# Mitigating Serverless Cold Start Latency with Predictive Function Invocation and Adaptive Caching

MSc Research Project
Cloud Computing

Hariharan Sathiyamoorthy
Student ID: 23201550

School of Computing
National College of Ireland

Supervisor: Ahmed Makki

**National College of Ireland**
**Project Submission Sheet**
**School of Computing**

| | |
|---|---|
| **Student Name:** | Hariharan Sathiyamoorthy |
| **Student ID:** | 23201550 |
| **Programme:** | Cloud Computing |
| **Year:** | 2024 |
| **Module:** | MSc Research Project |
| **Supervisor:** | Ahmed Makki |
| **Submission Due Date:** | 12/12/2024 |
| **Project Title:** | Mitigating Serverless Cold Start Latency with Predictive Function Invocation and Adaptive Caching |
| **Word Count:** | 7617 |
| **Page Count:** | 22 |

| | |
|---|---|
| **Signature:** | Hariharan Sathiyamoorthy |
| **Date:** | 11th December 2024 |

# Mitigating Serverless Cold Start Latency with Predictive Function Invocation and Adaptive Caching

Hariharan Sathiyamoorthy
23201550

**Abstract**

The serverless industry is estimated to be worth over $21.9 billion in 2024 due to its separation of infrastructure management from application, and at the same time, it provides scalability and immense cost reduction. There are few problems when it comes to serverless that is cold start latency, which hinders the performance. The purpose of the study is to create a hybrid approach by combining predictive modeling to predict the function invocation along with adaptive caching to reduce the cold start frequency and latency. The study involved setting up the OpenWhisk serverless platform on a cloud machine, simulating real-world serverless behavior, using machine learning techniques to predict the function invocation time, creating a robust cache algorithm, and combining these two by creating a framework called SmartFaaS, followed by comparing it with the vanilla OpenWhisk solution for benchmarking. The key finding from the study was that SmartFaaS consistently used 30% of resource throughout all the experiments and the overall execution time were maintained under 200 ms compared to OpenWhisk 1200 ms showing an approximate 83% reduction in latency. By Using the predictive model in SmartFaaS only 37% of cold starts were recorded showing efficient reduction in cold start frequency. From these results, SmartFaaS offers a more effective and cost-efficient solution for mitigating cold start issues in serverless computing. The research contributes to the current state of the art by demonstrating the effectiveness of combining predictive modeling with adaptive caching. In practice, this means developers and cloud service providers can achieve better performance and resource utilization.However, the limitations of this study were that a single node application was tested, and as of now, only Node.js applications can utilize SmartFaaS. Future work could involve testing SmartFaaS with diverse workloads and exploring other machine learning models.

# Contents

# 1 Introduction

Recently, considerable literature has grown around the theme of mitigating cold starts in serverless computing due to its performance hindrance, which makes it unattractive to cloud providers and developers. At the same time, serverless computing provides significant benefits such as isolating infrastructure management from application development, scalability, and cost management. Serverless computing is a combination of Function-as-a-Service (FaaS) and Backend-as-a-Service (BaaS) Jonas et al. (2019). These two paradigms rely on each other to make serverless work efficiently. FaaS divides the application into multiple microservices, which are combined to form a function, while BaaS handles all the storage needs for result execution. An example of function execution in serverless computing is shown in Figure 1.

This research addresses the cold start issue by providing a hybrid approach to reduce cold start latency as well as the number of cold starts. It combines predicting invocation times beforehand—thus reducing the number of cold starts—with caching libraries onto the containers and pre-warming them, which further decreases the frequency of cold starts.

## 1.1 Motivation

The absence of pre warmed container during function invocation can introduce cold start latency, this will significantly impact the performance and user experience. These cold start will result in high operational overheads and utilize more resources, this will be a concern when there are unpredictable workloads. Mitigating cold starts is critical for ensuring the scalability of serverless applications. As the demand for serverless computing grows, the ability to handle many concurrent requests efficiently becomes increasingly important. By reducing cold start latency, serverless platforms can provide a more responsive and scalable environment for applications, leading to better performance and user satisfaction. The demand for serverless computing growing exponentially, According to Advisory (2024) the serverless market have registered a healthy CAGR of over 23.17% by 2029. This emphasizes the need to mitigate the cold start issue on serverless platform.

## 1.2 Research Question

> *"Can implementing a predictive modeling technique to determine function invocation on a serverless platform, together with adaptive cache management, result in reduced cold start frequency and latency for Developers and Cloud service providers?"*

Comparatively, fewer studies have analyzed hybrid approaches to reduce cold start frequency and latency. Mitigating cold starts is usually carried out using four mechanisms: cache-based, application-based, snapshot-based, and prediction-based. Several studies have focused on specific mechanisms and provided valuable insights. The common approaches to reduce the number of cold starts include container-based strategies, which are categorized into container pre-warming, pools of warm containers, container scheduling, and keeping containers alive. For instance, Liu et al. (2023) identified key factors influencing cold starts by implementing a solution called FaaSlight, which significantly reduced cold start frequency by optimizing code size and loading times. Similarly, Oakes et al. (2018) analyzed the installation time of commonly used libraries and developed a plugin called SOCK, which significantly reduced installation times and, as a result, latency. Each approach has its own merits and drawbacks. This study combines two mechanisms—prediction-based and cache-based—to reduce cold start latency and frequency by warming containers using predictive modeling techniques and preloading libraries into the containers to minimize latency. The concept of containerization is the backbone that orchestrates all necessary actions in an isolated environment.These containers are highly efficient and adaptable, enabling them to run independently on any given operating system. However, scheduling and orchestrating containers can be challenging at times. Several technologies have emerged in recent years, such as Kubernetes, which has gained significant popularity. The leading serverless providers in the industry are Google Cloud Functions, Azure Functions, and AWS Lambda.

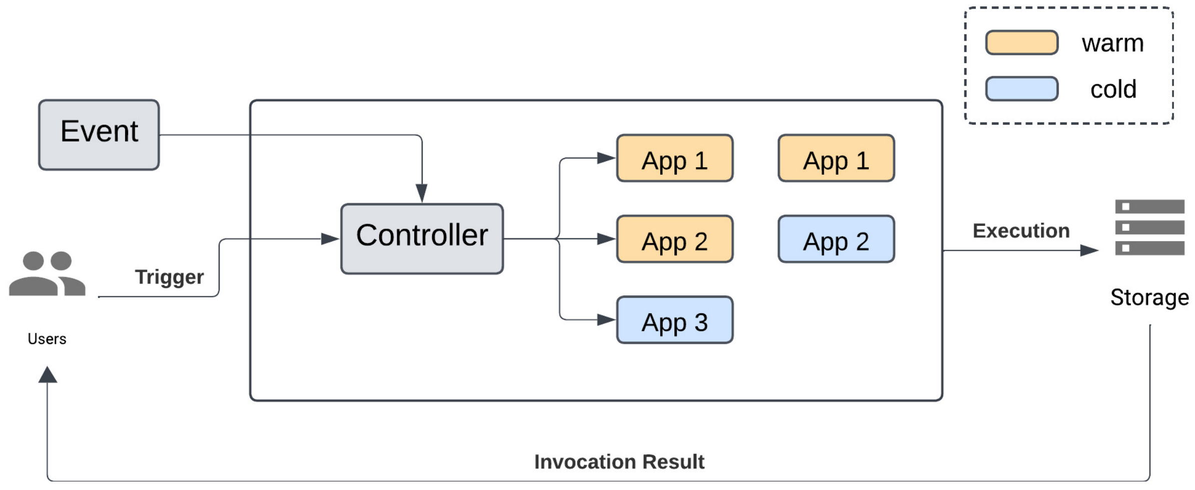


Figure 1: Example of Function execution in serverless computing

## 1.3 Research Objective

This study's short term goals focusing on developing the framework and simulating it with real world dataset using load testing tool and evaluating the framework with open source serverless software. The long term goals for this study are to integrate the framework with commercial serverless platform such as AWS Lambda, Google Cloud functions. Furthermore, future work could explore diverse workloads and runtime environments. This research main focal point is as follows:

- Set up an open-source serverless platform on virtual machines in the cloud.
- Simulate real-world serverless behavior using official Microsoft Azure Functions dataset and a load testing tool.
- Use machine learning techniques to analyze traces and forecast function execution patterns.
- Creating an adaptive cache management algorithm for frequently used libraries.

