Mitigating the cold start issue in Serverless using Reinforcement learning

Master Thesis

Rosen Kosakov (9605630)
Graduate School of Natural Sciences (GSNS)

Utrecht University

Dr. Nishant Saurabh
First supervisor

Dr. ing. Georg Krempl
Second supervisor

Diogo Landau, MSc
Daily supervisor

2 March 2025

Abstract

Serverless computing is a novel paradigm of cloud computing that allows cloud users to reduce underlying infrastructure down to the ”function” level of an application. Although, the function provides lightweight and efficient resource management, such control of resources comes at the expense of increased latency. The cold start problem might account for as much as 80% of the total latency, which creates the need for application optimization. This Master thesis proposes a two-tier approach for cold start mitigation utilizing Q-learning and K-means addressing latency, CPU, memory usage, and adherence to Service level objective. The Q-learning model takes into account the shortcomings of the existing approaches and is devised with adaptive to the latency level rewards due to the dynamic nature of the function invocations and uses multiple Q-tables to avoid long-size tables. The K-means clusters the invocations based on low, medium, and high latency. The method is trained with real-life function invocations from Huawei cloud and an evaluation dataset that was generated from function invocations in OpenFaaS. The method performed with 0.40 average reward per iteration during training and 0.46 and 0.47 during evaluation for one-by-one function invocation and parallel invocation respectively. In addition, the method proved cost efficiency by completing an iteration for around 5 seconds, required less than 8% CPU, and around 77 MB of memory for over 50 training iterations.

Contents

1 Introduction **1**
1.1 Motivation and Problem Statement . . . 1
1.1.1 Motivation . . . 1
1.1.2 Problem statement . . . 2
1.2 Research questions . . . 3
1.3 Literature Review Procedure . . . 5
1.4 Research methodology . . . 5
1.5 Threats to validity . . . 5

2 Backgroud **7**
2.1 Cloud Computing . . . 7
2.1.1 Characteristics . . . 7
2.1.2 Deployment models . . . 8
2.1.3 Service models . . . 9
2.2 Serverless . . . 9
2.2.1 Defintion and Application . . . 9
2.2.2 Advantages . . . 10
2.2.3 Limitations . . . 11
2.3 Cold start issue in Serverless . . . 12
2.4 Function as a Service (FaaS) . . . 13
2.4.1 Definition . . . 13
2.4.2 Advantages . . . 14
2.4.3 Limitations . . . 14
2.5 Reinforcement learning . . . 15
2.5.1 Defintion . . . 15
2.5.2 Significance in Cloud computing . . . 15
2.6 Q-Learning . . . 16
2.6.1 Key concept . . . 17
2.6.2 Applications and Variations . . . 17
2.6.3 Challenges and Improvements . . . 18
2.7 Serverless platforms and Tools . . . 18
2.7.1 AWS Lambda . . . 18
2.7.2 Azure Functions . . . 18
2.7.3 OpenFaaS . . . 20
2.7.4 OpenWhisk . . . 21

3 Related work **22**
3.1 Frameworks for Cold Start . . . 22
3.2 Function Fusion . . . 24
3.3 RL-based approaches . . . 24
3.4 Other ML-based approaches . . . 26
3.5 Other approaches . . . 27
3.6 Summary of Limitations . . . 29

4 Architecture **32**
4.1 High-level architecture . . . 32
4.2 K-means clustering algorithm . . . 32
4.3 Q-Learning agent . . . 33

5 Method **35**
5.1 Q-learning algorithm . . . 35
5.1.1 Parameters and Notation . . . 36
5.1.2 Actions (A) . . . 38
5.1.3 Environment States (S) . . . 39
5.1.4 Adaptive Reward functions (R) . . . 40
5.1.5 Q-tables (T) . . . 41
5.2 K-means Algorithm . . . 42

6 Implementation **46**
6.1 Data source and preprocessing . . . 46
6.2 Q-leaning algorithm setup . . . 47
6.2.1 Simulation Environment . . . 48
6.2.2 Q-learning algorithm training . . . 48
6.2.3 Q-learning algorithm evaluation . . . 48
6.3 K-means clustering . . . 48
6.4 Tools and Frameworks . . . 48
6.5 Performance monitoring . . . 49

7 Experimental design **51**
7.1 Goal . . . 51
7.2 Experimenatal Setup . . . 51
7.2.1 Hardware . . . 51
7.2.2 Software . . . 51
7.3 Source of Data . . . 52
7.3.1 Huawei Public Cloud Trace 2025 . . . 52
7.3.2 Custom tables . . . 53
7.4 Methodology . . . 53
7.4.1 Data preprocessing . . . 53
7.4.2 Simulation of Environment . . . 54
7.4.3 Training phase . . . 54
7.4.4 Evaluation phase . . . 54
7.4.5 Metrics . . . 54
7.4.6 Experimental scenarios . . . 56
7.4.7 Tools for Monitoring and Analysis . . . 56

8 Results **57**
8.1 Evaluation datasets . . . 57
8.1.1 Latency per invocation . . . 57
8.1.2 CPU usage per invocation . . . 58
8.1.3 Memory usage per invocation . . . 60
8.1.4 Average Failure rate to SLO adherence rate per invocation . . . 61
8.2 Accuracy of Method . . . 64
8.2.1 Average reward during training . . . 64

8.2.2 Sum of Q-values per iteration during Training 64
8.2.3 Best state/action for sequential function invocation 65
8.2.4 Best state/action for parallel function invocation 66
8.2.5 Average reward during evaluation 66
8.3 Cost efficiency for Method 67
8.3.1 Latency during Training per Iteration 68
8.3.2 CPU usage during Training per Iteration 68
8.3.3 Memory usage during Training per Iteration 69

9 Discussion and Conclusion **70**
9.1 Limitations 71
9.2 Future work 72

List of Figures

2.1 Serverless Architecture [9] 10
2.2 The Cold Start Latency Process [25] 13
2.3 FaaS architecture [59] 14
2.4 RL Approach [38] 15
2.5 The Q-learning workflow[1] 16
2.6 Explanation of Bellman equation[2] 17
2.7 AWS Lambda architecture [11] 19
2.8 Event Handling in Serverless with Azure3 19
2.9 OpenFaaS framework architecture [41] 20
2.10 OpenWhisk platform architecture [5] 21

4.1 High-level architecture 33
4.2 Architecture of K-means clustering algorithm 33
4.3 Architecture of Q-learning agent 34

5.1 The process of working of Q-learning agent 35

6.1 Overview of the Implementation of the Method as a Function 47

8.1 Average latency per sequential function invocation 58
8.2 Average latency per invocation parallel function invocation 59
8.3 Average CPU usage per sequential function invocation 60
8.4 Average CPU usage per invocation parallel function invocation 61
8.5 Average Memory usage per sequential function invocation 62
8.6 Average Memory usage per parallel function invocation 62
8.7 Failure rate to SLO adherence per sequential function invocation 63
8.8 Failure rate to SLO adherence per invocation parallel function invocation 63
8.9 Average reward during Training per iteration 64
8.10 Sum of Q-values during Training per iteration 65
8.11 Average reward during evaluation per iteration with sequential function invocation 67
8.12 Average reward during evaluation per iteration using parallel function invocation dataset 67
8.13 Latency in seconds of Method during Training per Iteration 68
8.14 CPU usage in % of Method during Training per Iteration 68
8.15 Memory usage in MB of Method during Training per Iteration 69

[1]https://shorturl.at/NFDLV

Acronyms

API Application Programming Interface. 1, 13, 14, 18

AWS Amazon Web Services. 1, 9, 18

CI/CD Continuous Integration and Continuous Delivery. 14

CLI Command Line Interface. 10, 18

DRL Deep Reinforcement Learning. 26, 30

FaaS Function as a Services. 11, 13

GUI Graphical User Interface. 18

HTTP Hypertext Transfer Protocol. 18

IaaS Infrastructure as a Service. 9

PaaS Platform as a Service. 1, 9, 10

QoS Quality of Service. 26

REST Representational State Transfer. 10

RL Reinforcement Learning. 2, 3, 15

SaaS Software as a Service. 9

SARSA State–Action–Reward–State–Action. 16, 30

SDK Software Development Kit. 10

TCN Temporal Convolutional Network. 26

UI User Interface. 10

VM Virtual Machine. 13

Chapter 1

Introduction

1.1 Motivation and Problem Statement

1.1.1 Motivation

Serverless computing is a relatively new paradigm in cloud computing that enables cloud users to abstract away the underlying infrastructure down to the level of a function in a certain application. Compared to typical cloud services such as Platform as a Service (PaaS), Serverless lets developers focus on the functionality of the service rather than on environment and system challenges such as fault tolerance and scaling [44].

Serverless functions are being turned on by events such as HTTP requests. If a function does not get used for an extended period, the platform discharges the resource associated with it. This enables lightweight and effective resource management. However, such resource control can come at the expense of increased response latency. The resources, being unused, are idle. As a result, the platform must enact an execution environment for function invocations that are not currently utilized, which increases the response latency of the functions. In practice, the time spent setting the execution environment is referred to as cold-start latency or problem [46].

According to Fuerst and Sharma (2021) [23], the cold start problem might be as much as 80% of the total latency, which creates the need for optimizing in FaaS applications. In fact, a variety of application scenarios such as machine learning, video processing, and Internet of Things can benefit from such optimization (Liu et al., 2023) [46].

One example is AI-powered chatbots that are widely used in customer support and care. Serverless environments are great for chatbots as they remove the requirements for a full backend, resulting in faster responses [40]. Serverless bots might manage multiple users while maintaining constant performance due to their intrinsic scalability [17]. The design utilized the Serverless Microservices method and was based on Amazon Web Services. The chatbot is built on a Serverless Microservice architecture utilizing AWS, including API Gateway, Lambda, DynamoDB, Simple Notification Service (SNS), and CloudWatch [40].

Another example is the Internet of Things (IoTs) which are characterized by unstable data bursts from several devices. Maintaining a consistent server arrangement for unpredictable data intakes can be a big challenge. Serverless technologies are particularly good at processing and storing large amounts of data quickly and efficiently without manual intervention [17]. Serverless computing has been used in several IoT applications [30]. For instance, the OpenWhisk platform's serverless functionality can be used to analyze photos recorded by a home monitoring camera. When a camera identifies an interesting item, such as a car or a person, it sends the

image to a serverless platform for analysis. A serverless function is used to extract features and communicate the results to users [29].

Several research papers have been published in this direction. A variety of different solutions have been proposed. The most notable ones are caching frameworks, and function fusion [25]. However, approaches do not account for the dynamic nature of cloud computing. Supporting this claim, Vahidinia et al. [77] state that taking into account the dynamic environment of the cloud, predicting the invocation of functions over a period of time must not be ignored. This implies the usage of Machine learning-based algorithms. Solutions that give fixed mechanisms to mitigate the cold start latency may give the most optimal result in a dynamically changing environment. Because of the uncertainty of function invocation times and the changing nature of cloud environments, there is a need for methods that self-adapt to provide automated decisions and monitor the environment during execution time. Reinforcement Learning which is a type of Machine learning, is particularly good at solving decision-making problems and seems to provide a dynamic self-adaptivity needed for a cloud environment. To get to know the existing approaches in detail a multivocal literature review has been conducted.

This research project aims at building an algorithm that is based on reinforcement learning. The research also considers already proposed solutions for mitigating the cold start problem, upon designing an algorithm. In addition to designing an algorithm, the solution will be implemented with an open-source framework such as OpenFaas and it will be tested and evaluated with other solutions. Upon evaluation of the algorithm, several metrics will be considered. These can include mitigation of cold start latency, cold start occurrence reduction, and cost and efficiency of the solution. The newly implemented algorithm will be compared with already existing ones. Some approaches have been designed to reduce the cold start latency.

1.1.2 Problem statement

As Serverless is gaining popularity among cloud-based solutions, the cold start problem is considered to be a huge burden when it comes to the performance of applications[2]. A latency of up to a few milliseconds can be detrimental to some cloud applications such as video streaming platforms. This is referred to as cold start latency. What's more, the occurrence of cold start latency also slows down the workflow. Several solutions have been designed to mitigate cold start latency. However, they do not account for how dynamically changing cloud environment. For this reason, Machine learning approaches can consider this changeable environment and adapt to it, in particular Reinforcement Learning (RL) algorithms. What's more, several RL existing approaches have been considered and their limitations are evaluated, along with some suggestions for future improvement, in order to come up with a novel solution.

Taking into account the need for an adaptable algorithm for the rapidly changing cloud, this research project aims at building an algorithm that is based on reinforcement learning. The research will also consider already existing solutions such as the Q-learning algorithm, and multi-agent RL algorithm for reducing the cold start problem, upon designing an algorithm. In addition to devising an algorithm, the prototype will be implemented with an open-source framework such as OpenFaas and it will be tested and evaluated with other solutions based on evaluation metrics

such as cost and cold start latency reduction.

1.2 Research questions

To address the motivations and problems described in Section 1.1, a set of research questions with corresponding research questions has been defined:

MRQ: How can we design a framework for Serverless computing that reduces cold start latency considering delay, the occurrence of the latency, and improve Service Level Objective (SLO) compatibility that is both efficient and cost-effective?

In order to, give begin answering this question, first an overview of the current methods and techniques is given. For this reason, one research and three sub-research questions have been defined:

RQ1: What are the current methods and techniques for mitigating cold start issues in Serverless computing?

SRQ1.1: How is cold start latency solved by the existing approaches for serverless applications?

SRQ1.2: What are the existing RL approaches for mitigating the cold start problem and what type of latency they are trying to reduce?

SRQ1.3: What are the gaps and limitations in the existing approaches, in particular RL-based approaches for solving cold start problem in serverless applications?

The overview of the methods and techniques showcased various types of approaches. However, Machine Learning approaches more specifically Reinforcement Learning-based approaches have been identified as potential for mitigating the cold start problem to a much bigger extent than the other categories. Moreover, works by [77] also outlines the potential of RL for cold start latency mitigation. The proposed solutions mention cost in the form of memory and CPU usage. Also, SLO improvement has also been mentioned as an important metric that should be taken into account. However, none of the approaches attempts to find the optimal balance between them. For this reason, the RL-based approach will attempt to incorporate the afore mentioned metrics within the reward function of the agent. RL method has been chosen as it solves complex decision making problems. For implementation, two phases have been distinguished: a design phase and an implementation in an open-source framework phase. For this reason, the following set of questions and sub-questions has been defined:

RQ2: How can we utilize Reinforcement Learning (RL) to mitigate cold start latency in Serverless computing?

RQ2 is focused in general on the goal of deploying RL as a tool in order to solve the cold start issue. The question attempts to explore the way RL, with the ability

to learn an optimal policy over some time, thus making an improvement in the system efficiency and resource allocation.

SRQ2.1: How can we design a cost-effective and SLO-aware RL-based framework for mitigating cold start latency in Serverless computing that can advance over state-of-the-art solutions?

The SRQ2.1 narrows the context down to setting up an RL-based method that take into account cost-effectiveness and SLO awareness. The question pose the challenge of reducing the cold start but also optimize resources (CPU and memory usage).

SRQ2.2: How can we implement and integrate an RL-based framework in an open-source serverless platform (e.g. OpenFaaS) considering the results from the previous question (SRQ2.1)?

As the design of the RL method is produced in SRQ2.2, the following step is implementation and integration with open-source framework such as OpenFaaS. The SRQ2.2 focuses on the practical concept of bringing the method in theory into practice. The method has to be integrated and tested making sure the effectiveness benchmark defined in SRQ2.1 is met.

After implementing a prototype for the framework, a validation of the framework shall be executed to determine its strengths and weaknesses. In section 3, different metrics are considered for RL-based approaches such as memory and CPU usage, cold start occurrence, and latency. The validation consists of two main objectives: finding already existing methods to compare the prototype and selecting the most suitable evaluation metrics for comparison. The next set of questions has been defined for this reason:

RQ3: How can we validate a constructed RL-based framework for cold start mitigation?

After the integration of the method, evaluation is required to examine rigorously the effectiveness of the proposed solution. RQ3 makes sure that the optimal evaluation approach is taken to ensure the method is performing up to the standards.

SRQ3.1: What are the acceptable evaluation metrics that can be used for validation?

SRQ3.1 focuses deeply into defining the specific metrics for evaluating the method. The metrics shows how well the method solves the problem by taking into account the resource usage.

These metrics can be grouped into performance and accuracy. Cost-effectiveness, namely CPU and memory usage were monitored closely. Also, the average reward and the sum of Q-values from Q-learning are taken into account.

1.3 Literature Review Procedure

With the introduction of this long proposal, a variety of academic sources were cited These sources were mostly discovered with Google Scholar. For this literature review, the snowballing and backward snowballing approach as described by Wohlin et al. [83] has been used for the usage of sources.

Therefore, the information collected has been validated by other sources that were discovered in Google Scholar when this validation was required. More than 70 relevant sources have been used including more than 25 academically acknowledged papers for related work.

1.4 Research methodology

The research project was be executed over three phases:

- **Preliminary analysis:** In the preliminary phase the objectives of this project have been explained in detail. Additionally, research questions for each stage of the project have been defined. Some threats to validity and limitations have already been defined. A big piece of this phase is conducting a multivocal literature review that results in Sections 2 and 3, describing Background information and Related work respectively. Section 1.3 gives a more detailed explanation of the literature review procedure. This phase attempts to answer RQ 1, SRQ1.2, SRQ1.2, namely gives information on the existing approaches, presents solutions for cold start latency reduction, and gives their limitations or work to be done for the future. After careful evaluation of the existing approaches, along with their limitation, a research gap is defined.
- **Methodology and Implementation:** Based on the findings from the last phase such as limitations from already existing approaches, a high-level architecture (architectural design) is constructed showcasing all the dependecies. Next, the framework was implemented locally, using Python and its Machine learning libraries, and a suggestion for implementation on an Open Source framework such as OpenFaaS was provided. The method is based on Q-learning with a reward function and K-means that incorporates both performance metrics and accuracy metrics. This phase answered RQ2, SQR2.1, and SRQ2.2. It resulted in a prototype that has been built from the conceptual design.
- **Evaluation:** Lastly, the prototype that has been implemented from the last phase was evaluated using different evaluation metrics for accuracy and performance. For this reason, libraries for monitoring and data visualization like Prometheus for monitoring were used. This phase answered RQ3, RQ3.1, and RQ3.2, regarding the comparable methods and evaluation metrics.

1.5 Threats to validity

The research and the approach for creating a framework for cold start mitigation may have limitations and threats to validity. Firstly, the solution was implemented

locally. However, this might differ for commercial tools such as AWS Labda, Azure function as they might work in different ways. As this research project has no access to these commercial frameworks, also due to security reasons, the prototype was implemented locally. Only certain components of the solution were imported to OpenFaaS to showcase that the method is deployable. Therefore, the solution might be limited to open-source frameworks. Secondly, the tradeoff between cost efficiency and accuracy may hinder how effective the algorithm is.

Moreover, a threat to both internal and external validity is that workloads that are used for training the algorithm might not be representative. This is also valid for validating the algorithm. Additionally, generalisability might be affected might be limited if training or testing data used for devising the algorithm is not representative.

Chapter 2
Backgroud

2.1 Cloud Computing

According to the US National Institute of Standards and Technology Cloud computing is "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction".[1] A cloud is formed by a scalable network of nodes [36]. "This cloud model is composed of five essential characteristics, three service models, and four deployment models". 1 It is a typical example of utility computing. In utility computing, hardware, and software resources are concentrated in massive data centers, and computing service customers pay for computation, storage, and communication resources as they are utilized by them [50].

2.1.1 Characteristics

Cloud computing approaches are different from other types of computing technologies because of their particular nature and characteristics. According to [48], cloud computing characteristics are categorized into two categories: basic and essential features. Bellow all 5 characteristics are listed with an explanation provided:

- **On-demand self-service:** This is an important element of cloud computing, which offers consumers computer services such as network storage and server time. Cloud computing services can be offered at any moment based on the request of the user. Even if human interaction is absent, on-demand self-service is applicable [27].
- **Broad network access:** Computing resources are distributed across a network such as the Internet and are used by various client applications running on diverse platforms (such as mobile phones, laptops) at a consumer's location [19].
- **Resource pooling:** The computer resources of a cloud service provider are 'pooled' together in order to serve multiple clients using the multi-tenancy or virtualization models, "with different physical and virtual resources dynamically assigned and reassigned according to consumer demand" [51]. The reason for establishing such a pool-based computing paradigm stems from two key factors: economies of scale and specialization. As a result of a pool-based architecture, physical computing resources such as hardware become 'invisible' to consumers, who have no control or knowledge over the placement, formation,

[1] https://www.govinfo.gov/app/details/GOVPUB-C13-74cdc274b1109a7e1ead7185dfec2ada

and uniqueness of these resources such as database, and CPU. Consumers, for example, do not where their data will be stored on the Cloud [19].

- **Rapid elasticity:** Elasticity has frequently been characterized as a critical feature of cloud computing, allowing deployed application services to react swiftly to changing workloads by acquiring and releasing shared computational resources at runtime. This characteristic provides the basics for cloud economies of scale and improves the usefulness of cloud services. An elastic system may dynamically alter its resource capacity in the face of changing workload circumstances, enabling it to maintain a predefined or tolerable performance threshold in a certain service while incurring minimal operational costs [12].

- **Measured service:** Although computer resources are pooled and shared by several users, in other words, multi-tenancy, the cloud infrastructure can employ applicable techniques to record resource utilization for each individual consumer via its metering capabilities [19]. These types of technologies provide automatic service that monitors and optimizes the usage of a service and conforms to the type of cloud computing resources at a specific level of abstraction. Furthermore, it monitors, manages, and reports on service spending of resources, which leads to the purchase of resources [27].

2.1.2 Deployment models

Cloud computing can be deployed in four main ways: public, private, hybrid, and community [16]. The models differ because they have various characteristics and effects on users. The deployment strategy is determined by the goals and requirements of the company. To choose the most suitable deployment approach, a corporation must do performance, security, and reliability assessments [27].

- **Public cloud:** Public clouds are often considered the optimal deployment option and many users refer to them as clouds. Cloud computing resource providers make public cloud services available to the public and maintain them. Data centers and high-speed networks may be used to deploy cloud systems. The multitenancy of a public cloud is distinguishable; users are different, and their data is not publicly available [32].

- **Private cloud:** Private clouds can be obtained through lease or ownership, with no security requirements, bandwidth constraints, or legal duties. The computer infrastructure in a private cloud is designed particularly for a business and cannot be shared with other organizations [66] When enterprises are not able to host their data remotely, cloud computing providers and customers take advantage of good infrastructure and security management. They chose private clouds for better resource automation and utilization [63].

- **Hybrid cloud:** The hybrid cloud approach is a mixture of the deployment options described above [63]. A management framework assists in providing a single cloud environment in a hybrid cloud. Because of the increasing need for cost, performance, and security, businesses are moving toward hybrid cloud options [27].

- **Community cloud:** The terms "community clouds" and "public clouds" are frequently used together. Community clouds give resources to people and groups with similar goals and interests, whereas public cloud users do not. The infrastructure of computers might be on-site or in a community cloud. Unlike public clouds, which are owned and controlled by a single provider or owner, community cloud resources are managed and owned by multiple members of a community [27].

2.1.3 Service models

Cloud service providers support one or more cloud delivery models. However, there are three main service models in Cloud computing that are discussed in this document:

- **Software as a Service (SaaS):** The user will utilize the provider's applications that are hosted on a cloud infrastructure. The apps are available from a variety of client devices using a thin client interface such as a web browser. Examples are web-based email or Google Docs.
- **Platform as a Service (PaaS):** The user needs to install customer-made or bought applications written with programming languages and tools supported by the provider. Typocal examples are Google App Engine, Microsoft Azure in the cloud infrastructure.
- **Infrastructure as a Service (IaaS):** The user is responsible for procuring processing, storage, networks, and other core computing resources from service providers. An example is AWS.

Recently, some other different service models have been deployed so cloud users to maximize the utility using cloud services [36].

