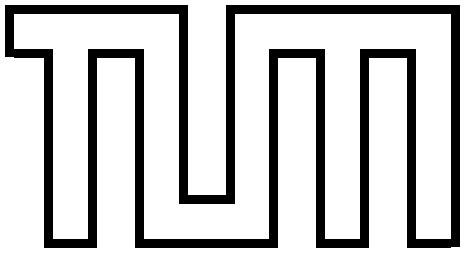


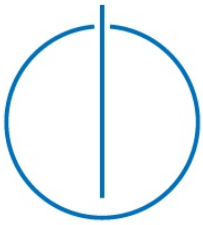
Technische Universität München

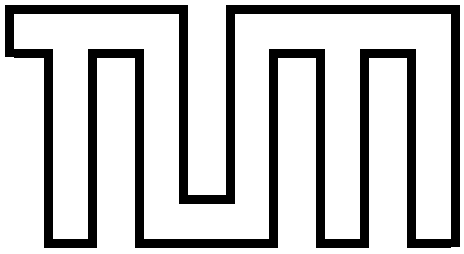
Fakultät für Informatik

Master's Thesis in Informatik

Apache OpenWhisk Serverless Platform: Benchmarking and Analysis

Aleksandr Kuntsevich





Technische Universität München

Fakultät für Informatik

Master's Thesis in Informatik

Apache OpenWhisk Serverless Platform: Benchmarking and Analysis

Apache OpenWhisk Serverless Plattform: Benchmarking und Analyse

Author: Aleksandr Kuntsevich

Supervisor: Prof. Dr. Hans-Arno Jacobsen

Advisor: Pezhman Nasirifard

Submission: 08.11.2018

I confirm that this master's thesis is my own work and I have documented all sources and material used.

München, 08.11.2018

(Aleksandr Kuntsevich)

Abstract

The serverless computing simplifies the development and deployment of scalable web applications, by delegating most of the operational concerns to the cloud providers. Apache OpenWhisk is a serverless platform employed by IBM Cloud. Despite the apparent benefits of serverless computing, some limitations of the serverless platform, such as a stateless nature of serverless functions, can introduce scalability bottlenecks. This thesis proposes an analysis and benchmarking approach for investigating potential bottlenecks and limitations of the Apache OpenWhisk serverless platform and provides achieved results.

Inhaltsangabe

Serverless computing vereinfacht die Entwicklung und Bereitstellung von skalierbaren Webanwendungen, indem die meisten betrieblichen Probleme an die Cloud-Anbieter delegiert werden. Apache OpenWhisk ist eine Serverless Plattform, die in der IBM Cloud eingesetzt wird. Trotz der offensichtlichen Vorteile des Serverless Computing, können einige Einschränkungen der Serverless-Plattform, zum Beispiel die Tatsache, dass die Serverless Funktionen stateless sind, Engpässe bei der Skalierbarkeit verursachen. Diese Masterarbeit schlägt einen Ansatz für Analyse- und Benchmarking vor, um mögliche Engpässe und Einschränkungen der Apache OpenWhisk Serverless-Plattform zu untersuchen und erzielte Ergebnisse zu liefern.

Acknowledgment

A demo paper "A Distributed Analysis and Benchmarking Framework for Apache OpenWhisk Serverless Platform" was created throughout the work on this thesis, authored by me, my advisor Pezhman Nasirifard and my supervisor Prof. Dr. Hans-Arno Jacobsen. Parts of the demo paper are used in this thesis.

Contents

List of Figures

List of Tables

Abbreviations

API	Applicaion programming interface.
BTU	Billable time unit.
CPU	Central processing unit.
CRUD	Create, Read, Update, Delete (about basic database operations).
FaaS	Function as a Service.
GB	Gigabyte.
HTTP	HyperText transfer protocol.
MB	Megabyte.
RAM	Random access memory.
REST	Representational state transfer.
SLA	Service level agreement.

SQL	Structured query language.
SSL	Secure sockets layer.
UML	Unified modeling language.

Chapter 1

Introduction

Serverless computing is a *Function-as-a-Service (FaaS)* concept that simplifies the development and deployment of scalable server applications by delegating most of the operational concerns to the service provider. The underlying idea behind it is that a developer only has to worry about the actual business logic of the application, expressed in a so-called function, while deployment, maintenance, scaling of the application are completely out of the scope of his or her responsibility. The cloud vendor is meanwhile to take care of provisioning the server and deploying the function on it. Another advantage that is typical for such platforms is that a user only has to pay for the actual invocations of the function, which means that an idling function costs nothing, while in the world of more traditional solutions it's more common to pay a flat rate price for a virtual machine on a hosting provider. [3]

Apache OpenWhisk is one of the platforms for serverless computing. It is developed by IBM as a base for a commercial serverless solution called *IBM Functions* [3]. However, Apache OpenWhisk itself is an open-source project available for use free of cost, as well as open for further contributions. The goal of this thesis is to evaluate and investigate the behavior of this platform.

1.1 Motivation

Since serverless computing is getting more popular in the tech world and draws more attention of people related to the software engineering industry, it is important to have the understanding of how serverless platforms perform in different circumstances to know whether it is a suitable tool for a particular problem. As IBM Functions and the underlying

Apache Openwhisk platform are one of the most well-known serverless platforms, more and more users start to rely on it. However, there is a lack of sufficient research in this field. As described in Chapter 3, a number of studies targeted comparison of different serverless platforms, however, none of them focus specifically on features of Apache Openwhisk.

1.2 Problem Statement

Although serverless paradigm offers several significant advantages in facilitating highly available and scalable distributed applications, the limitations, such as the short-lived stateless nature of the serverless functions, can cause scalability bottlenecks. To compensate for lack of state of the serverless functions, integrating other cloud resources, such as Database-as-s-Services is a practiced solution, but the scalability of external resources affects the scalability of the serverless application [4]. Furthermore, the automatic scaling offered by OpenWhisk is not predictable by the user, which can cause bottlenecks. For example, when the function is scaled down, the *cold start* problem causes latency issues [5]. Investigating such possible bottlenecks and limitations is important for the understanding of how the platform behaves in different circumstances, including various edge cases. To achieve this goal, it is necessary to implement a set of test cases which cover various potential use-cases of the platform, come up with an approach to set up the platform and implement a suitable solution for performance testing, as well as evaluate the results and come to conclusions.

1.3 Approach

This thesis' work is divided into multiple stages. Firstly, distributed setups of Apache Openwhisk and Kubernetes were installed on a private cluster, with the same amount of resources allocated for each of the platforms, which gives an opportunity to investigate scaling issues, and also compare the performance of the serverless platform with a traditional server-based solution. Next, a number of serverless functions were implemented which simulate real functions that are used by users of serverless platforms: CPU and RAM intensive, functions relying on an external web-resource or a database query. Apart from that, a performance testing tool was implemented, which allows performing different test cases by executing custom numbers of concurrent requests to the platform and gather statistics, such as latency, CPU and RAM usage. Furthermore, a set of Jupyter notebooks was created to perform transformations of the statistics and create graphs, which give a

clear understanding of how the system behaves in a certain test case.

1.4 Organization

The thesis is organized in the following way: Chapter 2 introduces the general concepts of the Apache Openwhisk platform and Kubernetes, which is essential for understanding the further chapters. Chapter 3 provides information about related studies which also targeted performance of various serverless platforms and elaborates on how this thesis is different from the existing research. Chapter 4 provides a detailed description of the platform setup, implementation of the performance testing application and approach to testing various aspects of Apache Openwhisk. Chapter 5 presents the evaluation of results, including graphs. Chapter 6 is a summarization and conclusion of the work.

Chapter 2

Background

2.1 Apache OpenWhisk

Apache OpenWhisk is an event-driven platform for serverless computing, that runs code in response to events or invocations. It is an open-source project and built on top of other existing solutions, such as Nginx, Kafka, Docker, CouchDB. [2]

The following subsections explain in detail how Apache OpenWhisk is handling actions, what components it consists of, as well as what options for a custom deployment of the solution are available.

2.1.1 Actions and namespaces

Apache Openwhisk includes various entities, such as actions, triggers, rules, packages, namespaces. For the further understanding of the thesis content, it's important to know what actions and namespaces are.

Action is essentially an alias for a serverless function, which is stored internally by Apache OpenWhisk. An action can be created by a user via REST API of the platform, or via *WSK-CLI* - a command-line utility enabling interaction between the user and the platform. Apache OpenWhisk supports actions in a large number of programming languages, including Javascript and Java which are used in this work. Creating an action requires calling a certain CLI command and providing the path to the source code of the function to it. An action can be invoked by executing a corresponding CLI command with the action name and required parameters provided to it.

limit	description	configurable	unit	default
timeout	a container is not allowed to run longer than N milliseconds	per action	milliseconds	60000
memory	a container is not allowed to allocate more than N MB of memory	per action	MB	256
logs	a container is not allowed to write more than N MB to stdout	per action	MB	10
concurrent	no more than N activations may be submitted per namespace either executing or queued for execution	per namespace	number	100
minuteRate	no more than N activations may be submitted per namespace per minute	per namespace	number	120
codeSize	the maximum size of the actioncode	not configurable, limit per action	MB	48
parameters	the maximum size of the parameters that can be attached	not configurable, limit per action/package/trigger	MB	1
result	the maximum size of the action result	not configurable, limit per action	MB	1

Figure 2.1: Limitations of Openwhisk actions [1]

Another important entity of Apache Openwhisk is *namespace*. A namespace is always related to a specific user, all the actions as well as other entities created and managed by a user belong to a corresponding namespace.

According to [1], there is a number of inbuilt limitations, which are bound to actions or namespaces. Figure 2.1 shows the list of technical limitations and the details. The following limitations are especially important:

- Timeout: with intensive inputs or a large number of requests an invocation might exceed the timeout and hence limit the maximum load the system can be tested with. The maximum possible timeout an action can be configured with is 300000 seconds.
- Memory: performance of the platform on memory-intensive functions can highly vary depending on how much RAM is allocated for one action, which is going to be tested later as a part of the evaluation. The maximum RAM an action can be configured with is 512MB.
- Concurrency: defines how many concurrent requests are accepted from each namespace (i.e. user). It is crucial to increase this limit due to the fact that the

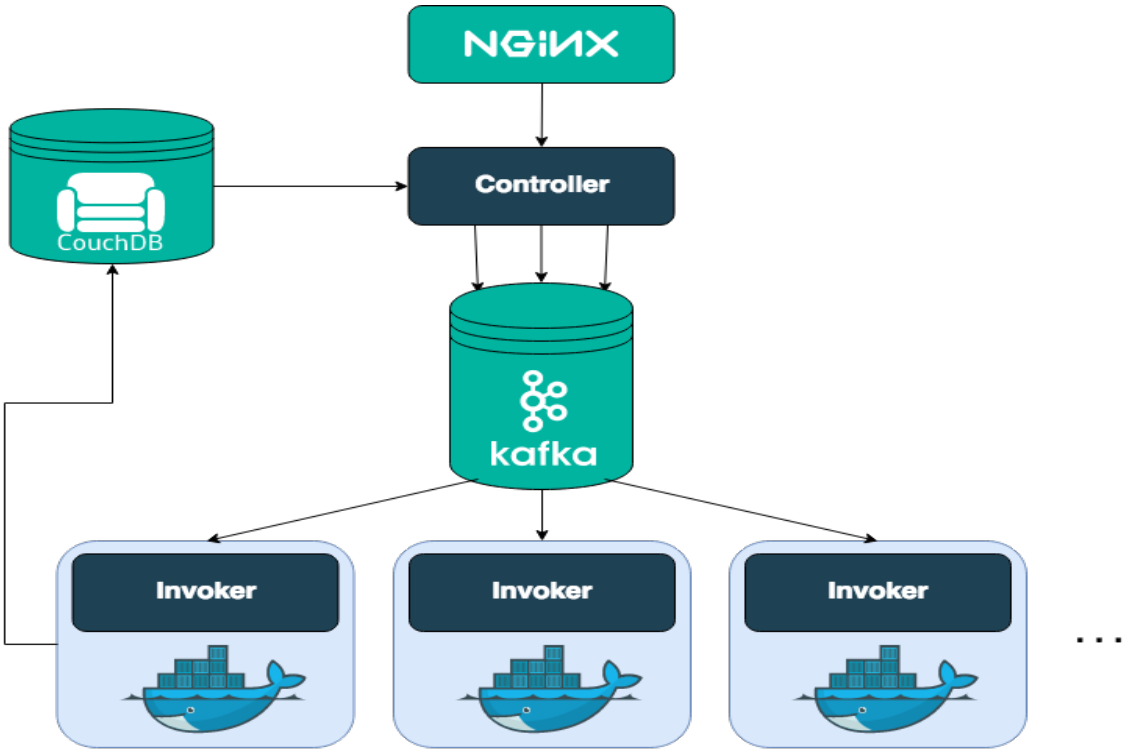


Figure 2.2: Openwhisk: internal flow [2]

default value of 100 is too low to perform very intensive testing scenarios with a large number of concurrent requests.

- Payload size: the content of a POST request cannot exceed a certain limit, which is 1MB and cannot be changed.

2.1.2 High-level architecture

Figure 2.2 presents the high-level architecture of Apache Openwhisk, as well as the internal flow of how it processes actions [2].

