

Evaluation of Language Runtimes in Open-Source Serverless Platforms

Karim Djemame¹ ^a, Daniel Datsev¹ and Vasilios Kelefouras² ^b

¹*School of Computing, University of Leeds, Leeds, UK*

²*School of Engineering, Computing and Mathematics, University of Plymouth, Plymouth, UK*
{K.Djemame}@leeds.ac.uk, V.Kelefouras@plymouth.ac.uk

Keywords: Serverless Architecture, Openwhisk, Fission, Cloud Computing, Containerisation, Performance Evaluation

Abstract: Serverless computing is revolutionising cloud application development as it offers the ability to create modular, highly-scalable, fault-tolerant applications, with minimal operational management. In order to contribute to its widespread adoption of serverless platforms, the design and performance of language runtimes that are available in Function-as-a-Service (FaaS) serverless platforms is key. This paper aims to investigate the performance impact of language runtimes in open-source serverless platforms, deployable on local clusters. A suite of experiments is developed and deployed on two selected platforms: OpenWhisk and Fission. The results show a clear distinction between compiled and dynamic languages in cold starts but a pretty close overall performance in warm starts. Comparisons with similar evaluations for commercial platforms reveal that warm start performance is competitive for certain languages, while cold starts are lagging behind by a wide margin. Overall, the evaluation yielded usable results in regards to preferable choice of language runtime for each platform.

1 INTRODUCTION


Cloud computing has emerged as one of the most successful technologies in bringing processing power to the general public. The computer utilities vision has shaped most recent developments in the field and brought forth a variety of paradigms for doing distributed computations in the cloud. Serverless computing (Kritikos and Skrzypek, 2018) offers the illusion of infinite resources that are dynamically provisioned by cloud providers, allowing users to invest less effort and capital on infrastructure management. This type of *elastic* provisioning becomes automatic, eliminating the need for resource planning and predictive analysis of resource demand, giving the ability to run scalable, fault-tolerant *functions* in response to triggers.


The serverless architecture has seen widespread adoption from tech industry giants such as *Amazon* (Amazon Web Services, 2015), *Google* (Google, 2021) and *Microsoft* (Azure, 2021), as well as the public domain, with open-source projects such as *Apache OpenWhisk* (OpenWhisk, 2021), *Fission* (Fission, 2021b) and *OpenFaaS* (OpenFaaS, 2021).

A serverless computing system is an ideal solution

to build and optimise any Internet of Things (IoT) operation with zero infrastructure and maintenance costs and little-to-no operating expense (Großmann et al., 2019) as it allows IoT businesses to offload all of a server’s typical operational backend responsibilities. Moreover, such a system is a natural fit for edge computing applications as serverless computing also supports the protocols which IoT devices require in actual deployment conditions (Mistry et al., 2020).

Although a serverless architecture offers scalability, fault tolerance and cost benefits, it also comes with a set of drawbacks related to the execution environment that affects the viability and design of applications (Baldini et al., 2017). Investigations into various aspects of the serverless architecture are therefore required to guide the decision making process of users, and highlight problem areas for future research. One of the most detrimental factors affecting performance in serverless architectures is the notion of *cold start* that happens when the first incoming request to an application leads to a time-consuming allocation of resources which delays the response and leads to bad user experience (Mohan et al., 2019). Subsequently, the choice of language runtime plays a non-trivial role in the performance of serverless applications. In particular, the cold start times differ significantly across different languages and platforms

^a  <https://orcid.org/0001-5811-5263>

^b  <https://orcid.org/0002-3340-3792>

(Jackson and Clynch, 2018).

Investigations focus mainly on commercial platforms, while research in the open-source domain is lacking (see section 2). Entities may wish to leverage their existing infrastructure to develop services based on the serverless paradigm, while also avoiding vendor lock-in, inherent in proprietary ecosystems (Baldini et al., 2017). Their options are many but comparisons between them are few, which can lead to a trial-and-error approach and an associated increase in development costs. Combined, these two factors make a great case for a performance evaluation and comparison of language runtimes in open-source serverless platforms. Even if the request overhead introduced by a particular language is minimal, it is a constant factor on each invocation and has significant cumulative impact in terms of cost, and noticeable effect on user experience for low-latency real-time applications.

The aim of this paper is to evaluate the performance impact the choice of language runtime has on function execution in local deployments of two open-source serverless frameworks, Apache OpenWhisk (OpenWhisk, 2021) and Fission (Fission, 2021b), by measuring runtime overhead through the use of empty functions. Both frameworks are chosen for their support to code serverless functions in any language, and have them run on a Kubernetes cluster (Fission) and non-Kubernetes (Openwhisk). The vision is to provide insight into the viability of each supported language in various use cases, and offer comparisons to established industry platforms by utilising published results from similar research investigations. The paper makes the following contributions:

- it proposes a cloud-based technical solution for benchmarking and analysis of two open source serverless platform using a set of test functions;
- it evaluates the language runtimes of these open source serverless platforms, demonstrating their performance in terms of effectiveness and efficiency;
- it makes recommendations on the suitability of the language runtimes, taking into consideration commercial offerings.

