Reinforcement Learning Based Serverless Container Autoscaler

Evan Ning[*] Nikita Lazarev[†] Varun Gohil[†]

***Abstract*—Cloud computing, characterized by vast data centers with millions of high-performance computers, has revolutionized the way developers run code, offering scalability without the constraints of hardware limitations. Serverless Function as a Service (FaaS) within cloud computing has emerged as a popular paradigm, freeing users from resource management responsibilities and adopting a pay-per-function-call model. While this approach is resource-efficient and cost-effective for users, it introduces challenges for serverless providers in maintaining Quality of Service (QoS). Effective resource allocation in serverless environments is critical, yet challenging. Under-provisioning can lead to function execution failures, necessitating resource redeployment and compromising QoS. Conversely, over-provisioning results in inefficiency as functions operate with more resources than required. The dynamic nature of serverless environments, characterized by diverse functions with varying workloads and short task durations, adds complexity to resource allocation. Current serverless providers often employ Finite-State-Machine (FSM)-based resource managers, necessitating manual tuning of parameters like autoscalers, load balancers, and CPU frequency governors. To address these challenges, machine learning methods, particularly reinforcement learning (RL), have been explored. RL's adaptability to dynamic serverless environments, where functions exhibit diverse characteristics, makes it a compelling choice. In this paper, we present an RL-based approach to resource management, leveraging its ability to simultaneously optimize multiple parameters without manual intervention. Our implementation utilizes RL algorithms, including Deep Q Learning, to provide scaling recommendations for cloud providers, demonstrating successful convergence in both horizontal and vertical scaling scenarios. To evaluate our approach, we constructed and replicated a serverless environment using vHive, vSwarm, and Kubernetes. The results indicate not only successful convergence in scaling but also rapid adaptability—a crucial attribute in the context of dynamic serverless environments. This research contributes valuable insights into the application of RL in serverless resource management, paving the way for future advancements in the field.**

[*]Milton Academy
[†]Massachusetts Institute of Technology

1 Introduction

Physically, cloud computing is a huge data center with millions of high-quality computers. The cloud provider can sell these individual computers for consumers to use at a time basis (typically on a monthly or yearly basis) and the consumer can run their code using the machines. Cloud computing is great as it allows developers, ranging from new startups to huge companies, to run code without being constrained by hardware. Many companies opt to use cloud computing because they not only do not need to build their own data centers, but also can easily expand as cloud computing offers virtually infinite resources.

Serverless Function as a Service (FaaS) is a type of cloud computing that has gained lots of popularity recently, with companies such as GrubHub and Netflix switching to it. FaaS frees customers from managing their own resources or function allocations, and instead customers pay on a per function call basis. This method is not only more resource efficient for everyone, but also more cheap for the customer while maintaining a similar Quality of Service (QoS). However, the serverless provider now has to manage the resources and functions for the customer in such a way that the QoS can still be maintained.

There are many challenges for the serverless provider when managing and allocating resources. One mistake is to provision not enough resources for a certain function. The function will not be able to execute, and the serverless provider will have to redeploy the function with more resources. This will make the latency of the function too high, and the QoS will be violated. On the other hand, if too many resources are provisioned, while QoS will be maintained and the function will return, it is inefficient as the function does not need that many resources. However, these are not easy problems to solve because of the nature of serverless environments. Due to the high amount of different functions each requiring different loads and all of them being relatively short tasks, it is difficult to adapt and allocate resources.

Many current serverless providers use a Finite-State-Machine (FSM) based resource manager to solve the problem of resource allocation. These require manual tuning of hard coded parameters, such as an autoscaler, load balancer, CPU frequency governors, etc. Recently though, machine learning methods have been tried, especially reinforcement learning (RL). Reinforcement learning is great as it is adaptable to a serverless environment, which is dynamic with various different functions requiring different loads. One agent can tune all the previous parameters at once. Also, the data is abundant, with instant reward able to be given making training easy. In this paper, we attempt to use an RL algorithm in order to solve the problem of resource management.

2 Related Work

In the realm of resource management for serverless computing, the Aquatope [7] paper represents a notable contribution by leveraging machine learning techniques. Specifically, Aquatope employs a Long Short-Term Memory (LSTM)-based model to analyze historical data, predicting future resource requirements. This predictive resource management approach offers insights into potential avenues for exploration and integration within a reinforcement learning (RL) framework, showcasing the versatility of machine learning in enhancing resource allocation strategies.

Additionally, the landscape of resource management in cloud computing has seen diverse methodologies. The FIRM [4] paper introduces an innovative application of reinforcement learning specifically tailored for microservice environments. Although distinct from serverless architectures due to the presence of 'paths' in microservices, FIRM demonstrates the efficacy of reinforcement learning as a robust resource management solution. This finding suggests the adaptability of reinforcement learning across different cloud computing paradigms and underscores its potential superiority in resource optimization.