Nginx is a web server, which is mainly used for SSL termination and forwarding appropriate HTTP calls to the next component. [2]

Controller is the next point which receives requests forwarded by Nginx. It serves as a REST API for all the actions a user can perform with the platform, including authentication, CRUD operations with entities, action invocations. It also has an inbuilt load balancer, which monitors the state of available invokers, and chooses the right invoker to execute an action. [2]

CouchDB is a NoSQL database. It is used to store user credentials, entities including actions, parameters, resource restrictions, results of invocations. [2]

Kafka is used as a bus between the Controller and Invokers. After an action is invoked, Controller publishes a message to Kafka, addressed to a dedicated invoker chosen by the load balancer. Even if at some point of time invokers are under too much load, the message will still be read from a topic by a chosen invoker and a function will be executed, once other invocations are finished. [2]

Invoker is a unit which directly invokes an action (i.e. a serverless function). It uses Docker to execute functions in a safe isolated way. Once a function invocation is requested, the invoker is starting a corresponding Docker container (e.g. Node.JS to execute a Javascript action) and injecting the code of the executed function. [2]

2.1.3 Deployment

The following options are available to deploy Apache OpenWhisk:

- A *Vagrant*[1] machine to create a system setup locally.
- *Ansible*[2] playbooks provide a way to deploy an OpenWhisk setup on one machine, as well as on multiple machines in a distributed way. The playbooks are structured in a way that allows deployment and re-deployment of every single component of the platform.
- Furthermore, Apache OpenWhisk can be installed on *Kubernetes* - in particular on Minikube, on a cluster provisioned by a public cloud provider or on a cluster managed privately. [6]

2.2 Kubernetes

"*Kubernetes* is an open source system for automating deployment, scaling and management of containerized applications" [7]. In the scope of this thesis it is only important to know that this platform provides a possibility to automatically deploy applications packed in a Docker container, by creating the following two entities:

[1] `https://www.vagrantup.com/`
[2] `https://www.ansible.com/`

- Deployment, that specifies which docker image should be used to deploy an application,
- Service, that defines how the deployed application should be exposed to external users.

The platform then automatically manages load balancing between workers without making a developer worry about it. A Kubernetes cluster is provided by various cloud platforms, but can also be installed directly on private bare metal.

Chapter 3

Related Work

Serverless computing is a relatively new topic and therefore the amount of existing research and work related to it is limited. However, a number of studies researched on serverless, its open problems and approaches for benchmarking and analysis of serverless platforms.

3.1 Limitations of serverless

[5] gives an overview of problems that certain aspects of serverless are prone to. For example, the *cold start* problem caused by the fact that while the function is idle it is scaled to zero, which is, on the one hand, an advantage because the user only pays for actual invocations of the function, but on the other hand leads to a larger unpredictable delay when the function is again executed for the first time. Apart from that any serverless platform always has certain *resource limits*, such as CPU, memory or bandwidth, which are needed to ensure that the platform is able to handle high loads and spikes. Another challenging problem is *scaling*, which can behave in an unpredictable for the user way. According to [8] FaaS systems currently lack a systematic objective benchmark due to the fact that such systems are inherently complex, include more operational logic such as underlying autoscaling mechanisms, have aspects, which are difficult to quantify. [8] defines the future implementation of an objective benchmark for cloud providers as an important goal.

3.2 Benchmarking of cloud platforms

A few studies made attempts to benchmark various serverless platforms from different perspectives. E.g. [9] implemented a microbenchmark to compare Function as a Service solutions. The researchers used Fourier transformation, matrix multiplication, and sleep functions to evaluate the behavior of platforms depending on how CPU and memory intensive executed functions are. Apache OpenWhisk was chosen as a baseline for the comparison between solutions. The process of evaluation consists of executing serverless functions from a local machine and measuring the latency of the function execution, based on different parameters, such as sleep time or matrix dimension. The study concluded that IBM Cloud Functions based on Apache Openwhisk is the most cost-effective solution, however, it does not perform well with high inputs for the Fourier function and multiplication of matrices of very high dimensions. [10] also presented micro-benchmarking approach for different providers in order to compare them.

[11] presented a performance analysis of a few cloud platforms - Amazon EC2, GoGrid, ElasticHosts, and Mosso - from the perspective of many-task scientific computing applications. [12] implemented a benchmarking suite which can execute and gather performance results of heterogeneous cloud function benchmarks over a long period of time. The suit periodically executes selected benchmarks, which are later available for examination. This solution targets a few popular cloud function providers, such as AWS Lambda, IBM OpenWhisk, Azure Functions, Google Cloud Functions. All cloud providers were tested with two types of CPU intensive benchmarks written in C and executed within Javascript wrappers. The results of the benchmarking mainly refer to the latency and resource allocation depending on the intensity of benchmarking functions.

Both [13] and [14] aim to benchmark and compare cloud platforms with respect to performance and cost, focusing on concurrency and latency. [13] provided relevant results regarding the function throughput per second depending on concurrent invocations of CPU and disk intensive functions for various cloud platforms. Besides, [14] presented an approach to investigate differences between the cold and warm run performance, however, neither IBM Functions nor the Apache OpenWhisk platform were the target of the performance tests.

Furthermore, IBM proposed a solution called SPECserverless [15] with the objective of creating standard tests for defining the baseline performance of different serverless platforms. The framework uses four types of serverless functions: CPU intensive, memory intensive, as well as jobs requiring a database and network connections. A number of relevant parameters were taken into consideration, such as frequency of invocation, size

of payload and concurrency of executed jobs. Apart from the technical capabilities of the platforms, the research also targeted vendor related matters, e.g. SLA, price, dashboard interface etc.

3.3 Focus of the thesis

All the presented related studies approach the issue from the perspective of comparison of different cloud platforms. However, the main focus of this work is to provide a deeper insight into particular features of the Apache OpenWhisk platform, such as scaling in a distributed environment and the cold start.

Chapter 4

Approach

The general approach of this thesis is to come up with a set of test cases based on multiple serverless functions to test the Apache Openwhisk platform, interpret the results, compare the system with a traditional server-based solution and possibly detect existing bottlenecks and limitations in the system. Furthermore, another part of the thesis is the implementation of a software which enables automation of these tests and retrieving statistics.

4.1 High-Level Architecture

Figure 4.1 presents the high-level component diagram of the final solution. The whole testing framework can be broken down into three main parts:

- Openwhisk: a custom distributed setup of the Apache Openwhisk platform,
- Kubernetes: a Kubernetes cluster, which is hosting a server-based application,
- Tester Application: an application, which allows executing various test cases against the Apache OpenWhisk platform, as well as Kubernetes, and gather relevant statistics.

Further sections explain each of these aspects in detail.

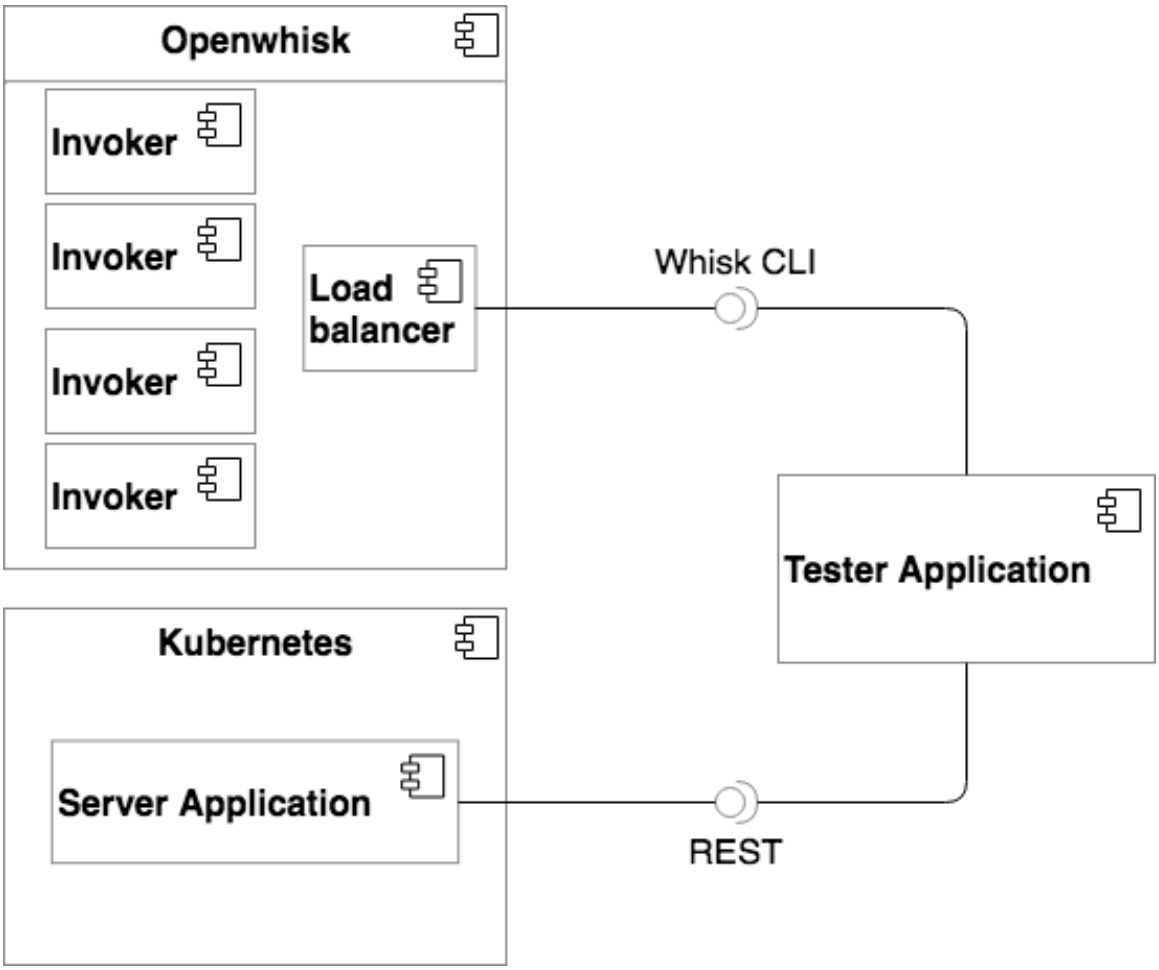


Figure 4.1: Openwhisk: UML Component diagram

4.2 Openwhisk Setup

Apache Openwhisk of the version 0.9.0 is used for testing. Chapter 2 contains description of available options for deployment of Apache Openwhisk. I decided to use the distributed deployment with Ansible playbooks for a few reasons:

- The setup has to be distributed to allow investigation of how the internal load-balancing functionality of Apache Openwhisk works, and figure out what scaling issues appear in different circumstances.
- Deployment via Ansible playbooks provides an option to install the system directly on bare metal without any intermediate solutions while installing the platform on a Kubernetes cluster would add another layer of complexity and it would be impossible to determine how much the Kubernetes performance is influencing the final results.

Table 4.1: Hardware used for Apache Openwhisk deployment

Type	CPUs	RAM	Disk Space	Number of machines
Large	4	10 GB	20 GB	1
Medium	2	5 GB	10 GB	8
Small	1	2.5 GB	10 GB	1

Table 4.2: Hardware used for Kubernetes deployment

Type	CPUs	RAM	Disk Space	Number of machines
Large	4	10 GB	20 GB	1
Medium	2	5 GB	10 GB	8

Table 4.1 shows how many machines are used for the Apache Openwhisk deployment, their parameters and their amount. Each machine has Ubuntu Xenial (v16.04) installed as an operating system, as well as Python 2, Docker and various small utilities required for the platform deployment. Components of Apache Openwhisk are deployed on the machines in the following way:

- A custom docker registry, Nginx and the Openwhisk *Controller* are installed on a large machine. The main reason for that is the fact that a docker registry is very demanding in terms of the disk space, and the *Controller* requires a host with a good performance to handle large numbers of concurrent requests and provide load balancing.
- *CoachDb* and *Kafka* are deployed on one medium machine.
- Other 7 medium machines are used to host *Invokers*.
- One small machine is used to host a MySQL instance, which is not a part of the Apache Openwhisk setup but required to test functions which execute SQL queries against a relational database.

During the deployment, Openwhisk is custom configured to allow unlimited concurrent requests instead of the default 100 requests per minute per namespace.

4.3 Server Application Setup

The server application serves a purpose of comparison of the Apache Openwhisk platform with a more traditional server-based application hosted on a cluster with automated deployment and scaling.

Table 4.3: Server application, endpoint description

Parameter	**Prime Number**	**Matrix**	**Forecast**	**Database**
URL	/prime-number	/matrix	/weather	/db-get
Method	GET	GET	GET	GET
URL params	num: int	size: int	location: string	delay: int
Response	*int*	*"Success"*	*string*	*"Success"*

Table 4.2 shows the parameters and number of machines used for hosting a Kubernetes cluster. A large machine is used to host the Kubernetes master, each of the 8 medium machines is used to host a Kubernetes slave. The essential part about this configuration is that the number of machines and their parameters is exactly the same as the Apache Openwhisk setup, which allows a fair comparison of the two solutions provided that the same resources are used. The Kubernetes cluster is installed and managed via *kubeadm*[1].

The server solution is REST based. It allows executing the same functions that are used to test the serverless platform. The application is written in Java, using the Spring Boot Framework[2], which simplifies development of server applications by automatically including a tomcat server into the final *jar* file, which is easily executable as a simple Java application, without any need to explicitly manage a web server and *war* files. Furthermore, the application is packaged into a docker image and pushed into a Docker registry. A corresponding Kubernetes deployment is configured to pull the docker image from the registry to automatically deploy the application.

Table 4.3 presents the description of endpoints provided by the server application. Figure 4.2 shows the class diagram of the server application. It essentially contains classes, identical with serverless functions, and an endpoint which provides a REST API for function executions.