The paper is structured as follows: section 2 reviews the related work and looks into the technological and research landscape surrounding serverless computing and runtime evaluation. Research questions are set in section 3 as well as an outline of the experimental design that will address them. The experimental environment setup and the test functions for various serverless use cases are described in section 4. In section 5 the experiment results are presented with a discussion on their significance and how they

compare to existing research. It also reflects on the research outcomes and any limitations encountered. Section 6 concludes with a summary of the research findings and suggestions for future work.

2 RELATED WORK

There has been extensive research around factors affecting function execution performance (Scheuner and Leitner, 2020) as well as some evaluations of open-source serverless frameworks, including the ones investigated in this work (Djemame et al., 2020). In regards to language runtime evaluation, Jackson and Clynch (Jackson and Clynch, 2018) do a performance and cost analysis of language choice on AWS Lambda and Azure Functions. Findings are unexpected, with Python and .NET C# performing better than the other language runtimes on AWS, and contrary to conclusions by Manner (Manner et al., 2018) a just-in-time dynamic language outperforms the compiled alternatives. Additionally, C# has the worst cold start times on AWS which makes it much less lucrative due to high cost and worse user experience. Microsoft’s Azure platform has a much better runtime for C# (Jackson and Clynch, 2018), performing close to six times faster than functions on AWS, and with much better cold start latency, showcasing the importance of a well-optimised runtime.

Vojta (Vojta, 2016) documents his findings on factors influencing function performance and does a comparison of three interpreted languages on AWS Lambda, noting minimal difference in warm start response times. The research however is not systematic and doesn’t compare other factors such as cold starts or compiled languages. Wang et al. (Wang et al., 2018) perform a comprehensive study of resource provisioning and how it affects performance on three commercial platforms (AWS Lambda, Azure, Google Functions). Among the investigations cold/warm start latency is considered for different language runtimes with results that are in line with (Jackson and Clynch, 2018). Virtual Machine (VM) instance size and memory are identified as factors affecting the severity of the cold start problem. Cui (Sbarski et al., 2022) also ran experiments on AWS Lambda, comparing dynamic and compiled languages with findings similar to (Jackson and Clynch, 2018) and (Wang et al., 2018), as expected due to them being performed on the same platform. The cold start times of statically typed languages appear to be almost 100 times slower than dynamic, although the method of measuring cold starts appears dubious, and might be biased since the cold starts are forced by redeploying the function in-

stance, which might not simulate the usual conditions for cold start invocation, adding potential overhead. The increase in memory size of the function instance correlating linearly with a decrease in cold start times was observed. A few notes on the general literature landscape and how this research fits in:

- Platform choice is predominantly AWS and Azure, and little research on the topic was found for open-source serverless platforms. This research aims to amend that, and provide insights into runtime performance of on-premise serverless deployments.
- Measurements are obtained either through platform metrics, or client-side using a stress-testing tool. This research aims to compare the two types and explore how this difference can affect serverless applications.
- Language choice is often limited with a few main options being investigated. This leads to a skewed view of the available landscape, as less popular languages are being overlooked. This research aims to investigate all default runtimes offered by each platform, allowing for a broader insight into available options.

3 PROPOSED APPROACH AND RESEARCH QUESTIONS

This investigation is about performing an evaluation of open-source serverless frameworks that are to be deployed on private infrastructures, based on factual data that can be measured so a quantitative experimental methodology using direct experiments is selected. Furthermore, the research methodology ensures a unified cloud testing environment and has the ability to modify the investigated variable (language runtime), so an experimental design is feasible when it comes to accuracy of measurements.

In order to establish the relevance and usefulness of this research the following Research Questions (RQ) are formulated as an anchor for the discussions:

RQ1: *What impact does choice of language have on function execution time?* Consequently, this will reveal if there exists a *preferable* choice of runtime when considering a particular platform. Additionally, a direct comparison between the platforms can indicate if there is an overall better choice for local deployments.

RQ2: *What overhead does the API mechanism incur?* This is an extension to the previous question and aims to investigate if the built-in web request mechanism for each platform has a detrimental impact on

function execution. Furthermore, this will reveal if language choice has any effect on the overhead by comparing with platform results from RQ1.

RQ3: *Are results competitive with commercial platforms?* A comparison with published results can uncover trends and show if a locally-deployed open-source platform can compete with existing commercial alternatives.

For presenting the results, the *median* method is chosen, and in particular the boxplot representation to summarise the findings for each platform. This gives a compact way to present all platform language results, without sacrificing information. The *mean* is also used when comparing with published research as this is the method most often seen in literature. As for the automation and evaluation tooling, the deployment procedure for Openwhisk and Fission platforms does not require extensive configuration and can be done manually and easily verified. For metric collection and visualisation a single execution variable, function execution time, is tracked.

4 EXPERIMENT DESIGN

Empty Functions. In order to measure the impact of a language runtime on function execution, any additional execution overhead needs to be eliminated. Since the runtime overhead cannot be obtained directly, the function execution of *completely empty* functions is measured. Since no time is spent within the function itself, by measuring the execution time this implicitly provides the runtime overhead.