These contributions collectively highlight the evolving landscape of machine learning applications in resource management within cloud computing. While Aquatope provides insights into predictive resource allocation, FIRM extends the applicability of reinforcement learning to microservices, thereby informing potential strategies for enhancing resource management in serverless environments. As we delve into our own exploration, these studies offer valuable benchmarks and methodologies for consideration and potential integration into our reinforcement learning-based resource management model.

3 Approach

3.1 Quality of Service (QoS)

Quality of Service is a metric that is typically measured by the latency of a function. When a function is called, there is time that it takes to reach the server, execute, and send the results back, which added together is the latency of that function call. QoS will usually be a certain latency value that, if exceeded, will violate the QoS. After the function is called many times, the latencies can then be sorted and plotted in a graph. While one aspect is to keep the median latencies lower, the more important part is to try to limit

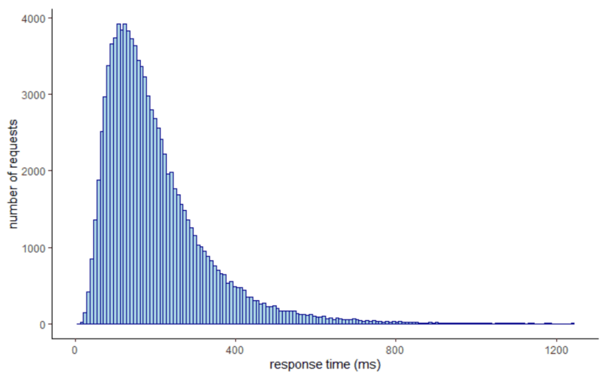

Figure 1: Latencies gathered by running a function many times. [6]

the length of the end or tail latencies. This is because the QoS violations will first occur from those tail latencies, so if there is a longer tail the probability of a violation will be larger. Another common measurement for QoS is the 90th or 99th percentile latencies. These are useful in order to judge how long the tail is, and to estimate the percentage of requests that will violate QoS.

Latency depends on a number of different factors. One common issue is cold starting. When a function is deployed for the first time, a new container needs to be setup and receive the function. This usually takes a substantial amount of time, so it is crucial to avoid cold starts whenever possible. Usually, the container will retain the function data for some amount of time after it is called, so that if a new call is initiated it will already be loaded and ready to run, leading to a quick execution. Another factor is the requests per second (rps) of the function. As the rps increases, so will the latency. In some circumstances, this occurs because of hardware limitations. In our case, each machine has a queue of functions. If the functions are being added to the queue faster than they are being executed and cleared from the queue, then it is clear that the latencies of each function will rise as well.

3.2 Cloud and ML infrastructure

In this experiment we used Cloudlabs [1] and vHive [5] in order to simulate our serverless environment. Cloudlabs provides a cluster of virtual machines in order to deploy the functions into containers. vHive is a serverless environment simulator and what actually deploys the serverless functions, and schedules them. In addition, we are using vSwarm [3], a collection of different serverless benchmarks. For this paper, I have used the different functions under the online-shop application. There are a total of 9 different functions, such as adservice and paymentservice, and the goal is to simulate an online shopping website. We then create an environment and API with kubernetes [2] so that after the function is deployed, we can gather data such as the latency values and the resource usage (cpu, memory

50th, 90th, 99th, and 99.9th percentile latency
Average idle, user, and system CPU cycles
Average free memory
Throughput (Requests per Second completed vs Requests per Second asked for)

Table 1: Metrics saved

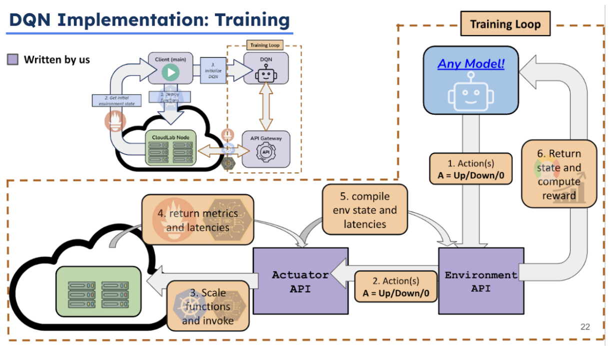
DQN Implementation: Training
Written by us
Client (main)
3. initialize DQN
DQN
Training Loop
2. Get initial environment state
1. Deploy functions
CloudLab Node
API Gateway
API
Training Loop
Any Model!
1. Action(s) A = Up/Down/0
6. Return state and compute reward
5. compile env state and latencies
4. return metrics and latencies
Actuator API
2. Action(s) A = Up/Down/0
Environment API
3. Scale functions and invoke
22

Figure 2: One iteration for the network.

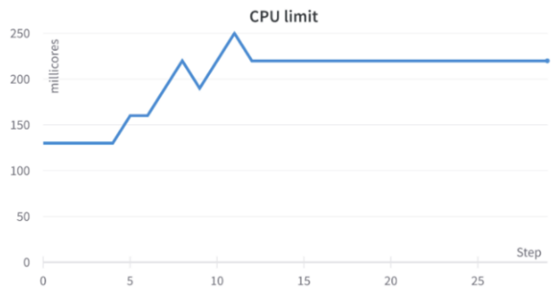

Figure 3: CPU allocated at each step.

