Using Application Knowledge to Reduce Cold Starts in FaaS Services

David Bermbach, Ahmet-Serdar Karakaya, Simon Buchholz
TU Berlin & Einstein Center Digital Future
Mobile Cloud Computing Research Group
Berlin, Germany
{db,ak,sibu}@mcc.tu-berlin.de

ABSTRACT

In Function-as-a-Service platforms (FaaS), which have become very popular lately, code is deployed in the unit of single functions and the cloud provider handles resource management. There, a key problem is the so-called cold start problem: when a request comes in and no idle container can be found for the execution of the target function, then a new container needs to provisioned. In that case, the request incurs an extra latency – the cold start latency.

Recent work has largely focused on reducing the *duration* of cold starts. In this paper, we present three approaches, complementary to related work, that reduce the *number* of cold starts while treating the FaaS service as a black box. In the approaches, implemented as part of a lightweight choreography middleware, we use knowledge on the composition of functions to trigger cold starts and, thus, the provisioning of new containers before the application process invokes the respective function. In experiments on AWS Lambda and OpenWhisk, we could show that our approaches remove an average of about 40% (in some cases up to 80%) of all cold starts while causing only a small cost overhead.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Cloud computing**; *Middleware*;

KEYWORDS

Serverless Computing, Function-as-a-Service, Cold Starts

ACM Reference Format:
David Bermbach, Ahmet-Serdar Karakaya, Simon Buchholz. 2020. Using Application Knowledge to Reduce Cold Starts in FaaS Services. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3341105.3373909

1 INTRODUCTION

The latest trend in cloud computing are so-called Function-as-a-Service (FaaS) offerings often also referred to as "serverless". In FaaS offerings such as AWS Lambda[1], Google Cloud Functions[2], or OpenWhisk[3], developers deploy code in the granularity of single functions. The cloud platform then automatically handles resource allocation depending on the number of concurrent requests. Based on this, FaaS has received a lot of attention, e.g., [2, 3, 7, 10, 12, 14, 16], as it can be seen as a major cloud evolution step – from a developer perspective, the cloud is moving from pay-per-provisioned-resource to a true pay-per-use model.

On a system level, FaaS platforms typically rely on container technology[4]: When developers deploy their function, this function code is bundled in a container and stored in a container repository. When a request comes in, a gateway component checks whether there is already an idle container instance that could serve the request. When there is no idle container, the gateway allocates a new one and directs the request to the respective machine. The time between identifying that there is no idle container and the container serving the request has been referred to as cold start latency or cold start time [2, 12, 14]. In many cases, the additional cold start latency effectively doubles the latency of function execution from a client perspective. Cold starts can occur when an existing container was unprovisioned due to a period of idleness or at the provider's discretion; usually, however, they happen (in significant numbers) for growing workloads.

Researchers have tried to address this problem, which has also been called the cold start problem, through a number of approaches, e.g., [14, 15, 17]. Most of these approaches have in common that they focus on single functions or try to increase the container initialization speed. In practice, however, functions are rarely deployed in isolation but are rather part of some kind of "business process". This process may be implicit, e.g., when reacting to events from other services, or explicit which can be seen in the availability of orchestration services such as OpenWhisk's Composer[5] or AWS Step Functions[6].

In this paper, we demonstrate how to use process knowledge to reduce the number of cold starts in a FaaS platform from a developer perspective, i.e., treating the FaaS platform as a black box. For this purpose, we make the following contributions:

(1) We propose a lightweight choreography middleware that can be deployed along with the functions and which avoids centralized orchestration components.

SAC '20, March 30-April 3, 2020, Brno, Czech Republic

ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/10.1145/3341105.3373909

[1] aws.amazon.com/lambda
[2] cloud.google.com/functions
[3] openwhisk.apache.org
[4] Our approaches do not assume the use of containers; for ease of writing, however, we will in the following pretend that FaaS platforms are container-based.
[5] github.com/ibm-functions/composer
[6] aws.amazon.com/step-functions

(2) We develop and discuss three approaches for reducing the number of cold starts based on hinting.

This paper is structured as follows: We start with an analysis of the state of practice in current FaaS platforms that offer function composition features (section 2) before describing our choreography approach (section 3). Afterwards, we discuss the basic hinting mechanism that we use to avoid cold starts during the execution of processes and derive three approaches which determine when and how to send hint messages (section 4). Next, we present the results of a large number of experiments which demonstrate that our approaches can avoid up to 80% of all cold starts at low monetary cost (section 5). Finally, we discuss the limitations of our approaches (section 6) and related work (section 7) before concluding this paper (section 8).

2 STATE OF PRACTICE IN USING PROCESS KNOWLEDGE TO AVOID COLD STARTS

The basic idea of our approach is to use the knowledge on process composition to reduce the number of cold starts that are encountered during the execution of our functions: When the developer deploys a composition of functions [A,B,C,D] rather than individual functions A, B, C, and D, then FaaS platform providers could theoretically know that functions B, C, and D will be invoked once a request arrives for function A. They could even provide estimates for the point in time when this will happen.

Our intended approach goes beyond this in two regards: First, we aim to enable the application developer to control the number of cold starts since a FaaS platform provider might decide not to treat all applications in the same way. Second, per definition a solution offered by the platform provider will be single platform. There are, however, scenarios where processes will span multiple platforms, e.g., when trying to avoid the "data-shipping architecture" criticized by Hellerstein et al. [8] or for deployments across cloud and edge [3, 16].

