Reinforcement Learning-based Optimisation to Reduce Function Cold Start Frequency in Serverless Computing

Siddharth Agarwal

Submitted in partial fulfilment of the requirements of the degree of

Master of Science (Computer Science)

Cloud Computing and Distributed Systems Laboratory

School of Computing and Information Systems

THE UNIVERSITY OF MELBOURNE, AUSTRALIA

June 2021

Copyright © 2021 Siddharth Agarwal

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm or any other means without written permission from the author.

Reinforcement Learning-based Optimisation to Reduce Function Cold Start Frequency in Serverless Computing

Siddharth Agarwal

Principal Supervisor: Prof. Rajkumar Buyya

Abstract

Serverless Computing, generally analogous to Function-as-a-Service (FaaS) offering of Cloud Computing, typically focuses on implementing business logics in the form of ephemeral and bounded time functions with a pay-per-use pricing model. Serverless computing model shifts the responsibility of infrastructure and resource related tasks like provisioning and management. to the cloud service provider (CSP) and allows the user to focus on the business logic. This shift of responsibility from the user and an abstraction of the underlying the resources, validates the serverless characteristic of the platform.

The serverless functions are executed as lightweight Virtual Machines (VMs) or containers that are created on-demand as per the requests. The inherent property of FaaS is to provide highly scalable and available function containers, serving the incoming request load. As the requests are generated, on-demand function containers are spawned that involves necessary bootstrapping, before the function could actually respond. The spawning incorporates downloading of function code, execution of an initialisation procedure that allocates required resources, creating code dependencies and setting up the runtime environment, etc. The bootstrapping holds back the invocation of request handler and introduces a delay in the response time of the application. This is known as the 'cold start' of the function container. Alternatively, cold start is the preparation time of function container before serving the incoming requests.

FaaS emerges as an advantageous platform for a variety of real-life applications such as Internet-of-Things (IoT) sensor input processing, stream and batch processing, APIs and mobile backends, etc. and even few machine learning inference tasks. These applications, by attributes, expect a quick and fault tolerant response to operate reliably. But the underlying function 'cold starts' poses as a hinderance in the reliable operation of the application, as an end user solely focuses on the application response time. To address this preparation delay, a handful of techniques like container pooling, continuous function pinging and leveraging application content to reduce cold start, etc. have been presented. These solutions fall short of analysing the invocation patterns of the respective application and availing an intelligent solution that take an informed decision to reduce the 'frequency of cold starts' that occur during an observed period of time. Therefore, we propose a Reinforcement Learning (Q-Learning) agent to inspect the function CPU-utilisation and application invocation patterns to intelligently reduce the frequency of application cold starts by ascertaining and preparing the optimal number of functions required, in advance. An evidence of Q-Learning agent's successful learning capability is presented that realises concerned metrics and application invocation patterns to predict

the optimal number of function instances over a learning period to reduce cold starts. This thesis makes the following key contributions:

1. A Reinforcement Learning Agent implementing model free Q-Learning in a serverless environment setting to reduce the cold start frequencies of a function.

2. Implementing an agent to dynamically learn the function invocation patterns to ascertain optimal number of function instances, reducing cold start occurrences.

3. Evaluation of our proposed agent against the baseline auto-scale policy of the serverless platform for a synthetic function workload pattern.

Declaration

This is to certify that

1. the thesis comprises only my original work towards the Master of Science (Computer Science),
2. due acknowledgement has been made to all the other materials used.
3. the thesis is less than 25,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Siddharth Agarwal, June 2021

Preface

Main Contributions

This thesis research has been carried out in the Cloud and Distributed Systems (CLOUDS) Laboratory, Faculty of Engineering and Information Technology, The University of Melbourne, Australia. The main contributions of this thesis are discussed in the chapters 2-5 and are based on the following publications:

- **Siddharth Agarwal**, Maria A. Rodriguez, and Rajkumar Buyya, A Reinforcement Learning Approach to Reduce Serverless Function Cold Start Frequency, *In Proceedings of the 21th IEEE/ACM International Symposium on Cluster, Cloud, and Internet Computing* (CCGrid 2021, IEEE CS Press, USA), Melbourne, Australia, May 10-13, 2021. – **Best Paper Award** (Runner-Up).

Acknowledgements

I would like to express my gratitude to my principal supervisor, Professor Rajkumar Buyya, for giving me an opportunity to pursue the research under his guidance. I am grateful for his continuous support, guidance and motivation throughout the research. I am also grateful to him for introducing me to the research topic and constantly encouraging me to perform better. I wish to extend my special thanks to my co-supervisor, Dr. Maria A. Rodriguez, for constantly supporting me through the research, providing her invaluable insights to shape the research more methodically and help me with even trivial challenges.

I thank all the members of CLOUDS laboratory for their constant support and motivation. I thank the university for providing me with the NeCTAR Cloud resources to conduct my research experiments.

I wish to show my appreciation to my parents, my sister, my family and my friends, for their unconditional love and support. I thank my father for constantly believing in me and providing the means to support my post-graduation. I thank my mother and my sister for always supporting me, motivating me to work hard and be a permanent support system, for which I will forever be grateful.

Siddharth Agarwal

Melbourne, Australia

June 2021

Contents

List of Figures 10

List of Tables 11

Chapter 1 - Introduction 12

1.1 Motivation 14

1.2 Methodology 15

1.3 Research Problem and Objectives 16

1.4 Thesis Contributions 17

1.5 Thesis Organisation 17

Chapter 2 - A Background on Serverless Computing, Function Cold Start and Reinforcement Learning 19

2.1 Background 19

2.1.1 Characterising Function-as-a-Service Execution Model 19

2.1.2 Utility of Function-as-a-Service Model 23

2.1.3 Challenge of Function Cold Start 24

2.1.4 Kubeless – a Kubernetes-native Serverless Framework 26

2.1.5 Reinforcement Learning: Q-Learning Technique 30

Chapter 3 - Literature Survey 33

3.1 Literature Review of Existing Works 33

Chapter 4 - A Reinforcement Learning Approach to Reduce Serverless Function Cold Start Frequency 39

4.1 System Model 39

4.1.1 Workload Model 41

4.2 Reinforcement Learning Approach 43

4.3 Reinforcement Learning-Agent Setup 47

4.4 Performance Evaluation 49

4.4.1 Analysis of Results 50

4.4.2 Practical Implications 57

Chapter 5 - Conclusions and Future Directions **59**

5.1 Conclusions 59

5.2 Thesis Summary 60

5.3 Future Research Directions 61

Bibliography ***62***

List of Figures

Figure 1. Thesis Organisation. 18

Figure 2. IaaS vs PaaS vs FaaS vs SaaS Cloud Execution Models. 20

Figure 3. Difference between Traditional applications and Event-based Serverless Applications. 20

Figure 4. Typical Cold Start for Different Runtimes taken from [9]. 25

Figure 5. Comparison of Cold Starts for Different Package Sizes (Zipped) taken from [9]. 26

Figure 6. High-level view of Kubernetes Deployment. 27

Figure 7. Kubeless Supported Runtime Environments. 28

Figure 8. Simple Interaction Diagram of Q-Learning Agent. 32

Figure 9. Deployed System Stack Architecture. 40

Figure 10. System Model. 41

Figure 11. Fabricated Workload Pattern. 43

Figure 12. High-level view of Proposed RL-based Agent Learning Process. 47

Figure 13. Proportion of Success vs Training Iteration (Batch). 51

Figure 14. Expected Future Rewards during Iterations. 52

Figure 15. HPA vs RL Agent – Mean Failed-Response Comparison. 53

Figure 16. Comparison of Provisioned Instances. 56

Figure 17. HPA vs RLAgent Performance Comparison. 56

List of Tables

Table 1. Resource Limits imposed by major Service Providers. 21

Table 2. Language Runtime Support for Serverless Providers. 22

Table 3. Pricing Model of Function-as-a-Service Providers. 23

Table 4. Instance Reuse Strategy for Leading Cloud Vendors. 24

Table 5. Summary of few relevant works. 38

Table 6. Function Handler Configuration. 42

Table 7. Summary of Results. 54

Chapter 1 - Introduction

Cloud Computing has seen an explosive growth over the decades and has revamped the use of existing resources. With the rapid growth of data and need for IT resources at hand, Cloud Computing has evolved from an on-premise infrastructure to offering utility-based edge services. In this era of accelerated technological advancements, Cloud Computing presents Serverless Computing as its emerging execution model. According to Cloud Native Computing Foundation (CNCF) [1], Serverless Computing introduces a new cloud native architecture that eliminates the requirement of server management for building and executing the applications. It describes an execution model where applications can be bundled as one or more functions, deployed on the platform and the tasks of execution, scalability and availability are handled on demand. The notion of application packaging in the form of functions without server management establishes Function-as-a-Service (FaaS), as serverless computing's most versatile offering. FaaS is an event-driven Cloud Computing (CC) paradigm that puts forward an architectural style to design applications in the form of function(s), without focusing on the prior resource planning. However, the entire responsibility of resource provisioning, management, patching, scaling and capacity planning lies with the Cloud Service Provider (CSP) [20].

FaaS applications are prepared as a set of time bound, loosely coupled, stateless and ephemeral function(s) (piece of code) and are deployed as light-weight virtual machines (VMs) or containers, offering a fine-grained billing model that corresponds to the costs incurred by the exact demand of resources. The idea of Serverless in no way implies the absence of severs for running the applications [1], instead allows the consumers to spend time on the business-critical tasks rather than focus on resource planning. Therefore, servers are still required to offer a serverless platform, but the service provider abstracts all the resource activities from the application developer or the consumer. This function-based abstraction increases application development agility, while lowering the costs of ownership and overheads.

The ease of application code deployment, highly available and on-demand scalable functions of FaaS platforms have attracted a wide variety of applications ranging from multiple domains such as REST APIs, stream processing, edge-computing, Internet-of-Things (IoT) services, etc. [1,50,54]**.** These applications have stringent response-time requirements and therefore expects a near real-time or instant feedback from the function. [20] Ideally, the FaaS platform was conceptualised to spin-up function instances proportional to on-demand requests and terminate the instances after serving the request. But, practically, commercial platforms like AWS Lambda, Azure Functions or Google Cloud Functions, may choose to re-use the function instance or keep the instances queued for a while to anticipate the future requests [19,30]. Some open-source serverless frameworks such as Fission, Kubeless or Knative, that are built over container-orchestration system, Kubernetes, are also known to exhibit similar properties to retain and re-use function instances to handle the subsequent requests [10,13].

As the workload is generated for the application, new function instances are requested from the serverless platform and a process of container initialisation precedes the serving of requests. The container initialisation process involves downloading of the code image from repository, set-up the code dependencies and runtime environment, set-up networking requirements of container and eventually executing the function handler to

serve the incoming requests. This process brings in an accompanying delay in the response time of the application, known as ‘cold start’, of the function container. This introduced cold start is typically of the order of few milliseconds to few seconds. From the consumer’s perspective, quick application response is the most important concern, to which ‘cold start’ poses as ongoing challenge for the serverless platforms [21,23,31]. Therefore, in other words cold start can be understood as the time taken by the platform to start executing an incoming request. Function cold starts are affected by a number of application related factors as well as the function requirements itself. Recent studies [23,41] have presented that factors like programming language, code packaging and deployment size, CPU or memory requirement limits, etc. have an effect over the function cold starts.

To deal with the resource requirements and serve the future workload, serverless platforms and frameworks make use of the underlying resource metrics to set threshold values for resources. Kubeless, an open-source Kubernetes native serverless framework, makes use of the native metric server and supports resource based autoscaling, for serving the incoming workload [13,24,44,53]. The default offering of Horizontal Pod AutoScaler (HPA), by Kubernetes, derives the new desired function instances based upon the average per-instance CPU-utilisation of the function. HPA starts requesting for new function instances when the function containers run out of requested memory or average the per-instance CPU-utilisation spikes above the specified threshold value [10,14]. This, in turn, leads to function cold starts, while serving the incoming workload and might eventually result in failed responses, if the function cold start time is greater than the request’s time-to-live. Therefore, the cold starts make it difficult to anticipate the future workloads as well as introduces a considerable amount of delay in the application response time, that has a negative impact on the application performance.

To address the challenge of function cold start in a serverless environment, several solutions have been explored by academia and employed by the commercial platforms [48,51,54]. Techniques such as resource pooling – keep idle functions in queue for a set period of time to reduce cold starts (AWS Lambda, Azure Functions) or keep empty container(s) with or without function dependencies to reduce the cold start time and pinging – continuously interact with the functions to keep them alive and in-memory to serve future requests, etc. address the challenge of cold start at the cost of resources. These solutions are solely dependent on resource threshold values and does not account for the application workload and therefore, presents an opportunity to explore the process of function cold start. Hence, this thesis focuses on the challenge of function cold start of serverless function containers by proposing a model-free reinforcement learning agent to analyse the application workload pattern for reducing the function cold start frequency. The agent generates a suitable reward system over the learning period to optimally predict the required number of function instances to reduce the subsequent function cold starts, while accounting for the application workload patterns.

1.1 Motivation

Serverless computing with its on-demand scalability, affordable pricing model and light-weight function containers, comes with inherent challenges and problems. [21,23] These challenges can be broadly listed as security and privacy, container isolation, caching, modes of execution, etc. Among them, the challenge of function cold start still persists and have been a focus of a number of studies to realise the possible solutions. As a recent study [30] discusses the ongoing trends of handling the function cold starts in commercial as well as open-source frameworks, it broadly categorises the approach to deal with cold start problem in two classes. (1) Optimising the environments i.e. an approach to reduce the cold start or the container preparation time itself, and (2) Pinging i.e. ways to minimise the frequency of function cold start occurrences.

FaaS or Serverless platforms were theoretically designed to spin-up a new function container for every incoming request, accounting for its high scalability and availability [20]. But practically, almost all the commercial as well as open source serverless offerings re-use the function containers in order to increase the resource utilisation and indirectly account for the lower number of container cold starts. This re-use of function container perhaps addresses the problem of cold start indirectly but suffers from increased failed responses, that can be accounted to the bounded time nature of the functions. Industry leading commercial platforms such as AWS Lambda, Azure Functions or Google Cloud Functions deal with the challenge of cold starts by keeping a queue of ready function containers in the memory for a default period of time, after which the resources are released [54]. Although this technique has shown to be successful in dealing with the frequent function cold starts on the platform but does suffer during a burst of incoming workload.

Consider a few academically explored solutions, such as keeping a warm queue of empty function containers, queue of containers with dependency mapping created, a container warmup technique for non-first functions in a chain of functions to reduce cold start latency or exploiting the application data similarity to live-migrate containers over peer-to-peer networks. They communicate a feasible solution with considerable performance improvements over the default setting, at the cost of resources, but fail to address the workload pattern of the respective applications that directly affects the frequency of the function cold starts. Since every application has its own customised resource requirements and workload pattern, it is of utmost importance to realise the request bursts for the optimal platform performance for the concerned application. Thus, these function container assignments are non-intelligent solutions that reduce the re-usability of the resources and increase the CPU and memory utilisation, while being unaware of the application workload. Recent studies have successfully identified factors like runtime environment, workload concurrency, CPU and memory setting and networking requirements, etc. that affect the function cold starts on the serverless platform. However, most works [9,48,51,54] focus on commercial serverless platforms such as AWS Lambda and fall short to evaluate open source [44] serverless frameworks such as Fission or Kubeless.

Therefore, this thesis leverages an opportunity to explore an open source platform, Kubeless – a Kubernetes native, easy to use serverless framework and address the challenge of function cold start. The primary focus of the solution is to present a smart, reinforcement learning agent that learns an optimal number of function instances required

to serve the incoming requests based on the application workload pattern. The proposed solution evaluates the feasibility of the Q-Learning algorithm in conjunction with the underlying resource metrics to develop an understanding of workload pattern to reduce the function cold start occurrences, by preparing the optimal function containers in advance. Our agent takes advantage of a CPU–intensive workload to train for learning ideal number of function containers and is benchmarked against the default Horizontal Pod AutoScaler (HPA) setting offered by the Kubeless platform.

1.2 Methodology

The objective of this thesis is to develop a model free Q-Learning agent that can learn to prepare the appropriate number of function instances in advance, via rewards, to reduce the function cold start frequency on a Serverless platform. To demonstrate the agent learning process and evaluate the results against the default setting on the serverless platform, the following methodology were used –

- **Problem Formulation**: To address the challenge of serverless function cold starts, we formulate the problem by focusing on the synthetic application invocation patterns and other relevant metrics like CPU utilisation, failed responses, etc., to focus on reducing the frequency of cold starts on the platform.

- **System and Workload Models:** We define the system model of our experimental setup and present an architectural view of the model. Also, we present the formulated synthetic application workload including the respective function model used for the study.

- **Model Building:** The Reinforcement Learning model is written in Python programming language and makes use of the standard libraries to implement the Q-Learning behaviour. The model is trained with the help of inputs from the serverless platform as well as the workload response data. The agent is deployed on the Master node of Kubernetes cluster to collect the required metrics of the platform to realise the rewards for the learning process.

- **Algorithm:** In this study we train a Q – Learning agent for the task of learning the appropriate number of function instances required to successfully serve the incoming application workload, focusing on reducing the number of function cold starts. A heuristic based variation of the Q – Learning algorithm is designed to assist the agent in learning process and support the actions in unpredicted situations during the testing phase. These heuristics speed up the learning process as well as compensate for the lack of state exploration by agent during training process.

- **Evaluation**: We evaluate our model against the default autoscaling policy provided by the serverless platform i.e. Horizontal Pod AutoScaler. The default setting is also tested against the similar workload model and stress tested over the same serverless cluster.

Our methodology including the discussed steps has produced an evidence of successful applicability of a reinforcement learning algorithm to the problem of function cold start in the serverless compute setting.