2.2 Serverless

2.2.1 Defintion and Application

The term 'Serverless' refers to a novel generation of PaaS made available by major cloud providers. These new services were first introduced by Amazon Web Services Amazon Web Services (AWS) Lambda, which was initially released in late 2014 [6] and received widespread acceptance in mid-to-late 2016. All of the main cloud service providers, including Google Cloud Functions, Azure Functions, and IBM OpenWhisk, now provide similar offerings [1]. So far, Serverless has been applied to various fields such as video processing [7], scientific computing [68], and machine learning [22]. It is predicted that by 2025, 50% of enterprises will adopt Serveless [24]. In a nutshell, Serverless computing frees developers from the administration of servers [46].

In Serverless, the infrastructure provider is responsible for receiving and responding to client requests, capacity planning, job scheduling, and operational monitoring. Developers just are required to take care of the logic for handling client requests. This is a huge shift from the previous generation of application hosting

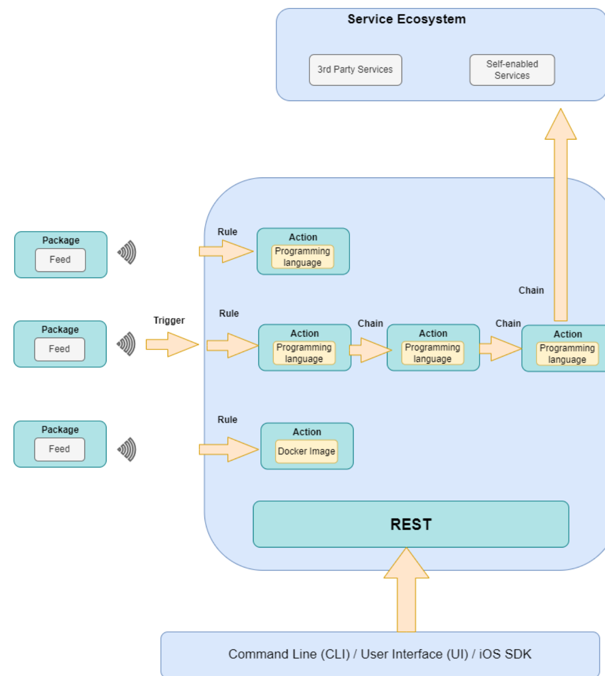

Figure 2.1: Serverless Architecture [9]

PaaS providers. Rather than operating servers constantly, 'functions' are deployed that act as event handlers and only pay for CPU time when particular functions are active [1]. Serverless can be used for more instances of a system besides functions. For instance, Serverless databases, Serverless SQL-as-a-service, Serverless edge computing, and Serverless Big data processing [75].

Figure 2.1 illustrates the architecture of Serverless. Packages that contain feeds generate a trigger which launch launches one or more actions that contain different information for services such as a Docker image or programming language. These actions are based on certain conditions (rule-based). A certain trigger can cause a sequence of actions to be executed (generating a chain of actions). The rule-based actions eventually execute components from the service ecosystem such as self-enabled services. It is also possible that events can be initiated via User Interface (UI), Command Line Interface (CLI), or iOS Software Development Kit (SDK). For this reason, a Representational State Transfer (REST) to communicate with these services and act as a proxy between them and the internal components of Serverless [9].

2.2.2 Advantages

Even though Serverless is thought of as a new paradigm in cloud computing, a lot of work has been done so far for its optimization. These works suggested myriad advantages that greatly improve the cost efficiency, boost up the scalability, reduce the amount of operational overhead, and improve the speed of development. For the rest of this subsection, these benefits are described in more detail.

- **Improved cost efficiency:** Serverless computing offers compelling cost savings, making it a wanted choice. The pay-as-you-go model charges organizations based on real usage of resources, rather than a server capacity [64]. This

approach reduces to a minimum of expenses related to idle server time, allowing organizations to optimize expenditures in real time. Additionally, not having a specialized infrastructure reduces overhead costs to a great extent [73]. According to [43], organizations that use serverless technology save up to 60% on infrastructure administration expenses compared to typical cloud models.

- **Scalability:** Serverless computing has built-in scalability. Cloud providers dynamically scale applications based on demand, resulting in errorless operation without the intervention of the user. This capability is particularly helpful for applications with changing demands for traffic. For instance, serverless technology can help e-commerce websites during sale events and startups with rapid user growth to meet dynamic needs without the constraints and expenses of pre-scaling[73].

- **Reduced operational overhead:** Traditional computing responsibilities, such as server administration and software patching, can get precious resources redirected from key goals. Serverless computing eliminates overhead costs by outsourcing these functions to cloud providers [14]. Deployment is simplified by removing the need for complex server setups. Rollback, or reverting to previous versions of programs in case of faults or difficulties, is much simplified [47]. This lets firms keep continuity in the face of adversity [73].

- **Improved speed of development:** Serverless architectures speeds up software development [62]. By eliminating the requirement to maintain infrastructure, developers can focus fully on developing and revising code, significantly decreasing development cycles [14]. Serverless technologies provide a quick execution environment, enabling quick deployment and iteration. This short turnaround allows organizations to adapt swiftly to market developments, consumer input, and emerging trends [73].

2.2.3 Limitations

Due to the fact that Serverless computing is very new, there are a number of issues that must be addressed. Some of these difficulties, such as security and privacy, are not specific to the serverless computing approach and can happen in any computer environment. Some issues are just corresponding to serverless services, such as cold start, programming, and debugging [72].

When there is no demand for functions and services, serverless computing scales to zero. Scaling to zero causes an issue known as a cold start. A cold start occurs when Serverless functions stay inactive for an extended period of time and require a longer start time the next time they are executed [72].

Since serverless computing is still in its early stages, its tools for development are limited. It presents a challenge to software engineers [72]. Debugging tools for Serverless computing are still not optimal. Monitoring tools are also paramount since developers must monitor the program and examine the performance of functions. More complicated integrated development environments are necessary so that developers may conduct refactoring activities such as merging or dividing routines and reverting functions to a prior version [30]. [49] have identified the lack of debugging tools in three stages. First, log-based debugging is not assisted by most FaaS

platforms. In the absence of mature screening and information extraction tools, the source code of functions is frequently modified, leading to manual analysis and scaling issues. Secondly, during the process of troubleshooting: during development and operation. During operation, cloud functions are executed as black boxes. The analysis suggests that recording the input, context, and output of cloud services is sufficient to recreate issues. During operation, fault management is one of the most challenging tasks in cloud computing [8]. After identifying a mistake, tools shall send alerts to predetermined endpoints and visualize the faults. Thirdly, the assessment requires keeping track of all essential parameters of a cloud function requires additional execution time. Additionally, storing log data requires a logging service and a cloud database [49].

2.3 Cold start issue in Serverless

Serverless computing assigns functions to multiple containers for execution [10]. If the container is ready, the function is directly allocated to this container, and the function is executed. In most cases, if no container is available, a new one is to be created to perform the function. The serverless paradigm disposes of containers after a certain period (τ) to prevent resource waste. This approach is referred to as the scale to zero [21]. To respond to server requests, containers must be restarted after scaling down to zero. Starting a new container and preparing a function for execution results in a specified latency. In the serverless paradigm, this delay interval is referred to as a cold start[42]. The primary reasons for cold starts are:

- To handle the initial request, a new container must be started as there is no ready container on the server. Preparing operations, such as creating a new container and loading necessary runtimes and libraries, may need some time. This leads to a cold start [25].
- The serverless paradigm of dynamic scalability feature allows for automating resource scaling as needed. Rapid growth in requests might create scalability challenges. Delays are another cause of cold starts [25].
- Serverless platforms distribute resources (CPU, RAM, etc.) amongst functions. Additionally, resource over-commitment caused by demand surges might lead to cold start delay [26].

Figure 2.2 depicts the cold start procedure, beginning with container creation and ending with function execution. These methods are essentially similar across all platforms.

- **Loading and Preparation Phase:** When a function request is delivered to the platform, the container shall be ready to run the function. The platform alters incoming data by parsing its headers and input parameters. It assigns resources, including CPU and RAM, to execute the function. To execute the function, start a container that has loaded necessary dependencies such as libraries, and runtime. Function code is delivered to the container and ready to be executed [25].

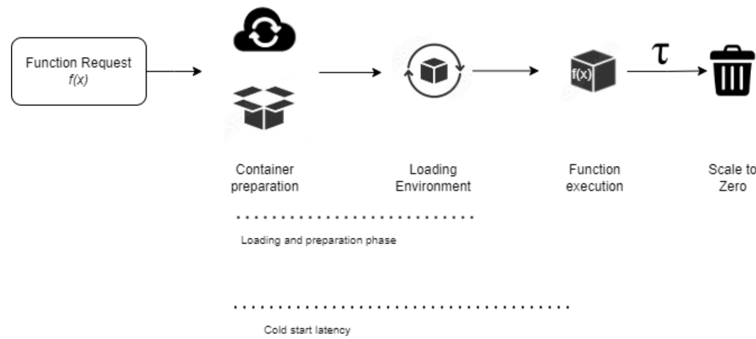

Figure 2.2: The Cold Start Latency Process [25]

- **Function Execution:** After loading and preparation of the container, the assigned function is executed. The platform monitors and controls the resources required to operate the function. This procedure requires external service exchange between the function and the network, including APIs. After execution, a response is built and forwarded to the client. A warm start is considered when keeping resources ready to respond to incoming demands and avoid a cold start. If no requests are received during a certain period, the containers are scaled to zero to save resources [25].

2.4 Function as a Service (FaaS)

2.4.1 Definition

In Serverless platforms, the main pattern of implementation is Function as a Services (FaaS) [4]. FaaS (serverless functions) are event-driven stateless functions. In FaaS, the business logic may be split into separate functions and when a pre-defined event occurs, the function is executed [67]. In this manner, FaaS applications are built by developers who build the application as a mix of Serverless services. The underlying serverless systems manage resources autonomously. As a result, developers are not responsible for the management of resources such as servers of VM instances so that FaaS applications to be operated. Instead, in the case of events, for instance, HTTP requests or a message published in the message queue might be configured [4]), that activate Serverless functions resources are being distributed dynamically. If a serverless function is not utilized for an extended period of time, the platform will reallocate the resources. This enables lightweight and effective management of resources [46].

Figure 2.3 depicts a generic FaaS architecture as well as the technological process between the components. There are two main components the Client that is external and the Cloud Function. The Client communicates with the Cloud function via API Gateway. The Client sends a request which triggers an Event with a certain context. Then, the flow is forwarded to the Function which communicates back and forth with the Backend. If a Result is found, it is returned to the Client via API gateway. [59].

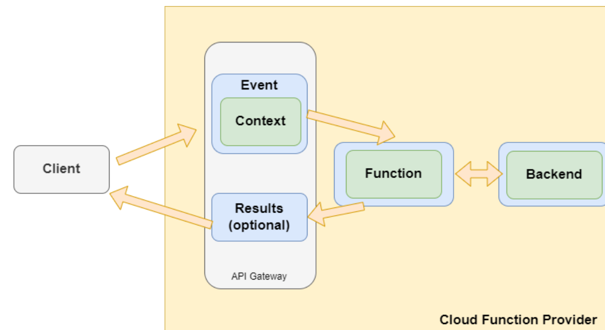

Figure 2.3: FaaS architecture [59]

2.4.2 Advantages

There are a number of benefits when using FaaS. Similarly to Serverless, it offers a flexible pricing model and on-demand scalability. These features have been discussed in the previous section. Moreover, cloud users benefit from a development model, no blackened maintenance, and quick provisioning:

- **Development model:** FaaS enables developers to construct cloud applications in high-level languages and provides API/CPI for packaging code and dependencies and deploying them on the platform, facilitating Continuous Integration and Continuous Delivery (CI/CD) [61].

- **No backend maintenance:** The FaaS approach transfers all back-end management from the application developer to the FaaS platform, which is in charge of setup and maintenance of underlying resources, as well as scalability [61].

- **Quick provisioning:** FaaS solutions leverage complex virtualization methods, such as containers, to provision fresh instances of the application in the range of tens of milliseconds[61].

2.4.3 Limitations

- **Vendor lock-in:** In cloud computing, vendor lock-in occurs when clients are dependent on a single provider's technological implementation. Customers may face large expenses and legal obstacles when switching vendors [57]. Migrating vendors in FaaS can be costly if the language runtime is not cross-vendor compatible [28]. FaaS infrastructures often connect well with a vendor's Serverless platform, but may not interact well with third-party services or other vendor platforms. Interoperability and portability are important for FaaS infrastructures and a significant burden to cloud computing adoption [57].

- **Complex workflows:** Allowing developers to make a combination of functions improves program modularity and promotes the reuse of routines. Research shall concentrate on finding appropriate composition models for FaaS, expressing these functions, and supporting function updates and hybrid-cloud deployments [78].

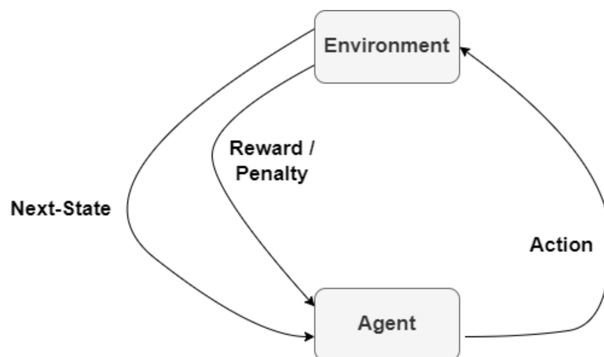

Figure 2.4: RL Approach [38]

- **Performance isolation:** Current FaaS platforms apply containerization and performance optimization approaches. However, these strategies make the performance isolation of the functions weaker along with their accompanying invocations. Reducing performance isolation and understanding its trade-off are open problems [78].

2.5 Reinforcement learning

2.5.1 Defintion

Reinforcement Learning (RL) is an area of Artificial Intelligence algorithms that consists of rewards, an environment, and agents. Figure 2.4 describes the RL model that dynamically calculates the best concurrency for each job (Action). To complete an assignment, the agent must switch its own state as well as the condition of its surroundings. Every task is either rewarded or penalized based on how it contributes to the achievement of a goal. The assumption that agents learn good decision-making rules through a series of trial-and-error interactions with a specific environment is formalized by RL. By taking into account the input obtained from a sequence of behaviors, the agent may form rules that optimize reward. Furthermore, RL requires no previous knowledge about future workloads and adjusts to changes dynamically during runtime; the algorithms have been validated in research as valid approaches to solving challenges related to cloud computing [65][84].

2.5.2 Significance in Cloud computing

It has also been shown that RL offers significant promise for automatically addressing decision-making challenges in complicated and uncertain environments. It appears to be a promising approach for regulating methods in cloud computing when compared to existing and more static-rule-based strategies mixed with time-series data analysis. There are various performance-related concerns. There is no need for human intervention since policy learning is performed via contact with the environment. Dynamism and adaptability are encouraged since learning is continual and the rules that are generated may be updated in response to changes in the cloud environment.

Several RL-assisted autoscaling techniques for cloud settings have been proposed in academic literature. The majority of them utilize model-free techniques, with

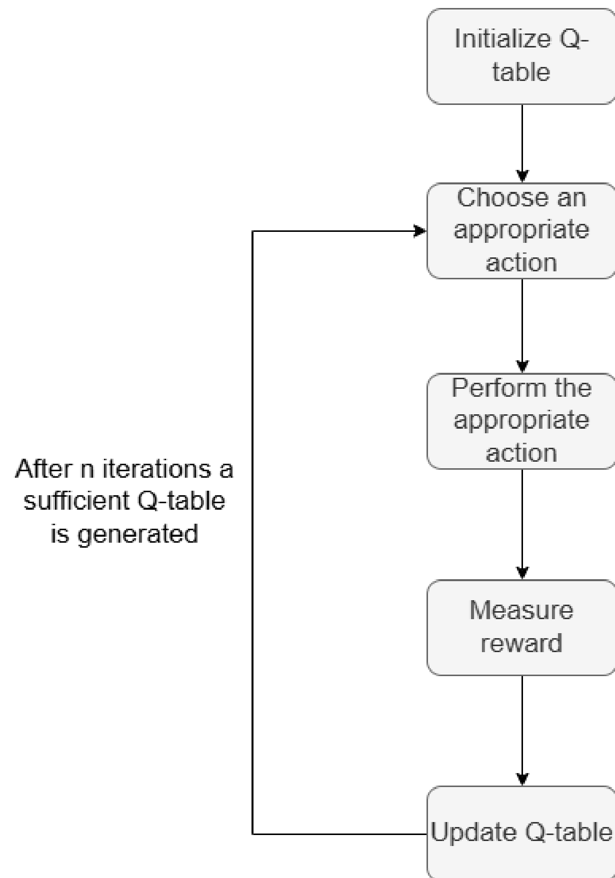

Figure 2.5: The Q-learning workflow[2]

Q-learning, Deep Q-learning, and SARSA being the most often used in RL-based algorithms. In certain cases, an RL environment based on Q-learning is examined; nevertheless, only a few of the existing methodologies incorporate testing over actual infrastructure or serverless settings [65][84].

2.6 Q-Learning

Q-learning is a key algorithm in the field of Reinforcement Learning, part of Machine learning focusing on training intelligent agents to make a sequence of decisions through trial and error. This model-free reinforcement learning algorithm allows agents to learn the most optimal selection of actions in finite Markov decision processes (MDPs) without a predefined model of an environment. It was first introduced by Chris Watkins in 1989 [31][52].

Figure 2.5 presents the general workflow of Q-learning. First, Q-tables are defined. Then an action is chosen and executed by the agent. A reward has been calculated and given to the agent which updates the Q-table with a Q-value. Then the process of selection an action to update the Q-table is repeated for several iterations until a good Q-table is created.

[2]https://www.datacamp.com/tutorial/introduction-q-learning-beginner-tutorial

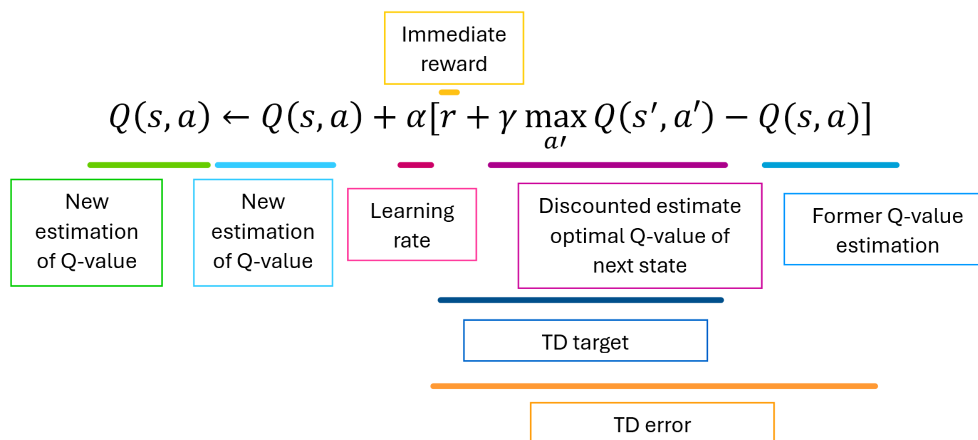

Figure 2.6: Explanation of Bellman equation[2]

2.6.1 Key concept

The key idea of Q-learning is learning the quality of actions known as Q-values. They showcase how good a certain action from a particular state is [31]. The Q-values are updated in an iterative way using the Bellman equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{\alpha'}\ Q(s',a') - Q(s,a)]$$

Where:

- $Q(s,a)$ - the current Q-value for state s and action a[31].
- α - the learning rate that determines the extent of new information overriding old information[31].
- r - the reward that has been received after taking an action a in state s[31].
- γ - a discount factor that creates a balance between the importance of immediate and future reward functions[31].
- $max_{\alpha'} Q(s',a')$ - the maximum Q-value for the next state s'[31].

The explanation of the Bellman equation is presented in Figure 2.6. The equation is used to estimate a new Q-value. The equation itself consists of two main parts an estimation of the new Q-value and a measure of Temporal Difference (TD) error.

The update rule encloses the temporal difference (TD) way of learning, in which the agent learns from both the estimated future rewards and immediate rewards[31].

2.6.2 Applications and Variations

A Q-learning algorithm has a wide application in domains such as autonomous systems, game playing, and robotics. One known variation of Q-learning is Deep Q-Learning (DQN), which is a combination of deep neural networks and Q-learning for handling high dimensional state space[31].

2.6.3 Challenges and Improvements

Even though Q-learning is a widely adopted approach, it has challenges associated with overestimation of Q-values and slow convergence rate. Recent publications have suggested approaches like Double Q-Learning and prioritization of experience addressing these shortcomings [31].

However, Q-learning has remained a powerful approach in RL, giving a framework for agents to learn the most optimal policies as they interact with the environment. The simplicity and versatility of Q-Learning make it a desirable choice in many AI applications[31].

2.7 Serverless platforms and Tools

Major cloud providers have commercialized serverless technologies, such as AWS Lambda and Microsoft Azure Functions. However, many commercial serverless systems hide the platform's fundamental characteristics and provide software developers with vendor lock in issues. To solve these constraints, the serverless computing community has developed open-source serverless platforms such as OpenWhisk and OpenFaaS.

2.7.1 AWS Lambda

The most well-known serverless platform is AWS Lambda. AWS Lambda provides developers with various interaction options, including the Command Line Interface (CLI), Hypertext Transfer Protocol (HTTP)-based Application Programming Interface (API), and Graphical User Interface (GUI). Furthermore, software developers can get access to the platform via language-specific client libraries by using associated IDE plugins, such as Visual Studio and Visual Studio Code [81].

Figure 2.7 depicts an architecture overview. The service is often divided into two planes: control and data. The control one is responsible for is in charge of management, while the data one monitors resources and invocations. The Control plane manages the API for establishing and changing functions, as well as integration with other cloud services such as transmitting Amazon S3 events or polling Simple Queue Service(SQS) queues. The Data Plane handles the invocation of the function. The important control component in the plane is the Invoke service, which distributes execution environments for function invocations. Event-triggered invocations are sent to the Invoke and may be queued. Synchronous invocations require additional management to react to calls, which is controlled by a load balancer component [11].

2.7.2 Azure Functions

Azure Functions provides various interaction methods such as CLI, API, and GUI and access the platform via associated plugins for Visual Studio and Visual Studio Code. Furthermore, Azure Functions gives three hosting levels for serverless applications: consumer, premium, and dedicated. Microsoft offers a general-purpose Azure Marketplace that includes Serverless applications [81].

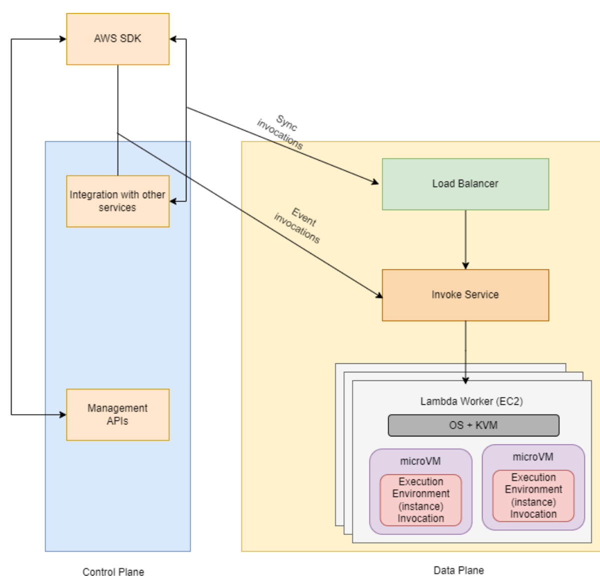

Figure 2.7: AWS Lambda architecture [11]

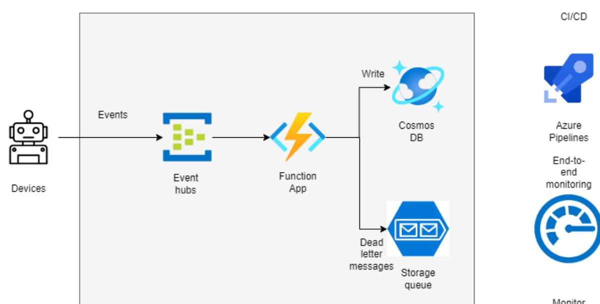

Figure 2.8: Event Handling in Serverless with Azure3

Figure 2.8 depicts a Serverless, event-driven architecture that obtains a stream of data, processes it, and stores the results in a back-end database. Events come from Azure Event Hubs. A Function App is started to handle the event. The event is logged in an Azure Cosmos DB database. If the Function App does not correctly save the event, it is saved in a Storage queue and handled later. **Event Hubs** are in charge of ingesting data streams. Event Hubs are responsible for high-throughput data streaming applications. **Function App** is a Serverless computing alternative approach. It follows an event-driven approach, in which an activator invokes a piece of code (a function). Azure **Cosmos DB** is a multi-model database service that can be used without the of a server. For this reason, the event-processing function saves Java Script Object Notation (JSON) records. Dead-letter messages are stored in the **Storage queue**. If an error appears while processing an event, the function keeps the event data in a dead-letter queue for future processing. **Azure Monitor** monitors performance indicators for the Azure services installed in the solution. Visualizing them in a dashboard provides visibility into the health of the solution. **Azure Pipelines** is a CI/CD service that builds, tests, and delivers applications [3].