Languages The language runtimes to be tested have been chosen based on language popularity and availability on each platform. An effort has been made to test all available ones, but a few have been excluded, in particular *Custom runtimes* for OpenWhisk and Fission. Both platforms support the use of custom containers, allowing the use of any language, custom executables and scripts. This is however outside of the scope of this paper as the interest is in measuring the overhead of the optimised runtime containers offered by the default installations. Table 1 summarises the final candidates. There is significant language overlap, which helps with comparisons across platforms. It should however be noted that in most cases language versions differ – the latest available ones were selected for each platform.

Cold Start Cold start of function containers is a major performance bottleneck and a by-product of the nature of serverless platforms that need to conserve resources while offering seemingly infinite auto-scaling capabilities to users (Baldini et al.,

OpenWhisk	Fission
Python 3	Python 3
Go 1.11	Go 1.9
Java 8	Java 8
NodeJS 12	NodeJS 8
.NET 2.2	.NET 2.0
PHP 7.4	PHP 7.3
Ruby 2.5	Ruby 2.6
Rust 1.34	Perl 5.32
Swift 4.2	

Table 1: Supported versions of chosen language runtimes

2017)(Lloyd et al., 2018). Functions are being executed in containers that are instantiated on demand, and depending on continued use of the function, are shut down to free resources for other tasks. The startup time of containers is therefore important to measure in order to establish the overhead incurred. Furthermore, choice of runtime has been shown to significantly affect container startup time (Jackson and Clynch, 2018). The cold start tests are based on the work in (Jackson and Clynch, 2018). First, the cold start timeout has to be identified for each platform. An *exponential backoff* strategy was used to find the time needed to wait between function invocations to ensure a new container is instantiated. For OpenWhisk this was found to be 10 minutes, while the default Fission installation appears to keep containers warm for 3 minutes.

A test suite was developed to measure the cold start times for each language runtime. A total of 144 cold start invocations on empty functions were performed per language, per framework. Invocations are 10 minutes apart, running for a total of 24 hours per language. This is done to ensure accurate measurements of the average latency, regardless of fluctuations that might depend on time of day or current load of the host machine. The measurement is the execution latency as logged internally by each platform, ensuring unbiased results.

Warm Start *Warm* starts occur when a previously instantiated container is reused for a function execution. In practice this leads to much faster execution times, since the expensive container bootup process has already been performed. Warm starts are preferred by serverless users since they offer the best possible performance and there are many examples of strategies for "pre-warming" containers in anticipation of traffic (Silva et al., 2020). They are also the most accurate representation of runtime overhead for the average case of functions that are invoked often, such as in a Web application that has multiple concurrent users at any point in time.

A test suite was designed to ensure that each in-

vocation would lead to container reuse, while also ensuring accuracy by taking multiple measurements throughout the day. A set of 3 test runs were performed for each language, each consisting of 120 invocations on empty functions, 1 minute apart. There is a wait period of 2 hours between each run. Overall the entire test includes 360 invocations that cover 12 hours in sets of 2 with 2 hours in between. The design was also inspired by (Jackson and Clynch, 2018), where a similar approach was used. Again, the spread of the test runs was done to ensure results unbiased by external factors.

API Access As serverless platforms use an event-driven model of operation one very popular application for FaaS is an API server, built as a set of functions that take the role of endpoint request handlers. Both OpenWhisk and Fission provide a built-in mechanism for making it easier to access functions via Web requests. OpenWhisk uses *Web actions* that can be triggered via an *API* (OpenWhisk, 2021), while Fission introduces the concept of *HTTP triggers* (Fission, 2021b).

The previous two experiments focused on raw function execution time as measured internally by each platform. This eliminated any hidden API latency that might distort the results, and addressed **RQ1**. It is also worth investigating the overhead incurred by the API layer offered by the platforms, as well as identify any potential correlation with the choice of language runtime. This experiment addresses **RQ2**.

In order to measure API access latency and compare with existing results the experiments were designed as identical to the cold and warm start scenarios described in the previous sections. In particular, the 144 cold start and 360 warm start invocations were repeated with the same timings; the only difference being that instead of triggering the functions internally, using the provided command line tools and obtaining the logged metrics directly from the platform logs, each empty function will be tied to an API trigger and called via an HTTP request. The execution latency will then be measured externally by a specialised API load testing tool.

Hypotheses Throughout the experimental design the following hypotheses were formulated based on observation of similar research:

- **Hypothesis 1:** *Compiled languages will perform worse than dynamic languages in cold start scenarios.* This is based on overwhelming evidence in literature where on commercial platforms compiled languages like Java or .NET take longer to initialise the environment container (Wang et al., 2018). A few outliers have been identified, most

notably Go (Jackson and Clynch, 2018).

- **Hypothesis 2:** Warm start results will be close together, regardless of language. Similarly to **HI**, this was formulated based on observed trends in literature (Jackson and Clynch, 2018),(Vojta, 2016), so another hypothesis is that something similar will be observed.
- **Hypothesis 3:** API overhead will be minimal and constant across all languages. This one is more of a conjecture, as there is no significant research being done in this area, however, the overhead should be minimal otherwise it would be impractical for actual applications.