[1] `https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/`
[2] `http://spring.io/projects/spring-boot`

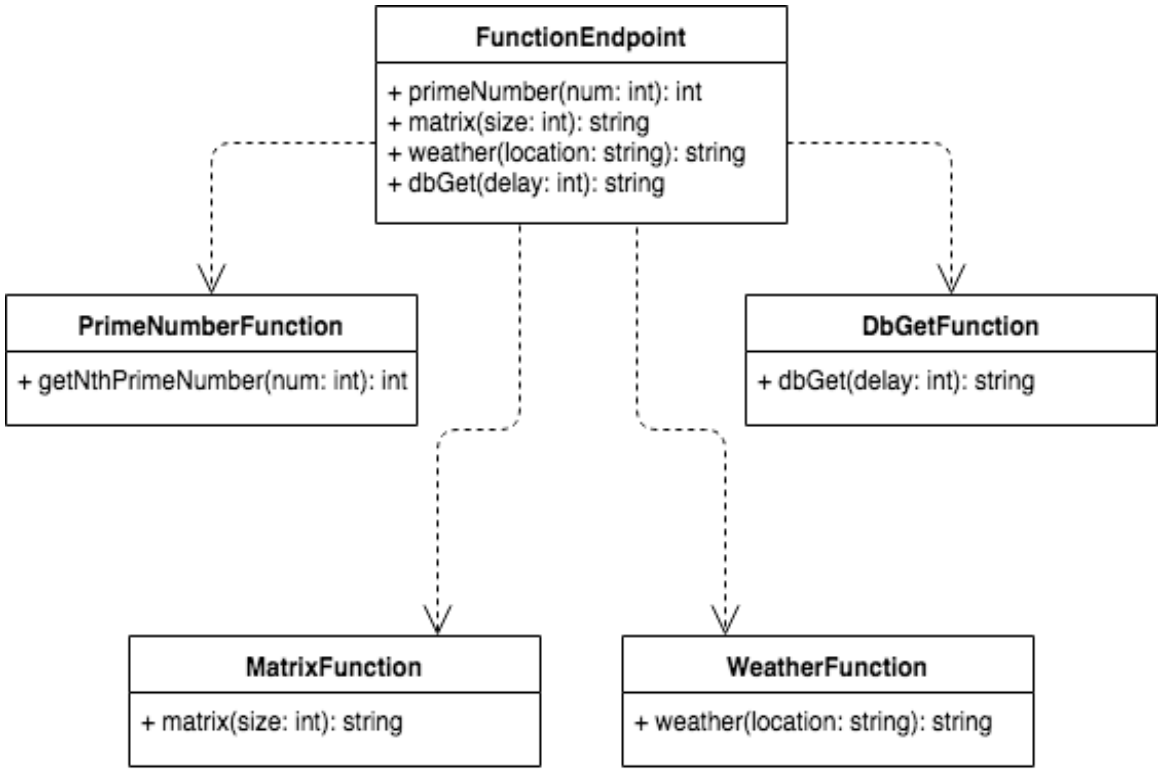


Figure 4.2: Server application: UML class diagram

4.4 Tester Application

The *Tester Application* provides a possibility to execute different test cases. It is based on Python scripts. The scripts perform concurrent requests against the target platform: Apache Openwhisk or the server application.

This component is essentially a performance testing software. The initial approach considered using an existing application for performance testing, such as JMeter[3], however, it does not provide enough flexibility in configuring a distributed testing setup and using WSK-CLI on tester machines.

4.4.1 Configuration

Table 4.4 shows the hardware that is used in the scope of the tester application. Each tester unit is hosted on a tiny machine and has a designated Openwhisk user and hence a

[3] `https://jmeter.apache.org/`

namespace created, which makes each of the tester machines operate as a separate unit, lowering the probability of reaching any of the namespace limitations.

Table 4.4: Hardware used for the Tester Application

Type	CPUs	RAM	Disk Space	Number of machines
Tiny	1	700 MB	5 GB	12

4.4.2 Core Functionality

The core of the tester application is a set of Python scripts which are executed to perform a certain number of concurrent requests from multiple tester machines.

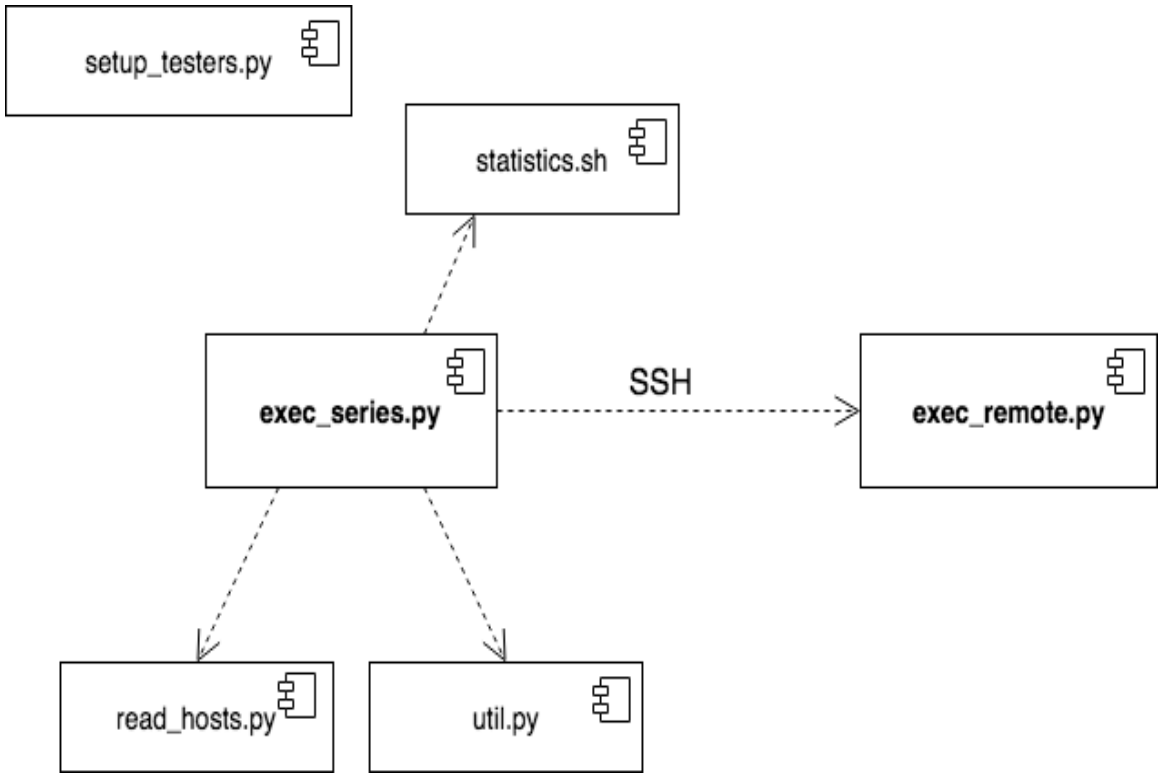


Figure 4.3: Tester application core: UML component diagram

Figure 4.3 shows the structure of the scripts, which all combined represent the core of the tester application.

The script *exec_series.py* is the entry point of the framework, as well as the main orchestrator of all smaller jobs which are a part of the testing process. It accepts the

following data as arguments:

- *Function name*, which defines what sample serverless function is chosen for execution.
- *Input* of the serverless function, e.g. the location of the forecast requesting function or the matrix size for the matrix multiplication.
- *Number of concurrent requests*, which have to be executed in total from all tester machines.

It also has access to config files containing IP addresses of machines, hosting invokers and tester units. The purpose of this script is to initiate a test case, start gathering statistics and retrieve it after the test case is finished. Figure 4.4 depicts this workflow.

The script *exec_remote.py* is executed by the main script on each of the tester machines via SSH, its only purpose is to actually send concurrent requests from each particular tester machine. The following software is installed on each of the tester machines in order to enable the execution of this script:

- Python v2.
- Libraries *concurrent* and *requests*, which are not a part of the default python distribution.

The script *statistics.sh* gathers CPU and RAM usage data on each invoker machine and saves the statistics into log files, which are then retrieved by the main script via SSH for further processing.

The script *setup_testers.py* is used separately before any test cases are executed. Its purpose is to set up a number of namespaces equal to the number of tester machines specified in the config files and also create all sample functions in each of the namespaces.

Apart from the script, there are two config files, which are used to specify the IP addresses of Openwhisk invokers and tester machines. Any positive number of machines can be specified in both cases, making the framework easily scalable.

Furthermore, there are helper scripts, containing utility functions.

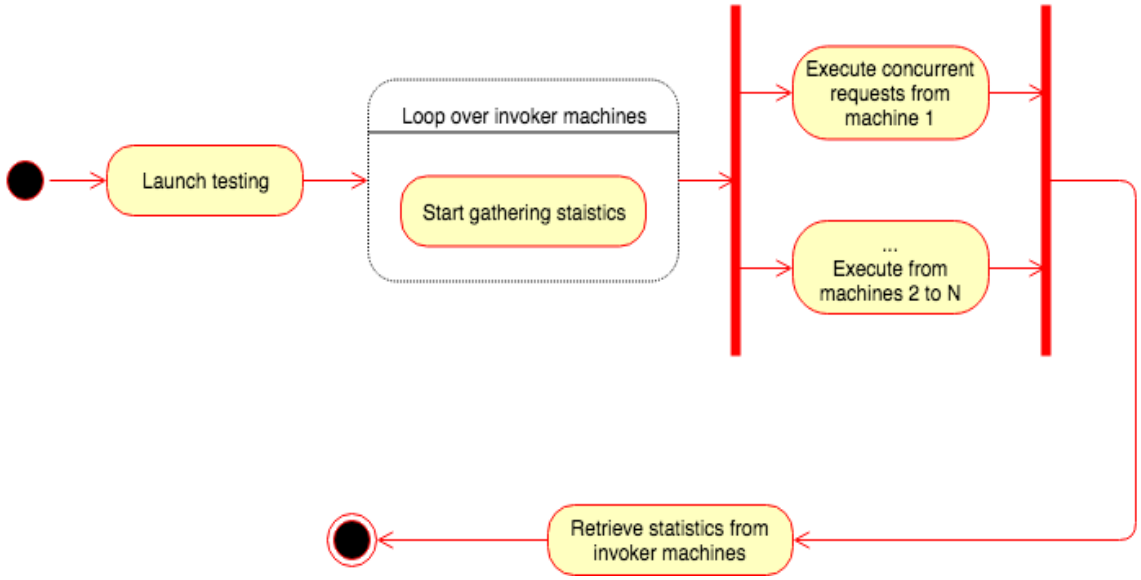


Figure 4.4: Core functionality workflow: UML activity diagram

4.5 Functions

Multiple functions are defined for executing on the platform. The functions are chosen in a way that allows covering various use-cases in terms of dependency on external resources and intensity. Further, the list of the functions is provided.

CPU intensive function

This function calculates the Nth prime number, given the N as the input. The intensity of the function can be easily increased or decreased by providing a larger or smaller input number.

```
function main(params) {
    var num = params.num;
    if (!num) {
        return {
            error: 'The input number must be specified'
        };
    }
    var result = getNthPrimeNumber(num);
    return {
        result: result
    };
}
```

Figure 4.5: Prime number function

Figure 4.5 shows what the main part of the function looks like. It accepts a number as an input, validates it, calculates the Nth prime number and returns the result. The underlying *getNthPrimeNumber(num)* function calculates the result by iterating over all numbers, checking whether it is a prime number and keeping the count.

RAM intensive function

It calculates a multiplication of matrices. As the experiments showed, the function is also highly CPU intensive, therefore to increase the RAM consumption, apart from the matrix size, an arbitrary ram loader value is also provided as an input, which can be seen on Figure 4.6.

```
function main(params) {
    if (!params.size) {
        return {
            result: "Matrix size is required"
        }
    }
    if (!params.ramLoader) {
        return {
            result: "ramLoader is required"
        }
    }

    var fakeList = [];

    var m1 = generateMatrix(params.size);
    var m2 = generateMatrix(params.size);

    for (var k = 0; k < params.ramLoader; k++) {
        fakeList.push(m1[k % 9].slice(0))
    }

    var result = multiplyMatrices(m1, m2);

    return {
        result: 'Success'
    };
}
```

Figure 4.6: Matrix function

There are two underlying helper functions, *generateMatrix*, which generates a square matrix of a given size, and *multiplyMatrices*, which produces a multiplication of two square matrices. The *ramLoader* parameter is used to fill a list of a given size with random values, to consume more RAM.

Weather forecast function

This function depends on an external web resource, making a call to the external API[4] provided by Yahoo. As shown in Figure 4.7, it accepts the location as an input and returns the temperature at that location as a result. In this particular function, the input does not have any effect on the computation intensity.

```
var request = require("request");

function main(params) {
    var location = params.location || "Vermont";

    return new Promise(function(resolve, reject) {
        request.get(url, function(error, response, body) {
            if (error) {
                reject(error);
            }
            else {
                var condition = JSON.parse(body).query.results.channel.item.condition;
                var text = condition.text;
                var temperature = Math.round((condition.temp - 32) * 5 / 9);
                var output = `${location}: ${temperature} degrees, ${text}`;

                resolve({params: output});
            }
        });
    });
}
```

Figure 4.7: Weather function

Query function

```
return new Promise(function(resolve, reject) {
  console.log('Connecting to MySQL database');
  var mysql = require('promise-mysql');
  var connection;
  mysql.createConnection({
    host: params.MYSQL_HOSTNAME,
    user: params.MYSQL_USERNAME,
    password: params.MYSQL_PASSWORD,
    database: params.MYSQL_DATABASE
  }).then(function(conn) {
    connection = conn;
    console.log('Querying');
    var queryText = 'SELECT SLEEP(?) as result';
    var result = connection.query(queryText, [params.delay]);
    connection.end();
    return result;
```

Figure 4.8: Query function

It is a function, making a request to a relational database. The purpose of the function is to investigate the behavior of the platform while processing requests, that heavily depend

[4] `https://query.yahooapis.com/v1/public/yql`

on a long-lasting query to a database. Figure 4.8 shows the function code, which contains the actual logic, various error handlers are omitted. It executes a query against a database, which sleeps for a given number of seconds, simulating a heavy query.