Regarding the state of practice in existing platforms, we analyzed two major platforms: IBM Cloud Functions, which runs OpenWhisk (in the following, we will refer to this exclusively as OpenWhisk), and AWS Lambda. For OpenWhisk, we learned in personal conversation with one of the lead developers that they had had a similar idea almost two years ago but that it had not been added to OpenWhisk Composer yet. For AWS Step Functions, the orchestration service for Lambda, we ran a simple experiment in which we deployed two different Step Functions descriptions. One contained a sequence of four code-wise identical functions, the other contained only one of these. We then ran Apache JMeter[7] to create load on these functions. For both deployments, we sent droves of ten concurrent requests every five seconds for five times.

The intuition behind this was that the first drove will always cause ten cold starts (so that we could measure cold start latency) while the following four should all meet warm and idle instances (so that we had a base value for comparison). If Amazon were using the process knowledge, then the first drove of the four-function deployment should only have a single cold start; if not, it should accrue four cold starts. The experiment consistently showed that Step Functions does *not* use the process knowledge.

[7] jmeter.apache.org

To conclude, implementing an application-side solution that treats the FaaS platform as a black box is the only option that (i) will work with multi-platform deployments, (ii) avoids discriminatory treatment by providers, and (iii) allows to retrofit cold start management to widely used FaaS services where the provider is not using the process knowledge.

3 FUNCTION COMPOSITION

While there are existing function composition offerings by individual providers, there are to our knowledge no cross-platform composition solutions yet. Developing one is, obviously, not a major contribution but is necessary for our main contributions in section 4.

As known from the service composition world, there are two fundamental approaches to composition: orchestration (a dedicated orchestrator component sequentially invokes all steps of the process) and choreography (each step invokes the next step upon completion). Since we wanted to avoid the problem of double billing as described by Baldini et al. [2], orchestration could only be achieved through "client-side scheduling". This strategy, however, comes with a number of disadvantages. Besides the extra machine necessary to invoke the functions, it is unclear where this machine should be deployed in a multi-cloud or even fog deployment which would either mean increased latency or running multiple client machines that need to coordinate the hand-over of process instances. Hence, we already tended towards choreography which is also more in line with the microservice paradigm which in turn corresponds nicely to FaaS. Finally, we decided that the substitution principle of the serverless trilemma [2] is the least important for our purposes as our focus is not on being able to build compositions of compositions but rather to have a simple composition middleware that is co-deployed with the functions of the process.

To get the desired multi-platform capabilities, we restricted our choreography middleware to node.js and built it on top of the serverless framework[8] which among others supports AWS Lambda and OpenWhisk out of the box and is often used for FaaS deployments in practice. To use our choreography middleware, developers have to add a workflow description (containing concise information on the process steps and their sequential or parallel ordering) and a small JavaScript library to their deployment. Beyond this, they need to make a small change to their function code in that they paste some JavaScript glue code into the file and modify the header of the original function so that it becomes a *step handler* instead. This asserts that the FaaS platform, instead of calling the original function directly, invokes the "glue code" which then delegates execution to the step handler. In the glue code, our middleware checks whether a workflow state has been passed (otherwise it simply invokes the step handler and returns its result), triggers our cold start avoidance mechanisms (we will describe those in section 4), and invokes the next function of the process.

4 MANAGING COLD STARTS IN FAAS

In this section, we will present our three approaches for reducing the number of cold starts from a client perspective. We refer to these approaches as the naive, the extended, and the global approach.

[8] serverless.com

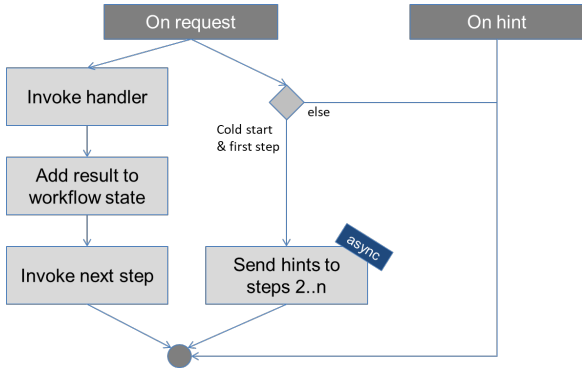

Figure 1: Naive Approach: Control Flow in the Middleware

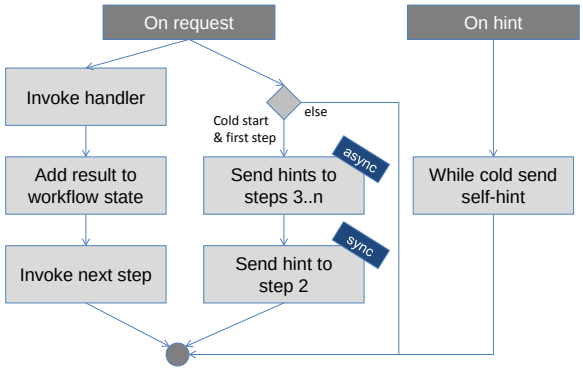

Figure 2: Extended Approach: Control Flow in the Middleware

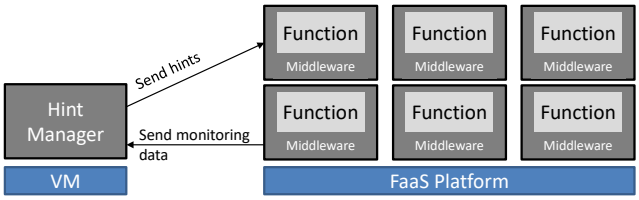

Figure 3: Deployed Architecture of the Global Approach

The naive and the extended approach can be implemented as part of the FaaS-deployed middleware presented in section 3, the global approach uses additional components[9].

