Universitat Rovira i Virgili

WebAssembly for Edge-Cloud Computing

Final Master's Project

Marlon Funk

directed by Dr. Marc Sánchez Artigas

Computer Security Engineering and Artificial Intelligence

September 4, 2023

Abstract In recent years, serverless computing has arisen as popular solution for building and deploying applications in cloud environments. One of the leading platform in this field is Apache OpenWhisk, which offers a severless framework for executing event-driven functions in response to various triggers. While having recognized efficiency, its underlying technologies are still evolving to fulfill constantly increasing demands. The runtime environment for function execution is a key aspect of severless computing. As a binary instruction format for virtual machines Wasm has gained attention for its portability and performance. The aim of this work is to investigate and evalutate the use of WebAssembly (Wasm) as an alternative to the current Docker containerization approach. The actual substitution was implemented before, however several questions regarding the performance differences between Wasm and Docker remain unanswered. After conducting several benchmarks, the superiority of Wasm regarding latency and capacity can be confirmed. Additionally, it shows fewer outlier occurrences compared to Docker and Wasm causes less strain on memory and CPU usage.

Contents

1 Introduction **3**
1.1 Motivation 3
1.2 Objectives 4

2 Related Work **4**

3 Background **5**
3.1 OpenWhisk 5
3.2 Webassembly 8
3.3 WebAssembly-flavored OpenWhisk (WoW) 9

4 Methodology **11**
4.1 Measuring of resources 12
4.2 Cold start 14
4.3 Concurrency 14
4.4 Capacity 17
4.5 Summary information for actions 17

5 Conduction of benchmark **17**
5.1 Cold-start tests 18
5.2 Concurrency tests 21
5.2.1 Test execution without load 21
5.2.2 Test exectuion with hash load 24
5.2.3 Test execution with mixed type of load 26
5.3 Comparison of performance 36
5.4 Capacity tests 38

6 Identifying Potential Enhancements **40**
6.1 Enhance cold-starts 40
6.2 Enhance warm-starts 42

7 Conclusion **44**

1 Introduction

1.1 Motivation

The integration of Wasm within OpenWhisk itself is not a novel idea, however with an increased level of detail and transparency it is possible to add improvements to the existing work. A solid foundation for the integration of Wasm runtimes into the existing OpenWhisk architecture is implemented in "Pushing Serverless to the Edge with WebAssembly Runtimes" (Gackstatter et al., 2022)[17]. Which was presented at the 2022 22nd IEEE International Symposium on Cluster, Clound and Internet Computing. We intend to extend this research by increasing the transparency and adding a different type of test to the evaluation. Thus providing further insights to the existing research.

While Gackstatter et al. did a good job for implementing a concurrent executor built upon Rust's async features, the evaluation in the paper [17] was incomplete. For instance, the authors did not realize capacity tests to determine how many concurrent activations can be performed with containers vs. WebAssembly with the same set of resources. Further, Gackstatter et al. did not present the implementation details of the actions, which were used in the benchmark. In addition benchmarks with mixed types of loads are missing from the assessment. Our evaluation is much complete and we develop some scripts to evaluate new WebAssembly-based additions to OpenWhisk in the future.

One of the major advantages of serverless computing is auto-scaling. These polygot platforms are designed to handle varying workloads by automatically scaling resources up or down as needed. With this arises the challenge of managing unpredictable workloads efficiently. In this regard the latency is of high importance to enable support for critical services. We aim to address those points by conducting various tests. To evaluate the difference in cold-start latencies between Wasm modules and Docker containers, the cold-start latencies for an increasing amount of concurrent requests are measured. In order to gain insights into the behaviour under heavy load from concurrent requests, the time needed for initialization is measured for a growing quantity of concurrent requests. We conduct these tests with different combinations of CPU- and IO-load. Subsequently, the maximum capacity for multiple instances of containers and modules are analyzed to obtain knowledge regarding the value of density for Docker and Wasm within OpenWhisk. With the knowledge gained from conducting the benchmarks, potential bottlenecks will be identified and possibilities for enhancements proposed.

To accomplish our goal, first the essential parts of the underlying theory will be explained. Based on that knowledge, the methodology of this work will be laid out in detail for each conducted experiment. Subsequently, the collected results will be compared and evaluated in detail. Afterward, with new insights in the topic of performance, critical steps in the architecture will be examined, and the time needed measured, leading to the final conclusion.

By substituting Docker with Wasm, the cold and warm latencies will be reduced while increasing the total capacity of concurrently running instances, leading to a higher number of requests being addressed more quickly and simultaneously.

1.2 Objectives

To conclude, the main goals of this thesis are the following ones:

1. Gain knowledge about the Function as a Service model and, in particular, the open-source OpenWhisk platform;
2. Understand WebAssembly (Wasm) and the internals of a modified OpenWhisk platform that replaces containers with a more lightweight isolation model built upon Wasm technology.
3. Design a test suite to capture the benefits of Wasm over Docker containers (if any) in terms of performance and resource footprint.
4. Run the benchmarking suite and compare the performance of both runtimes: Wasm and containers, and provide new insights.

2 Related Work

In order for the area of serverless computing to gain more significance for applications where low latency is crucial, it is of high importance to reduce the current cold-start latencies. These latencies, which can take as long as 24 seconds, as mentioned in [22], need to be minimized to make serverless computing more attractive for latency-sensitive tasks. This reduction in cold-start latencies is essential to make serverless computing more appealing for applications where high responsiveness is a key factor.

The utilization of WebAssembly to improve on the cold-start latencies of the current container technology has been discussed before in multiple works [20], [18], [23]. In [20] a new execution model for serverless functions at the edge is proposed, taking the OpenWhisk architecture as a role model. The model addresses the challenges of latency-sensitive applications by optimizing the execution of serverless functions. By deploying functions closer to the end-users, it aims to reduce latency and enhance responsiveness. The paper focuses on the architecture, explaining in detail how functions are triggered, executed, and managed. The proposed model aims to improve the overall performance of edge computing environments by efficiently handling the execution of serverless functions. The evaluation sets the focus on the whole time needed to instantiate the context, execute the function, and return the result. Thus making the results specific for their use cases. This master's thesis aims to provide more general information about cold-starts and warm-starts respectively, for a growing amount of concurrent requests.

Gadepalli et al. discuss challenges and potential advantages for efficient serverless computing at the edge in [18]. They address the topics of managing resources, minimizing latency, and optimizing serverless function execution by identifying ways for enhancing resource allocation, load balancing, and execution strategies. Points that are not addressed are concurrent requests and different types of workloads at the same time, which will be improved on in this work.

Shillaker and Pietzuch (2020) proposed a new framework for efficient stateful serverless computing called Faasm[23]. This framework utilizes WebAssembly for lightweight isolation while supporting shared memory. Additionally, the runtime *FAASM* is able to isolate the CPU and the networking using Linux cgroups[10]. To train a machine learning model, they achieve a 2-time speedup using 10 times less memory compared to traditional container platforms while doubling the throughput and reducing tail latency by 90%. For us, Faasm is of relevance due

to the possibility of replacing WebAssembly with Fasslets in the OpenWhisk system.

The primary reference paper utilized in this thesis is expanded on in the area of confidential serverless computing in [24]. This paper addresses the security challenges when executing functions with sensitive data by implementing reusable enclaves. To overcome cold-start issues, the the original WoW project is extended with enclaves that are resettable. The use and adaptation of WoW in the area of security enforces the importance of reevaluating the obtained results in [17].

3 Background

3.1 OpenWhisk

OpenWhisk[3] is an serverless event-based programming service. Actions (stateless functions) can be created in a variety of different programming languages and are executed after a defined trigger is activated. A trigger can be anything from a change in a document to data arriving from a sensor. With OpenWhisk its possible to deploy containers on local machines or in the cloud. Due to the compatibility with Kubernetes[9] its possible to create a flexible platform for serverless computing with the orchestration capabilities of Kubernetes.

An overview of the complete OpenWhisk system, including all components, is given by following the internal flow of an invocation of an action. The graphical representation of the OpenWhisk architecture can be found in Fig.: 1. The internal flow starts by entering a command in the wsk CLI, which sends an HTTP request to nginx[12]. Nginx is an open-source server that, among other features, implements reverse proxying, cacheing and load balancing. In OpenWhisk it is used to forward HTTP calls to the controller. The controller acts as an interface for all actions the user can take. It converts the HTTP call into an invocation of an action. Next, the controller conducts measurements of authentication and authorization by verifying the credentials contained in the request with credentials stored in the 'subjects' database in CouchDB[1]. A CouchDB server provides multiple named databases that can be accessed with a RESTful HTTP API. In the case of an incoming invocation request, it validates if the given user exists and if it has the required privilege to invoke this particular action. After ensuring the privileges of the user, the action is loaded from the 'whisks' database in CouchDB. In addition to the parameters given with the request to invoke the action, the database contains information about resource limitations. In the next step, the Load Balancer selects the best suitable Invoker. As part of the controller, the Load Balancer is able to see the status of all existing Invokers and thus is able to select the best-fitting one. Kafka[2] is responsible for handling the communication of messages between the Controller and Invoker. It is an open-source platform that implements high-performance data pipelines, among other functionalities. Those pipelines ensure the messages can be retrieved after a system crash. Through Kafka, the controller sends a message with the name and parameters to the Invoker, which returns an ActivationID. If the HTTP request is accepted, it terminates. Using Docker[5] for an action, an isolated container is created. The source code for the action and the parameters are given to the Docker container. After a defined timeout, the container is destroyed again. If a new invocation request for the same action arrives before the timeout expires, the container is reused, and the timeout for destruction is reset. After the execution, the Invoker extracts the results and stores them in the 'activations' database in CouchDB. An example of the result of a 'hello' action:

```
[
   {
```

```
 //The leading two fields are not part of the original result,
 //but from our test architecture
"no_concurrent_requests":1,
"responses":[
   {
      "activationId":"c5f8fe18ee504024b8fe18ee50f02489",
      "annotations":[
         {
            "key":"path",
            "value":"guest/hello1"
         },
         {
            "key":"waitTime",
            "value":129
         },
         {
            "key":"kind",
            "value":"wasm:0.1"
         },
         {
            "key":"timeout",
            "value":false
         },
         {
            "key":"limits",
            "value":{
               "concurrency":1,
               "logs":10,
               "memory":256,
               "timeout":60000
            }
         },
         {
            "key":"initTime",
            "value":100
         }
      ],
      "duration":105,
      "end":1692272392926,
      "logs":[

      ],
      "name":"hello1",
      "namespace":"guest",
      "publish":false,
      "response":{
         "result":{
            //The actual return value of the action
            "result":{
```

```
                    "result":"Hello, \"there!\""
                },
                "status":"success",
                "status_code":0,
                "success":true
            },
            "size":91,
            "status":"success",
            "success":true
        },
        "start":1692272392821,
        "subject":"guest",
        "version":"0.0.1"
    }
  ]
 }
]
```


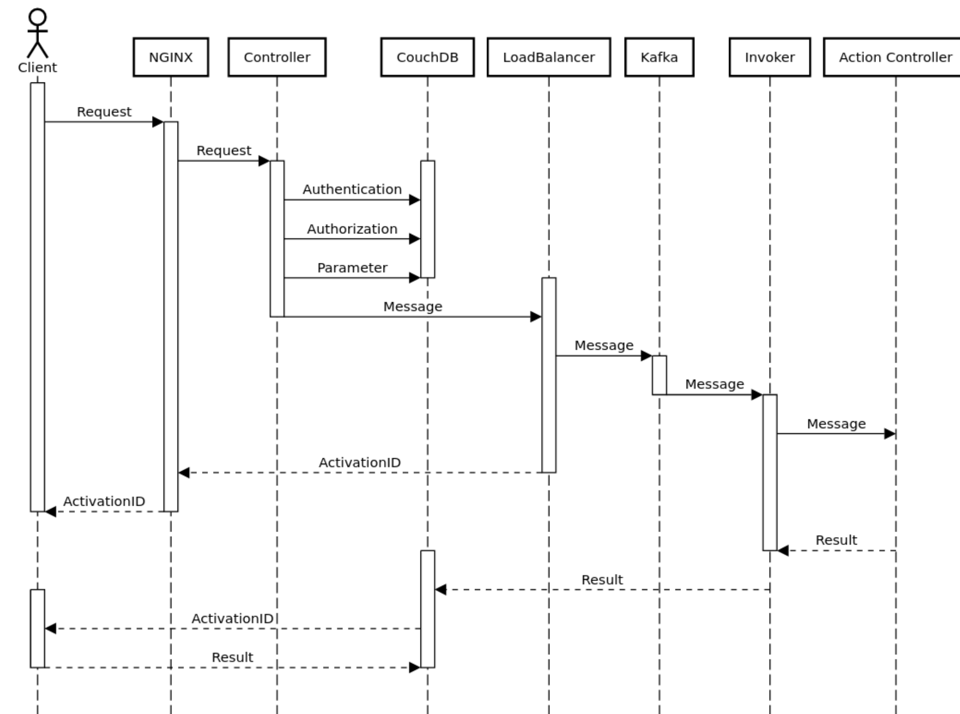

Figure 1: Sequence diagram of OpenWhisk (created with https://sequencediagram.org)

The source code of actions is unziped and stored in the variable called 'module'. This variable gets passed on from the Controller through multiple instances. This has an interesting effect in performance. First off, the size of the binaries will effect both network transmission and decompression. So, for large binaries the time to fetch, unzip and run the code will be longer.

3.2 Webassembly

WebAssembly (Wasm) stands for a binary instruction format developed for a stack-based virtual machine. It serves as a portable compilation target for various programming languages and is most commonly used for web applications on both client-side and server-side[15]. Wasm comes with both a compact representation and a fast execution[19].

