National College of Ireland

Leveraging Machine Learning to Reduce Cold Start Latency of Containers in Serverless Computing

MSc Research Project
Cloud Computing

Ryan Bannon
Student ID: 14488478

School of Computing
National College of Ireland

Supervisor: Horacio Gonzalez-Velez

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Ryan Bannon
Student ID:	14488478
Programme:	Cloud Computing
Year:	2022
Module:	MSc Research Project
Supervisor:	Horacio Gonzalez-Velez
Submission Due Date:	15/08/2022
Project Title:	Leveraging Machine Learning to Reduce Cold Start Latency of Containers in Serverless Computing
Word Count:	7220
Page Count:	23

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Ryan Bannon
Date:	15th September 2022

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Abstract

Leading cloud service providers offer Function-as-a-Service (FaaS) which allows users to outsource the provisioning and management of servers and focus solely on the business logic that drives their operations. This means users can run code in stateless compute containers living in an event driven ecosystem which is fully managed by the cloud provider. Therefore, this model masks the underlying infrastructure and implementation stack that these services are built upon. The core principles of the microservices approach include decentralisation, fault tolerance, continuous delivery, and deployment. However, decentralisation within an event-driven domain often results in chain reaction. Functions invoke services which call other function and so on. As such, one major drawback in serverless computing is a problem called cold start. The nature of these services forces the scaling components to warm, freeze, thaw and terminate container state, often times at the expense of latency upon subsequent invocations. This paper address this problem with a recent iteration of the LSTM neural network called Gated Recurrent Unit (GRU). Results from experiments that simulated real Azure Functions traces, recorded improvements of up to 28% with its function execution predictions and cold start solution.

1 Introduction

With respect to the many benefits serverless platforms have to offer, this paper explores the effectiveness of machine learning techniques at reducing the impact latency of cold starts. Machine learning (ML) models and techniques find hidden patterns in datasets that would not be otherwise known. This aligns with the expectations of real-world applications where the invocation of functions is unpredictable. In complex enterprises, improving these response times is crucial. By examining and reviewing past trends and extrapolating useful information from previous trends, this paper examines the impact machine learning can have at accurately warming serverless containers to achieve overall superior performance. The designed solution is currently limited to executing Python based programming workloads. This limitation is an area of focus in the future work section of this research report.

1.1 Research Question

Can machine learning techniques efficiently pre-warm function containers in a bid to reduce the latency of cold starts when compared to a vanilla serverless platform?

By design the orchestration of containers and their lifecycles often scales up and down from a range of $0 \rightarrow N$. Workarounds such as the keep-alive were introduced to counteract this aggressive approach to container freezing and termination. Keep-alive for instance, involves pinging serverless functions at a given time interval with fake data packets which invokes the service yet does not require any response from it. This approach among others ensures that functions remain active in a warm state. However, these universal configurations are not feasible in enterprise applications where load and demand themselves are not guaranteed. There are other trade-offs to be considered too from a memory perspective. Although constantly keeping containers alive sounds beneficial, this can drive the memory footprint of the server up. Without the natural

fluctuation of demand, the compute resources on the server could be consistently running at higher levels than otherwise required [1]. As a result, this may slow executions down, which presents a counter-productive problem of its own considering the goal is to reduce latency of requests. This of course, is a paradox that we must avoid. Using machine learning this paper investigates an optimal approach for finding a balance while also mitigating the impact of cold starts.

According to Verified Market Research, the serverless architecture was valued at $3 billion US dollars in 2017. Three years later in 2020, the market size was valued over $7 billion dollars. All signs indicate that this trajectory will continue, with predictions estimating growth value of $36 billion dollars by 2028 [2]. A major draw towards the cloud for many organisations has been embracing the emergence of serverless, due to its programming simplicity [3]. The industry shift is not met without its own challenges though. Functions that have been left idle periodically, understand to have an additional overhead of initialisation before they can be made actionable. This paper recognises this evident drawback of within this field and examines solutions with a keen focus on another thriving area of the industry in machine learning. The cloud has generated immense interest in machine and more specifically to this paper, deep learning due to its on-demand and wide ranging computational offerings [4].

Containerisation is the enabler for serverless computing. Behind the scenes container orchestration is driving the delivery of these services in question. A container is an isolated software unit where all necessary code and dependencies can run reliably, independent of the operating system it sits on. This completely streamlines deployment and all but eradicates old monolith strategies in this area. The orchestration of containers essentially means the challenge of automating the provisioning and management of them. Useful tools such as Kubernetes have been implemented widespread among the industry to tackle this issue. This does however, introduce the cold start problem previously stated. Cold starts are a challenge in each iteration of serverless computing with related work attempting to tackle the issue. AWS Lambda, Microsoft Azure Functions and Google Cloud Functions are three of the leading serverless platforms in the industry.

Also, in any business or organisation capital is the bottom-line. The concept of zero-administration is appealing for users but means less to those in departments in charge of paying for such services. Serverless like many other aspects of the cloud, abides by a pay-per-execution or computation model. Therefore, rewards can be found by developing fast running code, but trying to find the balance between performance and efficiency can be difficult. To demonstrate this point further, returning to keep-alive workaround for reducing latency, forcing the containers to remain in a warm state has cost implications that may otherwise be avoided.

1.2 Research Objectives

- Install and configure an open-source serverless platform on multiple virtual machine instances in the cloud
- Automate real-world serverless behaviour with simulations of an official Microsoft Azure Functions dataset through a load testing tool

- Leverage machine learning techniques to interrogate traces and predict function execution patterns
- Apply custom helper modules in a bid to decrease the impact of cold start latency
- Evaluate the tests based on pre and post solution metrics of serverless function executions

2 Related Work

Related work to this study correlates to the major components that build and facilitate serverless architectures. At the highest level, FaaS solutions require an environment which allows them to run and execute their business logic. Containers are currently the leading technology implemented to solve this challenge of standing such environments up reliably at speed and scale. The functions also require runtime libraries and dependencies are downloaded before they can become ready for use. Unpacking these statements, it becomes increasingly clear where latency can begin to creep into these systems. Cold start latency is a well-known and documented challenge in the serverless field. Many academic papers have been proposed and released in a bid to resolve this issue. This research report will explore and critically analyse such papers.

2.1 Runtime Awareness

Container lifecycle-aware scheduling is an area researchers have identified as a cold start prevention mechanism. CAS is a scheduling strategy on Apache OpenWhisk that is astute to each containers state in the serverless functions ecosystem [5]. As such, allocating the invocation of functions in certain containers can be optimised and more efficient with said awareness. Latency reduction of up to 81% for cold starts have been observed with this strategy. Similarly, the concept of function-execution scheduling was presented which ensures the declarative language of Allocation Priority Policies are specified with order of magnitude for functions [6]. These policies are fully managed and maintained by the developers, which lends greater control and flexibility back to them. A drawback of such methodologies are they are manually defined configurations. Package-aware scheduling is a concept whereby the reuse of function libraries and dependencies is coherently retrieved from cached instances. Optimising the use of pre-cached packages can significantly reduce function execution speeds [7]. The proposed algorithm assigns function invocations to the same worker nodes that have the necessary dependencies pre-loaded. As a result, the latency for the most part is reduced to the execution time of the function itself. A 66% performance improvement with this scheduling algorithm was reported. Package-aware scheduling reduces the number of cold starts required, but it cannot reduce the latency of them when they inevitably arise. It also has a tendency to overload the same container, which diminishes the work of load balancers and auto-scaling operations.

