Measuring Baseline Overheads in Different Orchestration Mechanisms for Large FaaS Workflows

George Kousiouris
Dept. of Informatics
Harokopio University
Athens, Greece
gkousiou@hua.gr

Chris Giannakos
Dept. of Informatics
Harokopio University
Athens, Greece
chrisgiannakoss@hua.gr

Konstantinos Tserpes
Dept. of Informatics
Harokopio University
Athens, Greece
tserpes@hua.gr

Teta Stamati
Dept. of Informatics
Harokopio University
Athens, Greece
teta@hua.gr

ABSTRACT

Serverless environments have attracted significant attention in recent years as a result of their agility in execution as well as inherent scaling capabilities as a cloud-native execution model. While extensive analysis has been performed in various critical performance aspects of these environments, such as cold start times, the aspect of workflow orchestration delays has been neglected. Given that this paradigm has become more mature in recent years and application complexity has started to rise from a few functions to more complex application structures, the issue of delays in orchestrating these functions may become severe. In this work, one of the main open source FaaS platforms, Openwhisk, is utilized in order to measure and investigate its orchestration delays for the main sequence operator of the platform. These are compared to delays included in orchestration of functions through two alternative means, including the execution of orchestrator logic functions in supporting runtimes based on Node-RED. The delays inserted by each different orchestration mode are measured and modeled, while boundary points of selection between each mode are presented, based on the number and expected delay of the functions that constitute the workflow. It is indicative that in certain cases, the orchestration overheads might range from 0.29% to 235% compared to the beneficial computational time needed for the workflow functions. The results can extend simulation and estimation mechanisms with information on the orchestration overheads.

CCS CONCEPTS

General and reference→ Cross-computing tools and techniques→ Measurement •General and reference→ Cross-computing tools and techniques→ Performance•Computer Systems Organization→Architectures→Distributed Architectures→Cloud computing

ICPE '22 Companion, April 9–13, 2022, Bejing, China

ACM ISBN 978-1-4503-9159-7/22/04□$15.00
https://doi.org/10.1145/3491204.3527467

KEYWORDS

Serverless, FaaS, Openwhisk, Orchestration, Performance, Overhead

ACM Reference format:

George Kousiouris, Chris Giannakos, Konstantinos Tserpes and Teta Stamati, 2022, Measuring Baseline Overheads in Different Orchestration Mechanisms for Large FaaS Workflows. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering, April 9--13, 2022, Bejing, China*, DOI: 10.1145/3491204.3527467.

1 Introduction

Serverless computing has emerged as a promising alternative execution model within cloud computing, promising better utilization of underlying resources, easier and inherent scalability as well as agnostic management. One specialization of the serverless paradigm, the Function as a Service [1] approach, aims to apply the serverless scope also in the way application logic is created, embedded and executed. The logic needs to be split up in smaller chunks of code, namely functions, that are in principle stateless. A series of functions may collaborate in a workflow (together as well as with external services) in order to implement the end to end logic needed by the application layer.

At the moment, as indicated by related reviews [2], the size of FaaS applications is rather small, indicating that approximately 82% of them have only up to 5 functions in the workflow. New tooling that is available for the domain, in terms of inherent orchestration mechanisms of the various platforms, as well as the hype around the domain may lead to more complex application flows creation. Recent research attempts have highlighted new and more versatile orchestration means [3] based on additional layers of orchestrators that can mitigate functional drawbacks of current FaaS platforms, offering visual design and implementing workflows of functions with support for more complex workflow primitives and subflow groups.

One of these solutions is the PHYSICS Design Environment[1], utilizing the IoT originating Node-RED tool. The latter is a visual, functional programming, event driven framework for application

[1] https://physics-faas.eu/

development. Due to the existence of a sophisticated runtime (based on node.js), its asynchronous nature as well as abundance of ready-made functional nodes, a Node-RED flow can be easily created and act as an orchestrator of functions that are executed on a FaaS platform. The functions can execute either inside the Node-RED runtime or can be invoked through relevant clients of the FaaS platform.

The contribution of this paper is to investigate and measure the performance overheads between primarily the Openwhisk built-in sequence operator and the Node-RED based orchestration mechanisms and compare the delays inserted by each orchestration mode. The delays can be due to initialization times of the orchestration logic or due to the inter-function invocation delays needed for passing through each step of the workflow. Then the orchestration overhead can be compared to the actual useful computational time of the workflow and one mode or the other can be selected based on the anticipated size and computational time needed for the workflow. The results may be useful in extending simulation models for FaaS applications so that the latter take orchestration overheads under consideration.

The rest of the paper proceeds as follows. In Section 2, related work is examined with relation to the work presented in this paper. Section 3 describes the orchestration mechanisms taken under consideration, while Section 4 introduces the measurement methodology and experiment setup. Section 5 presents the measurements and analysis of the results in order to model the time delays. Finally, Section 6 concludes the paper.

