

# Evaluating Container-based and WebAssembly-based Serverless Platforms

by

Abdul Monum

A thesis  
presented to the University of Waterloo  
in fulfillment of the  
thesis requirement for the degree of  
Master of Mathematics  
in  
Computer Science

Waterloo, Ontario, Canada, 2024

© Abdul Monum 2024

## **Author's Declaration**

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

## Abstract

Serverless computing, often also referred to as Function-as-a-Service (FaaS), allows developers to write scalable event-driven applications while the cloud provider manages the burden of provisioning and maintaining compute resources. Serverless computing is enabled using virtualized sandboxes like containers or lightweight virtual machines that form the execution units for FaaS applications. However, applications suffer from expensive startup latency (cold starts) due to the compulsory overhead of creating a sandbox and initializing the application code and its dependencies. FaaS platforms keep function executors warm in memory to avoid this latency which incurs additional memory overhead on the system. Recently, WebAssembly (Wasm) has emerged as a promising alternative for FaaS applications with its lightweight sandboxing, providing negligible startup delays and reduced memory footprint. However, Wasm applications experience slower execution speeds compared to native execution. This thesis presents a performance evaluation of WebAssembly-based serverless computing in comparison with container-based serverless platforms using analytical performance models and its experimental evaluation. The performance model for container-based serverless platforms is used from existing literature, reflecting the behavior of commercial platforms like AWS Lambda, IBM Cloud Functions, and Azure Functions. For WebAssembly-based serverless platforms, this thesis proposes a new performance model based on queueing systems. These models are verified experimentally using open-source platforms: Apache OpenWhisk for containers and Spin for WebAssembly. A suite of representative serverless applications is used to validate the models. The comparison of the performance models with experimental results highlights the trade-offs between container-based and WebAssembly-based serverless platforms, providing insights into their respective efficiencies in handling serverless workloads.

## **Acknowledgements**

I want to thank all the people who made this thesis possible, especially my supervisor, Professor Martin Karsten, for his guidance and support throughout my study. I also want to acknowledge Professor Samer Al-Kiswany for his advice and assistance throughout the project and Rogers Communications for their support.

Finally, I thank my parents and friends for their love and unwavering encouragement.

## **Dedication**

This is dedicated to my friends and family.

# Table of Contents

<b>Author’s Declaration</b>	<b>ii</b>
<b>Abstract</b>	<b>iii</b>
<b>Acknowledgements</b>	<b>iv</b>
<b>Dedication</b>	<b>v</b>
<b>List of Figures</b>	<b>ix</b>
<b>List of Tables</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Contributions . . . . .	3
<b>2 Background and Related Work</b>	<b>4</b>
2.1 Container-based Serverless Platforms . . . . .	6
2.1.1 Auto Scaling Techniques . . . . .	6
2.1.2 The Cold Start Problem . . . . .	7
2.1.3 Function Keep-Alive . . . . .	8
2.2 WebAssembly-based Serverless Platforms . . . . .	9
2.2.1 WebAssembly Primer . . . . .	9

2.2.2	FaaS Use Case . . . . .	11
2.2.3	WebAssembly Limitations . . . . .	12
2.3	Comparing WebAssembly and Containers for the Serverless Use Case . . . . .	13
2.3.1	Serverless platforms supporting WebAssembly . . . . .	13
2.3.2	Performance Comparison of Container and WebAssembly . . . . .	14
2.4	Analytical Modelling . . . . .	15
<b>3</b>	<b>Methodology</b>	<b>17</b>
3.1	Scheduling Assumption . . . . .	17
3.2	Container-based Model . . . . .	17
3.2.1	Limitations of the model . . . . .	20
3.3	WebAssembly Model . . . . .	20
3.4	Serverless Platforms . . . . .	22
3.4.1	Apache OpenWhisk . . . . .	22
3.4.2	Spin . . . . .	24
3.5	Serverless Applications . . . . .	25
<b>4</b>	<b>Evaluation</b>	<b>27</b>
4.1	OpenWhisk Evaluation . . . . .	27
4.1.1	Experimental Setup . . . . .	28
4.1.2	Warm Request Routing in OpenWhisk . . . . .	30
4.1.3	Analytical Model Validation . . . . .	31
4.2	Spin Evaluation . . . . .	31
4.2.1	Experimental Setup . . . . .	31
4.2.2	Analytical Model Validation . . . . .	32
4.3	Experimental Results . . . . .	32
4.3.1	Keep-Alive Experiment . . . . .	33
4.3.2	Load Experiment . . . . .	41
4.4	Discussion and Future Work . . . . .	57

<b>5 Conclusion</b>	<b>59</b>
<b>References</b>	<b>60</b>
<b>APPENDICES</b>	<b>67</b>
<b>A Appendix</b>	<b>68</b>
A.1 Model Modification Effect . . . . .	68

# List of Figures

3.1	Overview of the SMP-based performance model for container executors in serverless platforms. . . . .	18
3.2	OpenWhisk architecture [6]. . . . .	23
4.1	Mean response times against keep-alive time. . . . .	34
4.2	Running executor counts against keep-alive time. . . . .	35
4.3	Running cold executor counts against keep-alive time. . . . .	36
4.4	Running warm executor counts against keep-alive time. . . . .	37
4.5	Idle executor counts against keep-alive time. . . . .	38
4.6	Total executor counts against keep-alive time. . . . .	39
4.7	Cold start probability against keep-alive time. . . . .	40
4.8	Mean response times against arrival rate. . . . .	42
4.9	Running warm executor counts against arrival rate. . . . .	43
4.10	Running cold executor counts against arrival rate. . . . .	44
4.11	Running executor counts against arrival rate. . . . .	45
4.12	Idle executor counts against arrival rate. . . . .	46
4.13	Total executor counts against arrival rate. . . . .	47
4.14	Cold start probability against arrival rate.. . . . .	48
4.15	Average response time of the <i>aes</i> function against arrival time. The left plot shows Wasm model predictions generated using a nominal response time. The right plot uses the average response time of the workload before the capacity is reached to generate the model predictions. . . . .	49

4.16	Number of running executors during the <i>aes</i> function workload against arrival time. The left plot shows Wasm model predictions generated using a nominal response time. The right plot uses the average response time of the workload before the capacity is reached to generate the model predictions.	50
4.17	Average response time of the <i>matmul</i> function against arrival time. The left plot shows Wasm model predictions generated using a nominal response time. The right plot uses the average response time of the workload before the capacity is reached to generate the model predictions. . . . .	51
4.18	Number of running executors during the <i>matmul</i> function workload against arrival time. The left plot shows Wasm model predictions generated using a nominal response time. The right plot uses the average response time of the workload before the capacity is reached to generate the model predictions.	52
4.19	Average response time of the <i>whatlang</i> function against arrival time. The left plot shows Wasm model predictions generated using a nominal response time. The right plot uses the average response time of the workload before the capacity is reached to generate the model predictions. . . . .	53
4.20	Number of running executors during the <i>whatlang</i> function workload against arrival time. The left plot shows Wasm model predictions generated using a nominal response time. The right plot uses the average response time of the workload before the capacity is reached to generate the model predictions.	54
4.21	Load experiment results on OpenWhisk using the <i>aes</i> function. . . . .	57
A.1	The modified performance model used in the thesis is closer to the experimental evaluation on OpenWhisk. . . . .	69

# List of Tables

3.1	Description of the serverless benchmarks. . . . .	26
4.1	Response times of workloads analyzed in this study. . . . .	33

# Chapter 1

## Introduction

Serverless computing, also known as Function-as-a-Service (FaaS) is a popular cloud computing paradigm that has seen immense adoption in production workloads in recent years. These workloads include various domains such as scientific computing, multimedia processing, machine learning, and back-end web applications [15]. The serverless model allows developers to focus on writing their applications as one or a combination of scalable event-driven functions. Developers do not need to manage servers and the associated deployment complexity to run their applications. Instead, the cloud provider handles the responsibility of infrastructure management and resource scaling. The necessary resources are dynamically allocated using ephemeral sandboxes, typically containers or lightweight virtual machines (VMs), when an event triggers a function (e.g. HTTP trigger). The sandboxes form the execution units for FaaS applications providing an isolated environment. Major cloud providers follow this model and offer serverless platforms such as AWS Lambda [63], Microsoft Azure Functions [8], and Google Cloud Run Functions [13].

However, these container-based FaaS platforms suffer from the widely studied problem of cold starts [65, 28, 45, 54]. Cold starts occur when the sandbox environment has to be prepared before the execution of the function. The cold start period includes the virtualization process of the sandbox, initializing the runtime environment, and loading all the application dependencies. The cold start latency is significant - typically on the same order of magnitude as the execution time of the function [65]. Serverless platforms circumvent cold starts by releasing the container resources after a threshold period of inactivity (also called the keep-alive time) [63, 8]. Containers are kept in memory during this period so that subsequent invocations of the function execute in the already initialized environment and skip the expensive startup latency. While the keep-alive time policy reduces the number of cold starts, it introduces another problem. Keeping containers

alive consumes computing resources on the physical servers, and increases the resource requirements for the FaaS platform [28]. Thus, the keep-alive time reduces the number of cold starts at the cost of reducing the overall efficiency of the physical resources. A recent study from Microsoft [65] shows that function executions are short and invoked infrequently leading to both a high number of cold starts and high periods of inactivity where containers are consuming resources.

In contrast, WebAssembly (Wasm) has recently emerged as a promising alternate technology that is proposed as better suited to the FaaS model [34, 67]. Wasm is a portable, low-level bytecode format that executes securely and efficiently in a stack machine. Wasm serves as a compilation target for applications written in any programming language. Wasm runs applications in a sandbox environment that is more lightweight than containers while providing the same isolation guarantees as containers provide for FaaS applications. The lightweight sandboxing of Wasm allows for negligible application startup delay and a lower resource footprint than containers [41, 57, 46]. These benefits of running applications in Wasm have given rise to a new set of serverless platforms that create a WebAssembly executor for each function invocation and immediately remove them after the function execution [71, 21, 14]. Thus, a serverless platform that uses Wasm as a function executor provides instant application execution with better overall resource utilization. However, WebAssembly’s compute performance is found to be slower than native execution [34, 37, 74, 68]. The delay in execution comes from the compilation process of Wasm to machine code and the isolation restrictions of the stack machine [74]. Therefore, a tradeoff exists between Wasm and containers as function executors in serverless platforms. Wasm offers better resource efficiency and lower startup costs, whereas containers have faster execution since they execute machine code but suffer from higher cold start latency and lower resource utilization.

Several works compare Wasm and containers as function executors in serverless platforms [41, 67, 57]. However, these studies only evaluate individual functions, focusing on their execution times and memory footprint instead of evaluating complete workloads and overall platform performance. Since these works, WebAssembly runtimes have rapidly evolved and claim near-native execution for many workloads [21, 71]. Programming languages are also adapting their libraries and features to execute in the Wasm environment, further bridging the gap between Wasm and native execution. Several commercial and open-source serverless platforms that use Wasm for function execution have emerged. Therefore, this study aims to compare the current state-of-the-art WebAssembly and container performance in serverless platforms.

## 1.1 Contributions

This thesis presents a performance evaluation of WebAssembly and containers as function executors in serverless platforms using analytical performance models and experimental validation. Accurate performance models help ensure quality of service by providing useful metrics that help optimize management for workloads for both developers and cloud providers. The performance model for containers is used from the literature, which closely follows commercial serverless platforms such as AWS Lambda, IBM Cloud Functions, and Azure Functions. The model for WebAssembly is proposed using queuing systems [55]. The performance models are then validated experimentally on open-source serverless platforms using a suite of representative serverless functions from various domains. Apache OpenWhisk [3] is used for the container-based platform, while Spin is used for the WebAssembly-based platform. This work also introduces a modification to the performance model of container function executors that provides a fairer prediction. Moreover, a small algorithmic change in OpenWhisk is also introduced that aligns better with the performance model. Finally, the thesis compares the performance models to the experimental results and identifies the tradeoffs between the two function executors.

# Chapter 2

## Background and Related Work

Serverless computing is a cloud computing execution model that allows developers to deploy their applications to the cloud without managing the burden of provisioning and maintaining compute resources. Hence, the term 'Serverless' comes from the lack of the need to manage servers for developers. Serverless Computing is predominantly implemented as 'Function-as-a-Service' (FaaS), where developers package their applications as event-driven, stateless functions, and the cloud provider manages the resources. Since serverless computing is often synonymized with FaaS, this thesis uses both terms interchangeably. Developers write application logic in modular pieces of code in the programming language of their choice, and the functions and their dependencies are bundled together. These FaaS applications can consist of a single or a combination of multiple functions. The application is then registered on the FaaS platform. After submitting the serverless functions to the FaaS platform, functions are triggered by events such as HTTP requests, timers, or a new data arrival in cloud storage [65]. The event trigger prompts the serverless platform to automatically provision and start dedicated function executors (such as VMs, containers, or Wasm executors) with the necessary resources (e.g., CPU and memory) to run the functions. The serverless platform takes care of scaling the number of executors depending on the workload and the auto-scaling strategy used by the platform. The key characteristics of serverless computing are summarized below:

**Event-driven:** Serverless functions are executed when events trigger them. These events can come from any source, including HTTP calls, timers, message queues, sensors, or updates to cloud storage. HTTP triggers are the most dominant events [66], which are mainly used for building backend application services. Timer events execute serverless functions at regular intervals. Queue-based events are triggered when new messages arrive in distributed messaging systems like Apache Kafka [5]. Binding functions with events

allows users to develop useful application functionality and provide the necessary dynamic context to the application when it executes.

**Pay-as-you-go billing:** The key aspect of Serverless Computing is that users are only billed for the resources used during function execution. There is no cost to the user when the function executors are idle.

**Auto-scaling:** The serverless platform dynamically allocates resources for serverless function execution according to the workload. This is done using sandboxes such as containers, lightweight virtual machines, or recently with Wasm runtimes [77, 10], which act as execution units for the FaaS applications. The platform uses different autoscaling techniques, which scale up these executors when there is high demand or scale them down when the workload traffic is low to maintain better system utilization.

**Isolation:** Serverless platforms host functions on the same hardware from multiple tenants. Thus, the platform must isolate the execution of functions to secure them from other tenants. The isolation is done using virtualization technologies such as VMs, containers, or Wasm runtimes. Virtual machines provide a computing environment with an operating system, CPU, memory, disk, and network storage using a partition of physical resources. Containers use Linux kernel features such as namespaces [52] and cgroups [51] to provide isolated access to system resources on the shared operating system. Wasm runtimes run applications compiled to the Wasm format that execute securely in a stack-based virtual machine [41].

**Performance and resource restrictions:** Serverless platforms place upper-bound limits on the CPU and memory usage of functions, the execution duration of the function, and the number of concurrent requests. This simplifies billing for FaaS platforms and avoids resource contention when running functions from multiple users [44].

**Programming language support:** Serverless platforms support writing FaaS applications in multiple programming languages, including C/C++, Python, Java, JavaScript, Ruby, Go, and Swift. This provides flexibility for developers to write FaaS applications in the programming language of their choice.

These characteristics of FaaS platforms have made them attractive for many application domains, such as scientific computing, image and video processing, machine learning inference, web application backends, and real-time analytics [64]. The FaaS workloads in these domains also have unique characteristics. Most serverless functions execute in less than a second, showing that function execution times are on the same order of magnitude as the function startup time [66]. Moreover, serverless functions are infrequently executed - 81% of applications are executed only once per minute [66]. Conversely, most function

executions are accounted for by a small subset of applications [66]. These characteristics have important implications for the architecture and design of serverless platforms.

## 2.1 Container-based Serverless Platforms

The key enabler of serverless computing is the lightweight virtualization environments that form the function execution units in the Serverless architecture. Major serverless platforms use lightweight VMs (Firecracker [26] used in AWS Lambda and gVisor [31] used in Google Cloud Run) or Linux containers (used in Azure Functions and IBM Cloud Functions) as function executors. The common characteristic that allows them to be used as function executors is that they can be built and regenerated easily compared to traditional VMs while providing the required isolation and performance guarantees for FaaS applications. The serverless architectural patterns are similar for both executors, so for this thesis, lightweight VMs and containers are grouped under the term 'container-based serverless platforms.'

Executing a function in a container incurs two latencies, a **warm start** latency and a **cold start** latency. Cold starts occur when the serverless platform first creates a container, injects the application code from the remote storage, initializes the language runtime, loads all application dependencies, and finally executes the function. This initialization period can take a significant amount of time, contributing to the overall latency of the function execution as experienced by the user. Minimizing this startup overhead is a critical challenge in serverless computing. Hence, both the serverless provider and serverless user prefer that most function execution goes through the warm start latency. Warm starts occur when the function executor is already prepared with the environment and application-specific initialization and kept warm in memory so that later function invocations can execute in the already initialized environment. Therefore, keeping the function executors in memory removes the expensive cold-start overhead and improves the overall function latency.

### 2.1.1 Auto Scaling Techniques

The duration of keeping the function executor warm depends on the auto-scaling technique used by the serverless platform. The two major auto-scaling techniques used in container-based serverless platforms are *scale-per-request scaling* and *metrics-based scaling* [48]. In the *scale-per-request* scheme, requests are first routed to available idle executors. If no

executor is present or all existing ones are busy servicing other requests, the serverless platform creates a new executor that serves the request. If any executor does not receive a request during the keep-alive time, the platform terminates that container and releases the consumed system resources. Thus, requests are not queuing in this pattern, and the platforms scale the number of executors synchronously with request arrivals. Major commercial serverless platforms follow the scale-per-request scaling design, including AWS Lambda, IBM Cloud Functions, Microsoft Azure Functions, and Apache OpenWhisk. Since this is the most dominant scaling scheme adopted in the mainstream serverless platforms, this thesis focuses on container-based serverless platforms with the scale-per-request scaling pattern.

In the *metrics-based scaling* scheme, the serverless platform scales executors based on the target value of various metrics such as concurrency value, CPU and memory, and requests-per-second. When scaling executors based on the concurrency value metric, multiple requests can be processed by the same executor using a concurrency value limit. Users can set this limit based on their application needs, or the platform can adjust this dynamically based on measured concurrency in the user workload. CPU-based scaling is another metric where the serverless platform scales executors, keeping CPU and memory usage within a predefined range [47]. The requests-per-second metric drives the scaling of executors based on the arrival rate of requests to each executor. The serverless platform asynchronously creates these executors using these metrics at fixed intervals. If not enough executors are present, the platform queues the requests until the desired number of executors have been created, and then the requests are executed [47]. The queuing of requests contrasts with scale-per-request scaling, which creates a new container if no existing executor can service the request. Metrics-based scaling is used by several serverless platforms such as Google Cloud Run [13], Knative [43], OpenFaas [56], Kubeless [73], Fission [27], and AWS Fargate [2].

### 2.1.2 The Cold Start Problem

While different auto-scaling techniques minimize cold starts, the number of cold starts in serverless platforms is significant [66, 45, 65]. It is shown that cold starts occur frequently, and their latency can be in the same order or magnitude as the function execution time [66]. Fuerst et al. [28] showed cold-start latency can account for as much as 80% of the total response time. Du et al. [17] analyzed the execution-to-response time ratio for 14 serverless functions. Their findings revealed that in 12 of these functions, execution time accounted for less than 30% of the total response time, indicating that cold start latency is the dominant factor in overall response delay.

Several strategies have been proposed to address the cold start problem. These include reducing the security boundary of containers that better suit the FaaS model [54], reusing snapshots of the state of the executor after function instantiation [17], functions reusing the same executor instance, having a pool of overprovisioned warm containers to serve requests [27, 13], and smart scheduling algorithms that can anticipate surges in incoming requests and proactively scale up the number of function instances [66, 32, 1]. Although these techniques help mitigate the cold start problem, they add a layer of complexity and restrictions to the platform. Moreover, these solutions attempt to circumvent the problem of slow container virtualization compared to the execution of FaaS applications. While these solutions are important research contributions, this thesis focuses on container-based serverless platforms used in practice that rely on a keep-alive threshold to keep function executors warm and prevent cold starts.