Caching falls under this criteria also. Reducing the volume of data download in any event, often equates to increased speed. In this instance, libraries, modules, and dependencies within serverless functions have the overhead of being downloaded at runtime. A proposed solution is an in-memory cache of the serverless runtime environment which reduces latency of network calls [8]. A major flaw in this approach is the suspension of

container state, which renders the cache void. In-memory cache technology Redis as an external resource has been explored, which can double the speed of dependency installation. However, this was found to have a negative impact compared to standard AWS Lambda executions when a fault occurs – or cache misses – with Redis. Pipsqueak is a caching interpreter for large Python based packages. With heavy cache volumes, stale or unpopular entries are evicted in order to release idle computational resources that are being inadequately consumed [9]. Another valuable function of caching is using it as a means for the keep-alive method [1]. As a result of the volatility of container state, keeping such environments active and alive is a plausible mitigation solution for cold starts. FaasCache proposes the Greedy-Dual algorithm based particularly around the Least-Recently-Used policy. This cache policy focuses on expelling the lesser prominent artifacts. While FaasCache is effective at reducing cold start overhead up to three times faster, it is conceded that keeping these services alive has its pitfalls. The utilisation and memory footprint on the cache server also raises performance trade-off inquiries.

2.2 Warm & Pre-warming Pools

Kubernetes is being increasingly adopted across the industry as an orchestration tool to achieve this automated container management and provisioning. Naturally, when components are designed to expand and contract, the system becomes erratic. This does not suggest it is unstable. However, there are many moving parts to the system. This elasticity is a prime candidate for cold starts because the setup and configuration of on-demand provisioning is unavoidable. Many researchers have explored the notion of "warm" pools of containers that remain in a consistently runnable state. Pre-provisioned warm containers has resulted in 85% reduction in latency as opposed to cold start executions in some cases [10]. This can be directly correlated with the near zero overhead required for serving the incoming requests. This is achieved with pod migration processes with Knative control flows that pertain auto scaling logic based on the state of the warm pool. Incoming traffic is routed where possible to the warm containers, improving latency of the serverless functions. Pre-warming containers is a step behind managed warm pools. This entails a list of containers that are switched on with relevant libraries and dependencies downloaded. The difference between being considered warm in this instance is, the existing libraries are not yet loaded. Cold start latency is substantially reduced with network and libraries tasks pre-completed and containers pre-provisioned [11].

A major disadvantage to both pre-warming and warm pool approaches is cost. Both methods undoubtedly reduce cold starts, but they do so at the expense of ongoing consumption. In these cases where containers are initialised, the FaaS owner is being billed regardless of actual computational output and thus productivity. The strategy of snapshotting has been prototyped which assumes restoring a function from a snapshot is faster than building the runtime from scratch during cold starts [12]. This approach does not require a drastic change of the orchestration service, instead it simply pulls the function snapshot from a registry. These snapshots are compiled during every initial build and subsequent push of the function, which are referred to as checkpoints. Experiments with the pre-baked snapshots recorded performance improvements ranging from 125% to 1930% for Java workloads.

2.3 Machine Learning Predictions

Related work in cold start mitigation strategies have recently been guided in the direction of machine learning prediction models. The theory behind this being cold starts are inevitable in systems that are designed to scale down to zero. When requests are made and the environment is not ready to process it, provisioning and configuration is not an option. Administering and maintaining warm and pre-warm pools often results in wasted resources, and in turn, capital. With the progression of machine learning in the technology industry, it is not irrational to contemplate the prediction of function use. An Adaptive Warm-Up solution based on the Long-Short Term Memory (LSTM) model to prepare environments for function invocation through linear regression has been examined [13]. Furthermore, utilising LSTM models as a means to predict machine learning inference serving workloads has been theorised [14]. Model Ark (MArk) provisions Amazon EC2 and Lambda instances according to the model output, reducing the cold start concerns with pre-provisioned resources. Similar LSTM deep learning prediction models have also been used to forecast serverless function execution, which recorded a minimum 1.8 x times improvement over standard Apache OpenWhisk installations [15].

ETAS is another predictive scheduling algorithm. ETAS is implemented on Apache OpenWhisk platform, which schedules resource provisioning and releasing based on past function execution times, invocation history, and container status [16]. ETAS outperforms vanilla OpenWhisk latency by 30%. The development of reinforcement learning models has also gathered momentum in a bid to shorten cold start latency in serverless computing. Applied techniques such as Q-Learning and DynaQ+ which operate under a reward or penalise formation. The iterative goal of these algorithms is to predict the necessity of serverless resources. Based on performance, the model either receives or is deducted points. This structure promotes continuous motivation for the agent to deliver and learn over time [17].

2.4 Comparison Table

Table 1 contains a breakdown of methodologies from related work compared against this study.

Table 1: Comparison table

References	Runt. Awareness	Warm & Pre-warming Pools	ML	Real Data Simulations
[5]	✓			
[7]	✓			
[10]		✓		
[11]	✓	✓		
[12]		✓		
[13]			✓	✓
[14]		✓	✓	
[15]	✓	✓	✓	
[16]	✓	✓	✓	
[17]			✓	✓
This study	✓	✓	✓	✓

3 Methodology

3.1 Data Collection

The collection of data was a catalyst for the implementation and validation of the propsed solution. A dataset supplied online by Microsoft using the GitHub platform [18] was identified with documented real-world serverless invocation patterns. The dataset was released with log traces recorded on Microsoft Azure Functions over a sustained two week period starting on 31st January 2021 at 00:00. This dataset has been published and made available to the public under a Creative Commons Attribution 4.0 License by Microsoft Azure [19]. This file containing 1.98 million rows of data was legally copied, modified and redistributed for academic purposes.

In addition to the Microsoft dataset, the development and execution of the project experiments produced large volumes of data in the form of log files. These logs are designed for subsequent consumption by machine learning models to predict future invocations from past trends. No sensitive or personal information was stored or used in any dataset associated with this study.

3.2 Data Preparation

The knowledge discovery in databases (KDD) is an industry standard data mining methodology that was followed. Early analysis of the Microsoft dataset indicated additional attributes were required. The schema provided the following information of each function request:

- 'app' - encrypted serverless application id
- 'func' - encrypted serverless function id (unique within each application)
- 'end_timestamp' - timestamp (in seconds) when function execution finished
- 'duration' - total duration (in seconds) of execution

In Azure Functions, the unit of deployment is called an application, and an application has one or more functions. The total number of applications present in the dataset is 119, with 424 total functions. Nine functions in particular have over 50,000 records. They also account for 1.4 million of the total records, or 70% of the dataset. The inter arrival time of events do not follow the Poisson distribution model. While the distribution is positive and continuous (albeit discrete in the sample set), the probability of variable independence is a concern due to daily and weekly time patterns, $p(A \cap B) \neq p(A)p(B)$. The coefficient of variance is greater than 1, $CV = \sigma/\mu$, therefore, the data shows high levels of variability. The trace timestamps were modified from those returned in production to begin at zero before public release. The actual start timestamps were subsequently calculated with the end timestamp and duration values during implementation where $A =$ 31st January 2021 at 00:00.

$$start_timestamp_i = A + \Delta t(end_timestamp_i - duration_i)$$

With the records ordered sequentially along their natural timeline, the time difference

between the start timestamp of row n and row $n+1$ was calculated along the given axis. This attribute was critical for the simulation script to pause for x milliseconds between function requests.

$$delay_i = start_timestamp_{i+1} - start_timestamp_i$$

The load testing tool requires the delay variable x be in milliseconds. Using the below formula, the time spent in a paused state was equal to 14 days minus the initial time delay which is non-existent, as expected.

$$\frac{\sum_{delay_{i=1}}^{n}}{5^5 \times 4^4 \times 3^3 \times 2^2 \times 1^1} = 13.999$$

Additionally, with accurate datetime representations for each record, analysing some time properties offered interesting insights to the data. Wednesdays emerged with the greatest sum of executions across the two week period. Mondays and Tuesdays were also busy, whereas the remaining weekdays saw a significant decrease in traffic. While Saturdays recorded the least amount of function executions.