- Integrating adaptive caching with the predictive model to create a framework (SmartFaaS) that reduces cold starts
- Analyzing the outcomes in comparison to the vanilla open whisk solution

## 1.4 Report Structure

The rest of this paper is structured as follows. Literature review of related work in this topic. Sections include methodology, design specifications, implementation details, experiment evaluation, conclusion, future work discussion, and bibliography.

# 2 Related Work

To date, several studies have been carried out to mitigate the cold start issue and serverless computing overall. There have been four categories when it comes to mitigating cold starts: those are cache-based, snapshot-based, application-based, and prediction-based. These four categories have provided major improvements to reduce the cold start frequency and latency overall. Initiating a new container takes time, and upon invocation, if there is no existing container present, creating a new container will delay the process and label it as a cold start. Serverless computing needs to run at a higher level to execute complex business logic. This literature review will explore major contributions to mitigate cold start.

## 2.1 Cache Based Mitigations

The functions that need to be executed should be placed in the container initializing the required libraries, which leads to increased latency. A qualitative study by Daw et al. (2020) provided a Xanadu tool that mitigates the cascading cold start by providing resources at the right time; at the same time, it also supports explicit and implicit workflow specification options. This resulted in reduced platform overhead compared to native, and it also did perform well in a short workflow. It reduces the cascading cold start at the same time, providing minimal cost implication. This approach sometimes leads to resource overhead and low workflow invocation rates.

Detailed examination of cold start by Solaiman and Adnan (2020) introducing WLEC. It's a container management architecture to reduce the cold start. They have provided a way that modified the S2LRU structure, which is S2LRU++. By adding a queue, which will enhance container management. They evaluated this with OpenLambda and AWS local machine environments using six different metrics and a real-world image rendering application. This method showed 50% reduced memory consumption and reduced cold starts. This method will introduce added complexity in container management and unpredictable function invocation patterns. A significant analysis and discussion on the subject was presented by Oakes et al. (2018). They managed to identify the bottlenecks in container primitives, specifically in storage and network isolation. They have figured out that adding most-used Python libraries will add latency to the startup time, so they have introduced a container system called SOCK, which is optimized for serverless workloads, avoiding kernel bottlenecks and achieving a noticeable speedup over Docker environments. This methodology incorporates zygotes-based provisioning, which significantly improves the speedup times behind the scenes. It uses a three-tier caching strategy based on zygotes. They also did an image processing case study; it has shown reduced platform overheads compared to a conventional setup. Like WLEC, adding advanced optimization techniques may introduce increased complexity and become unpredictable on complex workloads. Another qualitative study by Fuerst and Sharma (2021) described how keeping functions alive after execution will reduce the cold start overhead. They have implemented a keep-alive strategy that is combined with caching, resulting in significant latency reduction. They have implemented concepts like reuse distances and hit-ratio curves. The main focal point of their study is the greedy dual keep-alive policy, which reduces overhead by 3 times compared to conventional setups. They have managed to implement this on the OpenWhisk platform. It has shown reducing requirements for dynamic, complex serverless workloads noticeably.

All these papers have given significant improvement for reducing cold start latency; for instance, Xanadu mitigates cold start by inferring function dependency models and allocating resources at the right time, and SOCK optimizes container performance by addressing kernel bottlenecks. Utilizing hibernated containers also resulted in a significant performance increase. This study focuses on caching npm packages for the application beforehand and preloading them for reduced latency. Along with adaptive caching, this study involves predictive modeling, which will further reduce the cold start latency to a great extent. Unlike the above papers that primarily focus on container management or resource allocation, this study includes machine learning and provides a hybrid approach that addresses the limitations of existing solutions and provides a more comprehensive strategy for reducing the cold start latency.

## 2.2 Prediction Based Mitigation

Recently, predicting invocation time prediction has gathered the vast majority of attraction due to its improved performance. In the background, this approach uses container-based strategies, which are basically four categories.

- Container Pre warming
- Container scheduling
- Pool of warm containers
- Keep alive containers

Container pre-warming can effectively reduce cold starts by loading necessary code onto the container, whereas the container scheduling predicts and provisions the resources beforehand, which will significantly improve latency. Maintaining warm containers in memory also enables warm starts. This section will discuss the wide variety of container-based techniques.

A qualitative study carried out by Xu et al. (2019) emphasizes that creating an adaptive warm-up strategy that predicts the function invocation times using the chain model and fine-grained regression algorithm along with the Adaptive Container Pool Scaling Strategy (ACPSS) will automatically align the size of the container to its needs; this will reduce the wastage of resources. These strategies show huge significance in reducing cold start latency and numbers. This methodology excels at all scenarios but may be less accurate in complex serverless workloads, thus leading to suboptimal performance. A significant analysis and discussion on the subject was presented by Vahidinia et al.

(2023). They have implemented a two-layer approach that uses predictive models to mitigate the cold start. The layer 1 has the reinforcement learning technique, which gives better suggestions upon creating containers, and the other layer uses the LSTM predictive approach that determines the number of required containers to pre-warm. This methodology shows major improvements in memory reduction and increased pre-warmed container execution. This approach's effectiveness depends upon the LSTM learning models and reinforcement learning models, which may lead to highly unpredictable workloads that cause inefficiencies that will predict the function invocation arrival before arrival and be less efficient when comparing with other machine learning algorithms for time series predictions. Their policy is not only helpful in mitigating cold start but also helpful in other problems such as provisioning virtual software. Again, this study also heavily relies on the TCN predictions, which may lead to highly volatile or unpredictable workloads. A detailed examination of FaaS by Sethi et al. (2023) showed that predicting the least recently used by machine learning methodologies can improve cold start latency. They also combine the affinity-based scheduling with this to increase the life span of the warm containers. The LCS approach showed huge performance improvements compared to the MRU (most recently used container) selection approach in evaluation. The memory usage with usage will be more when it's compared to the other approaches, making it less resource efficient. A recent study by Nguyen (2024) has proposed a new ensemble policy for reducing cold starts by low-coupling high-cohesion strategies unlike the regular policies. Behind the scene, the policy uses the temporal conventional network (TCN).

A significant analysis and discussion on the subject was presented by Li et al. (2021), emphasizing that borrowing other functions from other actions that share similar packages. Their pagurus system provided significant improvement to mitigate the cold start issue by advanced container scheduling, which shares the container across the actions into two variations, which are the inter-action scheduler and the intra-action scheduler. This will efficiently manage the container lifecycle. This methodology showed a massive performance boost of 10 ms even without warm container creation. This makes this model stand out from others, but the pagurus specifically relies on similarity between the actions to share between the containers; this will hinder performance when it's introduced to complex workloads. The study carried out by Roy et al. (2022) has implemented a novel technique called icebreaker, which basically reduces the service time and keeps alive costs by a heterogeneous system. They also implemented a dynamic node allocation policy that selects the most appropriate node for function invocation, which significantly reduces the operational overhead and noticeably reduces costs. By using heterogeneous nodes, the icebreaker can increase the number of warm functions within the same cost budget. This study also heavily relies on predicting the invocation probability to manage the node heterogeneity, which leads to performance bottlenecks on certain complex workloads.