Cloud Testbed In order to provision the physical and virtual resources required to ensure their proper operation, the experimentation was performed on a Cloud testbed available at the University of Leeds comprising a 14 node cluster. It uses OpenNebula 4.10.2 (OpenNebula, 2021) as a virtual infrastructure manager to offer virtual resources, including VMs and storage volumes. The typical node that was considered for measurement is a Dell PowerEdge R430 Server commodity server with two 2.4GHz Intel Xeon E5-2630 v3 CPUs with 128GB of RAM, a 120GB SSD hard disk and an iDRAC Port Card.

5 PERFORMANCE EVALUATION

5.1 Openwhisk and Fission

A summary of the experiment results for Fission is shown in Figure 1. For warm starts a relatively stable performance is observed across languages with average execution time in the 20-60ms range. Of note is that compiled languages are not necessarily slower than dynamic in warm start scenarios - Golang is the best performer, with .NET a close second. Java is performing the worst and also has the most fluctuation in the results. Request times are slightly higher, as expected, but overall follow the same trend as raw execution times, with a few minor exceptions which will be looked at more closely later in this section.

For cold starts some patterns are observed:

- Compiled languages (.NET C#, Golang, Java) are slower than their dynamic counterparts across the board. In particular, .NET performs very poorly with average cold start execution time over 4 seconds. This is in stark contrast with the warm start tests, where .NET was one of the top performers.
- .NET has the only inconsistency in the entire dataset when it comes to raw vs request execu-

tion times. Usually request times are slower but in this case .NET displays the opposite. Furthermore, it appears that Fission's API mechanism adds very little overhead, so the reversed behavior is attributed to statistical fluctuation.

- Overall cold start performance is fast, with most languages staying under 500ms execution time. This is attributed to the executor type used for instantiating environment containers – PoolManager, the strategy to keep a small pool of warm generic containers that can quickly be specialised for the particular runtime requested. Having such pool in place, the overhead is expected to be less prominent than in a full initialisation.

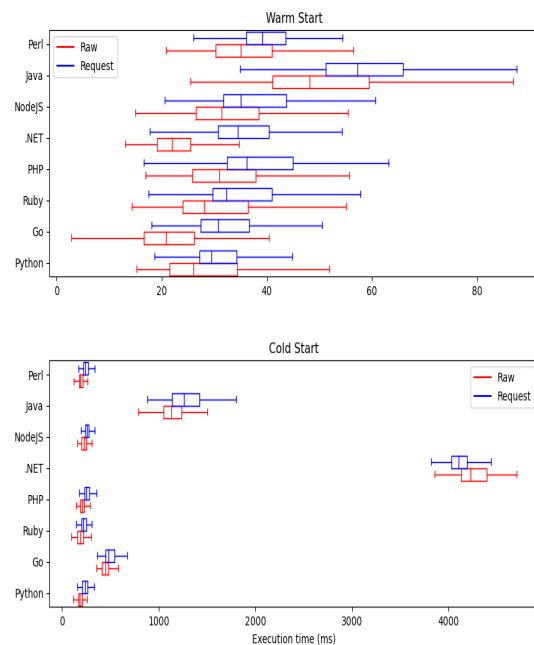


Figure 1: Summary results for Fission, both raw and request executions, presented with boxplots using Q3-Q1 interquartile range (IQR)

Overall, all Fission runtimes appear consistent in warm start scenarios. For cold starts, .NET and to a lesser extent Java are not recommended. Golang is the best performing compiled language, while Python is the winner in terms of overall performance. API mechanism appears very lightweight, adding minimal overhead.

Figure 2 contains the same summary for the OpenWhisk experiments. Warm starts show a similar consistency as in Fission, except for Ruby, which has surprisingly slow execution time averaging over 500ms. When compared to Fission for the same language the results are not repeated, which points to some sort of

inefficiency in the implementation of the Ruby runtime for OpenWhisk. Again, compiled languages are seen performing slightly better overall for warm starts with Rust, Swift, .NET and Golang tied for first place with PHP being the only dynamic language to achieve similar performance. Raw requests are also very consistent as seen from the low variance. Finally, the considerable overhead that the API mechanism incurs is observed, compared to Fission.

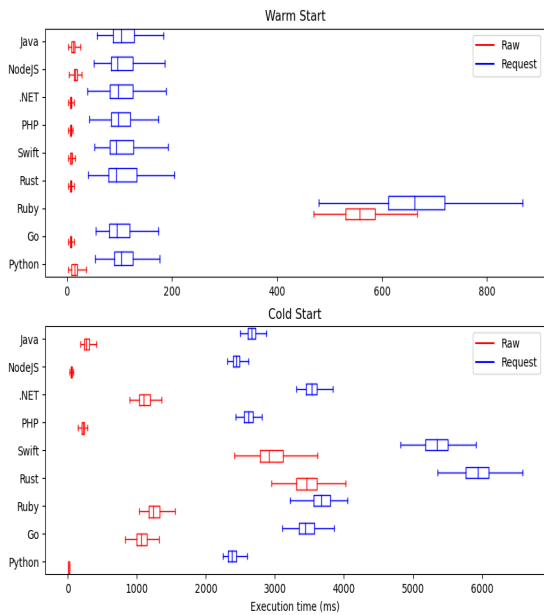


Figure 2: Summary results for OpenWhisk, both raw and request executions, presented with boxplots using Q3-Q1 IQR

For cold starts, Rust and Swift are the slowest languages, averaging around 3 seconds cold start for raw requests. However, Ruby is the next slowest, which can be linked to the bad performance observed during warm starts, further solidifying the issue with that particular runtime. Java however performs on par with dynamic languages such as PHP and NodeJS. Overall, Python and NodeJS are the clear winners in terms of cold start performance, averaging around 100ms overhead. The disparity between raw and request execution times is even bigger and more pronounced.