2 Related Work

A number of works [4,5] indicate the need for enhanced tooling for function grouping, reuse and composition in Function as a Service platforms. However, this also needs the ability to interact between these groups through appropriate orchestration mechanisms [6]. From the main open source FaaS platforms, Openwhisk[2] natively supports only a sequence operator at the runtime level [7]. Relevant add-ons such as the IBM Composer[3] exist, including an extended set of orchestration primitives, in the form of a code library [8]. For other open source platforms such as OpenFaaS, external plugins are also available (e.g. FaaS-flow[4]), that follow a similar approach, supporting more complex workflow primitives in a code-like manner. In this case the orchestrating code is by itself executed as a function. TriggerFlow [10] is an add-on mechanism that supports different workflow primitives, as well as eventing mechanisms.

From a visual workflow creation point of view, Kubeflow [9] includes a relevant language for pipeline definition and an editor extension for visual definitions of workflows. The defined workflow resembles more to a static definition of steps, without a respective runtime, orchestrating the execution of one task after the other, while the inputs and outputs are passed through external object storage services. AWS Step functions supports visual programming style and extended operators for function workflows (e.g. state management ones). Google Cloud Functions[5] are based on text based yaml files for the definition of a workflow. The same yaml approach applies for the AFCL approach presented in [15]. In general yaml based approaches can become very complex when the size or connections in the workflow scale, although in this case the solution comes also with a rich set of available constructs that can significantly speed up application creation.

[14] has moved the execution of scientific workflows to a FaaS model with Hyperflow and compares it to the traditional IaaS approach from a cost point of view, not covering however orchestration delays in particular. The paper proposes a number of architecture alternatives for workflow orchestration, with the Direct Executor variation (the main flavor used in its experimentation) being very similar to one of the variations we measure in this paper (the OW-NR mode described in Section 3). In [16], SWEEP acts as a workflow management system and language, executed as a server, thus may present scaling limitations if all workflows need to be regulated through a central instance. In terms of workflow primitives, it supports an increased number of them as well as the ability to handle both function and container execution. The overheads examined in this case relate to AWS API throttling or invocation retry aspects in case of failures.

What is evident from the related work is that a number of orchestrating options are currently available in FaaS platforms, however the performance footprint of these has not been extensively investigated. Orchestration overheads intersect two out of the 3 performance challenges identified in [11], namely the request overheads and the function lifecycle management aspects. One exception investigating performance issues in depth is Netherite [18], in which a distributed execution engine is presented. Netherite applies speculation for minimizing delays from state management, which aids in increasing the orchestration throughput and workflow latency.

In [8], a performance analysis is conducted for fork-join executions between Amazon Step Functions, Azure Durable Functions and IBM Composer, in the form of overheads from multiple concurrent executions. The most similar to our work is [12], in which the pure orchestration needs are measured in sequences of operations for the same providers as [8]. Interesting findings are reported with relation primarily to the state transition delays affected by the function input size and how this affects the total overhead. Other approaches investigate more futuristic implementations deploying the orchestrators at the Smart NIC level, for minimizing latency in function orchestrations [13]. Beldi [17] is a library and runtime system for supporting stateful serverless functions. It can be combined with this work in an effort to include state management operations in the orchestration overheads investigation.

[2] Apache Openwhisk, Available at: https://openwhisk.apache.org/

[3] Apache Openwhisk Composer, Available at: https://github.com/apache/openwhisk-composer.

[4] Faas-flow orchestrator for OpenFaaS, Available at: https://github.com/s8sg/faas-flow

[5] Google Cloud Functions Workflow Specification, Available at: https://cloud.google.com/functions/docs/tutorials/workflows.

In the context of this work, the PHYSICS Design environment, suggested by [3], utilizes the Node-RED framework in order to support visual editing of functions as well as workflows packaged and executed as functions on target FaaS platforms (mainly Openwhisk). In this manner, any workflow primitive that can be constructed through appropriate message handling in the runtime can be implemented and piggy back on the Node-RED workflow specification. Invoked functions can be external ones or internal ones included in the runtime. It also enables the grouping and reuse of function flows (in the form of patterns) that can be used for creation of workflow primitives and an abundance of IoT-related nodes.The specific approach is very adaptable to a variety of scenarios, including Edge/Cloud collaborations for data collection. A complete analysis of the reasons behind the selection of Node-RED as the main orchestrator for the PHYSICS platform can be found in [19].

3 Orchestration and Execution Variations

The first observed mode as mentioned in the Introduction is the sequence operator for OpenWhisk (OW) runtime functions. This targets workflows that are defined as function sequences directly in OpenWhisk, utilizing registered (to OpenWhisk) functions as their main building block (Figure 1a).

Selection rationale: this mode was selected since it is the simplest one, involving only native mechanisms

The second mode is a function sequence in Node-RED (NR), executed as a docker action in OpenWhisk (Figure 1b). The main difference in this case is that the function workflow is created in Node-RED and deployed within a custom docker image (invoked as a function) that contains the Node-RED runtime. Openwhisk supports black box images to be executed as functions. In this case all the functions reside and execute inside the same environment container.