[3]https://shorturl.at/quTV3

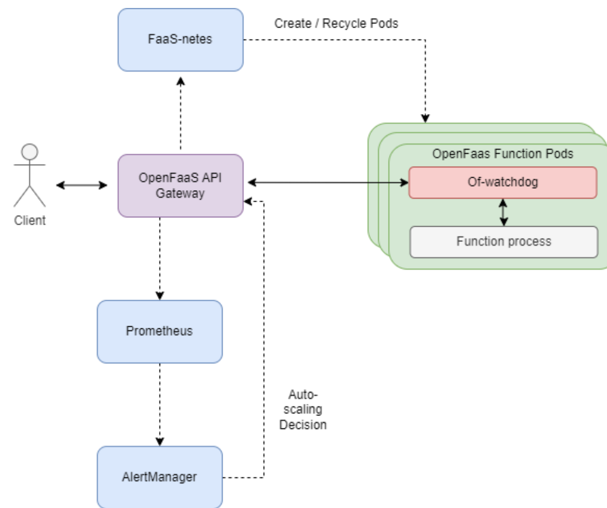

Figure 2.9: OpenFaaS framework architecture [41]

2.7.3 OpenFaaS

OpenFaaS's fundamental technology is built on Docker and Kubernetes. Because of its lightweight structure, OpenFaaS may be installed in public or private clouds, as well as on edge devices. Developers in OpenFaaS can exploit CLI, API, and GUI to create interface actions, but there are no corresponding development IDEs available. CLI is typically used by developers to interface with the OpenFaaS gateway. This gateway is linked to Prometheus, an external function monitor tool that keeps track of the values of function-related metrics. Furthermore, OpenFaaS provides workflow management via synchronous and asynchronous function chains, parallelism, and branching. OpenFaaS's default timeout is 30 seconds. The marketplace for OpenFaaS is the OpenFaaS Function Store [81].

Figure 2.3 illustrates the essential components of OpenFaaS. The API gateway allows for access to functionalities and tracks traffic data. Faas-netes are in charge for managing OpenFaaS function pods. Prometheus and AlertManager are exploited for auto-scaling. Function pods have a single container that runs two workflows: of-watchdog and function. Of-watchdog is a small server that handles incoming requests and redirects them to the function process. Of-watchdog supports three modes: HTTP, streaming, and serialization. In HTTP mode, the function process forks just once and remains warm throughout the function pod's life cycle. Streaming and serializing modes build new function processes for each request, leading to higher cold-start latency and poor performance. OpenFaaS supports auto-scaling that depends on requests per second (RPS). Prometheus gathers traffic measurements from the API gateway. AlertManager receives the RPS metric from Prometheus and sends an alert to the API gateway based on the auto-scaling rule given in the configuration file. The API gateway controls the alert and notifies Faas-netes to scale up or down function replicas. The open-source version lacks the scale-to-zero capability, which is exclusive to the commercial OpenFaaS Pro edition [41].

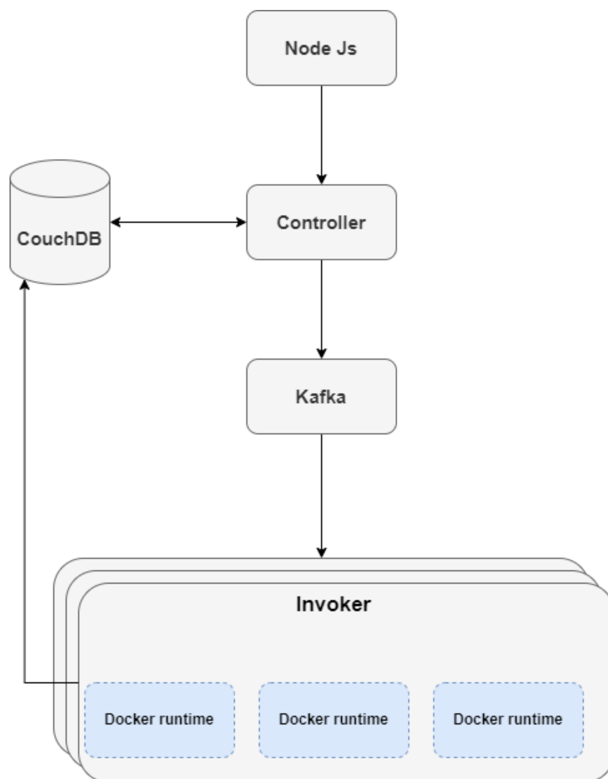

Figure 2.10: OpenWhisk platform architecture [5]

2.7.4 OpenWhisk

Apache OpenWhisk is an open-source serverless platform that lets users execute operations based on an activating event [5]. It incorporates various essential technologies in OpenWhisk, such as Nginx, CouchDB, Kafka, and Docker. Function calls can be changed into HTTP requests and fed into a Nginx server that assists the Web protocol. For monitoring, OpenWhisk relies on third-party technologies such as Prometheus. There is no marketplace for serverless apps in OpenWhisk [81] OpenWhisk uses the Function as a Service (FaaS) paradigm, with functions corresponding to actions and invocations as activations. OpenWhisk assumes a straight link between memory and CPU constraints for containers, therefore developers may only define RAM size for performing activities. Developers can design a chain of activities, with each action calling the previous one [5].

Figure 2.10 illustrates the OpenWhisk architecture, which consists of two essential components: Controller and Invoker which uses Nginx, Kafka, Docker, and CouchDB. Kafka connects and buffers messages for Controllers and Invokers. Moreover, Lean OpenWhisk is ehanced for edge devices, using fewer resources than the full version. Kafka is replaced with an in-memory queue, and the invoker is colocated with the controller. Figure 2.10 presents the OpenWhisk architecture of the platform [5].

Chapter 3

Related work

3.1 Frameworks for Cold Start

Liu et al.[46] proposed FaaSLight as a new approach focused on application optimization. FaaSLight minimizes cold start time by storing only the core application code, addressing one of the main reasons for cold start. Liu et al. report two shortcomings. First, there is a latency that is primarily from reading the lightweight file, which takes roughly 100 ms and is considerably lower than the cold-start delay. Second, to ensure the quality and availability of FaaS applications, FaaSLight may need to consider the varying scope of code execution between languages.

Solaiman et al.[70] proposed WLEC, which is integrated with the traditional lambda paradigm and is available on the OpenLambda Platform, to address cold start latency issues. WLEC utilizes S2LRU++ instead of the common two-queue S2LRU paradigm. S2LRU++ differs from other S2LRU models by using three queues instead of two. These queues contain containers that execute Lambda functions. The CMS (Container Management Service) controls queues for containers depending on utilization, invocation time, wake-up time, status, and other criteria. CMS keeps track of container status and moves, destroys, and initializes them as needed. CMS has three key functions: Initialization, QueuePlacing, OnRequest.

WLEC reduces cold start invocations by 31% and cold start durations by 23.5% on AWS VM. Compared to the fresh start scenario for AWS VM, there is a 31.25% increase in container invocations and a 70.2% reduction in start-up delay. More work can be done to address the cold start issue. Managing large and different containers may be challenging. Compatibility between platforms is also a challenge. Warm up architecture facing the struggle of addressing random container selection in serverless platforms. A warm stack, rather than a warm queue, might be a potential alternative.

Pan et al.[58] proposed SSC, a framework for serverless processes that pre-warms resources and allocates them automatically. The approach is mainly devised to address two challenges in serverless platforms: cold start optimization and resource allocation. The approach of operation for SSC is as follows:

The workflow in the SSC starts with users generating a Serverless workflow on their local machines and then handing it into a serverless platform. The user then pulls essential structural information from the process. SSC evaluates past invocation data before each workflow execution to recognize trends in invocation rates. SSC applies the Improved Gradient-Based Pre-warming Optimization Algorithm to automatically start or end a set number of containers, reducing cold-start issues and wasteful resources. After each execution, SSC applies the Critical Path Optimization Algorithm. SSC exploits a critical route-based method to identify a simplified path that greatly improves overall performance. SSC changes all critical path functions using a priority queue-based method to improve resource allocation for future execu-

tions. SSC carries out the serverless workflow on the serverless platform, removing the need for users to manage resource settings after submission [58].

Pan et al.[58] have also conducted an experiment for validation that resulted in 30% cost savings and a nearly 8% reduction in cold start hit rates, aiding energy conservation and sustainability in computing.

Daw et al.[18] proposed a tool, Xanadu, that reduces cold starts in chaining functions by providing resources as needed. It offers several instantiation alternatives based on runtime isolation needs and enables for function chaining with or without explicit workflow instructions. Xanadu's improvements for cascading cold starts rely on just-in-time resource supply. Daw et al. [18]'s implicit chain detection approach removes overhead latencies for standalone coupled functions. Xanadu performs better than Knative and OpenWhisk, lowering cold starts to a single event and removing costly resource pre-deployments. For future work, Daw et al. [18] identified three aspects for improvement: reducing prediction miss penalties, reducing function keep-alive time, and integrating Xanadu with other platforms.

Another approach by Hall et al. [29] that is WebAssembly-based provides a middle ground between running Serverless applications as stand-alone processes on the Operating System and in containers. WebAssembly's language and runtime features make it a possible option for container-based runtimes for Serverless architecture. WebAssembly is a binary format that makes sure that memory and execution are safe using language features and Runtime with robust sandboxing features. The criteria demonstrated regular performance across all access patterns. While execution speed is lower compared to container-based native binaries, when the cold start latency of containers is considered, the overall performance is faster. WebAssembly is a new technology with the potential to achieve native execution rates. Hall et al. [29] have identified three areas of improvement in the future: the creation of a full prototype for Serverless, custom runtime, and benchmark at scale.

Mistri et al. [53] proposed UniFaaS, a prototype edge-serverless platform that employs unikernels, which are small single-address-space Operating Systems that only contain the elements needed for a specific application, to perform functions. The outcome is a Serverless platform with minimal memory and CPU usage and with a high performance. UniFaaS is designed for deploying on low-power single-board computers. The UniFaaS platform wins OpenWhisk in terms of performance, memory consumption, and setup time. Mistri et al. [53] used Smart Citizen Kit to showcase the platform's usefulness in developing IoT apps.

Akkus et al. [4] created SAND, a new Serverless computing system that offers reduced latency, resource efficiency, and flexibility compared to tradtionsl platforms. To accomplish these features, SAND uses two important techniques: 1) Application-level sandboxing and 2) Hierarchical message bus. Akkus et al.[4] implemented and installed a comprehensive SAND system. The findings indicate that SAND performs better than leading Serverless solutions. In an image processing application, SAND outperforms Apache OpenWhisk by 43%. For future work, Akkus et al. [4] plan to address limitations concerning isolation of performance and load balancing, non-fork runtime support, and planning to review alternative sandboxing mechanisms. In particular, developing a deployment application functionality and sandboxes over several servers for management system load without increasing latency.

Ustiugov et al. [76] presents vHive, an open-source platform for Serverless experimentation, enabling researchers to explore and develop across the full stack. A

novel snapshot-based Serverless architecture powered by vHive is given the Container managment framework and the Firecracker hypervisor. Starting a function from a snapshot results in a 95% greater average execution time in comparison with memory-resident functions. The larger latency is caused by various page faults when transferring the function's state from disk to visitor memory one page at a time. The study shows that functions visit the same set of pages consistently across several invocations. Based on the insights made, the REAP orchestrator is present, which uses insights to speed up all future function invocations by prefetching the function's working set into the guest memory of a newly loaded function instance. The orchestrator records the working set of pages for a function upon its first invocation. In comparison with the baseline snapshotting, REAP reduces the cold-start latency by 3.7 times on average.

3.2 Function Fusion

Function fusion aims to decrease the cold start latency by combining consecutive functions and seeks to eliminate the cold start of the second function [25]. Lee et al. [39] proposed a Function Fusion-based technique to reduce cold start latency. Combining two successive functions eliminates the cold start delay from the second function, resulting in a single cold start delay. With fusion-based operations, combining concurrent processes eliminates the cold start of the second function. However, with Serverless computing, processing time may rise due to the consecutive execution of the functions. Therefore, a more optimized fusion-based approach for parallel functions is still being researched.

Tirkey et al. [74] proposed a new method for reducing cold start time by fusing functions while considering also branching and parallelism. The suggested solution includes two workflow components: fan-out and conditional branch. To determine the cold start delay time and circumstances for fusing, a problem model that is based on the description was created. The demonstration showed how fusion may improve process speed by lowering cold start latency time. The testing results show that the technique cuts cold start delay time down by approximately half compared to the original procedure. It is thought that the approach plan can lead to a high-quality solution. Additionally, the solution has been tested on a public cloud (AWS Lambda).

3.3 RL-based approaches

Vahidinia et al.[77] identify as important the need to reduce cold start latency in real-time applications as it influences performance and user satisfaction. Serverless systems typically exploit fixed solutions, such as keeping containers warm, which might result in memory waste. To determine the time needed to keep the containers heated, Vahidinia et al.[77] identified the need for a mechanism that analyzes the function invocation patterns. The solution shall compromise between lowering cold start time and memory utilization. For this reason, Vahidinia et al. present a unique two-layer adaptive method to address this issue. The first layer uses a detailed reinforcement learning algorithm that identifies function invocation trends over time to determine the best time to keep containers warmed up. The second

layer uses long short-term memory (LSTM) to predict future function invocation timings and evaluate the need for pre-warmed containers. Experimental findings on the Openwhisk platform show that the suggested strategy decreases memory usage by 12.73% and improves execution invocations on prewarmed containers by 22.65% in comparison with the original platform.

Agarwal et al. [2] present an agent that is model-free Q-Learning which uses per-instance CPU utilization and the number of available function instances to simulate environment states and determines the optimal number of function instances for a given workload. Q-Learning agents learn with trial and error while interacting with the Serverless environment. In every iteration, the agent analyzes the current environment and takes a particular action. The workload pattern is learned through held up feedback, which can be positive or negative, based on causes such as CPU use and response success/failure. This approach responds to deviations in the workload pattern with no prior knowledge, minimizing the frequency of cold starts in subsequent invocations. This solution uses the Q-Learning algorithm to forcast the ideal number of function instances for Serverless environments, reducing the frequency of cold starts over time.

Agarwal et al. [2] concluded that the approach effectively reduces cold start occurrences for a certain function workload and link the gap between successfully fulfilled requests within some iterations in comparison to the default autoscaling method. The solution exceeded the previous autoscaling strategy, getting moderate results with only a few training rounds and completing around 50.1% of requests. Agarwal et al. [2] also identified some future directions for research aiming at expanding the Q-Learning technique by experimenting with other reward structures, α and γ values, and function combinations. It is also possible to incorporate additional elements such as memory configuration and function size into the learning process of the agent to better comprehend the importance of actions in the state space. To avoid state action space explosion, DQNs (Deep Q-Learning Networks) should be used, instead of discretizing continuous values for state representation. This will enable for to estimate optimal actions.

Agarwal et al. [3] introduced another Q-learning technique that actively boot functions up depending on forecasted demand, taking into account indicators such as CPU utilization, current instances, and response failure rates. The solution was implemented on Kubeless and evaluated using a normalized real-world function demand trace with a matrix multiplication workload. The RL-based agent beats Kubeless' default policy and function keep-alive policy, increasing throughput by up to 8.81% while cutting down on computation load and resource waste by up to 55% and 37%, respectively. This is because of the reduced cold start. It is also observed that after 500 epochs of training, the RL-based agent beat other strategies, supporting forecast of a positive relation between success rate and fewer cold starts on the platform.

Agarwal et al. [3] identified some future research directions that will focus on memory usage and function package size for better learning quality and reduce cold starts. In addition to Q-Learning, SARSA, a policy-based approach that assembles quicker than Q-Learning, might be tested for cold start delay problems. The suggested approach, which is a modification of Q-Learning, uses discrete values instead of continuous values to represent the state. Deep Q-Networks and Proximal Policy Optimization could be used to estimate optimal actions without causing state space explosion.

Jeon et al. [33] propose using Deep Reinforcement Learning (DRL) to build customized caching policies for certain workload profiles and Quality of Service (QoS) requirements such as functions' response time in hierarchical edge clouds, including package caching. The suggested DRL-based caching approach depends on experience rather than algorithmic design. Simulation findings show that the proposed DRL-based caching method effectively meets response time QoS requirements.

Qiu et al. [60] propose and implement an altered multi-agent RL method using Proximal Policy Optimization (MA-PPO). In multi-tenant situations, MA-PPO trains each agent until convergence and achieves performance that is equivalent to S-RL in single-tenant scenarios, with less than 10% deviation. MA-PPO boosts S-RL performance (in terms of function tail delay) in multi-tenant scenarios by 4.4 times. Future work shall reflect on difficulties such as quick retraining for new or updated functions, network parameter sharing, transfer learning, and fault tolerance for agent disconnection or RL transition corruption.

3.4 Other ML-based approaches

Nguyen et al.[55] take a more in-depth approach to solving the cold start problem. First, real-world challenges in the Serverless cloud system were discovered. This set is broken down into data sets and predictive model needs. A Temporal Convolutional Network model was devised to anticipate function instances and arrival times 10 to 15 minutes in advance, taking into account the requirements. In addition to these capabilities, an ensemble policy was developed. This technique applies concurrency, low-coupling, and high-cohesion concepts to solve cold-start delay problems at both the function and infrastructure levels, impacting on earlier research. The suggested strategy and publicly accessible scripts can forward additional cloud issues, such as improving the provisioning of virtual software-defined network assets.

A following work by Nguyen et al. [56] is proposed which defines a unique 2-prong cold-start management policy that allows for feedback loops with higher management policies and manages lower-level cold-start optimization strategies. The policy offers two strategic advantages. This improves the integration and coordination of cold-start optimizations at lower levels. Additionally, it allows for higher-level resource management policies to monitor and communicate with the proposed cold-start management policy. Continuous feedback loops between management policies at various system levels are essential for AI-powered autonomous computing systems. Additionally, evaluation methodologies for both the Temporal Convolutional Network model and management strategy are given. The TCN model consistently exceeds leading serverless models on two trace datasets.

Chahal et al.[15] present an approach for the deployment of big models for inference and evaluation of their cost efficiency in an application. Improvements for transitioning applications to a serverless architecture are investigated. Delays in processing batches of various sizes at intervals of 15 minutes are monitored, which results in cold starts for each invocation of the Lambda function. At intervals of 2 minutes, the Lambda function is warm for each call of the batch. It has been revealed that a cold start latency takes longer to alter each batch of input photos than a warm start. This experiment suggests that cold start time is affected by parallelism and batch size. Increasing batch size causes longer cold start latency

time and a bigger gap between cold and warm start times. The future work involves broadening the technique proposed and evaluation of many deep learning frameworks to for execution, including MXNet, PyTorch, and various convolution and recurrent neural network models. Serverless platform options from several vendors have been assessed. The future study will aim at providing a cost-effective solution for managing FaaS resources.

3.5 Other approaches

Mohan et al. [54] suggested using pre-rendering networks to boost up container provisioning. The approach cut down cold start time by loading networks into paused containers via a Pause-Container Pool Manager (PCPM). Mohan et al.[54] identified that the notion of pre-creating or pre-fetching resources can widen beyond network entities. Pre-creating components of function environments and pre-fetching code dependencies using the container manager can make FaaS agile and memory-efficient.

Another concept by Lin et al. [44] that aims at reducing cold starts is having a pool of 'warm' containers on hand in case of future demands. Lin et al. [44] have used Knative which is a Kubernetes-based platform that exploits containers for computing. Lin et al.[44] have reached the conclusion that cold start latency can be classified into two categories: Platform overhead and application-dependent instantiation overhead. To address the cold start latency problem, a pool of warm pods is kept on standby and provided in advance for high-demand functions. This diminishes the cold start overhead. This pool can be shared across multiple services.

Lin et al. [44] showed that adding a pool drastically improves response time. With just one pod in the pool, P95 for both small and large applications is completely cut off, while response time is reduced by at least 50%. With more than 5 services, the improvement in response time is less than 50%. However, data from 1 to 5 services indicate that a big pool is not needed to lower cold start issues. For future work, this formulation may be customized to specific systems, such as multi-warehouse installations on a cluster of servers.

Wang et al. [79] establish their approach to replayable Execution which explores the intensive-deflated execution features of the majority of objective applications. It uses checkpointing to save an image of an app, enabling this image to be shared across multiple containers and resulting in swift restoration at service startup. Wang et al. [79] apply Replayable Execution to a representative FaaS Java framework to device a ReplayableJVM execution, which together with advantages from deterministic execution of a warmed-up runtime, offers 2 times memory footprint cut down, and over 10 times startup time improvement. Wang et al. [79] reveals that future research shall focus on using ROMization to prevent JIT optimizations in the shared codebase, among other warm-up strategies. The aim is to provide a clear optimization border between the ROMized codebase and the divergent execution route that still needs JIT optimizations. Evaluating bigger or more complex user FaaS services might be advantageous. Future research may look at using big return-oriented strategies like the one described here to remove PIE code.

Du et al. [20] propose Catalyzer, a serverless sandbox system approach, that presents resilient isolation and quick function start. Catalyzer avoid critical path ini-

tialization by restoring a virtualization-based function occurance from a well-formed checkpoint image, avoiding the need to start from scratch. Catalyzer improves recovery performance by getting user-level memory and system state on demand. Du et al. [20] propose a new OS primitive, sfork (sandbox fork), to decrease startup time by reusing the state of a running sandbox. Catalyzer minimizes startup costs by reusing state, enabling advancements across multiple Serverless operations. Catalyzer considerably cuts down on startup latency, reaches less than 1 ms latency in the best situation, and significantly decreases end-to-end latency for real-world applications. One important limitation is the management of shared memory. If a child sandbox inherits shared memory, it breaks the separation of parent and child sandboxes. Otherwise, it may change the meaning of MAP_SHARED.

Kaffes et al. [35] discuss a cluster-level scheduler in their work which is centralized and functions on a fundamental level. The scheduler distributes functions to specific cores, making it core-specific. It therefore enriches performance forecasting by directly controlling the sharing of specific cores among parallel running processes. The cluster-level scheduler's centralized architecture provides a big picture of resources, enabling work to be assigned to any core in the cluster. This architecture promotes scalability and flexibility by avoiding core overload and queue imbalances. This decreases the requirement for heavy function migrations away from busy cores. Kaffes et al. [35] suggest a centralized and core-granular scheduler that is capable of managing millions of function invocations per second. The scheduler is expected to improve Serverless computing adoption for latency and throughput-sensitive apps.

Silva et al. [69] concentrate on decreasing the impact of runtime environment setup and loading. Prebaking replaces the normal fork-exec operation with a mechanism that obtains snapshots of formally constructed functions and processes. To test the prebaking approach, Silva et al. [69] devised a prototype with the CRIU checkpoint/restore tool for the Linux kernel and inspected it experimentally. Results showed that using process cloning to start serverless functions reduces JVM startup cost, resulting in a 40% gain for NOOP routines and 47%-71% for more average functions. The suggested strategy enables platform maintainers to engage with the process before saving its state. The findings suggest that prebaking a Java method before improves performance. The prebaking method reduced both JVM startup and JIT overhead. Finally, it has been demonstrated that the advancements are proportional to the function's code size.