The implementation of this real-world scenario was sampled randomly on 2 days out of the total 14. These days were selected with a random module which returned subsets for Thursday 4th February 2021 and Friday 12th February 2021. The remaining 12 days of data were preserved for training operations for the machine learning models.

Figure 1: Azure Functions histogram by weekdays

3.3 Machine Learning Techniques

The machine learning techniques were an implemented strategy for predicting times in which functions would expect to execute. These identified timestamps are used by a running background process on the server to warm Docker containers and install the necessary dependencies at each given time minus the installation and docker creation times. The training data included the 12 days not used during the load testing experiments. As invocations arise, a module ensures system logs are stored in a particular directory where they can later be incorporated into the training datasets for future predictions. The prediction time interval was set to 6 hours for these experiments. The first machine learning technique used in this research was the Linear Regression algorithm. This was followed by an implementation of a Recurrent Neural Network (RNN) called Gated Recurrent Unit (GRU). An exploration of a conceptually similar neural network in Long Short Term Memory (LSTM) was also conducted. LSTM however, was never used in experiments as it was concluded that GRU produced better overall results during this study. Extensive research concluded these types of neural networks are extremely accurate at predicting time series data, but they were not the only algorithms considered. The supervised learning technique in Linear Regression was used and unsupervised learning methods are better suited to operations such as clustering. Other approaches such as Reinforcement learning were considered before opting with neural networks, as it was found that this learning type had been heavily explored in other cold start mitigation research. The key Python libraries and modules used were Pandas, Numpy and Tensorflow.

3.3.1 Linear Regression

Since data timestamp representations are continuous and linear, the motivation for applying Linear Regression was to test this fundamental strategy for predicting continuous time variables. This was an intriguing approach to benchmark basic statistical analysis before implementing a machine learning model with greater intricacies. The input x vector depicted the function execution events, while the predictions calculated the output y vector consisting of assumed numerical timestamps. The optimal lift and slope parameters for alpha hat and beta hat respectively, were calculated with the following formula, in order to minimize the mean squared error (MSE).

$$\hat{y}_i = \hat{\alpha} + (\hat{\beta} \times x_i), \text{where } \hat{\beta} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}, \text{and } \hat{\alpha} = \bar{y} - \hat{\beta}\bar{x}$$

An obvious downside to this approach is that timestamp y predictions follow a line, therefore, the interval between executions remains consistent along that same trajectory. Of course with future iterations over fresh training data, the slope will change as expected. While this doesn't accurately match real-world distribution, it provided some interesting findings and performed better than expected with latency times decreasing on average by 21%.

3.3.2 Gated Recurrent Unit (RNN)

Its difficult to discuss GRU Neural Networks without first discussing LSTM and more broadly Recurrent Neural Networks. RNNs are essentially networks with loops, which allow models to retain and repeatedly train on historically persisted information. The advantage to this is it provides greater context during predictions. An in-depth analysis of

the LSTM was completed initially due to its popularity in the industry as an RNN. GRU is fundamentally an LSTM model with fewer gates. These lesser gates manage to reduce complexity and the number of hyper-parameters required by the model, which typically promotes shorter fitting times. The fundamental principals from LSTM are still present in GRU however, with the update and reset gates ultimately determining what data it needs to keep, and what it needs to forget. RNNs typically receive 3-dimensional vectors as input which are passed to a hidden layer where they are recurrently processed in x cells before being sent to the output layer. Each sequence is a timestep which includes the vector of size n which is used to predict the y label of $n+1$. Often referred as the window size, the following matrix illustrates the concept of what the hidden layers in the network is attempting to predict with their computations. Interestingly, it was found during this study that a window size greater than 20 increased the time to fit the model, without any benefit in performance.

$$\begin{matrix} x \text{ inputs} & y \text{ labels} \\ \left\{ \begin{matrix} [1 & 2 & 3 & 4 & 5] \rightarrow \\ [2 & 3 & 4 & 5 & \underline{6}] \rightarrow \\ [3 & 4 & 5 & \underline{6} & \underline{7}] \rightarrow \end{matrix} \right\} & \left\{ \begin{matrix} [6] \\ [7] \\ [8] \end{matrix} \right\} \\ \ldots & \end{matrix}$$

While the data consisted of datetime representations for each function invocation, it became increasingly clear that greater context of time could be supplied to the model. Sine and cosine values were created based on the day, hour, and minute properties of events. The concept of time in these form (days, hours, and minutes) is cyclic. Therefore, patterns and relationships can be extracted by the network with these detailed inputs. The model showed improvements of up to 12% with this approach, versus those without it.

3.4 Validation Criteria

Returning to the problem statement and research question posed at the beginning of this paper, the only way to truly derive clear findings and metrics associated with the end product, is comparing the implementation to vanilla OpenWhisk instances. Analysing the total execution time with and without the solution applied will validate the existence of cold start reduction and serverless latency [15], which is the goal of this research. Apache JMeter is an open-source distributed load testing tool. JMeter was used to perform load testing operations that mirror the function executions of the Micorsoft Azure dataset against the serverless applications pre and post solution.

JMeter workloads operate based on configurations called test plans. The test plans are built and configured to accommodate the necessary use cases. Thread Groups allow for parallel processing of virtual users in isolated Java thread instances. The Azure dataset for this research requires many functions execute at similar time intervals which guarantees cross-over. As such, the thread group elements was a key enabler. The sampler elements facilitated the execution of command line arguments to run the functions. The results of the tests were captured through a listener element and written locally to a file as logs.

The validation of these experiments were achieved through a slight re-configuration of

the sampler inside the JMeter workloads. The initial tests directed the data flow through the standard OpenWhisk stack, while the secondary tests were transferred through a custom invoker module. The secondary tests included the resource handling logic for container creation as predicted by the machine learning models. The initial experiments were executed and ran uninterrupted for the expected total of 24 hours each. The logs were retrieved and kept for comparison against the secondary tests. The secondary experiments also used Jmeter to perform load testing, however, a smaller window of 6 hours was applied. In addition to function execution being automated through Jmeter, a custom controller module was running in the background which heated Docker containers according to the predicted time intervals from the Linear Regression and GRU outputs.

3.5 Tools and Platforms

In Table 2 a list of integral tools and platforms used in this research is outlined.