Selection rationale: This mode was selected since the runtime can act as an orchestrator as well as a function execution environment.

This mode has a disadvantage from a parallelization point of view in a general usage context. Its main advantage is the fact that it includes a complete runtime, that has message tracking abilities, and thus it is able to apply more complex workflow primitives like Fork-Join patterns[6]. Furthermore, function subflows can be grouped and reused, while IoT nodes can extend data collection and collaboration with external systems.

Given that we can expect a significant delay in the start-up of the Node-RED environment, the purpose is to investigate what (and if there) is the benefit of one mode versus the other compared to the number and duration of the functions that are part of the sequence. The motivation behind this is the fact that inter-function communication (ifc) times in the sequence will probably differ between the two cases of execution, which may create a margin of exploitation of one mode over the other.

This is reasonable to assume, given that in a native OW sequence, the invocation from one function to another needs to pass through the respective OW mechanism, the new action invocation to be queued, and a container found for execution (in all potential variations such as cold or warm). On the other hand, in the Node-RED Action Image, the invocation is passed between the functions inside the Node-RED runtime without the need to find further containers (reused or new ones). In the case of multiple users, the OW runtime may also be forced to cold start more given that each function needs a separate container. In the context of this work this parameter is not investigated, however container reuse delays for the warm executions are also expected.

A third option is also possible, that is having an orchestration flow inside NR that invokes typical functions, deployed in any manner in the Openwhisk environment. In this case the NR flow acts as a generic orchestrator, while we can apply parallelization if applicable (e.g. in the aforementioned Fork Join case). This mode of orchestration (in generic context in Figure 1c) is expected to have higher orchestration delays, since it needs to invoke the OW interface. It also suffers from a double billing issue (similarly to the IBM Composer, FaaS-flow and Hyperflow approaches of Section 2) since the coordinating Action needs to execute for the overall duration of the workflow execution. On the other hand, it exemplifies the benefit of the NR orchestration with any type of workflow primitive.

Selection rationale: This mode was selected since it resembles more the architecture of typical orchestrator alternatives examined in the related work, in which orchestrator functions are only used for coordination.

A summary of the various approaches is included in Table 1.

Table 1: Summary of different orchestration-execution modes

Mode	Execution Environment	Orchestration Environment	Advantages	Disadvantages
OW-OW	OW node.js image	OW Sequence	Low cold start, any combination of execution images	Simple sequences only
NR-NR	Custom NR action image	NR runtime	Arbitrary workflow primitives, IoT adaptation	High cold start, execution in a specific type of image
OW-NR	OW node.js image	NR runtime	Arbitrary workflow primitives, actual parallelism inside the flow, any execution image, IoT adaptation	Double billing for the orchestration action, higher cold start delays, more interactions

[6] Split Join Node-RED pattern, available at: https://flows.nodered.org/flow/7a5acfc999b1ad47bb32b5d37419c777

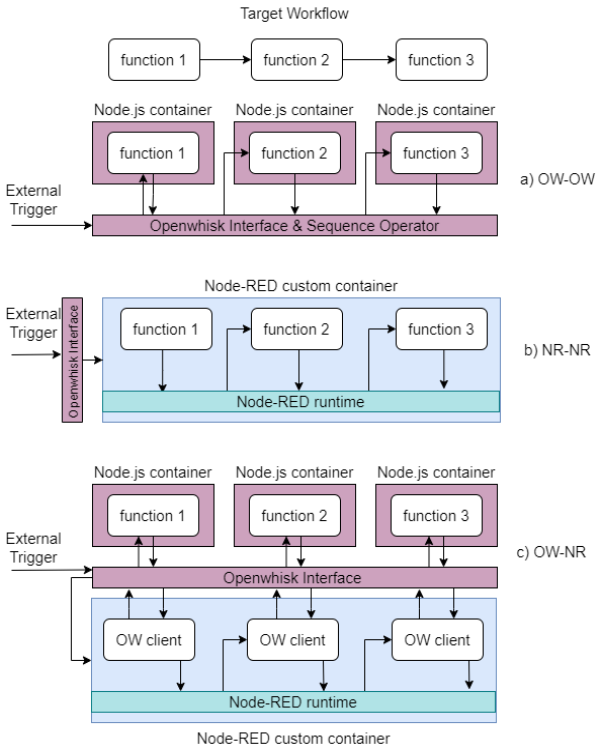


Figure 1: Comparison of the architecture of the three orchestration modes .

4 Experiment Description

4.1 *Experiment Setup*

Given that we want to measure the pure orchestration delays between function invocations in the three modes, we need to create a set of functions for which the execution delay is known and set a priori (i.e. a sleep function). In this manner one is able to measure the total execution time of the workflow (Figure 2) and subtract from this the total delays of the function executions (number of functions X inner sleep delay of each function). The remaining time will be any initialization time as well as the inter-function communication/orchestration delays (ifc).