In the future Silva et al. [69] would like to broaden their study to combine Node.JS and Python runtimes, which are supported by known public FaaS platforms. Silva et al. [69] want to improve the Prebaking connection of the techniques with other FaaS frameworks and assess its ease of usage with various designs. Silva et al. [69] want to use the latest version of the CRIU tool, which eradicates the need for privileged operations. Additionally, a test in-memory optimization on CRIU to boost snapshot restore time will be done.

Another method by Cadden et al. [13] called SEUSS allows for a fast deployment and high-density caching of Serverless services in a multi-tenant FaaS environment. SEUSS's goal is to provide a caching prototype for Serverless functions that supports several language runtimes, executes functions in isolation, and effectively obtains computational state at the application and system levels. To accomplish these qualities, deploying functions from unikernel snapshots is performed. New executions may be quickly deployed from a snapshot, cutting down on function start time by

avoiding steps like restarting the unikernel, loading the language runtime, and importing and compiling the code and dependencies. SEUSS combines low-delay cold starts and makes it simple for immutable memory caching to orchestrate abnormal request surges that typical approaches can not provide. These findings suggest that the Serverless function is not always a heavyweight but computationally primitive. Serverless functions provide general-purpose computing at any scale due to their capability to deliver high parallel executions in milliseconds.

As future work, Cadden et al. [13] take into account that distributed SEUSS can speed up installations from a distance by allowing for complex sharing approaches such as VM state coloring and on-demand paging. Cadden et al. [13] will use Unikernel Linux to run applications and ensure long-term maintence. To improve SEUSS's security and applicability, future work will involve constructing Unikernel Linux-based functionalities on a custom Virtual Machine monitor.

Kumari et al. [37] propose an integrated model, called an adaptive container provisioning model (ACPM), that reduces cold-start latency using runtime provisioning of containers. ACPM has the potential to reduce the occurrence of cold-starts and latency caused by cold-start, improving the overall performance of serverless execution. The model consists of two phases. In the first phase, an efficient deep learning model, as the long short-term memory (LSTM), has been used to predict the required number of pre-warmed containers in advance, which serves as soon as the function request arrives. This phase is combined with a container placement module for quicker container delivery to minimize the frequency of cold-start delay. The second phase is based on a sandboxing mechanism where a cluster of similar functions is placed in a single container instead of building a separate container for each new request. The second step uses a sandboxing approach to bundle related functions into a single container, eliminating the need for separate containers for each incoming request.

The experiment shows that the suggested ACPM beats alternative techniques in terms of end-to-end reaction time. The cold-start delay and frequency were significantly decreased compared to prior alternatives. The proposed concept offers a systematic approach to container placement, significantly reducing cold-start delays. Future experiments might include other characteristics like throughput, memory, and CPU consumption over time. While LSTM is effective at capturing dependencies in previous patterns, alternative RL models, such as Q-learning, can be used to learn incoming requests and cold-start patterns, reducing the cold-start problem [37].

3.6 Summary of Limitations

The approaches mentioned above showcase various solutions or partial solutions to the cold start latency problem. In this section, the focus is moved toward existing Reinforcement Learning approaches (described in Section 3.3), and a review of their limitations is presented.

Vahidinia et al. [77] presented a two-layered approach composed of two RL approaches that are combined together for better results regarding the reduction of memory consumption and the improvement of the execution invocations on prewarmed containers. However, a randomly generated data set for evaluating the proposed solution, which means that it was not tested on real scenario data. In

other words, a simulation has been initiated to evaluate the prototype. Moreover, the solution has been only compared with OpenWhisk static performance workflow. In addition, the results show that the proposed solution shows pretty similar results to OpenWhisk when it comes to the number of cold start delays. Furthermore, the solution has been evaluated layer by layer, and no simultaneous evaluation of all the components has been performed. Finally, the evaluation only considerers memory cost, no CPU usage has been monitored.

Agarwal et al. [3] noted that expanding action space or state space makes Q-table grow exponentially, which makes Q-learning unfeasible to perform Q-value updates and degraded time and space complexity. What's more, The proposed agent analyzes individual application demand, which limits its ability to generalize to other demand patterns. Additional training is required. Moreover, the agent expects to be guided by approximation, so that random actions to be avoided. The availability of techniques and tools to gather real-time metrics is needed.

Agarwal et al. [3] pointed out that future studies will focus on memory utilization and function package size in order to increase learning quality and avoid cold starts. To support analyses, metrics collection frequency, request queuing, and concurrency policy of functions should be extended. In addition to Q-Learning, SARSA, a policy-based technique that converges faster than Q-Learning, can be used to solve cold start difficulties. The proposed technique, which is a variant of Q-Learning, represents states using discrete values rather than continuous values. Deep Q-Networks and Proximal Policy Optimization can be used to estimate optimal actions while avoiding state space explosion.

After evaluating their solution, Agarwal et al. [2] found out that the CPU-intensive function composition causes significant resource use, which leads to unfavorable effects. This reduces the agent's capacity to learn effectively since it explores other state-action space values, which takes more time to gain the essential knowledge. Discretizing continuous CPU utilization values for state space representation might potentially interfere with the agent's optimal performance using Q-Learning approaches. Furthermore, the vast state-action space extends learning times and influences the agent's knowledge acquisition. Moreover, Agarwal et al. [2] identified approaches for the future for expanding the Q-Learning technique including experimenting with other reward structures, α and γ values, and function combinations. It is also feasible to include other aspects of the agent's learning process, such as memory configuration and function size, to better grasp the significance of actions in state space. To avoid state action space explosion, DQNs (Deep Q-learning networks) could be employed to represent states rather than discretizing continuous values.

Jeon et al. [33] proposed a Deep Reinforcement Learning approach for Serverless edge computing that attempts to improve function response time by creating customized caching policies. The suggested DRL-based caching approach relies on experience rather than algorithmic design. This study eliminates the need for complex stochastic algorithm designs since the DRL will handle the entire learning process automatically. However, it involves Deep Learning techniques which are beyond the scope of this project.

Qiu et al. [60] A multi-agent framework enables RL-based resource management controllers in serverless FaaS applications. A multi-tenancy problem has been identified. In addition, it was proposed that future research should tackle issues including

Approach	**Papers**
Frameworks	[46] [70] [58] [18] [29] [53] [4] [76]
Function Fusion	[39] [74]
RL approaches	[77] [2] [3] [33] [60]
Other ML approaches	[55] [15] [56]
Other approaches	[54] [44] [79] [20] [35] [69] [13] [37]

Table 3.1: Classification of papers based on a cold start mitigation technique

rapid retraining for new or updated functions, network parameter sharing, transfer learning, and fault tolerance for agent disconnection or RL transition corruption.

Chapter 4

Architecture

As discussed in Section 4.1, RL approaches generally consists of an agent and an environment that interact with each other through actions, penalty/reward, and states. Figure 4.1 presents the high-level architecture of the method aiming at creating the best scheduling policy in order to reduce the cold start problem while being cost-efficient simultaneously. The method itself consists of the following components: a Postgre SQL Database, a REST API, a Q-learning-based agent, and a K-means clustering algorithm. The input of the method is a dataset with Serverless traces of function invocations. The output from the Q-learning agent is the best scheduling policy for cold start reduction and metrics that are produced during runtime for the purpose of evaluation of the method including Q-tables. Both the architectures of the Q-learning agent and K-means clustering algorithm have been presented in Figures 4.3, 4.2.

4.1 High-level architecture

Figure 4.1 illustrates a bird's eyes view-point of the method developed as a high-level architecture. Firstly, the user pushes the data in the form of historic Serveless function traces. Then, the data is retrieved by the REST API which pass it to the PostgreSQL database where the traces are stored in tables. Then, the function traces are fed into the K-means clustering algorithm, which clusters the historic function traces into three clusters (low, medium, and high) and each function trace receives a latency label (in a string format). After the function traces are clustered, they are passed to the Q-learning agent for learning the behavior of the function invocations in order to come up with a scheduling policy for cold start mitigation. In addition to the scheduling policy in the form of Q-tables with Q-values, the agent returns evaluation metrics such as average reward per iteration for both training and evaluation stages. What's more, the scheduling policy is evaluated with the metric total sum of Q-values per iteration. The detailed architectures of the Q-learning agent and the K-means algorithm are explained in Section 4.3 and 4.2 respectively and the underlying graphs are illustrated with Figures 4.3 and 4.2.

4.2 K-means clustering algorithm

Figure 4.2 showcases the K-means clustering algorithm whose main goal is to group the data into three clusters based on the latency of the function invocation traces. The traces are taken from the PostGreSQL database via a query. Then, centroids are selected and the Euclidean distance of the data points are calculated and mapped to a cluster with a centroid. The process from the previous steps is repeated until there are no significant differences with the previous iteration. In this way, the

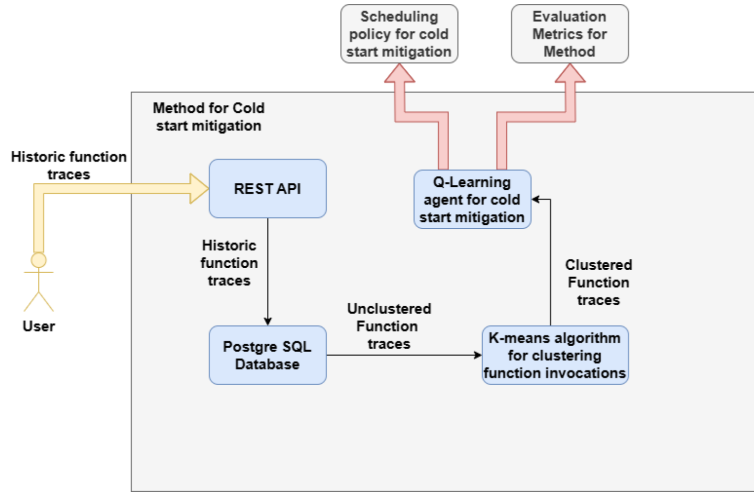

Figure 4.1: High-level architecture

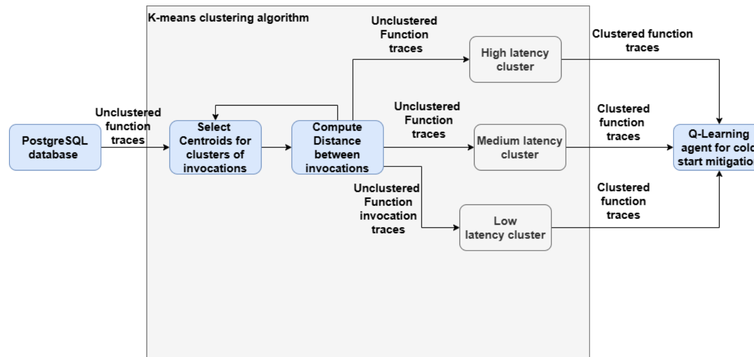

Figure 4.2: Architecture of K-means clustering algorithm

unclustered function traces are grouped into three clusters based on latency (low, medium, and high). The three clusters are then sent to the Q-learning algorithm for the next steps of the method.

4.3 Q-Learning agent

As mentioned in Section 2.5 Q-learning is a type of RL. Every kind of RL has three mandatory components: Agent, Environment, Reward, or Penalty (which in this case is a reward function). The agent takes an action based on a specific state of the Environment and the agent receives a reward that determines the next action. The different component in Q-learning is the creation of a data structure called a Q table that keeps track of the states and actions with Q values. Q-values show which state is more valuable so that a certain action is executed. Q-values are updated using the Bellman equation [80].

Figure 4.3 shows the architecture of the Q-learning agent for the method. First, both the number of actions and states are given to the Q-tables initially so that Q-tables are established. The Q-tables are three: for low latency (T_{low}), medium latency (T_{medium}) and for high latency (T_{high}). Then, the Serverless function invocation environment sends its current state to the Q-Learning algorithm for cold start mitigation and to the Policy for function scheduling which reflects on the current

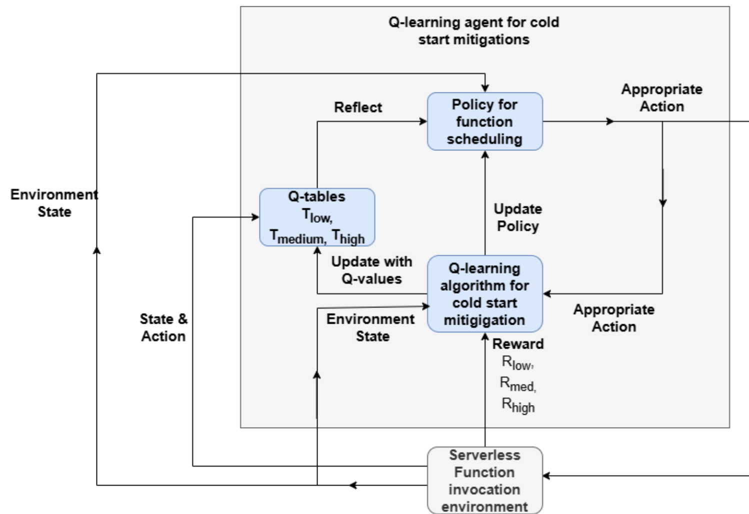

Figure 4.3: Architecture of Q-learning agent

policy. Based on the state, the policy sends an appropriate action to be executed to the agent and the environment to account for the changes. Based on the nature of the latency three reward functions have been defined: for high latency (R_{low}), medium latency (R_{medium}) and high latency (R_{high}). The environment then gives a reward to the agent. The reward is calculated by one of the three functions. Then, the agent updates one of the Q-tables with Q-values and also updates the current policy. The Q-tables reflect the current policy as well.

Chapter 5

Method

The method has been introduced to address solving the cold start problem in Serverless computing. The suggested approach attempts to come up with the best policy for scheduling function invocation based on historical data of Serverless invocation traces. As already mentioned, the proposed method consists of two main tiers a Q-learning algorithm and a K-means clustering algorithm. The Q-learning solution defines an optimal policy through trial and error by taking an action in a specific state while receiving a reward. The K-means creates three clusters for low, medium, and high latency function invocations.

5.1 Q-learning algorithm

A big part of the proposed method is the Q-learning algorithm. Q-learning is a specific RL type which is a model-based, value-based, off-policy algorithm that attempts at finding the best sequence of actions based on the current state of the agent. The "Q" represents quality which in this context means how valuable the action is in maximizing future rewards. As the method receives rewards, it updates its Q-values in the Q-table.

Figure 5.1 presents a process diagram for the Q-learning model. Firstly, 3 Q-tables were defined, a decision was determined and executed in a certain state, based on the latency label (low, medium, high latency), a reward value between 0 and 1 was calculated and a Q-value in the Q-tables was updated using the Bellman equation [80]. The process of selecting a decision and updating a Q-table is repeated until an optimal policy is reached (or model convergence).

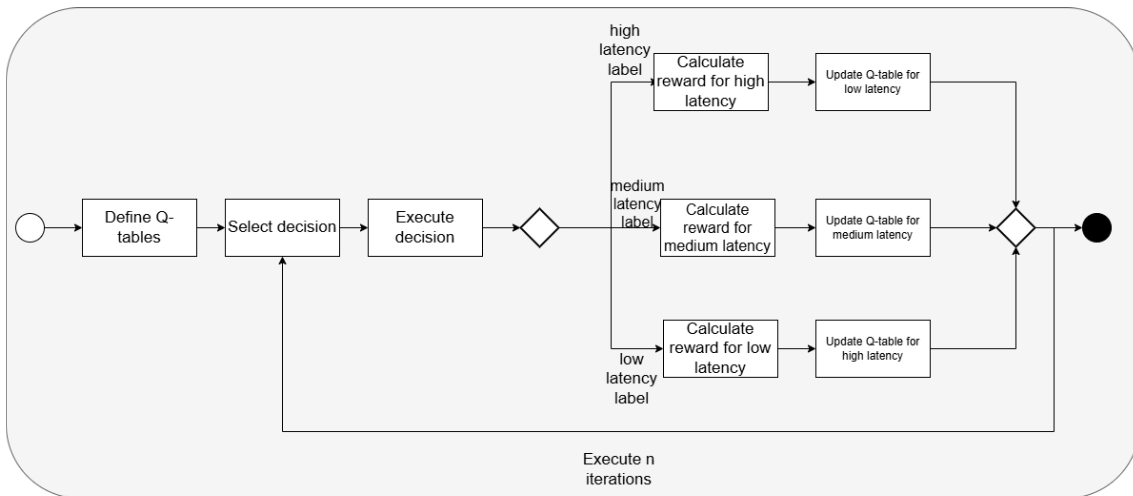

Figure 5.1: The process of working of Q-learning agent

5.1.1 Parameters and Notation

The environment for the Q-learning model is defined as a tuple:

$$\mathcal{E} = (S, A, T, R)$$

Where:

- S : Set of States - The set of all possible states, represented by combinations of CPU and Memory usage levels. There are $|S| = 9$ possible states, defined as:

  $$S = (CPU, Memory)|CPU, Memory \in Low, Medium, High$$

  Each state represents normalized CPU and Memory values, where:

  $$0 \leq CPU \geq 1 \; and \; 0 \leq Memory \geq 1$$

- A : Set of Actions - The set of all actions that are available to the agent, which correspond to changes in CPU and Memory usage. There are $|A| = 6$ possible actions:

  $$A = \left\{ \begin{array}{l} \text{Increase CPU, Decrease CPU, Maintain CPU,} \\ \text{Increase Memory, Decrease Memory, Maintain Memory} \end{array} \right\}$$

- T : Set of Q-tables - a collection of Q-tables that corresponds to a latency label (low, medium, high). The set is defined as follows:

  $$T = t_{low}, t_{med}, t_{high}$$

  Each Q-table maps state-action pairs to Q-values

  $$t(s, a) \in \mathbb{R} \; \forall s \in S, \; a \in A$$

- R : Set of Reward Functions - tailored to the latency label of function invocations. Reward functions are being calculated based on normalized CPU and Memory usage, and adherence to Service Level Objective (SLO). All values must be between 0 and 1:

  $$R = r_{low}, r_{med}, r_{high}$$

  Where:

  $$r_{low} = 0.30 * CPU + 0.30 * Memory + 0.40 * P(SLO)$$

  $$r_{med} = 0.35 * CPU + 0.35 * Memory + 0.30 * P(SLO)$$

  $$r_{high} = 0.40 * CPU + 0.40 * Memory + 0.20 * P(SLO)$$

  Where:

  $$0 \leq r_{low}, r_{med}, r_{high}, CPU, Memory, SLO \leq 1$$

- **Learning Objective**: The Q-learning model aims at maximizing the expected cumulative reward by learning an optimal policy π^*:

$$\pi^*(s) = arg\ max_{a \in A}\ t(s, a), \forall s\ \in\ S$$

The following (hyper-)parameters have been defined for the optimal performance of the Q-learning algorithm using [80]:

- Learning rate (α): $\alpha = 0.70$, so that appropriate control of newly acquired information overriding the existing knowledge is exercised. The value enables the model to adapt quicker without losing previous learning.

- Discount rate (γ): $\gamma = 0.95$, which makes sure that the model emphasizes significant future rewards, encouraging optimization in the long run.

- Exploration epsilon probability (ϵ): $\epsilon = 1.0$ makes sure that the model starts with a high degree of exploration, by randomly selecting actions to discover optimal policies.

- Epsilon decay (ϵ_{decay}): $\epsilon_{decay} = 0.0005$ ensures a gradual reduction of exploration probability (ϵ), by shifting the focus towards exploitation of learned strategies as training progresses.

- Number of Training episodes (n_{train}): $n_{train} = 10,000$, the algorithm is trained over 10,000 episodes, providing enough iterations for the Q-learning model to converge to the optimal policy.

- Number of Evaluation episodes (n_{eval}): $n_{eval} = 100$, after training, the algorithm is evaluated over 100 episodes to evaluate the robustness and performance under various scenarios.

- Probability of adherence to Service Level Objective ($P(SLO)$): The probability of adherence to SLO $P(SLO)$ is calculated as the reverse probability of nonadherence to SLO ($P(SLO')$) :

$$P(SLO) = 1 - P(SLO')$$

  Where:

$$P(SLO') = \frac{Number\ of\ SLO\ violations}{Number\ of\ function\ invocations}$$

  Where:

$$Number\ of\ SLO\ violations = \frac{Number\ number\ of\ invocations\ above\ threhold}{Number\ of\ function\ invocations}$$

  Where:

$$Threshold = 0.3\ seconds$$

  The threshold of 300 ms or 0.3 seconds is chosen as appropriate based on [82].

- Latency label: Latency label has been defined in order for a function invocation to be associated with a particular reward function and a Q-table. The latency label can be either low, medium, or high based on the latency level. The latency label has been allocated from the K-means clustering algorithm.

By fine-tuning the hyperparameters, it is ensured that the Q-learning model learns efficiently and adapts to the dynamic scenarios. The combination of well-calibrated learning rates, a balanced exploration-exploitation, and structured environment definitions allows the algorithm to effectively optimize resource allocation while addressing the cold start latency. The notation

5.1.2 Actions (A)

By addressing the cold start problem in Serverless computing, 6 distinct actions were defined to actively adjust the allocation of resources. The actions are tailored to improve the Serverless functions by balancing the usage of resources and limiting cold start latency. These actions include the adjustment of CPU and memory usage which are important parameters that affect the speed of execution and the cost of invocation:

- **Increase CPU usage**: The action increases the distributed CPU resources for a Serverless function invocation. When assuring that more CPU resource is deployed, the function's speed of execution might be improved which can help with cold start mitigation. However, such action may lead to the accumulation of high operational costs.

- **Decrease CPU usage**: This action limits the allocation of CPU resources for a function invocation. The target of the action is to reduce costs and optimize the utilization of resources, especially in scenarios where the CPU allocation of a certain function is bigger than its actual requirement.

- **Maintain current level of CPU usage**: As the allocation of CPU is optimal, this action makes sure that no changes are made to the resource allocation. It aids in maintaining a stable system and prevents unnecessary overhead in reconfiguration.

- **Increase Memory usage**: The action is supposed to increase the memory that is being allocated for the function's invocation. By allocating higher memory, this can lead to improved performance, which reduces overhead in garbage collection, allowing for faster data processing. The action is particularly useful for memory-intensive workflows of functions.

- **Decrease Memory usage**: By reducing memory allocation, optimization of resources and a decrease in costs are ensured. However, the action must be applied with caution to prevent degradation in performance or function failure because of insufficient memory.

- **Maintain current level of Memory usage**: The action makes sure that the memory allocation is unchanged, making sure that the current level of memory is optimal for the performance. It avoids unnecessary changes, ensuring that the operating environment is consistent with its working capabilities. By the

consistent exploration of the actions, the Q-learning algorithm learns the most optimal resource configuration to reduce the cold start problem while being cost-efficient. Combining these 6 actions enables the model to adapt to various workloads in real-time, by balancing between resource usage and performance.

5.1.3 Environment States (S)

In the Q-learning algorithm for cold start mitigation in Serverless computing, the environment is represented as a limited set of states. The states are tailored to capture the current allocation of resources in terms of CPU and memory usage which are important variables influencing the latency and performance of function invocations.

9 states are defined considering the combination of the three levels of CPU usage (low, medium, high) and three memory usage levels (low, medium, high). As the Q-learning model does not work well with continuous values, the CPU and memory values were made discrete (in the form of string values). Benchmark values for low, medium, and high CPU and memory usage values were used from [45]. CPU usage is a percentage of the whole CPU resource: low: below 20%, medium: 20%-70%, high above 70%. Memory usage is a percentage of the whole memory resource: low: below 30%, medium: 30%-60%, high above 60%. In this structured way, the state space enables the Q-learning model to effectively optimize and evaluate the resource allocation. The defined states are:

- **Low CPU, Low Memory** represents a configuration where CPU and memory allocation are at their lowest levels. The state corresponds to low-rate resource consumption but might result in below the performance of the standard for resource-intensive workloads.
- **Low CPU, Medium Memory** Represents a combination of minimal CPU allocation and a moderate amount of memory. The state is suitable for memory-intensive workloads with low computational requirements.
- **Low CPU, High Memory** designs a state with minimal CPU allocation but need of high memory resources. This combination may support workloads that require significant memory while keeping low computational capacity.
- **Medium CPU, Low Memory** presents a balanced combination with moderate CPU allocation but limited memory resources. The state might support computationally intensive tasks with low memory requirements.
- **Medium CPU, Medium Memory**: The state requires a moderate allocation of both memory and CPU. The configuration is balanced between memory and computational resources, which is suitable for general-purpose workloads.
- **Medium CPU, High Memory** represents a combination of moderate CPU allocation and big memory resources. The state is suitable for workloads with a configuration of moderate computational power and maximum memory demands.
- **High CPU, Low Memory** indicates a combination of maximum CPU usage but limited memory resources. The state is suitable for high computational workflows with minimal memory requirements.