Figure 3 compares the raw and request execution times for Fission, in order to showcase the differences observed between the two modes of operation. Cold and warm starts present similar results, with request times being slightly above their raw counterparts, which at first glance points at minimal overhead in Fission's API mechanism. However, there is an inconsistency in the cold start performance of .NET, with raw execution being around 120ms slower on av-

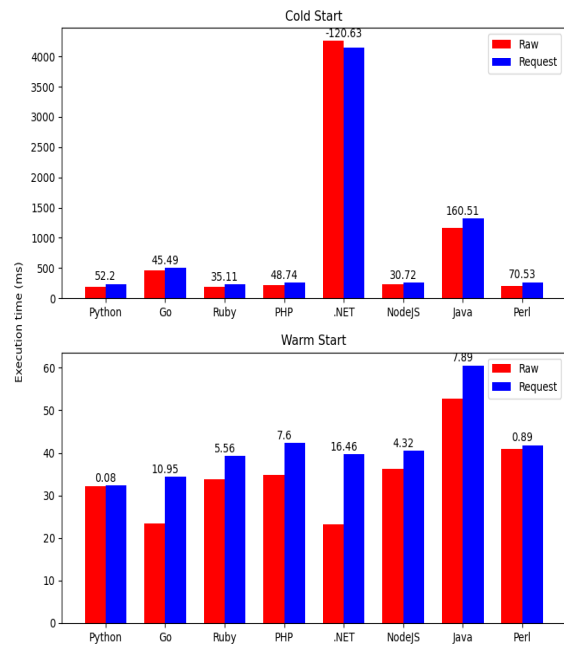


Figure 3: Raw/request difference in average execution times for Fission. Number above each pair is (Request time - Raw time) in milliseconds

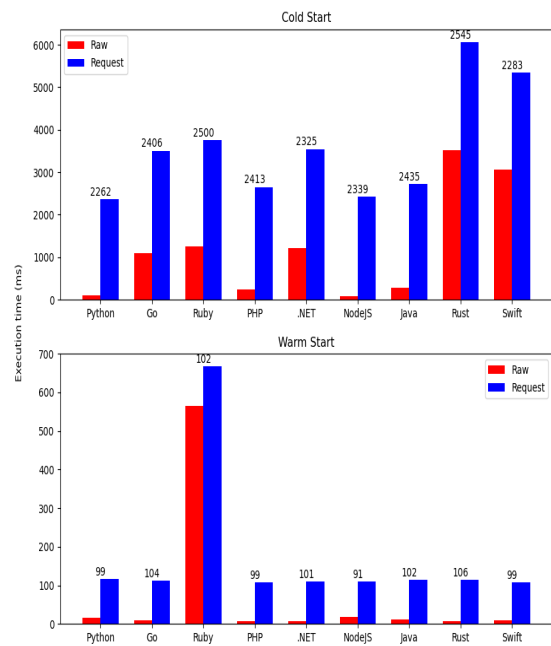


Figure 4: Raw/request difference in average execution times for OpenWhisk

erage than requests. This is not a significant difference however as the overall execution time is around 4 seconds for that particular case and is attributed to statistical fluctuation. For warm starts Python and Perl times are extremely close, with a sub millisecond average difference between the two modes, further rein-

forcing the view that times being measured are very close together and a small anomaly in any one direction is attributed to statistical error.

The results for Fission can be interpreted as an insight into the logging mechanisms that are used to implicitly obtain the raw measurements. The metrics obtained through Prometheus (Fission, 2021a) are leveraged, which in turn uses data logged internally by Fission into an InfluxDB time-series database. Furthermore, runtime environments in Fission always come with an HTTP server for receiving function execution requests. It is therefore reasonable to assume that the execution times logged in the database are retrieved from the web server request logs residing in each function pod and include the roundtrip time from the runtime container to the web server. Furthermore, when using HTTP triggers to test the API functionality, the router component which directly communicates to the function pods is exposed; a lot of overhead is skipped by bypassing the controller. This explains the closeness of the results of the two modes of operation, as in the case of API requests. The extra time it takes for the router to route the request to the function pod is simply measured, which could be minimal, especially in the warm start scenarios where the function pod addresses are already in the router cache.

Overall, due to the closeness of the results and inconsistencies that do not present any clear pattern, it is concluded that Fission raw results do not measure purely the function execution time and cannot therefore comment on the API overhead incurred by the HTTP triggers. The two modes of operation have similar performance and make general comparisons between languages.

OpenWhisk is a different story and Fig. 4 plots the same data as in the Fission case. The request times are always slower, and by a relatively consistent amount of 100ms for warm starts and 2300-2500ms for cold starts. This is an unexpected slowdown, especially for cold starts, since it imposes a significant overhead to an otherwise competitive raw performance, and points to an inefficiency in the request routing.