### 2.1.3 Function Keep-Alive

In scale-per-request serverless execution, the important parameter for overall function performance and resource utilization is the keep-alive threshold for the function container. When a function has finished executing in a container, the FaaS platform keeps the container 'alive' in memory for a fixed period instead of terminating it from the system. Keeping the container in memory allows successive function requests to be executed in the same container containing the function-specific initialized environment. The end-to-end function response latency skips the expensive initialization period and consists mainly of the function execution. However, keeping a container alive incurs additional resource overhead for the cloud provider. Specifically, a container not executing a function (idle state) consumes memory space [28]. Since FaaS applications experience infrequent request arrivals and only a small subset of applications account for the majority of function requests, maintaining function executors for infrequent requests imposes significant memory pressure on the system [66]. Hence, keeping containers alive in anticipation of future function requests reduces the efficiency and multiplexing of the servers [28]. Thus, the keep alive policy introduces a performance vs utilization tradeoff for the FaaS platform. Terminating a container too early incurs a performance penalty regarding end-to-end response latency to the user. Keeping the container alive longer adds a non-trivial memory overhead to the FaaS platform. Several research efforts try to find the optimal point in this tradeoff space by using a workload-aware keep-alive policy instead of a fixed threshold. Shahradeh et al. [66] uses histogram-based modeling of the Azure trace to create a dynamic keep-alive policy for individual applications. Nima et al. [48] proposes a performance model based on queuing systems and steady-state characteristics of Markov Chains that can be leveraged

to find the optimal keep-alive value with the desired service guarantee. Fuerst et al. [28] introduce a caching-based keep alive policy based on reuse distance and hit ratio curves that reduce resource overhead in Apache OpenWhisk by 30%. Given the diverse serverless workload heterogeneity, mainstream FaaS platforms use a fixed keep-alive policy for all FaaS applications [28].

## 2.2 WebAssembly-based Serverless Platforms

When serverless computing was first introduced, containers and lightweight VMs were the only virtualized environments to host and execute functions in isolation. However, FaaS workload dynamics introduce frequent expensive cold start latency and high resource overhead on container-based serverless platforms. The root cause for these challenges is that the virtualization technology is slow, and keeping function executors in memory is antithetical to the ideal FaaS principle of consuming resources only when needed [22]. The popularity of FaaS platforms indicates that the execution model is likely to remain prevalent [44]. However, there is an ongoing debate about whether containers and lightweight VMs are the most suitable environments for supporting FaaS applications [22, 21]. Wasm has recently been proposed as a promising alternative environment for the serverless execution model. [22, 21, 14].

### 2.2.1 WebAssembly Primer

WebAssembly (Wasm for short) is a portable binary instruction format for executable programs [33]. Wasm was first introduced for the web to solve JavaScript performance issues and language restrictions in the browser [68]. It allows legacy software and high-performance code in compiled languages like C/C++ to execute securely in a sandbox at the client's browser. Since 2019, Wasm has become a W3C standard, backed by major companies, including Mozilla, Google, Apple, and Microsoft. Wasm code is generated as a compilation target for programming languages and executes within a stack machine that supports functions and control flow abstractions. The Wasm stack machine is designed with formal semantics, allowing efficient and safe program execution without committing to any programming model [33]. Wasm code is executed in a memory-safe sandbox environment isolated from the host runtime [33]. The memory-safety property contrasts with Java and Python virtual machines that only support their respective programming language and need a managed language runtime that performs garbage collection to enforce memory

safety, negatively impacting performance. The key characteristics of Wasm are summarized below:

**Compact:** Wasm’s compact binary representation allows fast transfers over the internet, reducing load time and saving expensive bandwidth [42]. Wasm also has a corresponding assembly textual format that allows easy reading and debugging programs [79].

**Portable:** Wasm is designed to be portable. Wasm executables are independent of machine architecture, operating system, and language runtimes. This allows predictable application behavior regardless of the environment [68]. Once the application is compiled to the WASM format, it can be distributed and executed on all platforms (including Intel and Arm) and all operating systems (not limited to MacOS, Linux, and Windows) as long as a Wasm runtime is available.

**Safety:** Wasm is designed with formal semantics of execution and validation that provides a memory-safe sandbox environment for programs to execute, which is isolated from the host environment [33]. Wasm applications can only interact with the host environment through a specified interface.

**Performance:** The lightweight sandboxing of Wasm and binary code format allows programs to utilize the machine’s full performance. Wasm code is also streamable, i.e., validation and compilation can begin before an entire WASM code file has been transferred to the execution environment. The streamability property allows the Wasm runtime to use Just-In-Time compilation (JIT) or interpretation to execute programs [68].

The above characteristics make Wasm ideal for the browser environment, but its applications are not limited to browsers. Cloud applications also require portability to run across different machines, execute securely in a multi-tenant cloud without compromising performance, and have a compact representation for easy distribution [40]. Although Wasm was developed to target the web environment, the core of Wasm language is developed without web-specific features and assumptions [33]. Wasm programs interact with the external environment exclusively through a set of APIs; on the web, they utilize the existing browser APIs. Wasm’s environment-agnostic language design and realizing its potential for the cloud led the wider community to introduce standalone Wasm runtimes that support Wasm program execution outside the browser [77, 10]. While the browser already had standard APIs for the Wasm programs to interact with the browser, the standalone Wasm runtimes also need an interface to interact with the operating system on the server host without any dependency on the host environment. This is implemented through the WebAssembly System Interface (WASI) [75]. WASI is a standard set of modular interfaces that Wasm programs can use to securely interact with the underlying operating system. The interfaces are designed as abstractions of modern operating systems so that Wasm

programs can interface with any operating system. These interfaces include POSIX-like capabilities (e.g., file system, network, threading, and process management) and high-level interfaces like cryptography and support for neural networks. Since the Wasm program does not run on a virtualized operating system, it must securely interact with the host operating system through these interfaces without compromising other services on the host environment. WASI achieves this by following a capabilities-based security model [40]. The host has to explicitly grant permissions such as file and directory access and networking support to the Wasm program [24]. Thus, the host can limit what a Wasm program can do on a per-program basis and limit the system calls that it can access and the capabilities of the system call itself [40]. For instance, if the program needs to access a file, it needs the file descriptor with the required permissions attached. Thus, with WASI and standalone Wasm runtimes, users can run Wasm programs in the server environment. Several open-source and proprietary Wasm runtimes have emerged that are WASI-compliant, which include Wasmtime [10], Wasmer [77], WAVM [78], and WasmEdge [76].

### 2.2.2 FaaS Use Case

The most attractive use case for Wasm in the cloud is supporting serverless applications [71]. The key aspects of Wasm that make it suitable for FaaS applications and how they compare to containers are mentioned below:

**Isolation:** Similar to containers and lightweight VMs, Wasm provides a sandbox environment for serverless functions to execute without running the risk of escaping the sandbox and compromising other applications on the same host environment.

**Programming Language Support:** Since Wasm is a compilation target for compiled languages, it supports a wide variety of programming languages such as C/C++, Rust, Go, Python, Ruby, and Swift. Thus, developers can write their serverless functions in the programming language of their choice.

**Portable:** Moreover, Wasm is cross-platform. A user program only needs to be compiled once to Wasm, and the same Wasm binary can execute on different operating systems and machine architectures, thanks to the hardware abstractions provided in the Wasm runtime. In contrast, container-based serverless platforms require a new version of the function container for each operating system and machine architecture. This requires developers to have more knowledge of the server and its constraints.

**Instant startup:** A critical challenge for containers as function executors is the compulsory cold start overhead. The cold start time typically ranges from hundreds of milliseconds to several seconds when no function container is available. Wasm's compact binary

format and lightweight sandboxing allow applications to start almost instantly, ranging from microseconds to a few milliseconds [21, 71]. Thus, FaaS platforms can use Wasm runtimes as function executors without requiring complex scaling techniques to keep executors in memory.

**Vendor Lock-in:** Container-based serverless platforms are designed such that a serverless application written for one platform cannot run on another cloud provider’s serverless platform. For instance, a Lambda function cannot simply run on Azure. Each platform has its specific serverless function signature, uses a particular cloud software development kit (SDK), and has access to specific cloud services. Consequently, developers become locked into one cloud vendor, needing to understand their production environment early in the project and write their application specific to that environment. In contrast, Wasm serverless functions are essentially Wasm binaries that can execute on any machine with a Wasm runtime and use WASI interfaces to access external services securely.

### 2.2.3 WebAssembly Limitations

While Wasm offers negligible cold starts for FaaS applications and has a considerably lesser memory footprint, Wasm execution is slower than native execution [37, 68, 40]. Since Wasm is the intermediary representation before the machine code, extra time is needed to translate Wasm code into machine code. Wasm runtimes use either AOT compilation or JIT compilation for translation [68]. Thus, the performance of the Wasm program is highly dependent on the performance of the compiler engine used in the respective runtime. Moreover, execution in Wasm does not directly translate to native execution. Since a Wasm program must remain in its sandbox and only access resources it has explicitly been granted permission to, current programming language support for translating to WASI is still limited [40]. For instance, many features inherent in programming languages require Wasm to access WASI interfaces to perform the same tasks securely. WASI provides a standard set of system calls and capabilities that allow Wasm programs to interact with the underlying system securely. However, this abstraction layer means that some features available natively in programming languages may not yet be fully supported or require additional development effort to implement securely in Wasm. For example, file I/O operations, networking, and other system interactions must go through WASI interfaces, which can introduce performance overhead and complexity. As a result, while WASI is rapidly evolving, there are still gaps in support for certain language features, making it challenging to achieve parity with native execution in some cases.

## 2.3 Comparing WebAssembly and Containers for the Serverless Use Case

This section summarizes existing work in the literature that compares Wasm-based and container-based serverless environments and how this thesis compares to these works. These works can be broadly classified into two categories: 1) Presenting a WebAssembly-based serverless platform and comparing it with container-based platforms. 2) Performance evaluation-based works that compare different characteristics of both technologies.

### 2.3.1 Serverless platforms supporting WebAssembly

Several works have proposed a serverless platform based on Wasm and evaluated it against its container counterparts. Hall and Ramachandran [34] is one of the first works that advocates for a WebAssembly-based serverless execution for edge workloads to avoid the expensive cold start overhead of containers. The study characterizes simple serverless workload patterns based on single and multiple client accesses to investigate the interplay of cold start latency in containers and the slower execution speed of Wasm under these workloads. Despite Wasm’s slower execution speed, the benchmark results show that Wasm performs consistently and is faster on average across all workloads, while containers benefit from workloads that favor reuse. This study directly aligns with the aim of this thesis, but it differs in two ways: 1) The prototype presented in the paper is based on Google’s V8 engine that primarily supports Wasm in the browser. Since then, standalone Wasm runtimes have emerged that support WASI and are specifically designed for the server-side cloud environment. Thus, a new evaluation is necessary to study the current state-of-the-art Wasm-based serverless platforms. 2) The evaluation is based on simple client access workloads, whereas this thesis evaluates probabilistic performance models to compare both serverless technologies.

Sledge [30] is a serverless runtime designed for the edge based on their own Wasm compiler called aWasm and kernel bypass scheduling for optimized function startups. The evaluations showed that it performs within 1.1 times of native execution for PolyBench/C benchmark [59] and four times higher throughput and four times lower latencies compared to Nuclio, an open-source container-based serverless platform. Faasm [67] is another serverless platform tailored for stateful serverless applications that use Wasm for efficient memory isolation, Linux’s cgroups for CPU isolation, and dedicated system-level threads for function execution. The typical time it takes for Faasm’s functions to initialize is approximately 10 milliseconds. To enhance execution performance, Faasm employs techniques

such as function snapshots, warm faaslets, and parallel in-memory processing. Sledge and Faasm are experimental prototypes and do not use mature Wasm runtimes like Wasmtime or WAVM with ongoing development and industry support. As such, the performance of these systems would not reflect the use of serverless execution in Wasm for the average user. Therefore, we do not consider them in our evaluation.

Gackstatter et al. [29] propose a prototype called WOW, which integrates popular Wasm runtimes with Apache OpenWhisk to investigate the viability of Wasm as an execution environment for serverless workloads. Evaluations show that WOW reduces cold-start latency by up to 99.5%, enhances memory efficiency by over five times, and boosts function execution throughput by up to 4.2 times compared to the standard Docker-based container runtime on a Raspberry Pi and a bare-metal server across different workloads. De Palma et al. [57] present a serverless platform for private-edge cloud systems called FunLess, which leverages a WASI-compliant runtime, Wasmtime, as the function executor. They evaluate FunLess on a Raspberry Pi with three container-based serverless platforms: Fission, OpenFaaS, and Knative. The evaluations show that FunLess is a lightweight solution for the private edge cloud system, with a considerably smaller memory footprint and performance than container-based alternatives. This thesis complements the works discussed above but differs by evaluating a Wasm-based serverless framework with active development and industrial support rather than experimental prototypes.

### 2.3.2 Performance Comparison of Container and WebAssembly

Several works compare the performance of containers and Wasm to assess the viability of both technologies for serverless use cases. Mendki [50] evaluates function execution and startup times of C++ functions in a popular Wasm runtime, Wasmer, against Docker containers. The evaluation shows that Wasm functions have faster startup times and smaller memory footprints but considerably slower performance than native execution. Long et al. [46] also compare cold start and function execution performance in containers and popular Wasm runtimes. They report that many C/C++ functions running in Wasm runtimes have 10-50% faster execution times and 10x faster startup times than Docker. Kjørveziroski et al. [40] evaluate Wasm serverless functions on serverless orchestration frameworks such as Kubernetes to determine their performance at scale in a multi-cluster setting. They report that, compared to OpenFaaS, a popular open-source container-based serverless platform, Wasm serverless functions have 2x faster function instantiation and 1x smaller artifact size while providing comparable execution performance.

These works evaluate function startups and execution times in isolation rather than tandem, as in real serverless workloads. This approach limits the understanding of the

trade-off between slower Wasm execution performance and the cold start latency of containers. Since Wasm runtimes are undergoing rapid development, supporting new features and optimizing execution performance, Wasm performance has increased over the years. Therefore, this thesis aims to evaluate the current state-of-the-art serverless execution in Wasm. Additionally, Wasm has extended beyond compiled languages to dynamic languages such as Python, Ruby, Swift, and more. Since these languages are more popular in serverless workloads, this thesis also evaluates Python-based serverless functions running in Wasm runtimes.

## 2.4 Analytical Modelling

This thesis uses analytical performance models to compare containers and Wasm executors in serverless platforms. Analytical models are built using mathematical modeling and simulation to support resource planning and performance prediction for user workloads. They have been utilized to evaluate the performance of numerous cloud services, including IaaS, PaaS, and microservices [80, 48, 38]. For instance, in [80], a cloud data center is modeled as an open network with single arrivals, facilitating the extraction of response time distributions under exponential interarrival and service times. Similarly, Yang et al. [81] employs an M/M/m/m+r queuing system to derive response time distributions, decomposing it into waiting, service, and execution times. Khazaei et al. [38] have developed monolithic and interactive submodels for IaaS cloud data centers, ensuring scalability and accuracy. Additionally, Qian et al. [62] introduced a hierarchical model using the Erlang loss model and M/M/m/K queuing system to assess the quality of experience in cloud computing. Chang et al. [12] explored hierarchical stochastic models for IaaS performance under heterogeneous workloads, deriving closed-form solutions for key performance metrics. Malik et al. [49] utilized High-Level Petri Nets for the structural and behavioral analysis of VM-based cloud platforms. Finally, Tarplee et al. [70] implemented statistical programming to optimize computing resource allocation, considering uncertainty and variability in tasks, which aids in optimizing reward rate, cost, failure rate, and power consumption.

Only a couple of works have proposed analytical performance models for serverless computing. Nima et al. [48] propose a Markov Chain-based performance model that models the scale-per-request container serverless platforms. The performance model can be leveraged to predict the average serverless function performance and the resources needed for the serverless platform to support that workload. The analytical model is evaluated on AWS Lambda, showing that model predictions closely align with the workload performance on the cloud platform. However, the key parameter in the model is the keep-alive time used

by the serverless platform to autoscale function executors. The user cannot change this value for their workload in AWS Lambda, and thus, evaluating on a cloud provider platform does not show how good the performance model is when the keep-alive parameter is changed. This thesis uses this model to evaluate the performance of container-based serverless platforms. It is the first work to verify it experimentally on an open-source serverless platform, Apache OpenWhisk, where the keep-alive parameter can be tuned. The details of the model are summarized in Section 3.2. In a follow-up study, the authors present a performance model for metrics-based container serverless platforms. They validate this model using the open-source serverless platform, Knative [47]. This thesis only focuses on scale-per-request container serverless platforms as it is the most dominant scaling design in commercial serverless offerings. Another work by Rafael et al. [72] proposes a dynamic Petri net model that maps computational resources in the fog/edge environment to the computational requirements of serverless functions to optimize stream processing user applications. The work explores what-if scenarios for choosing appropriate function granularity, data size, and cost. While the paper presents a valuable performance model, it is specific to streaming applications and does not account for the dynamic scaling of function executors during workload fluctuation. Consequently, this thesis adopts the performance model proposed by Nima et al. [48].

# Chapter 3

## Methodology

This study uses analytical performance models to evaluate containers and Wasm as function executors for serverless platforms. The assumptions regarding the scheduling strategy of the models are discussed, followed by an introduction to the performance models for containers and Wasm, highlighting their respective strengths and limitations. The serverless platforms used for experimental verification and the serverless functions selected for the evaluation are then described.

### 3.1 Scheduling Assumption

In Chapter 2, two common scheduling patterns are discussed for serverless platforms. In this thesis, the performance models and serverless platforms selected follow the *scale-per-request* scheduling pattern. This scheduling design is the most common among commercial serverless platforms and the only supported pattern in serverless platforms using Wasm executors. Furthermore, it is assumed that each executor services only one function at a time, and no executors are shared between two different functions.

### 3.2 Container-based Model

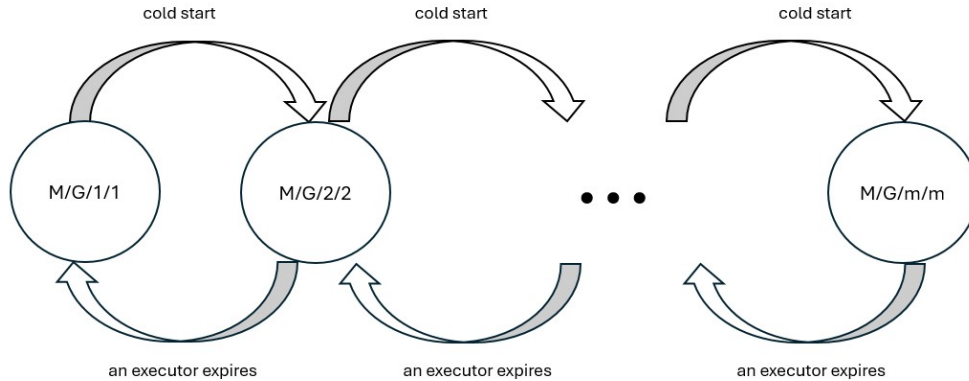


Figure 3.1: Overview of the SMP-based performance model for container executors in serverless platforms.

Nima et al. [48] propose an analytical performance model that closely follows the major commercial serverless offerings. The model is based on a continuous time Semi-Markov Process (SMP) [55] where states are represented by the number of warm executors ready to serve requests. If all  $m$  executors are busy when a new request arrives, a cold start happens, and the state transitions from  $m$  to  $m + 1$ . If any of the  $m$  executors reaches its keep-alive time, it expires, and the state transitions from  $m$  to  $m-1$ . Each state in the SMP is modeled as  $M/G/m/m$  queuing system. Fig 3.1 describes the overview of the model. The  $M/G/m/m$  queuing system assumes Poisson arrivals, a general service time distribution, and  $m$  number of servers who can serve a maximum of  $m$  number of requests. The SMP where each state is a  $M/G/m/m$  queuing system closely follows the scale-per-request pattern of commercial serverless platforms where the number of warm executors serves only one request at a time, requests follow non-preemptive first-come-first-serve discipline, no priority between requests, no distinction between different warm instances, and no queuing space between requests. The states in the SMP range from 0 (no warm executors available) to the maximum number of executors that can concurrently

serve all requests according to the available compute capacity. If the maximum number of executors cannot handle the number of requests, then those requests are rejected. As the workload fluctuates, the SMP shifts between different queuing systems. The overview of the SMP is illustrated in figure 3.1. The workload characteristics the model requires to create the SMP include the cold service time, warm service time, arrival rate, the keep-alive threshold time, and the maximum concurrency level. Instead of time-consuming experimentation and analysis, users only need to measure the warm and cold execution time of their function, which is relatively straightforward. Using these characteristics, the relevant parameters are calculated to set up the transition rate matrix of the SMP [55]. The performance model is then constructed based on the steady-state distribution of the SMP. Steady-state distribution in a Markov process refers to how likely the Markov process will be in each state. In this case, it refers to how likely the serverless platform will have each of  $m$  number of warm executors. Using the steady-state solution, the analytical model is built by finding the relevant metrics:

1. **Probability of Rejection:** The ratio of requests that will be rejected if the SMP reaches the maximum concurrency level.
2. **Cold Start Probability:** The likelihood of cold-starts happening in the system.
3. **Average Response Time:** The average response time including both warm and cold requests.
4. **Mean number of Warm Instances:** The number of instances in the warm pool.
5. **Mean number of Running Warm Instances:** The number of instances busy serving warm requests.
6. **Mean number of Running Cold Instances:** The number of instances busy serving cold requests.
7. **Mean number of Idle Instances:** The number of idle instances. These idle instances introduce memory overhead on the infrastructure.
8. **Mean Utilization:** The ratio of warm instances serving requests over the total warm instances present.