Due to the platform independence of WebAssembly, syscalls are not defined. To be able to interact with the operating system, the use of syscalls is implemented by the WebAssembly system interface (WASI). The key goals of WASI are to "propose as a standard engine-independent non-Web system-oriented API for WebAssembly"[4]. Among others, it provides access to the filesystem, to sockets, to clocks and to random numbers via a standardized interface between Wasm modules and the host operating system[4]. The design principles are defined in the WASI GitHub repository[7]:

- Capability-based security: Access to external resources relies on two kinds of capabilities: Handles and link-time capabilities. Handles are unforgeable, ensuring that an instance can only get access to a handle when another instance shares it. Link-time capabilities are used in scenarios where there is no need to identify multiple instances of a resource at runtime.
- Interposition: The capability of a Wasm module to implement a WASI interface in addition to a consumer Wasm module being able to utilize this implementation.
- Compatibility: Compatibility is provided between different platforms, libraries, and applications.
- Portability: Is defined explicitly for each API.

A WebAssembly consists of a single linear memory that can be accessed with 32 bit pointers. All variables and functions are represented by integer indices on the linear memory and can be accessed via $main, for example. A garbage collector is not provided[21]. As can be seen in Fig. 2 the stack grows in the direction of __data_end and the heap towards mem_max[16].

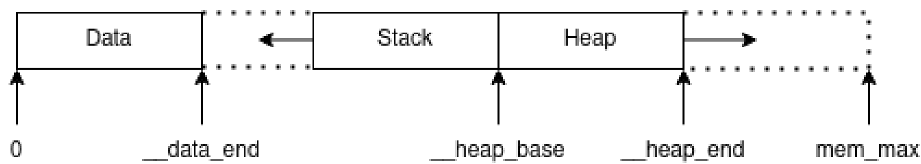

Figure 2: Memory layout of a Wasm module(created with https://app.diagrams.net/)

To give a simple overview over the relations between a C-Programm and the same code as Wasm here a simple "Hello World" in C and the corresponding Wasm counterpart are displayed:

```
#include <stdio.h>
int main() {
   printf("Hello, World!");
   return 0;
}
```

```
(module
 (type $FUNCSIG$ii (func (param i32) (result i32)))
 (type $FUNCSIG$iii (func (param i32 i32) (result i32)))
 (import "env" "printf" (func $printf (param i32 i32) (result i32)))
 (table 0 anyfunc)
 (memory $0 1)
 (data (i32.const 16) "Hello, World!\00")
 (export "memory" (memory $0))
 (export "main" (func $main))
 (func $main (; 1 ;) (result i32)
  (drop
   (call $printf
    (i32.const 16)
    (i32.const 0)
   )
  )
  (i32.const 0)
 )
)
```

In the given Wasm binary all variables are of the type i32. Besides i32 the following types are available[15]:

- Numeric Types:
  - signed and unsigned integers: i32 and i64.
  - single and double precision floats: f32 and f64.
- Vector Types: Can be processed by vector instructions, represented by a 128 bit vector.
- Reference Types: Refers to objects in the runtime.
- Value Type: Sets the values that Wasm code can compute.
- Result Types: Consists of a sequence of values as a result of a function.
- Function Types: Defines how parameters are mapped to results.
- Limits: Sets the size of memory and table types.
- Memory Types: Defines are region in memory.
- Table Types: Represents a table of elements.
- Global Types: Global variables can be accessed in a global scope.
- External Types: Variables and values that are imported.

3.3 WebAssembly-flavored OpenWhisk (WoW)

"Pushing Serverless to the Edge with WebAssembly Runtimes" [17] proposes an extended version of OpenWhisk that is able to run Wasm modules in addition to Docker containers. They name it "WebAssembly execution in Apache OpenWhisk", shortly WoW. The changes in WoW take effect after the LoadBalancer messages the Invoker through Kafka. The updated system

architecture can be seen in Fig.: 3, the Invoker injects the code into the Wasm Executor leading to the creation of the Wasm module. Subsequently, the Invoker instructs the Wasm Executor to start the module.

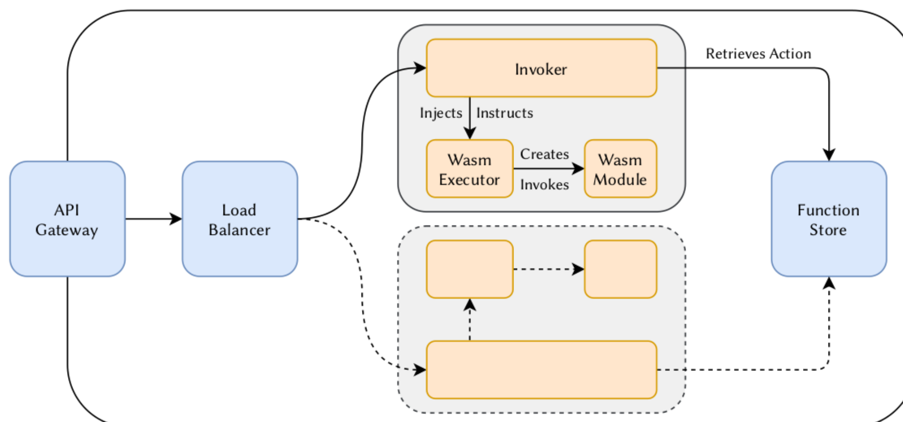

Figure 3: WoW system overview [3]

The architecture of the Wasm Executor is shown in Fig.: 4. It offers three entry points to be accessed by the Invoker */init*, */run* and */destroy*. For the container to be initialized, with the code of the module, */init* is invoked once for each container. To initialize a WebAssembly module, the request must contain the container_id, the capabilities and the module_bytes, which is of type Vec⟨u8⟩. The entry point */run* is used to start the container. In order to run a WebAssembly module, the container_id and optionally the parameters are required. The destruction of the container can be ordered with */destroy*. To destroy a WebAssembly module, only the container_id is needed.

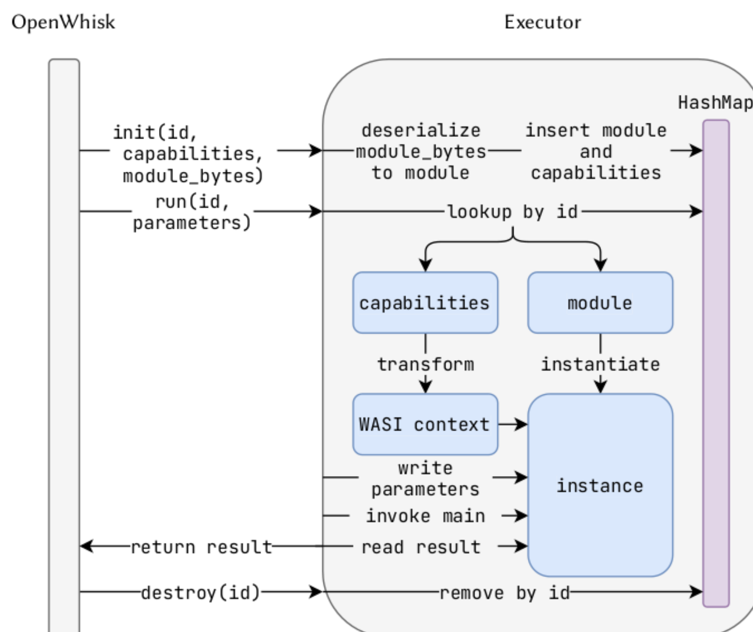

Figure 4: Wasm executor[17]

To enable support for WebAssembly modules in OpenWhisk, WoW adds these additions to the original OpenWhisk repository [17]. Through changes made in 'openwhisk/ansible/-files/runtimes.json' the new system is able to support the runtime 'wasm'. In 'common/scala/s-rc/main/resources/application.conf' the maximum for concurrent actions is set to '10'. We expanded this limit to '15'. The limit for actions per minute is removed in 'core/controller/src/-main/scala/org/apache/openwhisk/core/entitlement/Entitlement.scala'. Furthermore, the Invoker is now able to handle wasmruntimes thanks to small changes made in 'core/invoker/init.sh' and 'core/invoker/Dockerfile'. To be able to invoke a module, the containerpool 'openwhisk/-core/invoker/src/main/scala/org/apache/openwhisk/core/containerpool/' is expanded by three files:

- WasmClient.scala: defines a class named WasmClient that implements the WasmRuntimeApi trait. This client also implements functions to start and destroy modules.
- WasmContainer.scala: implements a WasmContainer class and a WasmContainer object. The class represents a WebAssembly module. It provides a method for obtaining the logs of modules, and it adds the desctruction of the runtime to to destroy of the container. The 'WasmContainer' object provides a method for creating a WebAssembly module using the WasmRuntimeApi.
- WasmContainerFactory.scala: acts as a factory to create WebAssembly modules. It uses the WasmClient to interact with the runtime.

In addition, in 'StandaloneOpenWhisk.scala' the system property of 'whisk.spi.ContainerFactoryProvider' is set to 'org.apache.openwhisk.core.containerpool.wasm.WasmContainerFactoryProvider'.

4 Methodology

All tests are executed with Docker containers and with Wasm modules. In the case of concurrency, the tests are conducted with different types of loads. Each test is repeated 10 times, which allows us to capture the variability of execution, including information about the coefficient of variation (CoV) and the standard deviation.

The test execution for Wasm modules was conducted with the OpenWhisk fork from WoW[17]. For the test execution of Docker containers, a slightly modified version of the original OpenWhisk repository was used. In 'OriginalOW/openwhisk/core/invoker/src/main/resources/application.conf' the idle-container timeout was reduced to 10 seconds. This change decreases the duration of the cold start tests by a margin. All used scripts and files, as well as the results, can be in this repository: https://github.com/MarlonFunk/MasterThesis.

Before beginning the benchmark, the maximum number of concurrent containers and modules has to be defined. When flooding the system with many requests, the log messages of OpenWhisk give the required size of one container or one module. The same size is allocated for all types of our actions:

```
[ContainerPool] Rescheduling Run message, too many message in the pool,
freePoolSize: 0 containers and 0 MB, busyPoolSize: 4 containers and 1024 MB,
maxContainersMemory 1024 MB, userNamespace: guest,
action: ExecutableWhiskAction/guest/prime@0.0.1,
needed memory: 256 MB, waiting messages: 0
```

From this log message, the initial configuration of 'maxContainersMemory' with 1024MB can be read. This amount is not sufficient to run enough concurrent requests with our tests. The amount of invoker memory can be defined in:

```
OpenWhisk/openwhisk/core/invoker/src/main/resources/application.conf:
userMemory: "{{ invoker_user_memory | default('1024 m') }}"
```

As an experiment, the value was set to '8192 m'. This should exceed the actual available memory of 8GB and give the maximum amount of concurrent running Wasm modules and Docker containers. To validate the behaviour the capacity test for Wasm was executed, starting with 30 concurrent requests. During this experiment, the following log message was returned from the server:

```
freePoolSize: 0 containers and 0 MB, busyPoolSize: 32 containers and 8192 MB,
maxContainersMemory 8192 MB
```

The actual maximum measured RSS memory was 63,332 KB for all executor threads. The details of how the measurements were done are explained in the following section. From this measurement, we can determine that the actual size of one Wasm module with the action 'sleep' is slightly less than 2MB. With 8GB of RAM about 4000 concurrent requests should be possible. To exceed this limit, we set 'invoker_user_memory' to 1,280,000:

$$5000 \text{ concurrent actions} * 256MB = 1280000MB$$

Additionally, the limits for the invocations of actions were increased:

```
OpenWhisk/openwhisk/core/standalone/build/resources/main/standalone.conf:
  config {
    controller-instances = 1
    limits-actions-sequence-maxLength = 50000
    limits-triggers-fires-perMinute = 50000
    limits-actions-invokes-perMinute = 50000
    limits-actions-invokes-concurrent = 50000
  }
```

With those configurations, all tests were conducted.

Alternatively, the action memory configuration could have been adapted, but the previously mentioned solution seems to be less complicated.

```
OpenWhisk/openwhisk/common/scala/build/resources/main/application.conf:
# action memory configuration
memory {
    min = 128 m
    max = 512 m
    std = 256 m
}
```

4.1 Measuring of resources

To measure the resources of running Wasm modules, a bash script is executed to measure the resources used. The ps command[13] is used with the parameter $-o$ to control the formatting of the output. The following specifiers are used, as defined by[13]:

- spid: The thread ID.
- stat: The current status of the process or thread.
- %cpu: The CPU utilization as a percentage. Computed by dividing CPU time by the time the process has been running.
- rss: The resident set size.

The output returned by ps is then piped to awk[14] which is used to only show running processes by filtering 'stat'. If the output of awk is non-empty the used resources are piped into a.log file.

```
#!/bin/bash

outfile="wasm_res.log"

echo "SPID STAT %CPU RSS" >> "$outfile"
while [ true ]; do # Keep logging until cancelled
    stats_executor=$(sudo ps -o spid,stat,%cpu,rss= -C "executor" )
    # Filter by state
    running=$(echo "$stats_executor" | awk '$2 ~ /R/ {print}')
    # Only log if not empty
    if [[ ! -z "$running" ]] ; then
        while IFS= read -r info; do
            echo "$info" >> "$outfile"
        done <<< "$running"
    fi
done
```

To measure the consumed resources by Docker containers, the command 'docker-stats' [6] is utilized. This command returns data for all running instances of Docker containers, including information about the CPU usage in percent and the total used memory. The gained data for each container is then summed up and saved in a log file.

```
#!/bin/bash

outfile="docker_res.log"
while [ true ]; do # Keep logging until cancelled
    docker_stats_output=$(docker stats --no-stream
            --format "table {{.Name}}\t{{.CPUPerc}}\t{{.MemUsage}}")

    cpu_sum=$(echo "$docker_stats_output" |
            awk 'NR > 1 { total += $2 } END { print total }')
    mem_sum=$(echo "$docker_stats_output" |
            awk 'NR > 1 { sub(/[A-Za-z]+/, "", $3); total += $3 }
            END { print total "MiB" }')

    echo "CPU%: $cpu_sum"" Memory: $mem_sum" >> "$outfile"
done
```

4.2 Cold start

A cold start refers to a container or module being initialized for the first time. In the case of Docker, this includes the creation of the container, which is more time consuming than executing a precompiled Wasm module. A cold start latency consists of the sum of waitTime and initTime. The waitTime refers to the duration needed for OpenWhisk to handle an activation request, from the moment the Controller receives the request until the Invoker provisions a container or module[3]. The initTime represents the time taken for initializing the function. This value is only returned for cold starts, as for a warm start, no initialization is required[3].