4.1 The Naive Approach

The intuition behind the naive approach is that a process which encounters a cold start in its first step is (under some assumptions) very likely to encounter cold starts for the other steps as well. This is particularly true for processes that are only triggered from time to time so that all containers have been unprovisioned but will also happen during phases of increasing request rates. The idea now is to notify all other steps whenever the first step encounters a cold start so that an additional container can be provisioned for these steps. Since FaaS platforms currently provide no interfaces for applications to do this directly, this can only be achieved by invoking all other steps (asynchronously), i.e., sending them a hint; each step in turn just terminates right away whenever it receives a hint message. This approach can easily be implemented as part of the choreography middleware from section 3. See figure 1 for an overview of the approach.

Issues: While the naive approach is indeed very simple to implement, it suffers from a number of issues that make its use limited beyond some special cases. The main problems are *hint misses* and *process overtaking*. Hint misses mainly come from the fact that the approach has the implicit assumption that all steps will have the same utilization. Hence, a high percentage of hint messages will actually be processed by idle, warm instances instead of provoking the desired out-of-process cold start. Process overtaking, on the other hand, happens whenever the cold start time of step 2 exceeds the processing time of step 1. In that case, the hint message might trigger a cold start for step 2 which will still be ongoing when step 1 terminates, invokes step 2, and is then likely to cause another cold start. In rare cases, hint misses and workflow overtaking together might even lead to additional cold starts: whenever a hint message blocks an otherwise idle instance which is at that precise moment invoked by the regular process execution. Of course, the naive approach can also not avoid cold starts in the first process step. It should, however, be noted that neither of our approaches will ever trigger an unnecessary instance start as hints are only sent when there is an observed lack of instances.

4.2 The extended Approach

With the extended approach, we specifically aim to address hint misses and process overtaking; it is more or less an improved version of the naive approach. Process overtaking is most likely to happen in step 2 of a process: in later steps, it can only happen whenever all previous steps aside from the first one did *not* encounter a cold start. Therefore, the simplest approach to address process overtaking is to use the naive approach but to send the hint to step 2 "synchronously", or more precisely: to send it asynchronously but to wait for the result before terminating the function. This way, process execution will wait until the (potential) cold start in step 2 has been completed.

Furthermore, to address the issue of hint misses, we propose a mechanism which we refer to as recursive hinting: upon receipt of a hint message, the function checks whether it had a cold start or not. In case of a hint miss, the function sends a hint message to itself to "enforce" a cold start at some point.

In recursive hinting, we have identified two tuning mechanisms (for readability reasons not shown in figure 2). The first is the maximum recursive depth which puts an upper limit on the number of recursive hints which can easily be implemented through a counter. This aspect is crucial as a safety mechanism, especially in combination with the second tuning mechanism.

The second mechanism describes how to send the self-hints. There are basically three different ways: asynchronous, synchronous, or synchronous with time-out. Asynchronous will make the

[9]Our source code is available at https://github.com/CloudFunctionChoreography

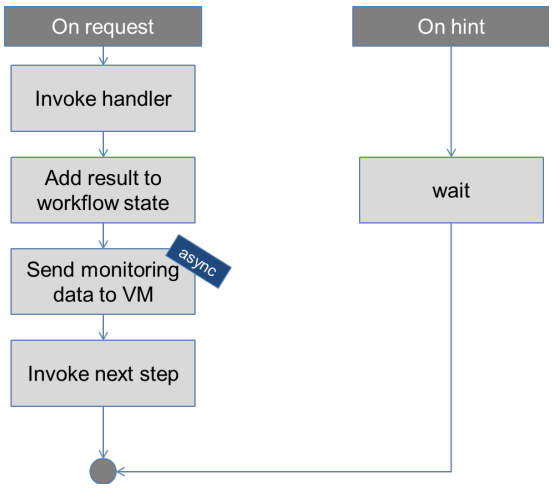

Figure 4: Global Approach: Control Flow in the Middleware

original function instance available to real requests as fast as possible. This, however, means that it needs to be combined with a short "sleep" period to avoid situations in which a self-hint not only invokes the same function but also the same function instance. Here, the maximum recursive depth limit is crucial to avoid infinite recursion. Synchronous, in contrast, means that the function instance will block until a cold start has been reached. It will, however, also block during the cold start which essentially means to block (almost) all warm instances when provisioning a new instance until that instance is available. We propose to use synchronous with time-outs as a middleground solution. Here, the self-hint should only block for the duration necessary to actually make one or two self-hint calls[10]. This asserts that the function instances become available again before the cold start of the new function instance completes but provides poorer guarantees regarding the actual enforcement of a cold start. We believe that this is the best configuration for this approach. Figure 2 shows an overview of the approach, especially in comparison to figure 1.

Issues: While the extended approach addresses the two core issues of the naive approach, there are still a number of problems. Arguably, the main problem remainining is that both approaches need to encounter a cold start first to avoid other cold starts, i.e., their main use is to avoid situations where the cold start durations add up over the course of a longer process execution. Furthermore, both approaches assume that the execution duration of different steps in a process is comparable which also means that each step needs approximately the same number of instances to process a constant workload. In an extreme example, where one step takes 1ms to complete and the next one takes 1s to complete, this is simply not true: the slower function needs approximately 1000 times more function instances for the same process workload. If the first step is the slowest, this "only" causes overprovisioning (i.e., cost) in the later steps. If the first step is the fastest, neither approach will have much effect on the number of cold starts.

[10]In our prototype, we configured this blocked time as the one way latency for invoking a function (determined through experiments) times the number of remaining recursive hints as defined by the self-hint counter.