Payload size function

A dummy function, which does not produce any meaningful response and does not perform any computations based on the input. Figure 4.9 presents the function. It is used to investigate the duration of invocations based on the input size, and therefore the simplicity of this function is beneficial to exclude any possible side-effects.

```
function main(params) {
    var payload = params.payload;
    if (!payload) {
        return {
            error: 'The payload is missing'
        };
    }
    return {
        result: 'Success'
    };
}
```

Figure 4.9: Payload size function

4.6 Preparing entities

Before we start executing test cases against the platform, all the actions and other entities have to be properly setup and configured. Firstly, users and namespaces have to be created via WSK CLI. A self-made script creates twelve users - one for every tester machine, sets authentication tokens and corresponding namespaces and creates actions for all the serverless sample functions described above.

In Javascript *prime number*, *matrix multiplication*, *weather forecast* and the dummy *payload size* functions are simply created with a single CLI command referencing the file with the source code. However, the *query* function requires an external library *promise-mysql*[5], therefore it is necessary to wrap it in a NodeJS project with the required dependency specified in the *package.json* file, build the project, archive it along with the automatically generated *node_modules* and create the action from a zip file.

In Java for all serverless functions it is necessary to first compile the source code into a *.class* file, and then pack it into a *jar*. Additionally, since Java, unlike Javascript, does

[5] `https://www.npmjs.com/package/promise-mysql`

not naturally handle JSON, which is passed as a parameter to the serverless function, a *gson*[6] library has to be in the classpath.

4.7 Testing Flow

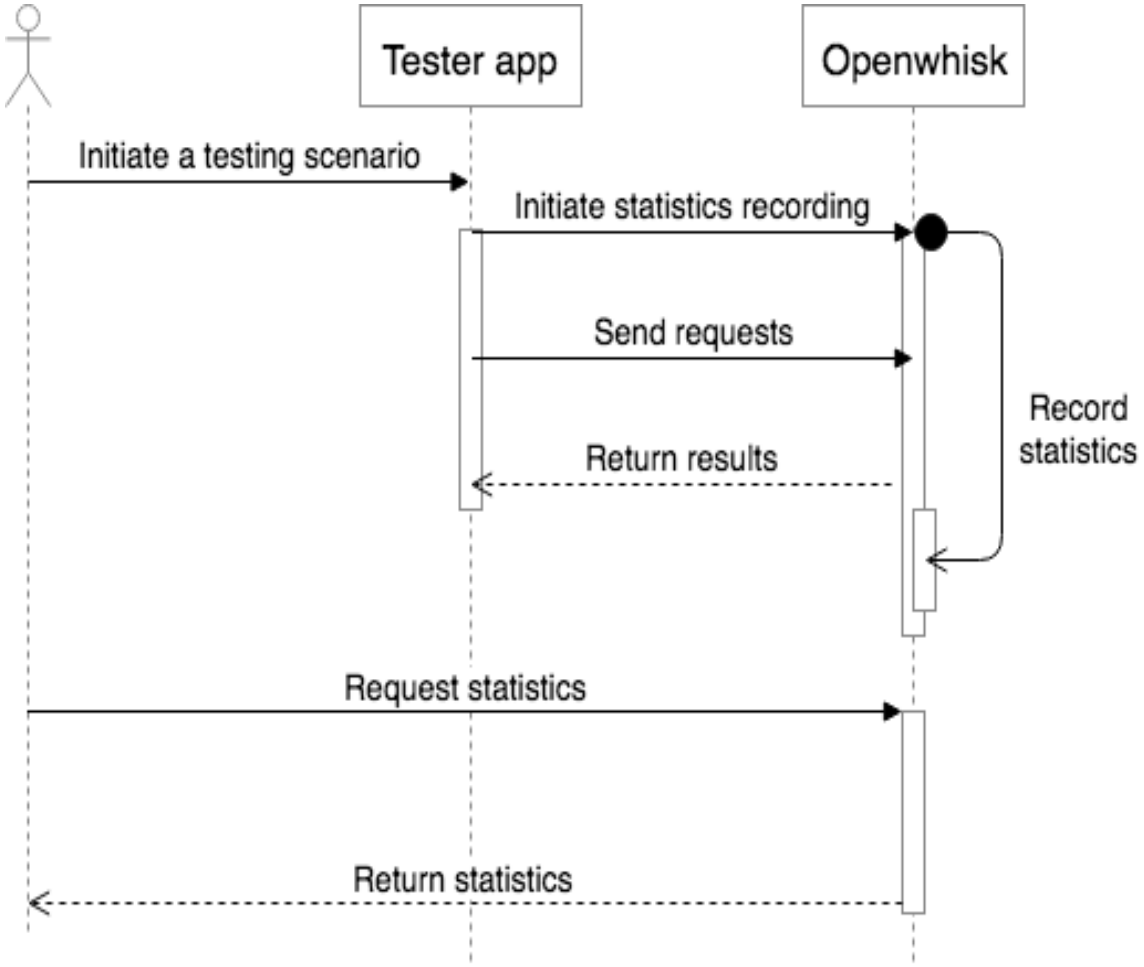


Figure 4.10: Testing Flow: UML sequence diagram

Figure 4.10 presents the general flow within a single test case. Firstly, it is initiated by a user directly executing a Python script. Then, it starts gathering CPU and RAM statistics on the invoker machines. Apart from that, it starts executing testing requests from the tester machines in a concurrent way using separate Openwhisk users and namespaces. Once all the requests are processed and completed, it measures the total execution time

[6] `https://github.com/google/gson`

and retrieves the statistics from the invoker machines. Later the statistics can be processed and visualized separately.

4.8 Test cases

A number of test cases were implemented to investigate the behavior of the Apache Openwhisk platform in different circumstances from various perspectives. A few groups of test cases are considered. Table 4.5 shows which sample functions are used, depending on a particular test case.

Table 4.5: Usage of functions in test cases

	Prime number	**Matrix**	**Weather**	**Query**	**Payload**
Performance	+	+	+	-	-
Scaling	+	+	+	-	-
RAM	-	+	-	-	-
Database	-	-	-	+	-
Payload	-	-	-	-	+
Cold start	+	+	-	-	-

Performance: Apache Openwhisk vs Spring Boot

Comparing how fast the Apache Openwhisk platform and the server solution process a certain load of requests. Includes executing sample functions on both platforms with varying inputs and number of concurrent requests.

Performance: Javascript vs Java

Comparing how fast the Apache Openwhisk platform processes requests, given the same serverless functions implemented in different programming languages: Java and Javascript. To make this comparison fair, the same algorithms are used in the functions with the only difference being the programming language syntax.

Scaling

Invoking different serverless functions with different combinations of inputs and number of concurrent request and investigating RAM and CPU usage on invoker machines. These

kinds of test cases are supposed to reveal how the system is scaling and if there are cases when some invokers stay idle and disregard a heavy load of requests. Includes testing with varying number of concurrent requests and varying computational intensity.

Varying RAM quota

Executing the same RAM intensive function with different RAM quota values. *Matrix multiplication* function is used in this case, with RAM quota values equal to 128 MB, 256 MB, 512 MB.

Database dependency

Executing serverless functions with a heavy dependency on a third-party relational database and evaluating how the system is handling the load and scaling. The *query* function is used in this test case, with the delay value from 1 to 5 seconds.

Payload size

Investigating how the latency depends on the size of the payload for the same function and its computational intensity. The dummy *payload size* function is used, with the payload size from 10KB to 1MB, which is the maximum allowed payload.

Cold start

Invoking a function after a long period of idling and comparing the latency to a "warm" invocation to investigate the *cold start* issue. *Matrix multiplication* and *prime number* functions are used with small inputs, to reduce the invocation duration and observe the latency of the warming up stage.

4.9 Graph representation

After the test case execution was finished, the results are available in the *results* folder in the root of the project. The results within the folder are distributed into directories named as the executed sample functions. Within each of the folders there are actual statistics files of two types:

- Files pulled from invokers, containing two columns: CPU and RAM usage on the specific Openwhisk invoker. The file naming follows the following pattern: *invokerId_functionInput_numberOfConcurrentRequests_.txt*.
- Files containing the execution duration. The file naming pattern is the following: *time_functionInput_numberOfConcurrentRequests_.txt*.

The results are interpreted, processed and represented graphically as a line chart. The graph drawing logic is totally separated from the tester application and can be done in any way. In this thesis *Jupyter*[7] notebooks and *matplotlib*[8] are used to process the results and draw corresponding graphs. The visualizations are presented in Chapter 5.

[7]Jupyter 5.5.0 `http://jupyter.org/`
[8]Matplotlib 3.0.0 `https://matplotlib.org/`

Chapter 5

Evaluation

This chapter presents the results of test case executions. For each test case, there is a description of it provided, a graph depicting the results as well as conclusions based on the results. As described in Chapter 4, there are multiple test cases targeting different objectives, which are further distributed into sections.

5.1 Performance: Apache Openwhisk vs Spring Boot

This section presents the comparison of Apache Openwhisk and Spring Boot application, in terms of how fast they process equal numbers of invocations. Each test case includes a series of tests with increasing amounts of concurrent requests and certain fixed inputs.

Firstly, we test both systems with the *prime number* function, with the following inputs:

- 100: a low input which will simulate a load where each request is knowingly easy to complete for both systems. It is experimentally determined that the serverless platform is able to adequately process a maximum of 4000 concurrent requests and gets overloaded if the number of concurrent requests is higher, so 4000 is chosen as a limit for this particular test.
- 2000: a fairly large input which makes each invocation CPU intensive. Number of concurrent requests equal to 3000 is chosen as a limit for this test.

Figure 5.1 depicts that for non-intensive low inputs Spring-Boot application hosted on a Kubernetes cluster works much faster than the Apache Openwhisk platform, where the difference becomes enormous on large numbers of concurrent requests.

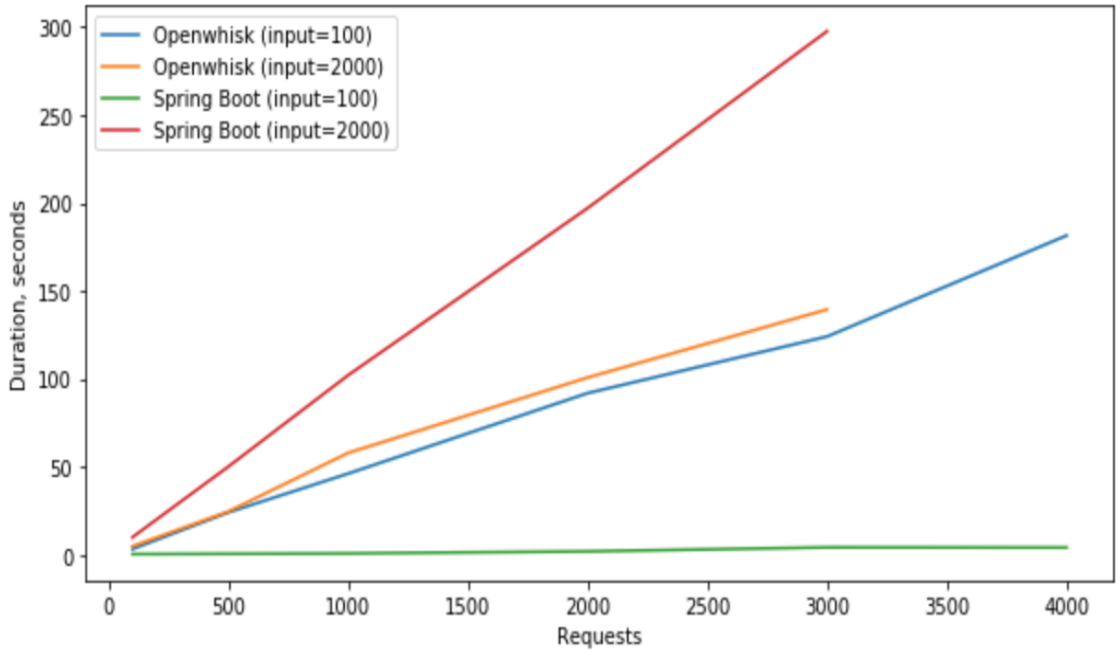


Figure 5.1: Comparison of latency: prime number function

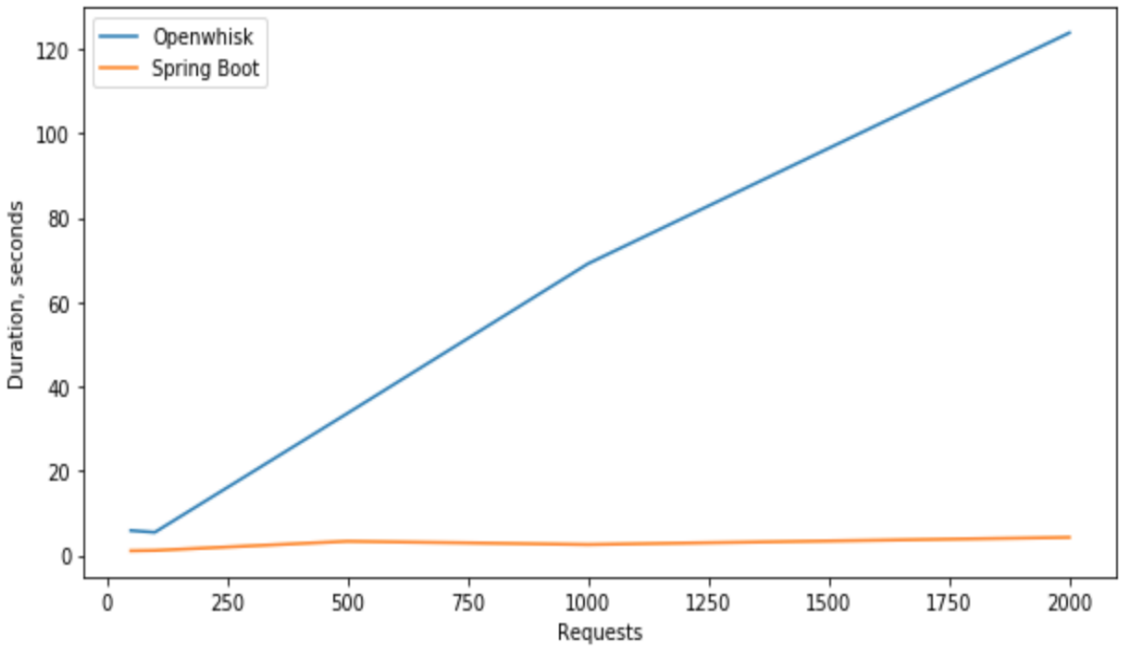


Figure 5.2: Comparison of latency: weather forecast function

However, the situation looks completely different on high inputs which make every single invocation of the function intense: the Apache OpenWhisk results look almost the same,

however, the server-based solution takes much longer to process the load and the latency increases linearly.