Table 2: Tools and Platforms

Type	Tool/Platform
Virtual Machines	Amazon Web Services (AWS) EC2
Operating System	Linux Ubuntu Server 18.04 LTS (HVM)
Serverless Platform	Apache OpenWhisk 1.0.0 (open-source)
Container Technology	Docker 20.10.17 CE
Machine Learning	Google Colab & Keras Tensorflow 2.8.0
Performance/Load Testing	Java (openjdk-11) & Apache JMeter 5.4.3
Programming Language	Python 3.10.4

4 Design Specification

4.1 Experimental Setup

The setup required for the experimentation of this study is highlighted in Table 2. Two 't2.large' AWS EC2 instances were provisioned and used. Each consisting of 2vCPUs, 8GB RAM and 16GB SSD volume storage. Interestingly, Ubuntu Server 18.04 LTS emerged as the most suited version for building and running the Apache serverless framework. More recent iterations of the operating system begin to conflict with the installer methods in OpenWhisk. After encountering multiple failed installations on later versions, Ubuntu Server 18.04 LTS was chosen. OpenWhisk operates similar to other iterations of serverless platforms on offer by cloud service providers. The fundamental components consist of the container technology Docker, real-time streaming service Kafka, web server Nginx, Consul networking, and NoSQL document database CouchDB [20]. While this project utilises local instances of this software, the web serverless framework - IBM Bluemix - is based on top of OpenWhisk. The Docker Compose plugin (version 1.21.2) was the preferring installation method, and the core software prerequisites for this approach include Docker and Python and Java.

4.2 Architecture Diagram

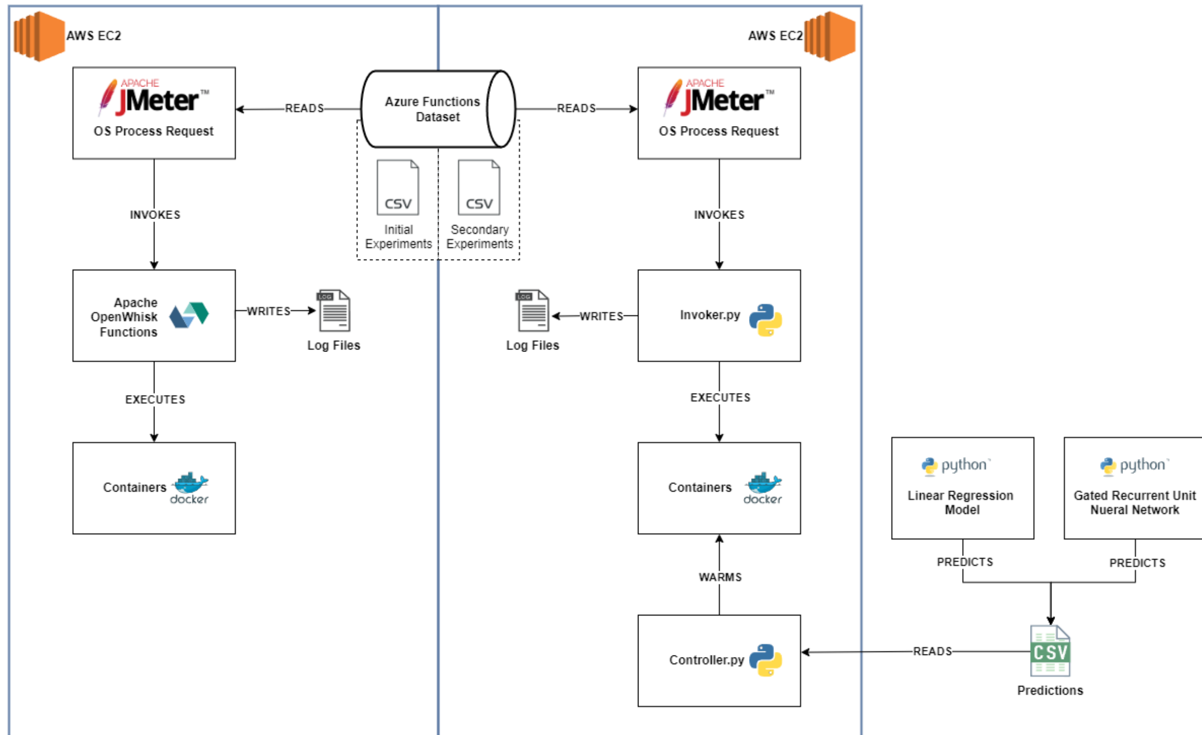

Figure 2: Project Architecture Diagram

4.3 Algorithms

Algorithm 1 *Custom Controller*

Input machine learning cold start predictions

Output warm docker containers over $\sum_{i=1}^{n}$ prediction time intervals

1: **function** DOES_CONTAINER_EXIST
2: query docker for warm containers
3: **return** *True or False*
4: **end function**
5: **function** CREATE_CONTAINER
6: create a warm docker container
7: **end function**
8: **for** each row $i \in N$ predictions **do**
9: **if** Does_Container_Exist = False **then**
10: Create_Container
11: **end if**
12: pause for i seconds
13: **end for**

Algorithm 2 *Custom Invoker*

Input $function_f$ request from Jmeter test plan
Output Log entry with metadata and time metrics for n *function* requests

```
1:  function DOES_CONTAINER_EXIST
2:      query docker for warm containers
3:      return True or False
4:  end function
5:  function CREATE_CONTAINER
6:      create a warm docker container
7:  end function
8:  function EXECUTE_FUNCTION
9:      run function_f in container_n
10:     terminate container_n
11: end function
12: if Does_Container_Exist = True then
13:     get name of existing warm container → container_n
14:     Execute_Function
15: else
16:     Create_Container
17:     Execute_Function
18: end if
19: write metrics to log file
```

Further description of the algorithms outlined above is documented in Section 5.2.

5 Implementation

This studies architecture is based on the Linux operating system only. Shell scripts automate the installation of all software related packages and dependencies such as Docker, Docker Compose, Apache OpenWhisk, Python, Java, JMeter and more. The method used for interacting with the local instances of the OpenWhisk stack requires the 'wsk' is stored on the VMs as an executable. Additionally, the 'apihost' and 'authorisation' properties were configured through said scripts. The default authorisation key for the standalone OpenWhisk stack was used during implementation. Also, the development and use of machine learning techniques was conducted outside of the described infrastructure. Google Colab notebooks [21] were used to develop each of the machine learning experiments which will be thoroughly detailed in Section 6.1. The predictions of these notebooks were dropped onto the implementation VMs for the main cold start validation experiments. A key future goal is to integrate these machine learning operations with the whole solution and consume the serverless logs as the training data.

5.1 Apache Jmeter

The Jmeter test plans were developed with user parameters stored and used to mirror the distribution and throughput of the Azure dataset. Initially, a start time parameter is created which contains the timestamp of the exact instant the test plan enters a new row of the dataset. This start timestamp remains consistent until the $n\text{th} + 1$ row is entered,

where the new timestamp overwrites this variable. The OS process sampler element facilitated the execution of serverless functions throughout the lifecycle of the test plan. Since the Azure dataset requests are broken into apps and functions, the test plan can retrieve the $(func) variable from row n as the workload runs. Each individual function ID from the dataset has its own related OpenWhisk action pre-defined, which allows the serverless request to be invoked and processed. The initial tranche of experiments were configured to utilise the 'wsk' client to execute their corresponding OpenWhisk actions for the $(func) value of row n. This concept also remained true for the second tranche of experiments, however, instead of using the 'wsk' client to run jobs inside of OpenWhisk, a custom invoker module is executed with parameters defining the specific function to run. Meanwhile, a flow control action element is in place which pauses the Jmeter load for x milliseconds. x is the delay time calculated between requests during data preparation in Section 3.2. The thread pause action will exit when one of two conditions is met, where t = current timestamp, st = start timestamp, x = delay time, and e = serverless execution time.