1.3 Research Problem and Objectives

As more and more enterprises start shifting to Cloud services, Serverless or Function-as-a-Service platform is emerging as the future of cloud computing. With its exciting pay-per-use pricing model, ease of application development and deployment and most importantly, by reducing the burden of server management, FaaS adds value to the businesses and therefore being adopted rapidly [2]. FaaS platform while providing the abstraction of unlimited scalability suffers from the problem of Function Cold Start and thus affects the deployed application performance. This thesis investigates the problem of cold start from user perspective as well as service provider perspective by focusing on the application invocation pattern analyses, while simultaneously highlighting the limitations of default autoscaling policies in successfully serving the application workload. To achieve this objective, our work addresses the following research questions –

- Can Reinforcement Learning (RL) algorithms be configured to address the challenge of serverless function Cold Start?

  With technology becoming more complex, designing systems that are able to solve the intricate challenges is becoming difficult. The potential of machine learning models like LSTM to utilise historical time-series data to address problems like cold start have been successfully explored. This sets forth a potential of machine learning models in problem states like serverless computing. Therefore, it emerges as an opportunity to explore popular Reinforcement Learning algorithms, that has previously been effective in solving complex problems [42], for a Function-as-a-Service platform challenge like function cold start with minimal human interference or human defined rules. However, configuring the challenge of cold start as a RL optimisation problem has its own limitations such as modelling a continuous state space problem to a discrete model, lack of generality and issue of large state spaces. Thus, it is required to conduct the applicability study of RL algorithm applied to a serverless setting while taking advantage of the process in cold start optimisation.

- Can a smart, RL-based agent analyse the application workload pattern and help reduce the function cold start frequency on the platform?

  For an application consumer, fast and desired response is expected. With Serverless applications designed in a way to support quick and fault tolerant feedback, successful response is the key contributor to the application performance. When the platform requests more function instances to cope with the incoming workload, a cold start delay is introduced in the response. The cold start or the instance preparation time, if larger than the time-to-live of the request, could also lead to unexpected responses indicating either a failed request or

unavailable resource. These challenges oppose the underlying principle of unlimited scalability or high availability of serverless applications. Therefore, it is necessary to design an agent that learns to analyse the application workload pattern to prepare the ideal number of function instances in advance for the next set of requests. Thus, the agent works in congruence with the FaaS principles to serve maximum workload while reducing the number of function cold starts.

1.4 Thesis Contributions

Based on the discussed problem of function cold start in Serverless computing, this thesis makes the following key contributions –

- An application of Reinforcement Learning algorithm to the FaaS platform for reducing the function Cold Start frequency.
- Implementing a smart agent to dynamically analyse and learn the application invocation patterns and other relevant metrics, to ascertain the appropriate number of function instances to reduce the occurrence of cold start.
- An evaluation of our agent against the baseline Horizontal Pod AutoScaler, a default autoscaling policy of serverless framework, for a synthetic application workload in a controlled environment.

1.5 Thesis Organisation

The thesis structure is shown in Figure 1**.** The remaining chapters of the thesis are organised as follows –

- **Chapter 2** presents the background on Serverless computing and Function-as-a-Service platform, discusses function cold start challenge and introduces Kubeless, the serverless framework used. This chapter introduces Q-Learning, a Reinforcement Learning algorithm and its environment model.
- **Chapter 3** presents a literature review on serverless computing and existing function cold start solutions, identifying a gap between the existing works and the proposed solution.
- **Chapter 4** proposes a variation of model free Q-Learning framework that utilises application workload pattern to deal with the function cold start frequency on the serverless platform. This chapter is derived from –
  - **Siddharth Agarwal**, Maria A. Rodriguez, and Rajkumar Buyya, A Reinforcement Learning Approach to Reduce Serverless Function Cold Start Frequency, *In Proceedings of the 21th IEEE/ACM International*

Symposium on Cluster, Cloud, and Internet Computing (CCGrid 2021, IEEE CS Press, USA), Melbourne, Australia, May 10-13, 2021.

- **Chapter 5** concludes the thesis, summarises the key findings and identifies the future research directions.

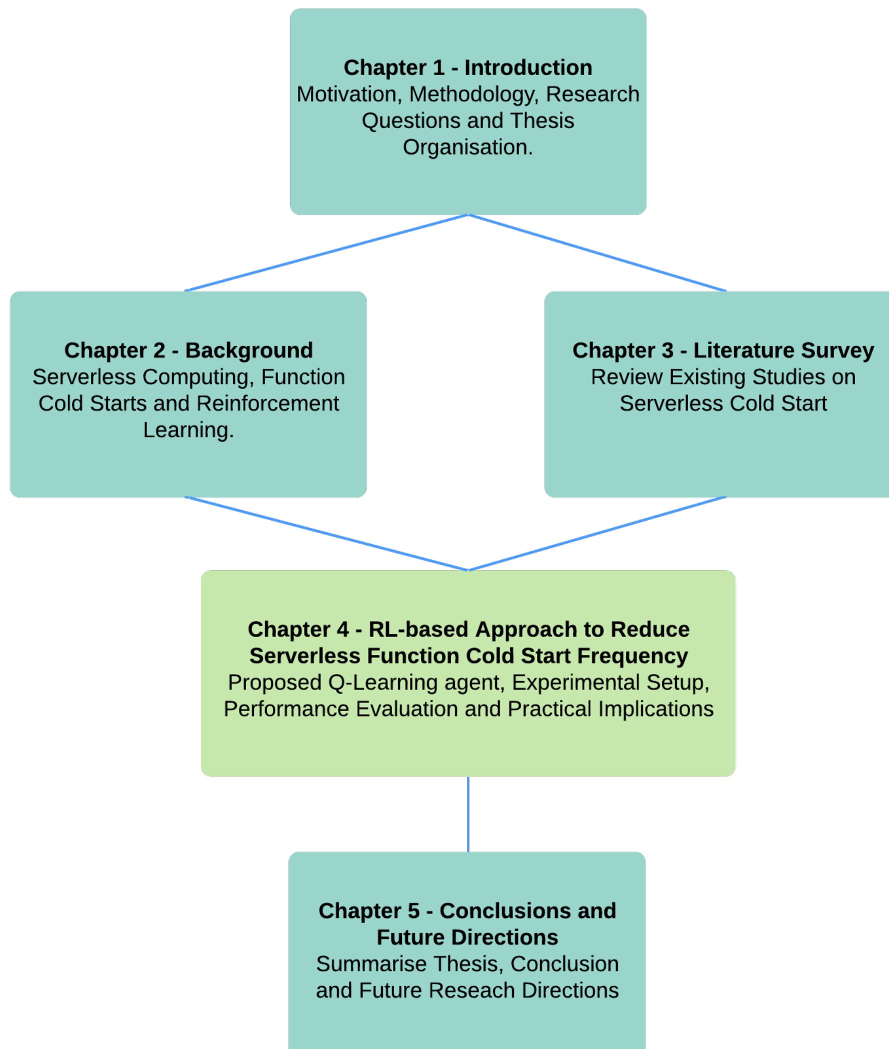

Figure 1. Thesis Organisation.

Chapter 2 - A Background on Serverless Computing, Function Cold Start and Reinforcement Learning

This chapter reviews the concept of Serverless Computing, specifically put to Function-as-a-Service model, focusing on function cold start challenge. The chapter highlights the benefits, limitations and challenges of serverless computing and introduces Kubeless – the choice of serverless framework for the project. A preface is established for the Reinforcement Learning algorithm and introduces model free Q-Learning algorithm used to model the problem of reducing function cold start frequency.

2.1 Background

2.1.1 Characterising Function-as-a-Service Execution Model

Over the years, Cloud Computing has enabled its consumers to shift from traditional ways of thinking about the IT resources to newer paradigm of on-demand, elastic, reliable and cost-efficient ways of using distributed resources. The cloud deployment model was broadly introduced under three categories – Infrastructure-as-a-Service, Platform-as-a-Service and Software-as-a-Service, all with different levels of abstraction and flexibility. But, with the emergence of microservices architecture and huge demand for on-demand elasticity of resources, newer cloud execution model – Serverless Computing came into existence [21,23]. The idea behind the Serverless execution model is to relieve the consumers i.e. the application developers, enterprises, etc., from the complexities of resource management tasks and shift these responsibilities to cloud service provider. Therefore, this model abstracts the underlying servers and other resources from the users (Figure 2) and provides an illusion of unlimited scalability at a very fine-grained price subscription, encouraging developers to focus on tasks that adds value to business.

Contrary to the name 'serverless', servers are still required to run the application code, but the service provider needs to manage the resources and provide the abstraction over them. The service provider may incur some costs even for idle resources, but the consumers only pay for the consumed resources, in-line with the 'pay-as-you-go' characteristic of Serverless [1]. A serverless computing platform is, in general, analogous with Function-as-a-Service (FaaS) execution model. Function-as-a-Service presents the characteristics of serverless computing by providing an event-driven computing that triggers the associated actions. FaaS allows the developers to build the applications on the principle of distributed micro-services (Figure 3) and deploy them as a piece of code

in the form of 'function', a container or a lightweight Virtual Machine (VM) that are executed, billed and scaled in response to associated events [50].

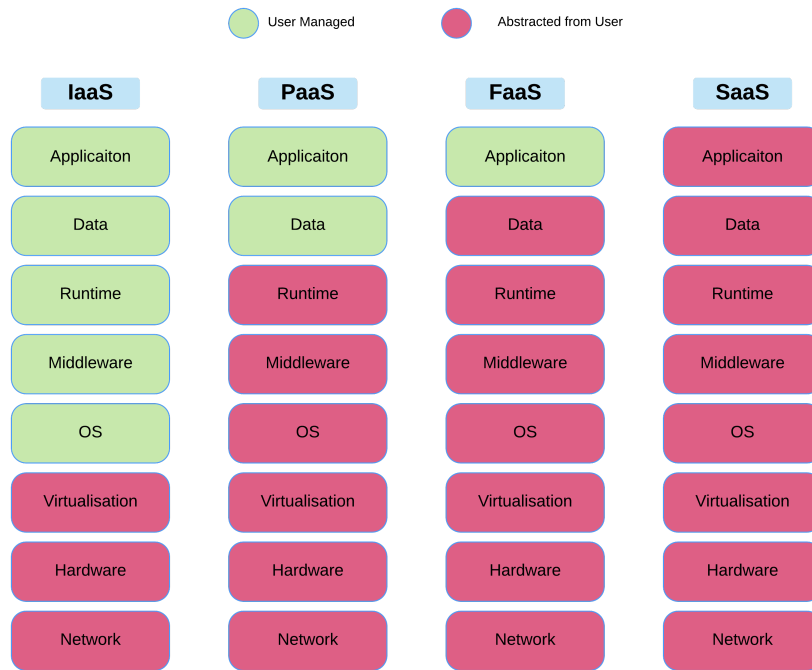

Figure 2. IaaS vs PaaS vs FaaS vs SaaS Cloud Execution Models.

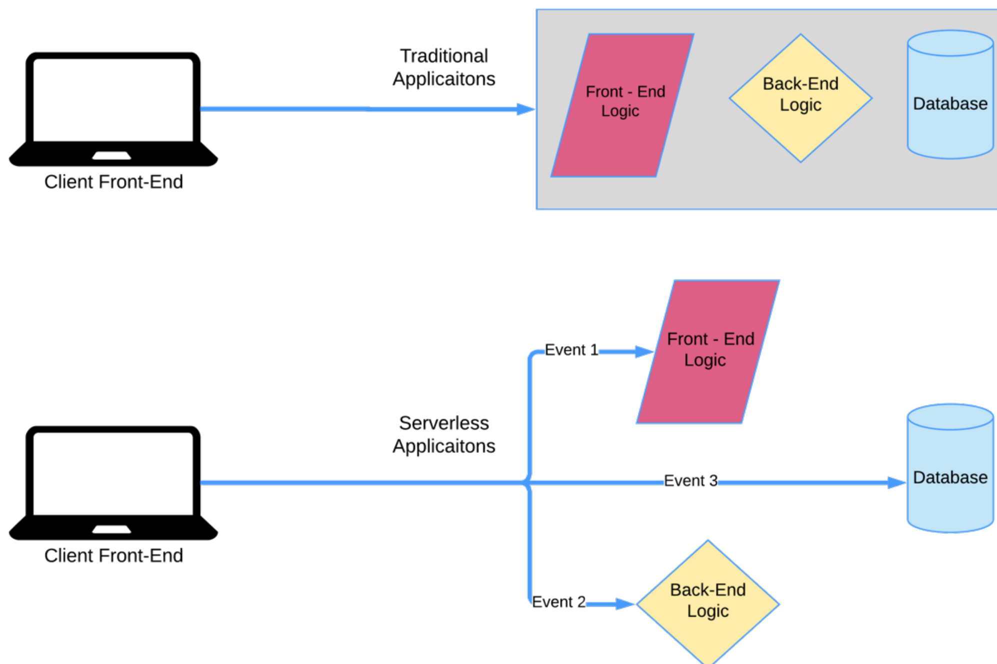

Figure 3. Difference between Traditional applications and Event-based Serverless Applications.

FaaS processing model was first introduced, commercially in 2014, by Amazon as AWS Lambda service and since then, there are a numerous commercial serverless offerings as well as opensource frameworks such as IBM Cloud Functions, Azure Functions, Google Cloud Functions, Knative and Fission. Serverless with its quick and simple offerings, are suitable for certain classes of applications that requires less amount of resources and run for a relatively lower amount of time. A common use case of serverless is HTTP API and highly parallel and sporadic tasks. Tasks like multimedia processing, stream processing or executing business logic in response to a database changes benefit from on-demand elasticity and efficient cost modelling of serverless [21,23].

Table 1. Resource Limits imposed by major Service Providers.

	AWS Lambda	**Azure Functions (Consumpt ion Plan)**	**Google Cloud Functions**	**IBM Cloud Functions**
Memory Allocation Limits (MBs)	128 MB to 10,240 MB	1500 MB	4096 MB	256 MB
Function Timeout	900 seconds	600 seconds	540 seconds	600 seconds
Function Burst concurrency	500 - 3000	600	3000	5000
Deployment size(zipped)	50 MB	N/A	100 MB	48 MB
Invocation Payload(synchronous)	6 MB	100 MB	10 MB	5 MB

Although Serverless or Function-as-a-Service model seems to offer a lot of advantages over the other traditional execution models, it does limit its users or consumer of services in various scenarios. One of the shortcomings of FaaS execution model is the restricted configuration of resources that are allowed for a function container such as limited amount of CPU or memory for the container or the maximum execution time of a function container and the concurrency policies of a function. The different resource limits imposed by various commercial platforms [3,4,5,6] are listed in Table 1. As FaaS model replaces the developer's need to worry about the underlying servers and availability, some developers overlook the application run-time requirements and run into resource limits over time. These limits not only affect the performance of the application but also, negatively impact the developer experience on the platform. Apart from the resource allocation, there is a limited native support for multiple programming language runtimes by different service providers. As the developers prefer to work in a specific programming environment, support for specific runtimes on various serverless platforms emerges as a hurdle and hence a trade-off between the overall offerings of the platforms, in the process

of application development and deployment. The different language runtimes supported by major commercial platforms [3,4,5,6,7] are listed in Table 2.

Table 2. Language Runtime Support for Serverless Providers.

	AWS Lambda	**Azure Functions**	**Google Cloud Functions**	**IBM Cloud Functions**
JS	✓	✓	✓	✓
Go	✓	✗	✓	✓
Python	✓	⚠	✓	✓
Ruby	✓	✗	✗	✓
Java	✓	⚠	✗	✓
C#	✓	✓	✗	✓
PHP	✗	⚠	✗	✓
C++	✗	✗	✗	✗

✗ - Not supported; ✓ - Supported; ⚠ - Experimental Support

When the enterprises move to serverless execution of the applications, pricing plays an important role in the shift from traditional approaches. With many Function-as-a-Service providers in the market, each provider has its unique pricing model based upon the levels of abstraction provided. These abstractions may include offering fully managed services like load-balancers, access to local development tools or providing a CI/CD pipeline integration, unique management of function containers or Virtual Machines (VMs), etc. Hence, different pricing model of industry wide serverless providers suit different application use case. A list of pricing models of major FaaS providers [3,4,5,6] are presented in Table 3.

Table 3. Pricing Model of Function-as-a-Service Providers.

	AWS Lambda	**Azure Functions**	**Google Cloud Functions**	**IBM Cloud Functions**
Requests ($ per 1M requests)	$ 0.20	$ 0.20	$ 0.40	N/A
$ per GB-second	$ 0.0000166667	$ 0.000016	$ 0.0000025	$ 0.000017
Free Tier Requests	1 M/month	1 M/month	2 M/month	5 M/month
Free Tier Compute time (GB-second)	400,000/month	400,000/month	400,000/month	400,000/month

2.1.2 Utility of Function-as-a-Service Model

Function-as-a-Service model abstracts the details of the underlying servers and takes away the responsibilities of tasks related to server management from the developer or consumer of services, shifting it towards the cloud service provider. This encourages enterprises to focus more on the application building rather than wasting resources in application capacity planning that incurs operational overheads. Therefore, there are two primary personas [1] involved for serverless computing: (i) developer or consumer and (ii) service provider. With the emergence of new technologies like Internet of Things (IoT) and edge computing, there is a need of a distributed application architectural style that leverages the resources to the maximum, with minimum effort to worry about the underlying resources. These applications have a characteristic requirement to be highly available and scale on-demand to compensate for the unpredictable workloads [19]. Hence from a consumer perspective, the serverless architecture of applications provide a microservices-style model that allows designing of applications in the form of individual function(s) or a set of related functions to execute business logics. These functions are event driven and are triggered to execute the deployed business logics which are billed on finer-grained pay-per-use pricing model, wherein the consumers only pay for the actual execution time of the function and not for idle times.