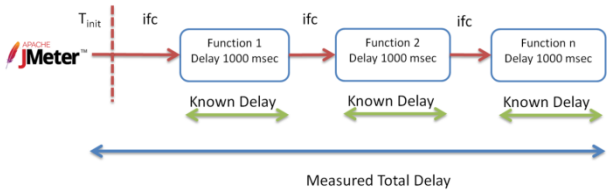


Figure 2: Measurement approach for the inter-function communication/orchestration delays (ifc)

As mentioned in the Introduction, the aim is to investigate the pure, baseline delays needed for transcending from one function to the next. For this reason, we have strived to reduce potential interferences from external factors that may influence the total time of the execution. One of them is any complications (e.g. cold starts) that may be caused by increased traffic, higher to the number of warm containers available. Therefore, only one client request was active at any given time. All the experiments were performed in warm containers, similarly to the approach in [12], in order to avoid cold start delays. Furthermore, the client resided in the same node in order to reduce any network latencies. The experiment was executed on a local node (AMD Ryzen 3500, 6 GB RAM) , running Ubuntu 20.04.3 LTS, with Linux 5.13.0-28-generic kernel and x86-64 architecture, Java version 11.0.13, Apache Jmeter version 2.13.20180731, Docker version 20.10.7, Openwhisk , Node-RED version v2.0.6 and Node.js version v14.17.06

4.2 *OW-OW mode*

An artificial delay function has been created as a native Javascript function (Snippet 1), accepting as a parameter the milliseconds of delay to apply. This function is registered as a nodejs function type in OpenWhisk (OW-OW case). The delay is received as an input argument and is applied through a setTimeout method. This in typical node.js environments may not be accurate in terms of the final delay, since it needs to be interpreted as the minimum delay to be asserted. However the actual delay may be larger due to contention in the node.js process. When the timer expires, the reactivation of the function is pushed to the end of the FIFO event queue. So if the event-loop has a large number of events, the reactivation may delay further than the desired delay. However, in cases such as our measurement, in which there were no other functions contending inside the same node.js process, there was no measured difference between the desired and the set value.

Observation: The specified delay implementation should be applied only in non congested nodejs eventloops.

Snippet 1: Delay function for the node.js action

```
function main(params) {
    return new Promise(function(resolve, reject) {
      setTimeout(function() {
        resolve(params);
      }, params.delay);
    })
}
```

Parametrically named sequences of varying numbers of functions (e.g., 1-25 with a step of 5), were set up as OpenWhisk function sequences, in order to be invoked by a respective Jmeter client. Actions were exposed as web actions.

4.3 *NR-NR mode*

A similar function flow is created for Node-RED (Figure 3). The core flow handles a control loop where delay and iteration numbers are dynamically passed on as input variables during the action invocation. This is needed to enhance experiment automation, so that we can pass as arguments the number of repetitions of the main delay loop, in order to simulate a flow of n functions, each with the desired delay. The respective flow should also be executable as an Openwhisk function, which means it needs to abide by the respective interface of the latter (one POST /init method and one POST /run method). The function node in

the middle subflow extracts parameters from the incoming message and passes them through the msg.delay and msg.iterations fields. The whole process is iterated until the number of needed delays are met.

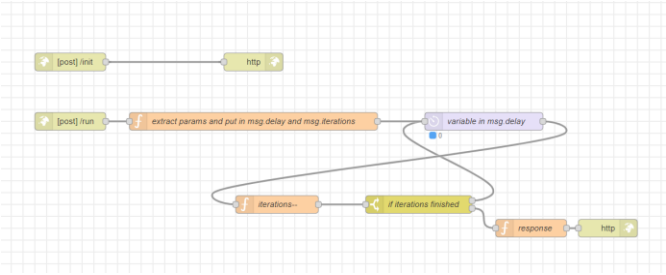


Figure 3: Node-RED Workflow implementation for dynamic delay and iterations (NR-NR mode)

A docker image with the Node-RED delay flow was created and registered as an OpenWhisk Docker Action for the NR-NR case. The image is available on dockerhub[7]. The delay flow is also available as a standalone flow[8].

4.4 *OW-NR mode*

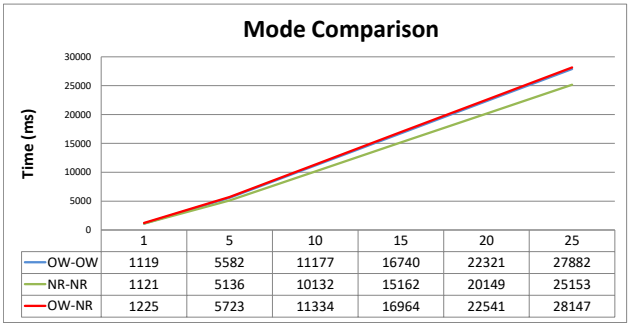
For the hybrid case, the purple NR delay node of the previous figure has been replaced by the Openwhisk Node-RED client node[9] that contacts the basic node.js delay action of the OW-OW case. Thus in this case the orchestration is performed through the Node-RED action flow but the actual execution is performed by a native node.js Openwhisk function. The respective image has also been made available on dockerhub[10] and is reusable provided that the credentials for an existing Openwhisk installation are renewed.Jmeter clients have been created that invoke both actions (OpenWhisk sequence with delay input parameter and Node-RED action with delay and delay loops parameters). Loops to implement multiple parameter values are applied in the Jmeter files to measure all the needed combinations automatically.