(a) $t = st + x \qquad \therefore x < e$

(b) $t = (st - e) + x \qquad \therefore x >= e$

Instances occurring in condition (a) will return and begin the next line of the test plan, row $n+1$ in thread $n+1$, since the current thread is being occupied by row n. Once execution has complete, a log entry is submitted by the results tree listener for observation after the test plan finishes. Where condition (b) is met, the delay time is greater than execution time. The flow control action pauses the thread for an additional $x - e$ milliseconds before the listener submits a log to the results tree and the subsequent line of the test plan begins. While Jmeter workloads are often developed through a GUI, the execution of these test plans were imposed with the '-n' parameter in the command line to execute in non-GUI mode.

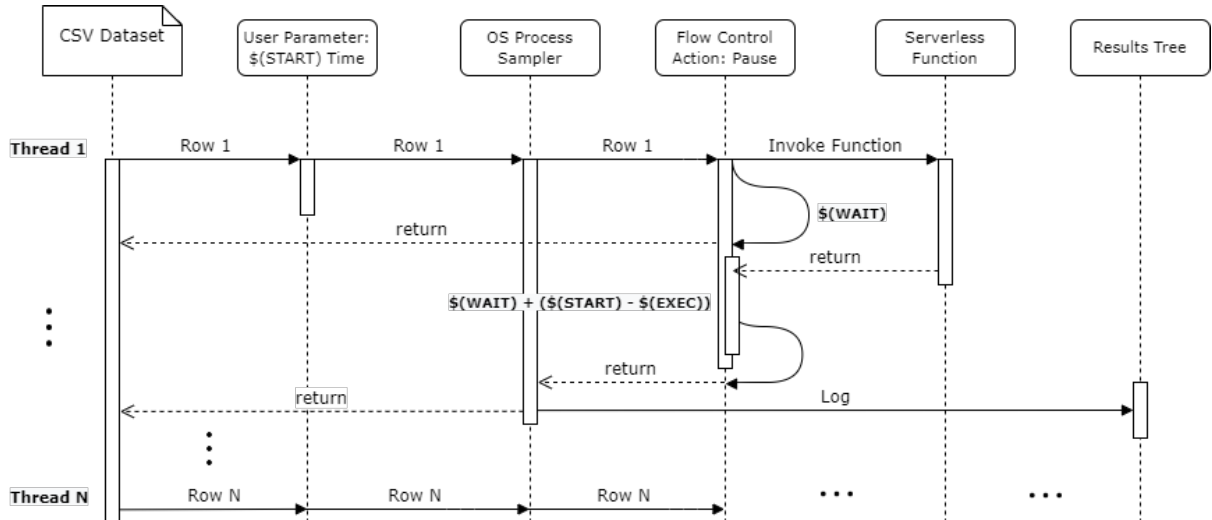

Figure 3: JMeter test plan UML illustration

5.2 Custom Modules

Extending the technology stack in OpenWhisk to warm containers is outside of the scope of this research. Therefore, custom modules were developed in the Python programming language as an alternative to the serverless platform to mimic this behaviour.

- Controller.py 1: runs on the server and iterates through the machine learning prediction values in order to create Docker containers at the identified set intervals.
- Invoker.py 2: function execution requests are routed to this module which identifies any warm containers created by the controller. If none exist, the invoker can create a container and execute the function itself, however, a penalty of 3 seconds is applied before the container is created and this request is logged as a cold start. It was observed that OpenWhisk averaged circa an additional 3 seconds during cold starts, with the custom module taking this into account.

Its important to note the distinction between functions executed directly through OpenWhisk and the custom invoker. Apache OpenWhisk has container lifecycle logic which the controller does not. Again this does not fall under the remit set out in this study. Therefore, containers post execution in OpenWhisk remain in a warm state for up to 10 minutes. This favours the latency of responses against the custom invoker which terminates each container prior to function execution. This research welcomes this disadvantage because it provides a clearer indication of success or failure in the machine learning predictions. During implementation the controller process was started with the nohup Linux command to ensure it runs in the background while the Jmeter test plans burden the invoker with simulated real-world requests.

6 Evaluation

The evaluation section of this paper is separated into two parts. The first part (Section 6.1 describes the controlled experiments that were taken while developing the machine learning models which required trial and error for adjusting many components such as parameters and epochs etc. The second part (Section 6.2) outlines the main findings of this paper by observing the cold start experiments from OpenWhisk against the documented solution from this study.

6.1 Controlled Experiments

6.1.1 Linear Regression

Predicting function executions with the least squared method during implementation appeared from the offset to offer extremely poor success. The residual sum of squares was an immense 2.5979e+18. Coupled with a similarly poor mean absolute error of 1.61 billion, it was instantly clear that accuracy was improbable. This is not surprising behaviour though, and understanding that timestamps are in themselves extremely large, exponential growth in error is not unforeseen for this reason. The exact timestamp prediction was not feasible with this algorithm, instead it was evaluated that shifting observations to the inter-arrival time between requests was the most appropriate, which drastically enhanced its performance.

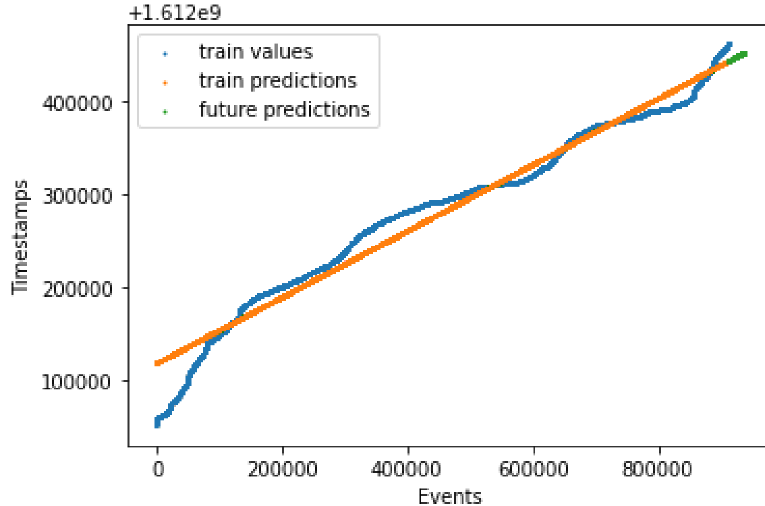

Figure 4: Example Linear trend from train to test predictions using Regression

6.1.2 Gated Recurrent Unit (RNN)

Experimentation of the window range referenced in Section 3.3.2 concluded a size of 20 proved a healthy balance between over and under-fitting the data. The input data batch for the GRU network suited a sequential model with 64 neurons. Each neuron recurrently processes the inputs and updates the hidden state that informs all cells in the network. The default activation function is hyperbolic tangent (tanh), however, this configuration resulted in greater loss than expected. The Rectified Linear Units (ReLu) activation function was applied due to its superior learning compared to sigmoid, and tanh. ReLu is less computationally expensive because it involves simpler mathematical operations while also avoiding the vanishing gradient problem neural networks have continued to combat, $max(0, x)$ [22].

Natural back propagation of the neural network influences the predictions from historic values. Incorporating future values by investigating model results with bidirectional traversal showed decreasing levels of precision. Unlike use cases such as natural language processing where possible forward variables could and should impact predictions, context of assumed future states appeared to negatively impair the models performance. A dense layer was also added to the model to deeply connect each neuron in the network. An extension to stochastic gradient descent optimization called Adam was applied to improve the weights between neurons. The learning rate of this algorithm was adjusted frequently in a bid to reduce loss. The optimal rate found was 0.001 which allowed the optimizer to learn in a steady manner, instead of rushing this learning with a greater fixed value [23].