On the other hand, cloud service provider is accountable for providing the abstraction of high availability and unlimited scalability of these application functions. The business logic wrapped up as light weight function containers, run for a specified purpose and therefore have limited access to resources. This helps the service providers to efficiently utilise their resources by using multi-tenancy model. This model also helps the service providers to use these resources for other purpose and cover the costs of management, since the service providers still need to pay for maintaining the abstractions during idle times. Every enterprise aims to add value for their customers and deliver a consumer-focused experience. In this process, enterprises usually need a business model that decrease their product's time-to-market while facing uncertainty and pressure from competition. [2] Microsoft estimates that there will be near 500 million new applications

in the next 5 years and it would be difficult for the current development models to support such large expansions. Serverless computing's Function-as-a-Service offering is designed to address these challenges by increasing development agility and decrease costs of ownership and overheads related to servers and other cloud resource. With cloud-based application development, FaaS also makes it possible for the enterprises to use individual mature services and enhance the product's integration capabilities by reducing overheads spent on middleware process.

2.1.3 Challenge of Function Cold Start

Serverless function model, assures high availability and on-demand scalability of the function containers. To service the incoming application requests, platform spawns new function containers, on-demand and a process of initialisation precedes the serving of requests. The initial bootstrapping process of new function container downloads the code, set up the code dependencies and runtime environment, set up networking requirements and once the container is ready, initiates the function handler code to service the request. This process introduces a non-negligible time-delay in the application response, known as 'function cold start'. These function cold starts are typically between 0.2 seconds to few seconds [8]. Function Cold Start can also be understood as the preparation time of the function container before executing the request, when it is spawned by the serverless platform on demand of the incoming request [23].

According to [8]**,** a study conducted on Azure Functions, the application workload times are highly variable and approximately 18% of the applications that are accessed more than once per minute, account for 99.6% of the total request arrivals or accesses. Conceptually, the Function-as-a-Service platforms were designed to execute new function containers for each incoming request, but to increase the resource utilisation modern implementations of serverless keep a queue of idle function instances for a limited period of time [30]. Function idle times of major FaaS providers are listed in Table 4. Hence, to service the incoming request with on-demand scaling, the serverless platform initially checks for the available function containers in the idle queue and if none available, will request a new function container and undergoes a typical delay of cold start. Typically, serverless functions execute lightweight business logic for a limited time period of few seconds and as the container cold starts are in the order of function execution time, it affect the application performance, portraying an illusion of occupied or unavailable server in cases of considerably larger cold start than request's time-to-live.

Table 4. Instance Reuse Strategy for Leading Cloud Vendors.

Service	**Idle Instance Time**
AWS Lambda	5 – 7 Minutes
Azure Functions	20 – 30 Minutes
Google Cloud Functions	15 Minutes

There are multiple factors like runtime environment, size of the deployment, programming language used, resource limits (CPU and memory), workload concurrency, etc. that are known to be responsible for the varying cold starts in the Function-as-a-Service model [51,52]. A recent study [9] compared the typical cold starts on the leading serverless platforms for the most common language runtimes used for a simple 'Hello World' program. It concluded that cold starts on different platforms, for multiple language runtimes, can range from anywhere between 0.2 seconds to 5 seconds in the worst case (Figure 4). The study also targeted the size of function deployment and shows a further increase in cold starts when a number of dependencies are configured for it. They compared a 'HelloWorld' JavaScript function with various number of NPM package references and shows that larger packages introduce a significant increase in the cold start (Figure 5). Therefore, the challenge of function cold start is an inherent characteristic of Serverless platforms that needs to be addressed to improve the application as well as platform performance [34].

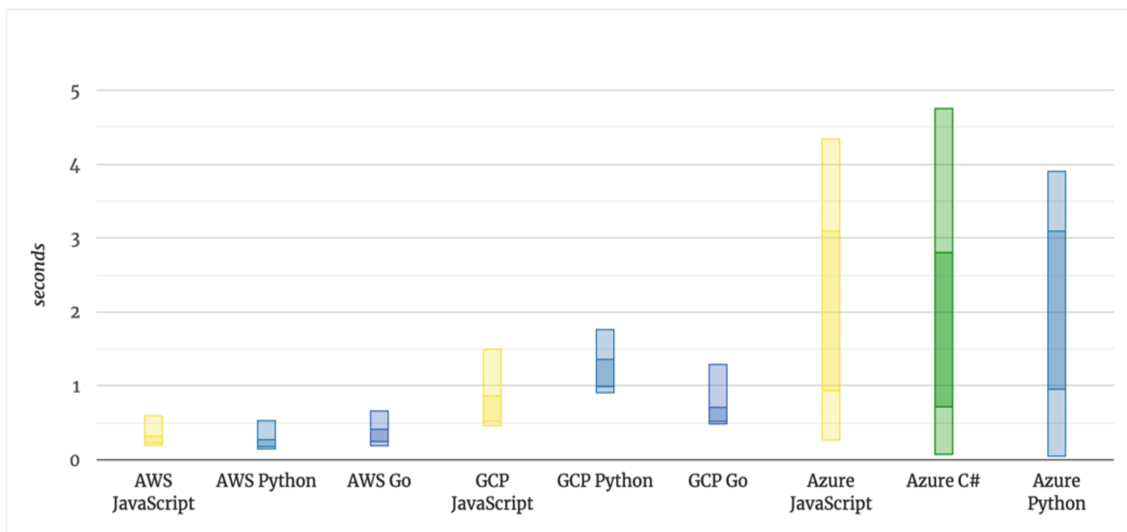

Figure 4. Typical Cold Start for Different Runtimes taken from [9].

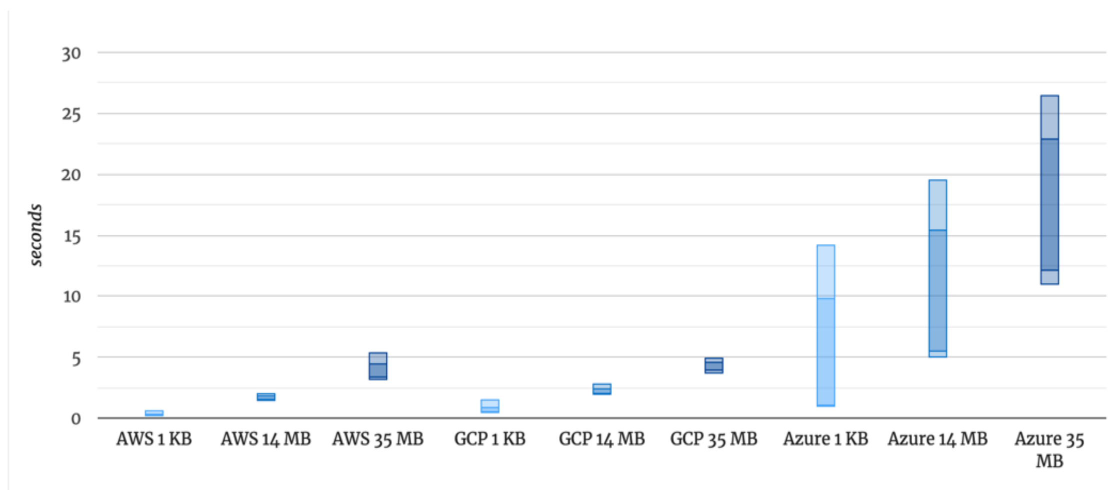

Figure 5. Comparison of Cold Starts for Different Package Sizes (Zipped) taken from [9].

2.1.4 Kubeless – a Kubernetes-native Serverless Framework

With the emergence of Serverless Computing, that promotes the elimination of concerns for server management, provisioning and other resource management tasks, there is an increased dependency on containers or lightweight Virtual Machines (VMs). A container is an application layer abstraction that stitches together the code and its dependencies, in an isolated user space environment. The development of containers and containerized applications have familiarized a need for orchestration tools to efficiently manage the tasks such as container creation, configuration, monitoring and observability, etc. According to the CNCF's Serverless Working Group [1], most of the open-source serverless products such as Kubeless, Fission or Knative are based on Kubernetes – a CNCF container orchestration project [10].

Kubernetes is an open – source, container management platform that facilitates efficient administration, configuration and automation of containerised workloads and services [10][11]. With the container development era providing OS-level virtualisation, Kubernetes a.k.a. K8s puts forward a framework to run applications in a distributed and robust fashion. A variety of services are provided by K8s such as –

- Service discovery and application load balancing.
- Application rollout and rollback automation.
- Container configuration management.
- Deployment fail – over management i.e. Self – Healing.
- Deployment scaling.

Kubernetes' primary function is to deploy and manage a large number of container-based workloads on a fleet of machines, known as Cluster. A cluster is a set of worker machines

called Nodes, that execute the application containers. This allows Kubernetes to coordinate a highly available cluster of component worker nodes, connected as a single unit and abstract the deployment of application containers without tying them to an individual host machine or worker node. Therefore, containerisation of modern applications on Kubernetes allow decoupling from the worker machines, providing deployment flexibility and availability, while efficiently distributing and scheduling the containers over the cluster of nodes. In Kubernetes, containers are generally wrapped up as a Pod, a fundamental unit of computation that is created and managed within a cluster. A Pod is a group of one or more co-located containers that share the pod resources and run in a shared context [12]. To deal with the lifetime activities of pods, Kubernetes further provides a layer of abstraction called Deployment that is responsible for the maintenance of desired state of pods (Figure 6) and therefore a set of pods are executed under this abstraction. With the notion of automating the management of application container, its lifecycle and maintaining its desired state, while providing an isolated runtime environment, Kubernetes emerges as a perfect platform to expand its services with serverless execution model.

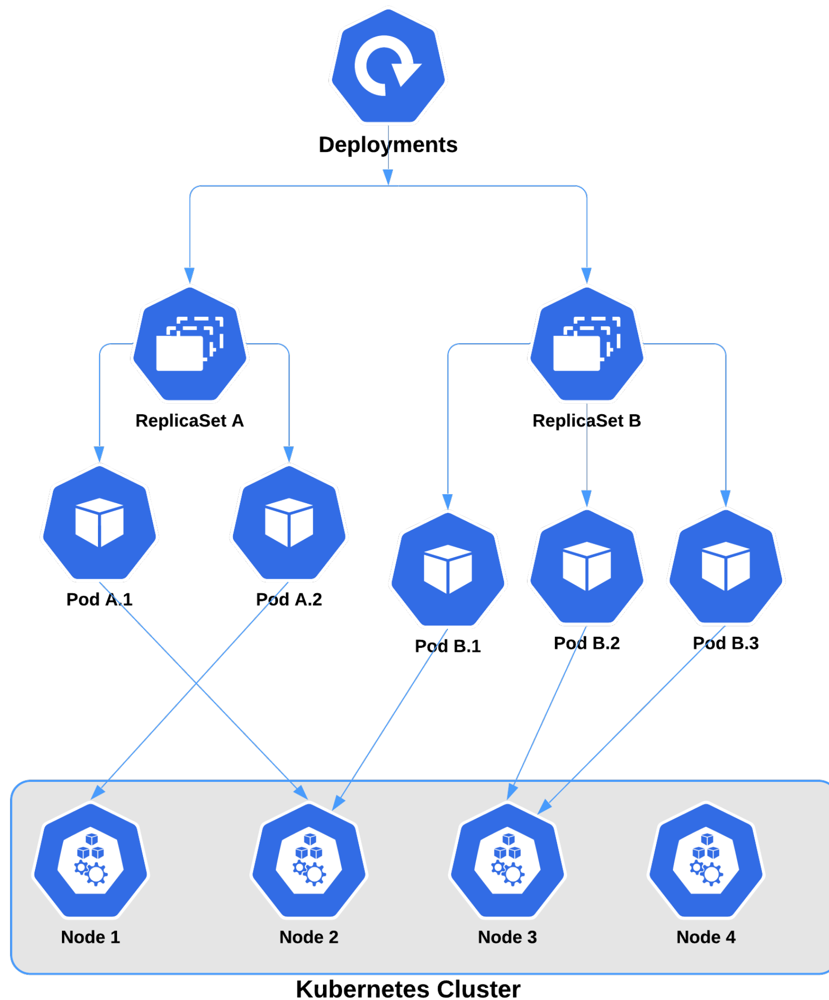

Figure 6. High-level view of Kubernetes Deployment.

Kubeless is a Kubernetes – native serverless framework that lets us execute our pieces of code or application logic as functions, without the hassle of underlying resource planning and management [13,24]. Kubeless is designed to leverage the underlying resource definitions and primitives of Kubernetes and is deployed over the Kubernetes cluster. In the serverless execution model, Kubeless provides services such as create, delete, list functions, autoscaling properties, API routing, monitoring and troubleshooting, etc. Kubeless is implemented as a Kubernetes controller that continuously watches for the changes in the function objects to react accordingly. It is written in Go programming language and uses Kubernetes client-go to connect with Kubernetes API-server for its functioning [13]. The serverless framework is built around three core entities – Functions, Triggers and Runtime.

Runtime refers to the language specific runtime environment in which the functions will be executed. By default, [7, 13] Kubeless supports different runtimes (Figure 7) and every runtime is encapsulated as a container image whose references are configured in the Kubeless configuration files.

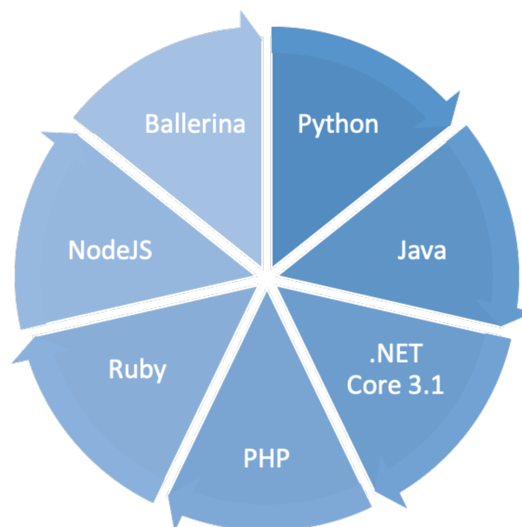

Figure 7. Kubeless Supported Runtime Environments.

Being a Function-as-a-Service model, a Kubeless function is a basic unit of deployment that represents the piece of code to be executed, wrapped as a container inside a Kubernetes pod. A function consists of business logic or code, metadata about the runtime and dependencies and follows an independent lifecycle as per Pod principles. Kubeless supports a range of function methods like deploy, execute, list, get, delete and logs. Functions can be implemented in any supported programming language and follows a generic interface that receives the details of event and its related context i.e. information about the function. In Kubeless, functions are designed to return a string value which will be used as a HTTP response to the source and following the serverless principle of bounded-time execution, it executes a function for a limited period of time (default 180 seconds) that can be programmatically configured [24,53]. A sample function is presented in Algorithm 1, with the Kubeless Command Line instruction to configure and deploy the function in Equation 1.

$$\$\ kubeless\ function\ deploy\ helloWorld\ --runtime\ python3.6\ --from-file\ test.py\ \backslash$$
$$--handler\ helloWorld\ --cpu \quad (1)$$

Algorithm 1: helloWorld
INPUT: $Event, Context$
OUTPUT: $Response$
$Data \longleftarrow Event['Data']$
$Print(Event)$
return $Data$

Once a function is configured and deployed, a trigger is associated to it. A Trigger represents an event source which when occurs, initiates the associated function. Kubeless ensures that the associated function is executed at most once, upon receiving the trigger event. A trigger is handled separately from the lifecycle of a function and supports methods such as create, update, delete and list, independently. Kubeless' architecture has many to many relationships between function(s) and trigger(s) and supports three types of triggers – HTTP trigger, CronJob trigger and Pub/Sub trigger. Since Kubeless inherits Kubernetes properties, a function is usually accessible within the cluster and to provide external routing a Kubernetes supported Ingress controller is required, in case of HTTP trigger. An example of HTTP trigger creation [13] is presented in Equation 2.

$$\$\ kubeless\ trigger\ http\ create\ helloWorld\ --function-name\ helloWorld \quad (2)$$

Kubernetes has introduced Horizontal Pod Autoscaler (HPA) as its default control mechanism to scale up or scale down the deployments within the cluster [10, 14]. It is implemented as a control loop within Kubernetes API resource and checks periodically for the specified target metrics such as CPU or memory utilisation, to adjust the replicas of the pods in a deployment. HPA has a default query period of 15 seconds to check and control the deployment and queries the Kubernetes resource metrics API for the specified metrics. For the per-pod-metrics such as CPU utilisation, the controller fetches the metrics and calculates the average value i.e. currentMetricValue, for the ready pods available under the deployment and if a target value i.e. desiredMetricValue is set, it produces a ratio (Equation 3) to adjust the number of desired pods [10].

$$desiredReplicas = ceil\left[currentReplicas * \left(\frac{currentMetricValue}{desiredMetricValue}\right)\right] \quad (3)$$

Just before the controller scales to the desired replica set, the scale recommendation is recorded and a maximum recommendation within the period is chosen. To prevent the resources from thrashing i.e. fluctuating number of pods based on current metric value, Kubernetes keeps a default downscale period of 5 minutes, to gradually scale down the deployment.

As discussed, in Kubeless functions are executed under the abstraction of a deployment, it benefits from the underlying offering of Horizontal Pod Autoscaler to automate the function scaling, based on a specified metric threshold. The HPA controls the replication of function pods once the scaling rule is set and the default metric supported by Kubeless is CPU utilisation referred to as 'cpu'. To support the metrics based autoscaling, Kubeless requires the function to be deployed with resource limits and refer to 'cpu' metric as the percentage of average CPU milli-core used out of requested, across all available function pods. The command line instruction in Equation 4 is used to deploy the autoscaling rule with 'cpu' metric threshold for a sample function deployment [13].

$$\$\ kubeless\ autoscale\ create\ helloWorld\ --\min 1\ --\max 10\ --metric\ cpu\ --value\ 50 \tag{4}$$

The above command will maintain a minimum of 1 function pod and scale to a maximum of 10 pods, while trying to maintain an average CPU utilisation of 1000 milli-core (50% of requested using Equation 1) across all the available function pods.