In this test, the cold start times of 1 to 15 concurrent requests are measured. With the binary cold-start-test created with an adapted version from the rust source code[17] an increasing number of concurrent requests are sent to the server. After each cycle of requests, the program waits for a defined time before the next concurrent requests are sent. That ensures the destruction of containers or modules, and in each cycle, a cold start is forced. The correct behaviour can be validated by counting how often initTime is returned for each cycle of concurrect requests. For each cold start, one initTime is required, thus after 15 cycles, 120 values are expected. If a container has not been deallocated and thus a warm start was executed, no initTime is returned. Also, the amount of initTime's in the result file must equal the amount of waitTime's. With this information, the correct execution of each test can be ensured.

The cold start tests are conducted an action named "hello" which simply prints:

```
Hello, < Input >
```

The use of an action with a low amount of lines of code and without adding libraries showcases the lower border of the cold start times. No CPU or memory-intensive computation is required.

4.3 Concurrency

Concurrency tests measure the impact of parallel execution of multiple actions on the starting time of containers and modules. For this test, we only validate the waitTime, in contrast to the preceding cold start test.

We reuse the concurrency.rs script from WoW[17]. A defined number of seconds is saved in the global variable 'INCREASE_AFTER', the default is 5 seconds. For this time period, the same amount of concurrent requests are sent. Initially, two concurrent requests are executed. The script stops after the amount of requests exceeds the value defined in 'ABORT_AFTER', which has a default value of 15. Directly before the test execution, a manual start of the corresponding action was done. This ensures that the first invocation in the actual test execution is a warm-start. After each period of 'INCREASE_AFTER' an additional concurrent request is sent. The resulting cold-start was later filtered out of the results.

Here three different cases of loads are interesting. To commence, the executed action is not resource demanding, displaying the ability of the OpenWhisk system to handle concurrent requests. For this scenario, we reuse the 'hello' action. In the next step, we use a hash function that takes as input a number of iterations as an integer and a string. The hash function is repeatedly executed on the input string for the given number of iterations, and the output is reused for the next iteration. The resulting computation is a good fit for our needs, as it has a constant time complexity and solely focuses on the CPU, not requiring a huge amount of

memory. In [17] an example for rust is already provided, 'hash.rs'. We reuse this example for the test execution with Wasm modules. The essential part of the rust code consists of:

```
let hash = {
    let mut prev_output;
    let mut hash = input.as_bytes();

    for _ in 0..iterations {
        prev_output = blake3::hash(hash);
        hash = prev_output.as_bytes();
    }
    hash.to_vec()
};
```

To be able to create an OpenWhisk action from code written in Rust, an additional fork of the original OpenWhisk repository is required. In order to validate the behaviour of the original OpenWhisk implementation instead of the fork, we decided to implement a hash function in Golang [8] that is used to create the test action for Docker containers. For a defined number of iterations, the hash algorithm is continuously repeated. In Golang, the following code was utilized:

```
hash := sha256.New()
hash.Write([]byte(input))
hashSum := hash.Sum(nil)

for i := 0; i < iterations; i++ {
    hash.Write(hashSum)
    hashSum = hash.Sum(nil)
}
```

The input parameters for the hash action are the following:

- var: "iterations", value: 100000. Defines the amount of iterations for the hashing to repeat.
- var: "input", a random string as the first input for the hashing algorithm.

To verify how the system behaves when not all functions are CPU-intensive, a mixed workload is created. Here fore we reuse 'prime.rs' and 'net.rs' ,which were also implemented in [17] for the evaluation of Wasm modules. In 'prime.rs' , the 20,000,001st prime number is calculated using the 'primal' crate:

```
let p = primal::Primes::all().nth(20000001 - 1).unwrap();
```

In the case of Docker, again, an action in Golang was implemented to calculate the 5000th prime number. To be independent of non-default Go libraries, a simple function to check for primes was implemented. Due to less performance with this function compared to the 'primal' crate, a lower value was selected.

```
func isPrime(n int) bool {
    if n <= 1 {
        return false
    }
```

```
    if n <= 3 {
        return true
    }
    if n%2 == 0 || n%3 == 0 {
        return false
    }
    i := 5
    for i*i <= n {
        if n%i == 0 || n%(i+2) == 0 {
            return false
        }
        i += 6
    }
    return true
}
```

Finally, to evaluate the behaviour under a mixed type of load, we execute the 'prime' action and simulate an IO-intensive workload at the same time. An example of an IO-intensive workload is a blocking HTTP request, as implemented in 'net.rs' [17]. This action and the corresponding function in Golang implement a 300ms sleep. The straightforwardness of this functionality does not require to be shown here. To realize concurrent requests with a mixed load, the 'concurrency.rs' [17] script needs to be adapted. We named the new script 'concurrency_mixed.rs' and define five different configurations:

- Configuration 1: 10% IO-intensive & 90% CPU-%intensive
- Configuration 2: 25% IO-intensive & 75% CPU-%intensive
- Configuration 3: 50% IO-intensive & 50% CPU-%intensive
- Configuration 4: 75% IO-intensive & 25% CPU-%intensive
- Configuration 5: 90% IO-intensive & 10% CPU-%intensive

To create these different configurations in Rust, the functionality of 'rand::distributions::WeightedIndex' is used to randomly select the following action with defined probabilities:

```
let probabilities = [<Probability-first-action>, <Probability-second-action>];
let mut rng = thread_rng();
let distribution = WeightedIndex::new(&probabilities).unwrap();
let mut random_index = distribution.sample(&mut rng);
match random_index {
    0 => {
        futures.push(make_request(&first_path, &auth));

    }
    1 => {
        futures.push(make_request(&second_path, &auth));

    }
    _ => {
        panic!("Unexpected value for var: {}", random_index);
```

```
    }
}
```

Where ⟨first_path ⟩and ⟨second_path ⟩are used to invoke either of the actions.

4.4 Capacity

To get the maximum capacity of parallel running functions inside OpenWhisk an action that implements a sleep for 5 seconds is invoked with an increasing amount of concurrency. The sleep ensures that no module or container is deallocated before the test finishes. The test waits for all actions to finish before invoking again with a higher number of concurrency. As soon as an action fails, the test aborts and the error message is returned. After conducting the same test with Docker containers, we are able to compare the amount of functions that can be packed with WebAssembly compared to Docker containers, resulting in the density.

4.5 Summary information for actions

To give additional information about the actions, the number of lines of code and the size of the source used are listed. In the case of Wasm, 'Size' refers to the precompiled ' ⟨action-name ⟩-wasmtime.zip' file. For Docker, it represents the size of the compiled binary. The following tables show that the actions are roughly equivalent in terms of complexity. Naturally, a precompiled Wasm module is larger than a simple binary.

	Lines of code	Size	Type
hello.rs	13	555KB	behaviour showcasing
hash.rs	21	459KB	CPU-bound
prime.rs	19	559KB	CPU-bound
net.rs	15	426KB	IO-bound (simulated)
sleep.rs	23	455KB	behaviour showcasing

Table 1: Lines of code and size of source files for Wasm actions

	Lines of code	Size	Type
hello.go	18	14KB	behaviour showcasing
hash.go	33	21KB	CPU-bound
prime.go	51	18KB	CPU-bound
net.go	17	14KB	IO-bound (simulated)
sleep.go	17	14KB	behaviour showcasing

Table 2: Lines of code and size of source files for Docker actions

5 Conduction of benchmark

The tests were conducted on a middle-class laptop, thus only the relation between the measured times is of importance. The lowest values do not represent the best latency obtainable, but solely the required time for this specific machine. Read from '/proc/cpuinfo' and '/proc/meminfo' the hardware specifications are:

- CPU: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz,
  4 cores, cache size: 8192 KB

- RAM: Mem totaL: 7928.68 mB

As operating system a Linux debian with kernel '5.10.0-25-amd64' is used.

To be able to recognize accumulations, the outliers are plotted with a slight value of transparency. The mean value is represented by a red square, adding more information to the plot.

5.1 Cold-start tests

In the plots of this section, the y-Axis is labeld with 'Warm latency (ms)', which refers to the measured 'waitTime' in ms.

The most significant distinction between Docker and Wasm can be seen in the case of the required time for a cold start. The longest cold start for Docker measures almost 40 seconds, with a total of 8 requests taking more than 30 seconds. For one concurrent request, the maximum is 3.04 seconds, which increases almost linearly to 24 seconds with 15 concurrent requests. In the cases of 5 and 12 concurrent requests, outliers are present.

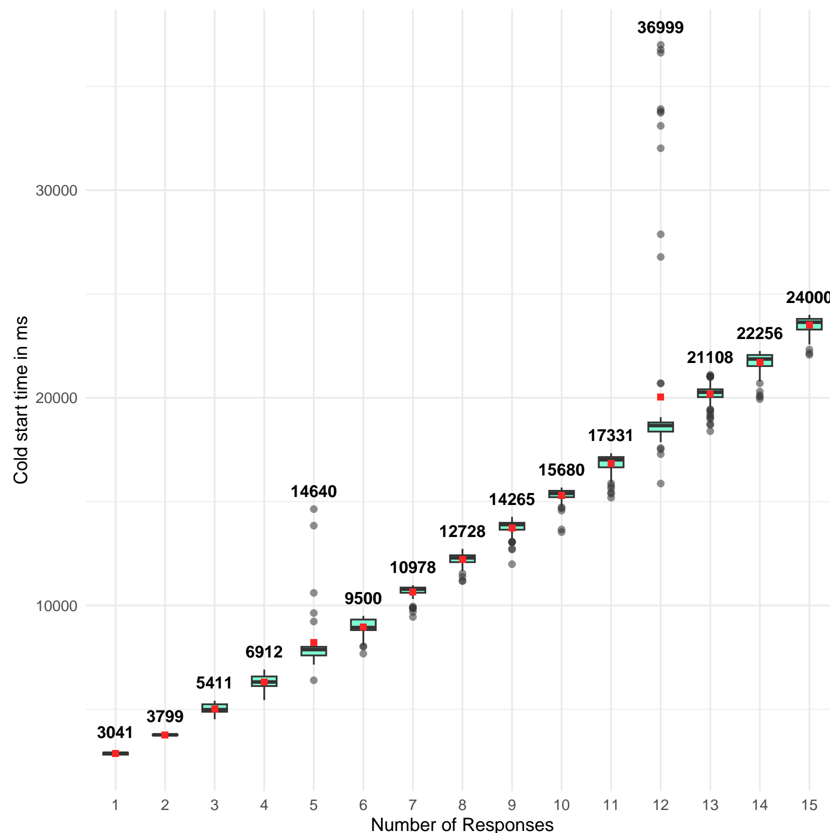

Figure 5: Cold-start test for Docker container

No. concurrent responses	Coefficient of Variation in %	Standard Deviation	Mean in ms
1	2.8952049	83.35295	2879.000
2	0.9208138	34.64102	3762.000
3	4.7713439	239.56918	5021.000
4	5.5690380	351.83268	6317.656
5	18.9652182	1556.41856	8206.700
6	4.9790999	446.30059	8963.479
7	3.2410157	345.38521	10656.696
8	2.7055461	330.21444	12205.094
9	2.8851798	396.49342	13742.417
10	2.4497701	374.52391	15288.125
11	2.7763803	467.37523	16833.977
12	23.2138688	4653.80035	20047.50
13	2.4350599	491.33608	20177.577
14	2.3154548	502.51034	21702.446
15	1.6593547	390.23321	23517.167

Table 3: Statistics for the Cold-start test for Docker container

Growing from 45ms to a maximum of 367ms maximum latencies measured with Wasm modules show a linear growth as well as the mean latencies, which displays identical behavior as Docker. The outlier for one concurrent request with 229ms is the first module instantiated. It is ensured that all values measured is from a cold start. Wasm outperforms Docker by a margin.

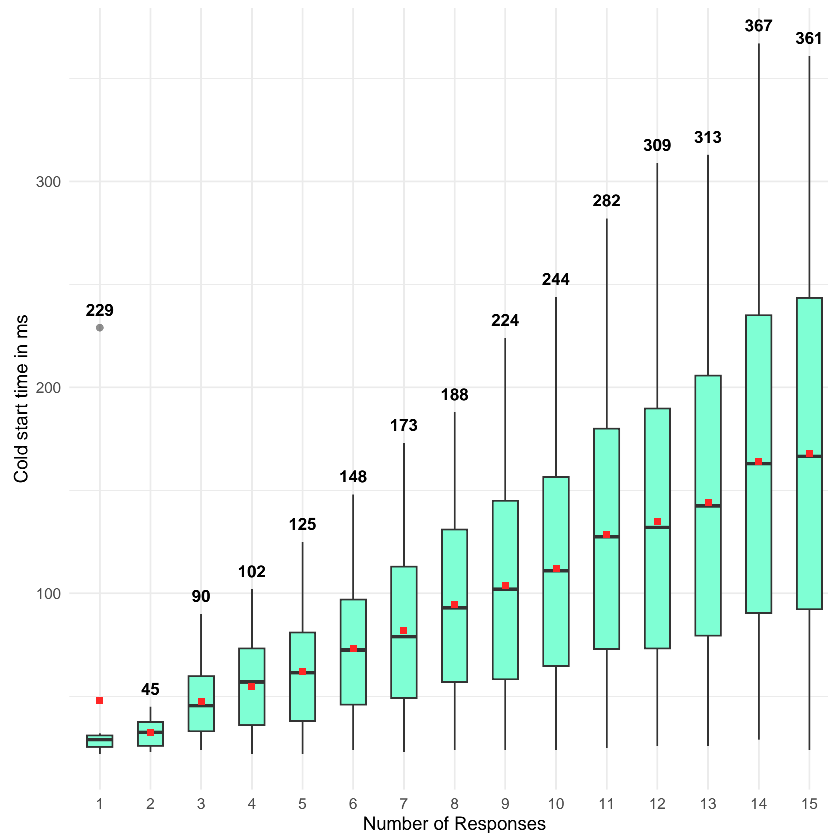

Figure 6: Cold-start test for Wasm module

No. concurrent responses	Coefficient of Variation in %	Standard Deviation	Mean in ms
1	133.39197	63.761361	47.80000
2	21.74142	7.044221	32.40000
3	33.53445	15.861795	47.30000
4	39.52966	21.642491	54.75000
5	42.89924	26.683328	62.20000
6	43.98450	32.233308	73.28333
7	46.73451	38.248856	81.84286
8	47.95512	45.347559	94.56250
9	50.00198	51.896503	103.78889
10	50.33115	56.370886	112.00000
11	51.09428	65.632929	128.45455
12	52.61631	70.918012	134.78333
13	50.90389	73.485642	144.36154
14	52.39570	85.906497	163.95714
15	52.59331	88.360267	168.00667

Table 4: Statistics for the Cold-start test for Wasm module

5.2 Concurrency tests

5.2.1 Test execution without load

In the case of Docker, the highest value measured is 57ms, which occurs for three concurrent requests. Until 8 concurrent requests, the mean values increase only slightly. Afterwards the growth accelerates, but only in small steps. The outliers are scattered more heavily than for Wasm. For 1 to 5 concurrent requests, the mean latency of Docker is lower than the means for Wasm, but not the maximum latency.