These metrics can be leveraged to inform key quality of service decisions for both the serverless provider and the end user. The serverless provider can use the *mean number of*

*idle instances* and *mean utilization* to realize the memory overhead and adjust the keep-alive time that better suits the workload. Serverless users can leverage the *mean number of running warm instances*, *mean number of running cold instances*, and *cold start probability* to realize the average number of instances required to provide their desired quality of service. Using this information, they can adopt a fine-grained serverless pricing plan that utilizes a prewarm pool of containers.

### 3.2.1 Limitations of the model

While the SMP-based performance model fairly represents the container-based serverless platform, it has certain limitations. Firstly, the process of transitioning from  $m$  to  $m - 1$  executors is not exactly Markovian. Continuous-time Markov process assumes exponentially distributed time between consecutive states, which is not valid in this case, as the service time of the last request before executor expiry is also included in the transition from  $m$  to  $m - 1$  executors. The warm executor has to process the last request and spend the keep-alive time before it expires. Secondly, the model assumes exponential inter-arrival times between the requests served by the warm executor to calculate its lifespan. However, there is no enforced restriction that the inter-arrival times of requests to warm executors should include the warm processing time. Since the requests are routed to a warm executor, the time between two consecutive requests must include the warm service time of the request. The equation in the paper that calculates the average inter-arrival time between requests routed to warm executors does not account for this condition. Solving the equation in the paper with multiple example values results in the average inter-arrival time calculation between warm requests being less than the warm processing time. I highlighted this to the authors of the paper, who acknowledged this limitation and mentioned it as a simplifying assumption to simplify the calculations. This thesis argues that using the expected value of the inter-arrival time distribution is a better-simplifying assumption to calculate the average inter-arrival between the warm requests. The model results presented in this paper are computed using this assumption as it fairly calculates the lifespan of a warm container instance compared to the original method in the paper. The effect of this change is discussed in Appendix [A](#).

## 3.3 WebAssembly Model

An ideal serverless platform provides the illusion of infinite resources and predictable latency for each request. Container-based serverless platforms aim to achieve this by max-

imizing container reuse through auto-scaling strategies. However, fluctuating workloads often result in frequent cold starts, leading to two types of latencies. On the other hand, Wasm applications have negligible cold start times, allowing Wasm-based serverless platforms to adopt a simpler design for managing serverless functions. In these platforms, each trigger of a serverless function initiates a new Wasm executor to execute the function. After the function finishes executing, the Wasm executor is removed. As new requests do not rely on the state of previous executors, all requests are serviced with consistent latency. Thus, the design of WebAssembly-based serverless platforms aligns more closely with the ideal characteristics of a serverless system. This thesis presents that such a Wasm-based serverless platform can be modeled using an  $M/G/\infty$  queue, given that the arrival distribution of the requests follows a Poisson distribution. The  $M/G/\infty$  queuing system assumes that the service time of a request is independent of both the arrival sequence of requests and the service times of requests. Assuming an infinite number of servers simplifies the analysis of this system as a request is immediately served when it arrives, and the processing of the request does not impact the processing of other requests being served or arriving later. The  $M/G/\infty$  queuing system closely approximates the Wasm serverless platform. Each request’s arrival starts a new Wasm executor, which services that request and tears down after the function execution. Thus, each request is serviced independently by its own Wasm executor regardless of the request arrival sequence and the number of requests present. Since the request is served immediately because of negligible startup times, this closely approximates an infinite number of servers available to handle the requests. Although finite computing resources place an upper bound on the number of Wasm executors that the platform can create, the analysis done in this thesis assumes that there are enough resources to create a new Wasm executor for each request. Therefore, this thesis uses the  $M/G/\infty$  queuing system to represent the Wasm serverless platform. The steady-state characteristics of the queuing system are used to build the performance model. Since a request is instantaneously served by a new executor independently and there are no separate cold and warm start latencies, the average response time is the mean service time of the distribution:

$$E[r] = E[s] \tag{3.1}$$

Moreover, the average number of executors present in the system is equal to the mean number of jobs in the system:

$$E[n] = \lambda E[s] \tag{3.2}$$

Compared to the container-based serverless performance model, this model is straightforward since the concept of cold, warm, and idle executors does not apply to Wasm-based

executors. Moreover, the *Mean Utilization* will always be equal to 1 since requests are served immediately and independently by their Wasm executor. The equations 3.1 and 3.2 are used for the Wasm performance model.

## 3.4 Serverless Platforms

The performance models discussed above are verified experimentally on open-source serverless platforms. Open-source implementations allow the visibility of the platform’s system components, which helps distinguish performance gains and system overheads. Moreover, having control of the system’s deployment allows finer-grain analysis and eliminates any obscurity from the results due to cloud-provider overhead. Secondly, this work evaluates the model on platforms that follow serverless practices commonly used in the industry. Although several serverless platforms are proposed in research that provides better performance in terms of cold starts and overall latency and throughput [17, 54, 28], these works are primarily experimental, lack ongoing development, and are proposed as proof of concept rather than to be used as a representative open-source platform. Hence, this thesis focuses on platforms that closely align with commercial serverless offerings. Finally, the serverless platforms should follow the scheduling assumptions mentioned in Section 3.1. Based on this criterion, this study uses the Spin framework by Fermion Cloud and Apache OpenWhisk as the reference Wasm and container-based serverless platforms.

### 3.4.1 Apache OpenWhisk

Apache OpenWhisk [3] is selected as the representative container-based serverless platform because it is the closest open-source implementation of a major cloud platform, IBM Cloud Functions [35]. OpenWhisk is also the most widely used open-source serverless platform that research literature has evaluated [17, 28, 54] and follows the scale-per-request scheduling pattern. Figure 3.2 summarizes the OpenWhisk architecture.

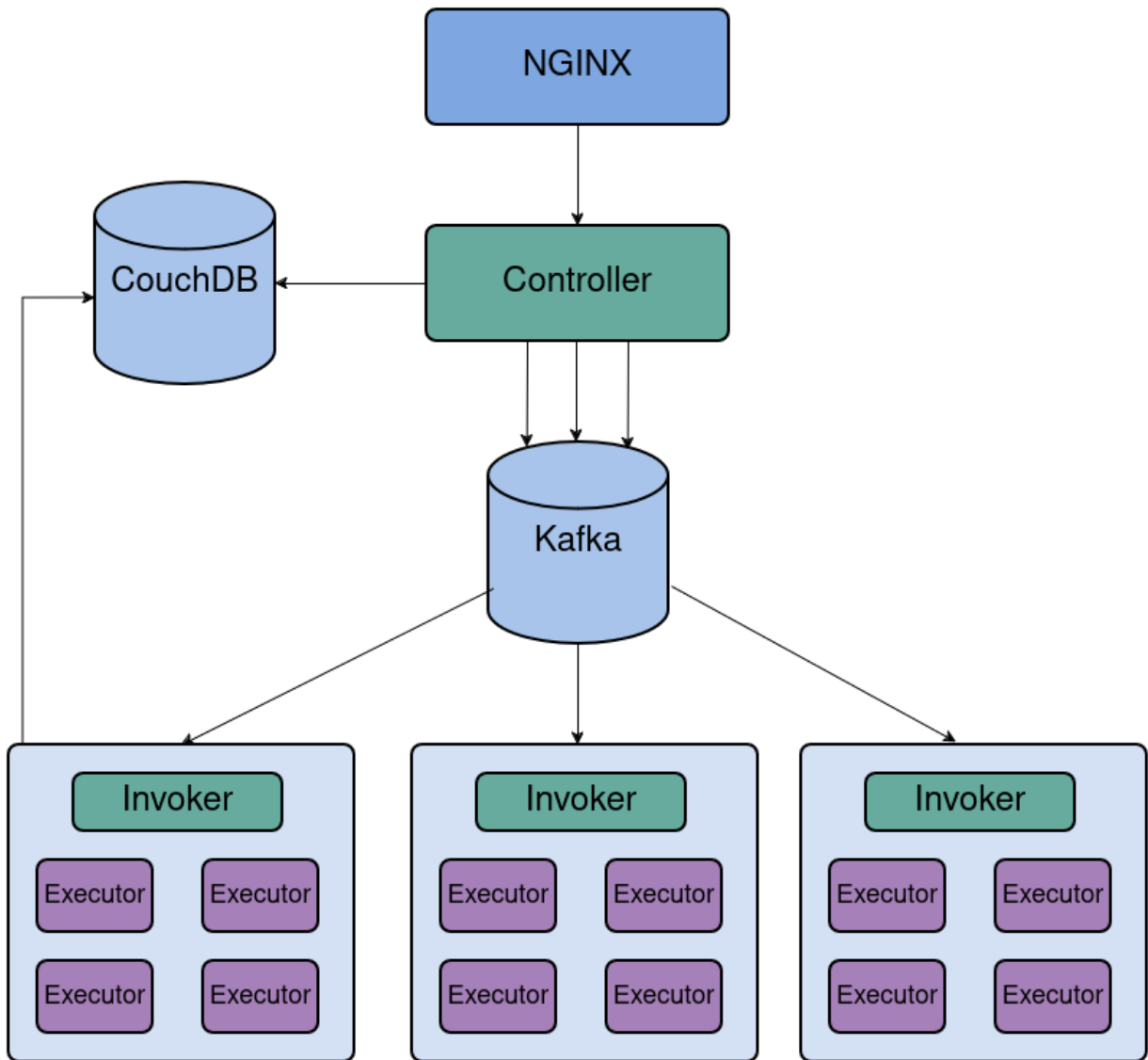


Figure 3.2: OpenWhisk architecture [6].

For brevity, the function execution and container management path in OpenWhisk are explained. A user interacts with OpenWhisk using its REST interface, which is implemented using Nginx [53]. User can create functions, trigger their execution, and query their status. These HTTP requests are forwarded to the Controller component. The controller component translates the purpose of the request, and the load-balancing logic determines the appropriate system executor component (Invoker in OpenWhisk terminology) to which the request should be routed. The load-balancing strategy makes this decision based on the available resource capacity of the Invokers and the function’s previous execution history. The function code, default parameters, and resource restrictions are pulled from the database (CouchDB [4] is the default database). The controller routes the request to the selected Invoker using the distributed message component, Apache Kafka [5]. The Invoker receives the function request and executes the function in a container. The request is routed to an existing idle container if available; otherwise, a new container will be spawned to service the request. The container can either be a Docker container or a Kubernetes-managed container if OpenWhisk is deployed on top of Kubernetes. The Invoker is also responsible for managing the lifetime of the container. By default, OpenWhisk uses a fixed 10-minute keep-alive time, after which it discards the container and informs the Controller.

### 3.4.2 Spin

The open-source Spin framework is selected as the reference Wasm serverless platform [71]. Spin is developed by Fermyon, a commercial cloud vendor offering Fermyon Cloud, a serverless platform based on Spin. It has also received wide industry support, such as smooth integration with Docker and Kubernetes for use cases involving deploying workloads using a hybrid approach of Wasm and containers [23]. Moreover, Spin remains up-to-date with the latest developments in Wasm and is seen to be the most widely adopted among other Wasm serverless platforms. Spin uses the Wasmtime runtime under the hood, which is WASI compatible and one of the most prominent and widely used Wasm runtimes. Moreover, Spin provides SDKs to support a variety of languages in which developers can build their applications, including Python, Javascript/Typescript, Rust, Go, Ruby, C, Zig, and others. It can also be extended by writing the HTTP handler of the desired language that can be ported to Wasm. Users write their application functions in the programming language of their choice and can build and run them using its simple command-line interface. Spin takes care of the necessary compilation of the application in the Wasm format, performs any optimizations to the Wasm binary to reduce startup time further, and finally executes the application using the Wasmtime engine.

## 3.5 Serverless Applications

This thesis evaluates the performance of both serverless platforms using a suite of representative applications selected from existing serverless benchmarks [39, 60, 40]. These applications cover a range of domains, including scientific computing, web applications, utility functions, image processing, and text processing. The selection criteria focused on including only those functions that could be compiled into Wasm. As support for Wasm is still evolving, many popular packages that require system resources, such as multithreading or legacy code, do not yet have official Wasm compilation targets [40]. However, ongoing developments in the Wasm language, including the WebAssembly Component Model [11] and WASI, are likely to reduce the gap between Wasm and traditional execution environments in the future. Current tools facilitate the development and execution of serverless functions in Wasm, and it is anticipated that Wasm serverless platforms will increasingly support more feature-rich runtimes and programming languages [22].

The applications used in this evaluation are categorized according to the following serverless use cases:

**Scientific Computation.** Serverless functions often consist of computational workloads and several research efforts propose serverless as the right fit for optimizing large-scale scientific workflows [15]. Thus, two microbenchmarks implemented as serverless functions are included. The *float* function involves floating point operations, and the *matmul* function computes square matrix multiplication.

**Web Applications.** A common use case of serverless functions is to perform back-end processing tasks in web applications [15]. The *chameleon* function serves dynamic HTML content for a static website.

**Security.** Serverless functions are also used in security-related applications [15]. The *aes* function performs AES symmetrical encryption and decryption.

**Image Processing.** Processing image data is one of the most prevalent serverless workloads [64]. Prominent serverless benchmarks include image processing functions in Python [15], [65], [39]. However, image processing libraries in Python like *Pillow* [58] currently do not compile to Wasm. Thus, an equivalent *image-thumbnail* Rust function is included, creating an image’s thumbnail.

**Text Processing.** Processing textual data is another typical serverless workload [39]. The *whatlang* function determines the language of a given text. The *whatlang* function determines the language of a given text.

Table 3.1 summarizes the serverless applications used for the evaluation in this study.

Name	Description	Benchmark	Language	Domain
float	Create 100,000 point objects which compute <code>math.cos()</code> , <code>math.sin()</code> and <code>math.sqrt()</code> .	Pyperformance [60]	Python	Scientific Computing
matmul	Performs square multiplication.	Wasm Orchestration [40]	Go	Scientific Computing
aes	Encrypt and decrypt a given text multiple times using AES symmetrical encryption.	Wasm Orchestration[40]	Go	Security
whatlang	Determine the natural language of a given string.	Wasm Orchestration [40]	Rust	Text Processing
chameleon	Dynamic HTML generation from a template using the chameleon module.	FunctionBench [39]	Python	Web Applications
image-thumbnail	Create a thumbnail of an image.	Wasm-learning [69]	Rust	Image Processing

Table 3.1: Description of the serverless benchmarks.

# Chapter 4

## Evaluation

This chapter summarizes the experiments performed to validate the performance models described in Chapter 3. All artifacts and source code used for the evaluation are publicly available on GitHub<sup>1</sup>.

### 4.1 OpenWhisk Evaluation

As discussed in Section 3.2, the complexity of the model that uses containers as function executors stems from the keep-alive time used by the serverless platforms. However, the original paper's authors validated the model on AWS Lambda where the Lambda platform fixes the keep-alive parameter of 10 minutes. The biggest strength of the performance model is that it helps find the optimal keep-alive value for the specific workload such that there are fewer idle executors and, simultaneously, the average function response time is closer to the warm response time. No existing work validates the performance model for varying keep-alive times. Therefore, the keep-alive time is varied to validate the performance metrics obtained from the model while keeping a constant arrival rate. This is done for each function in the Table 3.1. Additionally, the arrival rate is varied to test the serverless platform's behavior with increasing load until the system's CPU capacity is reached. The results are summarized in Section 4.3.

---

<sup>1</sup><https://github.com/abdulmonum/serverless-performance-evaluation.git>

### 4.1.1 Experimental Setup

OpenWhisk is set up using its Kubernetes deployment option. Although OpenWhisk could be deployed without a container orchestration system by only using Docker, the deployment scripts with the simple Docker setup are outdated and not maintained. The recommended way to deploy OpenWhisk for development and production is using the Kubernetes option [6]. A three-worker node setup is used, where the first node hosts the Kubernetes control plane, the second node hosts the OpenWhisk system architecture components as shown in Figure 3.2, and the third node hosts the function executors and the Invoker component that manages the executor’s lifecycles. The Kubernetes control plane node and the OpenWhisk system component node have 8 vCPUs (Intel E3-1230) and 16 GB of memory running Ubuntu 22.04. The node that runs the function workload has 40 vCPUs (2x Intel Silver 4114) and 192 GB of memory, also running Ubuntu 22.04. All experiments are performed on OpenWhisk built with Git commit *ef725a6* using the public deployment scripts<sup>2</sup>. The following changes are made from the default settings:

1. Increasing the limits on the number of concurrent invocations and the number of invocations per minute.
2. Increasing the Java heap memory of the Controller and Invoker components.
3. Increasing the total memory pool from which the Invoker component creates new containers.
4. Adding Python version 3.11, Rust version 1.72, and Go version 1.20 container runtime images that execute the OpenWhisk functions. When comparing results with Wasm executors, this is done to eliminate any performance gap due to language version optimizations.
5. Changing the keep-alive period setting based on the experiment. Like commercial serverless platforms, OpenWhisk does not support tuning the keep-alive period for specific workloads and only allows a global keep-alive period for all workloads. As such, OpenWhisk is rebuilt with each keep-alive time used in the experiment.

Python is used to simulate the client triggering the workload. The *asyncio* library is used to communicate with OpenWhisk’s REST API to create requests (triggering the function) and process the result for each request with a request-reply pattern. The resulting

---

<sup>2</sup><https://github.com/apache/openwhisk-deploy-kube>

reply contains the function response, some metadata, and OpenWhisk internal logging metrics, including the start and end time of the function, the waiting period before the function is executed, and the duration of the function. The workload is first executed for a warm-up period of 5 minutes. A warm-up period is chosen so that the workload reaches a steady state, where the serverless platform has created the initial number of executors necessary to service the workload. After the warm-up period, the experiment is run for 20 minutes, and the responses to all requests are recorded in a log file. The function responses do not contain any information about the function executors that executed the function. The data about function executors and their lifecycle throughout the experiment is gathered through the Invoker component’s container log. This log contains all the activity of function executors created during the experiment. The internal logging measurement provided in the OpenWhisk function response and the activity log of the Invoker component are used to calculate the relevant metrics of the performance model as described in Section 4.1.3. The experiment is repeated for every keep-alive period, which varies between 1 and 180 seconds. A fixed arrival rate of 1 request per second is used for all experiments. The experiment workflow is summarized in Algorithm 4.1.1.

---

**Algorithm 1** Keep-alive Experiment Workflow

---

```

1: Input: Array keep-alives, Array functions, Integer experiment_duration
2: for each element function_name in functions do
3:   for each element i in array keep-alives do
4:     Build OpenWhisk with keep-alive period i
5:     Wait until all OpenWhisk system components are running
6:     Deploy function function_name to OpenWhisk
7:     Invoke the workload for warm-up time
8:     Invoke workload for experiment duration experiment_duration
9:     Save experiment metadata, function responses, and Invoker container logs
10:    Tear down OpenWhisk
11:    Wait until all OpenWhisk components have gracefully terminated
12:   end for
13: end for

```

---

The predictions from the container-based performance model are generated using the publicly available source code<sup>3</sup> of the original paper with the modification as mentioned in 3.2.1. The performance model requires the function’s warm and cold service times to build the SMP. Each function’s warm and cold service time in Table 4.1 is measured using

---

<sup>3</sup><https://github.com/pacslab/serverless-performance-modeling>

an average of 10 runs. For measuring the cold service time, the function is triggered only when no function executor is available to service the request. For measuring the warm service time, the function is triggered only when a function executor is available to service the request.

