Controller*, and the *RL Agent*. In addition, we include a *Client API* to parse users' workload configurations, as well as a *Replay Buffer* to facilitate experience replay for the RL Agent. The system allows users to run workloads easily by submitting a JSON configuration file consisting of the workload functions, as well as each function's throughput target and SLA latencies, following the definition template. The configurations are then passed into the Client API, which deploys the functions and triggers the RL Agent to begin learning. A complete overview of the system's design is depicted in Figure 1.

3.2 Autoscaling with RL

Instead of relying on rule-based heuristics, an RL-based autoscaler learns to make scaling decisions directly from the characteristics of specific workloads and the state of its operating environment. The scaler's RL agent interprets the environment by reading key metrics, including the current number of containers, resource utilization values, and the target throughput values for each of the functions. Based on these conditions, the agent makes a decision on whether an increase, a decrease, or no change in the number of containers is needed. Once it selects and executes an action, the functions are invoked with the new scaling configurations, and the resulting tail latency values are collected. If any of these values exceeds the SLA value, the agent is heavily punished. Otherwise, the agent receives a reward based on the mean ratio between the measured latency and the SLA latency across of all the functions, with ratios closer to 0 yielding higher reward and ratios closer to 1 yielding lower reward. Thus, the lower the measured tail latency values relative to their respective SLA values, the higher the reward received by the agent. Table 2 summarizes the model's state space, action space, and reward function.

3.3 RL Environment

The RL Environment (denoted by 1 in Figure 1) consists of a Kubernetes cluster deployment. Kubernetes is a production-grade cloud platform used widely in industry, so we use it to run the autoscaler in a realistic production cloud environment.

3.4 Actuator API

The Actuator API (denoted by ❷ in Figure 1) is a shim layer that facilitates interaction between the RL Environment and the RL learner. To allow the RL Agent to take actions in the environment, observe its state, and receive feedback, the Actuator API is designed with three key functionalities. First, it translates scaling actions received from the RL Agent via the Environment Controller into scaling commands that accordingly update the number of containers in the Kubernetes cluster. Second, it samples resource utilization metrics from the RL Environment in real time. Finally, it invokes the user-submitted functions with the target throughput value for a short duration and collects the latency values from each invocation. Specific implementation details will be further discussed in Section 5.

3.5 Environment Controller

The Environment Controller (denoted by ❸ in Figure 1) is a multipurpose API layer between the RL Agent and the Actuator API. It acts primarily as an API gateway for the RL Agent: it unpacks environment metrics returned from the Actuator API and combines them with other environment conditions such as the number of containers and the functions' throughput targets to form a state vector that is fed to the RL Agent. Likewise, it aggregates the tail latency values received from the Actuator API and feeds them into a reward function to obtain a numerical reward, which is also returned to the RL Agent. The Environment Controller also translates the RL Agent's scaling actions into invocations of the Actuator API to interact with the RL Environment. Lastly, the Environment Controller supports the stability of the RL Environment by removing and replacing crashed containers that may potentially bottleneck the system while the RL Agent is training.

3.6 RL Agent

The RL Agent (denoted by ❹ in Figure 1) interacts with the RL Environment through the Environment Controller. At each timestep, based on a state $\mathbf{s}_t$ it receives from the Environment Controller, the RL Agent makes a scaling decision represented by an N-dimensional vector $\mathbf{a}_t$ with elements $a_i \in \{-1, 0, 1\}$, where N is the number of functions submitted by the user. The action $\mathbf{a}_t$ is then fed to the Environment Controller, which eventually returns a reward r_t based on the action's impact on QoS. The learner stores the tuple $(\mathbf{s}_t, \mathbf{a}_t, r_t)$ into the Replay Buffer, allowing this experience to be used for experience replay in the future.

Notably, in SCARLET's design, the RL Agent can be easily instantiated using any RL algorithm, provided that the implementation follows the OpenAI Gym API standard described in Section 4.3. No matter through which algorithm the RL Agent learns, our design allows it to observe a state, take actions, and receive feedback via a reward. This feature makes SCARLET compatible for experimentation in serverless resource management using any RL algorithm, which is highly beneficial for further research.