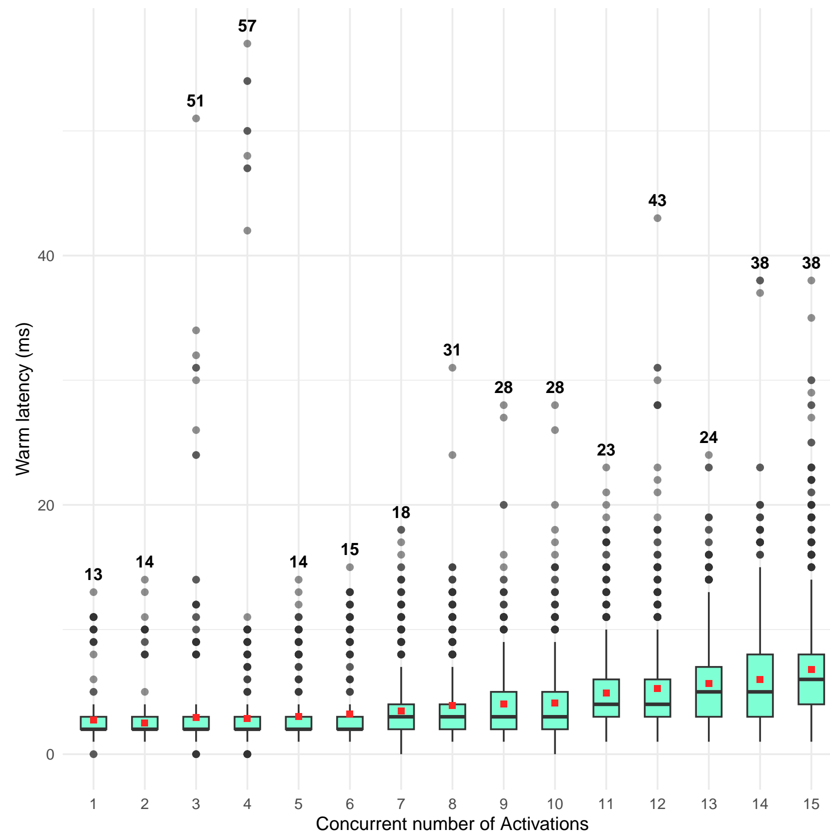

Figure 7: Concurrency test without load for Docker container

No. concurrent responses	Coefficient of Variation in %	Standard Deviation	Mean in ms
1	93.28363	2.548562	2.732057
2	73.33433	1.832381	2.498667
3	143.08814	4.190983	2.928952
4	191.32341	5.452845	2.850067
5	74.79927	2.256015	3.016092
6	75.45313	2.436387	3.229008
7	83.18531	2.886381	3.469821
8	76.49774	2.987159	3.904899
9	72.11771	2.901549	4.023352
10	69.77787	2.874676	4.119753
11	66.53349	3.272456	4.918510
12	73.75841	3.894727	5.280384
13	59.53093	3.385172	5.686408
14	66.72659	3.987966	5.976577
15	66.21479	4.495280	6.788936

Table 5: Statistics for the Concurrency test without load for Docker container

In this test scenario, Wasm shows marginal growth from a mean of 2.80ms to 4.84ms. With the exception of the maximum value 46ms all data points are closely spaced.

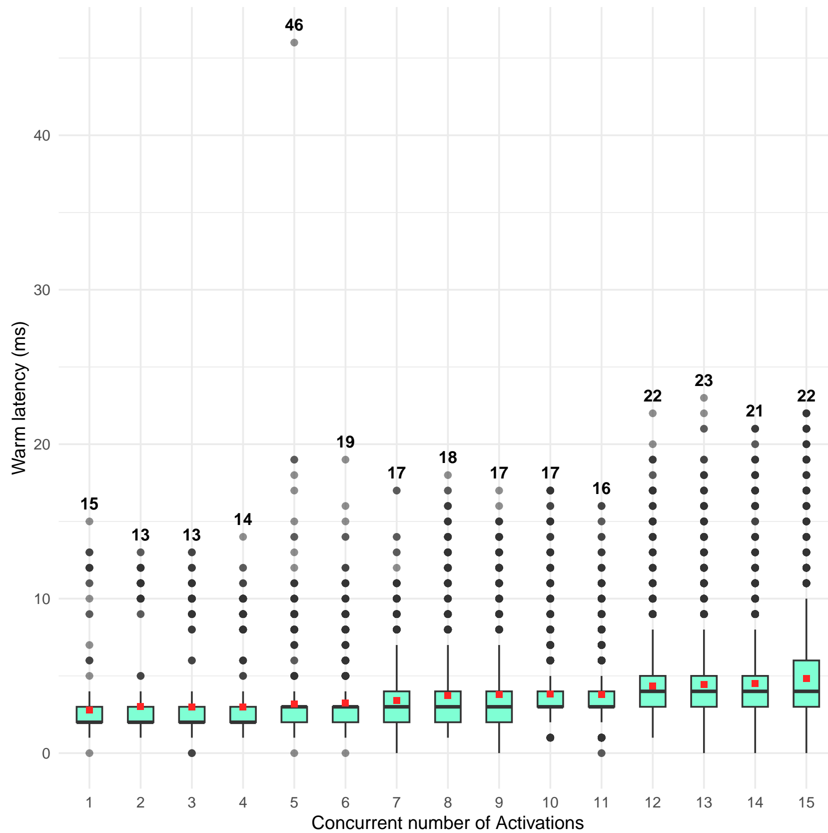

Figure 8: Concurrency test without load for Wasm module

No. concurrent responses	Coefficient of Variation in %	Standard Deviation	Mean in ms
1	92.28299	2.586586	2.802885
2	84.59156	2.559001	3.025126
3	82.91962	2.484764	2.996593
4	69.72616	2.082027	2.986005
5	82.97736	2.647694	3.190863
6	61.17928	1.983912	3.242784
7	60.31003	2.052996	3.404070
8	66.59699	2.495473	3.747126
9	64.87472	2.464868	3.799427
10	59.68282	2.294789	3.844974
11	53.29894	2.026897	3.802885
12	59.60124	2.581495	4.331278
13	60.89057	2.715076	4.458943
14	60.84321	2.740470	4.504151
15	62.68593	3.034308	4.840493

Table 6: Statistics for the Concurrency test without load for Wasm module

5.2.2 Test exectuion with hash load

For Docker without load, the maximum measured latency is 57ms by adding load, this value increases to 91ms. The growth exhibits the same behaviour as without load, with an increased advancement after 8 concurrent requests. The small size of the box plots for a lower amount of concurrent requests, as well as the low values for the standard deviation, show a high density.

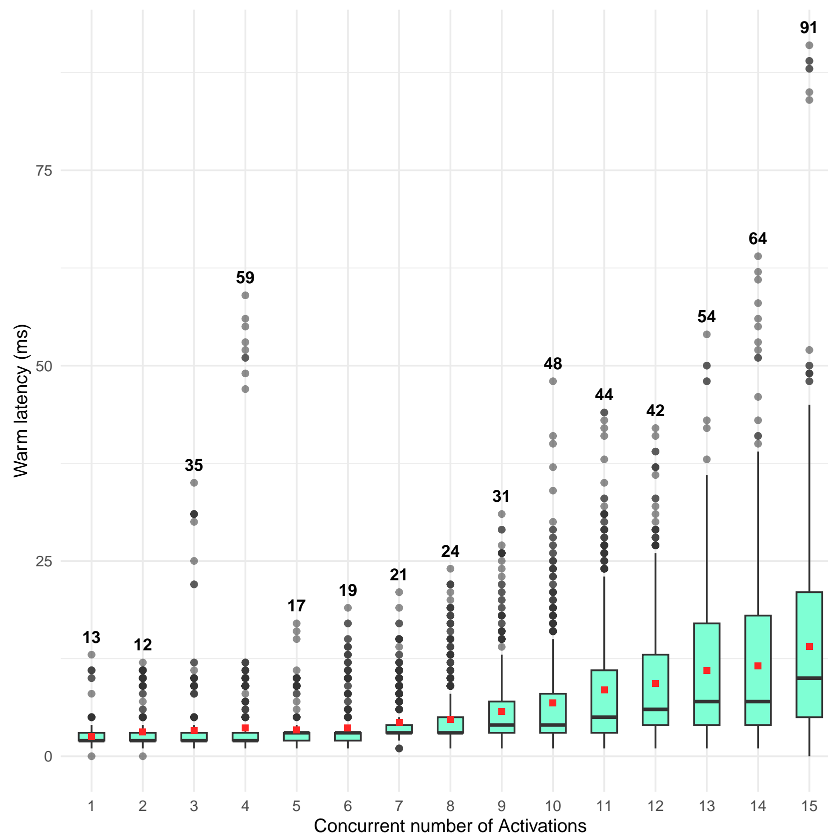

Figure 9: Concurrency test with hashing for Docker container

No. concurrent responses	Coefficient of Variation in %	Standard Deviation	Mean in ms
1	74.60631	1.909044	2.558824
2	72.55094	2.281087	3.144118
3	120.35861	3.952486	3.283925
4	177.21808	6.381626	3.601002
5	70.82116	2.369866	3.346269
6	77.03969	2.774665	3.601604
7	72.31940	3.125169	4.321343
8	82.66642	3.915222	4.736170
9	89.32469	5.113417	5.724528
10	93.26569	6.391071	6.852542
11	91.41650	7.744474	8.471637
12	84.32587	7.863878	9.325581
13	83.71822	9.224701	11.018750
14	90.36316	10.457244	11.572464
15	89.57771	12.624405	14.093243

Figure 10: Statistics for the Concurrency test with hashing for Docker container

With a standard deviation of 0.53 for one concurrent request and a maximum standard deviation of 2.55, the latencies for Wasm modules have a high density. The mean values are consistently lower than in the previous test scenario. The growth of mean latencies is extremely gradual, only in the maximum measured latency a change is visible. A reason for this behaviour could be the longer execution time for each Wasm module. As the test increases the amount of concurrent requests by time and with hashing load, each action takes more time, so the total amount of actions invoked in the same period of time is lower than with actions of less complexity. Thus, the strain on the OpenWhisk system is lower, and the measured mean latency values are smaller. The measured maximum latencies grow almost linearly.

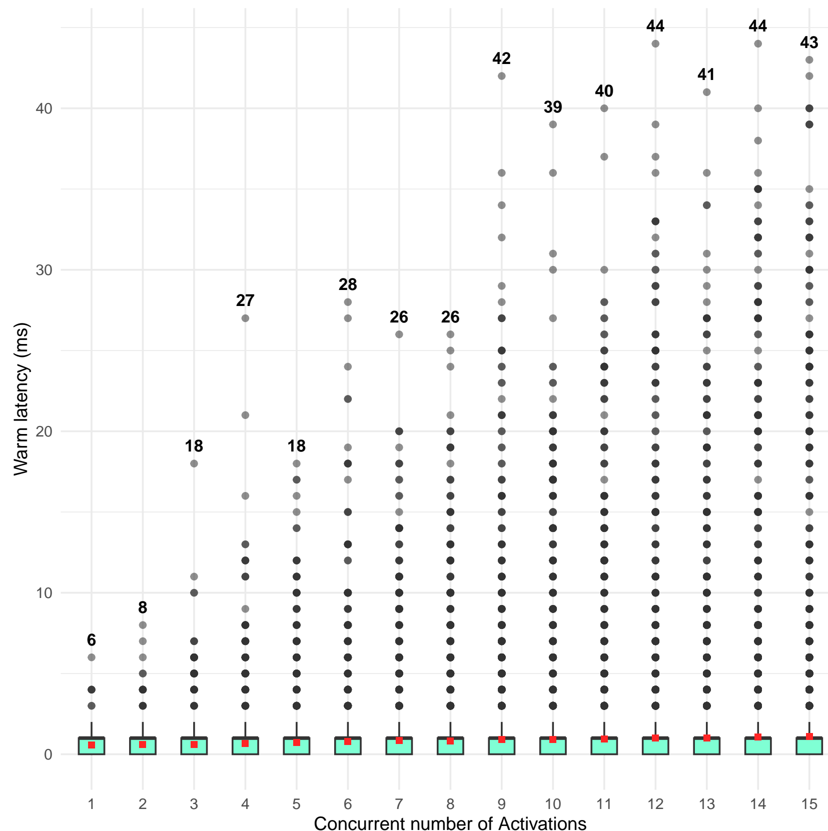

Figure 11: Concurrency test with hashing for Wasm module

No. concurrent responses	Coefficient of Variation in %	Standard Deviation	Mean in ms
1	91.55433	0.5370488	0.5865903
2	94.38191	0.5570982	0.5902595
3	111.05644	0.6772899	0.6098610
4	129.64084	0.8643753	0.6667462
5	122.20171	0.8950962	0.7324743
6	143.46836	1.1273378	0.7857745
7	145.64355	1.2469356	0.8561558
8	157.82147	1.3177278	0.8349484
9	192.07150	1.7436208	0.9077978
10	189.01211	1.7472731	0.9244239
11	195.90516	1.8746471	0.9569156
12	218.31174	2.1842222	1.0005061
13	202.81986	2.0239534	0.9979069
14	229.04933	2.4757957	1.0809007
15	233.72508	2.5527074	1.0921838

Figure 12: Statistics for the Concurrency test with hashing for Wasm module

5.2.3 Test execution with mixed type of load

In the different scenarios for mixed load, first the portion of IO-load and then the portion of CPU-load are given. For the first test, 'mixed load 10/90' this implies 10% IO-load and 90% CPU-load. Step by step, the IO-load is decreased and the CPU-load increased. For Docker, increasing the IO-load while decreasing the CPU-load leads to lower mean and maximum latencies, as well as a higher density consistently. In the case of Wasm, the mean values decrease too, but slower.