2.1.5 Reinforcement Learning: Q-Learning Technique

Reinforcement learning (RL) is an area of machine learning that involves training of machine models to make a sequence of decisions without direct supervision. It is the science of decision making and analogous to thinking in an optimal way or the idea of maximising the related cumulative reward. In a RL world, the environment is modelled as a Markov Decision Process (MDP), where it is expected to produce a stochastic reward and observe a stochastic change of state [17,28]. Markov Decision Processes can be understood as the environment model where all the states follow Markov property which states that 'the future is independent of the past given the present' [15]. In simple terms, it generalises that the current state holds all the relevant information about the past, to help the agent make the next decision and usually associates a transition probability with the states. We model the environment state space as a set of states $S = \{ s_i \mid i = 1,2, \ldots .n\}$ that the agent transitions between, by performing actions. Actions are part of the available action space $A = \{a_i \mid i = 1,2, \ldots .m\}$ that help the agent to move to a new state s_{t+1} at time *t+1*. The associated probability of transition from state s_t to state s_{t+1} can be correlated as $P(s_{t+1} \mid s_t\ , a_t\ )$ and this transition at a discrete time step yields a stochastic immediate reward that can be described as $R(s_t, s_{t+1})$.

The agent in the RL world directly interacts with its environment at discrete time steps by determining its current state s_t and choses an available action a_t. The environment responds to the performed action and transitions to a new state s_{t+1} at next discrete time step and the agent observes an immediate reward r_{t+1}associated with the transition. Therefore, the agent with its state-action map and rewards aim to learn a correlated policy $\pi(a, s) = \Pr(a \mid s)$. The main objective of reinforcement learning agent is to learn this optimal action policy to maximise cumulative reward function [15,17,18]. Value functions are one such useful methods that attempts to find the optimal policy to maximise expected rewards [16].

Q-Learning is a model-free, iterative Reinforcement Learning algorithm that makes use of Q-values to continuously learn and improve the efficacy of the actions. It is classified as a model-free learning algorithm because it does not require the transition probabilities associated with states to learn the optimal policy [18,29]. During the training period, the Q-Learning agent acts in the environment using possible actions and transitions between different states, observing rewards for the purpose of gaining information about the environment and learning optimal actions. The Q-value is a function that determines the quality of the action performed and is defined for each state-action tuple i.e. (s, a) pair. These values are stored in the form of a Q-Table mapping i.e. $\pi: Q \rightarrow S \times A$ and serves as a look-up table to decide a good action. Once the agent observes the immediate or the transition reward, the Q-values are updated according to the Bellman Equation as a value-iteration rule that uses the weighted average of old and the newly observed information (Equation 5).

$$Q^{new}(s_t, a_t) \leftarrow Q^{old}(s_t, a_t) + \alpha * (r_t + \gamma * \max_a Q(s_{t+1}, a) - Q^{old}(s_t, a_t)) \quad (5)$$

where –

- α is the learning rate ($0 < \alpha < 1$), that determines the useful proportion of newly obtained information.
- γ is the discount factor ($0 < \gamma < 1$), that determines the importance of the future expected rewards.
- r_t is the immediate reward for the transition from state s_t to s_{t+1} and is weighted by the learning rate i.e. αr_t to account for new information.
- $Q^{old}(s_t, a_t)$ is the old information for the current state and it is eventually weighted by the learning rate i.e. $(1 - \alpha)Q^{old}(s_t, a_t)$ to determine the current value.
- $\alpha * \gamma * \max_a Q(s_{t+1}, a)$ is the maximum future reward that the agent can expect to observe as per the Bellman Equation (temporal-difference learning). This helps the agent to capture the information that is expected to be observed from the next state and guides the agent to select highest return action at any given step.

In the process of learning, the agent makes a trade-off between exploration and exploitation, instead of taking random actions at each time step. This helps the agent in improved utilisation of obtained state-action information. One such policy is $\mathcal{E}$ - Greedy, where $0 < \mathcal{E} < 1$ is a controlling parameter for regulating the extent of exploration and exploitation for the agent [16,18]. The agent greedily decides to choose an already explored and believed best action, by choosing the maximum Q-value action against the state with a probability of 1 - $\mathcal{E}$ and thus exploits the acquired knowledge of the environment. On the other hand, with a probability of $\mathcal{E}$, the agent chooses to explore other available actions for a state, reflecting the ability of the agent to decide an optimal action policy.

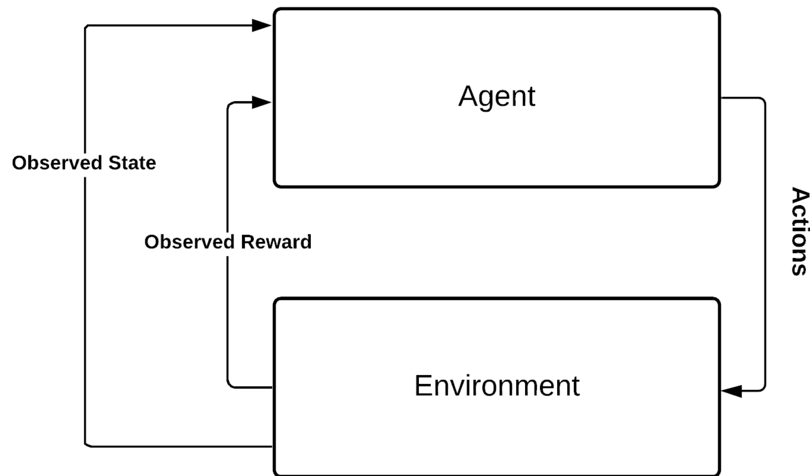

Figure 8. Simple Interaction Diagram of Q-Learning Agent.

To illustrate, Q-Learning can be visualised as a robot [28] that interacts with the totally unknown environment by performing some allowed actions being in a particular state i.e. a situation in the environment. While doing that, the agent may receive immediate rewards for the actions or delayed rewards which it might expect to observe in future, and transition to a new environment state. These rewards assist the robot in determining the correctness of an action from a particular state and with trail-error method it gradually learns to figure out the best possible actions to perform its tasks or an optimal way to act in an environment. A simple Q-Learning agent interaction loop is represented in Figure 8 and highlights the continuous process of acting in and observing the environment.

Chapter 3 - Literature Survey

This chapter presents an extensive overview on some of the existing research work related to Serverless Computing and function cold start. It lists various cold start optimisation approaches based on different factors and helps in highlighting the gap between the current studies and the proposed Reinforcement Learning based approach.

3.1 Literature Review of Existing Works

The evolution of Cloud Computing services has been made possible over the years because of the vital technologies like distributed systems, virtualisation, service-oriented architectures and utility computing [41]. Cloud computing introduces itself with a set of advantages such as elimination of upfront costs for resources, ability to scale on-demand or ability to pay for the use of resources for short term periods, etc. Even with the establishment of virtual machines that offer an opportunity for enterprises to switch to cloud computing, the users of services are still burdened with the complex activities of managing the resources themselves. The complexity of the responsibilities such as fault tolerance, consistent replication of services, efficient scaling of resources or system updates and migration in the cloud services demanded a simpler and newer model of execution. [20] discusses the potential of a newer model of Serverless Computing that was introduced by Amazon and catered to the requirements of less complex model. The study briefly introduces a serverless, function-based, commercial offering of AWS Lambda and discusses the primary differences between the traditional *serverful cloud computing* and *serverless computing*. The researchers identify the critically different characteristics between the two execution models and appreciates the simplification of application development and ease of resource use in the serverless computing model. According to the study, there are three critical distinctions between the traditional computing and serverless architecture – (i) decoupled computation and storage i.e. the independence of computing and storage services in terms of provisioning as well as pricing, (ii) code execution without resource management i.e. the user primarily focuses on the code and the cloud platform takes care of resource related tasks, and (iii) paying according to the resource usage instead of allocation. The researchers posit that the serverless computing model promotes business growth, making the use of cloud easier while attracting new users for a variety of popular application use cases such as Web API, data processing, integration of services, etc. The study continues to explore different application models to address the limitations of serverless computing and lists various observations of performance bottlenecks on commercial serverless offerings. They identify potential drawbacks in the serverless model and describe the possible future research directions to address security challenges, system challenges like start-up times or affordability, networking challenges and architecture challenges. The researchers conclude the study with their prediction of serverless services with an expectation to address the drawbacks.

A study [21] conducted at IBM Research, USA, investigates the current trends and potential challenges in the serverless computing architecture. The authors introduce the emerging paradigm of serverless computing as an application development architecture and programming model that allow pieces of code to be executed in cloud without the control over underlying resources. They postulate that the emergence of microservices architecture and use of containers has led to a gain in popularity of serverless. They further define serverless as a 'stripped down' programming model that executes stateless functions as its deployment unit in Function-as-a-Service (FaaS) offering. Among the traditional cloud execution models, the study puts Serverless on top of Platform-as-a-Service (PaaS), where the developer has no knowledge and control of the resources and in between Infrastructure-as-a-Service (IaaS) and Software-as-a-Service (SaaS). The study makes progress by surveying various serverless platforms and determines distinct characteristics to distinguish between them. It recognises factors such as cost, performance and limits, programming language, security and monitoring that a developer should consider while making an informed decision of selecting a serverless platform. In the similar context, the researchers highlight the benefits of serverless computing over traditional models, such as abstraction of underlying resources from a consumer perspective – allowing developers to focus on business logic or a stateless execution that empowers the service providers to distinctly manage the software stack. However, the serverless platforms are known to inherently put forward a trade-off between the ease of services and the constraints of serverless programming model that limits the developer capabilities and raises a challenge of vendor lock-in. The authors discuss various challenges of serverless under two categories – (i) system-level challenges where they identify costs of services, cold starts while scaling, security and resource limits, etc., and (ii) programming model and devops challenges that focuses on tools and IDEs, deployment, statelessness and code granularity in serverless model. They finish their discussion by posing a number of open problems in serverless computing and frame them as research questions for further investigations.

Since the inception of serverless computing, there has been many commercial and open - source offerings such as AWS Lambda, Microsoft Azure Functions, Google Cloud Functions, Fission and OpenWhisk, etc. They generally identify serverless computing as an emerging technology but [22] put together gaps that furnish serverless as a bad fit for cloud innovations. The authors criticise the current developments the domain of cloud computing by stating that we are yet to harness the potential of cloud resources. They assess serverless computing in terms of the services offered by different vendors and lay out their reasons of serverless being a disappointment towards cloud's actual potential. The analysis leverages services from AWS Lambda and draw their conclusions for three types of function interaction patterns – embarrassingly parallel functions, orchestration functions and function composition. It identifies serverless constraints such as limited lifetime that encourages cold starts, statelessness, no specialised hardware use and inter-function communication through slow storage since the functions can't communicate directly. The authors form different case studies to highlight shortcomings of the serverless platforms and present the objections raised by their subjects, in reference to the developments of serverless, with an aim to spark real innovation in data-rich cloud systems.

On the contrary, [23] emphasises that serverless offerings are economical and affordable as they remove the responsibility of resource management and complexity of deployments from the consumers. The study visits the various drawbacks and limitations

of serverless in a positive way to improve its utilisation and proposes possible future directions. The researchers define the serverless as an intersection of Function-as-a-Service (FaaS) and Backend-as-a-Service (BaaS) that follows a set of serverless characteristics. They discuss the opportunities offered by multiple serverless offerings and attempt to categorise a set of applications, with varying workloads, to leverage serverless services. Among various investigated serverless challenges, the study sheds light on the scheduling problem from a perspective of short-lived function instances. It states that service providers stop the execution of resources after a period of inactivity and upon waking up or restoring the resources, there is some service latency that reduces user's quality of service. These periods of activity are a characteristic of application invocation and configuring a same sleep timer for different applications is naïve. Therefore, the researchers suggest approaches like warm queue of a minimum functions or data probing, where execution of one function instantiates the other through data manipulation actions, or prediction and forecasting of requests. In relation to prediction, they propose the use of machine learning models with relevant variables to extract the patterns of activity and allow appropriate resources to be consumed during the activity period. They expand their research with an overview of other existing challenges and conclude by indicating potential approaches for future work.

Serverless computing - featuring affordability, on-demand scalability and light-weight containerization, comes with its inherent challenges and problems. These challenges can broadly be listed as security, privacy, caching, modes of execution, etc. Among them, the problem of cold start is still prevalent and has attracted academia for realising possible solutions. A current study [30] discusses the ongoing trends of handling the cold starts in commercial as well as open source serverless platforms and present their results by evaluating AWS Lambda offerings. The study presents a brief distinction between the virtual machine and container-based application models and connects Function-as-a-Service to the concept of serverless computing. They describe the problem of function cold start as the time taken to execute the function that involves following steps – (i) assign a container to function, (ii) access the function package and copy the function image on container, (iii) load the image into memory and unpack it and (iv) execute the function handler. They broadly categorise the approaches to deal with cold starts in two classes: (i) Optimising environments i.e. minimise the cold start delay itself and (ii) Pinging i.e. minimising the frequency of cold start occurrences. They address the approach of optimising environments either by reducing container preparation delay or reducing the delay in loading function libraries and review the offerings of OpenFaaS, OpenWhisk and AWS Lambda to discuss the solutions like cold and warm queues, application sandboxing or preloading function libraries. To adopt the pinging technique, the study investigates third-party tools such as CloudWatch or Lambda Warmer to continuously monitor the functions and schedule a job to periodically ping the functions to keep them alive. They further create a case study with I/O intensive and CPU intensive benchmarks for evaluating the AWS Lambda's warm queueing technique and perform a set of tests with different configurations. From the experimental evaluations, the authors conclude an inconsistency between the number of cold starts experienced and number of requests, with the absence of any correlation between the warm containers prepared by the platform and time interval of incoming requests.

In [31], adaptive function container warm up techniques are introduced to reduce the cold start latency. The study states that the cold starts affect the application responsiveness severely and investigates the existing optimisation techniques like container pool-based

strategy, function runtime optimisation and isolation of different functions, etc. These techniques are found to be useful at the expense of the resources and hence the researchers suggest a time series-based prediction model to reduce cold start latency. They propose two strategies – (i) Adaptive Warm-Up (AWU) technique and (ii) Adaptive Container Pool Scaling (ACPS) technique. AWU strategy utilises a function chain model, i.e., a sequence of functions to predict the function invocation time using LSTM networks, and non-first functions to keep the warmed function containers ready in queue. They leverage function chain model to improve the prediction accuracy of the model. The researchers also propose a container pool strategy, ACPS that seeks to dynamically adjust the number of empty containers in the container pool to reduce the waste of resources. Both approaches work in synchronisation as the failure of adaptive warmup strategy will automatically launch adaptive container pool strategy, by providing a pre-warmed empty container, thus reducing the overall cold start latency. It is highlighted in the study that even though the strategy learns the invocation time of the function chain, the first function in the sequence suffers cold start latency. They test their approaches by comparing the resource utilisation, idle time and overall cluster utilisation with other existing techniques.

Researchers in [32] explain the phenomenon of cold starts with respect to the Knative serverless platform and suggest a pod migration technique to reduce the cold start of the function containers. They posit that the cold start overhead is dependent on the underlying implementation of the function and since Knative leverages the concept of containers, they categorise overheads as platform dependent and application dependent overheads. The platform overheads such as network bootstrapping, pod provisioning, etc., are responsible for introducing a uniform delay across all function containers while application dependent overheads vary according to application implementation. To deal with the cold starts, a pool of pre-warmed containers, marked with selector 'app-label', are kept ready. When the requests arrive, first the pool is checked for existing pre-warmed containers and allocated to the application, otherwise new containers are spawned as per the request workload. The authors compare their proposed solution with existing pool-based techniques and comment on their limited effectiveness in reducing cold starts. They further attempt to structure the problem of function cold starts similar to other scientific models like relating pre-warmed containers to prefetching involved in caching, a similarity with knapsack problem where different function with values are used to optimise total function resources or a stochastic inventory model similar to a multi-product model with costs. The study concludes with an improvement in the cold start latencies of the containers for a single instance of pool and proposes subsequent improvements in the current technique.

Another research [33], studies and exploits the data similarity for reducing the cold starts and proposes a deployment system over a peer-to-peer network, virtual file system and content addressable storage to increase the computing capabilities, storage requirements and prevent network bottlenecks of system. They criticise the current container deployment technique of pulling each new container image from the storage bucket and introduces a live container migration technique over a peer-to-peer network. They analyse three different ways of application deployment i.e. scaling, versioning and live-migration, and propose to transfer blocks of files containing frequently used libraries and packages, over the network when required. With the proposed technique, researchers found a 37.9% reduction in the boot-time of containers. Similarly, [34] aims to reduce the number of cold start occurrences by utilising the function composition knowledge. It presents an application side solution based on a light-weight middleware that aims to enable the

developers to control the frequency of cold starts by treating the FaaS platform as a Blackbox. It establishes that applications are generally deployed as a set of functions and proposes three strategies; naïve approach, extended approach and global approach where a dedicated orchestration component invokes all the steps and follow a process of 'hinting' the next batch of functions involved. These techniques can be co-deployed alongside the serverless functions through a designated middleware and are shown to reduce cold start occurrences by 30%-40% by incorporating a low monetary cost.

Research in [35] explores network creation and network initialisation as the prime contributor to the cold start latency. It states that cold starts are caused due to work and wait-times involved in various setup processes like initialising networking elements. Based upon the investigations, the researchers posit that the function cold starts are independent of function and are affected by container start-up process. The study explains four stages of container lifecycle: (1) service invocation, (2) start-up, (3) run time and (4) clean-up. The analysis shows that the time taken in creating the network namespaces and initialising them, worsens with the increase in concurrency and contributes immensely to the start-up stage. The clean-up stage includes stopping the container, disconnecting its network and destroying it and this process demands cycles form the underlying containerisation daemon, hindering with the other three processes. Thus, a pause container pool manager is proposed to pre-create a network for function containers and whenever required, attach the new function container to configured IP and network. Their evaluation on OpenWhisk platform demonstrates a reduction of up to 80% in the cold start times with a negligible memory footprint.