Further, both systems are tested with the *weather forecast* function, which makes a request to an external web API. There is only one test case since there is no predictable change depending on what particular input is provided to the external API. The maximum number of concurrent requests is set to 2000 since it is absolutely enough in this case to see the general tendency. Figure 5.2 shows the results of the test case. While the server-based application is able to handle any number of the concurrent requests within the given boundaries without any significant change in the latency, Apache OpenWhisk shows a linear growth of the latency. Hence, the serverless platform is not able to handle invocations involving an access to an external web API as efficiently.

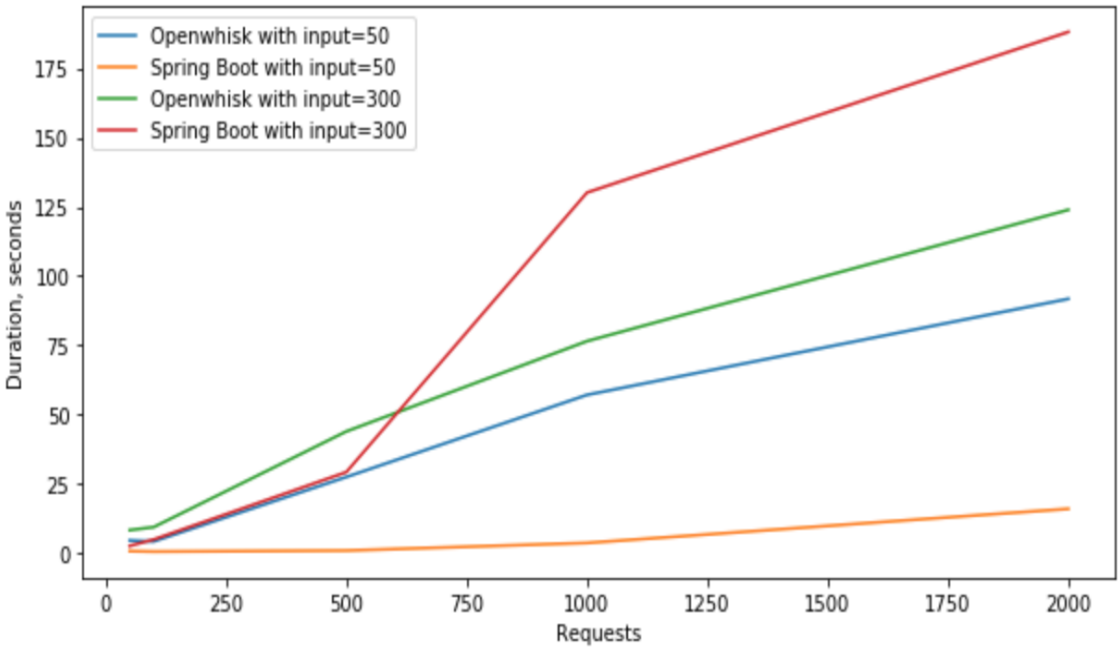


Figure 5.3: Comparison of latency: matrix multiplication function

Lastly, both Apache OpenWhisk and the Spring-boot application are tested with the *matrix multiplication* function. The same way as in the *prime number* case, this test case includes inputs, providing a low and a high intensity, where for each fixed input there is a scale of varying numbers of concurrent requests. The maximum number of concurrent requests in this test case is equal to 2000. The tests are done with inputs 50 and 300 as the size of each of the two multiplied matrices.

Figure 5.3 depicts results similar to the one related to the execution of the *prime number* function, even though apart from being CPU intensive, the *matrix multiplication* function

is also RAM intensive. The Apache OpenWhisk performs significantly worse than the server application on lower inputs, but better on higher inputs which make the invocation intensive.

5.2 Performance: Javascript vs Java

This section presents the comparison of how the platform performs with the same serverless functions written in different programming languages given the same inputs.

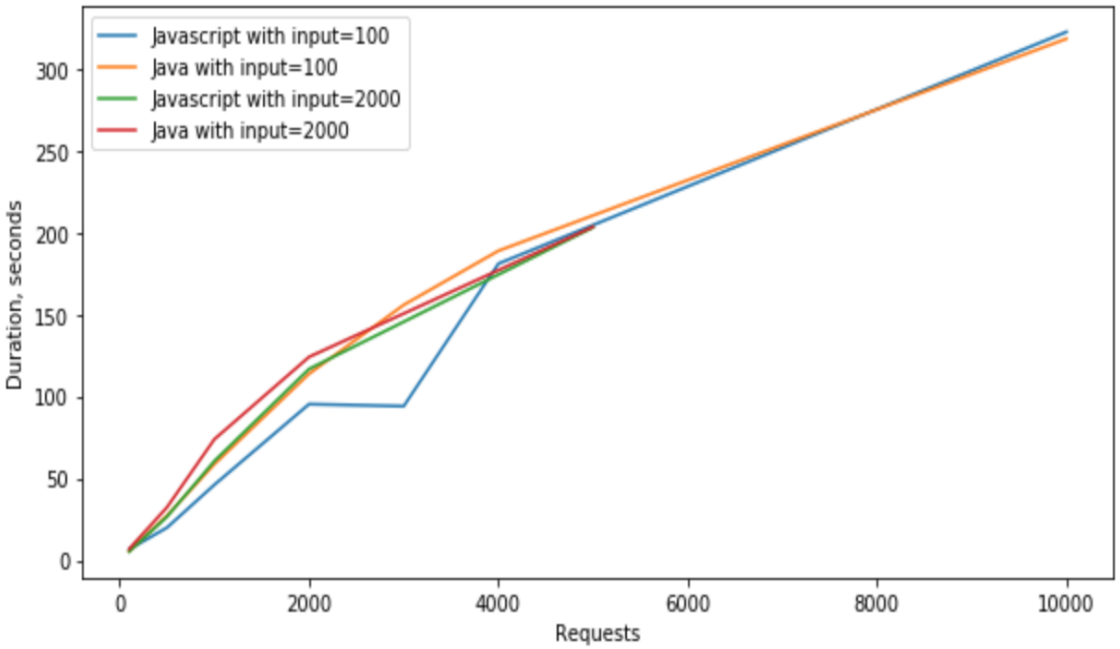


Figure 5.4: Comparison of latency Java vs Javascript: prime number function

Figure 5.4 depicts the results of testing the platform with the *prime number* function, with inputs 100 and 2000, and the maximum value for the concurrent requests equal to 10000. It is clear from the graph that the performance of the platform on this function does not depend on the programming language the function is written in.

Figures 5.5 and 5.6 show the results for the *weather forecast* and *matrix multiplication* function, which similarly to the *prime number* function results show that there is also no significant difference in the performance depending on the programming language.

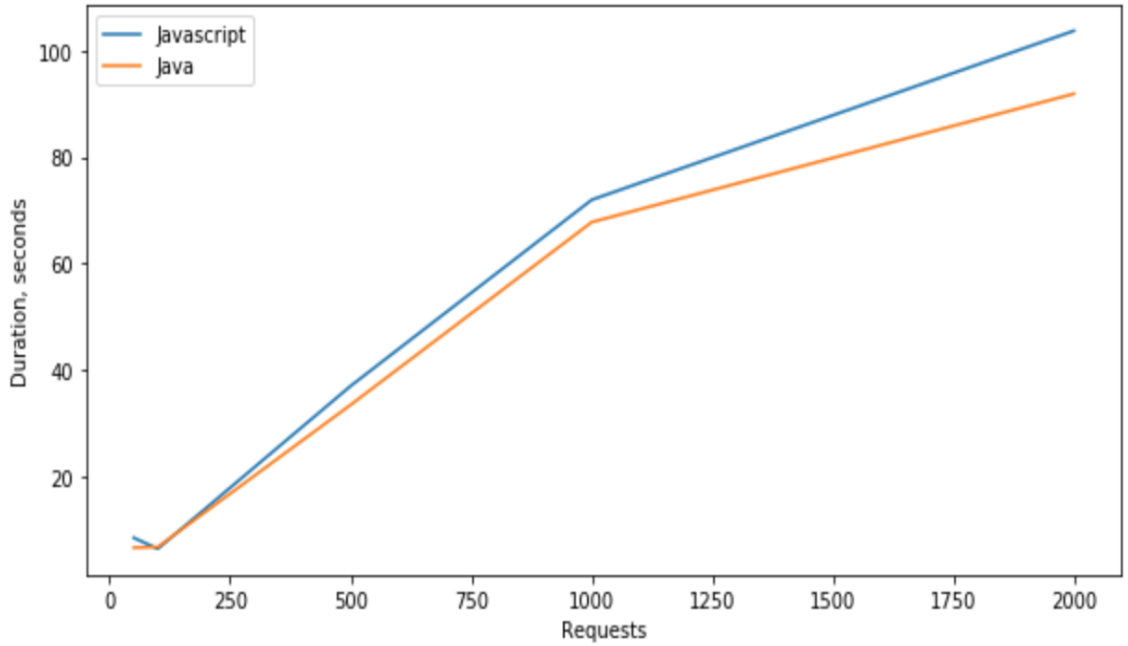


Figure 5.5: Comparison of latency Java vs Javascript: weather forecast function

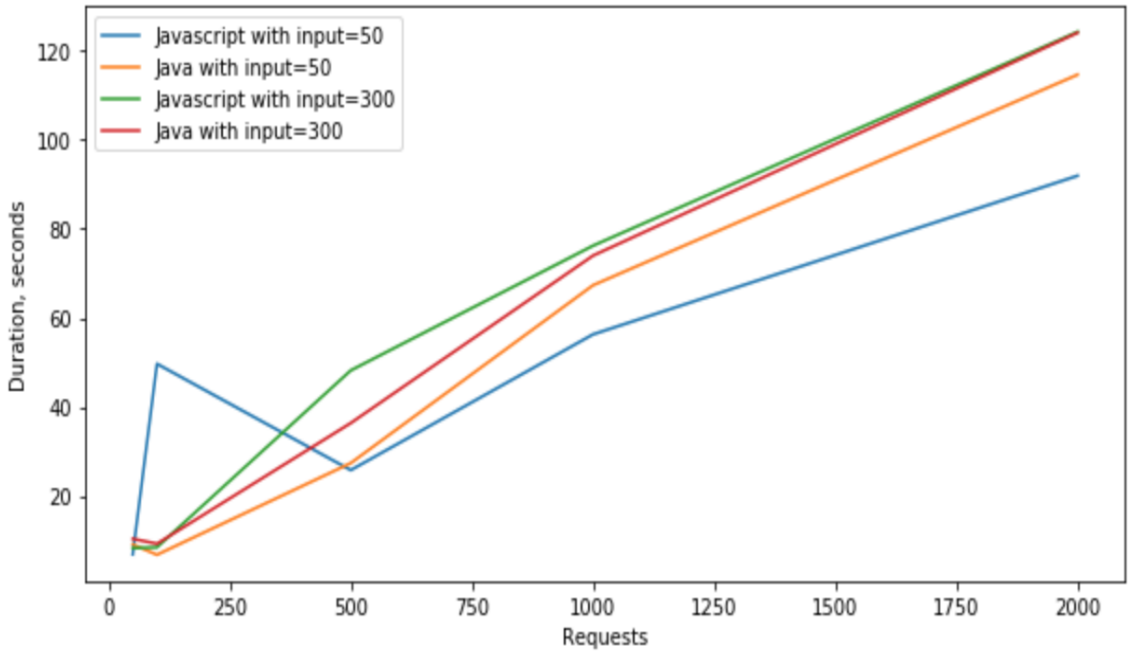


Figure 5.6: Comparison of latency Java vs Javascript: matrix multiplication function

5.3 Scaling

This section presents testing of the Apache Openwhisk platform with various test cases that include the *prime number*, *weather forecast* and *matrix multiplication* functions. The

goal is to investigate the CPU usage of all invoker machines and understand how the system is scaling given different inputs and number of concurrent requests.