5 Obtained Measurements and Analysis

5.1 *Initial Data Collection*

Initially, a set of measurements were performed for 1 to 25 functions, with a step of 5, and a set delay for 1000 milliseconds for each function of the sequence. Each execution was performed 40 times and the average of each case was extracted. Total response times of the three modes appear in Figure 4. For one warm function execution the execution time is approximately the same for the two modes. However, the more functions that are being included and executed (as a sequence) the more the average tends to be lower for the NR-NR case.

Given that the inner delay of each function in the sequence is the same, the main source of differentiation is the inter-function communication delay. The averaged per function orchestration delay appears in Figure 5. For example in the NR-NR mode case, on 15 Functions we get 15148 ms average execution time. From this value we extract the static delay of each function (15×1000 milliseconds) and the remaining part is divided by the number of functions used in the specific case (15 functions). This indicates the average time spent between function calls in each mode, as well as any initialization delays in the first call. The fact that Node-RED ifc times per function appear higher in lower function numbers and then start to get lower can be attributed to the fact that any initialization times (not cold starts, since all the executions are warm, but any delays in argument passing, reuse of warm container delays etc.), are divided between the number of functions. Thus, when function numbers are low, the effect of this initial delay is higher and diminishes as the number of functions grows, since it is averaged on them.

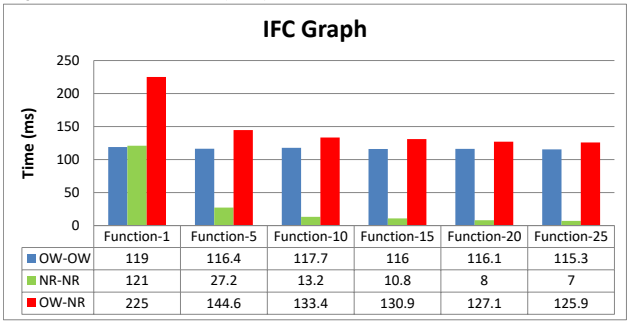


	1	5	10	15	20	25
OW-OW	1119	5582	11177	16740	22321	27882
NR-NR	1121	5136	10132	15162	20149	25153
OW-NR	1225	5723	11334	16964	22541	28147

Figure 4: Total Response Times of the different modes for varying number of functions in the sequence and a static delay of 1000 milliseconds for each function delay

Finding 1: From Figure 5, it is evident that we can use the average overhead per function time for the OW case, given that this is independent of the number of functions used. However, for the NR-NR and OW-NR cases, the initialization time significantly affects the average produced, as the number of functions grows.

Thus to have a more accurate approach, we need to estimate the ifc time (T_{ifc}) from the acquired measurements and split it from any initialization time (T_{init}).



	Function-1	Function-5	Function-10	Function-15	Function-20	Function-25
OW-OW	119	116.4	117.7	116	116.1	115.3
NR-NR	121	27.2	13.2	10.8	8	7
OW-NR	225	144.6	133.4	130.9	127.1	125.9

Figure 5: Average orchestration delay per function sequence size

5.2 *Model Estimation of the parameters for T_{init} and T_{ifc}*

The Pearson correlation coefficient for the dataset is 0.9105, indicating a strong linear relationship. Thus we can use a simple line to detect the dependency of the total flow delay time from the

[7] https://hub.docker.com/r/pekoto/noderedaction

[8] https://flows.nodered.org/flow/f0795ad9f25ad2affcadb8deb305fdf3/in/VOf-0UrN5e2j

[9] https://flows.nodered.org/node/node-red-node-openwhisk

[10] https://hub.docker.com/r/pekoto/owmode3

initialization times, intermediate function delays and inter function communication, defined as follows:

$T_{total}=T_{init}+n\times(T_{functiondelay}+T_{ifc})$

where:

- T_{init} is any initial environment or other initialization time happening once
- $T_{functiondelay}$ = preset sleep delay of each function in the sequence,
- n = number of functions (in sequence)
- T_{ifc} = inter-function communication, the delay needed to go from one function to the next. It is kept as n and not (n-1) that are the number of links in an n-sequence since we consider that in this way the T_{ifc} can model any repetitive actions needed in each step, including the first step. These, for example, can be warm container reuse times for the OW-OW and OW-NR cases that are needed in each function invocation. In the case of NR-NR it would include only the time needed in the Node-RED runtime to pass from one function to the next

From the experiments, the total delays T_{total} are known, the delay of each function is set and known as well as the n used in each case. Given that we have measurements for different values of n we can apply a simple regression to estimate the parameters T_{init} and T_{ifc}. The previous equation can be transformed to:

$T_{total}-n\times T_{functiondelay}=T_{init}+n\times T_{ifc}$ or $Y=a\times x+b$

where x is the number of functions in the sequence and the coefficients are the T_{ifc} (a) and T_{init} (b). The curve fit can be optimized via a typical method such as the ordinary least squares ols function[11] of GNU Octave, giving the values of 128.36 for T_{init} and 1.69 for the T_{ifc} (in the millisecond range) for the NR-NR case.