Work in [40] introduces the paradigm of Reinforcement Learning (RL) to the serverless platforms. It is focused towards provisioning VMs or containers on request-based autoscaling in the serverless offerings. The study is conducted using Knative serverless platform that supports parallel processing of requests per-instance, utilising the Horizontal Pod Autoscaler of Kubernetes. After performing extensive analysis of different workload profiles, the researchers demonstrate an association of latency and throughput on concurrency levels and suggest an improvement using adaptive scaling policies. The researchers also show that depending upon the workload, different concurrency levels of the container can influence performance and thus, propose a RL based model, specifically model free Q-Learning, to determine the optimal concurrency levels for individual workloads. It evaluates the performance of Q-Learning model, based on latency and throughput of the function containers and demonstrate the capability of applying Q-Learning algorithm to the task of auto-scaling in serverless platforms. The study concludes by highlighting the limitations of the research and comment on its generality and suggests further appreciation of the proposed approach to help in the performance analysis of individual resource usage, for the task of auto-scaling.

Research [22,23,36] has identified various factors like runtime environment, CPU and memory settings, dependency setting, the effect of concurrency, networking requirements, etc. that affect the cold start of a function. Most works [9,19,46,48,54] focus on commercial serverless platforms like AWS Lambda, Azure Functions, Google Cloud Functions and fall short to evaluate open source serverless platforms like OpenLambda, Fission, Kubeless, etc. Very few studies [37,38,39] have successfully performed analysis on open source serverless platforms and provided possible solutions by targeting the container level finer-grained control of the platform. As a novel approach, we explore the applicability and capability of RL strategies to reduce the function cold

start in a serverless environment. Contrasting to the existing works, we apply the model free Q-Learning algorithm for reducing the cold start occurrences, by identifying the invocation patterns of the specific workloads that are the primary source for requesting functions and focus towards learning the appropriate number of function instances. We plan to explore an open-source serverless framework and evaluate it against the non-intelligent, default auto-scaler strategy responsible for cold starts on the serverless platforms. A summary of few discussed researches and our methodology is presented in Table 5, highlighting the distinguishing parameters of individual studies.

Table 5. Summary of few relevant works.

Parameter	**Related Work**						**Our work**
	[31]	[32]	[33]	[34]	[35]	[40]	
Open Source Platform	✓	✓	✗	✓	✓	✗	✓
Commercial Platform	✗	✗	✓	✓	✓	✓	✗
Function Invocation Pattern	✓	✗	✗	✗	✗	✗	✓
Reinforcement Learning Technique	✗	✗	✗	✗	✗	✓	✓
Pre-Warmed Containers	✓	✓	✗	✗	✗	✗	✗
Other Techniques (Network creation, Migration, etc.)	✗	✗	✓	✓	✓	✓	✓
Cold Start Frequencies	✗	✗	✗	✗	✗	✗	✓

Chapter 4 - A Reinforcement Learning Approach to Reduce Serverless Function Cold Start Frequency

Serverless functions are requested by the platform, on-demand, as the number of incoming requests exceed the available function resources to serve the workload. This process initialises set-up of new function containers and introduce a non-negligible start up time or response delay, that is known as cold start. Various non-intelligent, resource binding solutions have been explored and employed for inherent cold start problem. However, Reinforcement Learning (RL) has shown potential and advancements in optimisation problems for different domains. This chapter proposes a RL-based approach to address the problem of cold start and is focused towards reducing the number of cold starts on the platform by realising the application invocation patterns. We model the problem of cold start according to RL framework and introduce an Epsilon-greedy policy assisted by heuristics, to aid the learning process. Experiments conducted on open source serverless framework and synthetic workload patterns show acceptable results compared to default setting of Horizontal Pod AutoScaler, offered by the serverless framework used.

4.1 System Model

The overall deployed system architecture is depicted in Figure 9 and the system model is presented in Figure 10**.** To realise the problem setup of function cold start on the Serverless or Function-as-a-Service platform and perform the relevant experiments, a Kubernetes service cluster is setup using Melbourne Research Cloud (MRC) in collaboration with NeCTAR services at The University of Melbourne, Australia. The main components of the experimental setup are –

- **Kubernetes – Kubeless service cluster:** Kubernetes [10] is an open-source container orchestration and management tool that builds upon the automated services such as deployment, updating, scaling, self-healing, etc. Kubeless is a Kubernetes native, open source serverless framework that helps in building applications in Serverless fashion on top of the services and primitives provided by the underlying orchestration platform [13]. For the purpose of experiments, Kubernetes, version v1.18.6, is configured and a service cluster of 4 nodes with NeCTAR Ubuntu 18.04 LTS (Bionic) amd64 [v30] OS image, 4 vCPUs and 16 GB RAM and 30 GB disk storage, is set up to accommodate all the assets of the deployment. The service cluster is installed with Kubeless, version v1.0.6, to provide serverless services with an implementation to support minimum scaling to 1 function pod or instance. To support the automation in the Kubernetes cluster setup over the cloud services, Ansible scripts (Infrastructure-as-a-Code) automation tool is used.

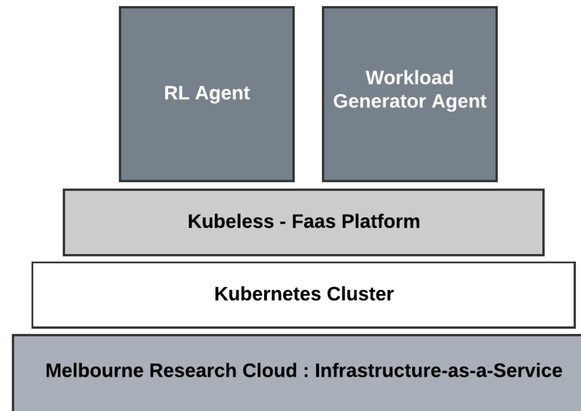

Figure 9. Deployed System Stack Architecture.

- **NGINX Ingress Controller:** By default, there is an isolation between the deployed function instances and the external network. Therefore, a communication channel is required to communicate with the services running in the function, known as Ingress. To setup this ingress and provide the ingress load balancing services i.e. incoming request load balancing on the configured service cluster, NGINX Ingress Controller [25], version v1.10.0, is installed as a Kubernetes deployment. Ingress controller, in turn, is responsible for Kubernetes Ingress resources for respective function deployments and providing the abstraction of a function endpoint and avoid any performance issues related to request load balancing.

- **Apache JMeter Non – GUI Agent:** In this work we are focused towards analysing the application workload pattern for the purpose of reducing function cold starts and thus to mimic and generate a synthetic workload, Apache JMeter [26] is used. It is a JAVA based tool designed to load test the functional behaviour of the web applications and allows to mimic the desired load for a serverless use case application. The Apache JMeter Non-GUI toolkit [27], version 5.3, is installed on a separate worker node outside the Kubernetes service cluster to avoid any interference with the learning process of the proposed Reinforcement Learning agent. All the workload is directed towards the endpoint provided by the Kubernetes Ingress resource for the respective function deployment.

- **Reinforcement Learning Agent:** The proposed Q-Learning agent is configured on the Master node of the Kubernetes service cluster. The agent directly interacts with the Serverless or Function-as-a-Service environment through different available actions. The actions are determined based on the maximum allowed function instances and the agent accordingly chooses to increase or decrease the number of running function instances. The Q-Learning agent transitions between various environment states that are a combination of relevant metrics like CPU utilisation and observes some reward for the associated action. These rewards are useful in determining the appropriateness of the performed action from a particular state and thus helps the agent in exploring or exploiting new and useful information. The agent uses a Q-Table to store and update the rewards that are

used for the learning process [29]. But to assist the agent and speed up its learning process, agent uses a heuristic based previous reward comparison to negatively or positively reward the actions. The agent scrapes the relevant metrics with the help of Kubernetes Command Line tools and directly integrates this information to ascertain the dependent rewards and regulate environment states. Therefore, the Reinforcement Learning agent, along with the proposed heuristics, ascertains the appropriate number of function instances to provision for successfully reducing the cold starts as well as reducing the failed number of responses.

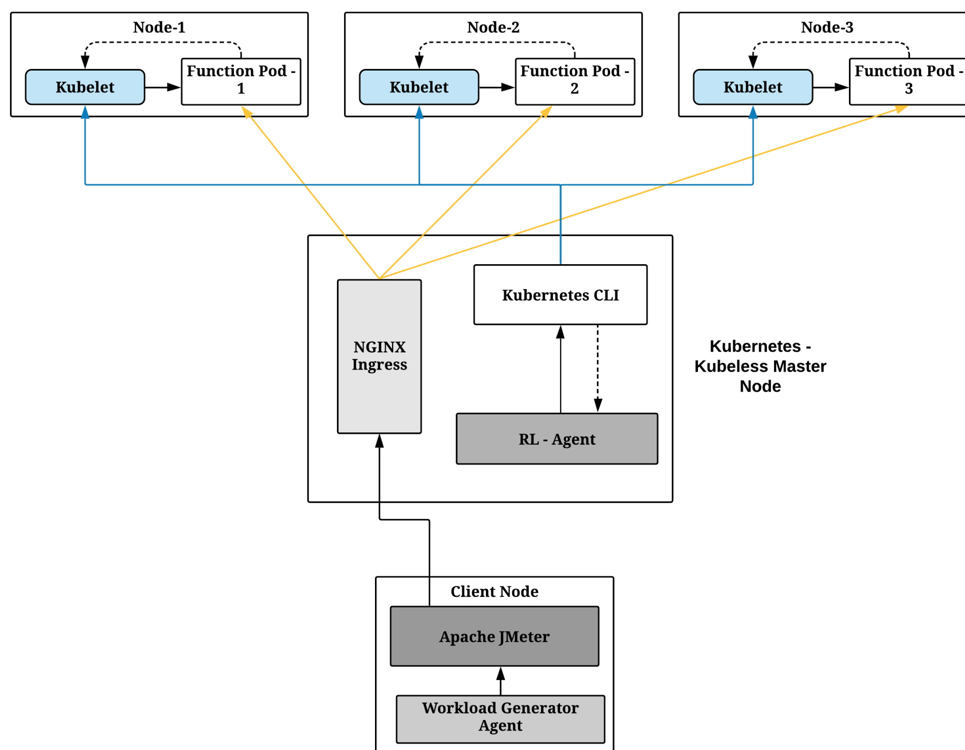

Figure 10. System Model.

4.1.1 Workload Model

FaaS platforms with their features such as ease of deployment, a fine-grained pricing model, focus on business logic and on-demand scaling, appears as an appropriate choice for a variety of applications. Applications that require higher levels of concurrency, account for infrequent request load or are highly dynamic in their demand, etc., are the ideal use cases for serverless model. Therefore, REST APIs, multimedia processing, batch jobs, CI/CD pipelines, etc., benefit from the underlying characteristics of serverless execution model. Therefore, to investigate the problem of function cold start and account

for the characteristic infrequent workload pattern of serverless model, we generate a stable infrequent workload from a defined quota of requests. The infrequent yet set quota of fabricated requests allow us to control the level of workload concurrency, based on the available resources for the experiment, uniformly distributing the pressure on the underlying resources to avoid any bottlenecks in learning. The request simulation uses the thread '*sleep*' method that enables the service cluster to serve quota of requests for a set time span and provide the RL agent with a delayed feedback/reward.

Algorithm 2: FibonacciSum

```
INPUT: Integer N
OUTPUT: Integer SUM
BEGIN
if N == 1 or N == 2 then
  | return 1;
else
  | return FibonacciSum(N − 1) + FibonacciSum(N − 2);
end
```

The abstraction of unlimited scaling for serverless applications suffer from container start-up time or cold starts. Some of the serverless applications are compute intensive and demand a considerable amount of resources such as CPU, memory or time-to-execute. These factors add to the problem of frequency of cold starts on the platform by keeping the available function instances or resources busy, while requesting new function containers for the infrequently arriving workload. Therefore, we put together a compute intensive process of Fibonacci sum calculation up to number 30 [36,45] in order to keep the underlying resources busy and realise the desired real-time behaviour of the serverless application. We implement the recursive model of Fibonacci sum calculation (Algorithm 2) in the function handler, responsible for serving the requests, with appropriate resource requirements to fit the serverless platform constraints. As Kubeless does not cite its concurrency policies [13], we specify the resource requirements in Table 6, such as CPU, memory requirements and time-to-execute, to be allocated for the purpose of evaluating resource metrics. The CPU intensive nature of the function workload also aids the evaluation of default autoscaling policy, by accounting for considerable number of function cold starts and analyse its performance against the proposed solution. Also, the experiment with CPU intensive function handler enables the RL agent to extensively capture the state of the serverless environment for learning the necessary function instances to lower the cold starts.

Table 6. Function Handler Configuration.

Function Handler	***N***	***Runtime Requirement***	***Execution Timeout***	***CPU Request***	***Memory Request***	***Service on Port***
Fibonacci Sum(N)	30	Python 3.7	120 seconds	250 milli-cores	64 MB	8080

Therefore, following the adapted workload model, the request generator is used to simulate a quota of parallel HTTP user requests against the specified Fibonacci function. We use Apache JMeter in the non-GUI mode [27], to concurrently send a number of requests at a variable rate over a period of time. JMeter features a configurable thread '*ramp-up*' period that tells JMeter how long to take for creating the desired number of request threads [26]. In our study, a set of requests are sent from the quota of 1500 requests with a ramp-up period of 250 seconds, engaging sufficient amount of resources from the function instances. This guarantees the demand for newer instances from the default auto-scaler, providing sufficient time for scaling or acknowledging the RL-based agent to analyse the workload pattern, observe the environment states and generate the rewards which complement the function cold start evidence. The generated infrequent workload pattern is presented in Figure 11.

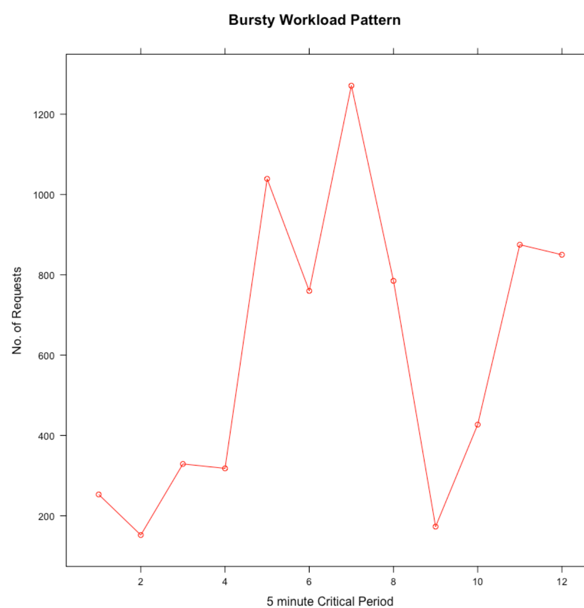

Figure 11. Fabricated Workload Pattern.

4.2 Reinforcement Learning Approach

When modelling an optimisation problem as an application of Reinforcement Learning, the primary goal is to prototype the problem environment. The environment in Reinforcement Learning is composed as a Markov Decision Process [15,28] i.e. the future environment state observed by the agent is independent of the past states, given the present state information. Therefore, we describe the structure of our Reinforcement Learning world that is leveraged by the Q-Learning agent to reduce the frequency of function cold starts on FaaS platform.

- **Environment**

In this thesis we are targeting the problem of cold starts on the serverless platforms dealing with infrequent workloads. For the purpose of our study we set up a simulated environment to mimic the real-time actions of the FaaS platforms [43]. The Kubeless – Kubernetes service cluster along with its components and resources, forms the environment for our RL agent, that provides serverless services. The simulated environment also includes host nodes, Python function deployment and JMeter workload generator, that serves as the foundation of learning process. To provide RL agent with observability of relevant resource metrics and interact with the environment setup, we also incorporate Kubernetes Command Line Interface that allows direct communication with the underlying Kubernetes API resource server.

- **State Space**

Through this study we attempt to deal with the challenge of frequent cold starts on the FaaS platform and observe the effect of characteristic sporadic demand to reduce them. Therefore, following the Markovian property of the environment states, we recognise the suitable factors that are significant in such analysis. The factors include the number of function instances available i.e. $P = \{1,2,3, \ldots . N\}$, to serve the incoming workload, the average CPU utilisation of the instances i.e. pressure on function pods during service into discrete categories i.e. $\%CPU\ utilisation\ (C) = \{\, n * 20 \mid n\ \in [1, 5] \,\}$ and a discretised response factor of failed number of requests during the analysis period i.e. $\%\ Failed\ (F) = \{\, m * 2 \mid m\ \in [0, 50] \,\}$. In an iterative learning method like Q-Learning, we observe the agent for a specific period of time called *iteration period* (I_t) while examining its performance for equally distributed critical time periods called *timeframes* of *T* duration i.e. $T = \left\{ \frac{I_t}{k} \,\middle|\, k\ \in [1,2,3 \ldots \ldots n] \right\}$; where *k* is the number of timeframes. Hence, we extend the learning of the agent over the iteration in multiple timeframes and use these critical time periods in the state formation to capture the exact state of the environment. Since we are administering Q-Learning algorithm with respect to Serverless environment, we discretise the environment states to avert the State Space Explosion problem and therefore the state space S is a vector with all the discussed factors i.e. with dimensions $P \times C \times F \times T$ with N maximum pods and n timeframes allowed in the process. Hence, an RL environment state in the FaaS setting can be described as a combination of the above factors, represented as a tuple $(p_i, c_i, f_i, t_i) \in S$.