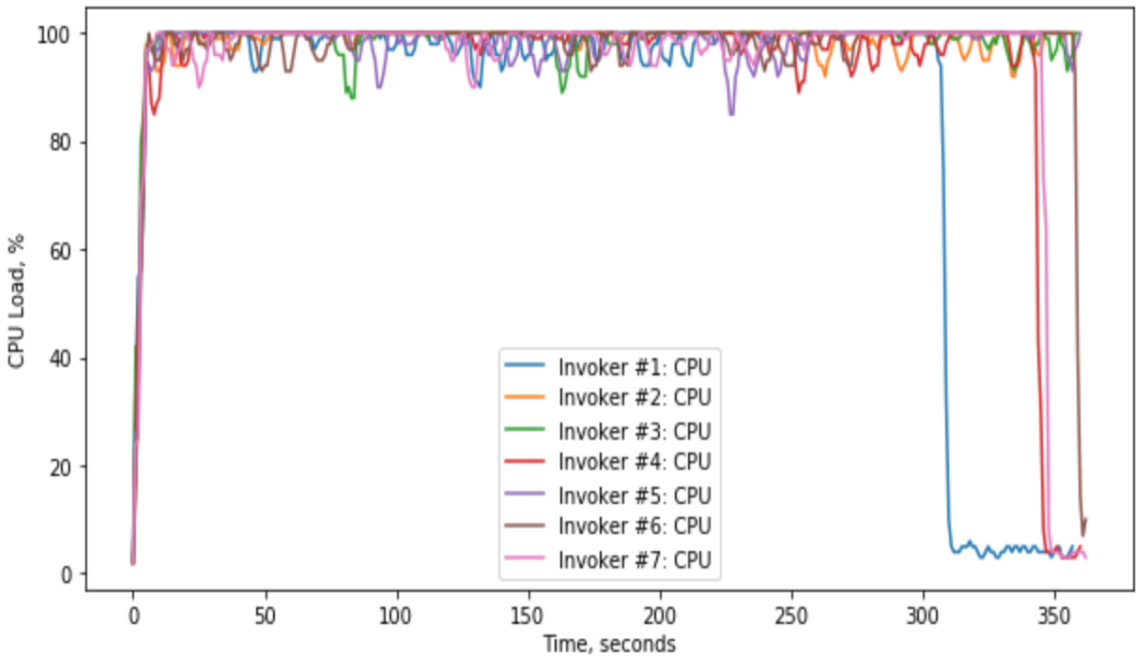


Figure 5.7: Scaling: prime number function, input = 1000, concurrent requests = 2000

The *prime number* function is tested with inputs from 10 to 3000, and each fixed input is tested with a varying number of concurrent requests from 100 to 5000. The *weather forecast* is tested with a number of concurrent requests from 50 to 2000. The *matrix multiplication* function is tested with inputs from 50 to 300, and number of concurrent requests from 50 to 2000. Figures 5.7, 5.8, 5.9 show how the system scales for the *prime number*, *weather forecast* and *matrix multiplication* functions respectively.

Graphs show that in the case of the *prime number* and *matrix multiplication* functions the platform scales uniformly, involving all invokers to process the load. However, 5.8 depicts that there are consequent drops in CPU usage on all invokers throughout the duration of the testing process with the *weather* function, which is likely to be caused by the delay of the request to an external API.

It is important to mention, that on versions of Apache OpenWhisk earlier than 0.9.0, the system always allocated 10% (or at least one if the total number of invokers is less than 10) strictly for docker actions[1]. Later the issue was fixed to involve all invokers into any kinds of invocations on small setups.

[1] `https://github.com/apache/incubator-openwhisk/pull/3751`

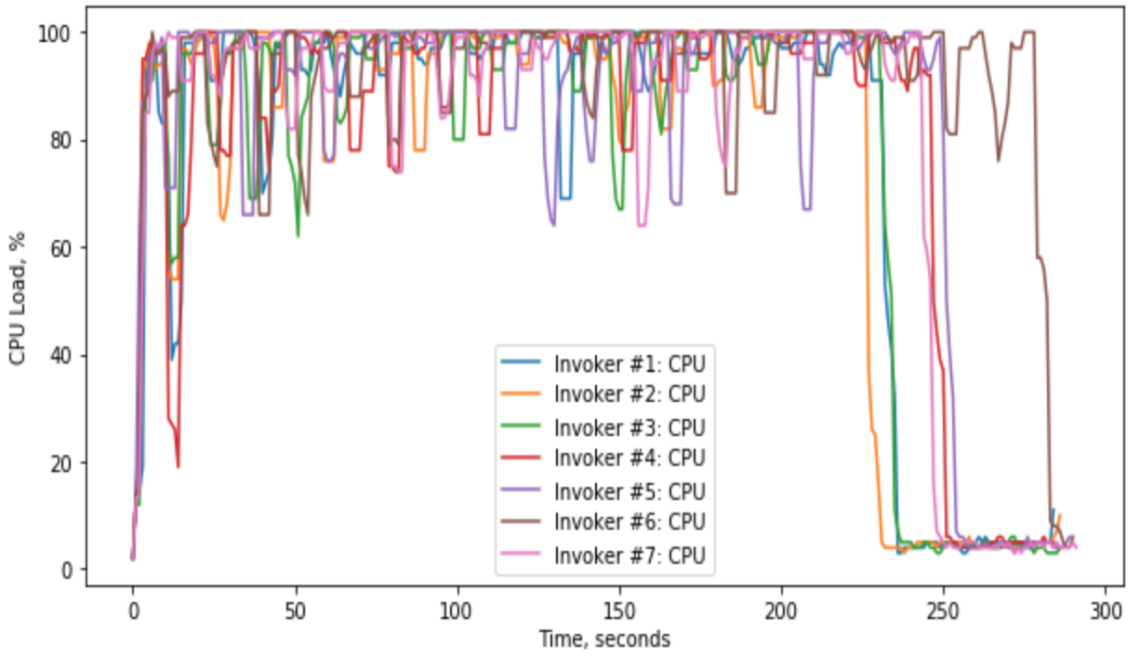


Figure 5.8: Scaling: weather forecast function, concurrent requests = 1000

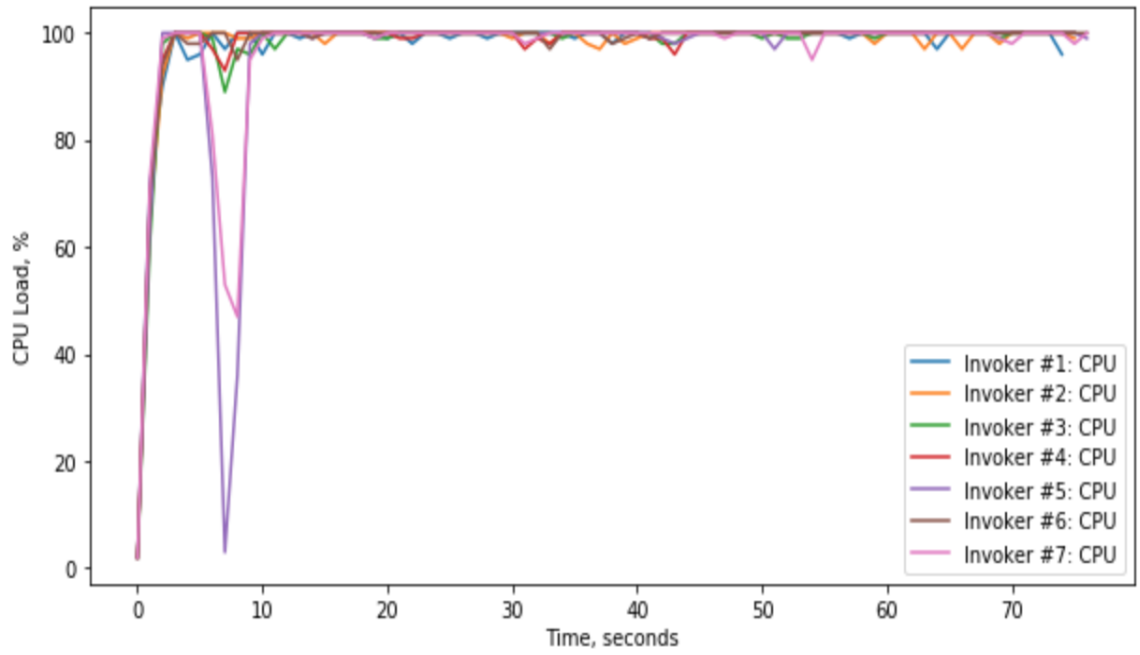


Figure 5.9: Scaling: matrix multiplication function, input = 300, concurrent requests = 1000

Further, we run tests checking the scalability of the platform depending on how intensive invocations are, i.e. a large number of non-intensive concurrent invocations compared to

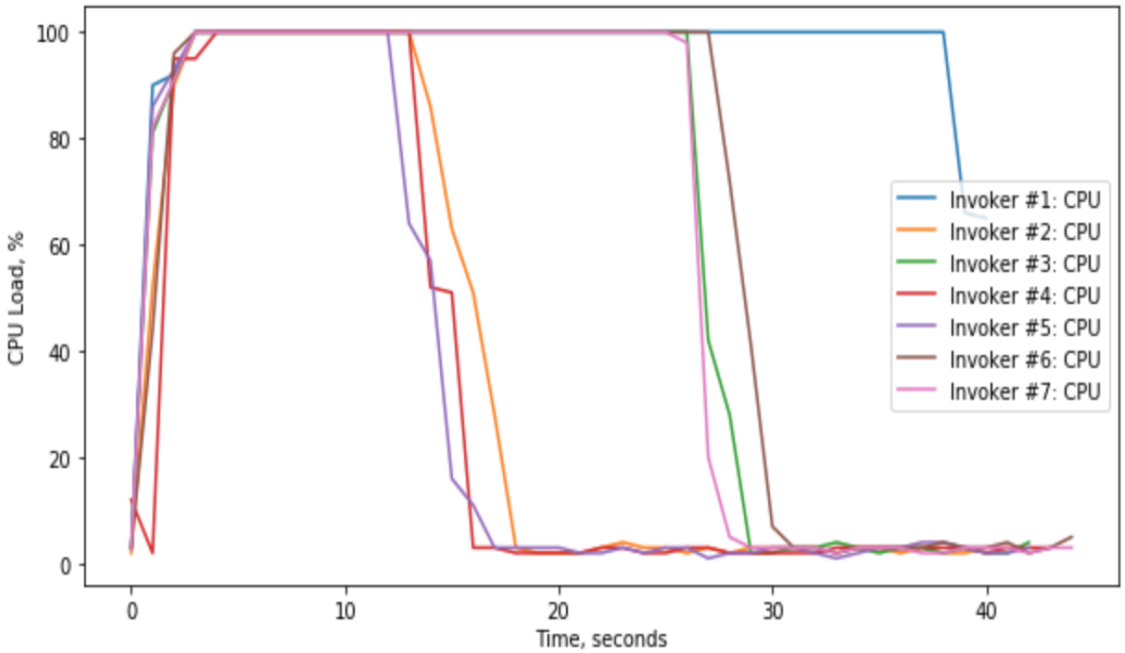


Figure 5.10: Scaling: prime number function, input = 10000, concurrent requests = 60

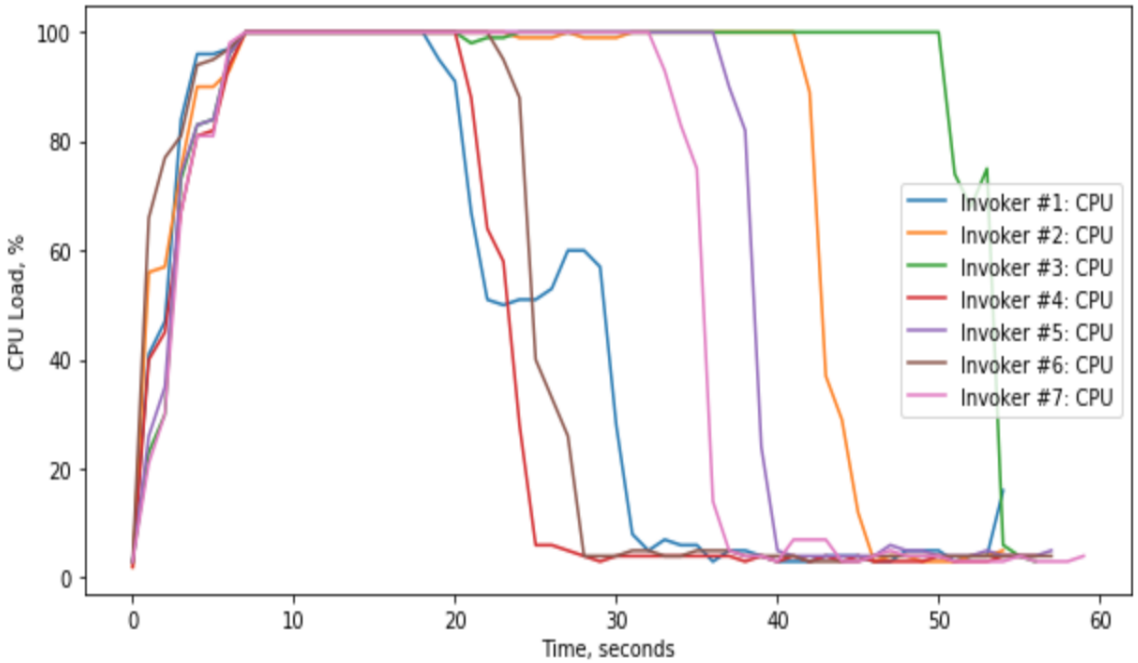


Figure 5.11: Scaling: matrix multiplication function, input = 500, concurrent requests = 60

a small number intensive invocations. *Prime number* and *matrix multiplication* functions are chosen for this test as the only sample function that vary in intensity based on the

input.

Figures 5.10 and 5.11 depict how Apache Openwhisk is scaling, given very high inputs and low number of concurrent requests for *prime number* and *matrix multiplication* functions. In both cases the system does not handle the load efficiently, making some of the invokers process more requests while other invokers are idle. Therefore we can conclude that the platform is more efficient in handling a larger number of simpler invocations, rather than fewer serious computations.

5.4 Varying RAM quota

This section shows how Apache Openwhisk processes the same number of concurrent invocations of the same memory intensive *matrix multiplication* function with the same input, depending on the RAM limit set per action.