With mixed types of loads and thus different types of containers, the maximum latency measured is significantly higher. For Docker, the longest warm latency is 589ms, with multiple values over 400ms. The mean value does not rise above 5ms.

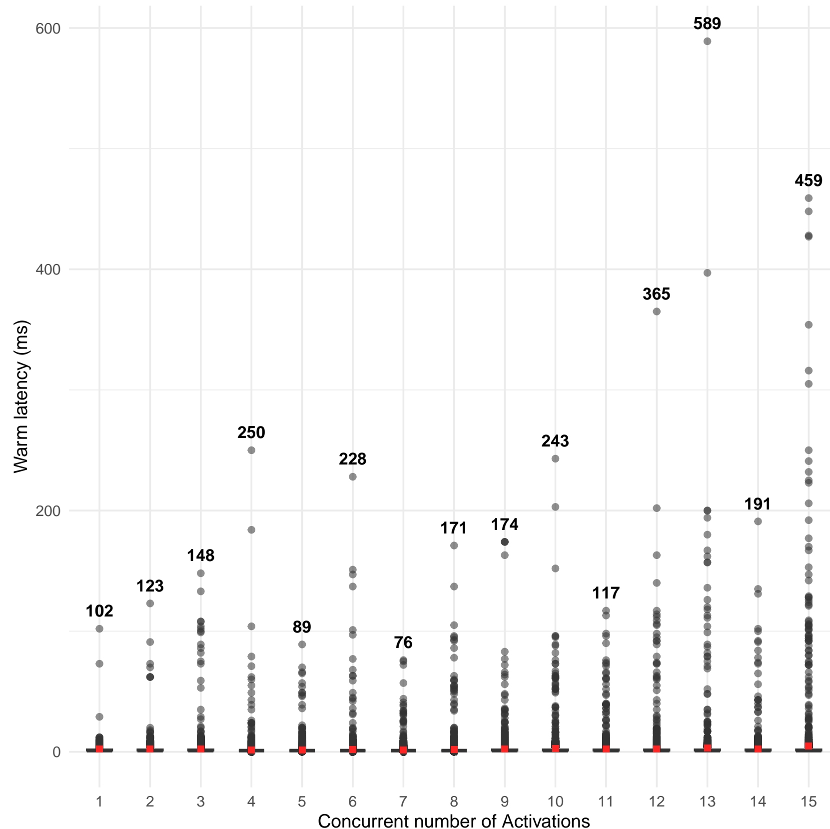

Figure 13: Concurrency test with mixed load 10/90 for Docker container

No. concurrent responses	Coefficient of Variation in %	Standard Deviation	Mean in ms
1	212.0841	4.836140	2.280294
2	245.4397	5.430822	2.212691
3	337.0182	8.010329	2.376824
4	421.3227	6.762235	1.605002
5	246.0607	3.698266	1.502989
6	404.4446	6.915386	1.709847
7	214.2260	3.368749	1.572521
8	342.8710	6.662441	1.943133
9	317.4047	7.107820	2.239356
10	337.4955	9.288928	2.752312
11	255.6141	6.265316	2.451084
12	434.5748	10.955845	2.521049
13	564.4459	17.009485	3.013484
14	311.2113	7.003935	2.250540
15	512.0065	25.420462	4.964871

Table 7: Statistics for the Concurrency test with mixed load 10/90 for Docker container

With a maximum latency of 83ms and a maximum mean 1.67ms Wasm displays the same behavior as Docker regarding these two traits. However, the standard deviation shows that the Wasm latencies are denser. The following figures show that, as IO-load increases and CPU-load decreases, the maximum latencies as well as the mean latencies decrease for Wasm and for Docker.

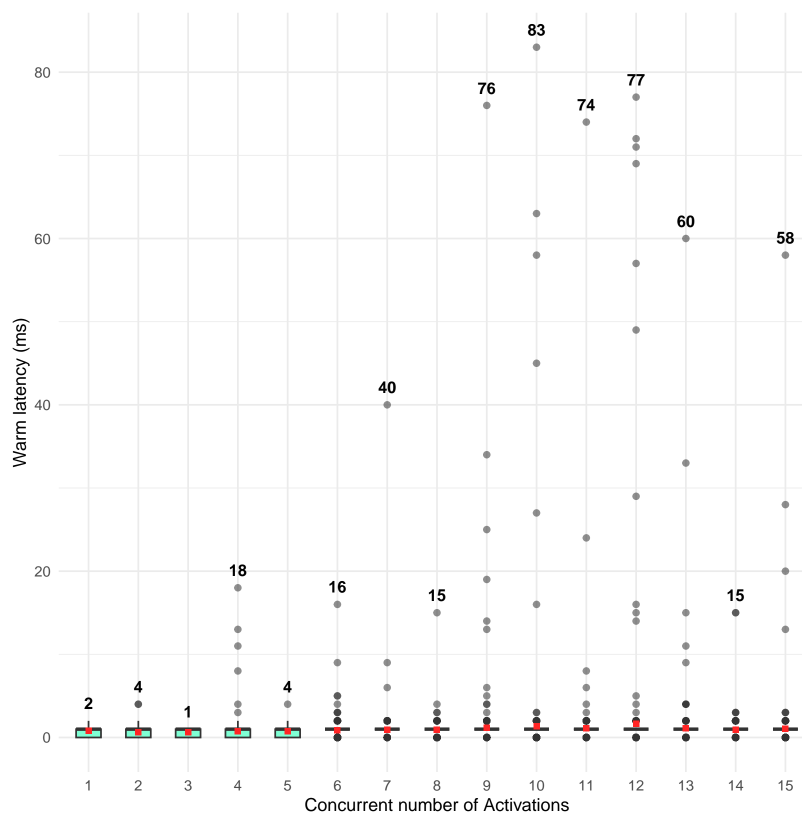

Figure 14: Concurrency test with mixed load 10/90 for Wasm module

No. concurrent responses	Coefficient of Variation in %	Standard Deviation	Mean in ms
1	76.89942	0.6168854	0.8021978
2	92.10132	0.5750241	0.6243386
3	72.93823	0.4764173	0.6531792
4	148.49717	1.1235630	0.7566225
5	67.49192	0.5040123	0.7467742
6	99.66890	0.8721028	0.8750000
7	179.90813	1.6609074	0.9231975
8	79.08173	0.7268353	0.9190939
9	302.62264	3.5816730	1.1835443
10	389.31945	5.2376528	1.3453355
11	284.36538	3.1748614	1.1164725
12	396.57010	6.6073695	1.6661290
13	269.45577	2.9579001	1.0977312
14	102.08584	0.9511753	0.9317406
15	268.57557	2.8616370	1.0654867

Table 8: Statistics for the Concurrency test with mixed load 10/90 for Wasm module

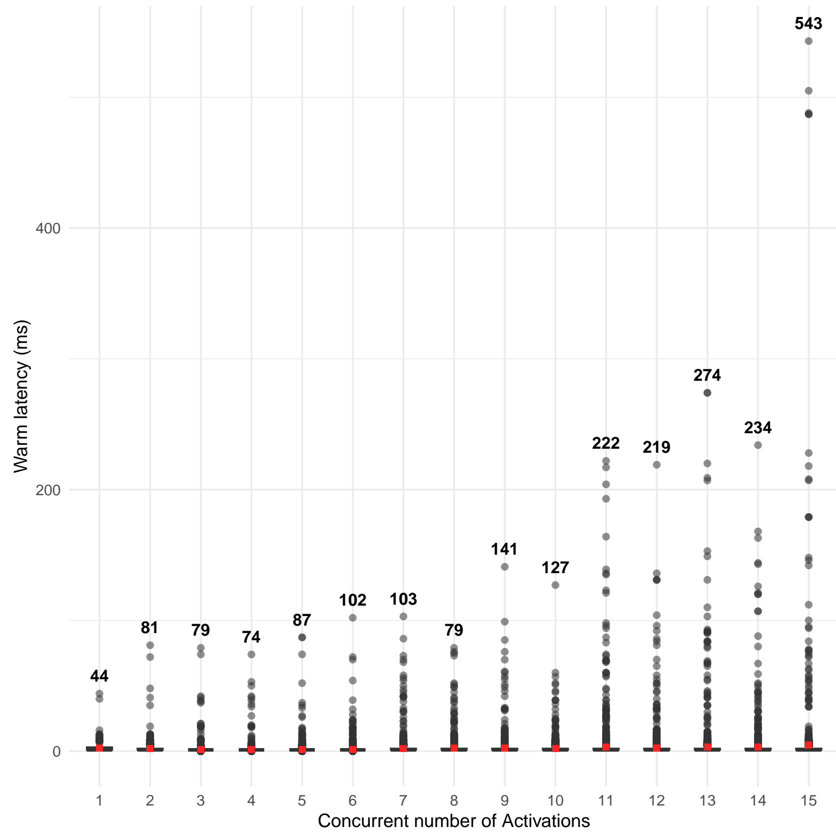

Figure 15: Concurrency test with mixed load 25/75 for Docker container

No. concurrent responses	Coefficient of Variation in %	Standard Deviation	Mean in ms
1	140.1339	3.452660	2.463830
2	230.7502	4.281440	1.855444
3	302.7471	3.712798	1.226370
4	286.1608	3.223677	1.126527
5	273.7594	3.846351	1.405011
6	260.3761	3.608860	1.386018
7	253.5950	5.287277	2.084929
8	216.4388	4.746323	2.192917
9	250.8908	6.008293	2.394784
10	203.5183	4.155698	2.041929
11	377.8533	11.310440	2.993342
12	326.3104	8.375841	2.566833
13	411.4154	13.353520	3.245751
14	347.8126	10.305942	2.963073
15	581.1660	26.497916	4.559441

Figure 16: Statistics for the Concurrency test with mixed load 25/75 for Docker container

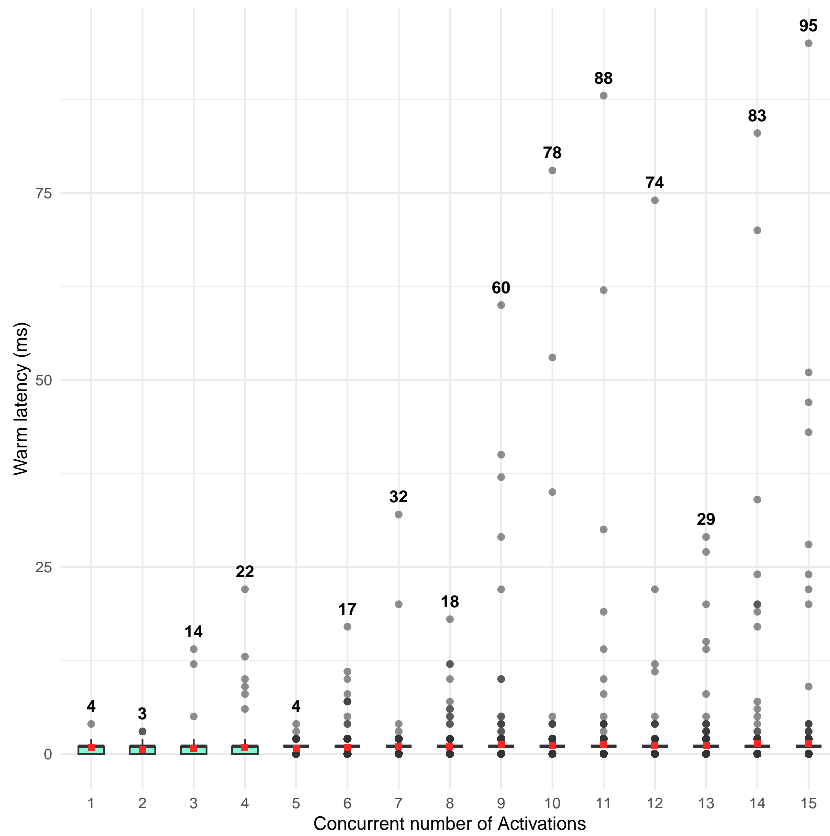

Figure 17: Concurrency test with mixed load 25/75 for Wasm module

No. concurrent responses	Coefficient of Variation in %	Standard Deviation	Mean in ms
1	82.16623	0.6909433	0.8409091
2	88.64437	0.5712110	0.6443850
3	133.59363	0.9511433	0.7119675
4	154.02402	1.2664766	0.8222591
5	61.25280	0.4791827	0.7823034
6	113.18491	1.0308719	0.9107856
7	148.43486	1.3926513	0.9382239
8	105.04751	1.0636060	1.0125000
9	259.47276	3.1181413	1.2017220
10	316.11297	3.6127197	1.1428571
11	328.14215	4.0806337	1.2435567
12	266.03089	2.8652198	1.0770252
13	166.40779	1.8508139	1.1122159
14	336.19894	4.5235756	1.3455056
15	344.89601	4.8415416	1.4037685

Figure 18: Statistics for the Concurrency test with mixed load 25/75 for Wasm module