One reason for this disparity is the fact that API access in OpenWhisk is facilitated the same way as any other request, through its top level HTTP web server. Therefore, it needs to go through more system layers to reach the invoker and function containers. Additionally, unlike in Fission, OpenWhisk's architecture is built around asynchronous invocation and has a Kafka message queue at the core of its system where function invocation messages are sent and await to be picked up by an appropriate invoker. This asynchronous design has some inherent delay whenever synchronously block waiting for the result

is tried.

Overall, API access in OpenWhisk has a clear and consistent overhead across all languages and test scenarios, and is much more pronounced in cold start scenarios.

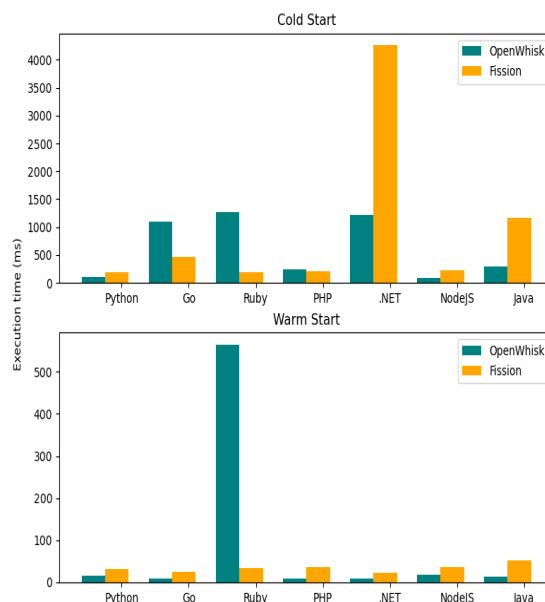


Figure 5: Average raw execution times for common languages

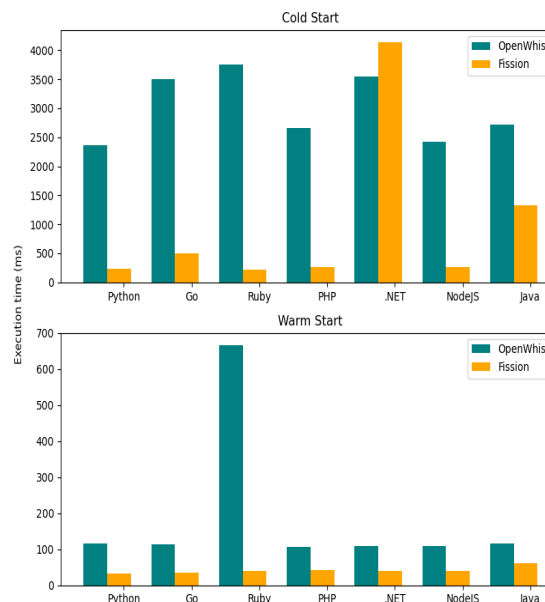


Figure 6: Average request time for common languages

Figure 5 does a platform comparison for the raw execution scenarios of common languages between OpenWhisk and Fission. With the exception of the

outlier Ruby, Golang and PHP are the only languages that perform better for Fission in cold start scenarios. The divide is most prominent for .NET with a 3 second difference. For warm starts, OpenWhisk is the clear winner (except for Ruby), but as observed previously, the comparison between the two frameworks is not entirely fair in the raw experiments, so any definitive conclusions cannot be made, especially since warm start performance is so close.

API requests are measured using the same tool so the results in figure 6 can be compared more confidently. OpenWhisk’s API overhead is clearly showing in all cases, with the only exception being .NET in cold starts, further showcasing the runtime’s bad performance on Fission.

5.2 Comparison with AWS Lambda

In order to address RQ3, a comparison with published research on commercial platforms is performed to establish any discrepancies. In particular, Jackson and Clynch (Jackson and Clynch, 2018) run benchmarks considering empty functions on AWS and Azure for .NET 2, Go, Python, Java and NodeJS, with complete overlap on the languages that are tested in this paper. For the purpose of the comparison only raw execution times are considered, since those are the results presented in the relevant literature. Another important point to consider is that AWS Lambda uses Firecracker micro-VMs (Firecracker, 2021) which provide enhanced security and workload isolation over traditional VMs, while enabling the speed and resource efficiency of containers.

In particular, focusing on the AWS results, warm starts have a consistently low runtime overhead, with Go being the slowest at 19ms average time, while Python and .NET performing the best with around 6ms. OpenWhisk’s fastest times are mainly all compiled languages at 8ms while the dynamic languages go up to 17ms, except Ruby, which for the purpose of the comparisons will be excluded as an extreme outlier. Similarly for Fission, Python, Go and .NET are the top performers, contradicting the bad performance of Go on AWS. However the overall warm start times in Fission are much slower than the ones presented by Jackson and Clynch, with the fastest averaging 23ms. For cold starts in (Jackson and Clynch, 2018) Java and .NET are the slowest with a significant margin. Go appears as an outlier as it performs better in cold starts than in warm starts at about 9ms, while Python is the clear winner at just below 3ms. OpenWhisk results show a clear distinction between compiled and dynamic languages; Java is the only one that is considered an outlier with an average execu-