4.3 The Global Approach

While at least part of the issues above could be addressed through incremental changes to the extended approach, we decided to develop the global approach which tries to tackle all of them through a fundamentally different mechanism. Both other approaches suffer from a lack of knowledge on the global system state but in exchange are very lightweight and are simply co-deployed with the individual functions (we will later discuss advantages and disadvantages of all three approaches in detail). For the global approach, we decided to introduce the hint manager as a central component that collects monitoring data from the process executions and runs simulations based on queuing theory to determine the best point in time when to send hints. See figure 3 for a high-level overview of components, their interaction, and their deployment.

Following Kendall's notation, a process of FaaS functions can be described as a sequence of A/S/c systems where especially c – the number of containers executing a specific function – is highly variable. For our approach, we made the assumption that both the arrival rate of new workflows as well as the execution duration of all functions follow an exponential distribution which means that each process step can be modeled as an M/M/c system. We also assume that the system is in a steady state as this significantly simplifies our simulation.

Please, note: We are aware that an exponential distribution is not necessarily correct for the execution duration. We are also aware that the steady state assumption is definitely *in*correct as it would imply that the ideal number of containers were already provisioned. Finally, we are also aware that the exponential arrival rate will in most scenarios, e.g., in case of increasing load, be incorrect as well. We nevertheless decided to use these assumptions as they significantly simplify our approach and leave the fine-tuning to future work. However, we carefully asserted that the experiments in section 5 use realistic use cases and indeed explicitly violate all but the assumption of having an exponentially distributed execution duration. This was done to demonstrate how well the approach works under adverse conditions.

The Middleware: As already indicated above, the global approach shifts the complexity of deciding when to send which hint to an external hint manager component running on a VM. As such, this significantly reduces the complexity of the functionality part that is co-deployed with the function as part of our choreography middleware. As shown in figure 4, the hinting part is completely removed from the FaaS-deployed middleware. Instead, the middleware collects a number of monitoring metrics (e.g., start and end timestamp of the function execution or whether it was cold start) and sends it to the external hint manager, asynchronously. Upon receipt of a hint message in a warm instance, the function blocks for a short period of time to assert that several hints sent in parallel cannot be served by the same container. We will later see that this is problematic for OpenWhisk's queuing model but works well for AWS Lamda. The middleware is still responsible for choreography, i.e., the hint manager is not involved in the data flow.

The Hint Manager: As already indicated, the hint manager has two core responsibilities. First, it acts as a sink for monitoring data; second, it tries to send hint messages at the optimal point in time. As part of the monitoring task, the hint manager assembles

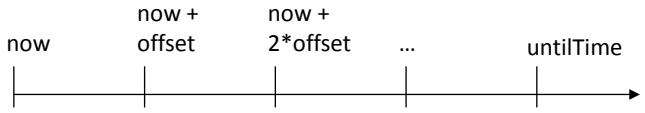

Figure 5: Global Approach: System Snapshots are Simulated for Timestamps every `offset` ms

received data for two sets of core metrics. The first group describes the current infrastructure state, namely the number of idle and busy instances as well as the average processing time for each step (both cold and warm durations). The second group describes the process state, namely the average arrival rate of new process instances and for each process instance the step which is currently being executed. These together provide a comprehensive overview of the system state (the "system snapshot") and allow us to make predictions about future snapshots.

The simulation itself continuously runs in sequential rounds. For a given simulation round, the hint manager determines a forecasting end, the `untilTime`. For this parameter, it selects the point in time when a process starting now would (on average) terminate if it were to encounter a cold start for every single step. While this is configurable it allows us to assume that all currently active processes will have a simulated end as part of one simulation round. Beyond the `untilTime`, we also have an `offset` parameter (default: 20ms) which describes the interval between simulated system snapshots.

Step 1: During a simulation round, the hint manager calculates a system snapshot for every timestamp marked with a line in figure 5. This works by iterating over each process instance (including predicted ones as defined by the observed average process arrival rate) and calculating the respective progress by adding respective (average) durations for (i) invocation, (ii) optional cold start delay, and (iii) either cold or warm execution. This is done based on the steady-state queueing model described above.

Step 2: Considering the already known (and during an iteration updated) number of idle and busy instances per step, this allows us to easily forecast the timestamps of future cold starts. From each of these timestamps, we subtract the average invocation and cold start latency which gives us the latest point in time when a hint message could still successfully trigger a cold start before the respective function instance is actually needed.

Step 3: Next, we exclude all events from this set for which that latest time is more than one second in the future[11] as well as all events for which we already scheduled a hint message in a previous simulation round.

Step 4: Finally, we group all events by function and identify for each group the timestamp between now and the first event where the respective function will have the highest utilization (see example below). For those timestamps, the hint manager schedules the respective hint messages – they are sent asynchronously – and then starts all over with the next simulation round.

In the example shown in figure 6, the first three simulation steps identified that a hint message each needs to be sent at the latest at

[11]The further we go into the future, the less accurate are our predictions; we can, however. easily run several simulation rounds per second and, thus, simply address the event in a later simulation round.

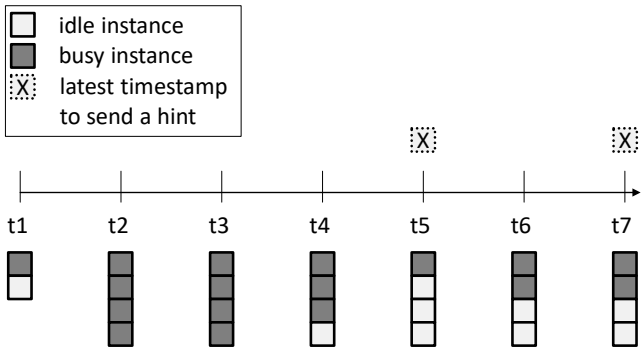

Figure 6: Global Approach: Example for Hint Scheduling

t5 and t7. If we were not grouping hint messages, this would mean to send four parallel hint messages at t5 (three to temporarily block the idle instances, one to trigger the desired cold start) and three at t7 (two to block the idle instances, one to trigger the desired cold start). While sending the hints for both of them together improves efficiency, we cannot send the hint messages later than t5. Out of the timestamps t1 to t5, sending hint messages at t2 or t3 involves the least number of hint messages as no additional hints are needed to block idle instances. In this case, we would send the two hint messages at t2 as sending them a bit early gives us an additional buffer in case the cold start delay is longer than anticipated.