- **Action**

The RL agent is structured to learn the appropriate number of function instances for critical time periods and provision them in advance to reduce the frequency of cold starts. In the process of learning required number of function pods for a specific timeframe, the agent explores different set of function instances over the iterations. Therefore, in a given state the agent should decide to increase or decrease the number of function instances for the upcoming timeframe, in order to compensate for the expected cold starts by the workload. The possible actions for an environment state can be formulated in terms of function instances that needs to be added or removed from the current set of instances. Hence, the set of possible actions range from $A = \{-(N-1), \ldots 0, \ldots . (N-1);\ \in I\}$; where *N* is the maximum allowed function instances for the experiment. As the number

of available pods are always greater than 1 (Kubernetes maintains a minimum of 1 pod in a deployment [10,12,14]) and less than the maximum allowed pod limit, not every action is appropriate for each state. Hence, to prevent the agent from performing invalid actions and elongate the learning process, we create a Python dictionary to store the allowed actions for every possible environment state. Thus, the allowed actions for a state s, with n current function pods and N maximum allowed pods, can be obtained as $n_{next} = \{k \mid k \in (0 < n + a < N); a \in A\}$.

- **Reward**

The motive of RL agent is to learn an optimal policy for reducing the density of cold starts on the platform. It interacts with the environment and makes a judgement of the goodness of performed action according to the observed reward and tries to maximise the cumulative reward. Therefore, reward is an important characteristic of the RL world that guides the agent throughout the decision-making process. In this context, we design the reward in a manner to appropriately capture the influential variables for cold starts. Hence, the reward R, incorporates the effect of average CPU utilisation C, number of function instances P and the failed responses F to the incoming workload i.e. $R \propto C, R \propto F$ & $R \propto P^{-1}$. Since all the observed metrics are important for the agent to ascertain the effect of action, we model the reward function for a state-action pair as –

$$R = a * Cr + b * Fr + c * (1/P) \quad (6)$$

where –

- ‘a’ is the proportionality constant for average CPU utilisation across all available function instances and is set to 0.3.
- ‘b’ is the proportionality constant for failed responses during the timeframe and is set to 0.3.
- ‘c’ is the proportionality constant for available function instances and is set to 4.
- ‘*Cr*’ represents the award against division of percentage CPU utilisation according to the state discretised model and assigns a score $Cr = n * 20; \{n \in [1,5] \cup (n-1) * 20 < C < n * 20\}$. corresponding to
- ‘*Fr*’ signifies the division of failed responses and assigns a failure class score $Fr = 2^m; \{m \in [0,4] \cup m * 20 < F < (m+1) * 20\}$.

The reward is structured in the described manner to utilise minimum number of function instances even while keeping the other variables constant, thus learning to approach towards an optimal number of function instances. This step also encourages the agent to move in a positive direction i.e. reduce the failed response over the training iteration, while trying to keep the number of function instances low.

- **Q-Learning Agent Workflow**

Q-Learning technique conventionally leverages the method of dynamic programming for the process of learning [16,42]. Hence, the use of a Q-table to store the Q-values that assess the quality of an action against a specific environment state is justified. The agent leverages the information of Q-values to perform further moves and gain additional

rewards for its decision. Therefore, Q-table is a mapping $Q_t \rightarrow (S \times A)$ for each state-action pair [18] [29]. During the RL environment initialisation, all the invalid actions $a_i \in A$, for state $s_i \in S$ are marked as negative ∞, to abide by the design of action space and the valid actions are initialised with a zero value for uniform chances. Since the agent's objective is to maximise the cumulative reward at each step, setting the Q-value of invalid actions for a specific state inhibits the exploration of particular insignificant action.

Reinforcement learning tasks are known to be time consuming and therefore different implementations incorporate approximation techniques or heuristics to expedite the learning process. In the similar context, we integrate a *previous reward heuristic* that assists our agent to determine the quality of the action performed from a state. A previous reward mapping $R_{prev} \rightarrow (S \times A)$, corresponding to each state-action pair is configured to operates as a repository for the obtained immediate rewards. The values in the previous reward repository works as a benchmark to evaluate the reward obtained by performing an action from a specific state in the current iteration i.e. the immediate reward. It assists the agent's judgment of the performed action by comparing the current reward with the previous reward for state-action pair. If the previous reward is greater than the current reward, then the agent is penalised, for it has performed a low value action and hence updates the obtained immediate reward as negative feedback (Equation 7) and vice-versa. This, in turn, is leveraged by Bellman Equation to update the Q-values and helps the agent to maximise the cumulative reward.

$$Immediate\ Reward = \begin{cases} -R(s_i, a_i), & R(s_i, a_i) < R_{prev}(s_i, a_i) \\ R(s_i, a_i), & R(s_i, a_i) \geq R_{prev}(s_i, a_i) \end{cases} \quad (7)$$

An overview of agent training is presented in Figure 12. In the learning process, the agent takes the maximum number of allowed function instances, iteration period and the number of timesteps as input. It performs the initial setup of modelling the state space, action space and Q-table that aids our agent's learning process and guide towards the reward maximisation. The agent determines the current environment state with the help of required metrics and performs an action according to the defined policy. As the agent is modelled to follow ε – Greedy policy to explore and exploit the information gained during the learning, it decides to choose a random action with a probability ε or alternatively, choose the action with highest Q-value (initially all valid actions equal to zero value) [18]. The ε-value is configured to 2.5% to maximise the use of acquired information and allow the agent to learn about actions as per the rewards.

The action chosen by the agent is, in fact, the number of function instances to be added or removed from the deployment and the resultant set of instances are used to serve the fabricated request workload, for a specified timeframe. Therefore, the agent leverages the utility function to interact with underlying Kubernetes API-server to scale the function instances to a desired number. Once the action is performed, the agent uses *sleep* method for a period of *timeframe* and allow the workload to be handled by the new set of functions to observe a *delayed reward* for the current state-action tuple. In the context of cold start problem, we replace the notion of immediate rewards with delayed rewards. As the state transitions are modelled to capture the variables such as CPU utilisation and failed responses, states are observed after each *timeframe*. The computed *delayed reward* is assessed against the *previous reward heuristic* and this information is appended to the

current Q-value using the Bellman Equation. The parameters of the equation are configured as follows, for this study. The learning rate α is set to 0.75 and discount factor γ is configured to 0.9, inferring that 75% of the newly obtained information is taken into account while giving a high weightage to the expected future rewards in the following environment states. The process of determining the current state, choosing an action, observing the reward and interpreting the obtained information is carried out in multiple iterations, for a superior learning of the agent and explore the environment.

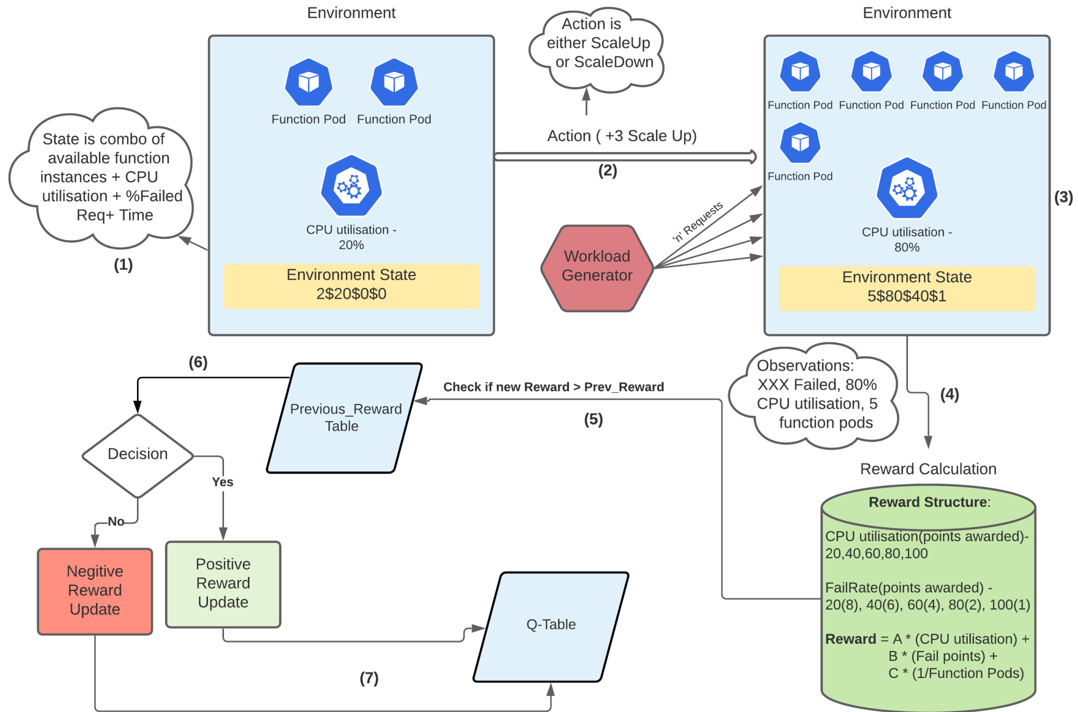

Figure 12. High-level view of Proposed RL-based Agent Learning Process.

4.3 Reinforcement Learning-Agent Setup

The discussed RL-based approach is implemented as a Python agent using standard libraries such as *numpy, pandas* and *pickle*. The Python-based implementation of RL approach is composed of two different modules – *MetricsCollector* and *RLAgent*. As the concerned function cold start is structured as a Reinforcement Learning problem, the Q-Learning environment leverages the resource metrics along with the response metrics of the FaaS instances. We have designed our Python-based *MetricsCollector* to directly interact with the Kubernetes and Kubeless Command Line interfaces and fetch the required resource metrics. It queries the specific deployments under the Kubeless cluster and feeds the bifurcated and processed information to the agent. We also incorporate

Python's built-in *logging* module to log the details about the resource metrics, response metrics of the functions and the information of agent's learning, to track and query the progress.

The *RLAgent* module is the primary implementation of the Q-Learning logic for the purpose of workload model analyses and reduce the cold start frequency. The different component methods of the *RLAgent* are as follows – *(i) EnvironmentSetup, (ii) GenerateActions, (iii) ActionPolicy, (iv) CalculateReward and (v) TrainAgent.* As discussed, the principal task of the Q-learning agent is to formulate the RL environment in terms of Serverless resources to commence the learning process. The agent receives the maximum number of function instances (N) used for servicing the synthetic workload, number of *timeframes* (k) and the *iteration period* (I_t), as input parameters and are used in the initialisation procedure. In order to conduct the analyses on FaaS platform for the applicability of RL-based agent to reduce cold start frequency, we provision 10 function instances as the maximum allowed limit i.e. $N = 10$. We constrain the scaling of function up to only 10 instances to uniformly distribute the load on the underlying infrastructure resources and avoid any performance bottlenecks, while abstaining from the problem of *state space explosion*. We ascertain the performance of the agent over the *iteration period* of $I_t = 60$ *minutes* and observe the state transitions for $k = 12$ *timeframes* of $T = 5$ *minutes* each, over the multiple iterations.

The *EnvironmentSetup* method is pivotal in providing the building blocks for the learning policy. It configures the state space, action space, the Q-table and the proposed previous reward heuristic mapping, according to the schema discussed in the previous section. The agent takes advantage of input variables and the default settings to implement a state space mapping $\pi\colon S \rightarrow P \times C \times F \times T$ i.e. a state space of size $\pi\colon S \rightarrow (10 \times 5 \times 51 \times 12)$ and represent the state as tuple with following constraints –

$$S_i = (p_i, c_i, f_i, t_i) \in S \,, \quad \text{where} \begin{cases} p_i \in \{1, 2, 3, \dots, 10\} \\ c_i \in \{20, 40, 60, 80, 100\} \\ f_i \in \{0, 2, 4, 6, \dots, 100\} \\ t_i \in \{0, 1, 2, 3, \dots, 11\} \end{cases} \tag{7}$$

The action space for the RL environment is described as the number of function instances added or removed from current replicas and hence mapped as $A = \{-9, -8, \dots 0, 1, 2 \dots 8, 9\}$ according to the defined schema (section 4.2). The Q-table and the heuristic previous reward table are mapped for each state-action pair and therefore are represented as matrix of dimension $\pi\colon Q_t \rightarrow (S \times A)$ and $\pi\colon R_{prev} \rightarrow (S \times A)$ where $S = 30600, A = 19$. The tables are initialised with default zero values as per the defined model in Equation 8.

$$Q_t \rightarrow States \overset{Actions}{\begin{bmatrix} a & \cdots & b \\ \vdots & \ddots & \vdots \\ c & \cdots & d \end{bmatrix}}_{(\mathbf{S\ x\ A})} \qquad R_{prev} \rightarrow States \overset{Actions}{\begin{bmatrix} j & \cdots & k \\ \vdots & \ddots & \vdots \\ l & \cdots & m \end{bmatrix}}_{(\mathbf{S\ x\ A})} \tag{8}$$

To reduce the computational complexity and expedite the setup process, we allow the agent to only perform valid actions. When an agent observes its current state, the *GenerateActions* method creates a list of all the possible actions from the current state and updates the invalid actions in the Q-table as negative infinity, corresponding to the current state. This strategy is known as the *lazy evaluation* in Python. Therefore, the assessment of the quality of an action will be conducted only for valid actions, thus reducing the exploration complexity of the agent's learning process. Once the valid actions are determined for a state, method *ActionPolicy* is responsible for following the ε-Greedy policy to balance the exploration and exploitation of the acquired knowledge of the environment. The method selects an action randomly from the list of valid actions with ε probability and chooses to select an action with maximum Q-value with a higher probability of 1- ε.

TrainAgent is a nested method that leverages the services of other component methods. After the environment setup is complete, the training executes for a period of 60 minutes over multiple iterations. During the respective *timeframe* of 5 minutes, the agent selects an action to perform and directly interacts with the platform through command line interface to increase or decrease the number of function instances. The agent waits for the aggregated metrics over the period and scrapes the relevant resource metrics and response metric through *MetricsCollector* module. It queries the NGINX Ingress of the associated function deployment, gathers the details of the incoming requests and filters them to compute the response metrics i.e. failed number of requests during the queried timeframe. These metrics serve as the input to *CalculateReward* method that returns the delayed reward for the action performed, according to the reward structure. This reward facilitates the heuristic-based comparison to assess the quality of action and update the Q-Table via Bellman Equation. This process helps the proposed agent to acquire knowledge about the environment states and its related actions, determining the appropriate number of function instances to service the fabricated workload during the iteration period. The RL-based agent learning is performed for 64 hours i.e. 64 iterations of 60 minutes each, to obtain a satisfactory result over the default setting offered by the platform.

4.4 Performance Evaluation

The performance of our proposed RL-agent is evaluated against the default autoscaling policy i.e. Horizontal Pod Autoscaler (HPA) supported by the Kubeless platform [14]. The objective of the proposed agent is to ascertain an appropriate amount of function instances for a specific application workload, to reduce the number of cold starts on the platform. We observe that infrequent workloads have irregular demands for function instances that lead to cold starts on the serverless platforms. Therefore, we hypothesise that if the right amount of function instance is provisioned for a timeframe, then there will be reduced number of cold starts as well as decreased failures of response. Thus, the performance of the agent and the default autoscaling policy is measured in terms of successfully serviced requests or number of failed requests by a set of function instances.

The RL-agent is trained for 64 iterations of 60 minutes, under the influence of fabricated and sporadic application workload. We use a CPU intensive workload function i.e. Fibonacci sum calculator up to number 30, to keep ample amount of resources busy for observing the effect of cold starts [36,45]. The infrequent incoming pattern of requests

are generated from a pool of 1500 requests during a specific timeframe of 5 minutes and therefore a limit of 10 function instances to comfortably balance the load and prevent state explosion problem. These constraints allow us to put a considerable load or pressure over the competing approaches and effectively evaluate them against each other. Kubeless leverages the default autoscaling policy of HPA that is implemented to support the resource scaling based on the average CPU utilization threshold. In the context of HPA, we make a significant decision to configure the *timeframe* of 5 minutes for the RL-agent. The HPA is configured as a control loop to query the underlying aggregated resource metrics every 15 seconds and perform a required scaling action. But to prevent the resources from thrashing i.e. frequent acquisition and release of resources like containers, CPU or memory, Kubernetes has a default downscaling time of 5 minutes [10, 14]. In other words, HPA tends to keep the resources occupied for up to a period of 5 minutes in order to account for irregularity and releases them after the reduced load. Therefore, it is worthwhile to analyse the performance of RL-agent where the default policy has a strict downsizing scheme, keeping the ample amount of resources bound to itself for a period of 5 minutes.

4.4.1 Analysis of Results

As discussed, we train the RL-agent for a period of 60 minutes over 64 iterations to analyse a specific application workload pattern and learn the ideal number of function instances to reduce cold starts. The agent is structured according to the experiment strategy with RL environment designed around the experimental constraints. To promote the Q-Learning process, the agent meticulously follows the described learning schema, aided by the established heuristics of previous reward comparison. We leverage the Python's *logging* module to log information during the agent learning and continuously monitor the progress in terms of proportion of successfully serviced request out the total workload.

The training is performed in 7 batches of iterations and Figure 13 illustrates the learning curve of the agent over the batch training. From the illustration, we observe that eventually RL-based agent successfully attempts to service 90% of incoming workload and is shown to increase the proportion of successfully served requests over iterations. This learning analysis supports our hypothesis that as the agent learns to provision ideal number of function instances, that signifies reduced number of cold starts, it begins to service a greater number of requests. In the context of learning, since we are focused towards analysing an application workload pattern, a similar fabricated request pattern is used among various iterations to strengthen the learning capability of the agent.

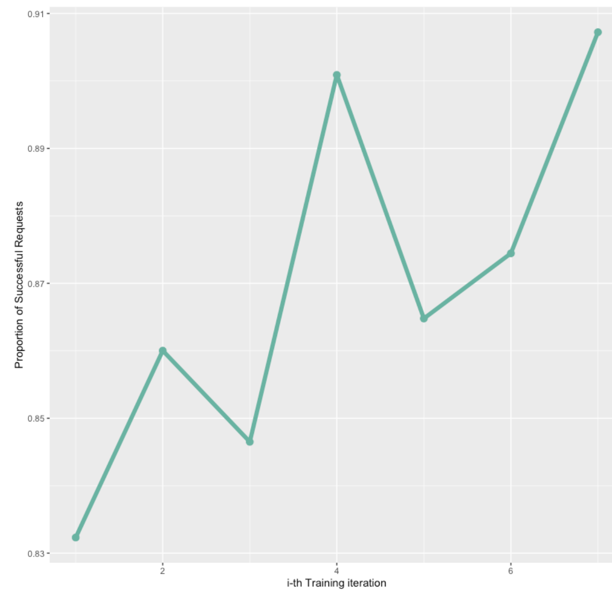

Figure 13. Proportion of Success vs Training Iteration (Batch).