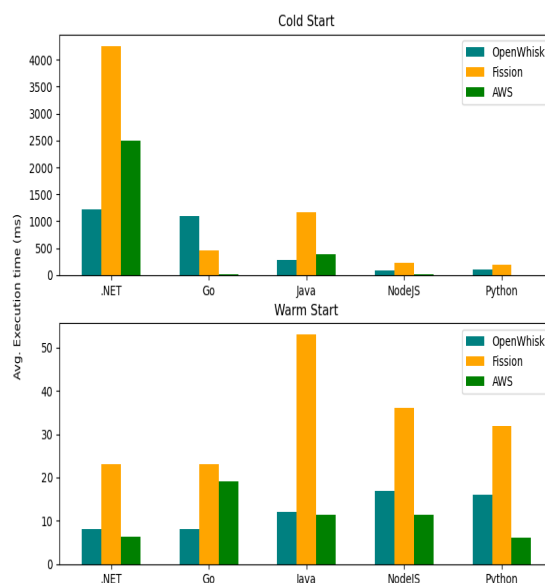


Figure 7: Comparison of common languages with (Jackson and Clynch, 2018)

tion time of 289ms. The faster language is NodeJS with 82ms, while the slowest ones (Swift and Rust) are much slower by about 500-1000ms than the worst performer on AWS, .NET. Fission also has a clear distinction between the slower compiled languages, with the fastest being Go, however still not performing as well as in AWS, and furthermore the cold start times for Fission are a bit higher than OpenWhisk on average.

The results are summarised in Figure 7. General observations include : 1) OpenWhisk’s warm start performance on compiled languages rivals those on AWS, while Fission exhibits some delays, especially for Java, NodeJS and Python; 2) the unexpected cold start performance of Go on AWS was not replicated in the experiments, although Go was amongst the top 2 compiled languages on both platforms; 3) Fission has a generally larger overhead, although this is attributed to the uncertain nature of the logging records for the raw measurements; 4) Cold start performance of dynamic languages on AWS could not be matched, and 5) With the exception of Go, the general trend of compiled languages performing worse in cold starts matches the observations.

Note that the investigation of AWS cold start runtime performance in (Sbarski et al., 2022) reports .NET and Java with the worst cold start performance while NodeJS and Python with the best results, Python displaying sub-millisecond cold start average for most memory sizes.

5.3 Evaluation of Research Hypotheses

The hypotheses formulated in section 4 are evaluated in light of the research findings.

Hypothesis 1: *Compiled languages will perform worse than dynamic languages in cold start scenarios.* For the most part this turned out to be correct. A few compiled languages came close to overturning this hypothesis, namely Go for Fission and Java for OpenWhisk. However, with the exception of Ruby on OpenWhisk, no compiled language had a better average performance than a dynamic one in cold start scenarios.

Hypothesis 2: *Warm start results will be close together, regardless of language.* This also turned out to be correct, with most languages averaging similar performance. Compiled languages on OpenWhisk had a particularly good showing in this regard, while on Fission Java and Perl were lagging a bit behind. However the differences were not significant enough to warrant a closer investigation. Ruby on OpenWhisk was once again excluded from this comparison since it appears to be an extreme outlier.

Hypothesis 3: *API overhead will be minimal and constant across all languages.* This hypothesis was the only one not informed directly by the literature and it turned out to be incorrect for OpenWhisk. The overhead imposed by the API mechanism was extremely large at 100ms for warm and 2.5s for cold starts. However it did not appear to be affected by a particular language as it was constant throughout. Fission results were closer to expectations but the analysis showed that the raw measurements might include hidden overhead which prevents from performing a comparison.

5.4 Review of Research Questions

The performance results mostly follow the research performed in (Jackson and Clynch, 2018),(Sbarski et al., 2022),(Wang et al., 2018). The few differences that were identified were mostly related to the superior performance of AWS, which was expected. The research questions posed in section 3 are reviewed in order to evaluate to what degree they were answered.

RQ1: *What impact does choice of language have on function execution time?* The choice of language has a significant impact, depending on the use case and platform. OpenWhisk has the overall best performance when measuring raw execution. Ruby is a problematic runtime for that platform and should be avoided. Otherwise all languages perform about the same in warm starts. For cold starts the choice is much more meaningful; languages like Rust or

Swift incur a much bigger overhead over choices like Python or NodeJS. As a general rule of thumb, compiled languages are slower although to differing degrees. Fission has the same consistent performance in warm starts across all available runtimes, with Java being a little bit on the slower side. Cold starts follow the same trend of compiled versus dynamic, but with less variability than in OpenWhisk - .NET is the slowest by a large margin, followed by Java, while Go is almost on par with the dynamic languages.

RQ2: *What overhead does the API mechanism incur?* This was answered for OpenWhisk, and the results were useful for comparing the two platforms. OpenWhisk has a prohibitively large overhead when the function is invoked through a web action; it is somewhat acceptable for warm starts but cold starts add a pretty noticeable delay which can definitely impact the performance of real-time applications. Fission's API overhead could not be established due to the nature of the logging facilities and concerns about the validity of the raw measurements. However, the overall performance is superior to OpenWhisk by a large margin for all but one language - .NET. Additionally, based on these findings it is concluded that any overhead present does not appear to be correlated with the choice of runtime.