Issues: Besides the added complexity of having to run the hint manager, the main limitation of the global approach is that the process types are limited (see section 6). Otherwise, the approach suffers from the assumptions discussed above and a few minor twitches. For instance, the approach does not account for hint misses or periodic instance unprovisioning. Both mean that the next few simulation rounds will work with slightly incorrect data before the deviation is detected through monitoring. The simulation itself is not very computationally intensive: for n parallel process instances with m function steps each, a simulation round roughly corresponds to adding $n * m$ integers with small values for m. Hence, scaling the hint manager up should be able to manage quite high numbers for n. Since the hint manager can also easily be sharded by process type, we believe that the hint manager is not a scalability problem. In cases where scale-up is no longer possible due to extremely high load, running a self-hosted OpenWhisk – adapted to consider process information for cold starts – will by far be the more efficient option.

5 EVALUATION

The evaluation of our three approaches is twofold. First, our proof-of-concept prototype already described in earlier sections shows that the approaches can indeed be implemented and deployed. Second, we ran a large number of experiments to assess the effectiveness, i.e., impact on the number of cold starts, and the efficiency, i.e., cost overheads, of all three approaches. In this section, we describe these experiments in detail.

5.1 Experiment Design and Setup

Our experiment parameters comprised three deployment options, two different process types, and two different workloads. Beyond experiments with our three approaches, we also needed to compare the results to a baseline value that used our choreography middleware but did not leverage any of our three approaches[12]. In total, this means 48 different setups which we each repeated four times. Furthermore, we randomly selected one setup combination and repeated it twenty times for each of the three approaches as well as the baseline configuration to assert stability of results.

Process Types: We used two different process types, the homogeneous and the heterogeneous, with 16 steps per process. Both used an exponential distribution of processing durations but had different mean values respectively. Since we had noticed in earlier experiments that the processing duration on OpenWhisk and AWS Lambda was highly variable and even showed temporal patterns, we implemented the code of each function as a simple sleep command to assert reproducibility of results. Please, note that the functions did not all sleep for the same duration but instead used a random value with preconfigured mean and variance. The homogeneous process had a mean value of 500ms for all process steps. The heterogeneous process started with a mean of 500ms for the first step but then increased by 50ms for every even step and decreased by 50ms for every odd step. This means that the sequence of mean values was 500, 550, 450, 600, 400, ..., 150, 900. Following this pattern, we could assert that our experiments included both situations where the first step is the longest and the shortest step. Please, note, that neither of our approaches is affected by the execution duration of functions aside from being able to monitor it (global approach).

Deployment Options: We deployed our example processes in three different setups. First, on AWS Lambda only. Second, on OpenWhisk only (we used IBM Cloud Functions). Third, in a federated setup where the first four steps run on OpenWhisk, the second on Lambda, the third again on OpenWhisk, and the final four again on Lambda.

Workloads: We ran two different workloads, the growth and the burst workload. Both workloads were implemented using JMeter and had a warm-up, a measurement, and a cool-down phase. For our experiments, we only considered the results of processes started during the measurement phase. We also ran an initial set of experiments where single requests were sent to a cold infrastructure. Since these results were as expected we did not do further experiments with that workload.

In the growth workload, the warm-up phase only deployed the example process and did not issue any requests. After 150s, the first process request was sent, after 650s the last. In between, the process arrival rate linearly increased from 0 to 5 req/s. The cool-down phase ran for another two minutes with a stable process arrival rate of 5 req/s. In each growth experiment, 1300 process instances were executed as part of the measurement phase (1815 in total).

In the burst workload, the warm-up phase had a gradual increase of arriving process requests from 0 to 3 req/s over about four minutes. Next in the measurement phase, the arrival rate suddenly jumped to 8 req/s stayed at that level for about 30s before dropping back down to 3 req/s and proceeding into the cool-down phase. Again, we only considered the 356 process instances that each experiment started as part of the measurement phase (1001 in total).

[12]We decided not to use a mechanism from related work as baseline since these mechanisms are either orthogonal to our approach or are provider-side mechanisms which we could not use.

Table 1: Baseline: Number and Percentage of Cold Step Executions for the Heterogeneous Workload

API Deployment	Burst	Growth
AWS Lambda	306 (5.3%)	93 (0.44%)
OpenWhisk	92 (1.6%)	85 (0.40%)
Federated	127 (2.2%)	70 (0.30%)

5.2 Results: Baseline

In our baseline experiments, we quantified the number and duration of cold step executions. Table 1 shows the absolute numbers and the relative share of cold step executions in case of the heterogeneous workload (as expected homogeneous vs. heterogeneous had little effect on the baseline results). Indeed, the numbers show very different scaling strategies in Lambda and OpenWhisk: While Lambda aggressively provisions new container instances, OpenWhisk queues requests first under the assumption that waiting for an existing instance will often be faster than incurring a cold start. This, however, can also be seen in the cold start latency: OpenWhisk consistently shows higher cold start latency values (ca. 6% in the best case, ca. 30% in the average case, ca. 60% in the worst case) than Lambda as the cold start latency on OpenWhisk typically includes significant amounts of queueing time.