- **High CPU, Medium Memory**: The state demands high CPU allocation and moderate memory resources. The combination assists workloads that are CPU-intensive with a moderately dependent requirement.
- **High CPU, High Memory**: Indicates a state with both the maximum allocation of CPU and memory resources. This combination demands the highest performance potential but for the greatest cost making it suitable for resource-demanding workloads.

By the definition of the 9 states, the Q-learning algorithm can continuously evaluate the effect of various resource allocation combinations. This allows the model to learn the most optimal resource distribution strategies for the cold start problem while keeping cost efficiency and workload performance.

5.1.4 Adaptive Reward functions (R)

To accurately capture the dynamically changing requirements of Serverless function invocations with various demands, three different reward functions have been defined. The reward functions are designed to reflect on the trade-offs between resource usage for CPU and memory and the level of adherence to the Service Level Objective (SLO) in low, medium, and high scenarios. By giving weights to each factor, the reward functions assist the Q-learning model to adaptively determine the most optimal resource allocation. The reasoning behind the allocation of the weights is related to the different levels of latency (low, medium, high) with CPU and memory usage. For example, for low latency reward, priority is given with a higher weight to the SLO adherence, while CPU and memory usage were given lower weights because lower latency scenarios are shown to be less dependent on resource overcommitment related to CPU and memory usage which causes the cold start latency [71]. While for medium and high latency cases, higher weights are given to CPU and memory usage at the expense of SLO adherence.

Reward Function for Low Latency (r_{low})

For low latency function invocations, the reward function prioritizes SLO adherence as the latency is directly correlated with computational power (CPU usage) and memory requirements. This ensures a balanced consideration between CPU and memory usage. It takes into account that invocations can deliver quick responses without the excessive consumption of resources. The reward function is defined as:

$$r_{low} = 0.30 * CPU + 0.30 * Memory + 0.40 * P(SLO)$$

In this case, the adherence to SLO (SLO) has the highest weight (40%) while placing an emphasis on minimizing latency by balancing between CPU usage (CPU) and Memory usage (Memory).

Reward Function for Medium Latency (r_{med})

For medium latency function invocations, the reward function gives an equal weight to CPU and memory usage, by slightly reducing the weight for SLO adherence. This

employs a balanced allocation of resources but also simultaneously keeps appropriate response times. The reward function is defined as:

$$r_{med} = 0.35 * CPU + 0.35 * Memory + 0.30 * P(SLO)$$

In this case, the emphasis is on the optimization of resource usage without making compromises on performance within acceptable limits. The weights are 35% for CPU and Memory usage (CPU and Memory) and 30% for adherence to SLO (SLO).

Reward Function for High Latency (r_{high})

For high latency function invocations, the reward function puts a bigger emphasis on limiting resource usage. This step reflects relaxing performance requirements while enabling the system to prioritize cost efficiency. The reward function is defined as:

$$r_{high} = 0.40 * CPU + 0.40 * Memory + 0.20 * P(SLO)$$

In this case, CPU usage (CPU) and Memory usage (Memory) are weighted weighted at 40%, while SLO adherence (SLO) has a weight of 20%.

The three reward functions allow the Q-learning model to dynamically adapt to the changing nature of function invocations. By applying the right reward function based on the latency label, the system makes sure that resource allocation is aligned with the right performance and cost requirements for each invocation.

The weighted structure of each reward function effectively assists the learning process of the agent by balancing efficiency and performance in addressing cold start latency in Serverless computing.

5.1.5 Q-tables (T)

By addressing the scalability issue from the larger size of the Q-table, a new approach involving the creation of three different Q-tables is employed. Each Q-table has a dedicated latency category of serverless function invocations: low latency(t_{low}), medium latency (t_{med}), and high latency (t_{high}). This separation reduces the complexity of the Q-learning algorithm while tailoring the process of optimization to the different latency requirements.

Each Q-table has the 6 actions as columns and 3 of the set of states for rows which were described in Sub-sections 5.1.2 and 5.1.3 respectively. Dividing one Q-table into three smaller ones ensures that the model focuses on optimizing the resources based on the latency label of the function invocation.

The allocation of states in the Q-tables is done through a scoring system. The scoring system is as follows: each state consists of two components for CPU and memory usage. The possible states for each component can be low, medium, or high. To each component's state, a score is given: 1 for low, 2 for medium, and 3 for high. As each state has two components the sum of scores for each component in the state is calculated. This scoring system is used to map the corresponding states to the three Q-tables. The states mapping to the Q-tables happens as follows: the first three states with the lowest score are put in the low latency Q-table (t_{low}), the states with the middle three scores are put in the medium latency Q-table (t_{med}) and the three states with the highest scores are put in the high latency Q-table (t_{high}). The detailed mapping of states to Q-tables with scores calculation can be found in Table 5.1.

CPU usage/Memory usage	Low	Medium	High
Low **Score:** **Allocated table:**	Low, Low 1 + 1 = 2 Low	Low, Medium 1 + 2 = 3 Low	Low, High 1 + 3 = 4 Medium
Medium **Score:** **Allocated table:**	Medium, Low 2 + 1 = 3 Low	Medium, Medium 2 + 2 = 4 Medium	Medium, High 2 + 3 = 5 High
High **Score:** **Allocated table:**	High, Low 3 + 1 = 4 Medium	High, Medium 3 + 2 = 5 High	High, High 3 + 3 = 6 High

Table 5.1: Mapping of each state to the respective Q-table by a score

Low Latency Q-Table (t_{low})

The Q-table for low latency function invocations (t_{low}) gives priority to configurations that ensure a quick response time. The states of this table are selected on the ability to limit latency and resource configuration while taking into consideration cost efficiency. The states selected for this table are: low CPU, low Memory; Low CPU, Medium memory; and Medium CPU, Low Memory.

Medium Latency Q-Table (t_{med})

The Q-table for medium latency function invocations (t_{med}) balances the cost and performance. States of this table are selected to reflect on the combination of providing enough resources and maintaining appropriate response time without incurring extra operational overhead. The selected states for this table are: Low CPU, Medium Memory; Medium CPU, Medium Memory; and High CPU, low Memory.

High Latency Q-Table (t_{high})

The Q-table for high latency function invocations (t_{high}) is suitable for the optimization of longer response time scenarios that are acceptable. The states selected for this table are: Medium CPU, High memory; High CPU, Medium Memory; and High CPU, High Memory.

The division of Q-tables lets the Q-learning model optimize adaptively the allocation of resources with different latency requirements, improving the efficiency and scalability of the algorithm while addressing the cold start latency in Serverless computing.

The Q-values in each Q-table are updated using Bellman's equation to reflect on the reward received for choosing an action (a) in a particular state (s).

5.2 K-means Algorithm

The k-means clustering algorithm is part of unsupervised machine learning that makes partitions from a dataset k separate clusters. Its target is to group data paints based on their similarity and limit the variance within each cluster. In the context of the method described, the K-means is employed for latency label assignment (low, medium, high) based on the latency level of each function invocation.

Steps in K-means algorithm:

- Initialization: Selects k initial random centroids from the data points which in the context of the method are the latency values of historic function invocations.
- Assignment step: Makes the assignment for each latency value to a cluster with the nearest centroid. Proximity is a measure of Euclidean distance:

$$d(x, c) = \sqrt{\sum_{i=1} n(x_i - c_i)^2}$$

  Where x is the latency value, and c is the centroid.

- Update step: Computing new centroids by taking the mean of all latency values that are assigned to each cluster:

$$c_k = \frac{1}{N_k} \sum_{i \in C_k} x_i$$

  Where c_k is the centroid of the new cluster k, N_k is the number of latency values in cluster k, and C_k is the set of points in the cluster k

- Iteration: Assignment and update steps are repeated until convergence of the centroids. In other words, the assignment of clusters can no longer change or the algorithm has reached a predefined number of iterations.

Latency labels application

In the context of the method, the K-means algorithm is used for the categorization of latency levels of function invocations into 3 clusters:

- Low latency
- Medium latency
- High latency

The three clusters correspond to a latency label and are related to a specific reward function and Q-table. The classification allows the Q-learning model to adapt its actions to various requirements of function invocations, by optimization of resource allocation and performance.

Algorithm 1 presents the pseudocode for the Mitigating Cols start issue using Q-learning and K-means clustering. The method takes as input historical data in the form of function traces, the number of clusters, the number of training episodes, the number of iterations, and hyperparameters for Q-learning. It outputs an optimal scheduling policy in the form of Q-tables. The method consists of 4 parts. The first part clusters the traces into three clusters based on their latency for 50 iterations. The second part initiates the Q-tables needed for the Q-learning model training. Training the Q-learning model is the third part where 10,000 episodes per 50 iterations, state is initialized based on CPU and memory usage. Then, an

action is selected based on an epsilon greedy policy that balances exploration and exploitation. The action is then executed by monitoring the reward value and the next state. Then, a Q-value is updated using the Bellman equation and the next state is assigned to the current state. The optimal policy is output as the fourth and final step of the algorithm. The optimal policy is presented as a set of Q-values in Q-tables. The Q-values shows which how suitable an action is in a specific state. In other words, the higher the Q-value for a pair of actions and states, the more suitable the action is in the specific state. Thus the optimal policy is a set of an environment state and the most suitable action to be taken in this state.

Algorithm 1 Mitigating Cold Start issue using Q-Learning and K-Means

1: **Input:** Historical function traces D, Number of clusters k, Training episodes n_{train}, $n_{\text{iterations}}$, Q-learning parameters $(\alpha, \gamma, \epsilon, \epsilon_{\text{decay}})$
2: **Output:** Optimal scheduling policy π^*
3: **1. Clustering function traces**
4: **for** *iteration* $= 1$ to $n_{\text{iterations}}$ **do**
5:     **while** $\neg$converged **do**
6:         Assign each data point in D to the nearest cluster
7:         Calculate the centroid of each cluster
8:         **if** cluster centroids not changed **then**
9:             converged $\leftarrow$ True
10:         **end if**
11:     **end while**
12:     Assign each trace to $k = 3$ clusters: $\{\text{low}, \text{medium}, \text{high}\}$
13:     Increment $n_{iteration} \leftarrow n_{iteration} + 1$
14: **end for**
15: **2. Initiate Q-tables**
16: **for** each latency label cluster $c \in \{\text{low}, \text{medium}, \text{high}\}$ **do**
17:     Initiate Q-table $t(s, a)$ with zeros
18: **end for**
19: **3. Q-Learning Training**
20: **for** $(n_{\text{iterations}} = 1)$ to 50 **do**
21:     **for** $n_{train} = 1$ to $10,000$ **do**
22:         Initialize state s taking into account (CPU and Memory usage)
23:         **while** s not terminate **do**
24:             Select action a using ϵ-greedy policy

$$a = \begin{cases} \text{random action with } \epsilon \\ \arg\max_{a'} Q_c(s, a'), \ \text{otherwise} \end{cases}$$

25:             Execute action a, monitor reward r and next states'
26:             Update Q-value:

$$Q(s, a) \leftarrow Q_c(s, a) + \alpha \left(r + \gamma \max_{a'} Q_c(s', a') - Q_c(s, a) \right)$$

27:             Update state $s \leftarrow s'$
28:         **end while**
29:         Increment $n_{train} \leftarrow n_{train} + 1$
30:     **end for**
31: **end for**
32: **4. Output Optimal Policy**
33: Return $\pi^*(s) = \arg\max_a Q_c(s, a)$ for all states s

Chapter 6

Implementation

The suggested two-tier method for defining the most optimal scheduling policy for cold start latency reduction combines Q-learning and K-means. To demonstrate how effective the method is, a prototype has been devised. The prototype is a rigorous experiment that observes the method's behavior and its performance. The implementation utilizes Docker[1], PostgreSQL[2], OpenFaas[3], Python[4], Flask API[5], Gym[6], and different images from Docker Hub[7] and GitHub[8]. In this chapter, the details of the implementation of the method are given.

Figure 6.1 shows the implementation of the method with the underlying technologies. The whole method is deployed in OpenFaaS as a function, consisting of Docker containers. The user is communicating with the method through Fask API, developed in Python, which gets function invocation traces as input data. Then, the data is stored in Postgres SQL database. The data is accessed by the Q-learning model developed with Python using Open AI Gym library and custom-implemented K-means algorithm using Python. The K-means algorithm clustered the function invocation traces and fed them to the Q-learning model. After a few iterations, the model outputs a policy and metrics for evaluation which are then passed to the user via the API as output data.

The process of implementation follows a sequential workflow:

6.1 Data source and preprocessing

The data used for experimentally testing the prototype of the method is historic Serverless function invocation traces from Huawei Cloud[9], from which two tables were used:

- Huawei Public Cold Starts Table: provides data about the cold start latency of each function invocation as well as latency behaviors and patterns associated with cold start [34].
- Huawei Public Request Table - contains historical data on the requests from function invocations. It provides data about CPU and memory usage and end-to-end latency associated with a particular function invocation [34].

[1] `https://www.docker.com/`
[2] `https://www.postgresql.org/`
[3] `https://www.openfaas.com/`
[4] `https://www.python.org/`
[5] `https://flask.palletsprojects.com/`
[6] `https://www.gymlibrary.dev/`
[7] `https://hub.docker.com/`
[8] `https://github.com/`
[9] `https://github.com/sir-lab/data-release/blob/main/README_data_release_2025.md`

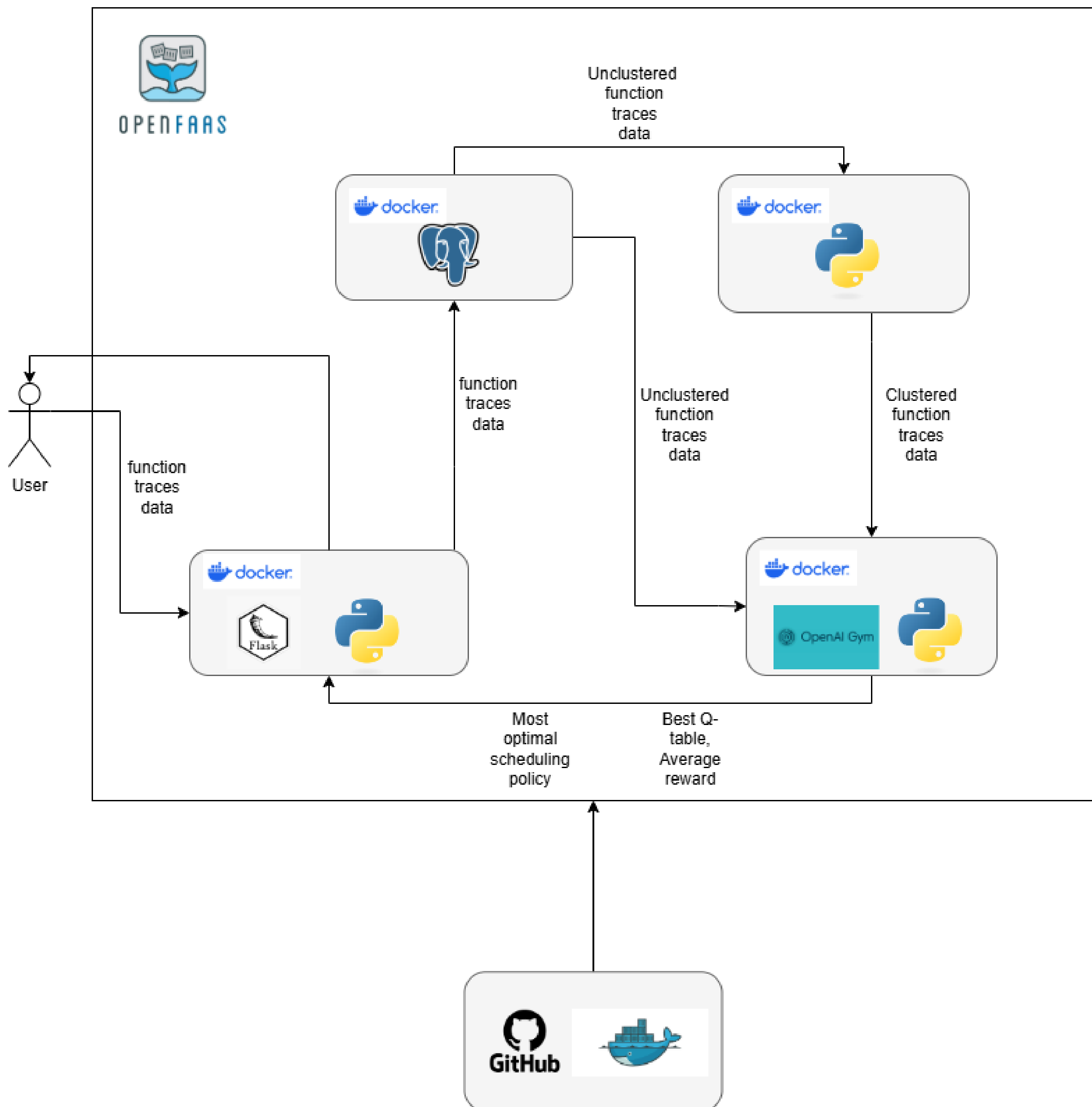

Figure 6.1: Overview of the Implementation of the Method as a Function

The data in the tables was used in the definition of the states, actions, and transitions for the Q-learning model as well as initializing the clustering of the function invocations with K-means clustering algorithm. The tables were stored in PostgreSQL database where data prepossing stage was performed associated with CPU and memory usage normalization and calculation of probability of adherence to SLO ($P(SLO)$).

6.2 Q-leaning algorithm setup

- **State space**: 9 states in total have been defined as a result of combinations of CPU and Memory usage levels. The levels are low, medium, and high.

- **Action space**: 6 actions in total have been defined to change CPU and Mem-

ory usage in real time. These actions are to increase, decrease, and maintain the current level.

- **Reward functions**: 3 separate reward functions have been designed for the corresponding latency levels of function invocations for low, medium, and high latency, by balancing between CPU and Memory usage with the Probability of adherence to service level objective (SLO).

6.2.1 Simulation Environment

A custom simulation environment was designed using the OpenAI Gym library in Python to accommodate the dynamic changes of Serverless computing. The environment is represented by the set of states, actions, and reward functions. It tracks dynamically the current state, actions being executed and the calculated reward based on the reward functions defined. Moreover, after each episode, a metric of the average reward is being calculated.

6.2.2 Q-learning algorithm training

Over 1 iteration for training, the Q-learning model was trained for 10,000 episodes. In total 50 training iterations were done in order to ensure convergence. An ϵ-greedy policy was defined to balance the trade-off between exploitation and exploration, with ϵ decrementing from 1.0 to a minimal value over the episodes. The Bellman equation was used for updating the Q-tables iteratively making sure convergence to the most optimal policy.

6.2.3 Q-learning algorithm evaluation

The Q-tables generated during the training phase of the Q-learning model are evaluated over 100 episodes. The evaluation is repeated over 50 iterations to ensure the most accurate evaluation of the most optimal policy proposed during the training phase. The dataset from the Huawei cloud is used for evaluation as well.

6.3 K-means clustering

End-to-end latency is chosen from the Huawei cloud dataset as the most appropriate measure of total latency. Three separate clusters were established for: low, medium, and high latency. A latency label corresponding to the cluster is assigned to each function invocation and the updated data is pushed to the table in the Postgre SQL database.

6.4 Tools and Frameworks

A number of tools and frameworks have been used to create the prototype of the proposed method:

- **Docker**: Used for containerization of the method.

- **OpenFaaS**: allows for the deployment and orchestration of the method as containerized with Docker.
- **Postgre SQL**: Stores the data from the Huawei cloud dataset and data pre-processing is done with SQL.
- **Python**: is the main programming language for writing the prototype, it is the backbone for providing a custom implementation of K-means, Q-learning, and data processing with Pandas [10] and Numpy [11].
- **Flask API**: serves as a proxy between the method consisting of the model, K-means algorithm, and the database and the Serverless framework (OpenFaaS). It assists in communication between both sides.
- **Open AI Gym**: Creates a custom model of the dynamic Serverless environment. It is responsible for tracking states and rewards.
- **Docker Hub and GitHub**: Assists for supplying rebuild images and source code for deployment of the method in OpenFaaS using Docker [12].

6.5 Performance monitoring

It is essential to evaluate the performance of the method both how well it fulfills the tasks given but also how much resources are needed for the method to fulfill the assignment. That's why, the importance of monitoring the performance of the method throughout training is implemented incrementally over the iterations of the training process. To ensure that the method was evaluated for different characteristics, two sets of metrics have been defined.

The first set takes into account the accuracy of the method or in other words how well it performs the task given. Metrics such as the average reward per iteration and average sum of Q-tables have been monitored. The average reward value is a key metric to showcase how well the method does its task. Its value is between 0 and 1 where 0 is the lowest and 1 is the highest value, the method can get. The higher the average reward the better the method does its job. The Q-values in the Q-tables show which action is more suitable to be taken in a certain state of the environment which forms pairs of actions and states. In this way, the method gives its most optimal policy for getting the task done and this is represented by the sum of the Q-values in the Q-tables per each iteration.

The second set of metrics takes into account the performance of the method in terms of the resources it demands. This is also important because if more resources are required, this is associated with additional costs for the user. Three metrics are considered here similar to the data from the historical traces. Latency per iteration is defined as the end-to-end latency from firing a request to receiving an output. This can be interpreted as the start time of an iteration until receiving the average reward value for the same iteration. Then CPU usage (consumption) per iteration which is again determined from the start time of an iteration to receiving the average reward

[10] `https://pandas.pydata.org/`
[11] `https://numpy.org/`
[12] `https://github.com/landaudiogo/cc-tutorials-2023`

value per iteration. The third metric is memory usage (consumption per iteration), which is determined in a similar manner to CPU consumption.

Chapter 7

Experimental design

The purpose of the experimental design is to evaluate how effective the proposed method that combines a Q-learning model with a K-means algorithm for clustering in order to combat the cold start latency in Serverless computing. This chapter of the report gives detailed information about the experimental setup, metrics, methodology, and tools that have been used to evaluate the performance of the proposed method.

7.1 Goal

The primary goal of the experiment is:

- Measuring the extent of reduction of the cold start latency achieved by the suggested approach.
- Evaluating the efficiency of resources associated with CPU and memory usage.
- Assessing the adaptability and scalability of the method under various workload conditions.

7.2 Experimenatal Setup

7.2.1 Hardware

For simplicity, the method was developed locally on a personal computer with the following parameters and tools:

- CPU: Intel Core 8th Gen i5
- Memory: 8 GB RAM
- Storage: 128 GB
- Environment for development: Visual Studio Code[1]

7.2.2 Software

- **Docker** used for containerization of the method, for deployment purposes.
- **OpenFaaS** is a Serveless framework that manages the orchestration and scalability of the method.

[1] `https://code.visualstudio.com/`

Field name	Description	Unit
day	day number	date
time	timestamp	seconds
clusterName	cluster being processed	int
funcName	unique identifier for a function	int
userID	unique identificator	int
requestID	unique request identifier	string
totalCost_cold_start	time spent on cold start	seconds
podAllocationCost	time taken to start a pod	seconds
deployCodeCost	end-to-end time of deployment	seconds
deployDependencyCost	time to fetch and load dependencies	seconds
schedulingCost	time for scheduling overheads	seconds
podID	pool name	string

Table 7.1: Schema of the table Huawei Public cold starts

- **PostgreSQL** is a relational database that stores historic data of Serverless traces from the Huawei cloud in tables. It was used for data preprocessing and manipulation.