In a similar fashion, for the OW-OW mode, we have a reverse effect, the T_{init} is set at 2.11 and the T_{ifc} at 118.84. The latter was expected since the individual values are very similar and close to the average value of the ifc graph and indicates the fact that we need increased initialization in each step.

Finding 2: In the OW-OW case, the similarity between the initialization time of the model and the time needed for 1 function indicates that the majority of the delay refers to the need to set up a (warm) container for the next function in the sequence.

For the OW-NR case, we have a similarly large initialization time to the NR-NR case for the orchestration logic in the area of 114 milliseconds, while due to the need for external invocations to the OW environment for each action execution we need an extra 121 milliseconds per function step.

The above times can be off-set by the estimated difference in the cold case. For the NR-NR case the difference between a cold and a warm start was measured at 7.373 seconds, and for the OW-OW case at 2.248 seconds. These times can be added as penalties in the final function, appearing in Table 2.

Table 2: Estimated Parameters for T_{ifc} and T_{init}

Mode	Warm Function Sequence Execution (ms)	Cold Penalty (ms)
OW-OW	$T_{total}=2.11+n\times(T_{functiondelay}+118.94)$	+2248
NR-NR	$T_{total}=128.36+n\times(T_{functiondelay}+1.69)$	+7373
OW-NR	$T_{total}=114.08+n\times(T_{functiondelay}+121.70)$	+9621

5.3 *Extrapolation of model estimation*

From the aforementioned functions we can easily create parameterized plots for different function sequences and inner function delays, to observe how the estimated total execution differs for different function numbers and delays (Figure 6). Indicatively, in some cases (e.g., OW-OW with 100 ms inner function delay) the orchestration delay per function is higher than the actual computation needed inside the function (100 ms), leading to overheads being over 50% of the total time. In the case of OW-NR, the behavior is in most cases dominated by the large inter-function communication time due to the need to trigger the external OW action for the execution. Since this is very similar to the action triggering in the OW-OW case, the graphs of the two modes significantly overlap for similar inner function delays. The orchestration overheads, as estimated by the functions in Table 2, are extracted and presented in Figure 7, by extracting the $n\times T_{functiondelay}$ from the function. In this case the observed overhead does not depend on the inner delay of each function, so we can better observe the pure total orchestration overhead.

Finding 3: The borderline value of 40 functions for the NR-NR mode in the cold start case is considerably high given the current landscape of FaaS applications[2]. On the other hand, in warm executions the NR-NR mode is always better.

However this low function usage per application might also be limited by the capabilities of the available design and development environments of current FaaS platforms. For the OW-NR case, the extrapolated boundary is around 60+ functions in the warm case.

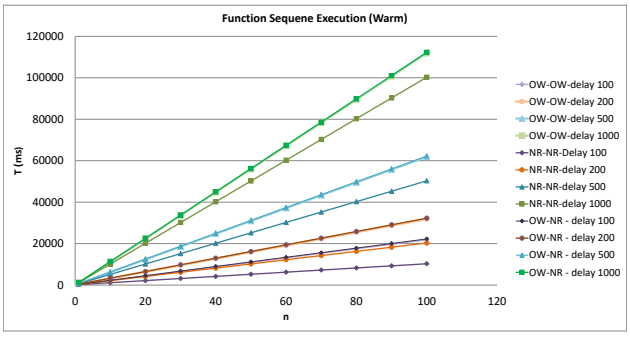


Figure 6: Usage of the estimated function to predict total time for different sequences of functions and inner function delays without the cold start penalty

[11] https://octave.sourceforge.io/octave/function/ols.html

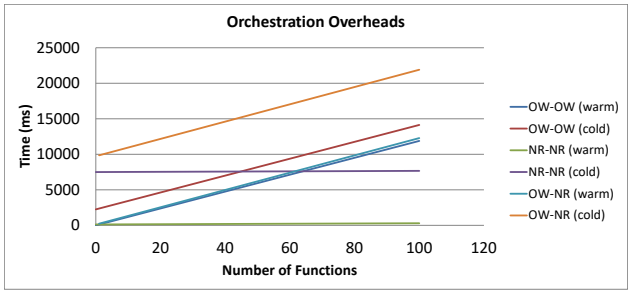


Figure 7: Estimated Orchestration overheads in absolute times after subtracting the $n \times T_{functiondelay}$ part from T_{total}

5.4 *Model Validation*

The created model was based on executions of 1000 millisecond functions from 1-25 functions with a step of 5. In order to validate whether it has the ability to be used for cases that have not been measured, we created a validation set consisting of extra executions for numbers of functions with inner delays of 1000 milliseconds ranging from 30-50 with a step of 5, as well as functions with inner delays of 100 and 200 milliseconds for the whole sequence size (1-50). The maximum value of 50 functions was selected since this is the maximum supported by the Openwhisk Sequence Operator. Each execution was performed for 40 times and the average of the response times was compared to the predicted one from the models of Table 2.