4 Implementation

4.1 RL Environment

The RL Environment is automatically set up in a cluster using a Python script. The script installs Kubernetes onto each of the nodes in the cluster, as well as the Prometheus Metrics API [11] and vSwarm onto the master node, which acts as a controller for the cluster.

4.2 System Actuation

To actuate the system, we implement the Actuator API shim layer using Python code. Specifically, we implement three functionalities: scaling, real-time metrics collection, and function invocation.

Horizontal Scaling. We use the open-source Kubernetes API [6] to perform horizontal scaling in the RL Environment. The Kubernetes API allows us to run commands to quickly and reliably deploy, scale, and terminate containers in the Kubernetes cluster. When the Actuator API receives a scaling decision from the Environment Controller, it triggers a Kubernetes API call to obtain the current number of containers for each function in the RL Environment. Then, these values are updated according to the scaling decision: they are either increased by 1, decreased by 1, or maintained. The new configurations are then passed through another call to the Kubernetes API that scales the functions to the updated replica counts. Scaling requests for separate functions are made in parallel through multiprocessing.

Real-time Metrics Collection. To collect node-level metrics such as CPU utilization, memory usage, and network transmission rate that constitute the environment state, we implement real-time metrics monitoring using Prometheus, the standard monitoring service for Kubernetes clusters [11]. The Actuator API contains a `sample_env(interval)` method that uses Prometheus service queries to collects each node's average CPU utilization, available memory, and total network traffic over the past `interval` seconds. The metrics are compiled into a Python dictionary to be unpacked by the Environment Controller.

Function Invocation. The open-source vSwarm Invoker [14] is used to invoke functions for specified RPS and duration via HTTP requests. Prior to each function invocation, a call is made to the Kubernetes API to collect the IP address and port of the containerized function to be invoked. These parameters, along with the function's user-submitted RPS target and a pre-defined per-timestep invocation duration, are then passed to the

`invoke_service()` method in the Actuator API, which makes the appropriate call to the vSwarm Invoker. When the invocation duration expires, the invoker returns a CSV file containing the latencies of each request, which is passed to the Environment Controller. Function invocation, like scaling, is done in parallel with multiprocessing.

4.3 Integration with RL

We have integrated system actuation with the RL learner by implementing the Environment Controller API layer described in Section 3. In the Environment Controller layer, Python dictionaries containing Prometheus metrics received from the Actuator API are unpacked, and the resulting metrics are combined with target RPS and the total number of containers to form the environment state.

The Environment Controller also takes the latency values from function invocation and uses them to compile statistics such as the 50^{th}, 90^{th}, 95^{th}, and 99.9^{th} percentile latencies for each function. These statistics are then passed into the reward function, which computes the instantaneous reward based on the formula defined in Table 2. At the conclusion of each timestep t, the environment state s_{t+1} and the instantaneous reward r_t are computed and are returned to the RL Agent.

Our implementation of the request-response pattern between the RL Agent and the Environment Controller follows the OpenAI Gym API model, which has become the standard for RL implementations [9]. As a result, other RL algorithms implemented with compliance to the OpenAI Gym API can easily be tested in the RL Environment by substituting them in for the Deep Q-Learning Agent we have implemented.

4.4 Deep Q-Learning Agent

We implement a Deep Q-Learning RL Agent (described in Section 3.5) using PyTorch [12]. The Deep Q-Network is implemented as a feed-forward network consisting of an input layer with n_d neurons, two fully-connected hidden layers with 128 neurons each, and an output layer with n_A neurons, where n_d is the dimension of each environment state, and n_A is the size of the action space. The ReLU activation function is applied to the first and second layers. Huber loss is used to minimize the TD error (see Section 2), and the Adam optimizer is used for parameter updates.

To balance exploration and exploitation, we use an ε-greedy algorithm. At each timestep, the agent explores (i.e. takes a random action) with probability ε and exploits (i.e. takes its best known action) with probability $1-\varepsilon$. We use the ε-decay technique to decrease ε throughout training in order to facilitate convergence as the agent learns the optimal policy.