- **Python**: The method was mainly developed on Python, implementing both a Q-learning model and a custom definition of the K-means algorithm. Libraries that were used for implementation are NumPy, Pandas, and Sqlalchemy.

- **Flask API** serves as an intermediary between the requests and the method (the database, the Q-learning model, and the K-means algorithm)

- **OpenAI Gym**: Used for modeling the environment for the Q-learning model and later for training an evaluation.

7.3 Source of Data

7.3.1 Huawei Public Cloud Trace 2025

The whole data set consists of 4 tables with data from 5 regions for 30 days or 1 month. The tables are: Huawei Public cold starts, containing the data about cold starts; Huawei Public request tables containing the event-level logs of individual requests; Huawei Public trigger types and runtime languages containing the runtime languages, trigger types, and CPU request for each function; Huawei Public time series containing the time series of various metrics on the platform per function, including quantiles of cold start time.

For the experimental design the first two tables were selected as suitable for obtaining data about function invocations. These are namely Huawei Public cold starts and Huawei Public request tables. The schemas of both tables are presented in tables 7.1 and 7.2

Field name	Description	Unit
time_worker	timestamp of the worker	seconds
time_frontend	timestamp of the frontend	seconds
requestID	unique request identifier string	string
clusterName	cluster being processed	int
funcName	unique identifier for a function	int
podID	pool name	string
userID	unique identificator	int
totalCost_worker	total time spent in function worker pod	seconds
workerCost	time spent in worker	seconds
runtimeCost	time spent on runtime time	seconds
totalCost_frontend	end-to-end time	seconds
frontendCost	time spent on front-end	seconds
busCost	time spent in bus	seconds
readBodyCost	time spent on reading request body	seconds
writeRspCost	time spent on writing response	seconds
cpu_usage	CPU usage	cores
memory_usage	memory usage	MB
requestBodySize	size of request body	Bytes

Table 7.2: Schema of Huawei Public request table

Field name	Table taken	Name from table	Description	Unit
total_latency	requests	totalCost frontend	End-to-end latency	seconds
cpu_usage	requests	cpu_usage	Normalized CPU usage	decimal
memory_usage	requests	memory_usage	Normalized Memory usage	decimal
id	-	-	Unique identifier	int
number_of_cold_starts	-	-	Total number of cold starts per function	int
number_abover_threhold	-	-	Total number of function requests above 30 ms	int
slo_violation_rate	-	-	Rate of SLO being violated	decimal
latency_label	-	-	Label for a latency level	string

Table 7.3: Schema of custom table

7.3.2 Custom tables

10 identical tables from columns from both tables described above have been made in order to fully adhere to the data requirements of the method. Each table contains 50,000 records or 500,000 records in total. The method requires data about the CPU, Memory usage, probability of adherence to SLO, and a latency label for every function invocation. The schema of the custom tables is presented in table 7.3

7.4 Methodology

7.4.1 Data preprocessing

The relevant column for the data used for the method has been extracted from the Huawei cloud traces dataset. The relevant data is associated with total latency, CPU, and Memory usage. Other columns were customly made to calculate the probability of SLO adherence of each function which is a measure between 0 and 1. Moreover, a latency label is created to classify the latency associated with each

function invocation, this is done through the K-means algorithm due to the various nature of each function being invoked. The CPU and memory usage were also normalized to be between 0 and 1 such that enabling the reward function to have a normalized value.

7.4.2 Simulation of Environment

By using the OpenAI Gym library, a custom simulation of the environment was developed. The environment consists of 9 predefined states that are based on CPU and memory levels and 6 actions that Q-learning can perform in a certain state so that a reward function is given to the agent. The reward function reflects the main goal of the method for reducing cold start latency by taking into account the CPU, Memory usage, and probability of SLO adherence.

7.4.3 Training phase

The Q-learning model was trained for 10,000 episodes for 50 iterations. Every episode consists of the agent interacting with the environment by exploring actions using the epsilon-greedy policy and updating the Q-values in the Q-tables based on the observed rewards. Three separate Q-tables were established and maintained for the low, medium, and high latency levels.

7.4.4 Evaluation phase

After the training phase was completed, the method was evaluated for 500 episodes for 50 iterations, using dummy function invocations, similarly to [77]. These function invocations were deployed in OpenFaas and during every invocation, their behavior was monitored and data for their latency, CPU, and memory usage was collected. The data was generated on an Ubuntu AWS EC2 virtual machine. For evaluation purposes, two scenarios for generating the data were determined. The first scenario invoked the functions **one-by-one** over 5 runs of 100 invocations (1000 invocations per run), 5000 in total. While the second scenario the functions were invoked **in parallel** for 5 runs of again 5 runs of 100 invocations (1000 invocations per run), 5000 in total. Table 7.4 shows the complete list of the deployed functions along with their description, input and output values, and for which workload they are most suitable. Later, the data were preprocessed and additional columns determining SLO adherence and latency label were added to the dataset. Table 7.5 shows the schema of evaluation datasets along explanation of each field name and their unit.

7.4.5 Metrics

Various metrics have been considered to evaluate the robustness of the proposed method:

- Average reward per iteration: At the end of an iteration an average reward value is calculated from the 10,000 episodes. This is to showcase how well the agent is performing when selecting the appropriate action in a certain state.
- Sum of Q-tables: At the end of each iteration, the sum of each Q-table is calculated and stored for evaluating the accuracy of the proposed prototype.

Function Name	Description	Input	Output	Workload
Hello World	Outputs "Hello, World"	-	string value	Min CPU and memory usage
Fibonacci Calculator	Calculates a Fibonacci number using recursion	number	number	CPU intensive for high numbers
Image Resizer	Makes the size of an image smaller	image	image	Moderate CPU and Memory usage
JSON Parser	Parses JSON file and retrieves data	JSON file	Data	Memory intensive for larger files
Matrix Multiplication	Multiples two matrices given size	number	string	CPU intensive
String Reverser	Reverse a string	string	string	Min CPU and memory usage
Sort a List	Sort a list of numbers of random numbers of a size	number	list	CPU intensive
Generate Random UUIDs	Generate a list with random Universal Unique Identifiers (UUIDs)	number	list	Moderate CPU usage
HTTP Fetcher	Makes a HTTP GET request to URL	string	string	Moderate CPU usage
Prime Number Checker	Checks if a number is prime	number	string	CPU intensive for large numbers

Table 7.4: Dummy functions used for generating data for Evaluation

Field name	Description	Unit
id	Sequence number appearing in the dataset	int
function_name	Name of the function	string
invocation	Sequence number of invocation in the run for a function	int
run	Iteration in which the data was generated	int
latency	end-to-end-latency	seconds
cpu_usage	Normalized CPU usage during invocation	decimal
memory_usage	Normalized memory usage during invocation	decimal
number_of_cold_starts	Number of cold starts per function	int
number_above_threshold	Number of invocations not adhering to SLO	into
slo_violation_rate	The rate of invocations not adhering to SLO	decimal
latency_label	The latency label for low, medium, high latency	string

Table 7.5: Schema of Evaluation Datasets

- Latency: shows the end-to-end latency in seconds starting with firing a request until the prototype returns the required output

- CPU usage: the CPU consumption in % from firing a request until the prototype returns the required output

- Memory usage: the Memory consumption in Megabytes (MB) from firing a request until the prototype returns the required output

7.4.6 Experimental scenarios

10 custom tables have been initiated with 50,000 records each which makes 500,000 records in total. This makes sure that the method is tested in different scenarios to see how adaptive it is to changing situations. For each iteration of the method, 10,000 records are accessed from one of the tables. Based on the sequence of iterations, which table and exact records should be accessed is determined. This process ensures that data is always different when fed into the method.

7.4.7 Tools for Monitoring and Analysis

Python with its libraries such as Matplotlib, Pandas, Numpy was used for data analytics, storing data and generating insightful visualizations and plots of experimental data that were generated and analyzed throughout the experiment. Additionally, PostgreSQL queries assisted in data retrieval. Moreover, libraries such as psutil and time were used for monitoring latency, CPU, and memory usage.

Chapter 8

Results

This master's project suggests a two-component method combining Q-learning and K-means clustering algorithms. The method aims at finding the most optimal policy for reducing cold start latency which optimizes the scheduling of invocation of functions by clustering latency levels with K-means and modifying in-real time resource allocation via a Q-learning model. To validate how effective this approach is, a prototype has been developed using libraries in Python and PostgreSQL, functions were deployed in OpenFaas and data about their performance was collected, which was used for evaluation of the prototype. The model was rigorously tested with different metrics for both effectiveness and performance. Additionally, the data generated by function invocation in OpenFaaS is presented in order to serve as a baseline for comparison with the results collected when evaluating the prototype.

8.1 Evaluation datasets

As already mentioned in Section 7.4.4, two datasets were generated from the invocation of 10 dummy functions in two scenarios. The first scenario the functions are invocated one by one for 500 invocations each (5000 invocations in total) over 5 runs per 100 invocations each. The second scenario invocates the functions in parallel again for 500 invocations each (5000 in total) over 5 runs per 100 invocations each. The behavior of each invocations has been monitored and data about their performance was recorded. The data include metrics such as average latency per invocation, average CPU usage per invocation, average memory usage per iteration, and failure rate (the number of invocations not adhering to SLO (latency that is bigger than 30 ms.)). The datasets generated were analyzed using Python and graphs showing the results of the metrics mentioned above were created. The data presented about two datasets serves as a benchmark for comparison with the performance and effectiveness of the prototype evaluated with similar metrics.

8.1.1 Latency per invocation

Functions being sequentially invocated

Figure 8.1 presents the average latency per invocation of the 10 functions while being invocated one by one. Looking across the diagrams, it is observed that values for most functions fluctuate between 0.0125 ms and 0.05 ms for 100 invocations. This is indicative that the demands of computation power of the functions are similar, with no significant outliers for latency.

As can be observed, functions such as image-resizer showcase higher latency values demanding a higher computational power compared to functions such as JSON parsing. Functions such as the Fibonacci calculator, UUID generator, prime

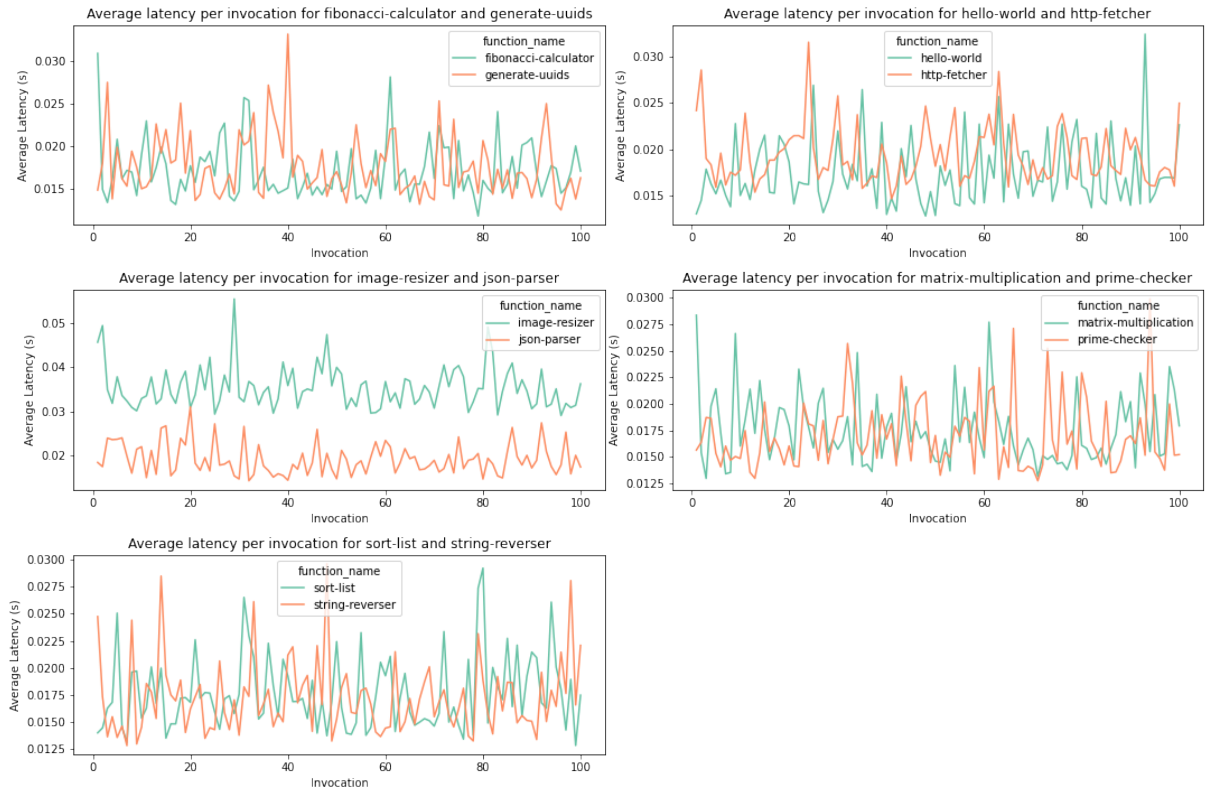

Figure 8.1: Average latency per sequential function invocation

checking, and matrix multiplication show similar latency values, showing similar performance characteristics.

Parallel function invocation

Figure 8.2 illustrates the average latency per invocation of 10 functions while being invocated in parallel. Looking at the line charts, it can be observed that there is a fluctuation in the latency of most functions ranging from 0.1 ms to 0.9 ms for 100 invocations. Functions such as fibonacci-calculator and generate-uuids have lower computational demands, while functions like image-resizers and matrix multiplication require greater computational power. An interesting event which could be considered as outlier is the spike of hello-world between the 20th and 25th invocation.

The scenario of invocation functions in parallel (Figure 8.2) seems to demand more computation power, generating larger average latency values than being invocation one by one (8.1), which seems logical as multiple functions are invoked at the same time, which also might create outliers as observed.

8.1.2 CPU usage per invocation

Functions being sequentially invocated

Figure 8.3 depicts the average CPU usage per invocation for the functions for functions invocated one by one. The graphs indicate a fluctuation in CPU consumption across invocations which proposes a variability in computational demands. Firstly,

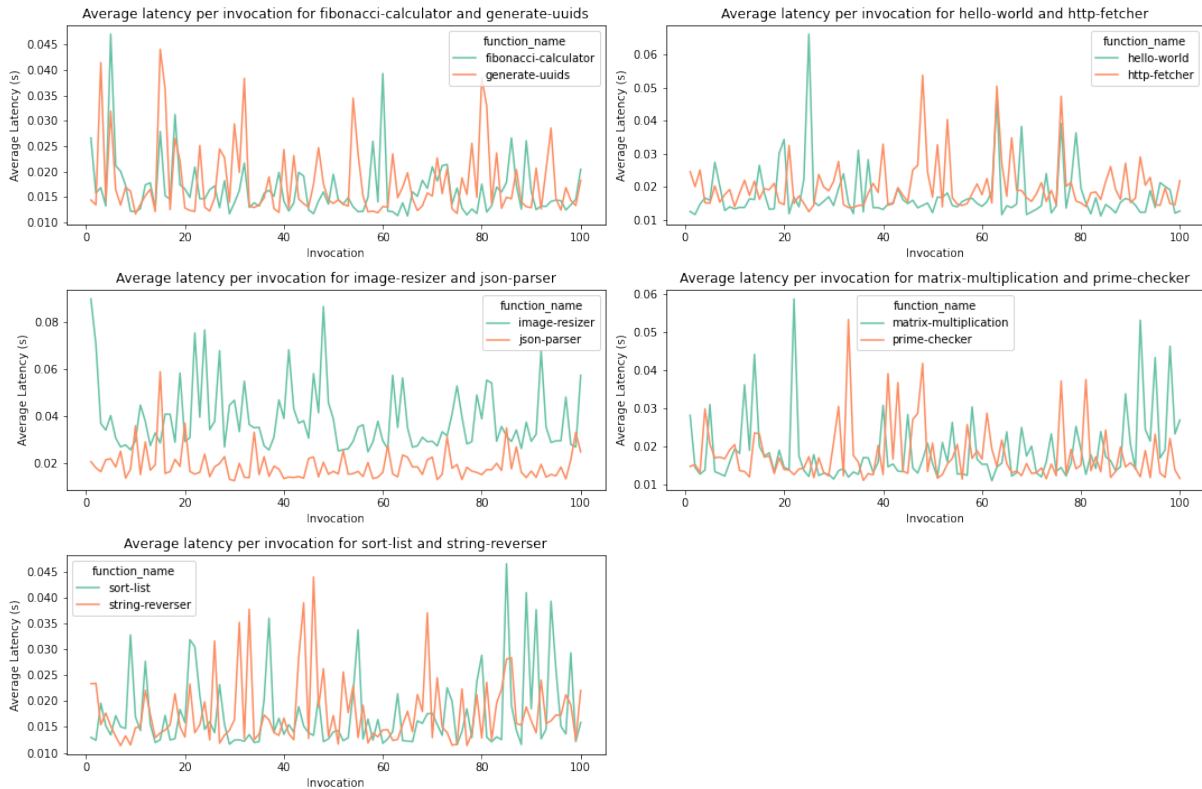

Figure 8.2: Average latency per invocation parallel function invocation

functions such as generate-uuids and fibbonacci-calcularor show significant variation which seems that during some invocations more processing power is needed. Secondly, functions such as matrix multiplication, requiring computational power, showcase a considerably higher CPU consumption compared to lightweight operations performed by hello-world function. The spikes that are observed in CPU consumption might be due to inefficiencies in the allocation of resources.

Parallel function invocation

The average CPU usage of functions per invocation, invocated in parallel is illustrated in Figure 8.4. Values for CPU consumption vary from 0.1 to 0.8%. The line diagrams showcase a variability in CPU usage per invocation in most functions. Functions such as image-resizer and json-parser have the most CPU consumption over the iterations. It seems that by increasing the workload of functions invocated at the same time, more CPU usage is required for specific invocations. This trend is observed by the spikes in most functions, which emphasizes the inefficiency of the consumption of CPU power.

In both cases, a fluctuation in the average CPU usage for each function invocation is observed, which emphasizes the dynamic nature of each function per invocation. However, bigger spikes and demands for greater CPU usage are observed in functions invocated in parallel (Figure 8.4)

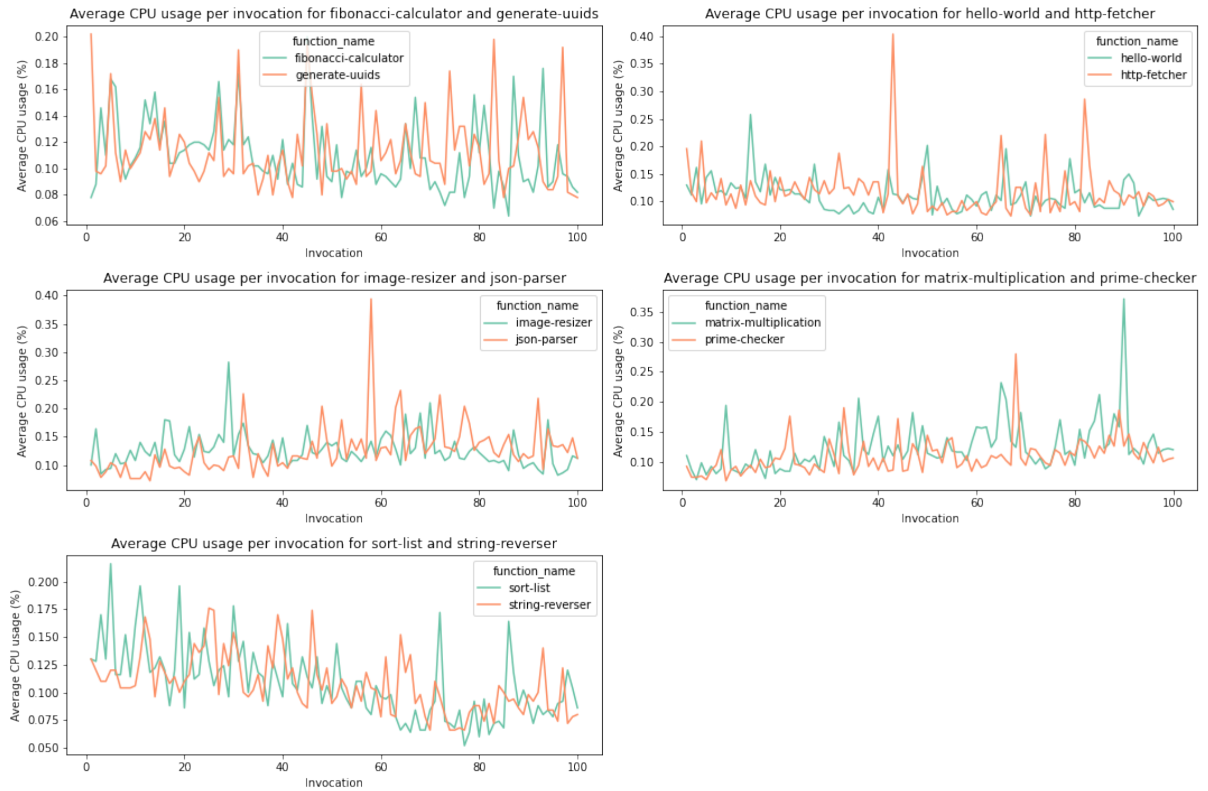

Figure 8.3: Average CPU usage per sequential function invocation

8.1.3 Memory usage per invocation

Functions being sequentially invocated

Figure 8.5 shows the average memory usage per invocation for functions invocated one by one. It illustrates a trend of increase in memory consumption over invocations. Functions such as json-parser and image-resizer showed a steady growth in memory usage, which might be due to inefficient memory management. Matrix-multiplication keeps a higher memory consumption which is consistent with its computational complexity. Other functions such as hello-world display limited memory usage which is expected for a simple workload.

Parallel function invocation

Figure 8.6 illustrates the average memory usage per invocation for functions invocated in parallel. It can be observed that values vary from 25 MB to 34 MB, as hello-world has the lowest average memory consumption over iterations while metrix-multiplication has the highest. A similar trend is observed here as well with most functions, the average memory usage keeps rising with each iteration.

The trend of a steady growth is observed in both cases. However, during parallel invocations, functions demand a higher memory resource (Figure 8.6) than when invocated one by one (Figure 8.5), which seems logical.

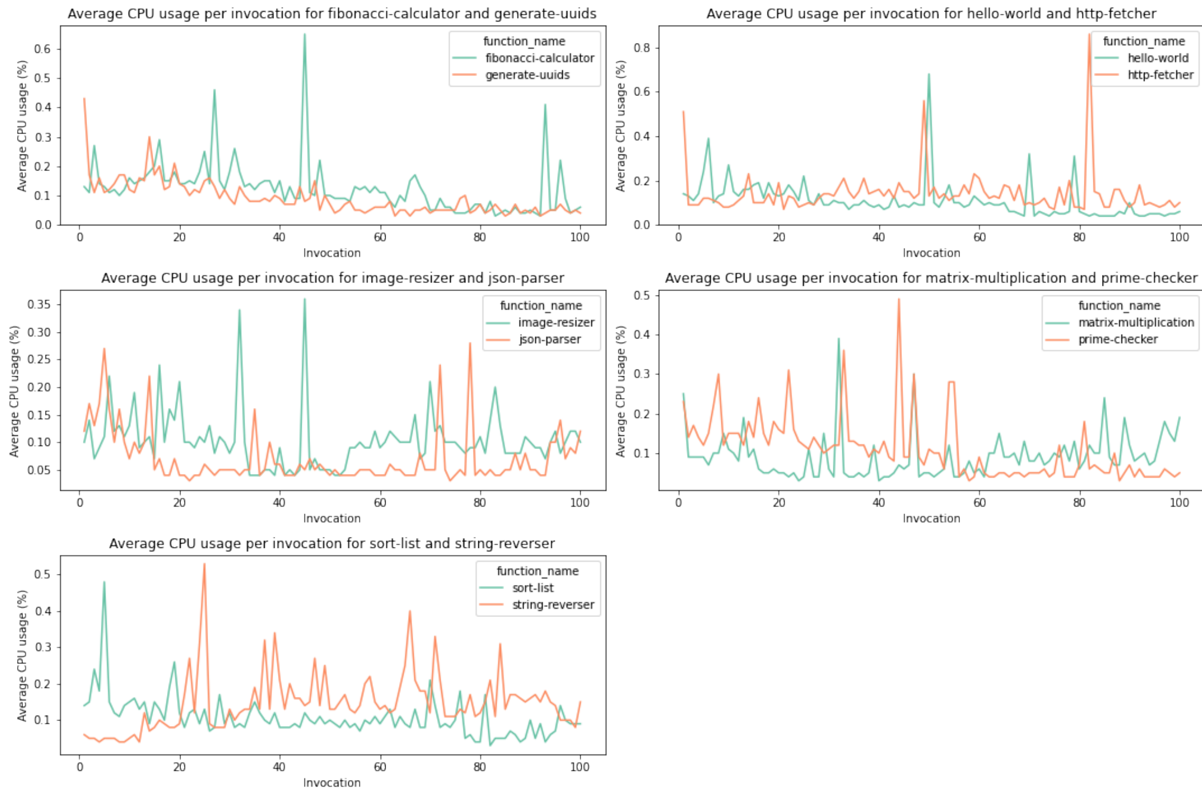

Figure 8.4: Average CPU usage per invocation parallel function invocation

8.1.4 Average Failure rate to SLO adherence rate per invocation

Functions being sequentially invocated

Figure 8.7 shows the average failure rate of each function per invocation for functions invocated one by one. The failure rate is the proportion of invocations adhering to SLO. Functions invocations of image-resizer and json-parser show considerable fluctuations in failure rate, some invocations reaching an average rate of nearly 1.0 which indicates complete failure. Computationally intensive functions like matrix multiplication show irregular failures. This variability in failure rate requires to be investigated further due to potential inefficiencies in resource allocation and scheduling.