The results of the Percentage error of the estimation of the orchestration times per validation point and mode appear in Figure 8. In this calculation, we have extracted the known delays ($n \times T_{functiondelay}$) so that the percentage of error is calculated on the actual orchestration time, without the size of the artificial delay affecting it. Accumulatively, OW-OW mode has a Mean Absolute Percentage Error (MAPE) of 8.4%, NR-NR of 19.88% while OW-NR an error of 6.2%.

Finding 4: A simple linear model can be used in this case, given the absence of more complex parameters like interference. The model tends to overestimate the orchestration delays, a factor leading to safer executions (from not violating QoS constraints point of view). This overestimation happens primarily for the NR-NR mode which has the minimum absolute overheads. Thus a small deviation in the model results in a large percentage error.

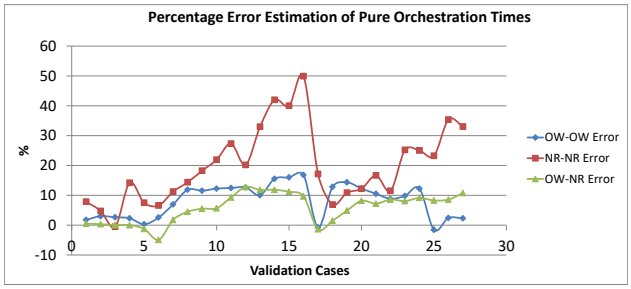


Figure 8: Percentage Error of the orchestration part prediction in validation points for the model accuracy

Another way of representation, related also to the function execution time, appears in Figure 9, by utilizing the functions of Table 2. In this case, the orchestration overhead (T_{total}-$n \times T_{functiondelay}$) is presented as a percentage of the actual useful function execution time ($n \times T_{functiondelay}$).

Finding 5: In many cases, the overheads may reach up to 250% of the useful computational time (in the case of OW-NR and small function delays). In most cases of the OW-OW mode, the percentage is higher than 100% (in lower function delays) while a typical range of orchestration overheads is between 10 and 20% for larger function delays. This would be even worse in cases where the executions were not only warm ones. The NR-NR mode presents the greatest benefits, having under 10% from as low as 10 functions in the sequence and even for small function delays of 100 and 200 milliseconds, with a minimum of 0.29% for 100 functions in the 1000 millisecond case.

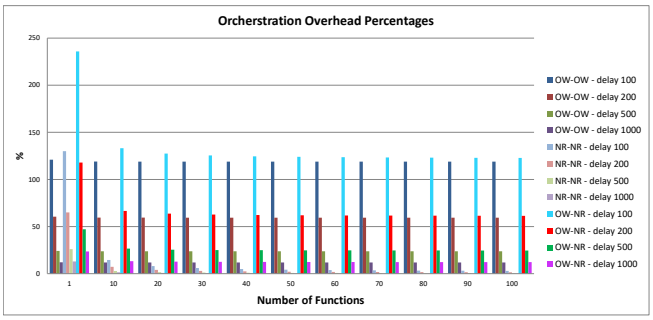


Figure 9: Estimated orchestration overheads in warm executions as percentage of the useful computational time of the workflow ($n \times T_{functiondelay}$)

All the data from the experiments have been made available[12], along with the Jmeter clients.

6 Conclusions

As a conclusion, orchestration delays, primarily with relation to the time needed for coordinating execution and passing outputs from one function to the next, can become a large overhead for the execution of large function workflows. At many investigated points, and for specific orchestration mechanisms, this overhead may even be higher, as a percentage, from the actual execution time of the functions, in case of linking together small pieces of code. A number of take-aways can be extracted from this work:

-OW orchestration time is primarily due to the warm container reuse time. In other cases (cold or prewarm executions) this would be even higher. This delay is unavoidable since in this mode we are not able to implement both orchestration and function logic in the executing container.

-The proposed orchestration through Node-RED enables the combination of orchestration logic and function execution in the same container. This aids in minimizing the needed containers, the setup of which is the largest part of the orchestration delay.

-Orchestration to external functions (OW-NR) should only be used in cases of large parallelization needs, such as a Fork Join pattern. In all other cases, it assembles the worst features of both approaches, performance wise.

[12] Data and Load files are available at: https://github.com/pekoto4349/measurements

-Baseline times, although examined at simple function chains, can be used in simulations of more complex workflow structures for the modes that support them (NR-NR and OW-NR), since they represent the time needed for completing a function hop. Consideration for other phenomena, e.g. number of cold starts, should be incorporated in such an analysis.