Servers (5×)	ProLiant XL170r Gen9
Processor	E5-2640 v4
Architecture	x86_64
Memory	65.73 GB
Ethernet NIC	Intel E810 10 GbE NIC
Operating system	UBUNTU22-64-X86

Table 3: Testbed hardware and software configuration.

Parameter	Value
Replay Buffer Size	10^6
Optimizer Learning Rate	10^{-4}
Discount Factor	0.99
Soft Update Coefficient	0.005
Exploration Factor	$\varepsilon(0.9), \varepsilon\text{-decay}(150)$

Table 4: Deep Q-Learning hyperparameters.

5 Evaluation

5.1 Experimental Setup

Hardware. We use Cloudlab to remotely provision hardware on which to run our experiments. For our experiments, we use five ProLiant XL170r Gen9 machines; the exact hardware specifications are shown in Table 3.

Serverless Benchmarks. To test the model in scenarios that are as realistic as possible, we conduct system evaluation using applications from the vSwarm Benchmarking Suite [14]. The vSwarm Benchmarking Suite is a collection of ready-to-run serverless benchmarks, each typically containing a number of interconnected serverless functions. vSwarm includes benchmarks such as `fibonacci-python`, `online-shop`, and DeathStarBench's [3] `hotel-app`, and these functions are designed to be realistic data-intensive workloads. Because the benchmarks' default manifest YAML files were written for deployment with Knative [5] rather than for direct Kubernetes cluster deployment, we modified the benchmarks' manifests to fit standards established by the Kubernetes API documentation [6]. In the model evaluation, we use one instance each of `fibonacci-python`, `hotel-app-geo`, and `hotel-app-profile` from vSwarm.

Model Hyperparameters. The Deep Q-Learning algorithm features many hyperparameters pertaining to various aspects of the algorithm. In our evaluation, we use the hyperparameters summarized in Table 4. The specific optimal tuning of these hyperparameters will be subject future research.

5.2 Results

We show that SCARLET allows a Deep Q-Learning agent to achieve quality-of-service. We evaluated the agent in SCARLET using the aforementioned implementation and experimental setup. Figure 2 shows the 90^{th}

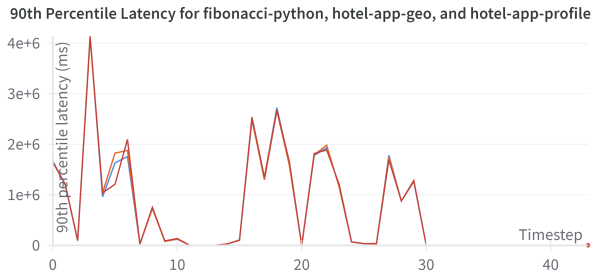

Figure 2: Tail latency from Deep Q-Learning experiment.

Figure 3: Reward from Deep Q-Learning experiment.

percentile tail latency for `fibonacci-python` (orange), `hotel-app-geo` (blue), and `hotel-app-profile` (red) over each of the learner's training iterations. The instantaneous reward computed at each step is also plotted and is shown in Figure 3. Notably, the reward begins converging at around the $t = 6$th iteration. The performance suffers a sudden degradation in a later timestep, likely due to a poor exploration decision, before converging again. Accordingly, latencies beyond the $t = 30$th timestep are minimized, demonstrating that quality-of-service is achieved and showing that the agent has learned an optimal scaling configuration for this set of functions.

6 Discussion & Future Work

6.1 Offline Learning

Currently, the RL agent faces a training overhead when it is initially exploring the environment and still learning, which leads to subpar performance. One solution is to rely on the rule-based autoscaler until the RL agent has accumulated sufficient experience to make good decisions. [19] During this initial phase, the rule-based autoscaler would make scaling decisions in the serverless environment (online); meanwhile, the RL Agent will train its policy offline by sampling data collected by the rule-based scaler. Eventually, the RL Agent's performance will equal or surpass that of the rule-based scaler, at which point the RL Agent would be deployed for online learning directly in the serverless environment, as it is in the current implementation.