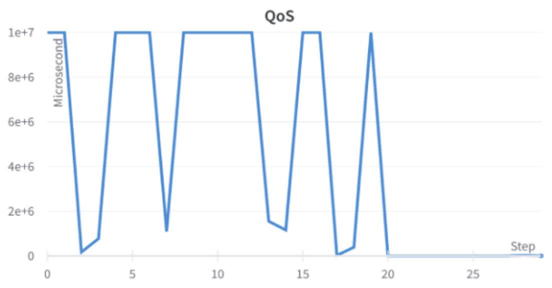

Figure 4: 90th percentile tail latency at each step.

and network). These metrics are useful as they will be important for us to calculate the overall efficiency and QoS when we test new scaling parameters.

Currently, we have written scripts in order to automate the set-up process on Cloudlab and to create the proper environment. Because each cloudlab experiment only lasts for a certain amount of time, it is actually very helpful to automate this process. The setup script involves downloading vHive and vSwarm, and configuring each machine properly.

Finally, there is a python function in order to run each function. Normally, we would have to go into the master node and create the invoker and run the function from there, which would take a lot of time and manual labor. With a python function, not only is it much faster, it can be automated by a simple for loop. This is crucial because when training or gathering data, it can take hours and running each function call manually is a lot of time.

3.3 Machine Learning Driven Autoscaling:

We applied machine learning-based autoscaling in a serverless computing environment through the use of Deep Q Learning. This approach allows for efficient and dynamic scaling of resources to match varying workloads. The system learns and adapts in real-time, enhancing resource allocation based on observed patterns and demands. This results in optimized performance and cost-effectiveness, demonstrating the effectiveness of machine learning in managing cloud resources adaptively and proactively.

Using our python APIs, we create a training loop as the image shows. In each loop, we first send an action to be deployed, such as adjusting the amount of CPU to allocate or increasing/decreasing the number of container replicas. Then we collect the results in the form of latency and resource utilization, and compile them into a state and a reward for the DQN model to receive.

4 Evaluation

We implemented two DQN agents, one for vertical scaling and one for horizontal scaling.

4.1 Vertical Scaling Results

For our vertical scaler, we found quite fast convergence. The QoS latency (90th percentile) reached convergence after around twenty 10 second steps, reaching around 1000-2000 microseconds. The reward similarly converged, while the actions converged after around 10. The reason for this delay between action convergence and reward/QoS convergence could be because the containers terminate/restart after each update, so when the actions converge it takes some time for the results to show up. The reward function also penalized high CPU usage, which is

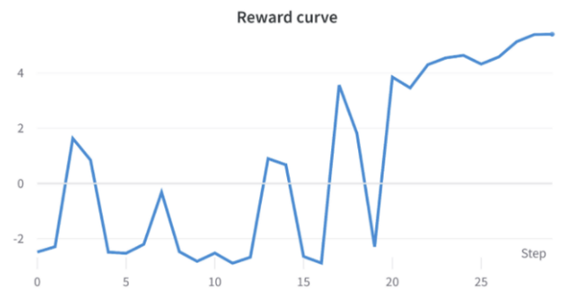

Figure 5: Reward at each step.

why it did not scale infinitely. These patterns were also similar for other RPS values as well as other functions.

5 Limitations & Future Work

5.1 Offline Learning

Currently, the RL agent's performance suffers at the beginning of training. One solution is to use the default autoscaler until the RL agent has accumulated sufficient experience to make good decisions. During this initial phase, the default autoscaler would make scaling decisions in the serverless environment; 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.

5.2 Multi-Tenancy

Our 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 the 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 the model 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.

5.3 Chained Functions

Some workloads, such as vSwarm's `video-analytics` benchmark, involve chained functions that are dependent on one another. Our 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.

6 Conclusion

In conclusion, our research endeavors involved the meticulous construction of robust infrastructure, faithfully replicating diverse serverless environments characterized by varying workloads. This infrastructure served as the testing ground for our innovative approach to resource management, driven by the implementation of Deep Q-Learning. By leveraging this data-driven methodology, we addressed the intricate challenges posed by dynamic serverless environments, aiming to optimize resource allocation and enhance overall system performance.

Github link: https://github.com/barabanshek/MIT_PRIMEs/tree/main

References

[1] Cloudlab. https://www.cloudlab.us/.

[2] Production-grade container orchestration. https://kubernetes.io/.

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

[4] 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.

[5] 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.

[6] Vitillo, R. Why you should measure tail latencies.

[7] Zhou, Z., Zhang, Y., and Delimitrou, C. Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows. In

Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (New York, NY, USA, 2022), ASPLOS 2023, Association for Computing Machinery, p. 1–14.