In the Q-Learning process, agent continuously perform prescribed steps of determining the current state, choosing an action according to the greedy policy and observe some reward associated to the transition, to update the corresponding Q-values. As per the greedy policy, the agent tends to select an action that maximises the overall cumulative reward from a state and adds a positive feedback in the form of additional information to the respective Q-value. Another factor to elevate the Q-value, corresponding to a state-action pair, is the expected future reward, that is eventually determined based on the experience of our agent. Therefore, in Figure 14, we can observe that the expected future reward or the probability of choosing a suitable step from the observed state is consistently increasing. Hence, the observations are in correlation with the fact that the agent is learning to adapt to the workload pattern and attempting to select an ideal action over the multiple iterations of learning.

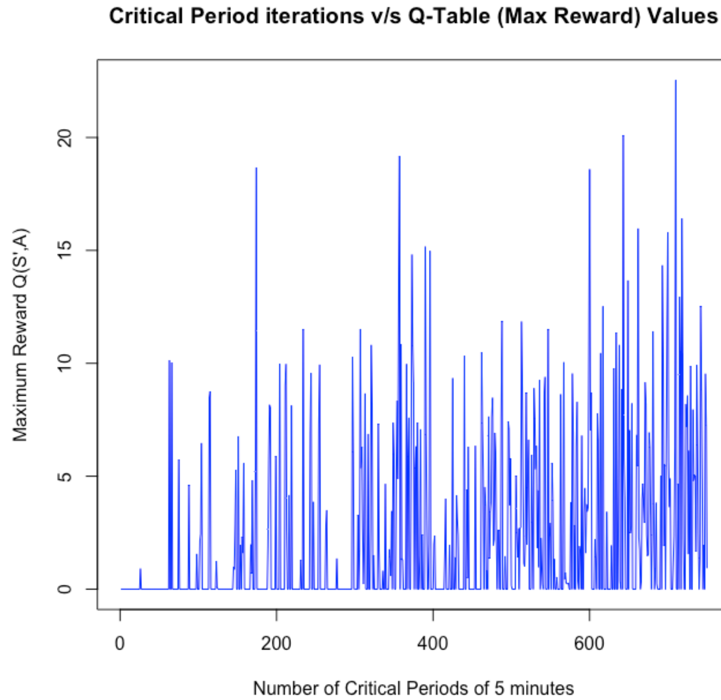

Figure 14. Expected Future Rewards during Iterations.

We assess the effectiveness of RL-based agent against the baseline Horizontal Pod Autoscaler which is accountable for requesting new function instances on the Kubeless platform, based upon the sporadic workloads. As discussed in earlier section, HPA is a Kubernetes controller that watches for changes in the specified metrics associated with the function deployment and acts accordingly. By default, it is configured to monitor the CPU metrics or the average CPU utilisation across the available function instances to be below a threshold and has a query period of 15 seconds [12,14,15]. To assess its capability, we set the threshold for CPU metrics to be 80% with scaling of the function up to 10 instances. Therefore, whenever the average CPU utilisation of the function deployment violates the threshold, new function instances are provisioned in real-time, representing a potential cold start in the system.

We conduct the assessment on baseline autoscaling policy for a period of 4 hours to avoid any bias and observe that the Kubeless HPA achieved a mean success rate of 558 requests per timeframe, accounting for an overall failure rate of 15.16% (with a mean of 100 failed requests, Figure 15) over the period of assessment. The assessment was conducted for the CPU intensive function of Fibonacci sum calculation of number 30, over an irregular application request pattern, described in the workload model, with a total of approximately 29000 requests over the 4 iterations. The observed results can be attributed to following considerations –

- A configuration of 15 second control loop in HPA that collects resource metrics from metrics-server, which in turn queries aggregated resource metrics from individual resources after every 30 seconds. Therefore, HPA does not seem to consider the resource threshold failure events in real-time and hence, fails to scale the function deployment at the appropriate moments of requirement. Also, since the HPA is collecting aggregate resource metrics over a period of time, it neglects

short spikes during the 15 seconds query time and fall short to account for the failed requests.

- In the context of the previously highlighted point, we make use of a CPU intensive function handler that executes or keeps the resource bound for a slightly longer period of time, in the order of few seconds. Therefore, the HPA fails to capture the resource pressure as well as the incoming workload in real time and generates considerably large proportions of failed responses.

- The observed result can also be attributed to the limited amount of scaling that is allowed to be performed during the assessment. Although, the HPA has a default downsizing policy of 5 minutes, it keeps the resources bound with itself but fall short to account for the irregular workload occurrences while performing the downsizing action during the 15 second control loop period.

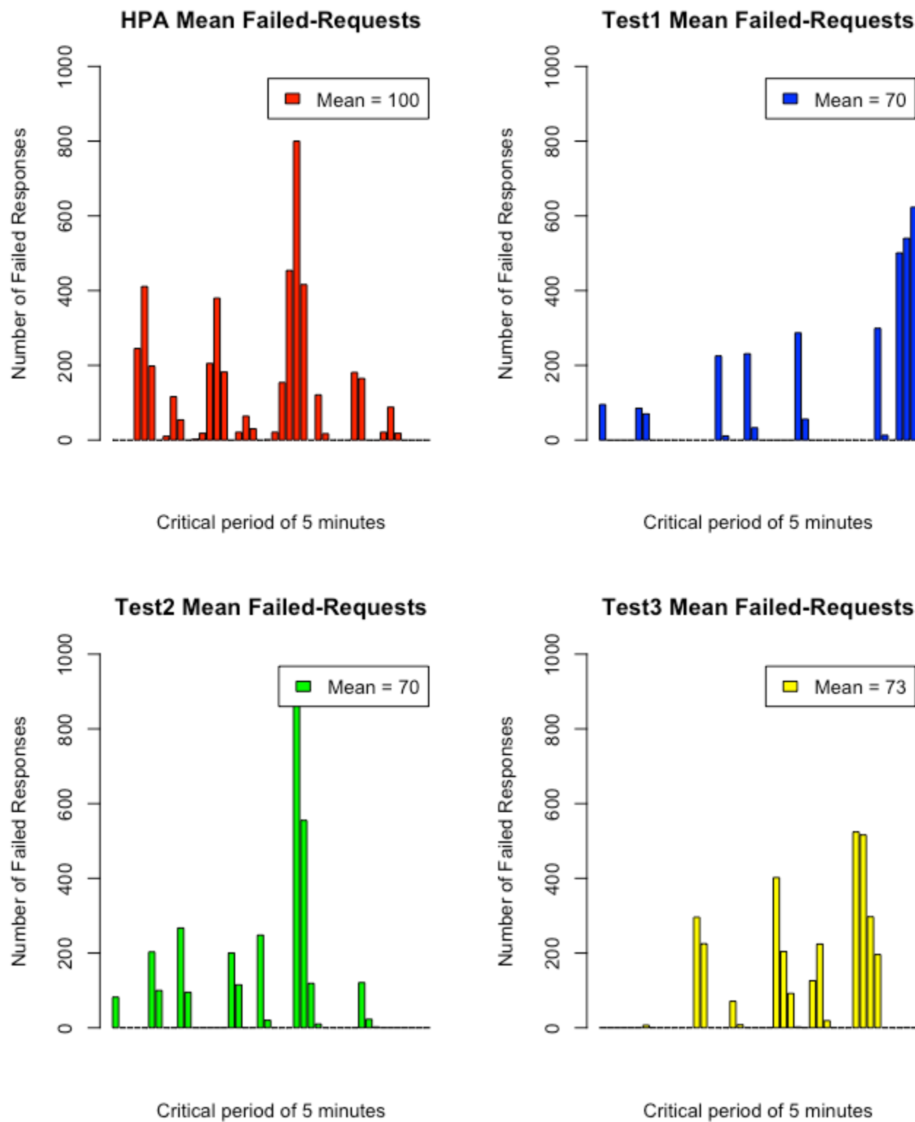

Figure 15. HPA vs RL Agent – Mean Failed-Response Comparison.

Similar to baseline HPA policy, we assess the quality of RL-based agent by computing the failed responses and success rate during a 4-hour evaluation. We conduct three different tests, under the similar conditions to avoid any bias and performance bottlenecks. As we deal with the problem of frequent cold starts on a serverless platform using a RL-based model, the agent must account for the unprecedented or the unvisited states and should be able to react in a typical manner. But, with a limited iteration of training, there is a high probability that the agent does not visit all the environment states or explore all the available state-action pairs. Hence, to compensate for the unseen situation or environment states, where usually an RL agent acts in a random fashion, we introduce a *heuristic-based approximation method* (Algorithm 3) to provision a sufficient amount of resources to anticipate the demand. The proposed solution favours to select an average number of function instances i.e. average number of function instances between the current and maximum limit. This way the agent is able to provision a favourable set of function instances to compensate for a fraction of cold starts and hence serve a higher fraction of incoming requests successfully.

Algorithm 3: TestAgent

BEGIN
$InitialState \longleftarrow ScrapeResourceMetrics$
$ActionList \longleftarrow Choose\ MaxQvalueAction(InitialState)\ from\ Q - Table$
if $length\ (ActionList) > 1$ **then**
  $nextAction \longleftarrow MaxPods - Mean\ (Current\ +\ MaxPods)$
  return $nextAction$
else
  $nextAction \longleftarrow ActionList$
  return $ActionList$
end

The performed RL-agent tests are compared against the HPA and the important findings are listed in the following table. The agent tests are performed using the described workload pattern with a total number of requests between 30,000 to 31,000 for the entire duration of tests. Hence, all the outcomes are reported in terms of proportion to effectively perform the comparative study.

Table 7. Summary of Results.

	HPA	Test1	Test2	Test3
Mean Success Rate (# of requests)	559	644	654	646
% Mean Difference	0	15.29	16.95	15.65
% Successful Requests	84.84	90.23	90.30	89.86
% Failed Requests	15.16	9.77	9.70	10.14

After 64 hours of training iterations, the RL-agent along with the heuristic assistance is shown to perform better than the baseline policy of HPA. We observe that during three separate tests, under the similar controlled conditions of workload model, the RL-based agent is consistently above par with the HPA and is able to successfully service 90% of the incoming workloads and reduce the failed proportion of requests by approximately 5.3%. This is presented in the Figure 15, where the RL-based agent is able to reduce the failed number of requests over multiple iterations with an average failure of approximately 71 requests that is an improvement over 100 failed requests by HPA. In Table 7, 'mean success rate' represents the average number of successful requests during a unit timeframe of 5 minutes. The proposed agent, through workload pattern analysis and training to provision suitable number of function instances, is able to continuously outperform HPA with higher number of successful requests by maintaining a considerable mean difference of proportions of 15.9%.

The performance of RL-based agent can also be evaluated by analysing the pattern of provisioned number of function instances to deal with the serverless workload. Figure 16 shows the variance in the provisioned number of function instances between the HPA and the RL-based agent. It is evident that under controlled simulated environment, the RL-based agent provisioned the functions in a similar pattern for separate tests that diverge from the default policy. The agent performs the actions based solely on the trained model and Figure 17 clearly represents and support the agent's better performance with a reduced number of failures over the testing iteration.

The difference between the two approaches can be attributed to the following characteristics of the proposed RL-based agent –

- The process of elimination of invalid states during the RL environment setup and lazy loading of Python, helps the agent to productively use the acquired information about the environment.

- The use of a heuristics-based approach, both for training as well as testing processes, expedites the learning process by awarding the agent's actions carefully and compensates for the un-explored states by provisioning average number of instances while testing the agent.

- Although the RL-based agent outperforms the baseline HPA, the lack of function container concurrency-policy adds to the failed number of requests. The CPU intensive function workload is configured with an execution time of 120 seconds and thus affected by the concurrency control of the instance.

- The composition of state space and reward function incorporates the effect of failures during the training and therefore, the agent tries to compensate for the failures in consequent steps of learning by exploiting the acquired knowledge.

On the account of listed evidence of the performance and comparison of RL-based agent against the baseline HPA, we can adequately conclude that the proposed agent successfully outperforms the default policy of HPA. We strengthen this claim by analysing the training and testing outcomes of the RL-based agent that is focused towards examining the workload pattern to reduce the failure of requests which is a direct consequence of an appropriate strength of function instances representing reduced function cold starts.

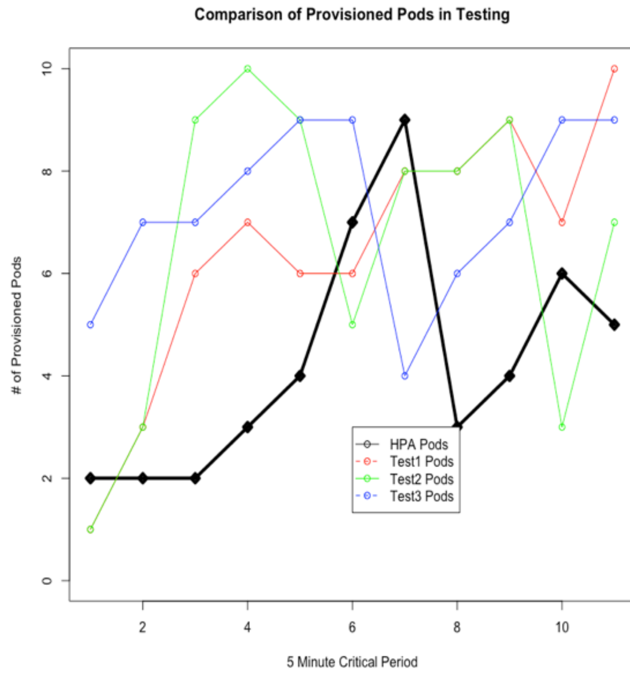

Figure 16. Comparison of Provisioned Instances.

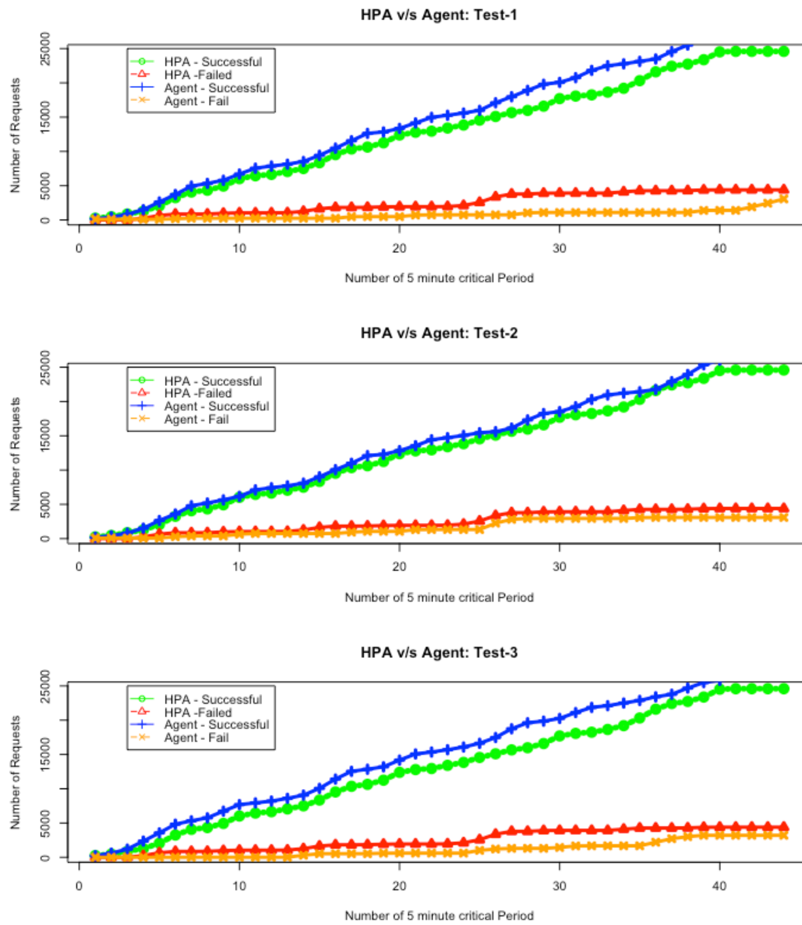

Figure 17. HPA vs RLAgent Performance Comparison.

4.4.2 Practical Implications

Function cold start is an inherent shortcoming of the Serverless execution model. Usually, the commercial platforms such as AWS Lambda, Google Cloud Functions, etc. try to optimise their runtime environments and code packaging policies to reduce the initialisation period. They address the frequency of cold starts on the platform by keeping a queue of idle function containers for a specific time, in memory, to provide successful outputs without significant delays [8]. In connection with cold starts, we have proposed a Reinforcement Learning technique to investigate the demand pattern of the application that attempts to reduce the frequency of function cold starts on the serverless platform. The proposed Q-Learning based agent is found to perform better than the baseline approach of Horizontal Pod Autoscaler of the Kubeless framework, under a simulated and controlled experimental environment. But there are certain points to recollect that are associated to real-time appropriateness of the proposed solution.

- We leverage the Reinforcement Learning environment modelling, specifically Q-Learning constraints [15,16] to devise a smart agent that learns to analyse demand pattern and increase the performance in terms of successful outcomes. The successful outcomes are the direct consequence of reduced function cold starts by appropriate resource provisioning. The RL-based solutions, in general, are expensive in terms of data as well as time. The agent interacts with the modelled environment, exploring and acquiring relevant information over multiple iterations that require a higher degree of exploration. Hence, as evidenced in this work, for an RL-based agent to outperform a baseline technique of HPA, a training period of 64-hours is exploited for satisfactorily analysing the invocation demand of an hour. Therefore, RL-based approaches are considerably expensive in practical applications with stringent optimisation requirements.