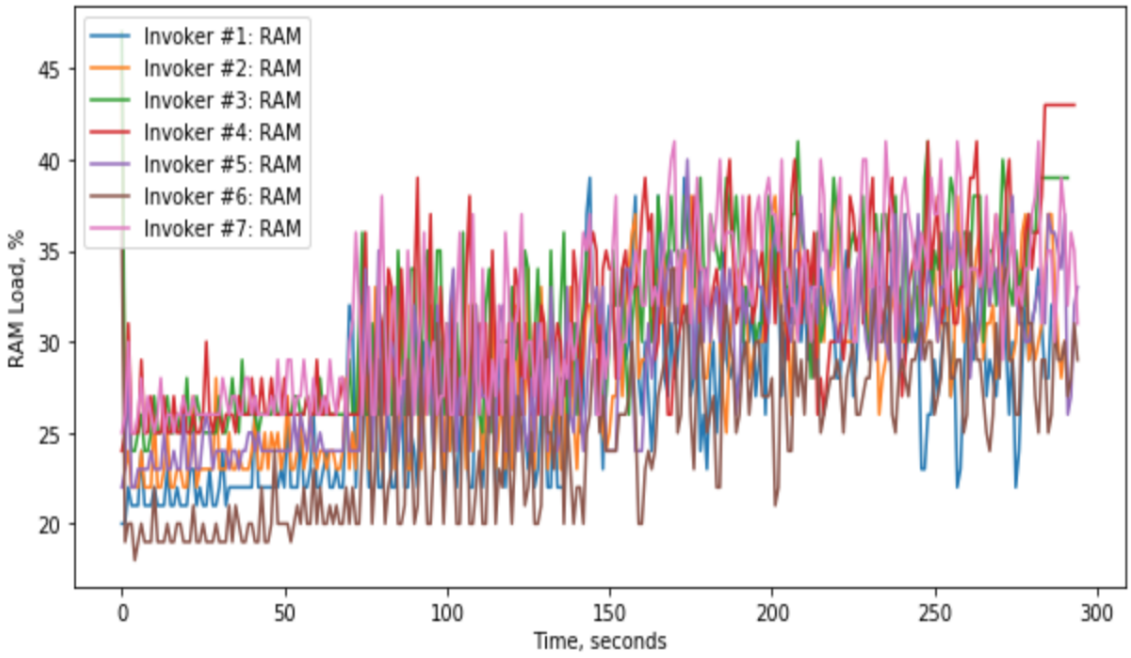


Figure 5.12: Scaling: matrix multiplication function, input = 300, concurrent requests = 1000, RAM limit = 256MB

As described in Chapter 2, the default RAM limit is 256MB, and the maximum value is 512 MB. In these test cases, 128MB, 256MB, 512MB RAM quota values were used for testing. Testing was done with the matrix size 300 and the number of concurrent requests equal to 1000.

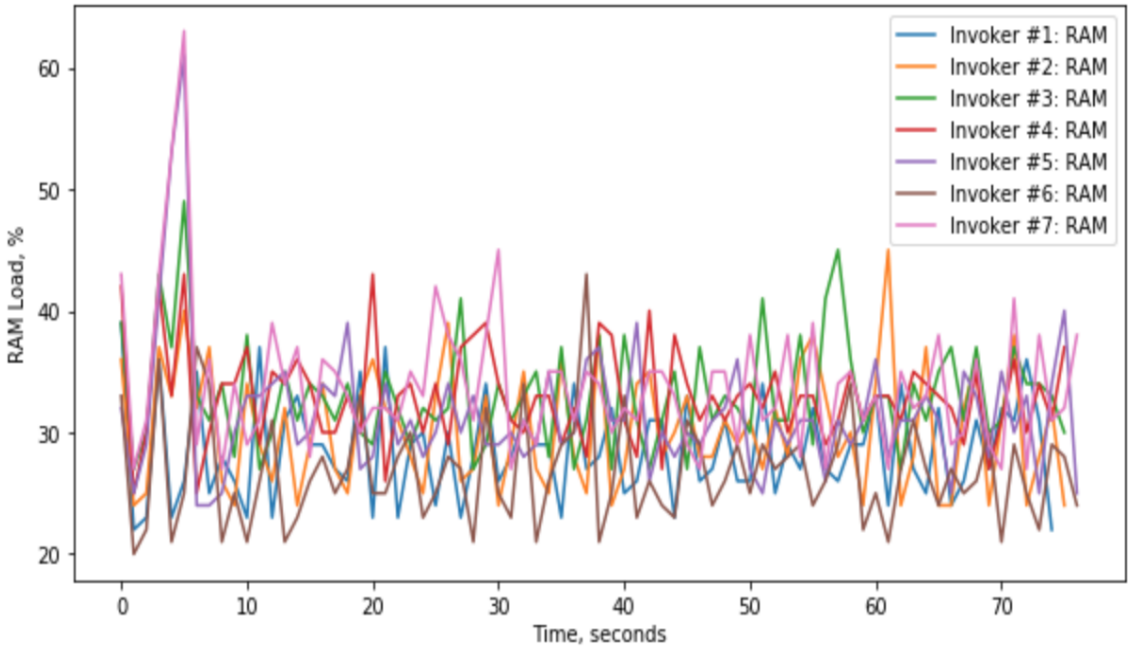


Figure 5.13: Scaling: matrix multiplication function, input = 300, concurrent requests = 1000, RAM limit = 512MB

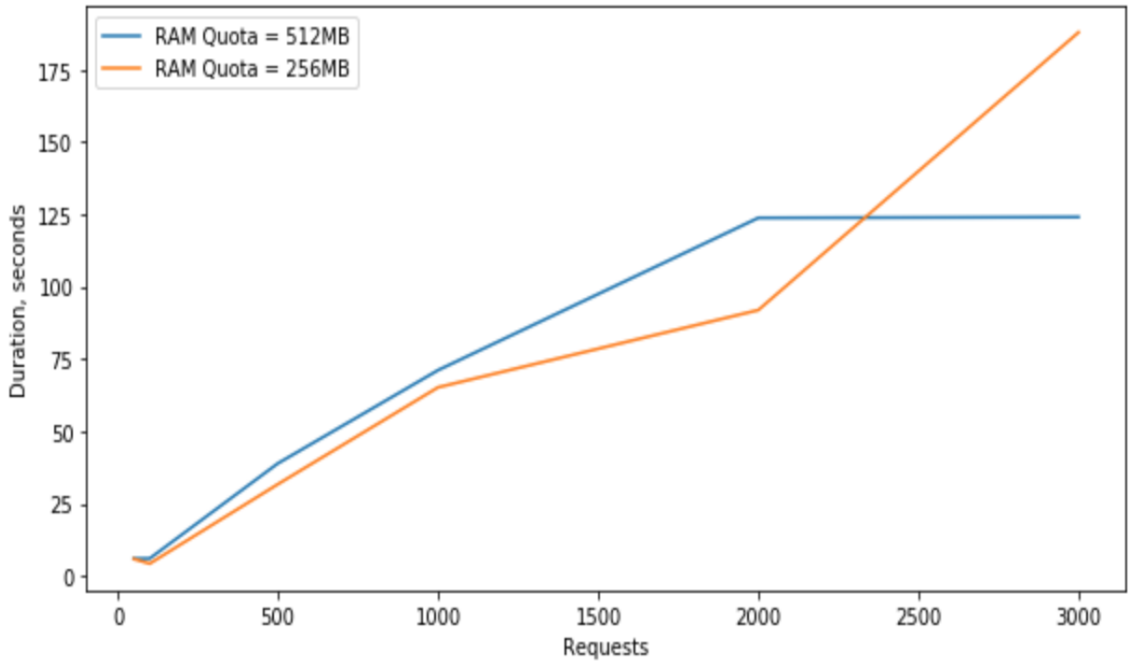


Figure 5.14: Latency: matrix multiplication function, input = 300

For the RAM limit of 128MB, the testing process was not able to produce any reasonable result, since many requests failed due to an insufficient amount of memory.

Figures 5.12 and 5.13 show graphs of the results for limits of 256MB and 512MB respectively. In both cases all invokers are involved in processing invocations, however, a higher RAM limit makes the computations slightly faster, and also makes the amplitude of the RAM usage larger.

However, Figure 5.14 depicts that there is no significant difference in the overall latency depending on the RAM quota.

5.5 Database dependency

In this test case, the platform is tested with a function which executes a query against a relational database. The query sleeps for a given number of seconds, simulating a heavy long-lasting query. Figure 5.15 depicts the CPU usage of the invokers.

For all invokers, the CPU usage value is low and slightly changing throughout the testing process. Hence, we can conclude, that the serverless platform is inefficient in scaling while performing invocations which heavily depend on a third-party resource, such as a database. Results of tests with smaller and larger sleep duration of the query show that the heavier the query is, the less efficient the scaling of the serverless platform gets.

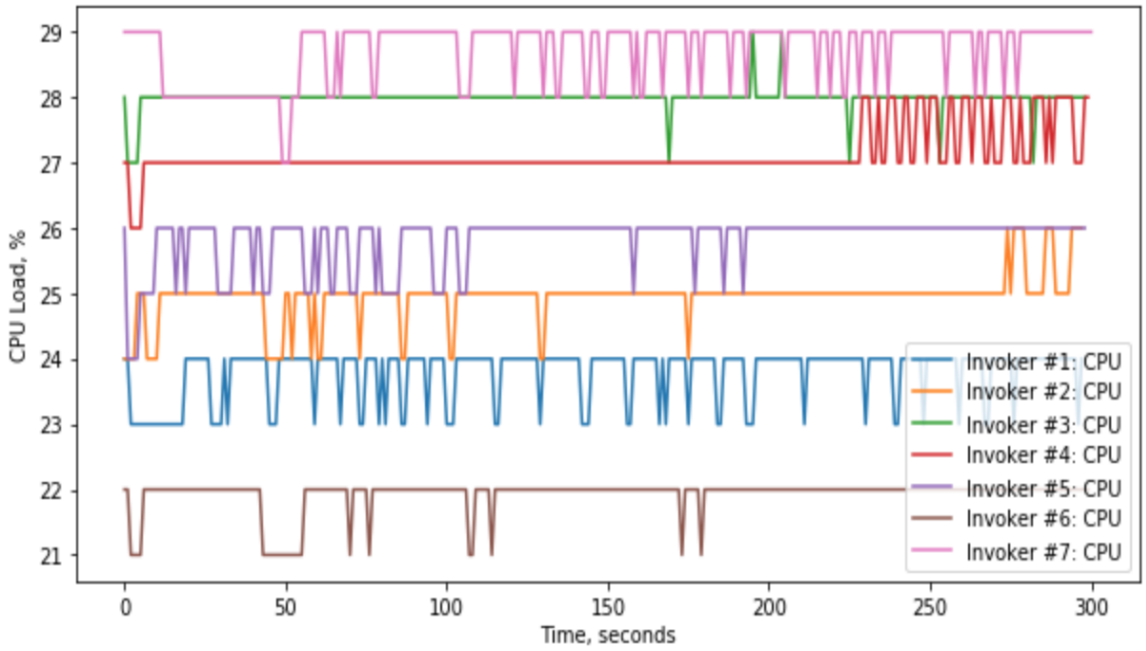


Figure 5.15: Scaling: database querying function, delay = 4 seconds

5.6 Payload size

In this test case, a dummy serverless function is executed with various payloads of different size: from 10 bytes to 1 MB. The duration of an invocation depends on the payload size in a linear way, as Figure 5.16 depicts.

An attempt to exceed the payload size further leads to the error 413: payload too large, which is consistent with the system restriction announced by developers and described in Chapter 2. Hence, we can conclude, that a reasonable difference in the payload size does not lead to unexpected significant changes in the latency, but can noticeably decrease the total execution time.

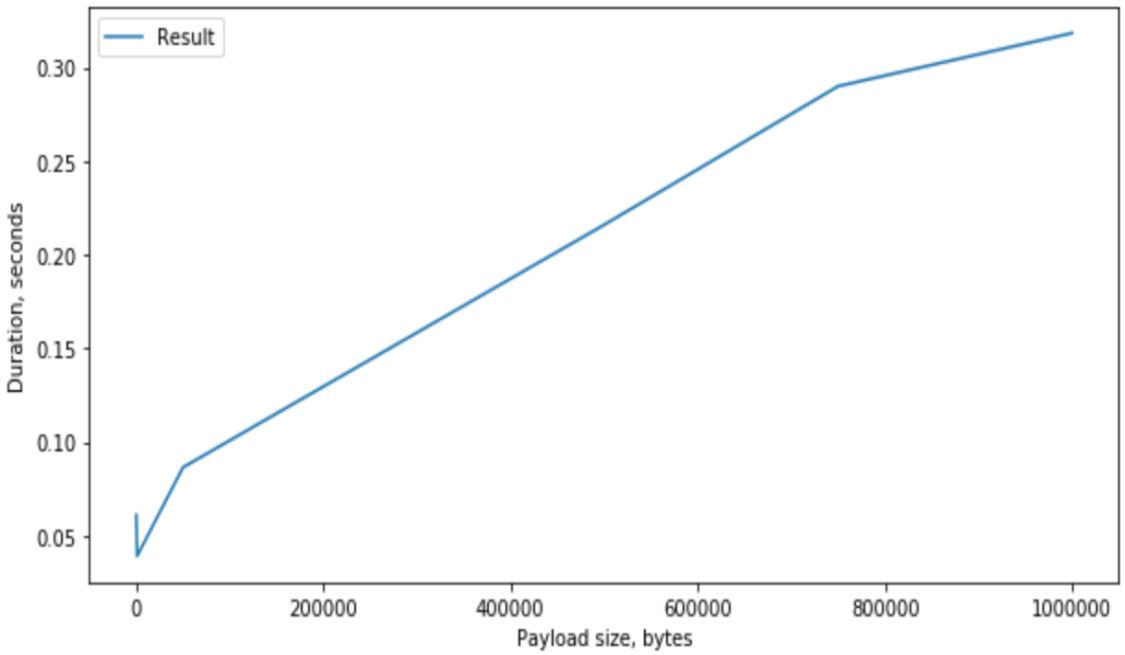


Figure 5.16: Latency depending on payload size

5.7 Cold start

The platform was tested with the *prime number* and *matrix multiplication* functions multiple times after the system has been idle. Also, the functions were executed again with the same inputs to compare the latency of a *cold* and a *warm* start. Small inputs were chosen to make the docker action initialization (as described in Chapter 2) a significantly longer process than the invocation of the function.