6.2 Multi-Tenancy

SCARLET's design accounts for the presence of multiple functions running simultaneously in the serverless environment, and the agent learns to scale optimally in these multi-tenant situations. However, one of SCARLET's limitations is that the total number of functions running in the serverless environment is predetermined and fixed, which limits its flexibility. Also, because the action space consists of all possible combinations of actions for each individual function, its size grows exponentially with respect to the number of functions present. To make SCARLET more comprehensive and adaptive to dynamic situations, a multi-agent design could be used. In a multi-agent RL system, each function would have its own agent, so changes in the number of functions running would be addressed by the assignment of either more or fewer agents.

6.3 Chained Functions

Some workloads, such as vSwarm's `video-analytics` benchmark, involve chained functions that are dependent on one another. SCARLET's current design does not account for chained functions. This is because each function in the chain has its own optimal scaling configuration with respect to its dependent functions; however, the RL Agent in the current implementation cannot learn to optimize quality-of-service because it does so by invoking individual functions separately. In future work, the inter-service dependencies could potentially be modeled by using critical service localization, as described in FIRM [17].

7 Conclusion

We design and implement SCARLET, a framework for serverless resource management in a production-grade cloud environment using reinforcement learning. SCARLET is open-sourced and is accessible via our project GitHub [10] and offers compatibility with other RL models. We use SCARLET to implement a Deep Q-Learning model and evaluate it using open-source serverless benchmarks. Our experimental results demonstrate that SCARLET allows the model to converge to an scaling optimal policy that maintains throughput while also minimizing tail latency, thereby satisfying quality-of-service.

References

[1] Application auto scaling. https://docs.aws.amazon.com/autoscaling/.

[2] Cloud native computing foundation serverless whitepaper. https://github.com/cncf/wg-serverless/blob/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf.

[3] Deathstarbench. https://github.com/delimitrou/DeathStarBench.

[4] Horizontal pod autoscaler. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[5] Knative. https://knative.dev/.

[6] The kubernetes api. https://kubernetes.io/docs/concepts/overview/kubernetes-api/.

[7] Load balancing and scaling. https://cloud.google.com/compute/docs/load-balancing-and-autoscaling.

[8] Open source serverless cloud platform. https://openwhisk.apache.org/.

[9] Openai gym. https://www.gymlibrary.dev/.

[10] Project github repo. https://github.com/barabanshek/MIT_PRIMEs/tree/main.

[11] Prometheus - monitoring system & time series database. https://prometheus.io/.

[12] Pytorch. https://pytorch.org/.

[13] Serverless functions, made simple. https://www.openfaas.com/.

[14] vswarm - serverless benchmarking suite. https://github.com/vhive-serverless/vSwarm.

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

[16] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N. J., Gonzalez, J. E., Popa, R. A., Stoica, I., and Patterson, D. A. Cloud programming simplified: A berkeley view on serverless computing. *CoRR abs/1902.03383* (2019).

[17] Qiu, H., Banerjee, S. S., Jha, S., Kalbarczyk, Z. T., and Iyer, R. K. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 805–825.

[18] Qiu, H., Mao, W., Patke, A., Wang, C., Franke, H., Kalbarczyk, Z. T., Başar, T., and Iyer, R. K. Simppo: A scalable and incremental online learning framework for serverless resource management. In *Proceedings of the 13th Symposium on Cloud Computing* (New York, NY, USA, 2022), SoCC '22, Association for Computing Machinery, p. 306–322.

[19] Qiu, H., Mao, W., Wang, C., Franke, H., Kalbarczyk, Z. T., Başar, T., and Iyer, R. K. AWARE: Automate workload autoscaling with reinforcement learning in production cloud systems. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)* (Boston, MA, July 2023), USENIX Association, pp. 1–17.

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

[21] Ustiugov, D., Petrov, P., Kogias, M., Bugnion, E., and Grot, B. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2021), ASPLOS '21, Association for Computing Machinery, p. 559–572.

[22] Yu, T., Liu, Q., Du, D., Xia, Y., Zang, B., Lu, Z., Yang, P., Qin, C., and Chen, H. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (New York, NY, USA, 2020), SoCC '20, Association for Computing Machinery, p. 30–44.