Finally, predictive warming has provided a huge performance boost to serverless computing. Techniques like container pre-warming and predictive resource provisioning have provided great benefits, and LCS (least recently used) strategies combined with affinity-based scheduling extend the warm container lifespan. The icebreaker technique provides a heterogeneous approach to mitigate the cold start issue. All these studies are heavily relying on predictive modeling algorithms alone; this study aims to achieve the function invocation prediction along with adaptive caching. By integrating two mechanisms, this study can proactively warm the containers and preload the necessary packages, which will address the cold start latency effectively rather than using predictive modeling alone. This hybrid approach not only improves the performance but also improves resource utilization by removing the container immediately after execution. Furthermore, the adaptive caching helps in maintaining a high cache hit rate, further minimizing the latency associated with cold starts. This solution addresses the gaps in existing solutions and offers a robust framework for reducing the cold start.

## 2.3 Comparison Table

| References | Technique | ML | Caching | Runtime Awareness |
|---|---|---|---|---|
| Fuerst and Sharma (2021) | Greedy-Dual Caching to Mitigate cold start |  | ✓ |  |
| Oakes et al. (2018) | Zygote based Provisioning |  | ✓ | ✓ |
| Daw et al. (2020) | Speculative Provisioning |  | ✓ |  |
| Solaiman and Adnan (2020) | Introduced WLEC container management architecture |  | ✓ | ✓ |
| Xu et al. (2019) | Adaptive Warm Up Strategy | ✓ |  |  |
| Vahidinia et al. (2023) | Reinforcement Learning | ✓ |  |  |
| Roy et al. (2022) | Heterogeneous Node Allocation | ✓ |  | ✓ |
| Nguyen (2024) | Ensemble policy for reducing cold starts using TCN | ✓ |  |  |
| Li et al. (2021) | Function borrowing from other conatiner sharing same libraries | ✓ |  |  |
| **This Study** | Predictive Modeling and Adaptive Caching | ✓ | ✓ | ✓ |

Table 1: Comparison of literatures with this study

# 3 Methodology

## 3.1 Action Plan

The SmartFasS consists of two main steps, which are caching the frequently used libraries and predicting the function invocation time to pre-warm the containers and reduce the cold start latency and frequency in serverless computing. Preloading the libraries into the container to minimize the overhead with function initialization. Using a machine learning model trained on the official Azure dataset Azure (2021). This dataset provides the key insights for understanding the function invocation predictions and optimizing the prewarming strategies.

The framework uses the Docker containerization technology to execute the node app. To monitor the performance, two servers were created.

- Server 1: Runs Apache OpenWhisk on a Kubernetes cluster (using KinD)
- Server 2: Runs the SmartFasS framework.

A simple Node.js app was created using the popular package `Lodash`. This app is executed on server 1 to analyze the behavior of cold and warm starts on the OpenWhisk platform. The same app also executed on server 2, which has the SmartFasS framework in it to monitor the performance. On both of the servers, custom JMeter scripts were created and simulated real-world serverless patterns using the Azure function trace dataset. These scripts generated traffic patterns that reflect real-world serverless invocation scenarios. A thorough evaluation of the framework's effectiveness compared to the baseline environment. The final evaluation highlights the significant importance of SmartFass on the serverless platform while at the same time emphasizing its ability to substantially reduce cold start latency and maintain the cold start resource efficiency.

## 3.2 Data collection and preparation

| Field Name | Description |
|---|---|
| `app` | Identifier for the serverless application |
| `func` | Identifier for the serverless function, unique within each application |
| `end timestamp` | Timestamp (in seconds) marking the completion of the function execution |
| `duration` | Total execution duration (in seconds) |

Table 2: Dataset Field Descriptions

To predict the invocation times accurately, some sort of real-world data is essential. The predictive model must be trained on that data to get desirable results. For this study, a dataset from Azure function trace Azure (2021) was utilized; this data is publicly available for use. This dataset contains serverless invocation patterns. And it has the Creative Commons Attribution 4.0 License by Microsoft Azure.This dataset has function invocation log traces for a two-week period starting from January 31, 2021. Along with this dataset, the SmartFasS has generated log files from which were incorporated, capturing function invocation patterns during the evaluation phase. Importantly, this study doesn't involve any personal or sensitive data. The dataset was modified for this study's predictive model. The dataset field descriptions are shown in Table 2.

Standard data processing techniques such as KDD were used to clean and analyze the data. The distribution of data doesn't follow any specific pattern; it is positive and continuous but exhibits dependency due to daily and weekly patterns. The data is also highly variable with a coefficient more than 1. To calculate The `funcStart_time` is calculated as the `Initial_time` plus the difference between the `end_time` and the `duration` of the function execution, represented as

$$\text{funcStart\_time} = \text{Initial\_time} + \Delta\text{t}(\text{end\_time} - \text{duration}) \tag{1}$$

JMeter requires a delay in milliseconds variable to calculate that variable difference between the consecutive timestamps that have been carried out. From this dataset, 10 days of data were chosen for training the model, and the remaining days were chosen randomly to carry out the simulation. Through this process, it ensures its suitability for predictive modeling and subsequent performance testing of the SmartFasS framework.

## 3.3 Caching libiraies

In this study, caching frequently used libraries and their time taken to install will optimize the cold start frequency to a greater extent. The strategy involves leveraging the `.npm` folder, which is mounted onto newly created containers to ensure that cached libraries are readily accessible. The SmartFasS first strips out all the packages that need to be installed and checks the local cache in the `.npm` folder. If the library is found in the cache, it will be installed using the command `npm install package name --offline`.

This ensures that the package is installed from the local cache in an offline mode, avoiding setup time. If the required library is not found in the `.npm` cache of the root machine, then it will install from the npm repository. The downloaded libraries are then cached locally for future use, and their install frequency is updated in Redis, ensuring that frequently used libraries are available in the cache. This approach reduces the cold start latency and potential overhead in serverless computing. According to Oakes et al. (2018), to efficiently cache the frequently used libraries, the $\mathtt{t_{setup}}$ needs to be calculated.

$$t_{\text{setup}} = \sum_{i=1}^{n} (t_{\text{download}} + t_{\text{install}}) \tag{2}$$

The SmartFasS calculates the $\mathtt{t_{setup}}$ through each and every execution and to all the libraries present in the script. Therefore, the SmartFasS will treat a library that is already in the cache as zero. However, comparing this to OpenWhisk necessitates a slightly different approach, as OpenWhisk relies on actions that require constant creation and invocation. OpenWhisk does not provide the actual time required to install the dependent libraries. The caching algorithm preloads the frequently used npm packages into the container before function invocation, this will reduce the time spent on loading libraries during execution. The algorithm employs an LRU strategy to manage the cache. Libraries that are least recently used are evicted first to make room for new ones, ensuring efficient use of cache space.

## 3.4 Predictive Model

Predicting the function invocation time can effectively reduce cold start numbers by pre-warming the containers efficiently and utilizing the cache algorithm. 10 days of data were selected from the Azure dataset and used to train the model. The logs generated from these function invocation events to further train the model. The continuous range of data prompts the implementation of linear regression. This foundational method will offer a comprehensive overview for predicting the invocation time. This method will provide timestamp predictions in the form of a straight line while also maintaining a consistent interval between executions. The linear regression will be a starting point for further depth analysis and provide a statistical benchmark. The linear regression may not perform well on new data or the newly generated logs. This paper also explores the recurrent neural network (RNN). First, the study explores the GRU (Gated Recurrent Unit); it has reduced complexity and can be helpful for training the model. GRU performs well when it comes to handling sequential data, they are specifiacly designed to do that making them better choice for time series predictions in serverless computing. They are computaionaly more efficient than other model that predicts time series such as Long short-term Memory (LSTM) because of the fewer parameters.GRUs perform well on short to medium-length sequences, which is often the case in serverless function invocation patterns.Additionally, the memory requirment is also very less when compared to other predictive models.These advantages make GRUs a suitable choice for the predictive modeling component of the SmartFaaS framework.The main idea behind this is to retain and process previously used data for better context in predictions.

$$\text{Inputs } X = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \end{pmatrix} \quad \text{Labels } y = \begin{pmatrix} 6 \\ 7 \\ 8 \end{pmatrix}$$

GRU is used to get the sequential pattern in data where predictions depend on previous values. The row of `X` represents a sliding window of 5 consecutive invocation times, and the `y` is the next invocation time. At each step it processes the input vector and updates the hidden state. The GRU has multiple different hidden layers and predicts the next step based on the window size. The optimal window size will provide higher performance gains. Adding cyclic time features for the day, hour, and minute properties of the timestamp will boost performance. These cyclic values will improve the accuracy of the time feature using sine and cosine transformation.

$$\text{Day of Week: } x_{\text{day}} = \sin\left(\frac{2\pi \cdot \text{day}}{7}\right), \cos\left(\frac{2\pi \cdot \text{day}}{7}\right) \tag{3}$$

These cyclic time features 3 were incorporated into the input matrix, enhancing prediction by adding the temporal patterns. The model's delay times as a wait period for container creation and initialization.