Parallel function invocation

The average failure rate with parallel function invocation is shown in Figure 8.8. As mentioned before, it is a proportion of adherence to SLO per invocations. The rate fluctuates for most functions over invocations. However, the failure rate is the highest and 1 in most cases for matrix-multiplication function which is directly connected to its demand for high computational power and memory consumption.

It is observed that the functions' failure reaches a maximum value in both cases over invocations. It is more significant in functions invoked in parallel (Figure 8.8) than in one-by-one invocations 8.7. It can be seen that the function matrix-multiplication has the highest rate in both scenarios which leads to its demands for

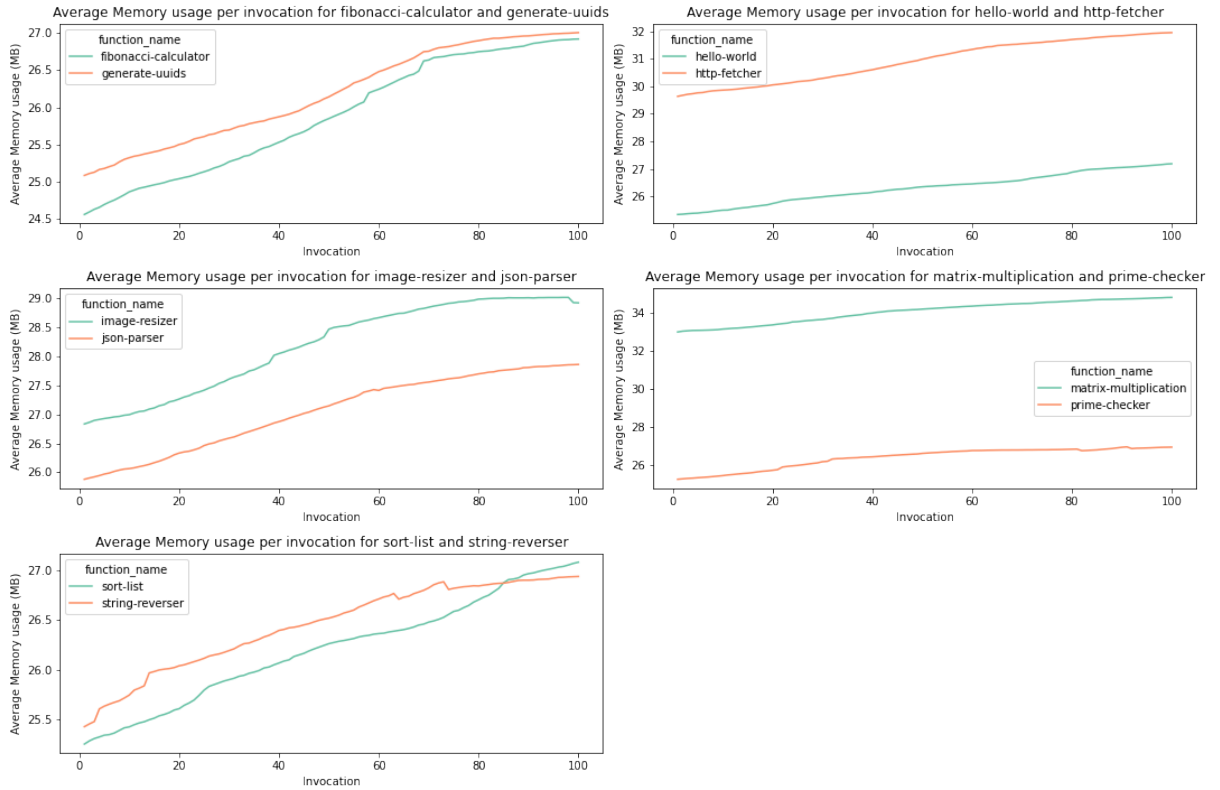

Figure 8.5: Average Memory usage per sequential function invocation

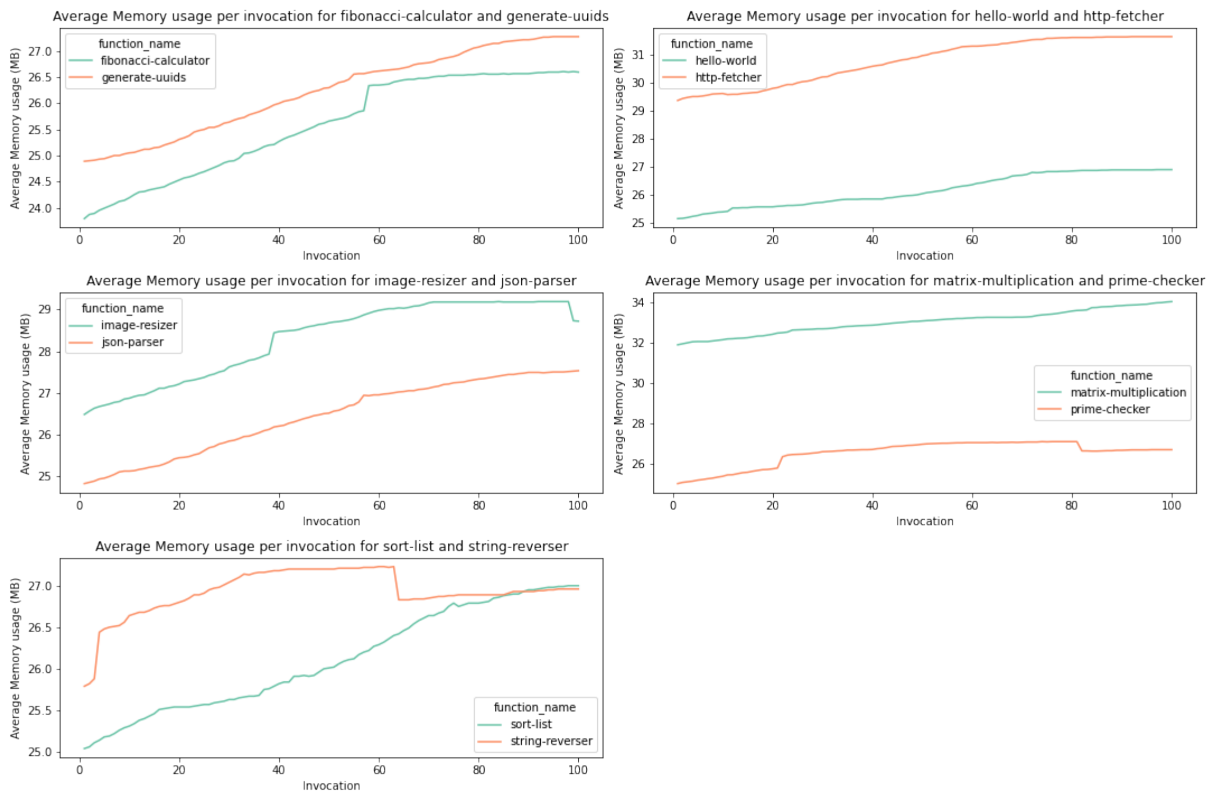

Figure 8.6: Average Memory usage per parallel function invocation

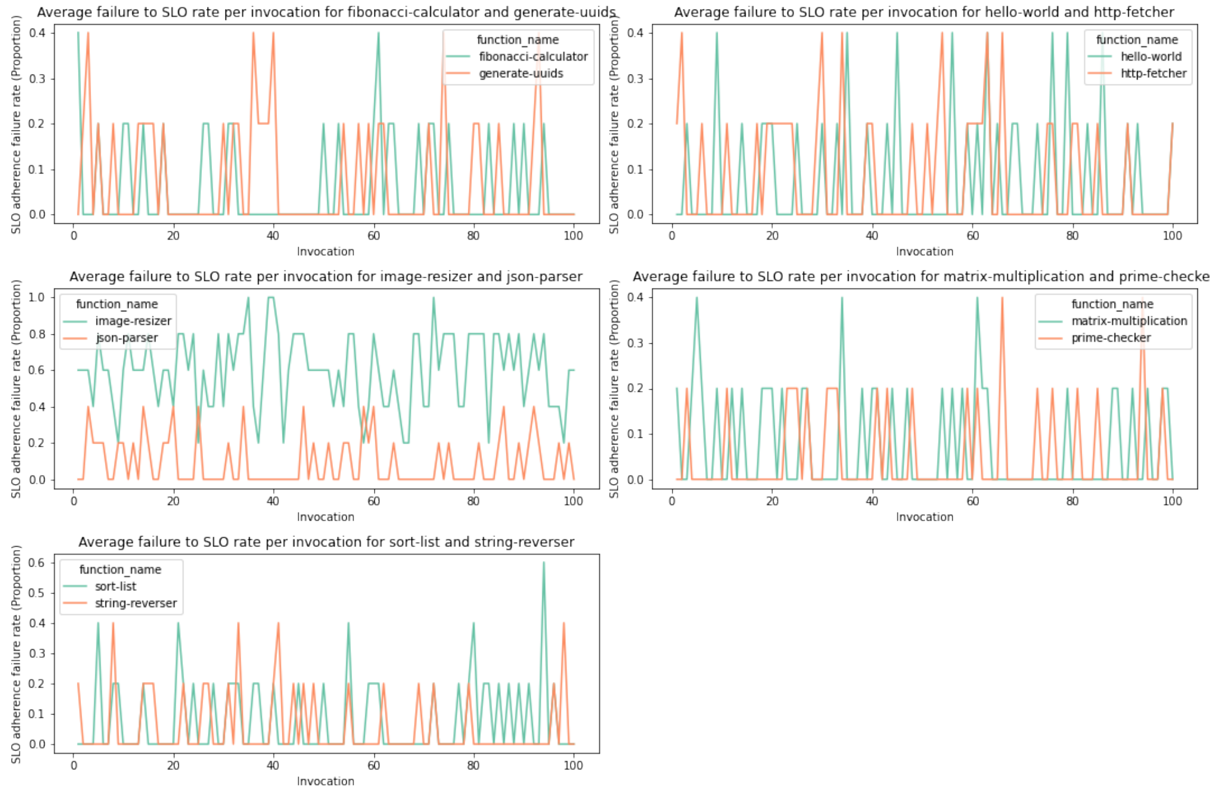

Figure 8.7: Failure rate to SLO adherence per sequential function invocation

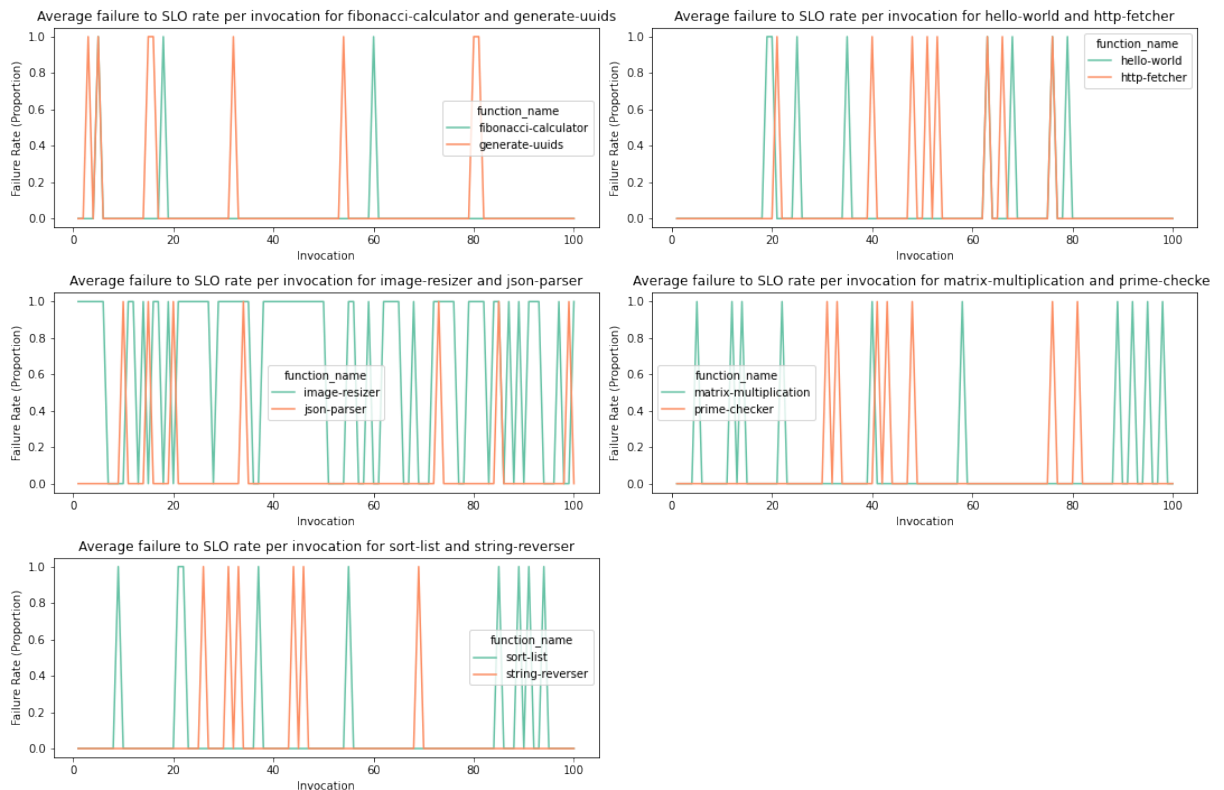

Figure 8.8: Failure rate to SLO adherence per invocation parallel function invocation

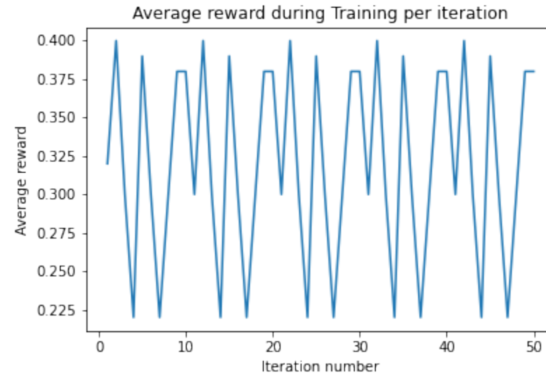

Figure 8.9: Average reward during Training per iteration

more resources.

8.2 Accuracy of Method

In this section, the method's accuracy is evaluated. It is essential for the method to be evaluated on how well it performs with the task of reducing the cold start problem. For this reason metrics such as average reward and sum of Q-tables were defined. The average reward metric per iteration shows the incremental change in the reward as the model is learning through trial and error by taking an action in a specific state. The average reward is being monitored for training and also for evaluation with the two datasets of function invocation. The sum of Q-tables during training measures the extent to which the agent is taking the right action in the right state. The higher the Q-table sum the better the agent can determine which pair of action and state goes together. The sum of Q-tables per iteration during training is monitored for the three Q-tables for low, medium, and high latency labeled function invocations.

8.2.1 Average reward during training

Figure 8.9 illustrates the average reward during training per iteration of the Q-learning model. The average reward is between 0.22 and 0.40 for 50 iterations. As it can be observed, the average read starts at around 0.32 in the beginning but gradually increases to 0.30 by iteration 25. The highest average reward 0.4 is reached at around iteration 45 and at the end the reward stabilizes to 0.38.

Overall, the Q-learning model is improving over time, with the reward increasing and then stabilizing. This is indicative of the model's learning process and performance optimization through the process of training.

8.2.2 Sum of Q-values per iteration during Training

Figure 8.10 shows the sum of Q-values changes for 50 iteration during training for different latency labels (low, medium, high). For low latency function invocations (Blue line), the sum shows fluctutuation between 100 and 150, with peaks and troughs periodically. The sum is generally higher compared to the values from the

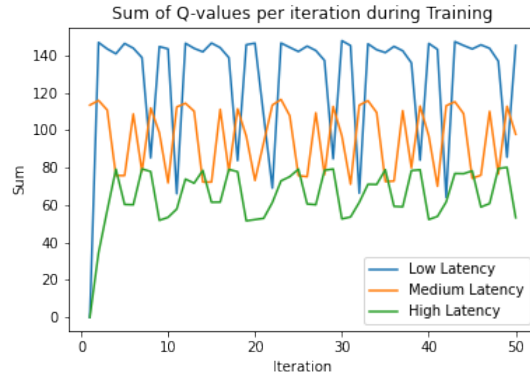

Figure 8.10: Sum of Q-values during Training per iteration

other latency label groups. The medium latency (Orange line) shows a variation in the sum between 80 and 120 also with periodical peaks and troughs. The high latency (Green line) shows a fluctuation between 40 and 100, showing the lowest values from the three latency label groups.

All three latency groups start from different sums, with the Low Latency Q-table sum starting the highest. The fluctuation is observed during the middle training iterations. By the end of the training, the low latency Q-table sum was the highest, followed by medium and high latency.

In general, the sum of low latency Q-table is the highest throughout the process of training, which seems to have a better performance compared to the medium and high Q-table sum. The peaks and troughs show ongoing changes and learning during training.

In order to explain further the optimal policy presented with Q-tables, they have been analyzed to the level of state and action pairs. Subsections 8.2.3 and 8.3.4 present the best (most chosen) state and action for each latency cluster. This is performed for both datasets: sequential (Table 8.1) and parallel function invocation 8.2. The two tables contain information about the total number of invocations, the most chosen state, the number of the most chosen state, the most chosen action, and the number of the most chosen action. The information provided is per latency cluster.

8.2.3 Best state/action for sequential function invocation

Table 8.1 provides information about the best state/action per latency cluster for sequentially invocated functions. It shows that more than half of the function invocation was clustered as high, followed by medium and low latency ones. For high latency, the most common state was high CPU and medium Memory (13816), and the most chosen action was Decrease CPU usage (13851). For medium latency, the most common state was medium CPU and medium Memory (6569), and the most chosen action was Decrease memory usage (6569). Lastly, for high latency, the most common state was low CPU and medium Memory (4615), and the most chosen action was Maintain memory usage (4615).

	Total number of invocations	Most chosen state	Number of most chosen state	Most chosen action	Number of most chosen action
low	9272	low CPU medium memory	4615	Maintain memory	4615
medium	13046	medium CPU medium memory	6569	Decrease Memory	6569
high	27667	high CPU medium memory	13816	Decrease CPU	13851

Table 8.1: Best state/action for each latency category for sequential invocation

	Total number of invocations	Most chosen state	Number of most chosen state	Most chosen action	Number of most chosen action
low	9818	medium CPU low memory	4894	Maintain memory	4894
medium	20932	medium CPU high memory	10492	Maintain CPU	10492
high	19111	high CPU medium memory	9014	Decrease CPU	9014

Table 8.2: Best state/action for each latency category for parallel invocation

8.2.4 Best state/action for parallel function invocation

Table 8.2 provides information about the best state/action per latency cluster for parallel invocation of functions. It shows that most of the function invocations were clustered as medium, followed by high and low latency ones. For medium latency, the most common state was medium CPU and high Memory (10492), and the most chosen action was Maintain CPU usage (10492). For high latency, the most common state was high CPU and medium Memory (9014), and the most chosen action was Decrease CPU usage (9014). Lastly, for low latency, the most common state was medium CPU and low Memory (4894), and the most chosen action was Maintain memory usage (4894).

8.2.5 Average reward during evaluation

Functions being invoked one by one

The average reward per invocation during the evaluation of one-by-one function invocation is shown in Figure 8.11. The average reward measures start at around 0.41 over the first iterations. Then some function is observed reaching peaks at 0.44 and troughs to 0.38 in the middle iteration. The highest average reward is 0.46. By the end of the evaluation, the average reward goes to 0.40.

Generally, the performance in evaluation shows improvements and setbacks, with a periodic fluctuation between 0.36 and 0.46.

Parallel function invocations

Figure 8.12 illustrates the average reward during evaluation per iteration when function are being invocated in parallel. The average reward starts at around 0.41 during

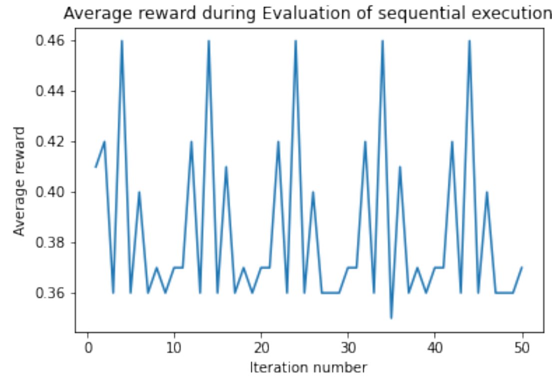

Figure 8.11: Average reward during evaluation per iteration with sequential function invocation

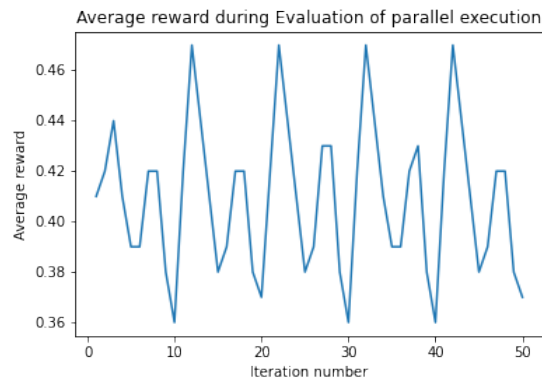

Figure 8.12: Average reward during evaluation per iteration using parallel function invocation dataset

the first iterations. The values show fluctuations reaching a peak of 0.47 and then troughs to around 0.36. At the end of the evaluation, the average reward goes to around 0.40.

Both scenarios in Figure 8.11 and 8.12 show similar results with the parallel function performing slightly better. However, during the evaluation of two datasets fluctuations were observed which indicates the dynamic nature of function invocation.

8.3 Cost efficiency for Method

Another important characteristic in addition to the Method's accuracy is how cost-efficient the method and usage of the resources. This is also essential as usually allocating more resources imposes additional costs to the user. Thus, the method has been evaluated during training for end-to-end latency, CPU usage, and memory usage. These metrics were monitored in order to cost-efficiency of the method to be ensured.