Similar to the approach taken through regression, the loss function during experimentation of this model was the mean squared error (MSE). Determining the number of epochs to run was an iterative exercise. A save best only checkpoint was applied during model compilation which restored the best performing model per epoch based on the loss function of the overall fitting. Typically 10 or more cycles were performed. However, upon evaluation the best model was repeatedly found within the first 5 epochs. The final model included a total of 14,545 hyper-parameters with an average 94.32 (ms) MSE. The accuracy in predictions followed the distribution trends well, however, it struggled to identify

spikes in time intervals. The justification for proceeding with GRU over LSTM was validated during comparisons between the two. LSTM totaled 17,425 hyper-parameters with an average 94.83 (ms) MSE, and clocked fitting times up to 20% greater than GRU.

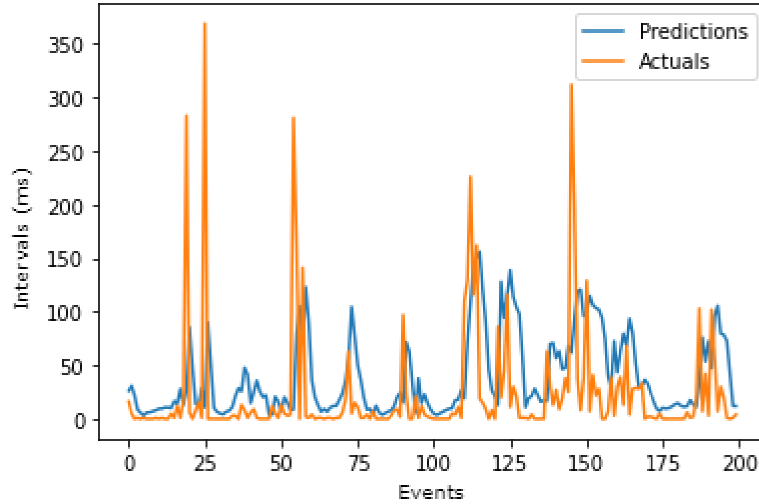

Figure 5: GRU interval predictions vs. actuals

6.2 Main Experiments

The main experiments during this study were separated into two phases. The initial or first phase had two randomly selected days worth of data from the Azure dataset. Jmeter test plans were triggered which simulated the real-world serverless invocations against actions in OpenWhisk. All actions executed the same Python program which returned a random prime number. The secondary phase also triggered Jmeter test plans against functions that returned random prime numbers, however, the invocation was passed through a custom module instead of OpenWhisk. This phase also shortened the window of execution to 6 hours. The custom module tracked the performance of the machine learning implementations. For the purposes of readability, this section will merge both initial and secondary phases into two experiments. Experiment 1 will detail the Apache OpenWhisk performance for data from Thursday 4th February 2021, and the performance of the linear regression algorithm predictions on that same day between 12:00 and 18:00. Similarly, Experiment 2 will detail the OpenWhisk performance for data from Friday 12th February 2021, and the performance of the gated recurrent unit predictions on the same day between 12:00 and 18:00.

6.2.1 Experiment 1

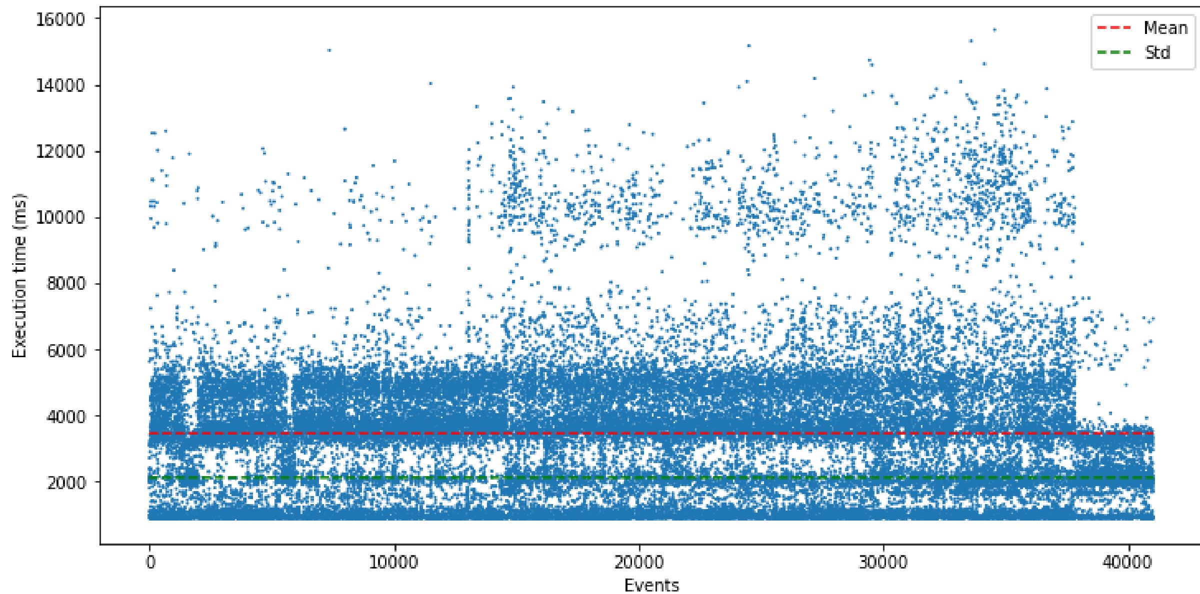

Figure 6: Phase 1 - Scatter plot of Jmeter test plan results (04/02/2021) executed through Apache OpenWhisk

Analysing Figure 6 the OpenWhisk run had difficulties with response times in hours of expected frequent activity. The traffic started and continued at relatively high to moderate rates until roughly 09:00, when operations went swiftly uphill. Significant cold start occurrences and worrying levels of latency was recorded during the daylight hours. Unsurprisingly at 22:00 cold starts and execution levels in general dropped rapidly with close to all requests being processed inside of total mean range leading into a Friday where total execution count can expect to be lower. The mean value was recorded at 3452(ms) with a standard deviation of 2120(ms) for the 41,023 functions executed.

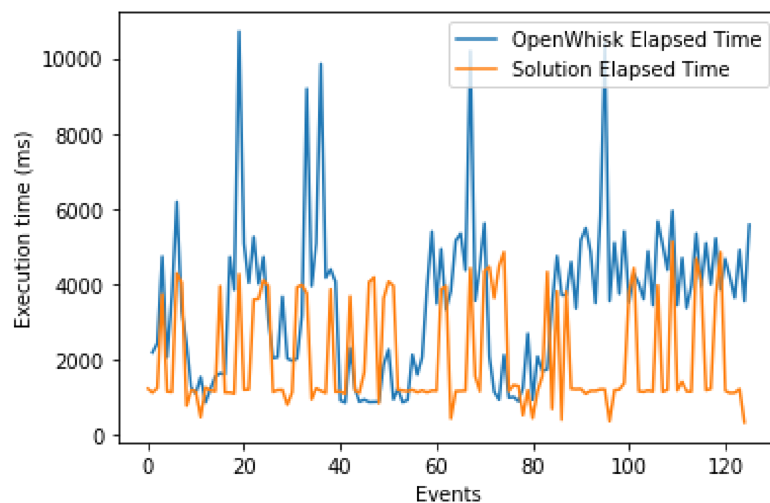

Figure 7: Phase 2 - Line graph of OpenWhisk experiment elapsed times compared to the proposed solution

Figure 7 visualises the results of the first 150 function executions through the custom invoker starting at 12:00. Additionally, the corresponding OpenWhisk results are graphed for comparison. The linear regression algorithm showed impressive levels of performance overall. It managed to drive the average elapsed response time down from 2711(ms) in OpenWhisk to 2134(ms), a decrease in latency of up to 21%.