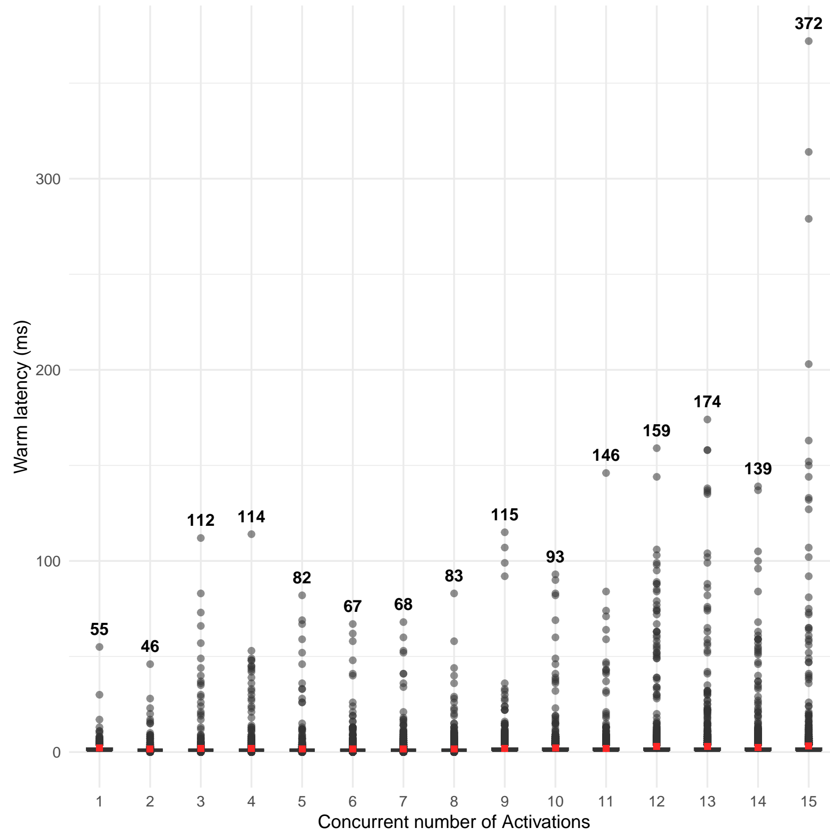

Figure 19: Concurrency test with mixed load 50/50 for Docker container

No. concurrent responses	Coefficient of Variation in %	Standard Deviation	Mean in ms
1	180.3958	3.946523	2.187702
2	181.3313	2.907808	1.603589
3	339.1992	6.760024	1.992936
4	307.6704	5.650164	1.836434
5	287.2045	4.453346	1.550584
6	233.1869	3.544704	1.520113
7	221.4474	3.642453	1.644839
8	203.3330	3.318982	1.632289
9	265.9789	5.026335	1.889749
10	247.8396	5.195400	2.096275
11	269.1971	5.221776	1.939759
12	334.9931	9.476812	2.828958
13	352.5422	10.130444	2.873541
14	286.0254	7.087849	2.478049
15	472.2494	15.290640	3.237831

Figure 20: Statistics for the Concurrency test with mixed load 50/50 for Docker container

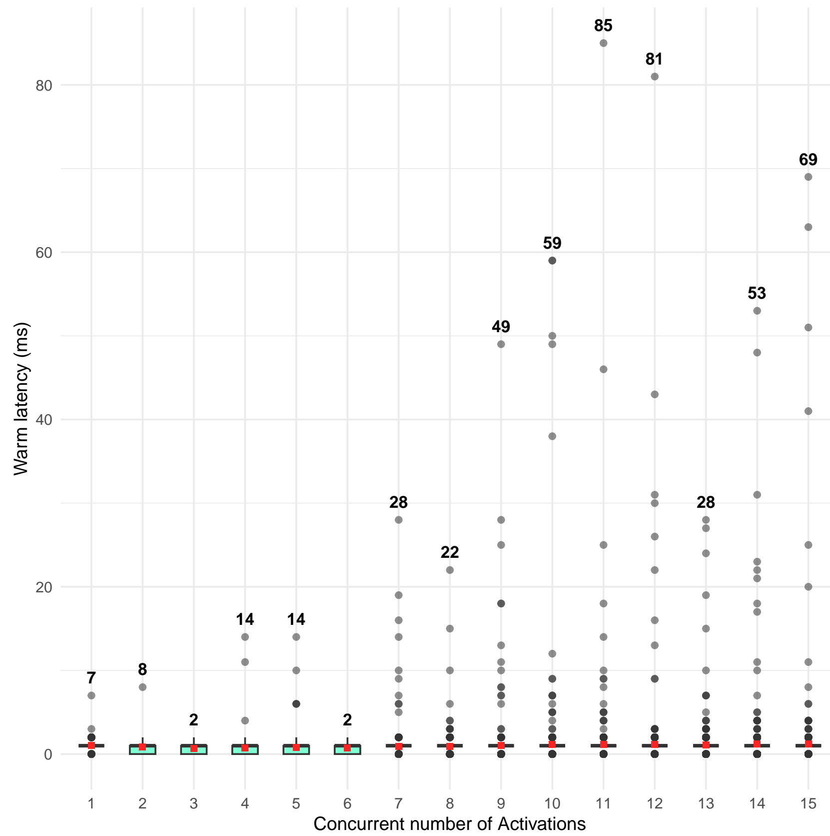

Figure 21: Concurrency test with mixed load 50/50 for Wasm module

No. concurrent responses	Coefficient of Variation in %	Standard Deviation	Mean in ms
1	72.93096	0.7709845	1.0571429
2	79.87409	0.6910455	0.8651685
3	71.85391	0.5038361	0.7011952
4	116.10657	0.8374608	0.7212864
5	104.57660	0.8089386	0.7735369
6	63.49168	0.4610451	0.7261504
7	156.70717	1.4027047	0.8951120
8	109.40149	0.9800115	0.8957935
9	207.34681	2.0806865	1.0034813
10	293.93375	3.3339932	1.1342669
11	268.57253	3.0639965	1.1408451
12	271.50582	3.0887104	1.1376221
13	148.63860	1.5785748	1.0620221
14	229.79070	2.7147656	1.1814079
15	291.99155	3.5090666	1.2017699

Figure 22: Statistics for the Concurrency test with mixed load 50/50 for Wasm module

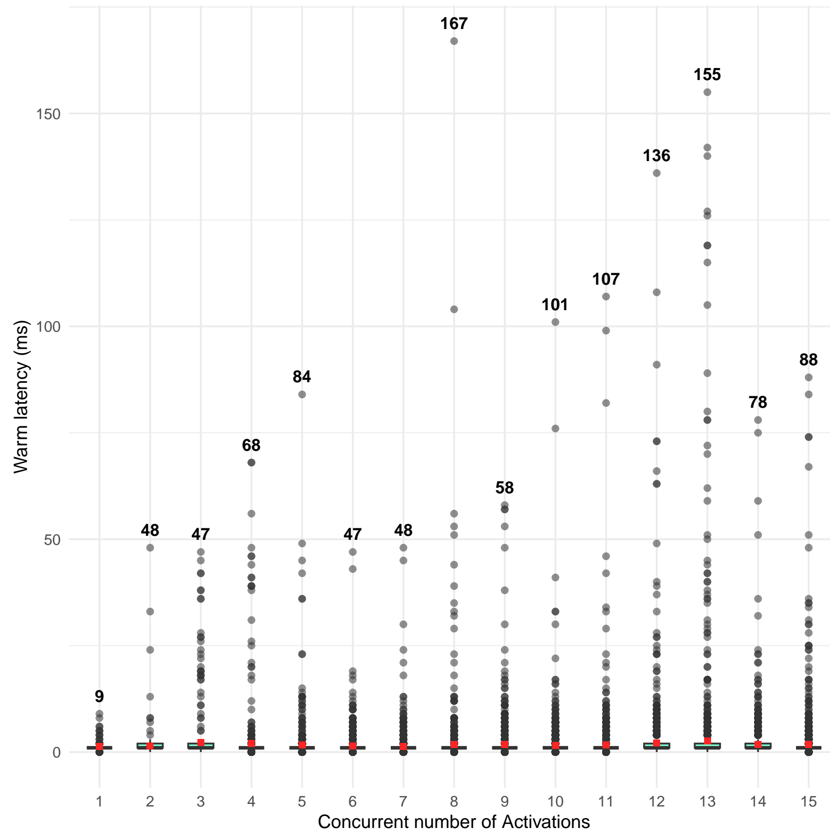

Figure 23: Concurrency test with mixed load 75/25 for Docker container

No. concurrent responses	Coefficient of Variation in %	Standard Deviation	Mean in ms
1	83.76495	1.125592	1.343750
2	204.93580	3.019869	1.473568
3	244.76570	5.418759	2.213855
4	298.38657	6.125137	2.052752
5	256.57174	4.360146	1.699387
6	177.40891	2.494032	1.405810
7	181.01360	2.481250	1.370753
8	346.80601	6.315520	1.821053
9	221.26176	3.835593	1.733509
10	237.86756	3.755025	1.578620
11	260.07814	4.498938	1.729841
12	283.41704	6.141850	2.167071
13	375.90175	9.830764	2.615248
14	195.75533	3.544136	1.810493
15	245.69100	4.500590	1.831809

Figure 24: Statistics for the Concurrency test with mixed load 75/25 for Docker container

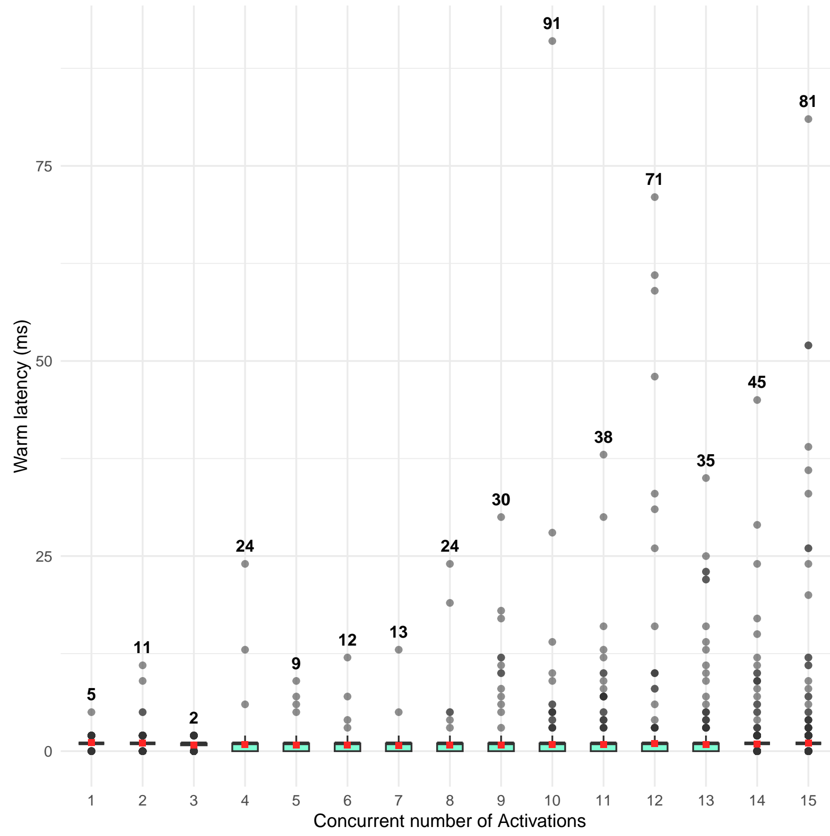

Figure 25: Concurrency test with mixed load 75/25 for Wasm module

No. concurrent responses	Coefficient of Variation in %	Standard Deviation	Mean in ms
1	57.34431	0.6160417	1.0742857
2	90.39264	0.9250709	1.0233918
3	64.88197	0.5271660	0.8125000
4	135.47802	1.1705956	0.8640483
5	87.43659	0.6730807	0.7697929
6	87.40576	0.6656492	0.7615622
7	85.66642	0.6264592	0.7312775
8	128.83087	0.9702078	0.7530864
9	166.85895	1.3107816	0.7855626
10	300.32478	2.5223355	0.8398693
11	171.04187	1.4701858	0.8595473
12	327.41570	3.1299210	0.9559471
13	187.22497	1.6243661	0.8676012
14	173.38020	1.5733451	0.9074537
15	289.44645	3.0064360	1.0386847

Figure 26: Statistics for the Concurrency test with mixed load 75/25 for Wasm module

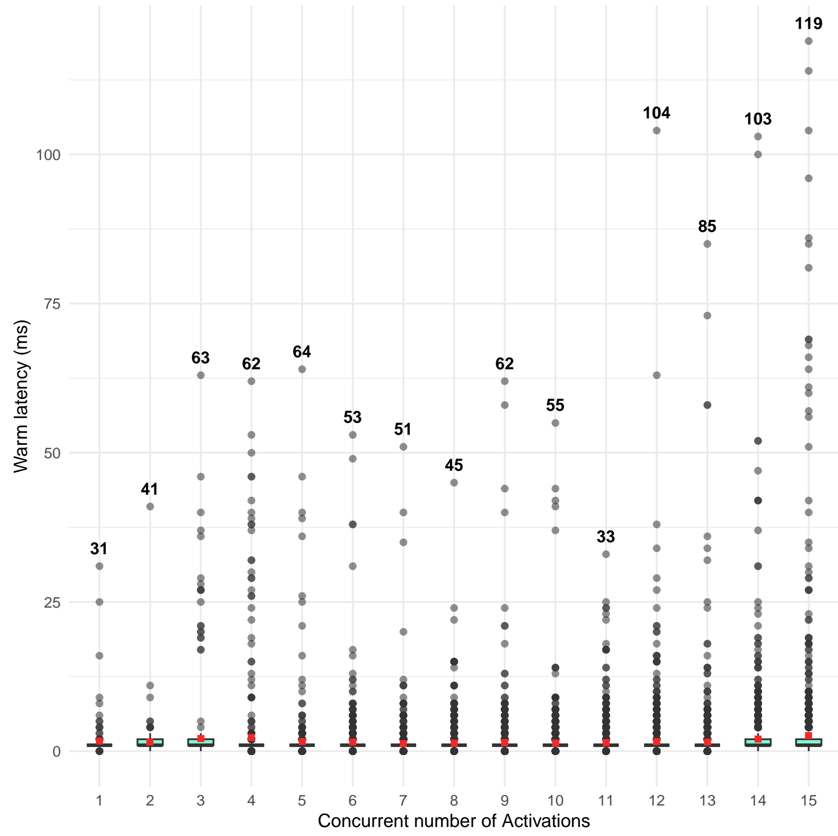

Figure 27: Concurrency test with mixed load 90/10 for Docker container

No. concurrent responses	Coefficient of Variation in %	Standard Deviation	Mean in ms
1	182.8886	3.156303	1.725806
2	154.2888	2.303588	1.493036
3	255.7437	5.403617	2.112903
4	278.5925	6.285046	2.256000
5	241.3429	3.791747	1.571104
6	215.0893	3.347087	1.556139
7	188.4199	2.441474	1.295763
8	154.7753	2.140149	1.382745
9	215.5783	3.103200	1.439477
10	192.0730	2.666411	1.388228
11	146.8115	2.074284	1.412889
12	228.1417	3.665485	1.606670
13	238.0157	3.684941	1.548193
14	232.9671	4.660480	2.000488
15	300.7822	7.776575	2.585451