We also found that the cold start latency and the execution duration varied a lot (partly with temporal patterns) whereas the number of cold starts was relatively stable. To assert comparability, we will hence focus on the number of cold starts in the following.

5.3 Results: Effectiveness

To evaluate the effectiveness of our approaches, we compared the number of cold starts in the baseline experiment to the results for our three approaches. As can be seen in figure 7, all approaches have an effect on the number of cold starts. As expected, the naive approach is too simplistic to deal with all scenarios but aside from the burst/heterogeneous/OpenWhisk experiment it nevertheless leads to a reduction of up to 30% of all cold starts.

As also expected, the extended approach works significantly better; across all experiments, it resulted in a reduction of about 40% of all cold starts. Considering the still very simple design, we deem this a very good result. Indeed, the extended approach also performs rather consistently across all configurations.

Finally, the global approach tends to outperform the extended approach in almost all configurations. In the Lambda-only deployments, it achieves reductions around 70-80%. As already hinted at before, though, it currently does not work very well with OpenWhisk (and consequently in the federated deployment). We believe that this is due to OpenWhisk's internal queuing model which simply queues our hints – which after all have been designed to

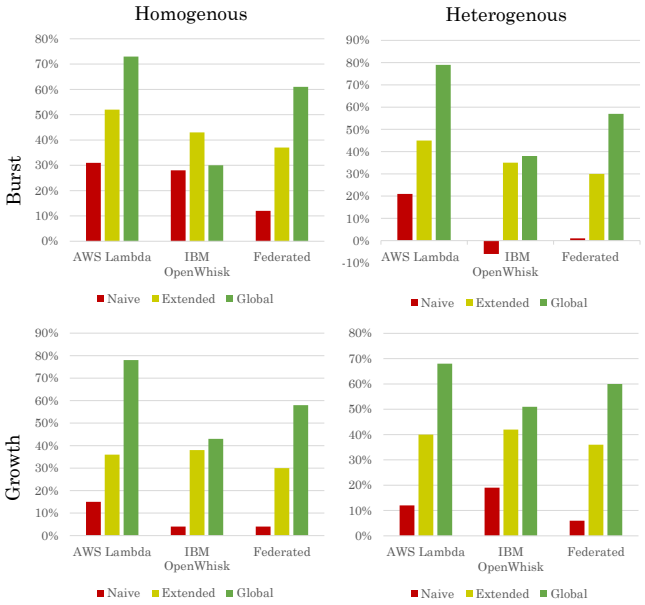

Figure 7: Relative Cold Start Reduction per Function

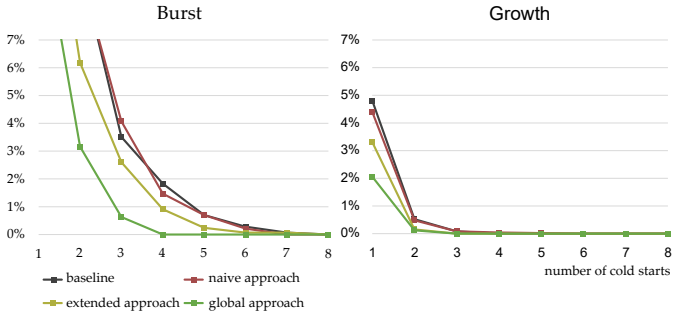

Figure 8: Percentage of Process Instances with n or more Cold Starts

complete fast – instead of provisioning new instances. As such, our hint messages actually lead to additional load on the system where the already full queues then result in cold starts once the actual requests arrive. While the global approach still manages to achieve a reduction of about 30-40% on OpenWhisk, the extended approach is the more efficient solution in that scenario.

So far, we have analyzed the number of cold starts on the level of individual steps. As an alternative analysis, figure 8 shows how the distribution of cold starts changes for the federated/heterogeneous setting (which is a bit of an average case): As already shown in our baseline experiments (table 1), there is a much higher rate of cold starts for the burst workload. This is also reflected in the per-process distribution of cold starts. Under the growth workload, the vast majority of cold starts is a single cold start per process whereas processes under the burst workload are much more likely to experience several cold starts.

As already expected based on figure 7, the naive approach has very little effect on the number of cold starts. Both the extended and the global approach, however, have a significant impact on cold starts and effectively shift the baseline curve from figure 8 to the left. This shows that our approaches are particularly effective in removing the long tail of processes incurring multiple cold starts. For the growth scenario, the lower number of single cold starts when using the global approach indicate that proactively sending hint messages under the assumption of a constant arrival rate is already quite effective. Therefore, we expect that an extension actually forecasting arrival rates, e.g., based on [4], might actually be able to remove a much larger share of cold starts. Regarding the lower percentage of single cold start processes in the growth/extended experiments, it appears that when two processes start in parallel where one encounters a cold start directly, its hint messages may actually avoid the cold starts in the following steps for the respective other process instance.

A good explanation for the effectiveness of our approaches can also be seen in the hint hit ratio, i.e., the number of hints that actually led to a cold start divided by the total number of hint messages sent. Table 2 shows which percentage of the hint messages led to cold starts. For the naive approach, only about 5-10% of all hint messages have the desired effect which explains the poor performance of that approach. The global approach in contrast achieves hit ratios of 40-60% which shows how much can be gained through careful timing of messages. In fact, this number underestimates the effectiveness of the approach as the total number of hint messages also includes hints which have been sent to block idle and warm instances, i.e., which were hint misses on purpose. For the extended approach, we report two numbers: the first number shows the overall hint hit ratio which is not much better than in the naive approach due to the high number of hint messages sent through our recursive hinting. The numbers in brackets show the hit ratio when considering a sequence of recursive hints as "one hint" which imply that recursive hinting leads to the desired cold start in 35-65% of all cases. This shows that recursive hinting – while subject to a number of cost/effect tradeoffs – is a fairly effective mechanism to provoke a cold start. We believe that this is the main cause behind the extended outperforming the naive approach.