The load experiment is performed similarly to the keep-alive experiment, except the keep-alive time is fixed, and the arrival rate is varied until the client cannot get a response from the serverless platform. A fixed keep-alive period of 60 seconds is used, at which most requests go through the warm start phase for this function. This keep-alive setting is estimated using simulations from the performance model. The workload is first invoked for a warm-up period of 5 minutes, followed by the experiment duration of 10 minutes.

### 4.1.2 Warm Request Routing in OpenWhisk

One of the container-based model assumptions is that the serverless platform first routes requests to the most recently used containers, using older executors only if the new ones are busy. Thus, the scheduler prioritizes recently available idle executors for requests, maximizing the chance of older containers expiring, effectively leading to the minimum number of containers necessary to service the workload. Upon inspecting OpenWhisk's source code, it is found that OpenWhisk chooses a random container from the pool of idle containers to execute the function. OpenWhisk maintains a hashmap of idle containers and chooses the function container for execution, which it finds first in the hashmap. Thus, the container selected for execution may not be the most recently used container for execution. Since the model assumes a higher priority of requests to newer idle executors, this work adds this functionality to the OpenWhisk system. The functionality is easy to add since OpenWhisk already tracks the recently used time for each available idle container. The change is added to the *schedule* function in the `ContainerPool.scala` file of the Invoker component. The Invoker component container image is built with the updated source code, and OpenWhisk is built using this custom image for all experiments. The updated container image is publicly available at DockerHub<sup>4</sup>.

The suspected problem with the OpenWhisk approach used for scheduling requests is that it may not be resource-conserving, leading to more idle containers than necessary. However, there was a negligible difference between the model predictions and the evaluation of OpenWhisk with and without the source code change discussed above. Therefore, only the results with the source code change are presented in the thesis as they better suit the assumptions of the performance model.

---

<sup>4</sup><https://hub.docker.com/repository/docker/abdulmonum/invoker>

### 4.1.3 Analytical Model Validation

This section outlines the methodology to measure the performance metrics of the model from the experiments.

**Mean Response Time:** The mean response time is calculated by taking the mean of all the requests' end-to-end response time. This includes both warm and cold start requests.

**Probability of Cold Start:** The probability is calculated by dividing the number of requests causing cold starts from the total number of requests made during the experiment. The requests causing cold starts are identified from the request's response, which contains OpenWhisk's internal logging metrics.

**Mean Number of Running Warm Executors:** This metric is calculated by observing the number of requests serviced by a warm container at regular intervals of 2 seconds, taking the average as the estimate.

**Mean Number of Running Cold Executors:** Similar to the estimate of the number of running warm executors, but only considering the requests that are serviced from containers that are cold started.

**Mean Number of Running Executors:** This metric is calculated using the sum of running warm and cold executors.

**Mean Number of Total Executors:** This is calculated by observing how many function executors are present in the system at regular intervals of 2 seconds, taking the average as the estimate. The function executors' lifetimes are calculated using the container log of the Invoker, which contains the timestamps for the creation and deletion of all the executors during the experiment.

**Mean Number of Idle Executors:** The mean number of idle containers is measured by subtracting the mean running executors from the mean total executors.

## 4.2 Spin Evaluation

### 4.2.1 Experimental Setup

Spin is set up using its officially released self-contained binary. Serverless functions are adapted using the respective language's SDK format provided by Spin. The Spin machinery builds the serverless function from the programming language to the Wasm format. It uses

the Wasmtime engine and language-specific tooling under the hood to build and optimize the Wasm binary. Rust functions are converted to Wasm using the Cargo tooling, Go functions are converted to Wasm using the tinyGo tooling, and Python functions are converted to Wasm applications using the py2wasm tool written by the Spin team. The serverless function is ready to serve and respond to HTTP requests using a simple ‘spin up’ command in the application directory. Spin version *2.4.2* is used for all experiments. Spin is hosted on the same machine that hosts container function executors for OpenWhisk. The machine has 40 vCPUs (2x Intel Silver 4114) and 192 GB of memory, running Ubuntu 22.04.

The client is simulated using Python to execute the workload using Poisson arrivals, similar to the OpenWhisk evaluation. The only difference is that there is no warm-up time before the experiment since there is no need to create an initial number of executors to service the workload as an independent Wasm executor serves each request.

## 4.2.2 Analytical Model Validation

Compared to the model of container executors, the model for Wasm executors has only two comparable metrics: the **mean response time** and the **mean number of total executors**. The mean response time is calculated by measuring the response time of each request and taking the average of all requests. The mean number of total executors is measured by observing the system every second, counting the number of inflight requests, and taking the average as the estimate.

## 4.3 Experimental Results

As discussed in Section 3.2, the execution times of Wasm are found to be slower than native execution [34, 37]. Table 4.1 compares the execution times of Wasm serverless functions with warm execution times in containers. Wasm execution is 1.2-3.2 times slower than the function’s native execution in a container. Go and Rust functions’ execution in Wasm is within two times the execution speed in the container except for the *matmul* function, which is a more computationally expensive function. Python code is 2-3 times slower in Wasm, suggesting that Wasm has limited execution performance benefits when executing interpreted language code by compiling the underlying language runtime to Wasm.

Function	Container Cold Response Time (s)	Container Warm Response Time (s)	Wasm Response Time (s)	Wasm Slowdown
aes	2.5849	0.0259	0.0308	1.2x
matmul	2.5625	0.0130	0.0359	2.8x
whatlang	2.5182	0.0141	0.0255	1.8x
image-thumbnail	3.0567	0.1903	0.3703	1.9x
float	4.1202	1.5288	4.8760	3.2x
chameleon	3.5557	0.0680	0.1536	2.3x

Table 4.1: Response times of workloads analyzed in this study.

### 4.3.1 Keep-Alive Experiment

The respective response times of Wasm and container execution in Table 4.1 are used as input to the performance models. The results for each performance metric in the models are summarized below. When the performance metrics for container and Wasm are comparable, they are presented on the same graph for direct comparison. As outlined in Chapter 3, the performance model metrics capture the steady-state behavior of the workload.

Figure 4.1 shows the average response time of the functions across both serverless platforms. The figure shows that the container model predictions closely align with the experimental evaluation on OpenWhisk. As the keep-alive time increases from 1 to 7 seconds, the mean response time decreases rapidly, with the response time becoming dominated by the warm start phase rather than the cold start phase. When the curve flattens, the mean response time is almost equal to the mean warm start latency, suggesting that the workload almost always goes through the warm invocation phase. The point where the curve starts to asymptote can be used as the ideal keep-alive parameter to get the best response time for the workload since increasing the keep-alive value further does not yield a substantial benefit in the average response time. The model predictions for Wasm also closely correspond with the results from the Spin evaluations. For *float* function, the workload experienced a higher response time than the simple average of 10 runs, suggesting overhead in Spin when running concurrent CPU-bound functions in Python.

Figure 4.2 plots the average number of executors busy serving requests as the keep-alive value is varied. Similar to the previous figure, the container model predictions closely follow the experimental result. The number of running executors is higher for smaller keep-alive values as executors are more frequently created due to a shorter keep-alive time.

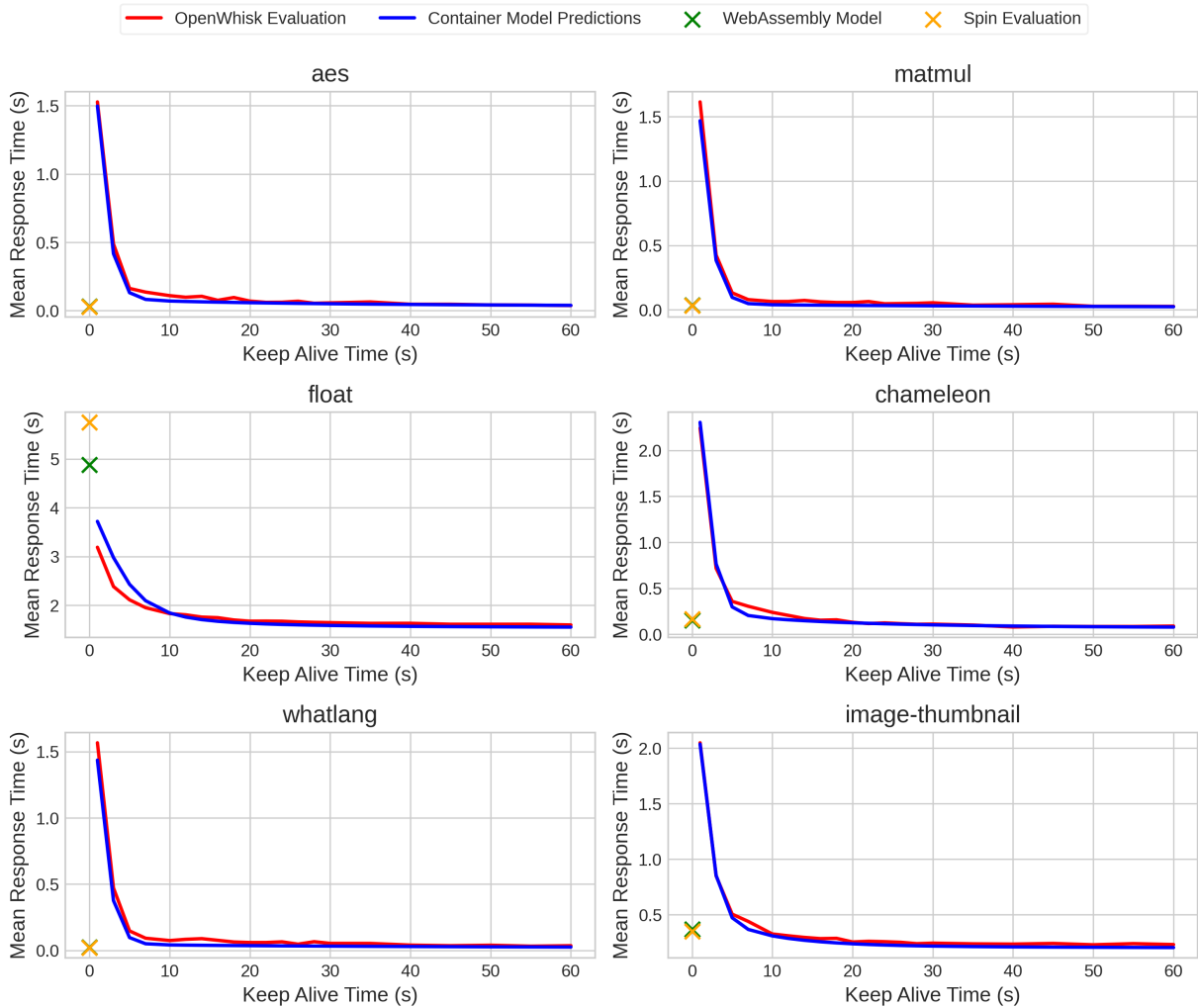


Figure 4.1: Mean response times against keep-alive time.

The number of executors stabilizes as enough executors are kept alive for the subsequent requests in the workload. For Wasm, the model and Spin evaluation closely align together. The number of running Wasm executors compared to container executors follows the same pattern as the response time difference between containers and Wasm, as seen from Table 4.1. For example, the *float* function has more running Wasm executors since its response time is 3x slower than the container, so more executors are needed to serve the workload.

Figure 4.3 presents the average number of cold executors in the system as the keep-

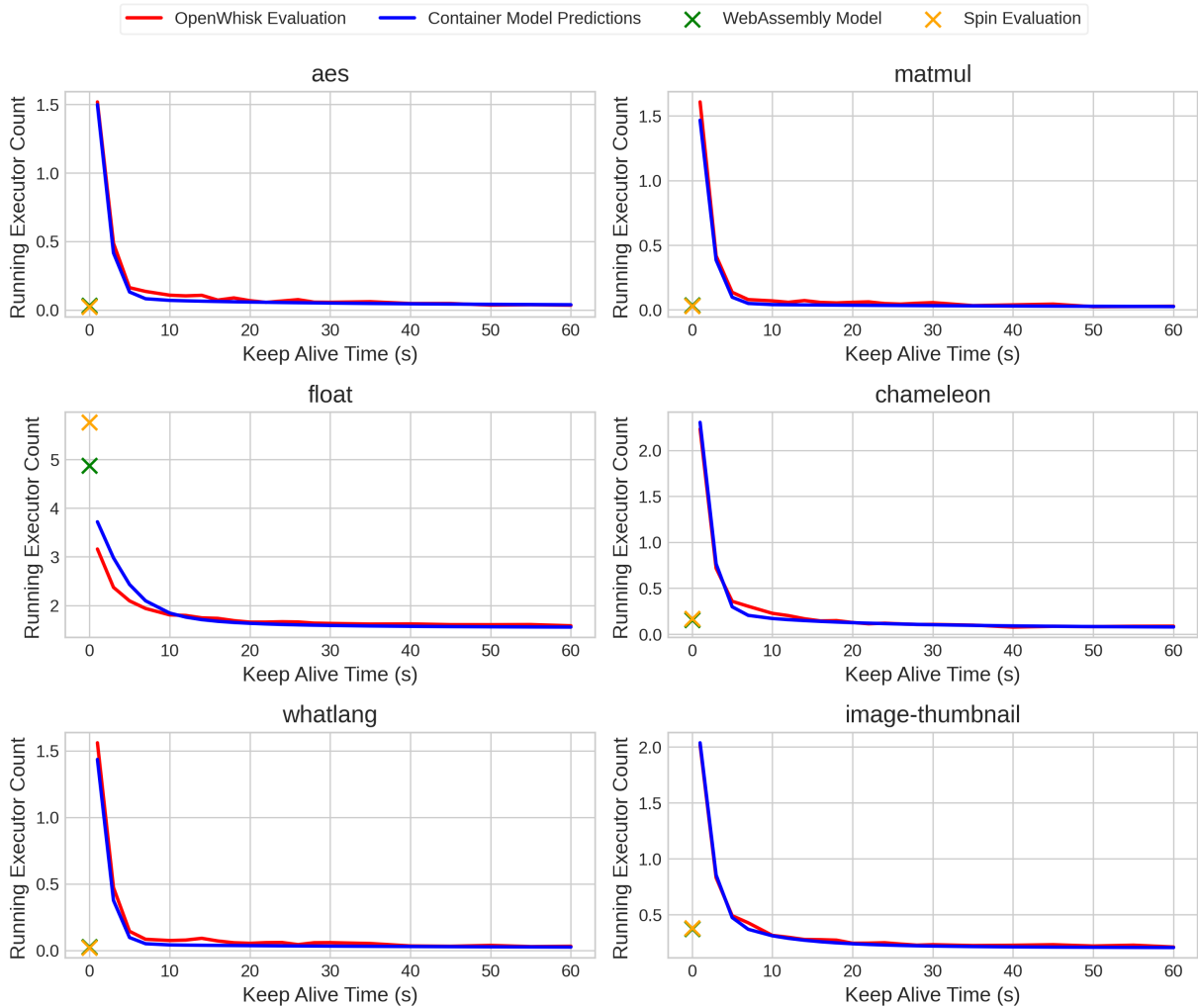


Figure 4.2: Running executor counts against keep-alive time.

alive time increases. For shorter keep-alive durations, the majority of requests experience cold start latency. As the keep-alive time increases, the number of cold starts becomes negligible, indicating that sufficient executors are available to handle incoming requests, with cold starts occurring only during workload spikes. The model predictions closely match the experimental results observed on OpenWhisk.

Figure 4.4 plots the number of warm executors busy serving requests. As the keep-alive time increases, the number of containers serving warm requests increases as containers are

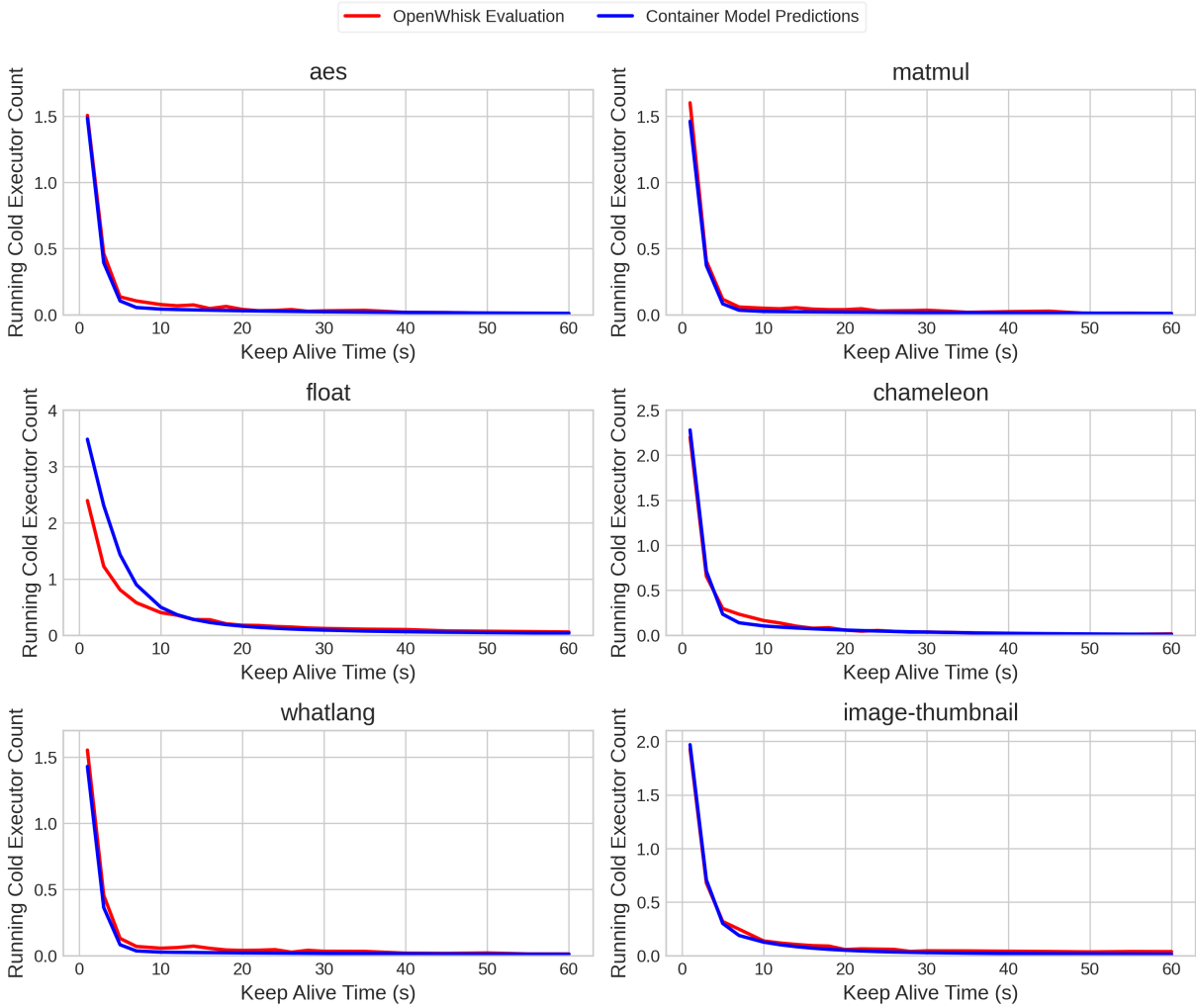


Figure 4.3: Running cold executor counts against keep-alive time.

not terminated as frequently. The number of warm containers stabilizes when the keep-alive time is high enough to ensure sufficient containers remain in memory to handle most requests. Consequently, nearly all requests are served by warm containers. The observed fluctuations in the experimental results are associated with functions that have very short execution times; therefore, measurement uncertainty is more prominent in these quick executions.