## 3.5 Real-world Simulation and Evaluation Metrics

The cache algorithm and the predictive model were combined into the Node.js framework to make it an end-to-end solution for mitigating cold start latency and frequency. The main validation criteria are comparing the solution to base OpenWhisk instances to reduce the cold start latency and frequency. The total execution time is analyzed with and without the SmartFasS framework. For simulating real-world serverless behavior, Apache JMeter is used along with the Azure Function Trace dataset. The test plan includes looping through the CSV dataset, and threads were created with respect to the rows from the dataset. From there, the SmartFasS framework is triggered on every iteration.

Custom flow control actions were designed to pause the iterations. All the experiments were set to run for a specific time frame to analyze the proper serverless workloads. This approach allowed for a controlled environment to assess how well SmartFasS handled varying workloads. Throughout the experiments, detailed log files were generated at each stage, serving as key evaluation metrics. These logs were analyzed to assess cold start patterns, latency improvements, and overall performance gains. The experiment setup also involves stress testing scenarios to evaluate SmartFasS under high-concurrency conditions. Furthermore, more resource utilization metrics were also analyzed to showcase the efficiency of the framework. The dynamic workload adaptability is tested during the course of the experiments, ensuring that the framework is providing a robust solution to mitigate cold start.

## 3.6 Tools and Platform

An overview of the primary platforms and tools utilized in this study is provided in Table 3.

,

| **Component** | **Details** |
|---|---|
| Virtual Machines | Google Cloud Platform (GCP) Compute Engine |
| Operating System | Ubuntu Server 22.04 LTS |
| Serverless Platform | Apache OpenWhisk (1.2.0) |
| Kubernetes Cluster | KinD (Kubernetes in Docker) for running OpenWhisk |
| Container Technology | Docker (27.3.1) |
| Cache Manager | Redis 6.0.16 |
| Machine Learning | Google Colab & Keras TensorFlow |
| Programming Languages | Node js 20.18 & Bash |
| Performance/Load Testing | Java (openjdk-11) & Apache JMeter 5.6.3 |
| Docker Image | node:20-alpine |

Table 3: System Configuration

# 4 Design Specification

## 4.1 Experimental Setup

The various tools and platforms for this project are shown in 3. Two instances of `e2-standard-2` from Google Cloud Platform with 8 GB RAM and 64 GB SSD volume storage. The OS for the experiment was Ubuntu Server 22.04 LTS, which provides long-term support and stability and security. For serverless computing, Apache OpenWhisk (1.2.0) was employed OpenWhisk (n.d.) in a Kubernetes KinD cluster for container orchestration and management within a Docker environment for better Isolation and Portability Zhao et al. (2024). For cache management, Redis (6.0.16) is used to store frequently accessed data, reducing latency and improving performance. For machine learning and training the model, Google Colab and Keras TensorFlow were utilized Google (n.d.), providing powerful tools for predictive analytics. The SmartFasS framework is created using Node.js and Bash. The load testing tool Apache JMeter is used to create a test plan for simulating real-world scenarios. The `node:20-alpine` docker image was used during container invocation.

## 4.2 Architecture Diagram

The Figure 2 shows the overall structure of SmartFasS and vanilla OpenWhisk setup.

## 4.3 Algorithms

The SmartFaaS consists of two main workflows (Figure 3), which are orchestration flow, which validates the runtime and checks for any warm container that is present; if not, it will create a container and install the required packages and pause the execution for certain milliseconds predicted from the machine learning model.

The Intiate workflows consist of checking for a warm container that is present; if there is any warm container present, it will execute the node app inside the container and create the log for evaluation. The Algorithm 1 explains how the framework checks for any warm container present inside the virtual machine. If none are present, it will create a new container and label it as cold. The Algorithm 2 explains how the cache management works; first, it strips out all the required packages from the simple app and

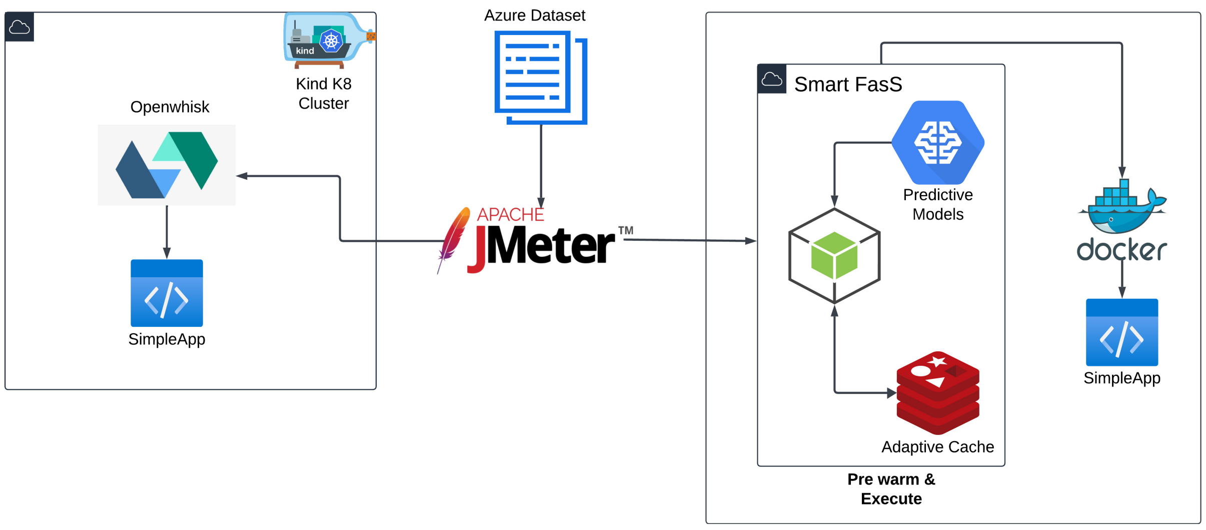


Figure 2: Project Architecture Diagram

checks whether it's present in the cache or not. If not present, it will freshly install from the npm registry and store them in the cache. The Algorithm 3 is the execution of a simple app inside the warm container; upon completion, it will remove the container from the virtual machine.These algorithms work together to preload and pre warm the container for execution. SmartFasS has logging functionality that logs every thing on to the container, two functions `readCSVfile` and `writeCSVfile` will handle all the logging inside SmartFaaS.