RQ3: *Are results competitive with commercial platforms?* Considering raw execution times, the open-source platforms investigated are not at the same level but still have a decent performance and can definitely be optimised further. OpenWhisk has very competitive warm start execution times, especially for compiled languages, surpassing some of the results seen in literature for languages like Java or Go. Cold starts are also faster for certain compiled languages but the best performers on AWS are ahead by a significant margin. Fission is generally slower in warm and cold starts than OpenWhisk with the exception of a couple languages like Go and Ruby. It is still far behind the top performers on AWS and Azure.

6 CONCLUSION

This paper investigated the impact the choice of language runtime has on function performance in local deployments of Apache OpenWhisk and Fission. Overall, compiled languages perform better in warm starts and worse in cold starts, but the difference in the latter is significant, making dynamic languages the overall better choice - Python being the best common denominator. When using the recommended languages OpenWhisk performs better than Fission in raw measurements, while Fission is the superior

choice for applications using HTTP triggers.

Some areas for further research include: 1) evaluation of more trigger types for invoking functions (e.g. database updates, timers, message queues); 2) evaluation of more platforms (e.g. Knative, OpenFaaS, Kubeless and Iron Functions); 3) investigation of performance under different configuration (e.g. different container sizes); 4) performance evaluation under load (e.g. a high-traffic scenario when server scaling is introduced may give insight into platform performance under stress); 5) further dive into Fission's internals in terms of provisioning new container types and 6) custom runtimes: both platforms offer the ability for a custom executable to be used as a runtime environment. Therefore, a comparison with the default offerings is useful to understand the performance impact.

ACKNOWLEDGEMENTS

The authors would like to thank the European Next Generation Internet Program for Open INTERNET Renovation (NGI-Pointer 2) for supporting this work under contract 871528 (EDGENESS Project).

REFERENCES

- Amazon Web Services (2015). AWS Serverless Multi-Tier Architectures With Amazon API Gateway and AWS Lambda. Technical report, Amazon Web Services.
- Azure (2021). Azure functions. <https://docs.microsoft.com/en-us/azure/azure-functions/>.
- Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., and Suter, P. (2017). Serverless computing: Current trends and open problems. *CoRR*, abs/1706.03178.
- Djemame, K., Parker, M., and Datsev, D. (2020). Open-source serverless architectures: an evaluation of apache openwhisk. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pages 329–335.
- Firecracker (2021). Firecracker: Secure and fast microvms for serverless computing. <https://firecracker-microvm.github.io/>.
- Fission (2021a). Fission: Metrics with prometheus. <https://docs.fission.io/docs/observability/prometheus/>.
- Fission (2021b). Open source, kubernetes-native serverless framework. <https://fission.io>.
- Google (2021). Cloud functions. <https://cloud.google.com/functions>.
- Großmann, M., Ioannidis, C., and Le, D. (2019). Applicability of Serverless Computing in Fog Computing Environments for IoT Scenarios. In *Proc. of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, page 29–34, Auckland, NZ. ACM.
- Jackson, D. and Clynch, G. (2018). An investigation of the impact of language runtime on the performance and cost of serverless functions. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion*, pages 154–160.
- Kritikos, K. and Skrzypek, P. (2018). A review of serverless frameworks. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 161–168.
- Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., and Pallickara, S. (2018). Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 159–169.
- Manner, J., Endreß, M., Heckel, T., and Wirtz, G. (2018). Cold start influencing factors in function as a service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion*, pages 181–188.
- Mistry, C., Stelea, B., Kumar, V., and Pasquier, T. (2020). Demonstrating the practicality of unikernels to build a serverless platform at the edge. In *2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 25–32.
- Mohan, A., Sane, H., Doshi, K., Edupuganti, S., Nayak, N., and Sukhomlinov, V. (2019). Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA. USENIX Association.
- OpenFaaS (2021). Openfaas - serverless functions, made simple. <https://openfaas.com/>.
- OpenNebula (2021). Open source cloud computing and edge computing platform. <https://opennebula.io/>.
- OpenWhisk (2021). Open source serverless cloud platform. <https://openwhisk.apache.org/documentation.html>.
- Sbarski, P., Cui, Y., and Nair, A. (2022). *Serverless Architectures on AWS*. Manning, 2nd edition. To appear.
- Scheuner, J. and Leitner, P. (2020). Function-as-a-service performance evaluation: A multivocal literature review. *Journal of Systems and Software*, 170:110708.
- Silva, P., Fireman, D., and Pereira, T. (2020). Prebaking functions to warm the serverless cold start. In *Proc. of the 21st International Middleware Conference, Middleware '20*, page 1–13, NY. ACM.
- Vojta, R. (2016). AWS journey — API gateway & Lambda & VPC performance. <https://www.zrzka.dev/aws-journey-api-gateway-lambda-vpc-performance/>.
- Wang, L., Li, M., Zhang, Y., Ristenpart, T., and Swift, M. (2018). Peeking behind the curtains of serverless platforms. In *Proceedings of the 2018 USENIX Annual Technical Conference*, page 133–145, USA. USENIX Association.