Figure 4.5 shows the average number of idle executors during the workload. The in-

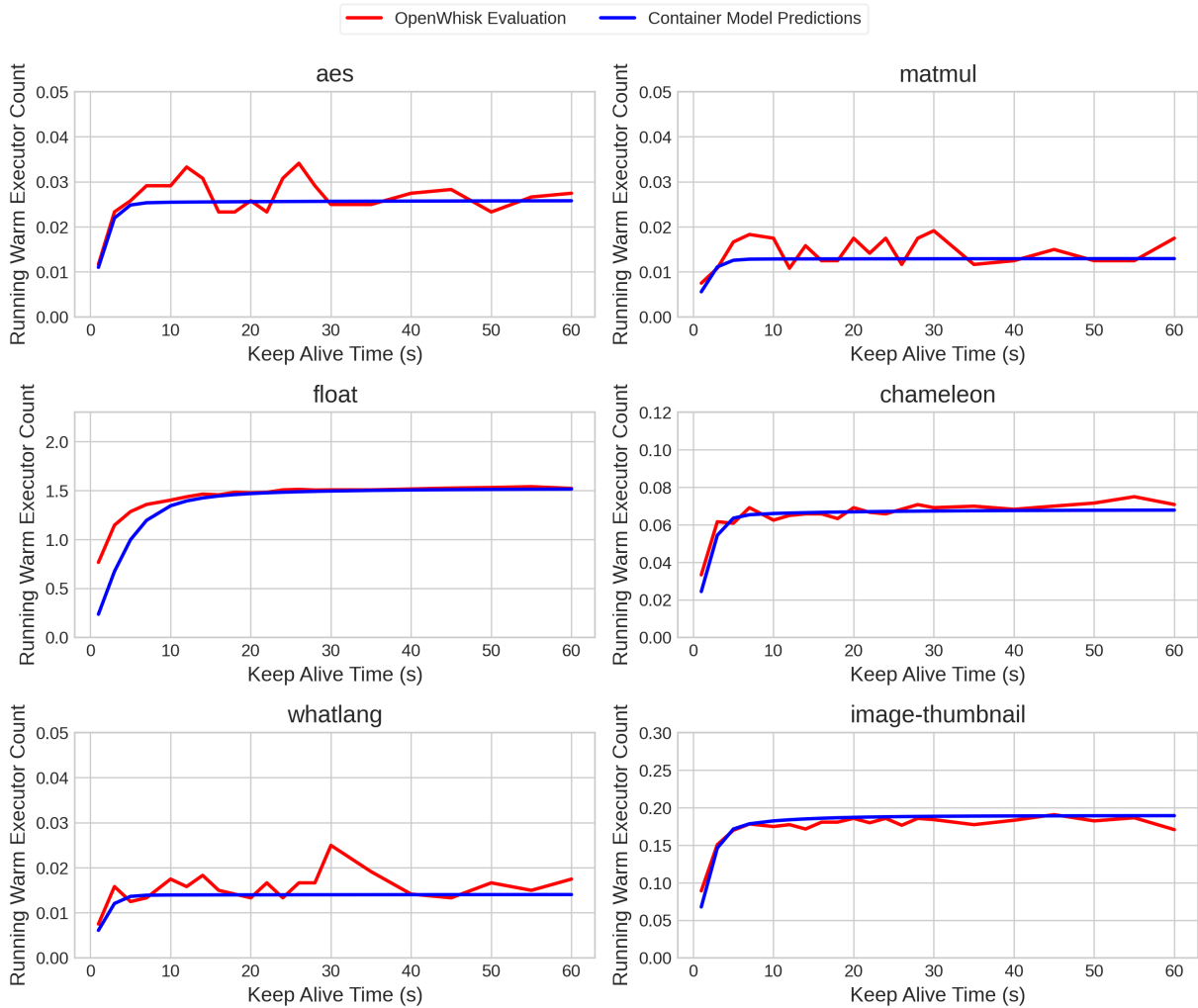


Figure 4.4: Running warm executor counts against keep-alive time.

crease in idle containers follows a pattern similar to a logarithmic increase. For shorter keep-alive times, containers are primarily occupied with serving requests and are terminated soon after, leading to minimal idle time before they are discarded. As the keep-alive time increases, more containers are retained in memory for extended periods than necessary, resulting in more idle containers. The model closely predicts the increase in idle containers, as shown in the figure. There are no idle executors in Wasm, as executors are terminated when they finish serving the request.

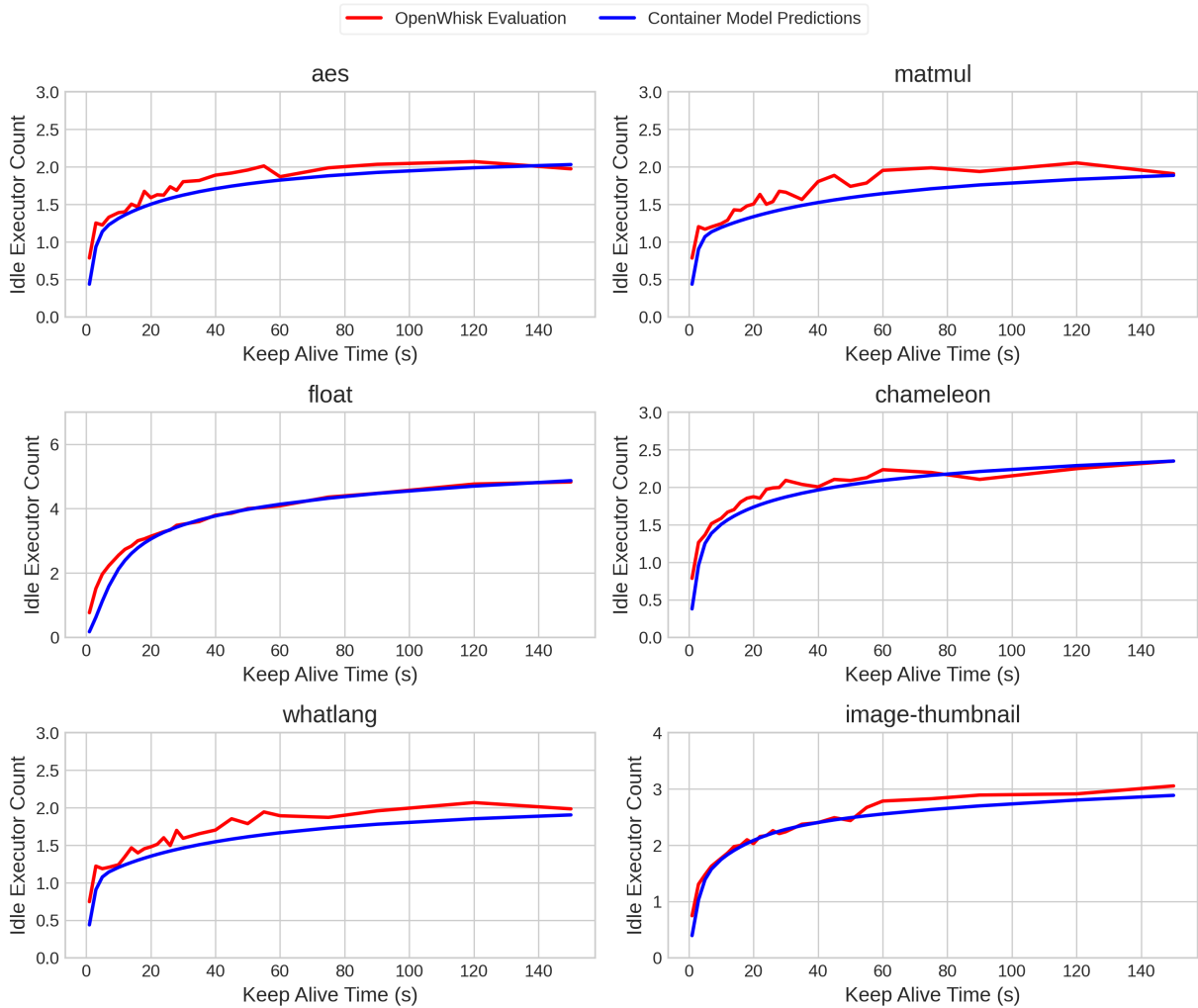


Figure 4.5: Idle executor counts against keep-alive time.

Figure 4.6 presents the average number of total executors during the workload. For container executors, the graph reveals a notable inflection point. Initially, the total number of executors decreases as the keep-alive time increases, reflecting a period where cold starts occur, and the number of cold requests decreases as keep-alive time increases. Figure 4.4 shows the same decrease for this period. Subsequently, the number of executors reaches an inflection point where cold starts are infrequent, and enough executors are available to handle requests. As keep alive value increases, the total number of executors increases.

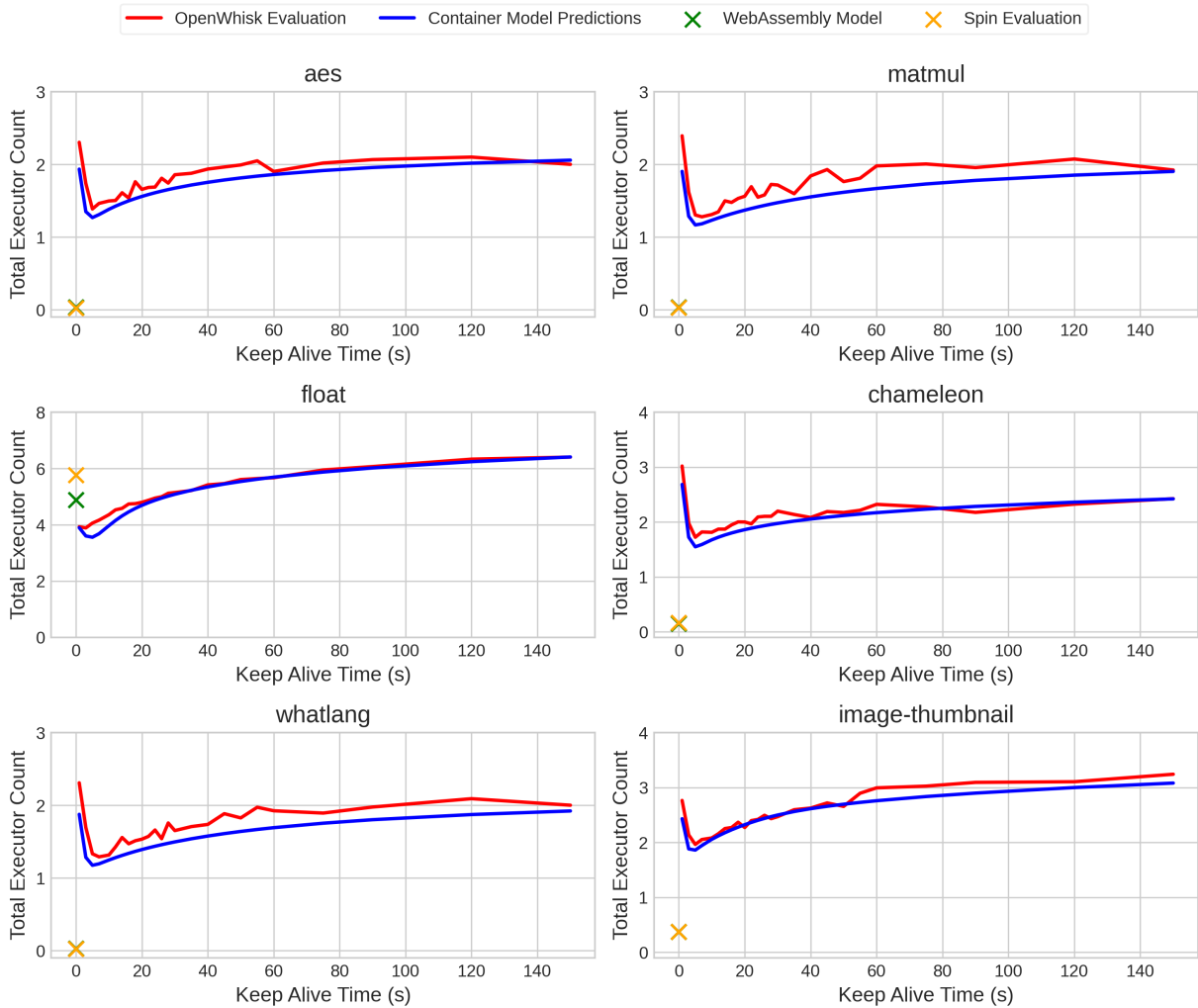


Figure 4.6: Total executor counts against keep-alive time.

This increase corresponds to the increase in idle containers observed in figure 4.5 as more containers are kept in memory. For Wasm executors, the total number of executors corresponds to those actively serving requests. Both models for Wasm and container executors closely predict the total number of executors in the system, as verified by the Spin and OpenWhisk evaluations, respectively.

Figure 4.7 plots the cold start probability as the keep-alive increases. The cold start probability predictions from the model and the evaluation results are similar for all func-

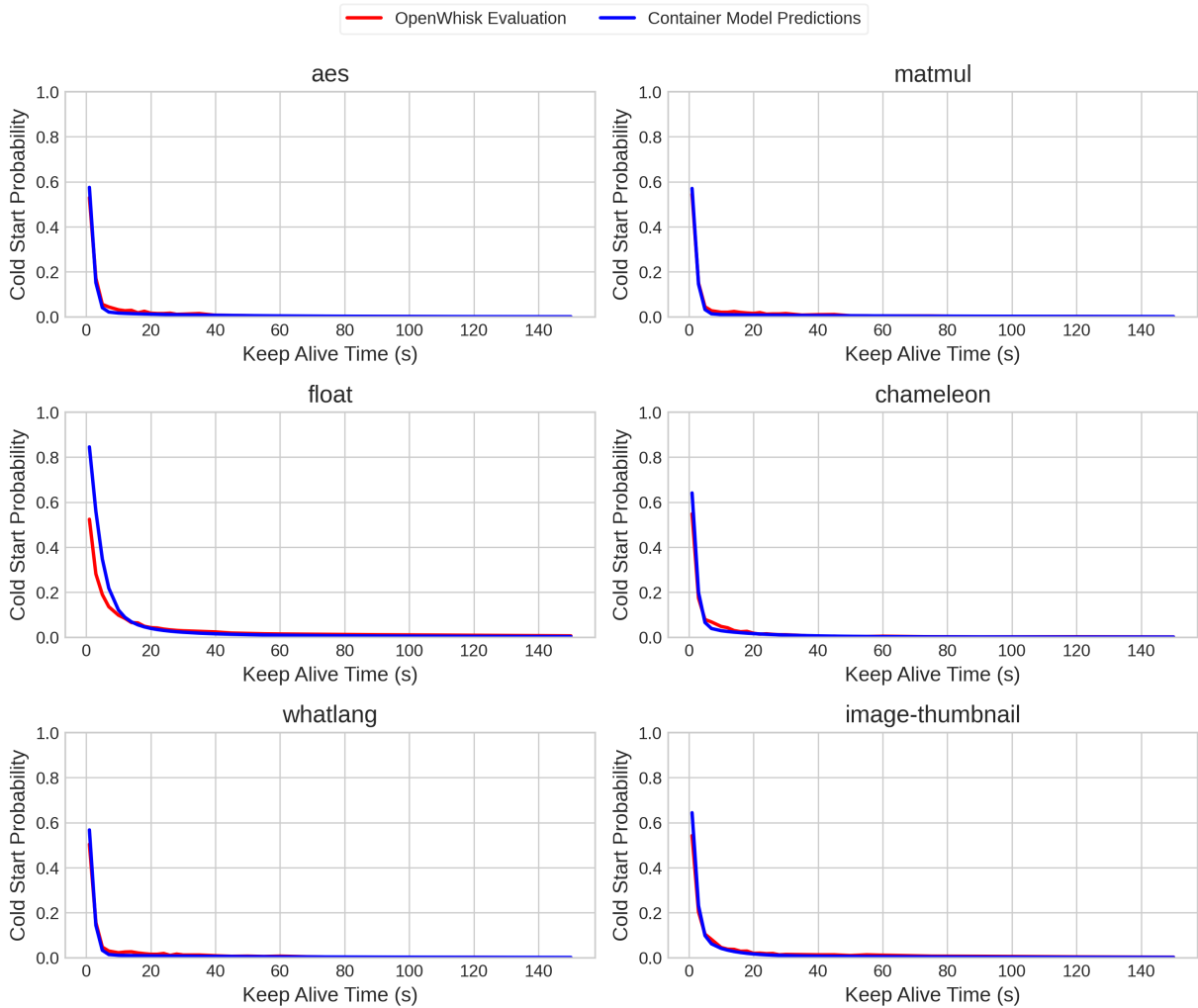


Figure 4.7: Cold start probability against keep-alive time.

tions except the *float* function, which predicts a higher probability for very small keep-alive values. The higher probability prediction of the model comes from the average lifespan calculation of the executor, which is highly sensitive to the keep-alive parameter.

Figure 4.7 plots the probability of cold starts as the keep-alive time increases. The model predictions for cold start probability are consistent with the evaluation results for most functions. However, for the *float* function, the model predicts a higher probability of cold start for very short keep-alive times. This discrepancy arises because the model's

calculation of the average executor’s lifespan is highly sensitive to the keep-alive parameter.

### 4.3.2 Load Experiment

This section summarizes the experimental results by testing the system on high load by increasing the arrival rate of the requests till the system’s CPU capacity is reached. When testing with higher arrival rates on OpenWhisk using functions except for the *float* function from Table 3.1, bottlenecks and issues arise within OpenWhisk, limiting the experiment’s validity. Since the *float* function has two orders of magnitude higher execution time than other functions in the benchmark suite, the CPU capacity of the system is reached easily with a relatively lower arrival rate, as discussed later in Figure 4.8. Thus, only the experiment results for the *float* workload are comparable for both containers and Wasm executors. On the other hand, since the Spin framework is a relatively lightweight serverless platform, no such bottlenecks were observed when testing with Spin. The results in the section are outlined as follows:

First, the results for the *float* function are analyzed for both container and Wasm executors, using figures 4.8 to 4.14. When performance metrics for containers and Wasm are comparable, they are presented on the same graph for direct comparison. Next, the performance of Wasm functions with Spin is examined using the *aes*, *matmul*, and *whatlang* workloads. The evaluations were conducted on 10 system cores, with additional cores disabled. The use of 10 cores was deemed sufficient for the experimental objectives.

Finally, the discussion addresses the challenges encountered, the fixes attempted, and the partial results obtained when testing workloads other than *float* from Table 3.1 on OpenWhisk, using the *aes* function as a case study. For this part of the experiment, the system running the function workload is restricted to 10 CPU cores, and the system hosting Kubernetes control plane has 40 vCPUs (2x Intel Silver 4114) machine with 192 GB of memory and 1024Gb NVMe disk. The Kubernetes control plane operates on a machine with 40 vCPUs (2x Intel Silver 4114), 192 GB of memory, and a 1024 GB NVMe disk, selected for its SSD to enhance etcd performance [20].

Figure 4.8 illustrates the average response time of the *float* workload as the arrival rate increases. For OpenWhisk, the response time remains close to the container’s warm response time up to an arrival rate of 11 requests per second (req/sec). The response time experiences a modest increase between 8 and 11 req/sec, likely due to increased latency from the contention of resources with increased concurrent execution. This period reflects that sufficient executors are available, and the workload goes through the warm start phase. Beyond this point, the response time increases sharply to around 8 seconds at 12 req/sec.