5.4 Results: Efficiency

While our approaches help to reduce the number of cold starts, they also increase monetary cost. In this section, we want to give an overview of the costs incurred by using one of our approaches. For this purpose, we use the results from the same experiments as in the last section. Ideally, we would be able to directly quantify the costs of each approach and then compare them. This, however, would not be a fair comparison as we have seen a lot of variance (even including temporal patterns – 2-3x performance variation on OpenWhisk and up to 1.8x on Lambda) in our performance measurements. Furthermore, the vast majority of resource usage is caused by the monitoring instrumentation necessary not for our approach but rather to obtain measurement results. Without these, we would not be able to get any measurement insights. Hence, we use the time measurements from the baseline experiment and

Table 2: Average Hint Hit Ratio

		Burst			Growth		
		Naive	Extended	Global	Naive	Extended	Global
Homogeneous	Lambda	10.47%	10.85% (35.64%)	38.21%	9.82%	10.15% (34.41%)	61.08%
	OpenWhisk	9.32%	16.2% (51.77%)	39.29%	5.90%	23.41% (67.78%)	48.75%
	Federated	10.59%	12.82% (41.67%)	58.05%	4.76%	14.08% (46.92%)	58.26%
Heterogeneous	Lambda	11.63%	11.17% (35.94%)	37.71%	6.22%	9.23% (31.52%)	41.56%
	OpenWhisk	11.61%	13.16% (44.00%)	50.75%	10.46%	21.54% (65.63%)	55.35%
	Federated	7.92%	9.98% (35.00%)	42.12%	6.67%	14.74% (48.44%)	63.09%

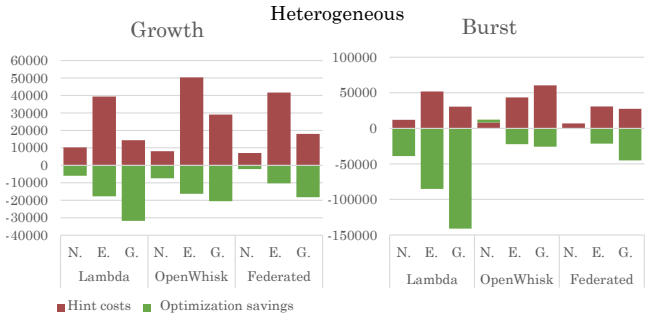

Figure 9: Hinting Costs and Optimization Benefits in ms Execution Duration Compared to the Baseline (Heterogeneous Process)

use the information on the number of cold starts from the other experiments to calculate cost effects based on a cost model.

The main cost driver is the billed duration which approximately corresponds to the execution time of a function. We will use this duration as our cost metric as the monetary cost depends on the respective provider. To quantify the duration, we need to compare hinting costs and the optimization benefits. The hinting costs can be calculated as the average cold execution duration of hints times the number of such executions plus the corresponding number for warm executions ($t^h_w * n^h_w + t^h_c * n^h_c$). Likewise, the optimization benefits can be calculated as the number of avoided cold starts times the average warm execution duration plus the remaining number of cold starts times the average cold execution duration minus the baseline value ($n^f_a * t^f_w + n^f_r * t^f_c - baseline$)[13].

As can be seen in figure 9, which shows the hinting costs and optimization benefits for the heterogeneous process as an example, there is not necessarily a significant cost overhead when using one of our approaches. When considering the execution time as the only cost metric, using the global approach for the Lambda-only setup will actually lead to cost savings. This is due to the fact that the optimization benefits outweigh the hinting costs – as seen in other experiments, it also improves the platform utilization by creating less function instances in total. For all other setups, the cost increase is low – the maximum increase of billed duration across all experiments was 1.3%.

[13]This is a slightly simplified model that does not account for the additional execution duration due to the extended's process overtaking avoidance scheme. This value can be expected to be rather small in comparison to other cost factors.

The billed duration as determined above does not cover all costs associated with our approaches. The remaining aspects, however, are hard to quantify without a concrete application. To give two examples:

First, on AWS there are additional invocation costs (which affect hint-heavy approaches) and costs for the API Gateway[14] which routes the calls to Lambda. API Gateway, however, comes with a one million request free tier, then charges for batches of millions or billions of requests. Depending on the number of requests in the application itself, adding one of our approaches can either put the monthly bill into the next higher category or may not have any effect at all. Quantifying these costs is therefore impossible, but the overall price is also very low (2-3 USD/month).

Second, there are additional costs for the VM running the hint manager. This is again almost fixed cost as the smallest instance type will suffice for rather complex deployments with a lot of load as the simulation is not very computationally intensive. Such an instance is again often part of a free tier and the per-month cost of a scaled hint manager cannot be translated into variable cost without a concrete application.

All in all, we would expect the monetary overhead to be much less than 5% for typical deployments.

In terms of effects on other tenants, hints are only sent when needed and not as keep-alive hints so that there should not be a negative on other tenants. In our experiments, we could see that the overall utilization of function instances remained constant for all setups aside from Lamda-only where the instance utilization increased.