6.2.2 Experiment 2

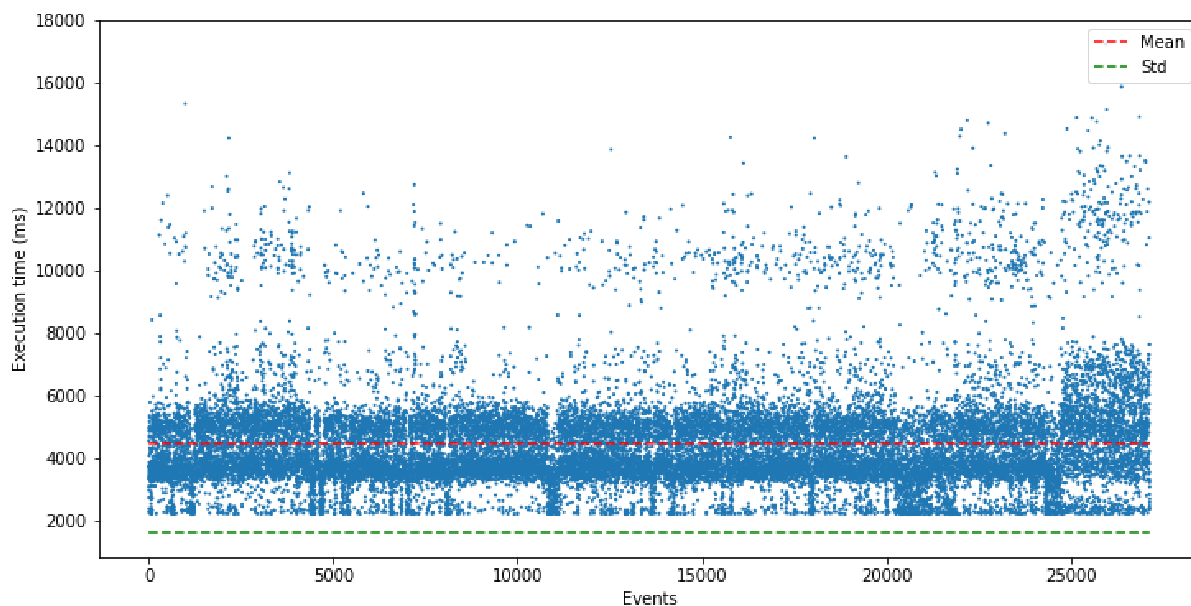

Figure 8: Phase 2 - Scatter plot of Jmeter test plan results (12/02/2021) executed through Apache OpenWhisk

Figure 8 shows interesting results compared to Figure 6 which appeared to follow standard diurnal patterns. In stark contrast, this OpenWhisk run had a consistent stream of requests and response time patterns rarely fluctuated. Most surprisingly, cold start and latency levels spiked at 22:00 instead of declining as expected. This behaviour continued into the early hours of a Saturday where total execution count can expect to be the lowest of the whole week. The mean value was recorded at a higher 4474(ms), with a lesser standard deviation of 1628(ms) for the 27,143 functions executed. Although a decrease in roughly 15,000 executions, average response latency was larger than previously seen in Experiment 1. The standard deviation was lesser in this run however, meaning the data was more closely aligned to the mean with less variance.

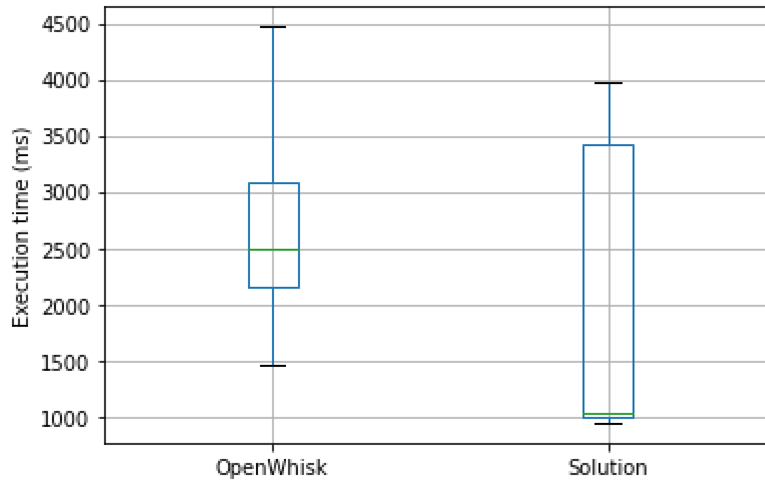

Figure 9: Phase 2 - Boxplot of OpenWhisk experiment elapsed times compared to the proposed solution

Figure 9 shows a boxplot of the results from the function executions through the custom invoker and OpenWhisk starting at 12:00. The upper and lower whiskers from both boxplots are very close in values. The solutions median is extremely close to both the lower whisker and first quartile, which suggests variance is poor around these percentiles. This is proven further by the size of the 75th percentile which shows huge variance. The length between the first and third quartile is indicative of data that clusters hevaily around each whisker value. The resulting median is caused as a majority of the values range closer to the first quartile. Its clear to observe the penalty applied to cold starts in the custom modules replicated OpenWhisk behaviour, however, the response times between cold and warm executions had little fluctuation, resulting in similar processing times within both categories. The OpenWhisk boxplot shows levels of variance not found in the solutions implementation. However, one striking property is the comparison between medians. It is evident from this figure that the requests were processed and handled at greater speed with the help of pre-heated containers from the GRU predictions. An overall decrease in latency of up to 28% over standard OpenWhisk executions. The total number of cold start occurrences was 967 as illustrated in Figure 10.

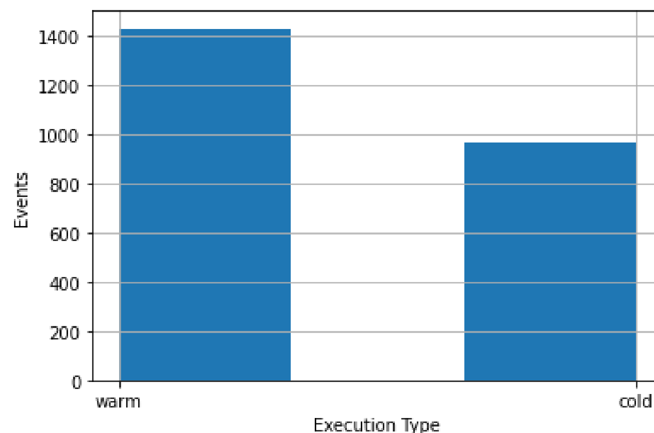

Figure 10: Phase 2 - Histogram of proposed solution cold start metrics with GRU

6.3 Discussion

Initial discussions focus on the results delivered by the GRU neural network. Although satisfying metrics were recorded, the number of cold starts clocked is an area that draws attention. Revisiting and potentially adjusting more parameters could continue to drive this count in a downward trajectory. However, it's important to highlight the aggressive intent of this research to begin with. The implementation was specifically designed to be unforgiving of miscalculations by the model. The custom controller queries whether any containers for incoming functions are available. If this is true, no action is applied and the GRU prediction is essentially rendered void. Additionally, containers are immediately terminated subsequent to function execution. This approach demands high levels of precision from the model as this skipped interval will undoubtedly result in implications during the experimental timeframe. Apache OpenWhisk on the other hand keeps containers alive with a grace period of roughly 10 minutes post processing. In terms of cold starts, this slightly favours the serverless framework in this particular use case, which benefits the outcomes of the experiment results. Removing the logic which prevents container creation for each prediction would in theory drive cold start numbers down rapidly. This approach however, poses cost and resource concerns due to expected memory footprint increases on the server even during idle periods. With additional resources being consumed on the machine, function executions could be affected due to the servers compute being constrained. Also, speculation over the peculiar trend in distribution observed by the OpenWhisk run on 12th February 2021 could have potentially contributed to the number of cold start instances during this test.