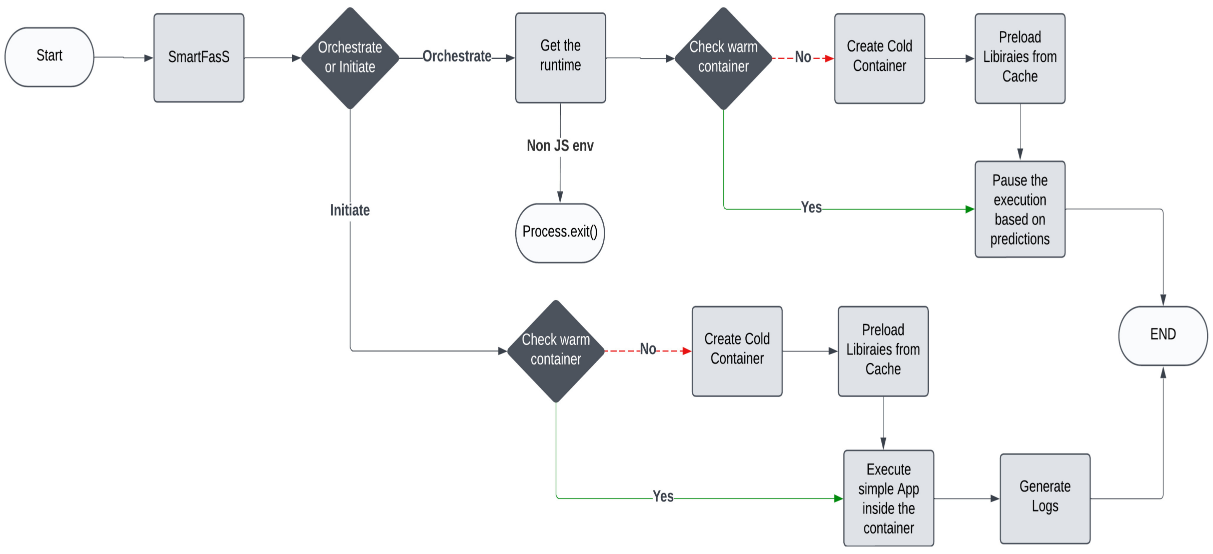


Figure 3: SmartFaaS Execution Flow

## 5 Implementation

To install all the required software, a custom shell script was created that will automate the process of installing all the dependencies (KinD, OpenWhisk, Node.js, and Docker)

**Algorithm 1** Check Warm Container

```
Input: runtime
Output: {executionType, containerName}
TRY
Execute a command to count the number of warm containers for the given runtime.
Parse the output to get the count of warm containers.
if there is at least one warm container then
    Set executionType to 'warm'.
    Execute a command to list the names of warm containers.
    Sort the container names.
    Select the first container name.
else
    Set executionType to 'cold'.
    Create a new cold container and get its name.
end if
Return the executionType and containerName.
CATCH(error)
Log the error.
Create a new cold container and get its name.
Return 'cold' as executionType and the new containerName.
ENDTRY
```

**Algorithm 2** Install Npm Package

```
Input: containerName, args
Output: Installation Status
TRY
Connect to the redis client.
Get the package names from the specified file.
Split the package names into an array.
for all npmPackage in the array do
    Check if the package is available in the npm registry.
    if the package is available then
        Check if the package is cached.
        if the package is cached then
            Install the package offline.
        else
            Install the package online.
        end if
    end if
    Store the package in the cache.
end for
Return 'success'.
CATCH(error)
Return.
FINALLY
Disconnect from the client.
ENDTRY
```

**Algorithm 3** Execute Container

```
1: Input: containerName, args, executionType
2: Output: {execOutput, executionTime}
3: TRY
4: Copy the specified file to the container.
5: Record the start time.
6: Execute the file inside the container.
7: Record the end time.
8: Kill the container.
9: Remove the container.
10: Return the execution output and execution time.
11: CATCH(error)
12: Return.
13: ENDTRY
```

Kubernetes SIGs (n.d.). The Helm package manager was used to manage the KinD cluster. The helm chart provided the OpenWhisk templates to install the necessary dependencies such as the controller, invoker, CouchDB, and Kafka. The linear regression and Gated Recurrent Unit (GRU) models were created and trained on the Google Colab platform along with TensorFlow. The SmartFasS is a node-based framework that has two workflows, orchestrate and initiate, to mitigate the cold start problem. The initiate function will be executed from Apache JMeter, and the orchestration will be along with the prediction output from the machine learning models. Both SmartFaaS and OpenWhisk were deployed on identical hardware setups and the same operating system to ensure a consistent environment and eliminate OS-related performance variations. The workloads for both the setups were identical; the same set of functions and invocation patterns were used to simulate real-world serverless behavior. On both setups, a load testing tool was used to create a test plan for simulating real-world scenarios, ensuring that both frameworks were subjected to the same load conditions. Both frameworks were equipped with monitoring and logging mechanisms to track performance metrics such as latency and resource utilization. This data was used to ensure a fair and objective comparison between SmartFaaS and OpenWhisk.

## 5.1 SmartFasS

The framework was created to mimic the behavior of serverless computing platforms like OpenWhisk. The execution environments in the SmartFasS are managed manually and carry out the tasks, such as checking if a warm container is present, caching, and preloading the packages into the container. In contrast, OpenWhisk will abstract away the infrastructure management and handle the provisioning by itself. That's the limitation in SmartFasS compared to OpenWhisk. The SmartFasS have been tested under heavy workloads that handle multiple concurrent executions seamlessly and can handle high concurrency.

The SmartFasS starting point is the `main.js`, where the main component gets initialized; from there, all the command line arguments get parsed, and depending upon the command line argument, it will execute either the orchestrate or initiate functions. The initiate function retrieves the runtime and checks whether it is a JS environment or not.

As of now, only this framework supports the JavaScript environment; in the near future it will incorporate all the other programming languages, and then it uses the custom functions like `checkWarmContainer`, `installNpmPackage`, and `executeContainer`. The orchestrate function will create the container and label them warm or cold depending upon the creation and install the required packages beforehand and make them readily available for the execution. The Initiate function will execute the simple app inside the pre-warmed container, and the output is logged for all the executions. For easier execution, all the scripts were added to the `package.json` file, and the available scripts are shown in Table 4.

| **Scripts** | **Description** |
|---|---|
| orchestrate | Runs the orchestration process using the main script and function.js |
| initiate | Initiates the process using the main script and function.js |
| coldMitigation | Runs the orchestration process and JMeter load script, logging outputs to respective log files |
| jmeter | Executes the JMeter load script and logs the output |
| OpenWhisk | Executes the JMeter script for OpenWhisk actions and logs the output |
| killall | Kills all Docker containers with the name ”coldMitigation” |
| rmall | Prunes all stopped Docker containers |

Table 4: SmartFasS Scripts

The logs from the framework have been stored in the `logs` folder, which will be used for evaluation and to compare the results with the OpenWhisk implementation. To reduce the project complexity, the OpenWhisk implementation, which was also added inside the smartFasS as a separate function, will get the compressed version of the simple Node app and execute it in the OpenWhisk environment using the `wsk` CLI tool.

| **Timestamp** | **Container Name** | **Execution Type** | **Duration (ms)** |
|---|---|---|---|
| 2024-11-16T12:43:21.151Z | coldMitigation_node_2bacb479-6971-47e0-a250-0f476154c5d2 | cold | 443.590 |
| 2024-11-16T12:43:51.348Z | coldMitigation_node_a3ae5893-62e8-4699-a780-4e05f2fae0fa | warm | 353.429 |
| 2024-11-16T12:44:08.577Z | coldMitigation_node_b87aa478-ea59-49a1-b780-bb66407c508c | warm | 286.520 |
| 2024-11-16T12:44:10.831Z | coldMitigation_node_f1c81320-28fd-4195-8a41-92b02f59aaca | warm | 352.808 |
| 2024-11-16T12:44:46.889Z | coldMitigation_node_5d3f2620-8efb-4f19-94d1-516b267962b0 | warm | 298.784 |
| 2024-11-16T12:46:00.912Z | coldMitigation_node_85b31d00-0227-4417-9166-cb791df88a82 | warm | 375.696 |