Figure 28: Statistics for the Concurrency test with mixed load 90/10 for Docker container

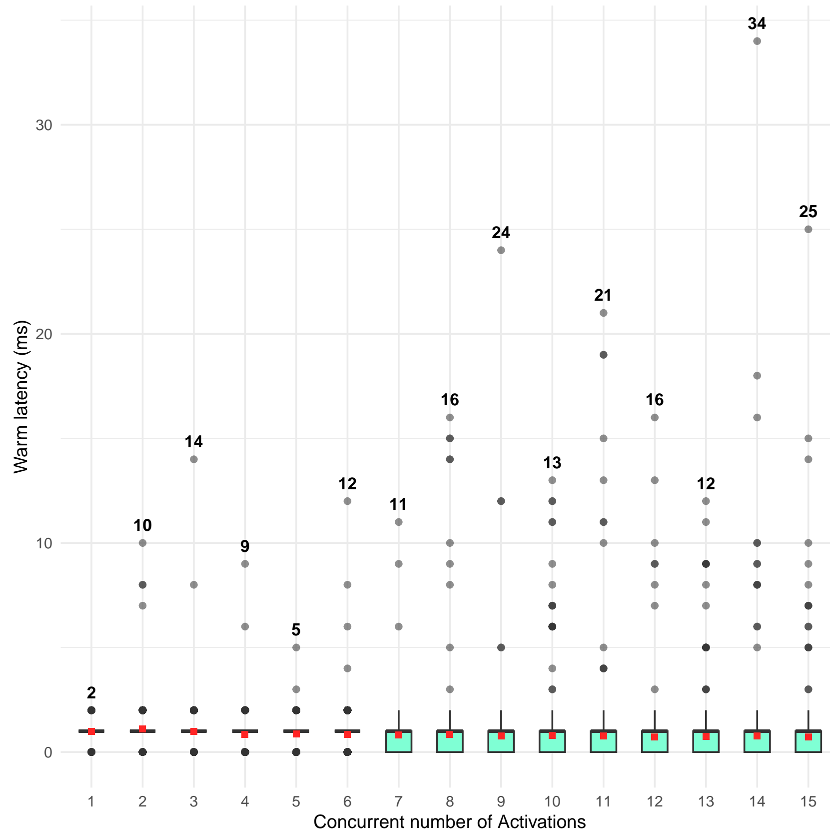

Figure 29: Concurrency test with mixed load 90/10 for Wasm module

No. concurrent responses	Coefficient of Variation in %	Standard Deviation	Mean in ms
1	55.02741	0.5438003	0.9882353
2	87.47505	0.9722377	1.1114458
3	86.66844	0.8474247	0.9777778
4	77.11526	0.6545277	0.8487654
5	60.90973	0.5290699	0.8686131
6	83.59541	0.6987459	0.8358663
7	80.43115	0.6629094	0.8241950
8	128.53614	1.0942151	0.8512898
9	117.95994	0.9065293	0.7685061
10	118.27473	0.9336351	0.7893784
11	143.55072	1.1129923	0.7753304
12	107.55283	0.7807660	0.7259372
13	104.89804	0.7716864	0.7356538
14	143.85471	1.0944240	0.7607843
15	126.30338	0.9103551	0.7207686

Figure 30: Statistics for the Concurrency test with mixed load 90/10 for Wasm module

5.3 Comparison of performance

The prior sections give detailed information for each combination of test and platform. This makes cross-comparisons between both types of runtimes challenging. For this reason, this section aims to ease cross-comparison by summarizing the obtained information and introducing

the slowdown ratio, defined as how many times the latency with Docker is higher than for Wasm. For each test scenario for all concurrent requests, the mean of means as well as the mean coefficient of variation values are computed.

Again, it is visible that the difference in latencies between Docker and Wasm is largest in the case of cold starts. Giving a relation between standard deviation and mean, the coefficient of variation (CoV) here is significantly higher for Wasm, as Wasm has lower mean values.

	Mean	CoV in %
Docker	16158.29	36.58
Wasm	119.41	62.45
Slowdown ratio	135.31	

Table 9: Comparison cold-start tests

For concurrency without load Docker has a higher mean and CoV.

	Mean in ms	CoV in %
Docker	4.57	85.25
Wasm	3.98	64.90
Slowdown ratio	1.14	

Table 10: Comparison concurrency no load

After adding load, the mean values for Docker rise, while the same values for Wasm decline. As most mean values for Wasm are below 0, a mean Cov of 194.25 is no surprise.

	Mean in ms	CoV in %
Docker	7.39	112.40
Wasm	0.87	194.25
Slowdown ratio	8.49	

Table 11: Comparison concurrency with hashing

For the concurrency tests with mixed loads, the slowdown ratio is decreasing slightly as IO-load increases and CPU-load decreases. For both Docker and Wasm, the mean values fall, but more gradually for Wasm, as here the mean of means is not significantly more than 1ms. In either scenario, the mean CoV falls as well.

	Mean in ms	CoV in %
Docker	2.33	442.78
Wasm	1.01	291.30
Slowdown ratio	2.30	

Table 12: Comparison concurrency with mixed load 10/90

	Mean in ms	CoV in %
Docker	2.38	426.97
Wasm	1.05	263.89
Slowdown ratio	2.27	

Table 13: Comparison concurrency with mixed load 25/75

	Mean in ms	CoV in %
Docker	2.19	348.28
Wasm	1.00	228.38
Slowdown ratio	2.38	

Table 14: Comparison concurrency with mixed load 50/50

	Mean in ms	CoV in %
Docker	1.86	287.25
Wasm	0.87	219.32
Slowdown ratio	2.13	

Table 15: Comparison concurrency with mixed load 75/25

	Mean in ms	CoV in %
Docker	1.70	248.83
Wasm	0.79	117.30
Slowdown ratio	2.15	

Table 16: Comparison concurrency with mixed load 90/10

The evaluation of the obtained results shows that Wasm has a lower latency for all cases, except in the case of test execution without load for a lower number of concurrent requests. With a simple action, the most invocations were executed. Furthermore, Wasm records a lower CoV for all warm latencies. That shows, Docker outperforms Wasm only if a multitude of warm actions are invoked with a low amount of parallelity. In all other scenarios, Wasm proves to be the superior solution. The heavy outliers while invoking the functions could be reduced by using an rt-patch for the kernel and configuring the operating system. As none of those steps were taken its not possible to conclude whether the outliers originate from OpenWhisk or if a high priority task from the operating system interrupted the execution.

5.4 Capacity tests

The capacity tests reveal the maximum amount of concurrent running instances for both Docker containers and Wasm modules. With this information, we can calculate the density, which is roughly 10 times higher for Wasm. The use of Docker causes about four times more load on memory and the CPU.

	Docker	Wasm	vs. Docker
max. CPU usage	773.37	182	**4.25x**
RSS memory	4947.56	1030.58	**4.8x**
Capacity	81	878	**10.84x**

.

Table 17: Capacity test results

In both cases, the error causing the abortion was:

```
Curl("Failure when receiving data from the peer")
```

Multiple test executions for Wasm all returned similar results, the capacity was never higher than 900 modules. Which raises the question of why the error occurs when neither the CPU nor memory are near the maximum of their respective capacities. An in-depth analysis of the OpenWhisk system and all components regarding this error would exceed the limits of this thesis, however it provides an opportunity to continue the research. Wasm shows a linear growth in both CPU usage and memory required, and for Docker, a drop is recorded during the test execution. This test was executed twice, with similar results.

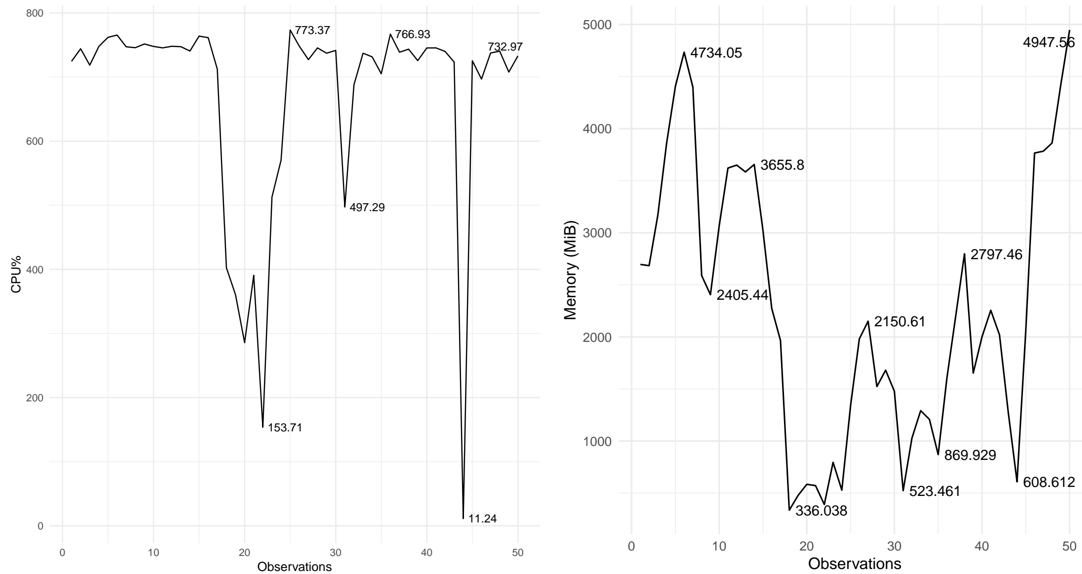

Figure 31: Evolution of CPU (left) and memory (right) usage during the capacity test for Docker

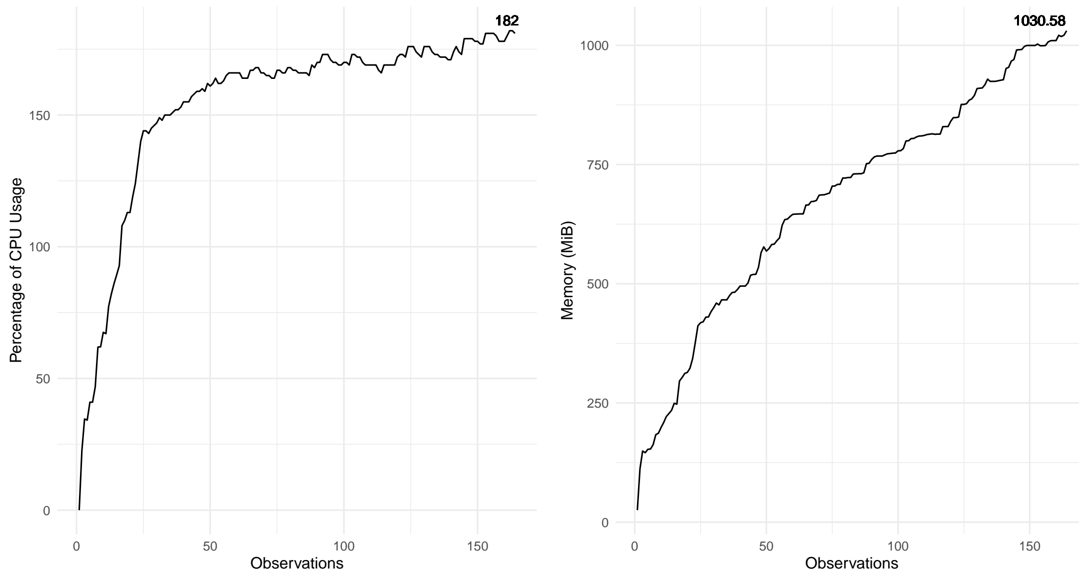

Figure 32: Evolution of CPU (left) and memory (right) usage during the capacity test for Wasm

6 Identifying Potential Enhancements

After evaluating the results, it shows that Wasm has rather low warm-up latencies. However, the question arises if it's possible to improve on the existing code to decrease the latencies even further. For this, several durations in the Executor and in the runtime environment are measured.

We remember the size of each action:

Action	Size
hello.rs	555KB
hash.rs	459KB
prime.rs	559KB
net.rs	426KB
sleep.rs	455KB

6.1 Enhance cold-starts

In the initialization function of the Executor the Wasm runtime is initialized by decoding the received code from the request. Here two durations are of interest, first the time to initialize the Wasm runtime and the overhead related to preparing the code.

```
OpenWhisk/openwhisk/wow/ow-executor/src/core.rs : init()
    let activation_init: ActivationInit = activation_init?;
    println!("/init {}", activation_init.value.name);
    let container_id = req.param("container_id").unwrap().to_owned();
    println!("Initializing container with id {}", container_id);
    // Get the WasmRuntime instance from the request state
```

```
        let runtime = req.state();
        // Decode the code
        let module_bytes = util::b64_decode(activation_init.value.code)?;
        // Unzip the module bytes
        let module = util::unzip(module_bytes)?;

    OpenWhisk/openwhisk/wow/ow-executor/src/core.rs : init()
        runtime.initialize(container_id, activation_init.value.annotations, module)?;
```

In Table: 18 the measured times for both sequences are displayed in μs.