Before discussing other areas of this study, there was justice in opting for the GRU network over the LSTM based model. However, throughout development it was observed that mean squared errors continued to decline in the later epochs of LSTM compared to GRU where the error stagnated and even increased as the iterations went on. This behaviour may indicate that for larger datasets with this type of distribution, LSTM could emerge the optimal model of the two [22]. This hypothesis requires further experiments though with larger datasets, which would come in abundance had this solution been capable of consuming the outputted logs. This area is a key point noted in the future work section of this paper.

Linear Regression was also used to predict function executions. This algorithm is a core staple in statistical analysis and was used a baseline for measuring GRU model performance. While it under performed compared to the model, it surprisingly achieved levels of success that was not anticipated from the offset. Lastly, Jmeter proved a crucial tool in validating the research objectives described in the beginning of this study and managed to simulate the distribution of Azure Functions invocations extremely well.

7 Conclusion and Future Work

The serverless paradigm allows powerful systems to be created with consumers only being billed for what they use. These platforms are becoming increasingly popular and indications suggest trends will continue in this direction. This paper addresses the issue of latency in serverless computing. Enterprise systems have little to no margin or tolerance for system latency. Therefore, the cold start problem can transpire into a real-world blocker, forcing users to pursue their development endeavours elsewhere. The aim of this paper was to formulate a cold start mitigation strategy, with machine learning technologies. The architecture and design of this research project utilised cloud tools and frameworks and validated the implemented solutions performance against the Apache OpenWhisk serverless framework using a real-world dataset supplied by Microsoft. This study differs from other related work with its comparison of two similar recurrent neural networks and load testing of actual serverless executions.

Future work would focus heavily on productizing this solution. Deploying the implemented deep learning model on the servers and scheduling its execution against the collected logs would be a desired solution. The one-to-one relationship between containers and incoming requests has been identified in related work, exploring sandboxing within containers for superior resource allocation and efficiency is of keen interest in making the system lighter in weight [24]. Additionally improving upon and exploring other machine learning models would be carried out. Collecting and observing compute and memory metrics on the servers would be useful for understanding any resource implications for warming containers in the manner proposed by this solution. Lastly, facilitating the ability of this solution on other operating systems and execution of serverless functions in many other languages would be relevant.

To conclude, its evident that leveraging machine learning techniques can contribute to cold start reduction. In particular deep learning RNNs have great ability in extracting patterns from time series sequences. This study favoured GRU throughout the experimental process but recurrent neural networks overall are proficient for the tasks discussed in this research paper.

References

[1] A. Fuerst and P. 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*, pp. 386–400, ACM.

[2] Verified Market Research, "Serverless architecture market size, share, opportunities & forecast." `https://www.verifiedmarketresearch.com/product/serverless-architecture-market/`.

[3] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing,"

[4] K. E. ArunKumar, D. V. Kalaga, C. Mohan Sai Kumar, M. Kawaji, and T. M. Brenza, "Comparative analysis of gated recurrent units (GRU), long short-term memory (LSTM) cells, autoregressive integrated moving average (ARIMA), seasonal autoregressive integrated moving average (SARIMA) for forecasting COVID-19 trends," vol. 61, no. 10, pp. 7585–7603.

[5] S. Wu, Z. Tao, H. Fan, Z. Huang, X. Zhang, H. Jin, C. Yu, and C. Cao, "Container lifecycle-aware scheduling for serverless computing," vol. 52, no. 2, pp. 337–352. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.3016.

[6] G. De Palma, S. Giallorenzo, J. Mauro, and G. Zavattaro, "Allocation priority policies for serverless function-execution scheduling optimisation," in *Service-Oriented Computing* (E. Kafeza, B. Benatallah, F. Martinelli, H. Hacid, A. Bouguettaya, and H. Motahari, eds.), vol. 12571, pp. 416–430, Springer International Publishing. Series Title: Lecture Notes in Computer Science.

[7] C. L. Abad, E. F. Boza, and E. van Eyk, "Package-aware scheduling of FaaS functions," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pp. 101–106, ACM.

[8] B. C. Ghosh, S. K. Addya, N. B. Somy, S. B. Nath, S. Chakraborty, and S. K. Ghosh, "Caching techniques to improve latency in serverless architectures,"

[9] E. Oakes, L. Yang, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Pipsqueak: Lean lambdas with large libraries," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pp. 395–400, IEEE.

[10] P.-M. Lin and A. Glikson, "Mitigating cold starts in serverless platforms: A pool-based approach,"

[11] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, V. Sukhomlinov, and N. Nayak, "Agile cold starts for scalable serverless," p. 6.

[12] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in *Proceedings of the 21st International Middleware Conference*, pp. 1–13, ACM.

[13] Z. Xu, H. Zhang, X. Geng, Q. Wu, and H. Ma, "Adaptive function launching acceleration in serverless computing platforms," in *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 9–16. ISSN: 1521-9097.

[14] C. Zhang, M. Yu, W. Wang, and F. Yan, "MArk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving," p. 15.

[15] S. K. Govindan, "A deep learning based framework to initialize new containers and reduce cold start latency in serverless platforms," p. 28.

[16] A. Banaei and M. Sharifi, "ETAS: predictive scheduling of functions on worker nodes of apache OpenWhisk platform,"

[17] A. Zafeiropoulos, E. Fotopoulou, N. Filinis, and S. Papavassiliou, "Reinforcement learning-assisted autoscaling mechanisms for serverless computing platforms | elsevier enhanced reader."

[18] Azure, "Azure public dataset." `https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsInvocationTrace2021.md`.

[19] Azure, "Azure public dataset license." `https://github.com/Azure/AzurePublicDataset/blob/master/LICENSE`.

[20] K. Djemame, M. Parker, and D. Datsev, "Open-source serverless architectures: an evaluation of apache OpenWhisk," in *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pp. 329–335.

[21] Google, "Google colaboratory." `https://colab.research.google.com/?utm_source=scs-index`.

[22] A. Saeed, C. Li, Z. Gan, Y. Xie, and F. Liu, "A simple approach for short-term wind speed interval prediction based on independently recurrent neural networks and error probability distribution," vol. 238, p. 122012.

[23] H. Salem, A. E. Kabeel, E. M. S. El-Said, and O. M. Elzeki, "Predictive modelling for solar power-driven hybrid desalination system using artificial neural network regression with adam optimization," vol. 522, p. 115411.

[24] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards high-performance serverless computing," p. 14.