Table 5: Logs from SmartFasS

## 5.2 Model Training

For creating the machine learning models, the Google Colab platform was used. By using that, it has provided a powerful set of tools on the fly, which is really helpful for the study. First and foremost, this study explored the linear regression algorithm because of the continuous nature of timestamps. The usual train and test datasets were created, which are used for predicting the function invocation time. The result provided by the linear regression is quite interesting; the least-squares method showed very low performance for the dataset, and the residual sum and the mean absolute error were also pretty high. This may be caused by the length of the timestamp. Predicting an accurate timestamp might be overkill for this algorithm. This algorithm showed slightly better performance at changing the observations to inter-arrival times between requests. Then this study explored the neural networks because it is widely used for time invocation predictions. Without LSTM we cannot progress in neural networks; the LSTM algorithm was explored in the study to understand the predictions versus actual values, but this study didn't include it because GRU (Gated Neural Network) provided better results. Due to their specific nature, GRUs are better suited for time series predictions in serverless computing since they perform well while handling sequential data. Because they have fewer parameters, they are computationally more efficient than other time series prediction models like Long Short-Term Memory (LSTM).In serverless function invocation patterns, short to medium-length sequences are frequently handled successfully by GRUs.Furthermore, in comparison to other predictive models, the memory need is likewise quite low.GRUs are a good fit for the SmartFaaS framework's predictive modeling component because of these benefits In GRU this study has used a balanced window size of 20; this provided the stability between the underfitting and overfitting data. First the `tanh` activation function was used, and it resulted in poor performance; then the popular `Relu` activation function was used, which resulted in better computational efficiency. Also, the bidirectional traversal influenced the predictions. Furthermore, a dense layer was added to connect the neurons deeply; this resulted in a mean squared error (MSE) of 104.0309. The optimal model was found within the 5 epochs. The actual and predicted values from GRU are shown in Figure 4. Finally, the delay time calculated from the GRU and linear regression was considered for the final evaluation.

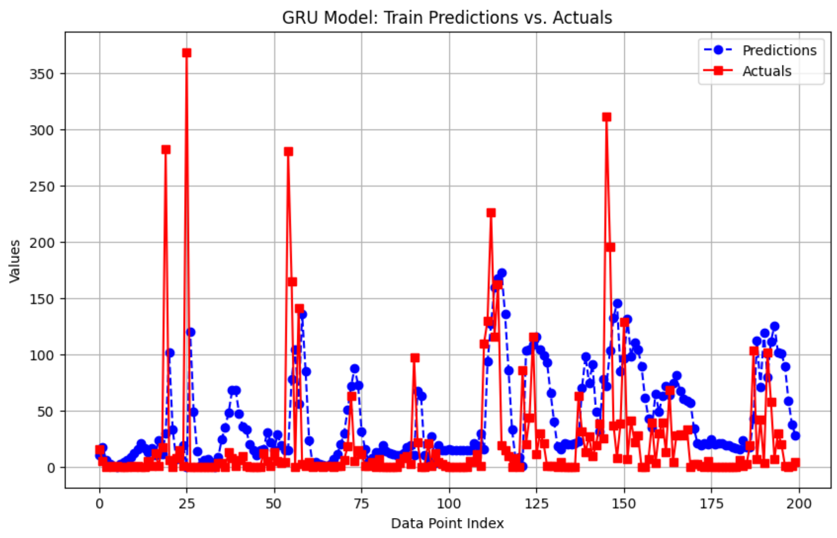


Figure 4: GRU Predictions

## 5.3 Apache JMeter

To create a real-world simulation of serverless computing, Apache JMeter was used along with the Azure function trace dataset, which has the function invocation trace of two weeks of data. This study encompasses a custom test script that will be running alongside the SmartFasS framework. The workflow of the test plan is shown in the figure. The test plan includes an OS system sampler, which will execute the SmartFasS function on each iteration, and the test plan has two flow control actions that will execute the iteration based on two conditions:

a) waitTime $<$ total execution time

b) waitTime $\geq$ total execution time

These conditions will decide the flow execution. if condition `a` is true, the thread will move on to the next line; on the other hand, if the condition `b` comes true, it will pause the execution for a certain period of time and resume the execution. The pause delay time is calculated by the `waitTime - total execution time` From this, the delay variable is set, and all the results from this are stored in a separate file under the `logs` folder. For OpenWhisk implementation, a separate test plan was created using the same logic;

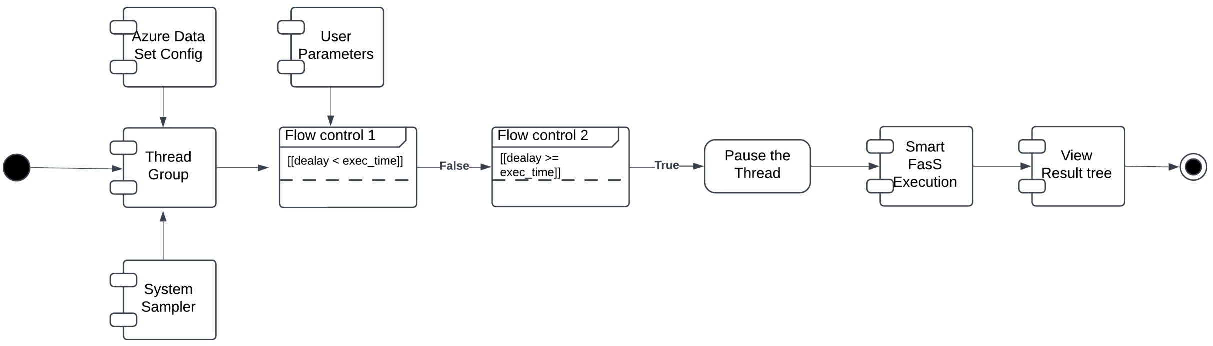


Figure 5: JMeter Test Plan

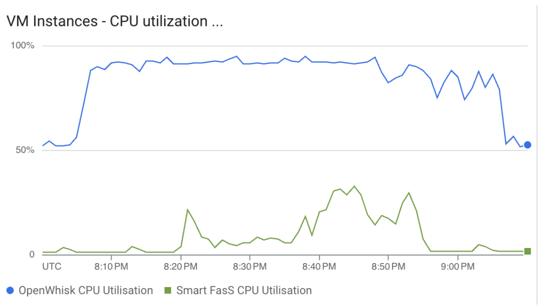
the only difference was in the OS sample. Instead of calling the SmartFasS function, the script will execute the `wsk` command that will execute the simple node app and store the results.

## 6 Evaluation

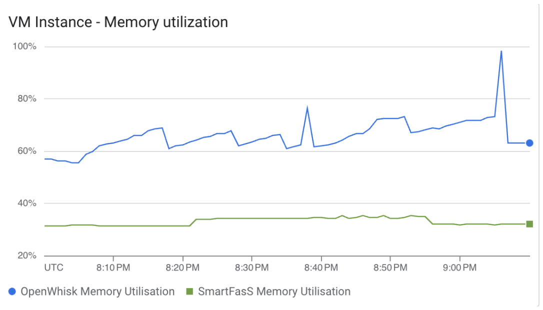
This section explains the in-depth analysis and evaluation done between smartFasS and OpenWhisk. Total, this study carried out three experiments that all have shown provided significant improvements over the vanilla OpenWhisk. In experiment 1, done to compare the CPU and memory utilization between the two servers, the main experiments were 2 & 3, which emphasize the importance of SmartFasS and the significance of caching and function invocation prediction. In experiment 2 demonstrate the need for robust cache algorithm to improve the overlall performance of the system and reduced data retrieval times, leading to faster function execution. Experiment 3 shows the effectiveness of function invocation prediction and importance of pre warming the conatiner beforehand, SmartFaaS was able to preload necessary resources, thereby minimizing latency and improving response times. The combined results of these experiments highlight the superior performance of SmartFaaS over OpenWhisk, particularly in terms of resource utilization, execution speed, and scalability.