Task	Type	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10	**Mean**
`hash`	Overhead	526	368	557	591	326	374	503	486	403	401	**453.5**
	Init	910	886	1580	1606	886	861	840	841	830	849	**1008.9**
`hello`	Overhead	444	448	446	676	425	589	492	394	560	520	**499.4**
	Init	1100	1073	1084	1918	1084	1077	1107	1078	1118	1130	**1176.9**
`net`	Overhead	353	325	383	553	371	318	364	329	335	417	**374.8**
	Init	798	802	782	1198	822	844	846	818	1190	1104	**920.4**
`prime`	Overhead	680	709	449	619	869	448	444	418	419	431	**548.6**
	Init	1281	2044	1011	1099	1890	1056	1044	1069	1042	1080	**1261.6**
`sleep`	Overhead	429	340	401	328	415	426	369	350	395	442	**389.5**
	Init	933	875	923	865	880	920	884	868	898	884	**893.0**

Table 18: Duration comparison cold-starts in μs

It can be easily seen that the size of the source code is directly related to the required time for both types. With 559KB and 555KB are 'prime.rs' and 'hello.rs' the actions with the largest source file. For those two actions, the highest mean values were measured. Moreover, it can be noted that the time to initialize the runtime is significantly higher than the overall overhead to prepare the module in all cases. In addition Fig.: 33 eases the comparison of proportions by displaying the mean values in a stacked bar plot.

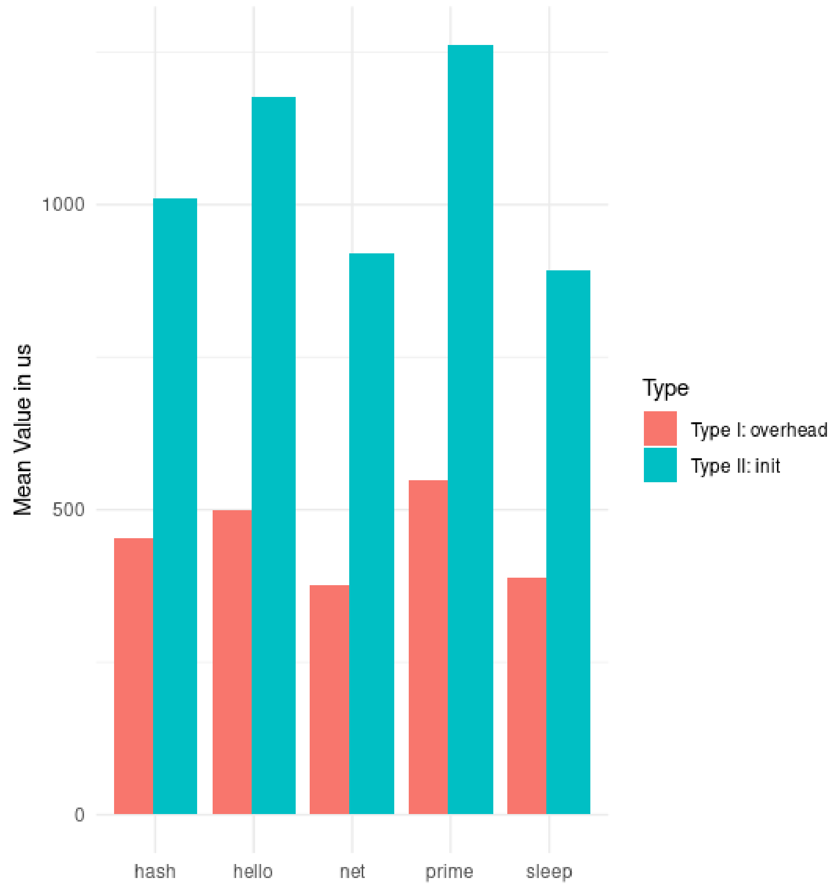

Figure 33: Mean cold-start duration comparison in μ

6.2 Enhance warm-starts

To enhance latencies occurring with warm starts, several durations in the run functions are measured. Improving those times would benefit every warm-start and, naturally, every cold-start. We measure the time for cloning the runtime:

```
OpenWhisk/openwhisk/wow/ow-executor/src/core.rs : run()
    //Cloning runtime
    let runtime = req.state().clone();
```

Furthermore, the duration needed for providing and linking the WASI and instantiating the module was determined:

```
OpenWhisk/openwhisk/wow/ow-wasmtime/src/wasmtime.rs : run()
    //Providing WASI
    let ctx = build_wasi_context(&wasm_action.capabilities, json_bytes.len())?;
    let wasi = Wasi::new(&store, ctx);
    wasi.add_to_linker(&mut linker)?;

    //Instantiating the module
    let module = &wasm_action.module;
    let instance = linker.instantiate(module)?;
```

```
let main = linker.instance("", &instance)?.get_default("")?;
pass_string_arg(&instance, json_bytes)?;
```

The following table displays the duration required for providing WASI as 'Type I' and the duration required for instating the module as 'Type II'. For Cloning the runtime in all cases 0μ were recorded, thus its not included in the table.

Action	Type	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10	**Mean**
`hash`	I	109	107	108	111	112	260	114	109	118	114	**126.2**
	II	128	123	127	145	225	164	126	133	133	107	**141.1**
`hello`	I	175	187	105	436	177	121	244	107	149	119	**172**
	II	239	241	150	244	243	144	245	156	151	165	**177.8**
`net`	I	113	275	174	214	127	116	115	121	111	125	**149.1**
	II	113	173	179	179	141	118	143	159	118	171	**149.4**
`prime`	I	112	128	148	110	153	154	124	126	108	183	**134.6**
	II	130	157	147	167	154	174	167	154	171	211	**163.2**
`sleep`	I	106	115	152	150	196	126	119	175	199	124	**146.2**
	II	129	144	185	177	202	199	177	189	202	151	**175.5**

Table 19: Duration comparison warm-starts in μ

With all five actions, used for the benchmark, the time required for cloning the runtime is 0μs. For adding WASI times between 106μs and 436μs where measured. Hereby 436μs represents a heavy outlier. For instantiating the module the minimum required time is 107μs and the maximum 245μs. In most cases the required amount of time for instantiating the module is 10-15μs longer. In the worst case not more than 380μs are required for creating the module, which is concordant to presented average of 340μs in WoW[17]. A graphical representation of the mean values is given by the following figure Fig.: 34.

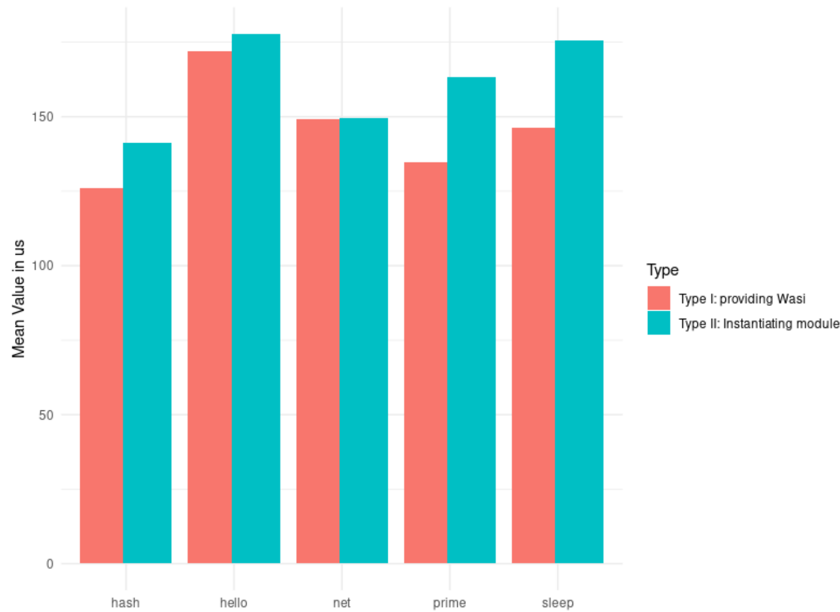

Figure 34: Mean warm-start duration comparison in μ

7 Conclusion

The primary objective of this work was to evaluate the substitution of Docker by Wasm in the OpenWhisk system. For this evaluation, we focused on aspects of cold-start and warm-start latencies for a growing amount of concurrent requests in different environments. Additionally, the maximum capacity of concurrent running containers and modules was of interest. To answer whether the substitution improves the performance, various tests were conducted to measure the latencies in several environments as well as gain knowledge about the highest count of concurrency before system failure.

Only in one specific scenario did Docker show a slightly higher mean latency than Wasm, while simultaneously showing higher maximum latencies. The lower measured CoV for Wasm in all but one scenario demonstrates that Wasm is better suited for tasks that require more consistent latencies. Thus, current containerization solution can be improved by replacing Docker containers with Wasm modules.

Additionally, the sending of the module, starting from the Controller, until the invocation of the action effects the performance as larger binaries require more time to be sent and to be decompressed. From this, further opportunities for enhancements arise. However, implementing an improved solution including a new set of benchmarks would be beyond the scope of this work and thus gives further opportunities to continue this research.

In the future, these research results could be expanded by evaluating the performance of different Wasm runtimes. In addition, the combination of OpenWhisk with lithops[11] may have the capability to create promising opportunities, giving a prospective way of exploration.

References

[1] Apache couchdb. https://couchdb.apache.org/, 2023. Accessed: 2023-05-06.

[2] Apache kafka. https://kafka.apache.org/, 2023. Accessed: 2023-05-06.

[3] Apache openwhisk. https://openwhisk.apache.org/, 2023. Accessed: 2023-05-06.

[4] Bytecode alliance: Webassembly system interface. https://wasi.dev/, 2023. Accessed: 2023-27-06.

[5] Docker. https://www.docker.com/, 2023. Accessed: 2023-05-06.

[6] Docker stats command. https://docs.docker.com/engine/reference/commandline/stats/, 2023. Accessed: 2023-21-08.

[7] Github: Wasi. https://github.com/WebAssembly/WASI, 2023. Accessed: 2023-23-08.

[8] The go programming language. https://go.dev/, 2023. Accessed: 2023-16-08.

[9] Kubernetes. https://kubernetes.io/, 2023. Accessed: 2023-06-06.

[10] Linux control groups: cgroups. https://man7.org/linux/man-pages/man7/cgroups.7.html, 2023. Accessed: 2023-29-08.

[11] Lithops. https://lithops-cloud.github.io/, 2023. Accessed: 2023-23-08.

[12] Nginx, inc. nginx. https://www.nginx.com/, 2023. Accessed: 2023-05-06.

[13] Standard unix command: ps. https://man7.org/linux/man-pages/man1/ps.1.html, 2023. Accessed: 2023-02-08.

[14] Text processing utility: awk. https://man7.org/linux/man-pages/man1/awk.1p.html, 2023. Accessed: 2023-02-08.

[15] Webassembly. https://webassembly.org/, 2023. Accessed: 2023-27-06.

[16] Webassembly memory layout. https://bytecodealliance.github.io/wamr.dev/blog/understand-the-wamr-heaps/, 2023. Accessed: 2023-27-06.

[17] P. Gackstatter, P. A. Frangoudis, and S. 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.

[18] P. K. Gadepalli, G. Peach, L. Cherkasova, R. Aitken, and G. Parmer. Challenges and opportunities for efficient serverless computing at the edge. In *2019 IEEE 38TH INTERNATIONAL SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS (SRDS 2019)*, Symposium on Reliable Distributed Systems Proceedings, pages 261–266. Red Hat; Inst Natl Sci Appliquees Lyon; Univ Lumiere Lyon 2; ENS Lyon; IEEE; IEEE Comp Soc; Inria; Liris; Inst Genre, Groupement Interet Sci; Citi Lab; CNRS; LIa; Univ Lyon; IDEX Lyon; Univ Lyon, Labex Milyon, 2019. IEEE 38th International Symposium on Reliable Distributed Systems (SRDS), Lyon, FRANCE, OCT 01-04, 2019.

[19] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with webassembly. *SIGPLAN Not.*, 52(6):185–200, jun 2017.

[20] A. Hall and U. Ramachandran. An execution model for serverless functions at the edge. In G. Ramachandran and J. Ortiz, editors, *PROCEEDINGS OF THE 2019 INTERNATIONAL CONFERENCE ON INTERNET OF THINGS DESIGN AND IMPLEMENTATION (IOTDI '19)*, pages 225–236. Assoc Comp Machinery; IEEE; ARM; IBM; IEEE Comp Soc; ACM SIGBED, 2019. ACM/IEEE International Conference on Internet of Things Design and Implementation (IoTDI), Montreal, CANADA, APR 15-18, 2019.

[21] D. Lehmann, J. Kinder, and M. Pradel. Everything old is new again: Binary security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 217–234. USENIX Association, Aug. 2020.

[22] J. Manner, M. Endress, T. Heckel, and G. Wirtz. Cold start influencing factors in function as a service. In A. Sill and J. Spillner, editors, *2018 IEEE/ACM INTERNATIONAL CONFERENCE ON UTILITY AND CLOUD COMPUTING COMPANION (UCC COMPANION)*, International Conference on Utility and Cloud Computing, pages 181–188. IEEE; ACM; IEEE Comp Soc, 2018. 11th IEEE/ACM International Conference on Utility and Cloud Computing (UCC-Companion) / 5th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (BDCAT), Zurich, SWITZERLAND, DEC 17-20, 2018.

[23] S. Shillaker and P. Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433. USENIX Association, July 2020.

[24] S. Zhao, P. Xu, G. Chen, M. Zhang, Y. Zhang, and Z. Lin. Reusable enclaves for confidential serverless computing. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4015–4032, Anaheim, CA, Aug. 2023. USENIX Association.