Table 5.1: Cold and warm start latency

Function	Cold start duration, ms	Warm start duration, ms
Prime number	110	8
Matrix	375	17

Table 5.1 presents the average duration of invocations in the cold and the warm start situation. It is clear that the latency is significantly higher in a *cold start* situation, which is an issue in cases when a serverless API is not executed frequently.

5.8 Observed limitations of Apache OpenWhisk

This section does not target any specific features of serverless in particular, but simply provides information about the behavior of Apache Openwhisk. While executing various test cases with larger inputs or high numbers of concurrent requests, a few issues were noticed.

Very long computations

Extremely large inputs which make an invocation very resource intensive, exceed the specified timeout. The maximum configurable timeout for a single action is 5 minutes, therefore the platform is not suitable for long computations and much more efficient in dealing with a larger number of light concurrent invocations.

Irresponsiveness after a heavy load

Once a large number of CPU-intensive invocations is executed, the system becomes irresponsive and does not produce any reasonable responses, also for new less-demanding invocations.

Caching

No way of caching results of invocations was found. This functionality can significantly simplify use-cases when the same functions are often executed with identical arguments. E.g. it would be useful to have the option to have a Redis storage configured to store results of invocations based on a key represented by a hash created from the function name and the inputs combined. However, there is currently some work in progress[2].

[2] `https://github.com/ddragosd/openwhisk-cache-redis`

Invokers reserved for docker actions

In older versions of Apache Openwhisk (before v0.9.0) some of the invokers might be always idle, even while a large number of invocations is being processed. It happens due to the fact that the system always reserves 10% or at least one invoker only for docker actions. Therefore small setups can suffer from this issue. For example, a setup containing 4 invokers would a have only 3 invokers actually involved in most of the invocations, which makes 25% of hardware allocated for invokers obsolete.

Chapter 6

Summary

This thesis proposed an approach to investigating the behavior of the serverless Apache Openwhisk platform, as well as its limitations and bottlenecks. In particular, a performance testing application was designed and created, also various test cases and test functions were implemented. Furthermore, this work presented an evaluation and visualization of achieved results. Compared to existing research in this field, this thesis focuses more on investigating the Apache Openwhisk platform, rather than comparing different serverless platforms.

6.1 Status

The goal of the thesis was achieved: a tester application was implemented, various test cases were executed and results evaluated. The list of test cases addressed different aspects of the serverless platform: its performance compared to a traditional server-based system, its scalability in certain circumstances, such as invocations that access an external web resource and a database, as well as the cold start issue.

6.2 Conclusions

According to the results achieved in this work, we can conclude that Apache Openwhisk is generally prone to the known issues that are commonly associated with the serverless paradigm. Firstly, the platform lacks efficiency in dealing with very intensive long-lasting jobs due to its timeout restrictions. Secondly, the system does not scale as efficiently if an

action has dependencies on third-party systems which are a black box from the Openwhisk perspective. The cold start issue is also very obvious, with the "cold" invocation latency being multiple times larger than the duration of invocations on "warm" containers, despite the declared docker container pre-warming which is supposed to solve this issue. Apart from that, in many tasks, a server-based solution shows better performance than the serverless platform.

However, Apache Openwhisk has a good enough performance on functions, which do not perform any large computations and also scales well given a large number of concurrent invocations. This particular solution and FaaS platforms, in general, are a good alternative to server applications in cases, where small and easily maintainable API is required without dedicated backend developers and infrastructure teams being involved, e.g. for certain types of mobile applications.

6.3 Future Work

There are multiple ways to continue this work.

Frontend

In this work, python scripts are used as an interface to the tester application. The results are processed and visualized separately via Jupyter notebooks. However, for a more convenient use by multiple users the tester application could be extended with a frontend application with a graphical dashboard for managing and executing test cases.

Evaluation of other platforms

The implemented framework can also be used for similar evaluations of other serverless platforms and even other systems, provided that they have a REST interface. However, adaptations in the implementation and design of the test cases would be required. Apart from that, the commercial *IBM Functions* solution which uses the Apache Openwhisk platform could be evaluated in a similar way, including aspects not related to the open-source system, such as pricing and SLA.

Appendices

Appendix A

Appendix

A.1 User guide

A.1.1 Apache Openwhisk

A detailed description of the platform installation can be seen at https://github.com/apache/incubator-openwhisk/blob/master/ansible/README_DISTRIBUTED.md.

A.1.2 Tester framework

This section contains instructions to how to use the implemented framework, going all the way from pulling the project from the repository.

Clone

Firstly, the project has to be cloned from the repository at https://github.com/i13tum/openwhisk-bench.

Update config files

Further, the *scripts/config/invoker_hosts* and *scripts/config/tester_hosts* files have to be updated with the actual IPs of Openwhisk invoker machines and tester machines correspondingly. The count of machines in each case can be absolutely arbitrary, and the framework will take it into account automatically.

Prepare tester machines

To make the scripts work correctly, make sure that the following software is installed on all tester machines:

- Python,
- Python packages *concurrent* and *requests*, which are not included into Python v2 by default.

Also make sure that the host which is used as the starting point for the framework, has an SSH access to all the tester machines with login *ubuntu*.

Prepare namespaces and actions

Next, create *jar* files for functions implemented in Java by executing the *functions/java/create_jars.sh* script. Further, namespaces and actions have to be created for further test case executions. It can be done manually via WSK CLI, however, with a larger number of tester machines this would be a long tedious task. Therefore a script *setup_testers.py* was implemented. A simple execution of this script without any arguments creates a number of namespaces equal to the number of tester machines and creates sample functions for each of the namespaces. In case something in the action configurations has to be changed, such as the RAM quota or timeout values, it can be done in the script *create_functions.sh*.

Execute a single test case

The script *tests.py* is supposed to be used to execute a test case. There is a number of test cases which are already defined in the script. The general syntax is the following:

python tests.py test_case_name

Each test case is internally executing the script *exec_series.py*, which is responsible for a single test execution and accepts the name of the function, the function input and the number of concurrent requests as an input.

Statistics

After the test case execution was finished, the results are available in the *results* folder in the root of the project. The results within the folder are distributed into directories named as the executed sample functions. Within each of the folders there are actual statistics files of two types:

- Files pulled from invokers, containing two columns: CPU and RAM usage on the specific Openwhisk invoker. The file naming follows the following pattern: *invokerId_functionInput_numberOfConcurrentRequests_.txt*.
- Files containing the execution duration. The file naming pattern is the following: *time_functionInput_numberOfConcurrentRequests_.txt*.

Statistics can be evaluated and visualized with Jupyter notebooks. Examples can be seen in the *results_final* directory.

A.2 Code snippets

exec_series.py

```
import os
import time
from multiprocessing import Process
from read_hosts import invokerHosts, invokerMapping, testerHosts, testerMapping


def executeRemote(hostname, functionName, inputNum, processesPerHost):
	os.system("ssh ubuntu@{0} python -u - {1} {2} {3} {0} {4} "
		"< ./exec_remote.py > logs/log_{0}_{1}_{2}_{3}.txt"
		.format(hostname, functionName, inputNum, processesPerHost,
			testerMapping[hostname]))


def launchStatistics():
	print("Launching statistics...")
	for host in invokerHosts:
		os.system("ssh ubuntu@{0} 'bash -s' < statistics.sh & "
			.format(host))


def retrieveStatistics(functionName, inputNum, numOfProcesses):
	for host in invokerHosts:
		filename = "{0}_{1}_{2}_.txt".format(invokerMapping[host],
			inputNum, numOfProcesses)
		os.system("ssh ubuntu@{0} 'cat result_statistics'"
			" >> results/{1}/{2}"
			.format(host, functionName, filename))
	print("Retrieved statistics.")


def runSeries(functionName, inputNum, numOfProcesses):
	proc = []
	processesPerHost = numOfProcesses / len(testerHosts)
	launchStatistics()
	print("Starting testing {0} with input={1} and {2} requests..."
		.format(functionName, inputNum, numOfProcesses))
	startTime = time.time()

	for host in testerHosts:
```

```
                p = Process(target=executeRemote, args=(
                        host, functionName, "{0}".format(inputNum), "{0}"
                                .format(processesPerHost),
                ))
                p.start()
                proc.append(p)

        for p in proc:
                p.join()

        timeDif = time.time() - startTime
        print("Completed all requests in {0} seconds"
                .format(time.time() - startTime))
        retrieveStatistics(functionName.replace("-", "_"),
                inputNum, numOfProcesses)

        return timeDif
```

exec_remote.py

```
import concurrent.futures
import sys
import time

import requests
from requests.packages.urllib3.exceptions import InsecureRequestWarning

requests.packages.urllib3.disable_warnings(InsecureRequestWarning)

functionNameArg = sys.argv[1:][0]
inputArg = sys.argv[1:][1]
numOfProcessesArg = int(sys.argv[1:][2])
testerHostArg = sys.argv[1:][3]
testerKeyArg = sys.argv[1:][4]

mysqlHost = "172.24.63.183"
apiHost = "172.24.42.82"
namespace = "tester{0}".format(testerKeyArg)
k8sHost = "172.24.42.47:30899"

ramLoader = 69999
```

```
def runPrimeNumber(inputNum):
	return "https://{0}:443/api/v1/web/{1}/default/prime-number.json?num={2}"
		.format(apiHost, namespace, inputNum)
# Other function URLs are defined similarly


funcOptions = {
	"prime-number": runPrimeNumber
	# + other functions
}


def get(url):
	try:
		time.sleep(0.001)
		return requests.get(url, verify=False)
	except:
		print("Delay due to a request error")
		time.sleep(0.01)
		return get(url)


def runInParallel(functionName, input, numOfProcesses):
	print("Starting testing on host {0} with namespace {1}..."
	.format(testerHostArg, namespace))

	with concurrent.futures.ThreadPoolExecutor(
	max_workers=numOfProcesses) as executor:
		futures = executor.map(
			get, [
				funcOptions[functionName](input)
				for _ in range(numOfProcesses)
			])
		for response in futures:
			if response is None:
				print('No response was received')
			else:
				print('{0} {1}: {2}'.format(time.time(),
					response.status_code, response.content))


runInParallel(functionNameArg, inputArg, numOfProcessesArg)
```

Bibliography

[1] "Openwhisk system details." https://github.com/apache/incubator-openwhisk/blob/master/docs/reference.md. Accessed: 2018-10-12.

[2] "System overview." https://github.com/apache/incubator-openwhisk/blob/master/docs/about.md#the-internal-flow-of-processing. Accessed: 2018-10-09.

[3] "Ibm cloud functions." https://console.bluemix.net/openwhisk/. Accessed: 2018-09-08.

[4] P. Nasirifard, A. Slominski, V. Muthusamy, V. Ishakian, and H. A. Jacobsen, "A serverless topic-based and content-based pub/sub broker: Demo," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Posters and Demos*, pp. 23–24, ACM, 2017.

[5] I. Baldini, P. C. Castro, K. Chang, P. Cheng, S. J. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. M. Rabbah, A. Slominski, and P. Suter, *Serverless Computing: Current Trends and Open Problems*, pp. 1–20. Springer Singapore, 2017.

[6] "Apache openwhisk documentation." https://openwhisk.apache.org/documentation.html#documentation. Accessed: 2018-10-12.

[7] "Kubernetes." https://kubernetes.io/. Accessed: 2018-10-12.

[8] E. van Eyk, A. Iosup, C. L. Abad, J. Grohmann, and S. Eismann, "A spec rg cloud group's vision on the performance challenges of faas cloud architectures," in *Proceedings of the 2nd International Workshop on Serverless Computing*, pp. 1–4, ACM, 2018.

[9] T. Back and V. Andrikopoulos, "Using a microbenchmark to compare function as a service solutions," in *Service-Oriented and Cloud Computing*, pp. 146–160, Springer International Publishing, 2018.

[10] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: distributed computing for the 99%," in *Proceedings of the 2017 Symposium on Cloud Computing.*, pp. 445–451, ACM, 2017.

[11] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," in *IEEE Transactions on Parallel and Distributed Systems*, pp. 931–945, IEEE, 2011.

[12] M. Malawski, K. Figiela, A. Gajek, and A. Zima, "Benchmarking heterogeneous cloud functions," in *Euro-Par 2017: Parallel Processing Workshops*, pp. 415–426, Springer International Publishing, 2018.

[13] H. Lee, K. Satyam, and G. Fox, "Evaluation of production serverless computing environments," 2018.

[14] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverlesscomputing: An investigation of factors influencing microservice performance," in *Proceedings of the IEEE International Conference on Cloud Engineering*, IEEE, 2018.

[15] N. Kaviani and M. Maximilien, "Cf serverless: Attempts at a benchmark for serverless computing." `https://docs.google.com/document/d/1e7xTz1P9aPpb0CFZucAAI16Rzef7PWSPLN71pNDa5jg/edit`, 2018. Accessed: 2018-08-16.