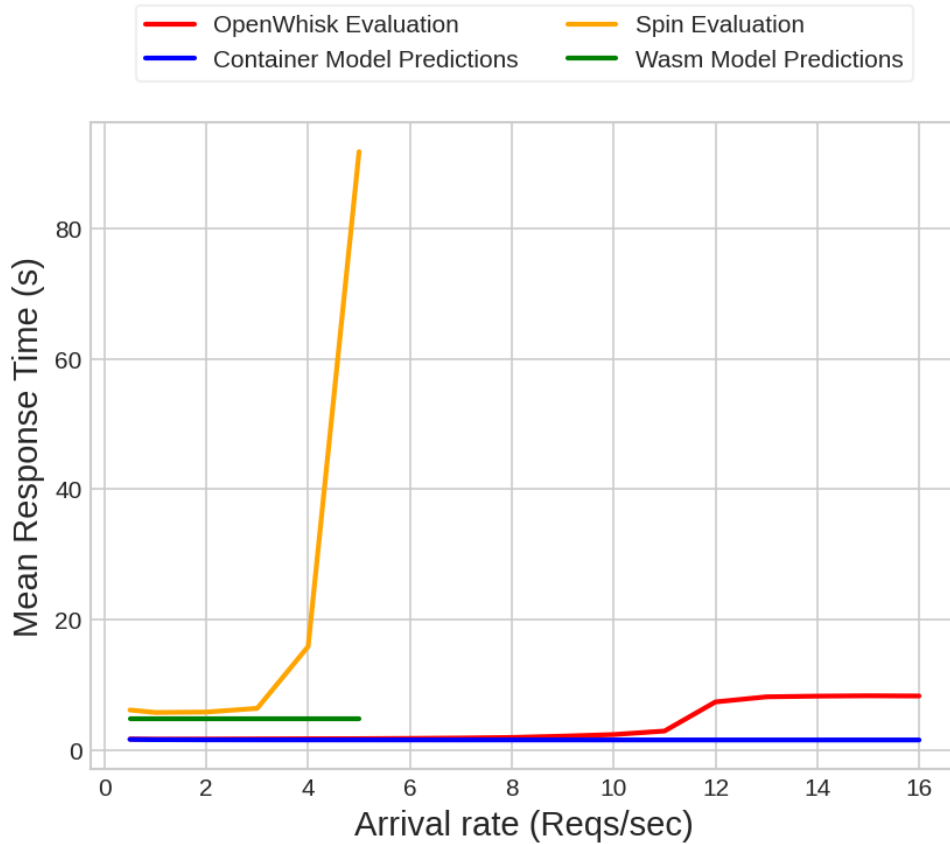


Figure 4.8: Mean response times against arrival rate.

It remains approximately constant until 16 req/sec when clients experience timeouts while waiting for responses. This indicates that OpenWhisk approaches its capacity around 11 req/sec. In contrast, the workload in Spin maintains a response time of 5-6 seconds up to an arrival rate of 3 req/sec but then sees a significant increase to about 90 seconds at 5 req/sec, suggesting that Spin reaches its capacity at around 4 req/sec. The figure demonstrates that OpenWhisk can handle an arrival rate approximately three times higher than Spin. This is consistent with the observation that the response time for the *float* function in Wasm is roughly three times slower than in native execution. Both performance models closely match the experimental results until the system’s CPU capacity is reached.

Figure 4.9 illustrates the increase in running warm executors in OpenWhisk as the workload arrival rate rises. When the system approaches its capacity at around 11 requests

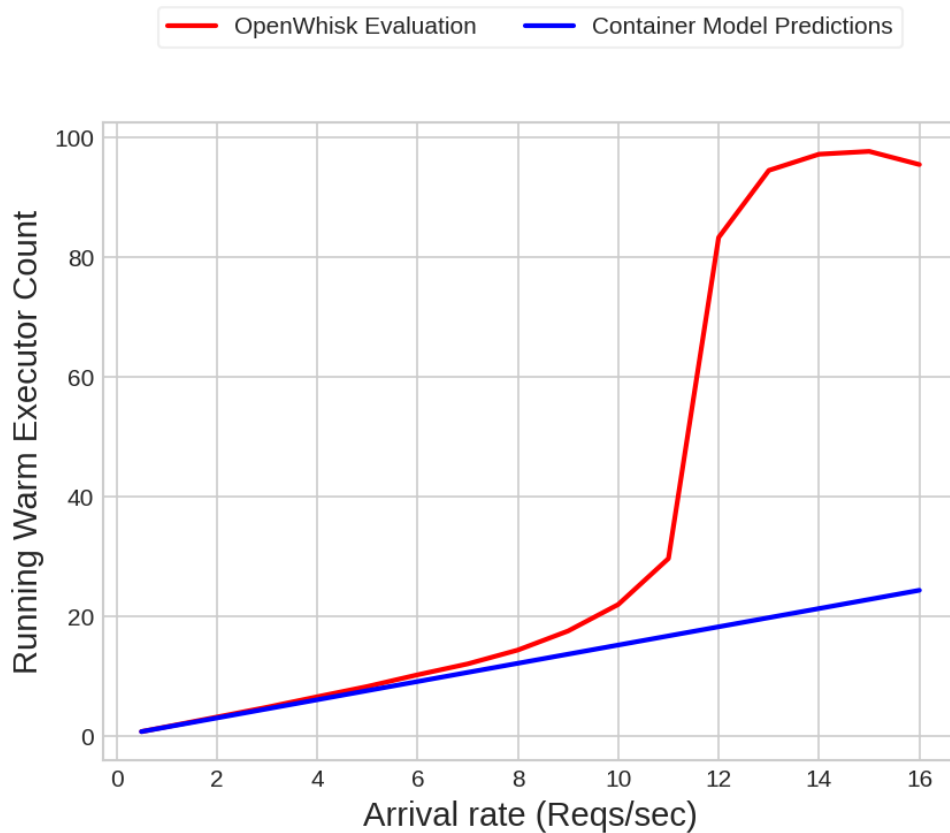


Figure 4.9: Running warm executor counts against arrival rate.

per second (req/sec), the number of warm executors increases sharply, exceeding 80. This surge corresponds with the rise in response time to approximately 8 seconds, as shown in Figure 4.9. The increased response time triggers OpenWhisk to create additional executors to manage the incoming requests. Both the model predictions and the experimental results align closely until they begin to diverge at an arrival rate of 8 req/sec, just before the system reaches its capacity.

Figure 4.10 plots the system’s average number of cold executors. Cold starts are significantly fewer than warm starts because the keep-alive time is long enough to maintain enough executors in the system. The experiment and model prediction is approximately similar until the CPU capacity is reached at 10 req/sec, with only an average of 0.3-1.5 more cold containers seen in the experiment than in the model prediction. There is a slight drop in cold executors starting at an arrival rate of 12 req/sec, likely due to creating more

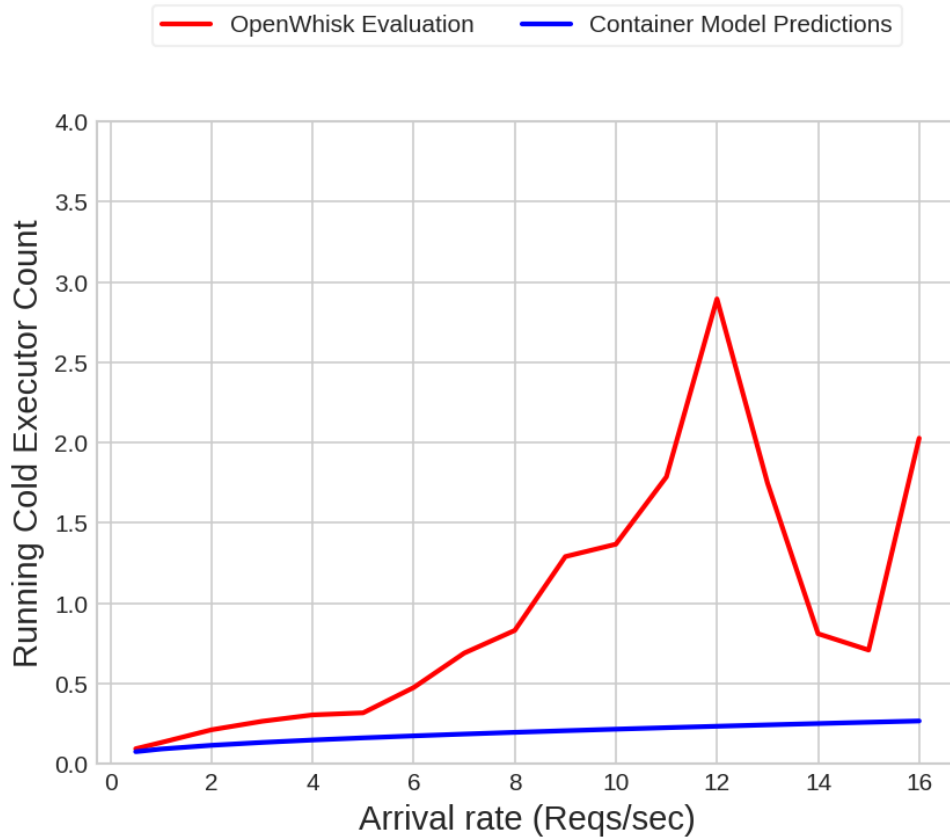


Figure 4.10: Running cold executor counts against arrival rate.

containers than needed when the system is overloaded.

Figure 4.11 plots the system’s running executors. The graph follows a similar pattern to the response time graph. This pattern is expected because the number of executors created is influenced by the function’s response time, given that the serverless platform uses a scale-per-request autoscaling strategy. The number of warm containers primarily determines the number of running executors, as the workload predominantly undergoes the warm start phase (as also observed in Figures 4.10 and 4.9). The number of running executors in Spin is roughly 3x more than the container executors in OpenWhisk. This is consistent with the observation that the response time in Wasm for the function is about three times slower than native execution. The number of running executors in OpenWhisk approaches 35 when the system reaches capacity at 11 requests per second (req/sec), reflecting a number closer to the available system cores. The same can be seen for Spin as it reaches capacity

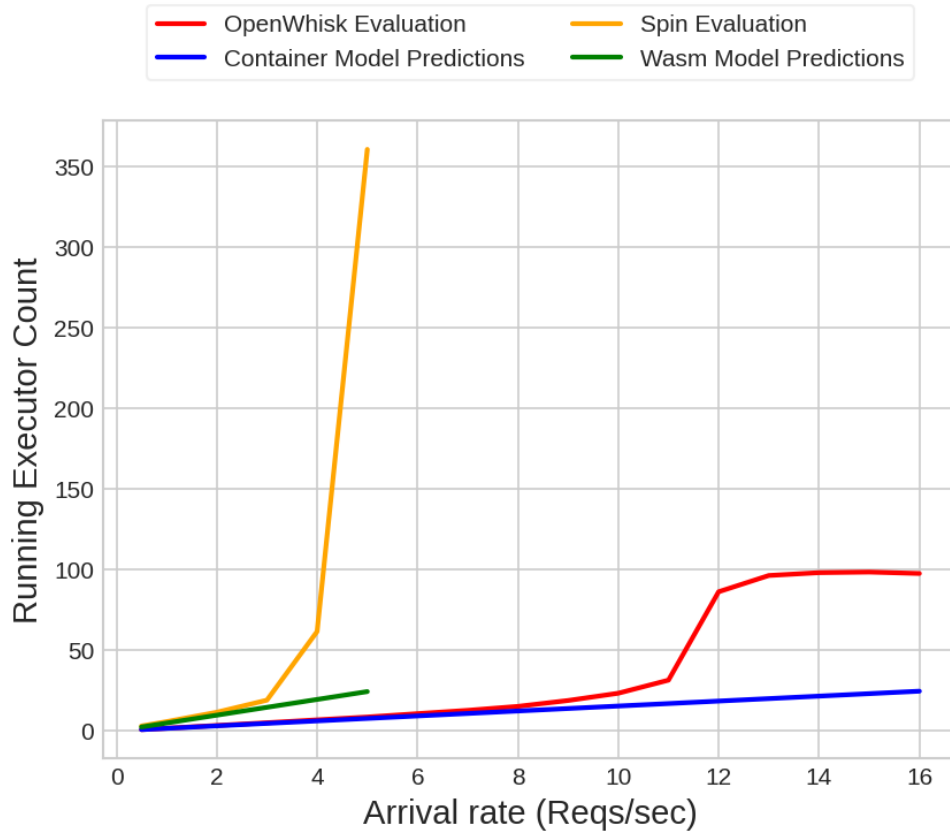


Figure 4.11: Running executor counts against arrival rate.

between an arrival rate of 3 and 4 reqs/sec, where the number of running executors is closer to 40.

Figure 4.12 shows the number of idle executors as the arrival rate varies. The number of idle executors increases with the arrival rate, as more executors are required to handle the workload, leading to a higher number of idle executors. The model reasonably aligns with the experimental results of up to 8 requests per second (req/sec), but beyond this point, the experimental results show approximately 8 more idle containers than predicted by the model. Figure 4.9 also reveals a divergence between the number of running warm executors and the model. This divergence is likely due to the slight increase in the function’s response time from 8 to 10 req/sec caused by contention and the model’s limitation in underestimating idle containers at higher workloads. After reaching capacity at 11 req/sec, the number of idle containers begins to decrease, which is expected as the system becomes overloaded

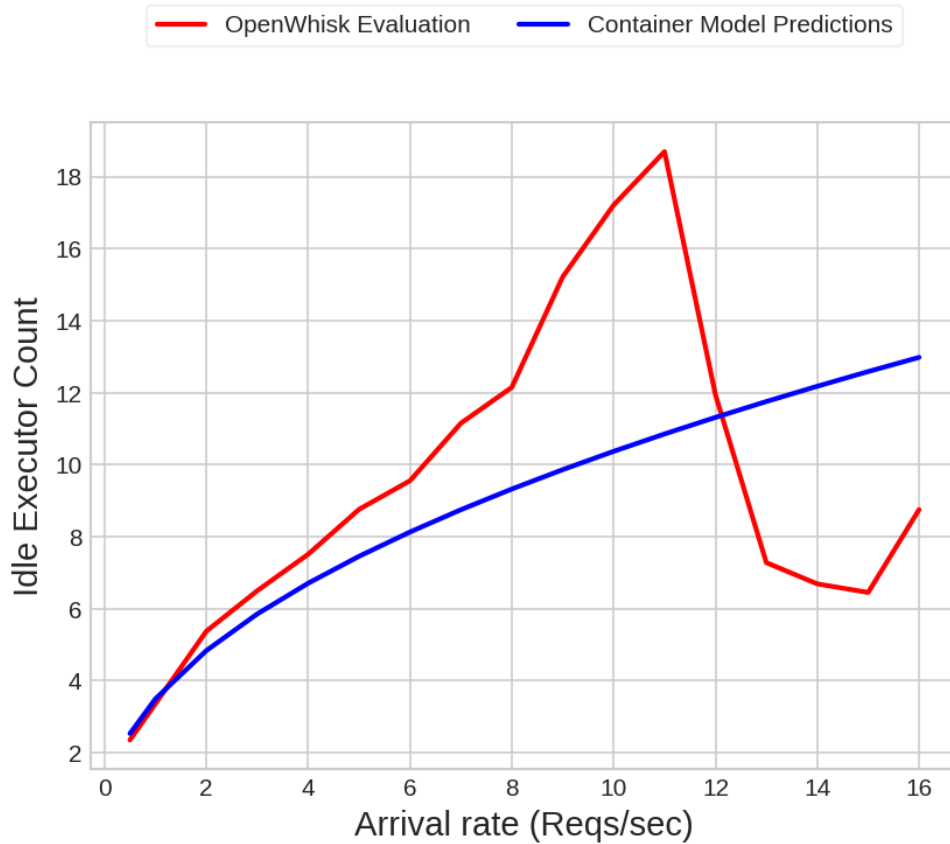


Figure 4.12: Idle executor counts against arrival rate.

and more containers are occupied with serving requests. Notably, at 10 req/sec (when the system is approaching CPU capacity), there are around 17 idle executors compared to 20 busy warm containers, as shown in figure 4.9. This indicates approximately 50% resource efficiency when the system is near capacity for this particular workload.

Figure 4.13 shows the mean number of total executors during the workload. The graph follows a similar pattern to the number of running executors shown in figure 4.12, as the total number of executors on the serverless platform is largely dominated by the number of running warm executors. The main difference is that this graph also includes idle executors, which are significantly fewer than warm executors.

Figure 4.14 plots the cold start probability as the arrival rate increases. The cold start probability is closer to 0 since the keep-alive time is high enough that almost all requests

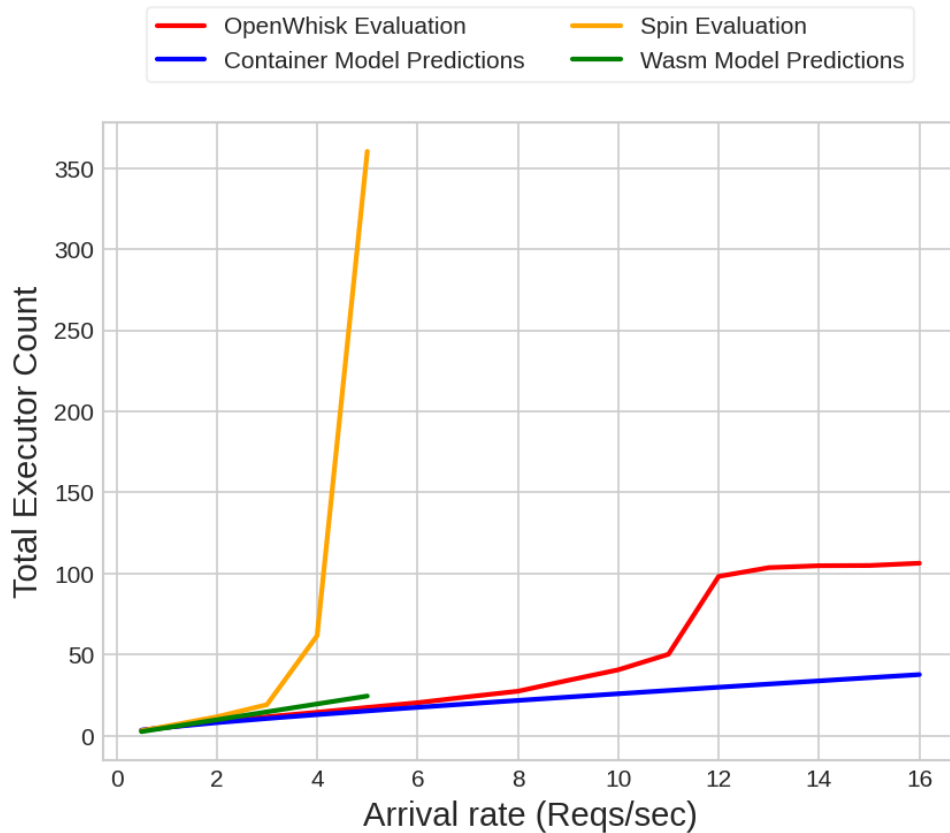


Figure 4.13: Total executor counts against arrival rate.

go through warm starts.

Figures 4.15 through 4.20 display the Wasm model predictions and evaluation results on Spin for the *aes*, *matmul*, and *whatlang* functions. In all three cases, the system reaches CPU capacity when the average number of running executors hits ten, corresponding to the number of system cores. For each workload, the response time decreases to a value lower than the nominal response time, calculated from an average of ten runs. This reduction can be attributed to efficiencies gained from repeated computations and cache hits, which lead to faster execution times. Based on a simple average of ten runs, the model predictions do not account for this decrease, resulting in an overestimation of the number of running executors. These predictions are shown in the left plots of Figures 4.15 through 4.20. In contrast, the right plots, which use the average response times for all arrival rates before reaching capacity to generate model predictions, align more closely with the experimental

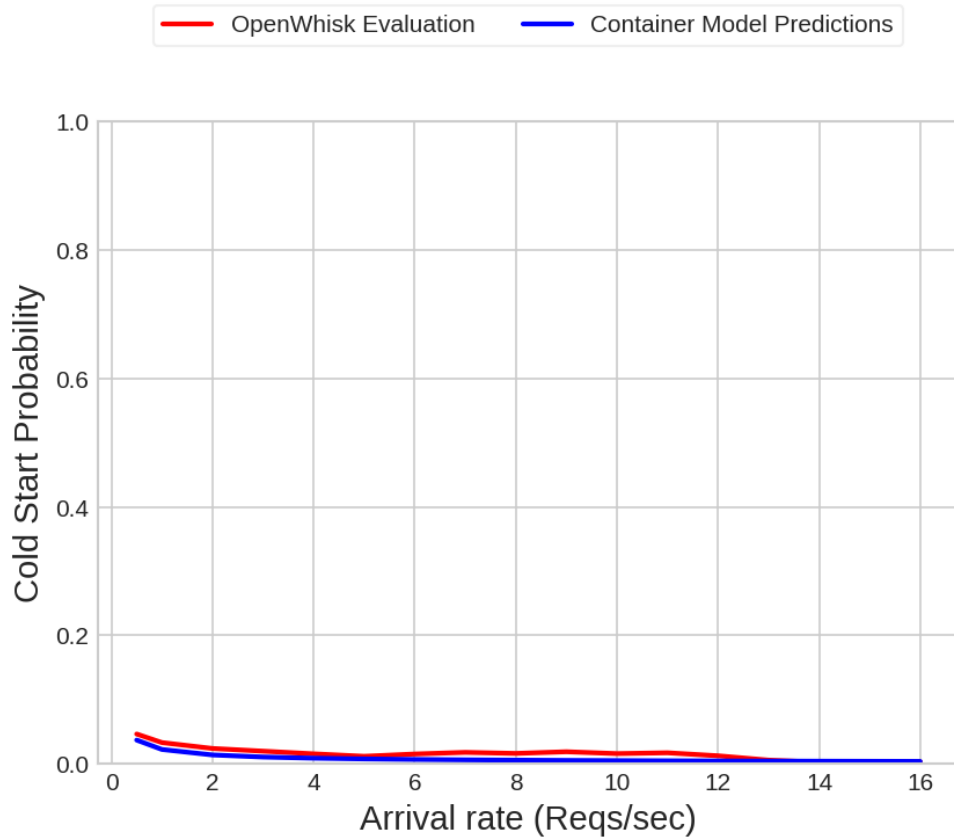


Figure 4.14: Cold start probability against arrival rate..

results. However, the *whatlang* function still shows discrepancies beyond an arrival rate of 200 requests per second (req/sec) as the response time keeps decreasing. The right plot shows that serverless users should use the average response time of the function at the expected workload arrival rate to generate more accurate model predictions, as this approach better reflects actual performance and avoids the overestimation of running executors.