### 6.1 Experiment 1

In this experiment, both systems were set to run in parallel to analyze the CPU, memory, and disk performance for both systems. On both servers the experiment was set to run for one hour, that is, 27/11/2024, 8 pm to 9 pm, by executing a node app with the resource-heavy package `lodash` performing a simple text manipulation. Figure 6(a) shows the CPU utilization between both the servers. Over the period of one hour, SmartFaaS utilized approximately 30% of resources, indicating handling tasks more efficiently, whereas



(a) CPU utilization OpenWhisk vs SmartFaaS



(b) Memory utilization OpenWhisk vs SmartFaaS

Figure 6: Resource utilization comparison between OpenWhisk and SmartFaaS

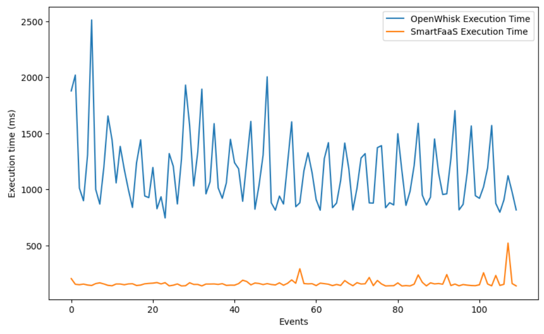
OpenWhisk consistently used higher resources, around 90%, indicating less adaptiveness during workload. This indicates that SmartFasS is the more cost-effective solution.

The Figure 6(b) shows the memory utilization between both the servers at the same period of time. The OpenWhisk server steadily maintained ( 60–80%), indicating higher resource allocation, but on the other hand, shows consistent memory utilization of ( 30%) throughout the testing. SmartFasS indicates resilience by reducing memory spikes, offering better resource utilization for production environments. These figures are obtained from the GCP monitoring dashboard for observing instance activity.

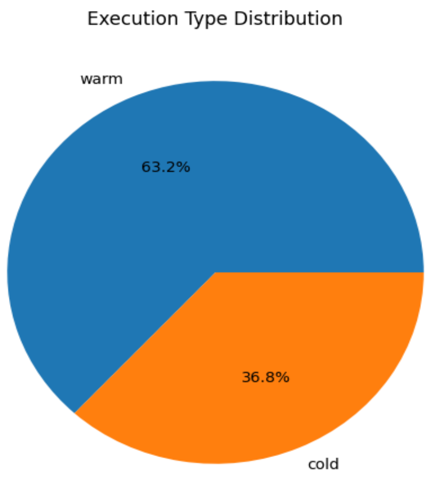
## 6.2 Experiment 2

In Experiment 2, the machine learning model delays were used for pre-warming the containers using SmartFaaS. This experiment uses the linear regression model for predicting the time of invocations. For comparison, the vanilla OpenWhisk setup was used. This experiment also uses the same node app for execution on both the servers. The dataset used for this experiment was a specific time frame from the Azure dataset, which is `2021-02-01 10:00 - 14:00`. The JMeter script will trigger the `initiate` function on the SmartFasS, and the `orchestrate` function uses delay times from the machine learning model. For the OpenWhisk, the Azure dataset was used to run the test script with the same flow control configurations. The logs from both executions were stored and analyzed in the Colab platform.

The logs have provided significant insights about the performance between the two servers. SmartFaaS consistently maintains a lower execution time that is under 500 ms. This occurs because of the container reuse and preloading the libraries into the container. On the other hand, OpenWhisk showed a higher execution time between 1000 ms and 2500 ms, which is pretty high when compared to SmartFaaS. One thing to keep in mind is that for OpenWhisk, for every iteration, it needs to create and invoke the action; this may influence the latency, but to make the comparison fair between them, this approach was implemented. The Figure 7(a) shows the average execution time between the SmartFaaS and OpenWhisk. The initiator logs from the SmartFasS show the distribution between the warm and cold starts held during the SmartFasS implementation; it shows over 63% of executions were warm starts from pre-warming the container beforehand Figure 7(b). This shows the efficient resource optimization over the vanilla OpenWhisk server.



(a) Execution time between OpenWhisk and SmartFaaS

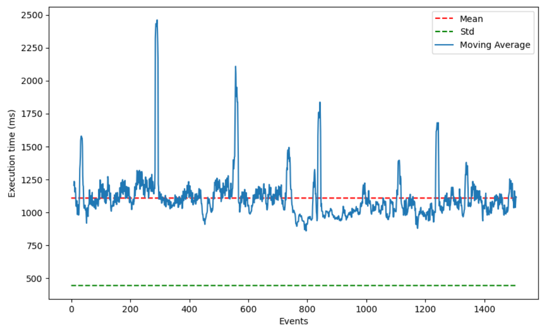


(b) SmartFaaS - Warm vs Cold Distribution

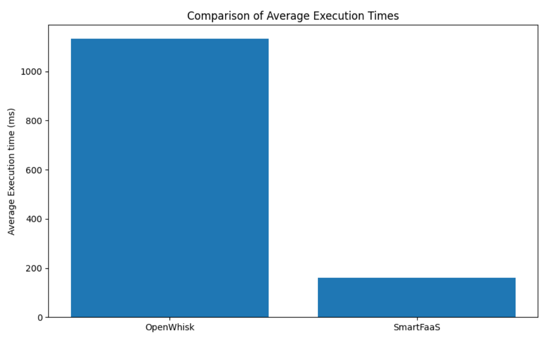
Figure 7: Experiment 2 Analysis

## 6.3 Experiment 3

To analyze the proposed solution more effectively, another day was chosen from the dataset, which is `2021-02-04 12-18`, and the GRU prediction delays were used to `orchestrate` the container by preloading the libraries and making it readily available for execution. The `lodash` node app was used here for the two servers.The JMeter script will execute the `initiate` function inside SmartFasS based on the flow control action. This experiment was carried out two times, first for 6 hours and then next for approximately 70 mins. For evaluation, the second simulation was taken, and the logs from both JMeter and SmartFaaS showed interesting results. The OpenWhisk execution logs from JMeter were evaluated; Figure 8(a) shows that the `X-axis` represents the number of events that were executed and the `Y-axis` shows the execution time of values between 500ms and 2500ms. The moving average was calculated; this helps to show the trends in latency without any sudden spikes. The mean for all the events is shown in the red line, and the standard deviation (std) in the green line indicates the variability or dispersion time around the mean.



(a) Phase 2 - JMeter OpenWhisk Execution logs



(b) Execution time between OpenWhisk and SmartFaaS

Figure 8: Experiment 3 Analysis

The Figure 8(b) shows the average execution. The SmartFasS execution time shows huge improvements compared to the OpenWhisk platform. SmartFasS maintained consistent execution time under 200 ms, emphasizing the benefits of preloading the libraries and prewarming the container, reducing the setup overhead. whereas the OpenWhisk server's average execution time was around 1200 ms, indicating a drastically slower performance and absence of preloading or prewarming the container. The performance gain of over 70% indicates that SmartFaaS is more efficient in handling diverse workloads under scenarios requiring high responsiveness.

## 6.4 Discussion

All the experiments have provided valuable insights on SmartFaaS efficiency. The Experiment 1 shows the comparison between the resource utilization, cost-effectiveness, and better adaptiveness from Figures 6(a) & 6(b). Still, the performance metric for that experiment was carried out for 1 hour, which may not be valid for longer-term performance metric calculation, and the only single application was tested throughout all experiments; increasing the workload may provide interesting results. In Experiment 2, we see the importance of cache and preloading the libraries into the container; SmartFaaS showed significant reduction in function execution times compared to OpenWhisk. Figure 7(b) and the huge performance improvements over OpenWhisk. The cache hits and misses weren't monitored, which may affect the real-world scenarios, and introducing diverse workloads may bring newer challenges. Furthermore, the OpenWhisk setup had to invoke and create every action for each iteration, which may influence cold start latency. The third experiment, which showed the importance of the function invocation prediction, SmartFaaS prediction accuracy provides better strategies for pre-warming the containers beforehand, reducing the cold start latency, and this allows SmartFaaS to scale more efficiently. Figure 8(b), as of now, SmartFaaS is being compared with OpenWhisk; comparing it with all other solutions may bring valuable insights to improve SmartFaaS. From the above experiments it shows that SmartFaaS reduces overhead from preloading libiraies at runtime which leads to faster execution times. SmartFaaS shows optimized resource allocation by dynamicaly allocating them based on the workload which reduces resource usage at idle times. The system is designed to scale efficiently with increasing workloads, maintaining performance levels even under high demand.The cache management techniques to ensure that frequently accessed data is readily available, reducing latency and improving performance. Finally, detailed monitoring and analytics, allowing for real-time performance tuning and identification of potential issues. Furthermore, using other machine learning models may bring greater performance improvements, such as classification algorithms instead of regression algorithms. The linear regression model showed greater performance in warm and cold start distributions when compared to GRU. The loading test tool helped to achieve the real-world simulation efficiently by defining a proper flow control plan that pauses the execution, which remains crucial for the study.