- A classical Q-Learning approach is applicable for discretised environment variables [18]. To constrain the serverless environment within the requirements of Q-learning algorithm, we categorize or discretise the different variables of cold start and model the problem of function cold starts around it. The size of Q-table is a variable of state space and the action space and under the presented controlled experimental settings, the size is large. But, as the state space or the action space increases, the size of the Q-table grows exponentially [15,16]. For instance, if the state space extends by a factor of 100 and action space increases by a factor of 10, the Q-table enlarges by a factor of 1000. Therefore, Q-Learning experiences a state explosion, making it infeasible to perform updates on Q-values and an increased space and time complexity.

- The RL-based agent is setup to analyse the demand pattern of a particular application and modelled to provision suitable number of function instances to deliver successful response. Therefore, the learning of the agent can't be generalised for other demand patterns and thus requires a respective learning to be commissioned.

- For the purpose of this thesis, we train the proposed agent for 64 hours and evaluate it for the iteration period of an hour. As discussed in the result analysis, there are 30600 state-action pairs to be explored within 768 separate timeframes

and the probability that the agent explores every option is very bleak. Therefore, with limited amount of training the agent is advised to be guided by certain action approximations, such as average number of instances used in this thesis, to avoid acting in a random manner.

- As the agent positions its learning on the resource metrics that affect the cold starts in a serverless environment, the availability of suitable tools and techniques to scrape these required metrics is essential. Also, the availability of certain platform constraints such as frequency of querying resource metrics, the concurrency policy of the function instances or the request queuing policy would further extend support to the analyses.

Chapter 5 - Conclusions and Future Directions

This chapter concludes the thesis and presents a summary of the proposed work. It also highlights the future directions of the current work as well as other possible perspectives to target the problem of Function Cold Start.

5.1 Conclusions

In the Cloud computing era where enterprises are looking towards faster time to market and compete within the industry, Serverless Computing framework emerges as a fascinating choice. With its Function-as-a-Service (FaaS) execution model and micro-services inspired application development architecture, FaaS offers the best of both, Cloud Computing and Service Oriented Architectures. FaaS leverages the concept of containers to provide an abstraction of underlying servers and furnish the services at a consumption-based pricing model. To expedite the process of application development and allow the enterprises to add business value, rather than focusing on the time-consuming tasks of resource management during application development, FaaS takes off the resource management responsibilities from developers by offering them a platform with an abstraction of highly available and scalable resources.

FaaS model executes the piece of code inside a container, known as a function and prepares new function containers on demand. The serverless execution model is suitable for cases with highly irregular demands to economise the resources and profit from the pay-per-use pricing. To service the incoming demand, new function containers undergo an initialisation process that puts together all the essential components like runtime environment, code image, code dependency etc., before executing the function handler. This bootstrapping process consume time in the order of few seconds, known as function cold start and introduces a delay in the response of the function container. Apart from affecting response time, cold starts may give an impression of unavailable resources in cases of large delays, acting against the characteristic of serverless model. Therefore, with infrequent demand patterns, there is an abundance of cold starts on the platform and hence can be thought of as an optimisation problem.

To deal with the challenge of function cold start, multiple solutions have been proposed that can be classified under two categories - (i) reducing the duration of cold start or (ii) reducing the frequency of cold starts that happen on the platform. Various solutions like pool-based approach that keeps a queue of prepared containers idle for a period of time and pinging technique that occasionally interacts with the FaaS platform to keep the function containers alive, have been proposed. These solutions target the cold start variables like resource thresholds, runtime environment, etc. and ignore the primary cause of infrequent demand that requests function container initialisation from the platform. Since these application demand patterns can't be hard coded into the function

initialisation rules, a Reinforcement Learning based agent presents as a judicious choice to reform the cold start issue. Machine Learning techniques, specifically Reinforcement Learning models have been historically found useful in optimisation problems such as resource management and are highly adaptive to the complexities of the problem [42].

In this thesis, we visited the problem of function cold start by addressing the frequency of cold starts on the platform and analyse the application demands through Reinforcement Learning technique. To generate the necessary application workload, we leverage services of Apache JMeter to produce infrequent request patterns and a CPU intensive function handler to complement the invocation pattern and observe relevant cold starts. We setup the serverless platform using Kubeless framework and model the RL environment for the agent to examine the necessary metrics or environment observations, to make guided decisions in provisioning appropriate number of function instances. The prior provisioning of functions results in lesser number of cold starts on the platform for the irregular application demands. Therefore, in this thesis we present an evidence of Q-Learning based agent that reduces the number of function cold starts that utilises a direct consequence metrics of decreased failure rate during the iterations. To expedite the learning procedure and guide the agent effectively, we make use of heuristics to better assess the quality of actions and to behave in a non-random fashion for the unvisited states. We evaluate the performance of our proposed agent against the default autoscaling policy of Kubeless i.e. Horizontal Pod AutoScaler and successfully observe that after a training of 64 hours the Q-Learning agent was able to outperform HPA and verified our hypothesis of strong association between success rate and reduced number of cold starts on the platform. After the test analyses, the Q-Learning agent is found to successfully serve an average of 15.9% proportion of the incoming requests in a single timeframe while reducing the overall failure rate by approximately 5.3%.

5.2 Thesis Summary

The study investigated the applicability of Reinforcement Learning technique to a Serverless environment for the problem of function cold start. We structure the RL environment in terms of serverless platform resources and other relevant metrics to inspect the application invocation patterns and propose a Q-Learning agent to leverage these settings. With the help of appropriate heuristics and training model, we presented an evidence of effective learning and assessed the performance of agent against baseline autoscaling policy. The agent was found to be performing superior to the baseline and the analysis of the results were presented.

Chapter 1 introduced the concept of Serverless Computing in the Cloud Computing domain and briefly describe the Function-as-a-Service platforms and their challenge of cold start. It also presents the motivation for this thesis, outlining the research questions addressed by the study and list the adopted research methodology.

Chapter 2 extensively presents the background of Serverless Computing, Function-as-a-Service execution model, its utility and existing challenge of function cold start. It investigates an open-source serverless framework – Kubeless and highlights the necessary information about Reinforcement Learning and Q-Learning technique.

Chapter 3 examines few scholarly works on Serverless computing and highlight existing techniques to address the cold start challenge and discusses their respective approach towards function cold starts, presenting a variation from the proposed technique.

Chapter 4 presents the proposed RL-based agent that analyses the application workload pattern to reduce the frequency of function cold start. It presents the details about the agent modelling, workload structure and evaluate the performance of the proposed solution against the baseline autoscaling policy, HPA. Finally, the report concludes by highlighting the important results and practical implications of the adopted RL-based model.

5.3 Future Research Directions

This thesis addressed the challenge of frequent function cold starts on the Function-as-a-Service platform due to the infrequent application demands. We proposed a Q-Learning model in combination with heuristics to reduce the frequency of cold starts. We evaluate the feasibility of model under controlled settings of state-action structure and reward calculation. As part of the future research directions, other important variables such as memory utilisation and function package size can also be identified and leveraged to assess the quality of learning and benefit from a broader perspective, to reduce the cold start frequency [47]. Similar to Q-Learning, application of other policy-based techniques such as SARSA, that is known converge quickly than Q-Learning, can also be experimented within the domain of cold start problem.

As an adaptation of classical Q-Learning technique, the proposed solution includes discretisation of continuous values for state representation. In this context, to avoid the problem of state space explosion, techniques such as Deep Q-Networks i.e. function approximators can be leveraged [16]. These models abstain from storing the Q-values in the form of large tables and instead utilise a deep neural network or artificial neural network to efficiently estimate the Q-values. Another Deep Reinforcement Learning (DRL) technique known as Policy Gradient [49] can also be explored that typically works towards optimising the action policy of the agent by maximising the parametrised reward function, rather than finding optimal action rewards for the agent.

As described in the thesis, the proposed solution looks into the problem of frequent cold starts that are primarily affected by the sporadic demands. An application of Reinforcement Learning technique can also be explored to optimise the duration of cold starts that is further affected by underlying modelling principles of different serverless platforms. A number of approaches have been explored in context of reducing the cold start duration, but there is need to investigate the feasibility of RL techniques to optimise the cold start duration and the presented research outcome paves a path for future investigations and evolutions in the domain of Serverless computing.

Bibliography

[1] "CNCF WG-Serverless Whitepaper v1.0," Cloud Native Computing Foundation (CNCF), 2018. [Online]. Available: https://raw.githubusercontent.com/cncf/wg-serverless/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf. [Accessed 2020].

[2] A. D. Buchen, "Why Is Serverless the Future of Cloud Computing?," Alibaba, 21 January 2021. [Online]. Available: https://www.alibabacloud.com/blog/why-is-serverless-the-future-of-cloud-computing_597191. [Accessed 2021].

[3] "Function Quotas," Google, [Online]. Available: https://cloud.google.com/functions/quotas. [Accessed 2021].

[4] "Lambda quotas," AWS , [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html. [Accessed 2021].

[5] "Azure subscription and service limits, quotas, and constraints," Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/azure-subscription-service-limits. [Accessed 2021].

[6] "System details and limits," IBM , [Online]. Available: https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-limits. [Accessed 2021].

[7] B. D. Rooms, "A Comparison of Serverless Function (FaaS) Providers," Fauna, 6 February 2020. [Online]. Available: https://fauna.com/blog/comparison-faas-providers. [Accessed 2020].

[8] M. Shahrad, R. Fonseca, I. 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 2020 USENIX Annual Technical Conference*, 2020.

[9] M. Shikov, "Comparison of Cold Starts in Serverless Functions across AWS, Azure, and GCP," January 2021. [Online]. Available: https://mikhail.io/serverless/coldstarts/big3/. [Accessed 2021].

[10] "Kubernetes Documentation," Kubernetes, [Online]. Available: https://kubernetes.io/docs/home/. [Accessed 2020].

[11] "Kubernetes Deployment: The Ultimate Guide," Platform9, [Online]. Available: https://platform9.com/docs/deploy-kubernetes-the-ultimate-guide/. [Accessed 2020].

[12] D. Sanche, “Kubernetes 101: Pods, Nodes, Containers, and Clusters,” Medium, January 2018. [Online]. Available: https://medium.com/google-cloud/kubernetes-101-pods-nodes-containers-and-clusters-c1509e409e16. [Accessed 2020].

[13] “Kubeless - Kubernetes native Serverless,” Kubeless, [Online]. Available: https://kubeless.io/docs/. [Accessed 2020].

[14] T. Nguyen, Y. Yeom, T. Kim, D. Park and S. Kim, “Horizontal pod autoscaling in Kubernetes for elastic container orchestration,” in *Sensors (Basel)*, 2020.

[15] R. Sutton, “Reinforcement learning: Past, present and future,” in *Asia-Pacific Confrenece on Simulated Evolution and Learning*, Berlin, Heidelberg, 1998.

[16] S. Zychlinski, “Qrash Course: Reinforcement Learning 101 & Deep Q Networks in 10 Minutes,” towards data science , 10 January 2019. [Online]. Available: https://towardsdatascience.com/qrash-course-deep-q-networks-from-the-ground-up-1bbda41d3677. [Accessed 2020].

[17] “Reinforcement Learning : Markov-Decision Process,” towards data science, [Online]. Available: https://towardsdatascience.com/introduction-to-reinforcement-learning-markov-decision-process-44c533ebf8da. [Accessed 2020].

[18] “Q-Learning,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Q-learning. [Accessed 2020].

[19] H. Lee, K. Satyam and G. Fox, “Evaluation of production serverless computing environments,” in *Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018.

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

[21] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski and P. Suter, “Serverless computing: Current trends and open problems,” in *Research Advances in Cloud Computing*, Singapore, Springer, 2017, pp. 1-20.

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

[23] H. Shafiei, A. Khonsari and P. Mousavi, “Serverless computing: A survey of opportunities, challenges and applications,” in *arXiv preprint arXiv:1911.01296v3*, 2019.

[24] G. Chandra, “Kubeless — Kubernetes Native Serverless Framework,” ITNEXT, July 2019. [Online]. Available: https://itnext.io/kubeless-kubernetes-native-serverless-framework-3d0f96e03add. [Accessed 2020].

[25] "NGINX Overview," NGINX, [Online]. Available: https://docs.nginx.com/nginx-ingress-controller/overview/. [Accessed 2020].

[26] "Apache JMeter - Getting Started," Apache Jmeter, [Online]. Available: https://jmeter.apache.org/usermanual/index.html. [Accessed 2020].

[27] P. Shah, "Jmeter | Pass Command line properties," 24 January 2020. [Online]. Available: https://medium.com/@priyank.it/jmeter-pass-command-line-properties-65a431875024. [Accessed 2020].

[28] M. Ashraf, "Reinforcement Learning Demystified: Markov Decision Processes (Part 1)," BecomeSentient, 16 February 2021. [Online]. Available: https://becomesentient.com/markove-decision-processes/. [Accessed 2021].

[29] M. Mayank, "Reinforcement Learning with Q tables," ITNEXT, 2 March 2018. [Online]. Available: https://itnext.io/reinforcement-learning-with-q-tables-5f11168862c8. [Accessed 2020].

[30] P. Vahidinia, B. Farahani and F. S. Aliee, "Cold start in serverless computing," in *Proceeddings of the International Conference on Omni-layer Intelligent Systems (COINS)*, Barcelona, Spain, 2020.

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

[32] P. M. Lin and A. Glikson, "Mitigating cold starts in serverless platforms: A pool-based approach," in *arXiv preprint, arXiv:1903.12221*, 2019.

[33] K. Mahajan, S. Mahajan, V. Misra and D. Rubenstein, "Exploiting content similarity to address cold start in container deployments," in *Proceedings of the 15th International Conference on emerging Networking EXperiments and Technologies*, Orlando, FL, USA, 2019.

[34] D. Bermbach, A. Karakaya and S. Buchholz, "Using application knowledge to reduce cold starts in FaaS services," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, Brno, Czech Republic, 2020.

[35] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing*, Renton, WA, USA, 2019.

[36] J. Manner, M. Endreß, T. Heckel and G. Wirtz, "Cold start influencing factors in function as a service," in *Proceedings of the 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, Zurich, Switzerland, 2018.

[37] K. Solaiman and M. Adnan, "WLEC: A not so cold architecture to mitigate cold start problem in serverless computing," in *Proceedings of the 2020 IEEE International Conference on Cloud Engineering (IC2E)*, Sydney, NSW, Australia, 2020.

[38] J. Santos, T. Wauters, B. Volckaert and F. D. Turck, "Towards network-aware resource provisioning in kubernetes for fog computing applications," in *Proceedings of the 2019 IEEE Conference on Network Softwarization (NetSoft)*, Paris, France, 2019.

[39] S. K. Mohanty, G. Premsankar and M. D. Francesco, "An Evaluation of open source serverless computing frameworks," in *Proceedings of the 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Nicosia, Cyprus, 2018.

[40] L. Schuler, S. Jamil and N. Kühl, "AI-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments," in *arXiv preprint arXiv:2005.14410*, 2020.

[41] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *Proceedings of the 2018 IEEE International Conference on Cloud Engineering (IC2E)*, Orlando, FL, USA, 2018.

[42] H. Arabnejad, C. Pahl, P. Jamshidi and G. Estrada, "A comparison of Reinforcement Learning techniques for fuzzy cloud auto-scaling," in *17th IEEE/ACM International Symposium On Cluster, Cloud And Grid*, Madrid, Spain, 2017.

[43] A. Galstyan, K. Czajkowski and K. Lerman, "Resource allocation in the grid using reinforcement learning," in *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, New York, New York, USA, 2004.

[44] J. Li, S. Kulkarni, K. Ramakrishnan and D. Li, "Understanding open source serverless platforms: Design considerations and performance," in *Proceedings of the 5th International Workshop on Serverless Computing*, Davis, CA, USA, 2019.

[45] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *Proceedings of the IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019.

[46] L. Wang, M. Li, Y. Zhang, T. Ristenpart and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, Boston, MA, USA, 2018.

[47] A. Saha and S. Jindal, "EMARS: Efficient Management and Allocation of Resources in Serverless," in *Proceedings of the IEEE 11th International Conference on Cloud Computing*, 2018.

[48] G. McGrath and P. Brenner, "Serverless computing: Design, implementation, and performance," in *Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, Atlanta, GA, USA, 2017.

[49] L. Weng, "Policy Gradient Algorithms," 2018. [Online]. Available: https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html. [Accessed 2020].

[50] M. Roberts, “Serverless Architectures,” martinFlower.com, 22 May 2018. [Online]. Available: https://martinfowler.com/articles/serverless.html. [Accessed 2020].

[51] M. Shikov, “Serverless: Cold Start War,” 2018. [Online]. Available: https://mikhail.io/2018/08/serverless-cold-start-war/. [Accessed 2020].

[52] N. Malishev, “AWS Lambda Cold Start Language Comparisons, 2019 edition,” 4 September 2019. [Online]. Available: https://levelup.gitconnected.com/aws-lambda-cold-start-language-comparisons-2019-edition-%EF%B8%8F-1946d32a0244. [Accessed 2020].

[53] B. Wu, “Kubeless: A Deep Dive into Serverless Kubernetes Frameworks,” Alibaba Cloud, June 2019. [Online]. Available: https://alibaba-cloud.medium.com/kubeless-a-deep-dive-into-serverless-kubernetes-frameworks-1-fe3e581a27ec. [Accessed 2020].

[54] T. Lynn, P. Rosati, A. Lejeune and V. Emeakaroha, “A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms,” in *Proceedings of the 2017 IEEE 9th International Conference on Cloud Computing Technology and Science (CloudCom)*, Hong Kong, China, 2017.