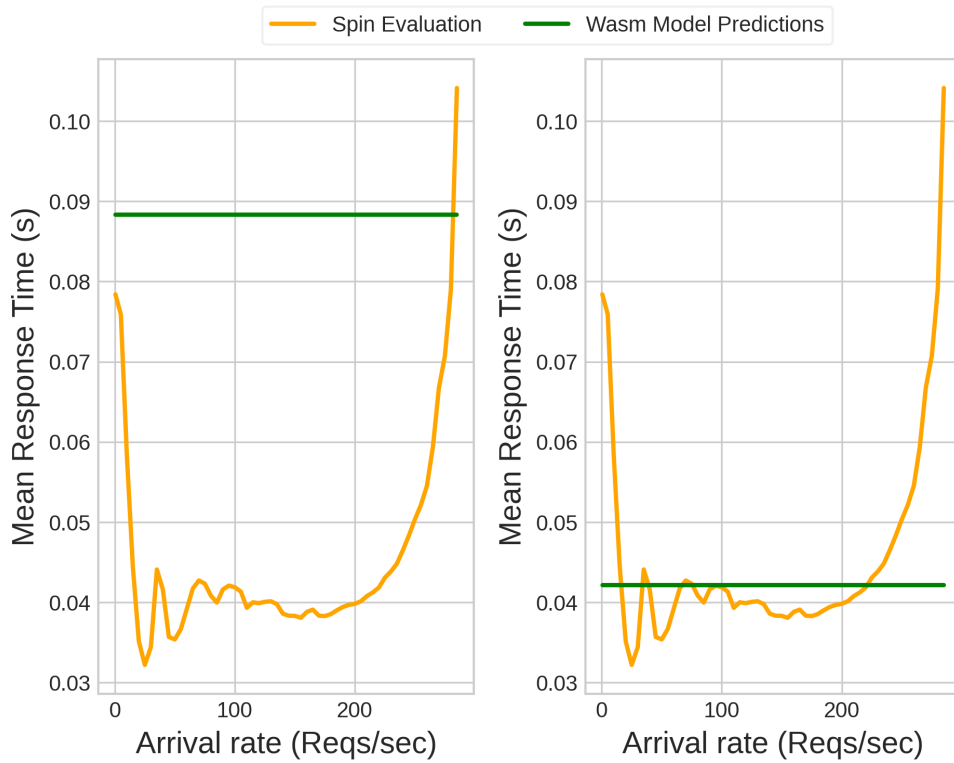


Figure 4.15: Average response time of the *aes* function against arrival time. The left plot shows Wasm model predictions generated using a nominal response time. The right plot uses the average response time of the workload before the capacity is reached to generate the model predictions.

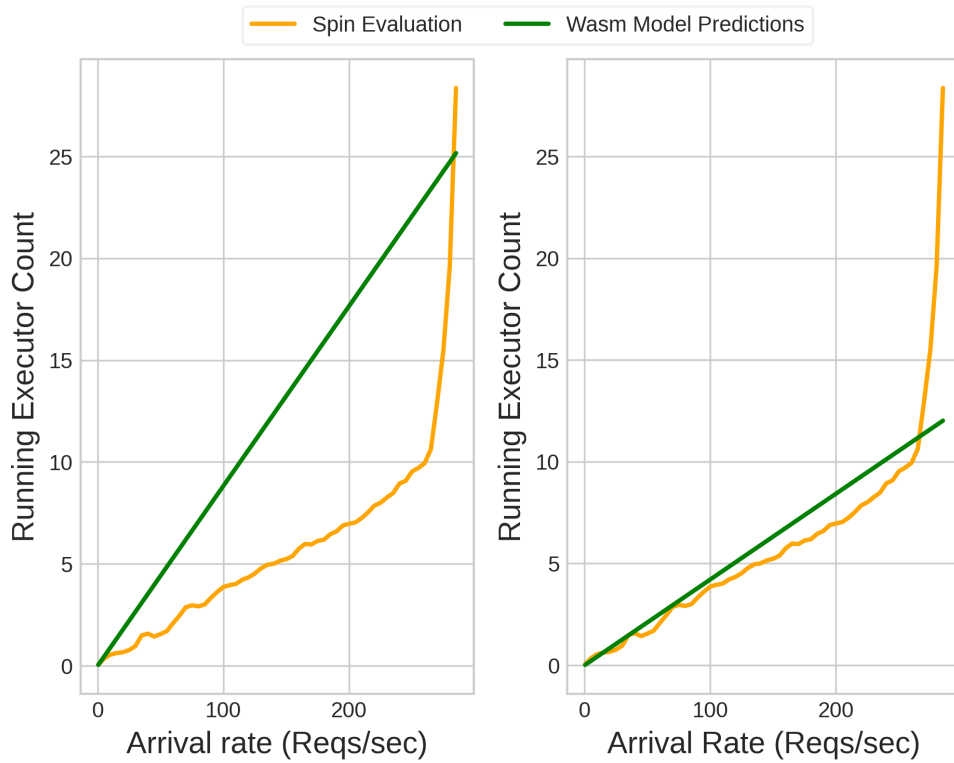


Figure 4.16: Number of running executors during the *aes* function workload against arrival time. The left plot shows Wasm model predictions generated using a nominal response time. The right plot uses the average response time of the workload before the capacity is reached to generate the model predictions.

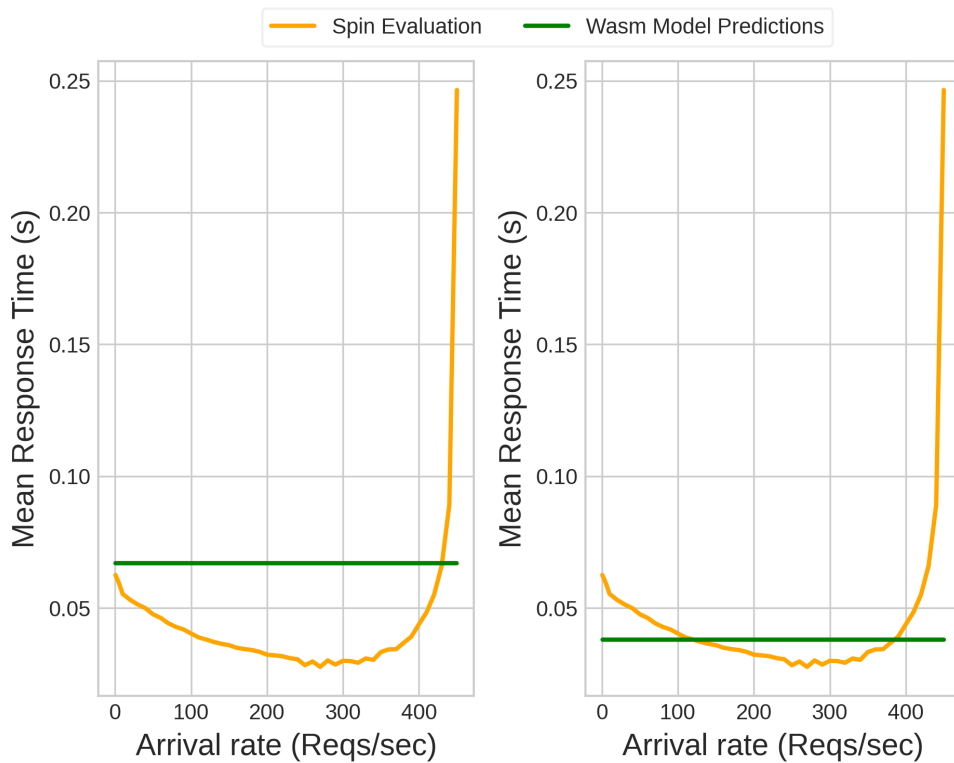


Figure 4.17: Average response time of the *matmul* function against arrival time. The left plot shows Wasm model predictions generated using a nominal response time. The right plot uses the average response time of the workload before the capacity is reached to generate the model predictions.

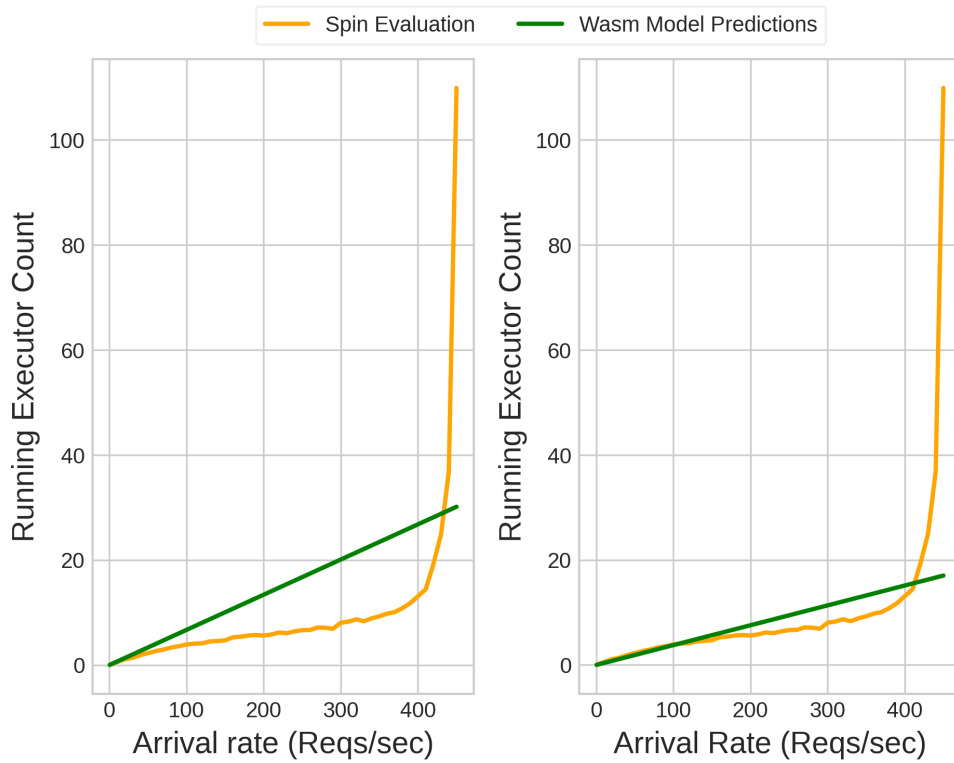


Figure 4.18: Number of running executors during the *matmul* function workload against arrival time. The left plot shows Wasm model predictions generated using a nominal response time. The right plot uses the average response time of the workload before the capacity is reached to generate the model predictions.

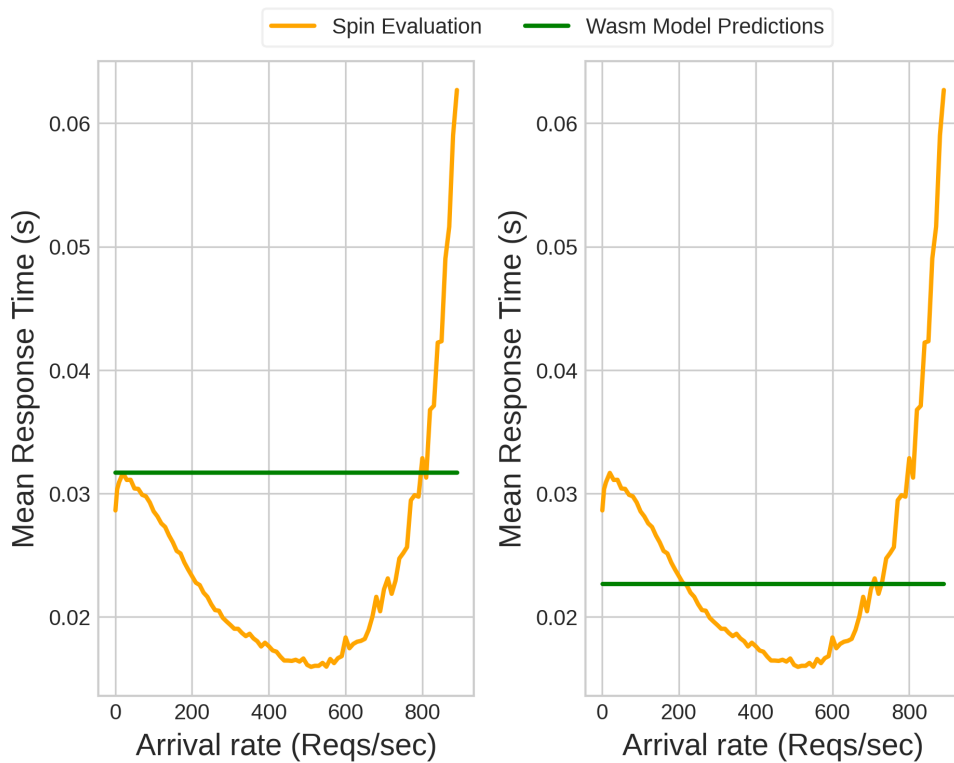


Figure 4.19: Average response time of the *whatlang* function against arrival time. The left plot shows Wasm model predictions generated using a nominal response time. The right plot uses the average response time of the workload before the capacity is reached to generate the model predictions.

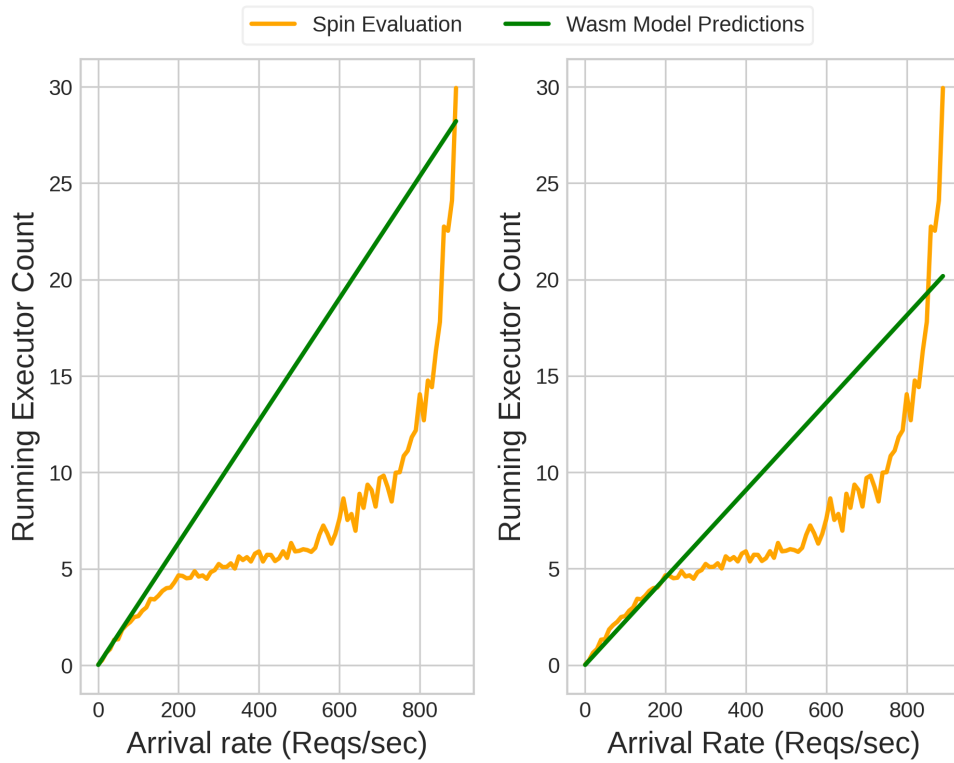


Figure 4.20: Number of running executors during the *whatlang* function workload against arrival time. The left plot shows Wasm model predictions generated using a nominal response time. The right plot uses the average response time of the workload before the capacity is reached to generate the model predictions.

## OpenWhisk Limitations with High Request Arrival Rate

During the load experiment with higher arrival rates, several issues in OpenWhisk caused deviations from the performance model's assumptions, leading to inconclusive results. The model assumes a Poisson distribution for request arrivals, but in OpenWhisk, requests form queues at various points in the platform before reaching the function executor. When these queues are processed, they can send requests in bulk to the Invoker. If the number of requests exceeds the available warm executors, many cold starts occur, with latency significantly higher than the interarrival time of requests. This situation results in the creation of too many function executors in a short period.

Kubernetes imposes a limit of 110 pods per physical system [16]. Consequently, if all 110 executors are busy—whether they are cold or warm starts—any additional requests must wait in a queue because no more than 110 executors can be created. Requests that cannot be serviced within 60 seconds will time out, causing errors and preventing the system from reaching a steady state. The key issues and fixes include:

**Queue Formation at Kafka Message Bus:** A queue formed when the controller sent activation requests to the Invoker component via the Kafka message bus and waited for acknowledgments from Kafka. Delays in receiving acknowledgments caused a queue of requests. Although no direct fix was found, invoking the workload using non-blocking requests, which return an immediate response with a request ID instead of making the client wait for the function's response, circumvented the issue. The client then queries with the request ID later to retrieve the function response and relevant metrics, reducing the queue to a rare occurrence of fewer than eight requests.

**Cache Invalidation Process:** The Controller component periodically creates a short queue of requests while invalidating function code cache entries not accessed in 5 minutes. This process locks the cache, causing requests to wait. After cache invalidation, all requests in the queue are scheduled simultaneously, resulting in a burst of requests to the Invoker component. The solution involved updating the source code to delay the cache invalidation process.

**Slow Kafka Message Consumption:** Even after addressing the above queues, another queue formed due to the slow consumption of Kafka messages by the Invoker component. The Invoker routes function requests to warm executors or cold start new ones if available executors are busy. Slow message consumption led to multiple requests being scheduled together, causing cold starts when there were more requests than available executors.

Figure 4.21 presents the experimental results on OpenWhisk with increasing arrival rate

using the *aes* function. The workload maintains a steady state between 0 and 50 req/sec, during which the model predictions follow with the OpenWhisk results. The response time decreases with higher arrival rates, likely due to system efficiencies such as cache hits. These efficiencies are also consistent with the Wasm results as seen in figure 4.15. However, the model's predictions, based on nominal response times that don't account for the concurrent execution of the same function, are slightly higher. As the arrival rate increases, the experiment shows fewer running warm and idle executors due to the reduced response times. Beyond 50 req/sec, cold starts occur due to request bursts scheduled by the Invoker caused by slow Kafka message consumption. The 'Running Cold Executor Count' plot reflects the increase in cold starts, which correlates with a rise in the 'Total Executors' plot. After 75 req/sec, all requests begin timing out.