-Enabling easier orchestration, both functionally and performance-wise, can help increase the observed typical number of functions (as reported by [2]) and lead to more sophisticated FaaS workflows. Using combinatorial orchestrator and execution environments can aid in minimizing the significant orchestration delays of such workflows.

In order to enhance reproducibility, all relevant artefacts (docker images, delay flows, jmeter clients and final output data) have been made available as indicated in each section. For the future, the investigation of the orchestration overheads will be extended to take under consideration diverse traffic conditions. Moreover, embedding intelligence in the combined environment itself could help determine in which cases the orchestration should adapt to different calling modes.

ACKNOWLEDGMENTS

The research leading to the results presented in this paper has received funding from the European Union's Project H2020 PHYSICS (GA 101017047).

REFERENCES

[1] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha, “A preliminaryreview of enterprise serverless cloud computing (function-as-a-service)platforms,” in2017 IEEE International Conference on Cloud ComputingTechnology and Science (CloudCom), pp. 162–169, IEEE, 2017

[2] Eismann, S., Scheuner, J., Van Eyk, E., Schwinger, M., Grohmann, J., Herbst, N., Abad, C. and Iosup, A., 2021. The State of Serverless Applications: Collection, Characterization, and Community Consensus. IEEE Transactions on Software Engineering.

[3] Kousiouris, G. and Kyriazis, D., 2021. Functionalities, Challenges and Enablers for a Generalized FaaS based Architecture as the Realizer of Cloud/Edge Continuum Interplay. In CLOSER (pp. 199-206).

[4] C. Abad, I. T. Foster, N. Herbst, and A. Iosup, “Serverless computing (dagstuhl seminar 21201),” inDagstuhl Reports, vol. 11, SchlossDagstuhl-Leibniz-Zentrum f`ur Informatik, 2021.

[5] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, “A mixed-methodempirical study of function-as-a-service software development in industrial practice,”Journal of Systems and Software, vol. 149, pp. 340–359,2019.

[6] F. Amato and F. Moscato, “Exploiting cloud and workflow patterns forthe analysis of composite cloud services,”Future Generation ComputerSystems, vol. 67, pp. 255–265, 2017.

[7] I. Baldini, P. Castro, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah and P. Suter, “Cloud-native, event-based programming for mobile applications,” in Proc. of the International Conference on Mobile Software Engineering and Systems, pp. 287–288,2016.

[8] Barcelona-Pons, P. Garcıa-Lopez, ́A. Ruiz, A. Gomez-Gomez,G. Parıs, and M. Sanchez-Artigas, “Faas orchestration of parallel workloads,” in Proc. of the 5th International Workshop on Serverless Computing, pp. 25–30, 2019..

[9] E. Bisong, “Kubeflow and kubeflow pipelines,” in Building Machine Learning and Deep Learning Models on Google Cloud Platform, pp. 671–685, Springer, 2019.

[10] A. Arjona, P. G. Lopez, J. Sampe, A. Slominski, and L. Villard, “Trig-gerflow: Trigger-based orchestration of serverless workflows,”Future Generation Computer Systems, 2021.

[11] Erwin van Eyk, Alexandru Iosup, Cristina L. Abad, Johannes Grohmann, and Simon Eismann. 2018. A SPEC RG Cloud Group's Vision on the Performance Challenges of FaaS Cloud Architectures. In Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18). Association for Computing Machinery, New York, NY, USA, 21–24. DOI:https://doi.org/10.1145/3185768.3186308

[12] López, P.G., Sánchez-Artigas, M., París, G., Pons, D.B., Ollobarren, Á.R. and Pinto, D.A., 2018, December. Comparison of faas orchestration systems. In 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion) (pp. 148-153). IEEE.

[13] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2021. Speedo: Fast dispatch and orchestration of serverless workflows. Proceedings of the ACM Symposium on Cloud Computing. Association for Computing Machinery, New York, NY, USA, 585–599. DOI:https://doi.org/10.1145/3472883.3486982.

[14] Malawski, Maciej, et al. "Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions." Future Generation Computer Systems 110 (2020): 502-514.

[15] Ristov, Sasko, Stefan Pedratscher, and Thomas Fahringer. "AFCL: An abstract function choreography language for serverless workflow specification." Future Generation Computer Systems 114 (2021): 368-382..

[16] John, Aji, et al. "SWEEP: accelerating scientific research through scalable serverless workflows." Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion. 2019..

[17] Zhang, Haoran, et al. "Fault-tolerant and transactional stateful serverless workflows." 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI20), 2020.

[18] Burckhardt, Sebastian, et al. "Serverless workflows with durable functions and netherite." arXiv preprint arXiv:2103.00033 (2021

[19] Kousiouris, G., Ambroziak, S., Costantino, D., Tsarsitalidis, S., Boutas, E., Mamelli, A. and Stamati, T., 2022. Combining Node-RED and Openwhisk for Pattern-based Development and Execution of Complex FaaS Workflows. arXiv preprint arXiv:2202.09683..