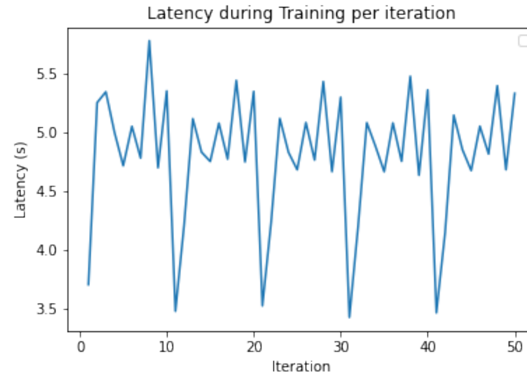

Figure 8.13: Latency in seconds of Method during Training per Iteration

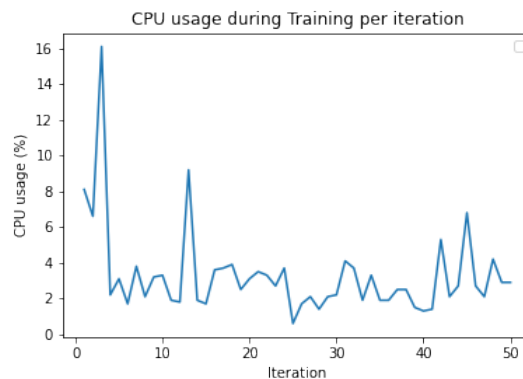

Figure 8.14: CPU usage in % of Method during Training per Iteration

8.3.1 Latency during Training per Iteration

Figure 8.13 represents the latency in seconds during training for 50 iterations. The line chart shows the change in latency over the period of training. Initially, the latency starts at less than 4 seconds during the first iterations. Then it fluctuates between 3.5 to above 5 seconds with a peak observed at 5.5. At the end, the latency drops down to 4.5 seconds. In general, the latency fluctuates during training but seems to stabilize in the end. The model seems to be trained in less time as each iteration takes around 5 seconds or less.

8.3.2 CPU usage during Training per Iteration

Figure 8.14 shows the CPU usage during training of the method per 50 iterations. The CPU usage is expressed as a percentage of the total CPU power. In the first iteration, the CPU usage starts at 8%, making a spike up to more than 16%. However, the value stabilizes after iteration 15 and it is between 2% to 8% for the rest of the training process. This showcases that the method does not require large CPU power in order to output satisfactory results.

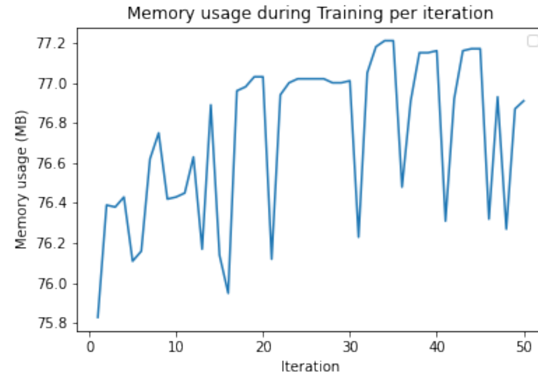

Figure 8.15: Memory usage in MB of Method during Training per Iteration

8.3.3 Memory usage during Training per Iteration

Figure 8.15 shows the Memory usage during training of the method per 50 iterations. The Memory usage is expressed as Megabytes (MB). It can be observed that the method requires a similar amount of memory power during the process of training (between 75 and 77.5 MB). The method first consumes less memory but as the number of iterations grows the memory usage increases steadily.

Generally, section 8.1 evaluated the datasets of function invocation for 2 scenarios (one-by-one invocation, parallel invocation) generated for evaluation of the method for latency, CPU and memory usage, and failure rate. It confirmed the need for a dynamically adjusting to the needs method for managing the cold start problem in Serverless computing and the approach of "one-size-fits-all" is not the most optimal one in combating the cold start problem. Section 8.2 showed how accurate the method is by evaluating the average reward functions during training and evaluation, alongside the sum of each Q-table during training. During training the method gave a 0.4 maximum average reward with values ranging from 0.22 to 0.40. The sum of Q-values in the Q-tables is the highest in the low-latency clusters showing that the method performs best there. During evaluation, the method showed a consistency with performance giving values of 0.38 to 0.48 average reward for both evaluation scenarios. Section 8.3 evaluated the method against its cost efficiency, namely for latency, CPU, and memory usage for training. The results showed that it takes around 5 seconds for the method to run an iteration of 10,000 episodes, which requires relatively low CPU resources (less than 8% during most iterations) and demands a consistent amount of memory (between 75MB and 77.5MB).

Chapter 9

Discussion and Conclusion

Serverless computing is a recent discovery of PaaS first introduced by AWS as Lambda Functions [9]. This service model offers many advantages such as reduced operational overhead, improved scalability, cost efficiency, and faster development. However, this service mode has its limitations in the face of a Cold start problem which is the latency accumulated when starting a new instance of a function. This can occur in two ways: the instance has been started for the first time, or the instance has not been invocated for long enough, thus it has been shut down by the system.

The cold start problem creates a big overhead mostly associated with latency and not adherence to Service Level Objective. To address this a multivocal literature review was conducted to identify the possible shortcoming of the existing literature. Several solutions have been proposed such as function fusion, frameworks for cold start mitigation, RL approaches, Deep learning approaches with Neural networks, and others. The RL approaches specifically Q-learning ones are the most popular due to their simplicity and their performance with the task given (RQ1, SRQ1.1-1.2). However, a few limitations of the existing Q-learning approaches include the exponentially growing size of Q-tables, not a single approach looked into memory usage, CPU usage, SLO, and package size (SRQ1.3).

By taking a look at limitations and future work, this Master thesis proposes a two-tier approach for cold start mitigation (RQ2). It includes a Q-learning agent and a K-means clustering algorithm. The first part of the method takes into account the growing size of a Q-table and proposes breaking it down into 3 Q-tables for low, medium, and high latency function invocations. Due to the dynamic nature of function invocations, the notion of one-size-fits-all does not apply here. Thus, three functions were defined for low, medium, and high latency, ensuring the method adapts to various situations dynamically. The reward functions consider CPU, memory usage and adherence to SLO but latency is also considered when selecting the suitable reward function. The second part of the method clusters the function invocations based on latency as the agent works best with discrete values, the functions were clustered based on latency: low, medium, and high (SRQ2.1).

A prototype of the method described above has been created to allow for the evaluation of the effectiveness of the proposed solution. A sample deployment to OpenFaaS which is an open source Serverless framework is proposed (SRQ2.2). The model has been trained with real historic data function invocation traces from the Huawei cloud and evaluated with function invocations deployed in OpenFaaS. For 10,000 episodes and 50 iterations, the average reward during training showed 0.40, and during evaluation 0.46 for one-by-one function invocation and 0.47 for parallel invocation (for 50 iterations of 500 episodes), showing that the method works well with different workflows (RQ3). Moreover, the method works best with low-latency function invocations. Additionally, the method is cost-efficient with CPU usage below 8% and memory usage consistency between 75 MB and 77 MB per iteration.

The latency per iteration was around 5 seconds (SRQ3.1).

9.1 Limitations

The method comes with a proposed implementation that was not replicated within the real prototype. Only some of the components such as the K-means algorithm and not the final version of the Q-learning agent have been deployed in an open source Serverless framework. Initially, the framework was chosen to be Open Whisk but due to complications with installation from the operating system, OpenFaaS was chosen instead. Moreover, the Flask API has not been implemented due to time constraints.

As can be seen, the model performs with the highest average reward of 0.40 for training and 0.46 for evaluation which fails to converge and gives sub-optimal results. The training has been run for 50 and 100 iterations and no significant changes were observed. Additionally, the Q-learning model only considers CPU usage, memory usage, latency, and SLO adherence but ignores other factors such as network delays, a workflow model of function invocations (concurrency, conditional invocation), dependencies loading, and programming language overhead.

Also, the effectiveness of the method might be decreased due to extreme workloads limiting the RL approach in adapting to highly dynamic cases. While the method shows that cold start latency is reduced to some extent, it does not fully eliminate it, functions with high demands for provisioning might still experience high cold start latency. The performance of the Q-learning agent depends highly on adjusting the hyperparameters such as the learning rate and discount factor).

Moreover, the prototype was planned to be deployed on an Open source Serverless framework but might perform differently on a commercial size one such as AWS Lambda or Azure Functions. What is more, the training was done with function traces from a single cloud provider, and the evaluation datasets were generated from the invocation of functions in an open-source framework which might make it hard for the results presented in this project to be generalized. Instead of comparing the prototype to existing solutions, the method was evaluated with historical data of function invocation traces. There are also other state-of-the-art approaches that the prototype can be compared to such as AI-driven autoscalers.

The prototype was developed locally on a personal computer and data was generated on a virtual machine which indicates that the experiment that evaluates the method was conducted in a controlled environment which might not fully represent the real-life cloud infrastructure with some other behavior such as network latency variation or multi-tenancy. Lastly, the evaluation of the method might be biased due to the metrics defined contained a bias. CPU usage, memory usage, and average reward might not be the perfect metrics for method evaluation.

The full evaluation of the method was not fully completed. In other words the method's output (the optimal scheduling policy) was partially evaluated. The method produces Q-values for a set of states and actions. The bigger the Q-value is, the better the action is to the corresponding state. Table 8.1 and 8.2 showcase the best action for low, medium, and high latency clustered invocations for sequential and parallel function invocation. However, the next step of inputting the optimal policy (set of states and actions), in order to observe how much cold start latency is reduced was not performed due to time constraints.

9.2 Future work

Mitigating the cold start problem in Serverless is not a black or white solution which is one-size-fits-all. Considering this more advanced solutions such as Deep Q-learning or multi-agent Q-learning with multiple reward functions and Q-tables might show better results. Also, methods combining predictive models or RL approaches with meta-learning could be more suitable for the task. Methods exploring both cold start reduction, energy, and cost efficiency could be a real asset as they provide a sustainable solution to multiple problems in cloud computing. Finally, the method could benefit a cross-platform validation or deploying it on multiple Serverless cloud providers not only open source ones would greatly reduce the problem of a simulated environment as it will take the method in a fully workable real-life environment.

Bibliography

[1] Gojko Adzic and Robert Chatley. "Serverless computing: economic and architectural impact". In: *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 2017, pp. 884–889.

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

[3] Siddharth Agarwal, Maria Rodriguez Read, and Rajkumar Buyya. "On-Demand Cold Start Frequency Reduction with Off-Policy Reinforcement Learning in Serverless Computing". In: *Available at SSRN 4661993* ().

[4] Istemi Ekin Akkus et al. "{SAND}: Towards {High-Performance} serverless computing". In: *2018 Usenix Annual Technical Conference (USENIX ATC 18)*. 2018, pp. 923–935.

[5] Areej Alabbas et al. "Performance analysis of Apache openwhisk across the edge-cloud continuum". In: *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. IEEE. 2023, pp. 401–407.

[6] Amazon Web Services. *AWS Lambda Release*. Released on November 13, 2014. 2014. URL: https://aws.amazon.com/releasenotes/AWS-Lambda/8269001345899110.

[7] Lixiang Ao et al. "Sprocket: A serverless video processing framework". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2018, pp. 263–274.

[8] Michael Armbrust et al. "A view of cloud computing". In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[9] Author(s). *What makes serverless architectures so attractive?* Accessed: December 20, 2023. 2016. URL: https://developer.ibm.com/opentech/2016/09/06/what-makesserverless-attractive/.

[10] Ioana Baldini et al. "Serverless computing: Current trends and open problems". In: *Research advances in cloud computing* (2017), pp. 1–20.

[11] Daniel Barcelona-Pons et al. "Faas orchestration of parallel workloads". In: *Proceedings of the 5th International Workshop on Serverless Computing*. 2019, pp. 25–30.

[12] Ahmed Barnawi et al. "The views, measurements and challenges of elasticity in the cloud: A review". In: *Computer Communications* 154 (2020), pp. 111–117.

[13] James Cadden et al. "SEUSS: skip redundant paths to make serverless fast". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–15.

[14] Paul Castro et al. "The rise of serverless computing". In: *Communications of the ACM* 62.12 (2019), pp. 44–54.

[15] Dheeraj Chahal et al. "Migrating large deep learning models to serverless architecture". In: *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE. 2020, pp. 111–116.

[16] Deyan Chen and Hong Zhao. "Data security and privacy protection issues in cloud computing". In: *2012 international conference on computer science and electronics engineering*. Vol. 1. IEEE. 2012, pp. 647–651.

[17] Claudio Cicconetti, Marco Conti, and Andrea Passarella. "A decentralized framework for serverless edge computing in the internet of things". In: *IEEE Transactions on Network and Service Management* 18.2 (2020), pp. 2166–2180.

[18] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. "Xanadu: Mitigating cascading cold starts in serverless function chain deployments". In: *Proceedings of the 21st International Middleware Conference*. 2020, pp. 356–370.

[19] Tharam Dillon, Chen Wu, and Elizabeth Chang. "Cloud computing: issues and challenges". In: *2010 24th IEEE international conference on advanced information networking and applications*. Ieee. 2010, pp. 27–33.

[20] Dong Du et al. "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 467–481.

[21] Simon Eismann et al. "A review of serverless use cases and their characteristics". In: *arXiv preprint arXiv:2008.11110* (2020).

[22] Lang Feng et al. "Exploring serverless computing for neural network training". In: *2018 IEEE 11th international conference on cloud computing (CLOUD)*. IEEE. 2018, pp. 334–341.

[23] Alexander Fuerst and Prateek Sharma. "FaasCache: keeping serverless computing alive with greedy-dual caching". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021, pp. 386–400.

[24] Gartner. *The CIO's Guide to Serverless Computing*. Accessed on: 2023-12-15. 2020. URL: https://www.gartner.com/smarterwithgartner/the-cios-guide-to-serverless-computing/.

[25] Muhammed Golec et al. "Cold start latency in serverless computing: A systematic review, taxonomy, and future directions". In: *arXiv preprint arXiv:2310.08437* (2023).

[26] Muhammed Golec et al. "HealthFaaS: AI based Smart Healthcare System for Heart Patients using Serverless Computing". In: *IEEE Internet of Things Journal* (2023).

[27] Lewis Golightly et al. "Adoption of cloud computing as innovation in the organization". In: *International Journal of Engineering Business Management* 14 (2022), p. 18479790221093992.

[28] Jake Grogan et al. "A multivocal literature review of function-as-a-service (faas) infrastructures and implications for software developers". In: *Systems, Software and Services Process Improvement: 27th European Conference, EuroSPI 2020, Düsseldorf, Germany, September 9–11, 2020, Proceedings 27*. Springer. 2020, pp. 58–75.

[29] Adam Hall and Umakishore Ramachandran. "An execution model for serverless functions at the edge". In: *Proceedings of the International Conference on Internet of Things Design and Implementation*. 2019, pp. 225–236.

[30] Hassan B Hassan, Saman A Barakat, and Qusay I Sarhan. "Survey on serverless computing". In: *Journal of Cloud Computing* 10.1 (2021), pp. 1–29.

[31] Beakcheol Jang et al. "Q-Learning Algorithms: A Comprehensive Classification and Applications". In: *IEEE Access* 7 (2019), p. 123456. DOI: 10.1109/ACCESS.2019.2941229.

[32] Mehrdad Jangjou and Mohammad Karim Sohrabi. "A comprehensive survey on security challenges in different network layers in cloud computing". In: *Archives of Computational Methods in Engineering* 29.6 (2022), pp. 3587–3608.

[33] Hongseok Jeon et al. "Deep reinforcement learning for qos-aware package caching in serverless edge computing". In: *2021 IEEE Global Communications Conference (GLOBECOM)*. IEEE. 2021, pp. 1–6.

[34] Artjom Joosen et al. "Serverless Cold Starts and Where to Find Them". In: *arXiv preprint arXiv:2410.06145* (2024).

[35] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. "Centralized core-granular scheduling for serverless functions". In: *Proceedings of the ACM symposium on cloud computing*. 2019, pp. 158–164.

[36] M Kriushanth, L Arockiam, and G Justy Mirobi. "Auto scaling in Cloud Computing: an overview". In: *International Journal of Advanced Research in Computer and Communication Engineering* 2.7 (2013), pp. 2278–1021.

[37] Anisha Kumari and Bibhudatta Sahoo. "ACPM: adaptive container provisioning model to mitigate serverless cold-start". In: *Cluster Computing* (2023), pp. 1–28.

[38] Snehalika Lall et al. "Multi-agent reinfocement learning for stochastic power management in cognitive radio network". In: *2016 International Conference on Microelectronics, Computing and Communications (MicroCom)*. IEEE. 2016, pp. 1–6.

[39] Seungjun Lee et al. "Mitigating cold start problem in serverless computing with function fusion". In: *Sensors* 21.24 (2021), p. 8416.

[40] Jyri Lehvä, Niko Mäkitalo, and Tommi Mikkonen. "Case study: building a serverless messenger chatbot". In: *International Conference on Web Engineering*. Springer. 2017, pp. 75–86.

[41] Junfeng Li et al. "Analyzing open-source serverless platforms: Characteristics and performance". In: *arXiv preprint arXiv:2106.03601* (2021).

[42] Yongkang Li et al. "Serverless computing: state-of-the-art, challenges and opportunities". In: *IEEE Transactions on Services Computing* 16.2 (2022), pp. 1522–1539.

[43] Zhe Li et al. "A Survey of Cost Optimization in Serverless Cloud Computing". In: *Journal of Physics: Conference Series*. Vol. 1802. 3. IOP Publishing. 2021, p. 032070.

[44] Ping-Min Lin and Alex Glikson. "Mitigating cold starts in serverless platforms: A pool-based approach". In: *arXiv preprint arXiv:1903.12221* (2019).

[45] Qingyuan Liu et al. "Harmonizing efficiency and practicability: optimizing resource utilization in serverless computing with JIAGU". In: *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 2024, pp. 1–17.

[46] Xuanzhe Liu et al. "FaaSLight: general application-level cold-start latency optimization for function-as-a-service in serverless computing". In: *ACM Transactions on Software Engineering and Methodology* (2023).

[47] Jeremy Lloyd. "Containers and Serverless". In: *Infrastructure Leader's Guide to Google Cloud: Lead Your Organization's Google Cloud Adoption, Migration and Modernization Journey*. Springer, 2022, pp. 255–272.

[48] Mahzad Mahdavisharif, Shahram Jamali, and Reza Fotohi. "Big data-aware intrusion detection system in communication networks: a deep learning approach". In: *Journal of Grid Computing* 19 (2021), pp. 1–28.

[49] Johannes Manner, Stefan Kolb, and Guido Wirtz. "Troubleshooting serverless functions: a combined monitoring and debugging approach". In: *SICS Software-Intensive Cyber-Physical Systems* 34 (2019), pp. 99–104.

[50] Dan C Marinescu. *Cloud computing: theory and practice*. Morgan Kaufmann, 2022.

[51] Peter Mell and Tim Grance. "Draft NIST working definition of cloud computing". In: *Referenced on June. 3rd* 15.32 (2009), p. 2.

[52] Sean Meyn. "Stability of Q-Learning Through Design and Optimism". In: *arXiv preprint arXiv:2307.02632* (2023).

[53] Chetankumar Mistry et al. "Demonstrating the practicality of unikernels to build a serverless platform at the edge". In: *2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2020, pp. 25–32.

[54] Anup Mohan et al. "Agile cold starts for scalable serverless". In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. 2019.

[55] Tam N Nguyen. "Managing Cold-start in The Serverless Cloud with Temporal Convolutional Networks". In: *arXiv preprint arXiv:2304.00396* (2023).

[56] Tam n. Nguyen. "Holistic cold-start management in serverless computing cloud with deep learning for time series". In: *Future Generation Computer Systems* 153 (Apr. 2024), pp. 312–325. ISSN: 0167-739X. DOI: `10.1016/j.future.2023.12.011`. URL: `http://dx.doi.org/10.1016/j.future.2023.12.011`.

[57] Justice Opara-Martins, Reza Sahandi, and Feng Tian. "Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective". In: *Journal of Cloud Computing* 5 (2016), pp. 1–18.

[58] Shanxing Pan et al. "Sustainable serverless computing with cold-start optimization and automatic workflow resource scheduling". In: *IEEE Transactions on Sustainable Computing* (2023).

[59] Roland Pellegrini, Igor Ivkic, and Markus Tauber. "Towards a security-aware benchmarking framework for function-as-a-service". In: *arXiv preprint arXiv:1905.07228* (2019).

[60] Haoran Qiu et al. "Reinforcement learning for resource management in multi-tenant serverless platforms". In: *Proceedings of the 2nd European Workshop on Machine Learning and Systems*. 2022, pp. 20–28.

[61] Ali Raza et al. "Sok: Function-as-a-service: From an application developer's perspective". In: *Journal of Systems Research* 1.1 (2021).

[62] Sasha Rosenbaum. *Serverless computing in Azure with. NET*. Packt Publishing Ltd, 2017.

[63] Martin Sarnovsky and Jan Paralic. "Hierarchical intrusion detection using machine learning and knowledge model". In: *Symmetry* 12.2 (2020), p. 203.

[64] Johann Schleier-Smith et al. "What serverless computing is and should become: The next phase of cloud computing". In: *Communications of the ACM* 64.5 (2021), pp. 76–84.

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

[66] Hichem Sedjelmaci, Sidi Mohammed Senouci, and Mosa Ali Abu-Rgheff. "An efficient and lightweight intrusion detection mechanism for service-oriented vehicular networks". In: *IEEE Internet of things journal* 1.6 (2014), pp. 570–577.

[67] Mohit Sewak and Sachchidanand Singh. "Winning in the era of serverless computing and function as a service". In: *2018 3rd International Conference for Convergence in Technology (I2CT)*. IEEE. 2018, pp. 1–5.

[68] Vaishaal Shankar et al. "Serverless linear algebra". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, pp. 281–295.

[69] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. "Prebaking functions to warm the serverless cold start". In: *Proceedings of the 21st International Middleware Conference*. 2020, pp. 1–13.

[70] Khondokar Solaiman and Muhammad Abdullah Adnan. "WLEC: A not so cold architecture to mitigate cold start problem in serverless computing". In: *2020 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2020, pp. 144–153.

[71] GALURA MUHAMMAD SURANEGARA et al. "SERVERLESS CLOUD COMPUTING DEPLOYMENT FOR PRE-TRAINED MACHINE LEARNING MODEL". In: *Journal of Engineering Science and Technology* 19.4 (2024), pp. 72–79.

[72] Jayson E Tamayo. "A Literature Review on the challenges of using Serverless Computing in Web Services". In: *Asian Journal of Business and Technology Studies* 3.1 (2020).

[73] Vamsi Krishna Thatikonda. "Serverless Computing: Advantages, Limitations and Use Cases". In: *European Journal of Theoretical and Applied Sciences* 1.5 (2023), pp. 341–347.

[74] Kanchan Tirkey et al. "A Novel Function Fusion Approach for Serverless Cold Start". In: *2023 International Conference on Communication, Circuits, and Systems (IC3S)*. IEEE. 2023, pp. 1–5.

[75] Achilleas Tzenetopoulos et al. "Faas and curious: Performance implications of serverless functions on edge computing platforms". In: *High Performance Computing: ISC High Performance Digital 2021 International Workshops, Frankfurt am Main, Germany, June 24–July 2, 2021, Revised Selected Papers 36*. Springer. 2021, pp. 428–438.

[76] Dmitrii Ustiugov et al. "Benchmarking, analysis, and optimization of serverless function snapshots". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021, pp. 559–572.

[77] Parichehr Vahidinia, Bahar Farahani, and Fereidoon Shams Aliee. "Mitigating cold start problem in serverless computing: a reinforcement learning approach". In: *IEEE Internet of Things Journal* 10.5 (2022), pp. 3917–3927.

[78] Erwin Van Eyk et al. "The SPEC cloud group's research vision on FaaS and serverless architectures". In: *Proceedings of the 2nd international workshop on serverless computing*. 2017, pp. 1–4.

[79] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. "Replayable execution optimized for page sharing for a managed runtime environment". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019, pp. 1–16.

[80] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8 (1992), pp. 279–292.

[81] Jinfeng Wen et al. "Rise of the planet of serverless computing: A systematic review". In: *ACM Transactions on Software Engineering and Methodology* (2023).

[82] Jinfeng Wen et al. "Unveiling Overlooked Performance Variance in Serverless Computing". In: *arXiv:2305.04309v2* (2025). URL: `https://arxiv.org/abs/2305.04309v2`.

[83] Claes Wohlin et al. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[84] Anastasios Zafeiropoulos et al. "Reinforcement learning-assisted autoscaling mechanisms for serverless computing platforms". In: *Simulation Modelling Practice and Theory* 116 (2022), p. 102461.