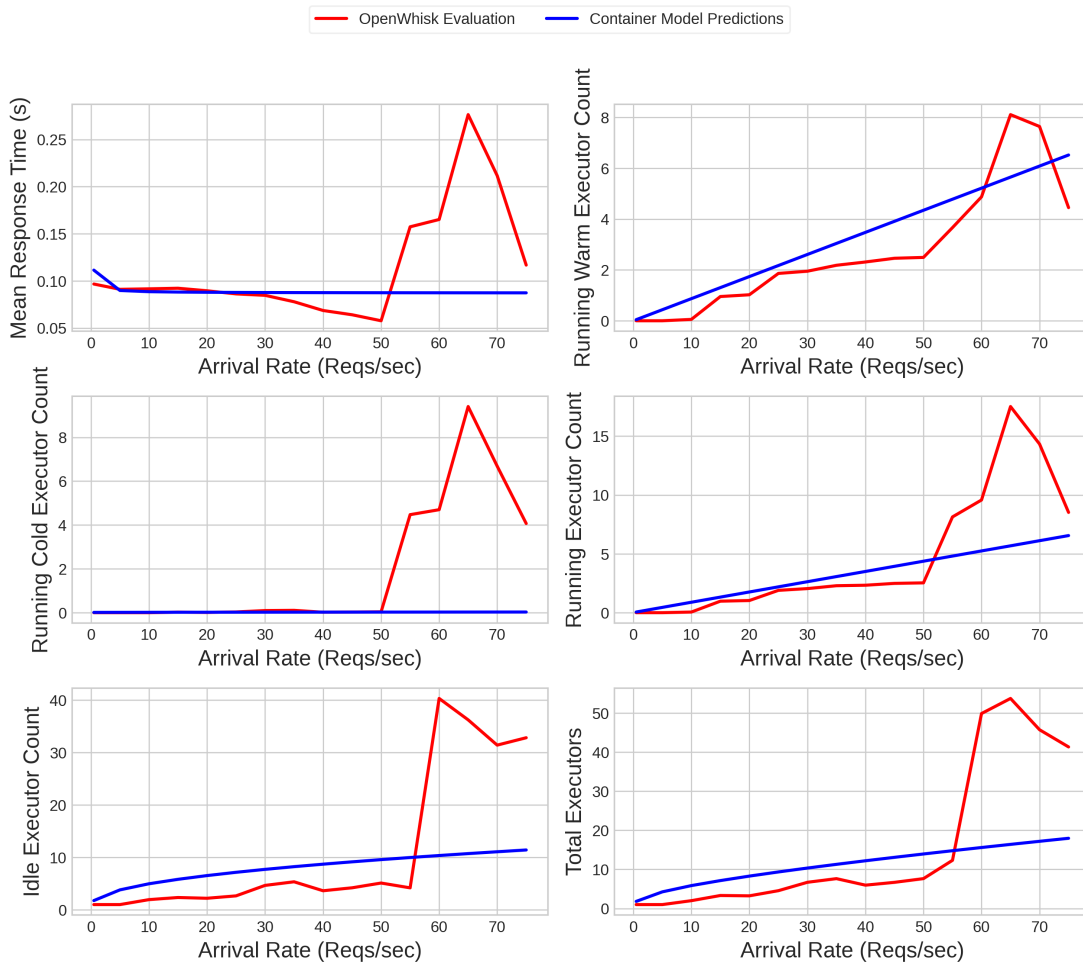


Figure 4.21: Load experiment results on OpenWhisk using the *aes* function.

## 4.4 Discussion and Future Work

The above results show that the performance models provide predictions consistent with the experimental evaluation for the workloads evaluated in this study. Serverless users can utilize these models to predict performance metrics for their workloads under steady-state conditions. The inflection point observed in the total executor count plot (figure 4.6) and the average response time plot (figure 4.1) can guide developers in selecting a keep-alive period that optimizes response time while minimizing resource usage, enabling better cost estimation for a given performance level. Although Wasm has lower computing perfor-

mance than containers, its lower resource footprint makes it cost-effective for serverless users when slower response time is not a concern. Wasm executors deliver better execution times in cases where running the same workload in containers would frequently experience cold starts.

Additionally, both performance models scale reasonably well with increasing load. However, further evaluation is needed for container executors, as the assessment in this thesis was incomplete due to issues encountered with OpenWhisk. Developers should use the average runtime performance at the expected concurrency level of the workload as input to the performance models for more accurate predictions.

Future research can address the identified challenges with OpenWhisk and evaluate under heavier workloads. Additionally, performance models for alternative scheduling patterns in container-based serverless platforms, as discussed in Section 2.1.1, could be developed and assessed using the methodology presented in this thesis. Finally, introducing and evaluating performance models for serverless workflows [18] in containers and Wasm executors represents a valuable direction for future work.

# Chapter 5

## Conclusion

This thesis presents a performance evaluation of containers and Wasm as function executors for serverless platforms. The methodology introduced utilizes analytical performance models to estimate both performance and resource utilization of serverless workloads in the steady state, rather than focusing solely on individual execution comparisons. The performance model helps serverless users find the suitable configuration that optimizes the performance and cost of their serverless application. Serverless providers can leverage the performance model to offer a workload-aware serverless platform that better utilizes system resources and saves infrastructure costs. Through experimental verification of the performance models on open-source serverless platforms, it is shown that the performance models provide fair and reasonable predictions for the workloads evaluated in this study. Serverless users can leverage both performance models to decide whether container-based or WebAssembly-based serverless platforms offer better performance for their specific applications. Additionally, this thesis is one of the first works that evaluates the validity of the container-based performance model on an open-source serverless platform while introducing the performance model for serverless execution in WebAssembly.

# References

- [1] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. Cose: Configuring serverless functions using statistical learning. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, page 129–138. IEEE Press, 2020.
- [2] Amazon Web Services. Amazon ecs and eks with aws fargate. <https://aws.amazon.com/fargate/>, 2024. Accessed: 2024-09-11.
- [3] Apache Software Foundation. Apache OpenWhisk. <https://openwhisk.apache.org/>. Accessed: 2024-07-10.
- [4] Apache Software Foundation. Apache couchdb. <https://couchdb.apache.org/>, 2024. Accessed: 2024-09-11.
- [5] Apache Software Foundation. Apache kafka. <https://kafka.apache.org/>, 2024. Accessed: 2024-09-11.
- [6] Apache Software Foundation. Apache openwhisk. <https://github.com/apache/openwhisk>, 2024. Accessed: 2024-09-11.
- [7] Apache Software Foundation. Apache openwhisk. <https://openwhisk.apache.org/>, 2024. Accessed: 2024-09-11.
- [8] Microsoft Azure. Microsoft azure functions. <https://azure.microsoft.com/en-ca/products/functions>, 2024. Accessed: 2024-07-09.
- [9] Bytecode Alliance. Componentize-py. <https://github.com/bytecodealliance/componentize-py>, 2023. Accessed: October 3, 2024.
- [10] Bytecode Alliance. Wasmtime documentation. <https://docs.wasmtime.dev/>, 2024. Accessed: 2024-09-11.

- [11] Bytecode Alliance. Webassembly component model. <https://component-model.bytecodealliance.org/>, 2024. Accessed: 2024-09-11.
- [12] Xiaolin Chang, Ruofan Xia, Jogesh Muppala, Kishor Trivedi, and Jiqiang Liu. Effective modeling approach for iaas data center performance analysis under heterogeneous workload. *IEEE Transactions on Cloud Computing*, 6:1–1, 01 2016.
- [13] Google Cloud. Google cloud functions. <https://cloud.google.com/functions?hl=en>, 2024. Accessed: 2024-07-09.
- [14] Cloudflare. Cloudflare workers, 2024. Accessed: 2024-07-10.
- [15] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing. Middleware '21, page 64–78, New York, NY, USA, 2021. Association for Computing Machinery.
- [16] Kubernetes Documentation. Considerations for large clusters. <https://kubernetes.io/docs/setup/best-practices/cluster-large/>, 2023. Accessed: 2024-09-15.
- [17] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.
- [18] Simon Eismann, Johannes Grohmann, Erwin van Eyk, Nikolas Herbst, and Samuel Kounev. Predicting the costs of serverless workflows. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, page 265–276, New York, NY, USA, 2020. Association for Computing Machinery.
- [19] Juan José López Escobar, Felipe Gil-Castiñeira, and Rebeca P. Díaz Redondo. Decentralized serverless iot dataflow architecture for the cloud-to-edge continuum. In *2023 26th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pages 42–49, 2023.
- [20] etcd Project. etcd. <https://etcd.io/>, 2024. Accessed: 2024-09-14.
- [21] Fastly. Edge compute, 2024. Accessed: 2024-07-10.

- [22] Fermyon. The next generation of serverless is happening. <https://www.fermyon.com/blog/next-generation-of-serverless-is-happening>, 2024. Accessed: 2024-09-11.
- [23] Fermyon. Spinkube. <https://www.spinkube.dev/>, 2024. Accessed: 2024-09-11.
- [24] Fermyon. Writing apps with spin v2. <https://developer.fermyon.com/spin/v2/writing-apps>, 2024. Accessed: 2024-09-11.
- [25] Fermyon. Github repository: Code things, October 2023.
- [26] Firecracker Project. Firecracker microvm. <https://firecracker-microvm.github.io/>, 2024. Accessed: 2024-09-11.
- [27] Fission Project. Fission. <https://fission.io/>, 2024. Accessed: 2024-09-11.
- [28] Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '21*, page 386–400, New York, NY, USA, 2021. Association for Computing Machinery.
- [29] Philipp Gackstatter, Pantelis A. Frangoudis, and Schahram Dustdar. Pushing serverless to the edge with webassembly runtimes. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 140–149, 2022.
- [30] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. Sledge: A serverless-first, light-weight wasm runtime for the edge. In *Proceedings of the 21st International Middleware Conference, Middleware '20*, page 265–279, New York, NY, USA, 2020. Association for Computing Machinery.
- [31] Google. gvisor: A user-space kernel for containers. <https://gvisor.dev/>, 2024. Accessed: 2024-09-11.
- [32] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C. Nachiappan, Mahmut Taylan Kandemir, and Chita R. Das. Fifer: Tackling resource underutilization in the serverless era. In *Proceedings of the 21st International Middleware Conference, Middleware '20*, page 280–295, New York, NY, USA, 2020. Association for Computing Machinery.
- [33] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. *SIGPLAN Not.*, 52(6):185–200, jun 2017.

- [34] Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. *IoTDI '19*, page 225–236, New York, NY, USA, 2019. Association for Computing Machinery.
- [35] IBM. Ibm cloud functions. <https://cloud.ibm.com/functions>, 2024. Accessed: 2024-09-11.
- [36] Serhii Ivanenko, Jovan Stevanovic, Vojin Jovanovic, and Rodrigo Bruno. Hydra: Virtualized multi-language runtime for high-density serverless platforms, 2024.
- [37] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, Renton, WA, July 2019. USENIX Association.
- [38] Hamzeh Khazaei, Jelena Mistic, and Vojislave B. Mistic. Modelling of cloud computing centers using m/g/m queues. In *2011 31st International Conference on Distributed Computing Systems Workshops*, pages 87–92, 2011.
- [39] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504, 2019.
- [40] Vojdan Kjorveziroski and Sonja Filiposka. Webassembly as an enabler for next generation serverless computing. *Journal of Grid Computing*, 21(3):34, Jun 2023.
- [41] Vojdan Kjorveziroski and Sonja Filiposka. Webassembly orchestration in the context of serverless computing. *Journal of Network and Systems Management*, 31(3):62, Jul 2023.
- [42] Vojdan Kjorveziroski, Sonja Filiposka, and Anastas Mishev. Evaluating webassembly for orchestrated deployment of serverless functions. In *2022 30th Telecommunications Forum (TELFOR)*, pages 1–4, 2022.
- [43] Knative Project. Knative documentation. <https://knative.dev/docs/>, 2024. Accessed: 2024-09-11.
- [44] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, Bingsheng He, and Minyi Guo. The serverless computing survey: A technical primer for design architecture. *ACM Comput. Surv.*, 54(10s), sep 2022.

- [45] Xuanzhe Liu, Jinfeng Wen, Zhenpeng Chen, Ding Li, Junkai Chen, Yi Liu, Haoyu Wang, and Xin Jin. Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing. *ACM Trans. Softw. Eng. Methodol.*, 32(5), jul 2023.
- [46] Ju Long, Hung-Ying Tai, Shen-Ta Hsieh, and Michael Juntao Yuan. A lightweight design for serverless function as a service. *IEEE Software*, 38(1):75–80, 2021.
- [47] Nima Mahmoudi and Hamzeh Khazaei. Performance modeling of metric-based serverless computing platforms, 2022.
- [48] Nima Mahmoudi and Hamzeh Khazaei. Performance modeling of serverless computing platforms. *IEEE Transactions on Cloud Computing*, 10(4):2834–2847, 2022.
- [49] Saif U. R. Malik, Samee U. Khan, and Sudarshan K. Srinivasan. Modeling and analysis of state-of-the-art vm-based cloud management platforms. *IEEE Transactions on Cloud Computing*, 1(1):1–1, 2013.
- [50] Pankaj Mendki. Evaluating webassembly enabled serverless approach for edge computing. In *2020 IEEE Cloud Summit*, pages 161–166, 2020.
- [51] Michael Kerrisk. Linux programmer’s manual: cgroups(7). <https://man7.org/linux/man-pages/man7/cgroups.7.html>, 2024. Accessed: 2024-09-11.
- [52] Michael Kerrisk. Linux programmer’s manual: namespaces(7). <https://man7.org/linux/man-pages/man7/namespaces.7.html>, 2024. Accessed: 2024-09-11.
- [53] NGINX, Inc. Nginx. <https://nginx.org/en/>, 2024. Accessed: 2024-09-11.
- [54] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with Serverless-Optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association.
- [55] Bernt Oksendal. *Markov Processes for Stochastic Modeling*. Academic Press, 2013. Accessed: 2024-07-10.
- [56] OpenFaaS. Openfaas. <https://www.openfaas.com/>, 2024. Accessed: 2024-09-11.
- [57] Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, Matteo Trentin, and Gianluigi Zavattaro. Funless: Functions-as-a-service for private edge cloud systems, 2024.

- [58] Pillow Contributors. Pillow. <https://pypi.org/project/pillow/>, 2024. Accessed: 2024-09-11.
- [59] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. <https://web.cs.ucla.edu/~pouchet/software/polybench/>, 2024. Accessed: 2024-09-11.
- [60] Pyperformance. Python performance benchmark suite, 2017. Accessed on 2023-10-23.
- [61] Python Software Foundation. asyncio — asynchronous i/o. <https://docs.python.org/3/library/asyncio.html>, 2024. Accessed: 2024-09-11.
- [62] Haiyang Qian, Deep Medhi, and Kishor Trivedi. A hierarchical model to evaluate quality of experience of online services hosted by cloud computing. In *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*, pages 105–112, 2011.
- [63] Amazon Web Services. Aws lambda. <https://aws.amazon.com/lambda/>, 2024. Accessed: 2024-07-09.
- [64] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. Serverless computing: A survey of opportunities, challenges, and applications. *ACM Comput. Surv.*, 54(11s), nov 2022.
- [65] Mohammad Shahrads, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. 10 2019.
- [66] Mohammad Shahrads, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.
- [67] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433. USENIX Association, 2020.
- [68] Benedikt Spies and Markus Mock. An evaluation of webassembly in non-web environments. In *2021 XLVII Latin American Computing Conference (CLEI)*, pages 1–10, 2021.
- [69] Second State. Wasm-learning: Webassembly learning materials, Year or Date Accessed.

- [70] Kyle M. Tarplee, Anthony A. Maciejewski, and Howard Jay Siegel. Robust performance-based resource provisioning using a steady-state model for multi-objective stochastic programming. *IEEE Transactions on Cloud Computing*, 7(4):1068–1081, 2019.
- [71] Fermion Technologies. Develop serverless webassembly apps with spin, 2024. Accessed: 2024-07-10.
- [72] Rafael Tolosana-Calasanz, Gabriel Castañé, José Bañares, and Omer Rana. *Modelling Serverless Function Behaviours*, pages 109–122. 12 2021.
- [73] VMware. Kubeless. <https://github.com/vmware-archive/kubeless>, 2024. Accessed: 2024-09-11.
- [74] Zhen Wang, Jianda Wang, Zhendong Wang, and Yang Hu. Characterization and implication of edge webassembly runtimes. In *2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, pages 71–80, 2021.
- [75] WASI Project. Wasi: Webassembly system interface. <https://wasi.dev/>, 2024. Accessed: 2024-09-11.
- [76] WasmEdge Project. Wasmedge: Webassembly runtime for cloud native, edge, and decentralized applications. <https://wasmedge.org/>, 2024. Accessed: 2024-09-11.
- [77] Wasmer, Inc. Wasmer. <https://wasmer.io/>, 2024. Accessed: 2024-09-11.
- [78] WAVM Project. Wavm: Webassembly virtual machine. <https://wavm.org/>, 2024. Accessed: 2024-09-11.
- [79] WebAssembly. Webassembly binary toolkit (wabt). <https://github.com/WebAssembly/wabt>, 2024. Accessed: 2024-09-11.
- [80] Kaiqi Xiong and Harry Perros. Service performance and analysis in cloud computing. In *2009 Congress on Services - I*, pages 693–700, 2009.
- [81] Bo Yang, Feng Tan, Yuan-Shun Dai, and Suchang Guo. Performance evaluation of cloud service considering fault recovery. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, page 571–576, Berlin, Heidelberg, 2009. Springer-Verlag.

# APPENDICES

# Appendix A

## Appendix

### A.1 Model Modification Effect

Section 3.2.1 discusses a modification to the container-based performance model that introduces a fair calculation for the inter-arrival time of requests to warm executors. Figure A.1 illustrates the impact of this modification in comparison to the original calculation method used in the referenced paper [48]. The figure shows the number of idle containers and the total number of containers in the steady-state for the *float* function when the keep-alive time is varied. The modified performance model provides a closer prediction than the unmodified model for the experimental result. Although the effect of the change is less significant, the change offers a closer estimation of the number of idle containers. The original paper underestimates the expected inter-arrival time between warm requests, leading to a shorter predicted average lifespan of a warm executor and, consequently, an underestimated number of idle containers. Since the change introduced always includes the warm processing time of the function, it provides a fair prediction of the average lifespan of the warm executor.

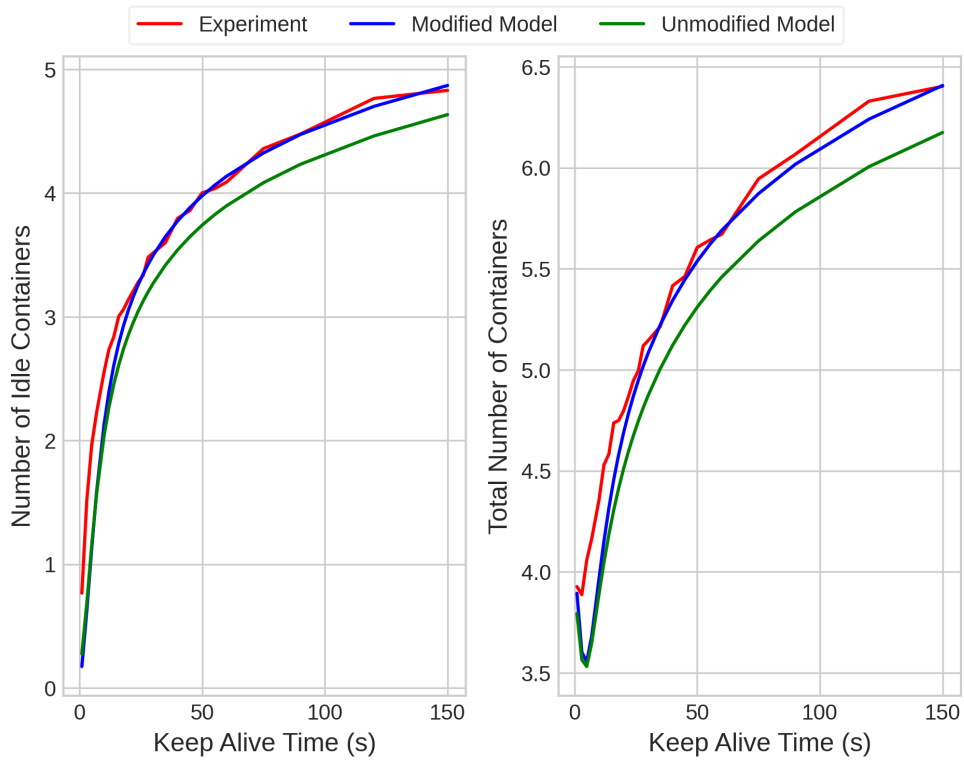


Figure A.1: The modified performance model used in the thesis is closer to the experimental evaluation on OpenWhisk.