The limitation of this study is that the current implementation of smartFasS is limited to Node.js applications which restricts its applicabilty to other programming environments. The evaluation was conducted using a sinlge node.js application with `lodash` package, this may not accurately track the performance accross diverse workloads. SmartFaaS dosent monitor the cache hits and misses which are crucial for understanding the the effectiveness of the caching mechanism.

# 7 Conclusion and Future Work

Returning to the question posed at the beginning of this study 1.2, was addressed and significant contribution in reduction of cold start was made. The primary goal of the study is to develop a hybrid approach that combines predictive modeling to forecast function invocation with adaptive caching to lower cold start frequency and latency. This has been addressed. This study involves two servers, one running Apache OpenWhisk and the other running the SmartFaaS framework. A Node.js app was used to simulate serverless behavior, and custom JMeter scripts were created to generate serverless-like scenarios. The SmartFaaS framework utilized predictive modeling and adaptive caching to reduce cold start latency and frequency. The key finding from the study was that SmartFaaS consistently used 30% of resource throughout all the experiments and the overall execution time were maintained under 200 ms compared to OpenWhisk 1200 ms showing an approximate 83% reduction in latency. By Using the predictive model in SmartFaaS only 37% of cold starts were recorded showing efficient reduction in cold start frequency. Experiment 2 showed that SmartFaaS maintained execution times under 500 ms, while OpenWhisk had higher execution times between 1000 ms and 2500 ms. Experiment 3 further highlighted the benefits of preloading libraries and prewarming containers, with SmartFaaS maintaining execution times under 200 ms compared to OpenWhisk's 1200 ms. The number of warm starts was also significantly high in both experiments. This shows the importance of SmartFaaS for mitigating cold start latency.

The implications of this research are significant for developers and cloud service providers, as SmartFaaS offers a more efficient and cost-effective solution for mitigating cold start issues in serverless computing. This research is evident in the substantial performance improvements observed in the experiments. This study's shortcoming is that the current implementation of smartFasS is confined to Node.js apps, hence restricting its applicability to other programming environments. The assessment was performed utilizing a singular Node.js application with the `lodash` package, which may not reliably monitor performance across varied workloads. SmartFaaS does not monitor cache hits and misses, which are essential for assessing the efficacy of the caching mechanism. Future work could involve testing SmartFaaS with diverse workloads and comparing it with other serverless solutions to gain more insights. Adding runtime environments other than Node.js to get the performance insights. Exploring other machine learning models, such as classification algorithms, may also bring greater performance improvements. Additionally, integrating SmartFaaS with commercial serverless platforms like AWS Lambda or Google Cloud Functions could provide valuable feedback and potential for commercialization. With this intergrations the comercial software could adapt similar strategies like resource optimization, caching libiraies and pre warming technique. This also could reduce the overhead and improve signifiacant performance. A follow-up research project could focus on optimizing the predictive model and adaptive caching algorithm for different types of serverless applications, further enhancing the framework's efficiency and scalability.

## References

Advisory, M. I. R. . (2024). Serverless computing market size & share analysis - growth trends & forecasts (2024 - 2029). Retrieved December 6, 2024, from `https://www.mordorintelligence.com/industry-reports/serverless-computing-market`.

Azure (2021). Azure functions invocation trace 2021, `https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsInvocationTrace2021.md`. Accessed: 2023-10-10.

Bannon, R. (2022). *Leveraging machine learning to reduce cold start latency of containers in serverless computing*, Master's thesis, Dublin, National College of Ireland. Submitted.
**URL:** *https://norma.ncirl.ie/5971/*

Daw, N., Bellur, U. and Kulkarni, P. (2020). Xanadu: Mitigating cascading cold starts in serverless function chain deployments, *Proceedings of the 21st International Middleware Conference*, Middleware '20, Association for Computing Machinery, New York, NY, USA, p. 356–370.
**URL:** *https://doi.org/10.1145/3423211.3425690*

Fuerst, A. and Sharma, P. (2021). Faascache: keeping serverless computing alive with greedy-dual caching, *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, Association for Computing Machinery, New York, NY, USA, p. 386–400.
**URL:** *https://doi.org/10.1145/3445814.3446757*

Google (n.d.). Google Colaboratory, `https://colab.research.google.com/?utm_source=scs-index`.

Govindan, S. K. (2020). *A deep learning based framework to initialize new containers and reduce cold start latency in serverless platforms*, Master's thesis, Dublin, National College of Ireland. Submitted.
**URL:** *https://norma.ncirl.ie/4535/*

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

Kubernetes SIGs (n.d.). kind: Kubernetes IN Docker, `https://kind.sigs.k8s.io/`.

Li, Z., Chen, Q. and Guo, M. (2021). Pagurus: Eliminating cold startup in serverless computing with inter-action container sharing.
**URL:** *https://arxiv.org/abs/2108.11240*

Liu, X., Wen, J., Chen, Z., Li, D., Chen, J., Liu, Y., Wang, H. and Jin, X. (2023). Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing, *ACM Trans. Softw. Eng. Methodol.* **32**(5).
**URL:** *https://doi.org/10.1145/3585007*

Nguyen, T. n. (2024). Holistic cold-start management in serverless computing cloud with deep learning for time series, *Future Generation Computer Systems* **153**: 312–325.
**URL:** *http://dx.doi.org/10.1016/j.future.2023.12.011*

Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A. and Arpaci-Dusseau, R. (2018). SOCK: Rapid task provisioning with Serverless-Optimized containers, *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, USENIX Association, Boston, MA, pp. 57–70.
**URL:** *https://www.usenix.org/conference/atc18/presentation/oakes*

OpenWhisk (n.d.). OpenWhisk Documentation, `https://openwhisk.apache.org/documentation.html#openwhisk_architecture`.

Roy, R. B., Patel, T. and Tiwari, D. (2022). Icebreaker: warming serverless functions better with heterogeneity, *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, Association for Computing Machinery, New York, NY, USA, p. 753–767.
**URL:** *https://doi.org/10.1145/3503222.3507750*

Sethi, B., Addya, S. K. and Ghosh, S. K. (2023). Lcs: Alleviating total cold start latency in serverless applications with lru warm container approach, *Proceedings of the 24th International Conference on Distributed Computing and Networking*, ICDCN '23, Association for Computing Machinery, New York, NY, USA.
**URL:** *https://doi.org/10.1145/3571306.3571404*

Solaiman, K. and Adnan, M. A. (2020). Wlec: A not so cold architecture to mitigate cold start problem in serverless computing, *2020 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 144–153.

Vahidinia, P., Farahani, B. and Aliee, F. S. (2023). Mitigating cold start problem in serverless computing: A reinforcement learning approach, *IEEE Internet of Things Journal* **10**(5): 3917–3927.

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

Zhao, H., Pan, S., Cai, Z., Chen, X., Jin, L., Gao, H., Wan, S., Ma, R. and Guan, H. (2024). faashark: An end-to-end network traffic analysis system atop serverless computing, *IEEE Transactions on Network Science and Engineering* **11**(3): 2473–2484.