6 DISCUSSION

While we have shown in our evaluation that our three approaches indeed reduce the number of cold starts by up to 80% – and that at a very low overhead, our approach violates all design goals of FaaS platforms: In contrast to other execution platforms, FaaS intends to isolate the application provider from all aspects of resource management. With our approaches, we however have again added an application-side infrastructure management component. In this regard, we believe that the most desirable situation would be when cloud providers would actually use the process knowledge that already exists, e.g., as done in SAND [1]. Aside from the management perspective, this would also be much more efficient as a FaaS provider can simply trigger the initialization of a new container and does not need to rely on the fairly inefficient hinting mechanism.

[14]aws.amazon.com/api-gateway

Even then, however, the knowledge from multi-platform processes could not be used unless FaaS providers start to cooperate (which we deem highly unlikely). All in all, we do not see much of an alternative to an application-side solution at the moment.

In terms of efforts, we would recommend the extended approach which, as a middleware component, can simply be co-deployed with the functions and still removes about 40% of all cold starts. For a Lambda-only deployment, the global approach is also an option if the effort for running the additional hint manager in a fault-tolerant and scalable way is worth it, e.g., for latency-critical workloads with a lot of volatility. In that case, we would recommend to augment the hint manager with workload prediction functionality, e.g., based on [4], instead of assuming a constant process arrival rate. We cannot recommend the naive approach and will ignore it in the following.

Regarding availability and scalability of our approaches, the extended is as available as the FaaS platform and scales along with the function instance. As such, we do not see a problem there. The global approach, however, runs the hint manager which is at the moment a single point of failure. In terms of availability, the state in the hint manager is not important: should the hint manager instance fail, the application will see some more cold starts for a short while but an instance started from scratch should be able to catch-up in less than a minute. Regarding scalability, the hint manager can be scaled up quite far. In fact, we believe that the resource bottleneck will not be the simulation process but rather the VM's bandwidth from receiving monitoring data and sending hints. Here, the hint manager could easily be sharded per process type. If an application achieves such a high process load that the biggest VM type is overloaded when handling a single process type, then the overall costs to operate that application will in all likelihood be so high that operating an open source FaaS system such as OpenWhisk, where one could then implement a more efficient cold start manager, would be the more economic alternative anyhow.

Arguably, an alternative to our approaches would be to fuse all functions into a single function. This, however, is only possible for single-platform processes and when the resulting artifact does not violate platform limits in terms of execution duration and executable size. Preferably, this fusion would also be done automatically to maintain code clarity for developers – we are not aware of any approach in this regard.

The process types supported by our approach are limited to linear workflows and parallel branches (where all branches are executed). Conditional branching can only be supported by sending hints to all branches independent of the actual process execution which may create unnecessary function instances. Our approaches cannot deal with cyclic processes unless the number of repetitions is known when process execution starts.

Finally, one might argue that the process knowledge is not always known. This is correct. Nevertheless, the process information exists implicitly and could be mined from logs, e.g., [20], or collected via state-of-the-art tracing approaches, e.g., [19]. Such an approach would also offer much more flexibility in considering situations where, e.g., process P1 stores a file which then triggers a pub/sub message which then triggers another process P2. Effectively P1 and P2 are the same process which, however, is hard to detect without advanced process mining and tracing.

In future work, we plan to work on a provider-side alternative to our approaches, e.g., as part of the open source OpenWhisk, but also to explore how a "learned" process could be used as part of our existing approaches.

7 RELATED WORK

In this section, we discuss related work, starting with function composition before focusing on provider-side and client-side approaches for cold start optimization.

7.1 Composition of Serverless Functions

Lopez et al. [6] evaluated IBM Composer, Amazon Step Functions, and Azure Durable.All three are platform-specific and use an external orchestrator which coordinates the function invocations in a centralized way. Our approach, in contrast, uses a choreography-based process model which inherits several advantages, such as the independence from cloud providers and therefore the ability to easily integrate into federated FaaS systems.

Another solution to function composition is to implement the orchestrator as an additional function which, according to Baldini et al. [2], leads to "double-billing". Therefore, OpenWhisk provides the ability for sequential function compositions without double-billing, so-called function sequences [2, 6].

Beyond these, Fouladi et al. [5] presented an approach that distributes large compilation tasks across serverless functions. Their purpose, however, is not function composition but rather to split the processing of a large DAG into individual tasks, distribute those across functions, and collect the results. Also, Malawski et al. [13] have extended the HyperFlow system to distribute subtasks from scientific workflows to AWS Lambda and Google Cloud Functions. Comparable to Fouladi et al. [5], their focus is not on function composition but rather on distribution of processing tasks.

7.2 Optimizing Cold Starts within the FaaS Platform

The cold start problem has been identified as a challenge for serverless computing [2, 21]. Also, multiple studies have discussed cold start latency and its influence factors, e.g., [11, 14, 17].

Based on these, a number of approaches have been developed which aim to reduce the cold start latency directly:

OpenFaaS[15] is a simple example for provider-side cold start optimization. It always keeps a single warm container for every deployed function [18]. This, however, only addresses scenarios in which the arrival rate increase is less than one divided by the average cold start latency. For every additional concurrent request, there are still cold starts. Likewise, OpenWhisk [2] uses so-called "stem cells" which are running containers that use a base image without the function code and its libraries. This reduces the cold start time as containers are already "semi-ready".

Oakes et al. [15] proposed an approach called Pipsqueak which aims at reducing the size of deployment packages. For this goal, they extend the OpenLambda platform [9] to give single functions access

[15] https://www.openfaas.com/

to pre-deployed, shared python libraries. A similar approach [22] involves the use of unikernels instead of containers. As these are more light-weight they tend to have a shorter initialization duration for function instances.

SAND [1] is an alternative FaaS platform that collocates functions of the same application within one container and can, thus, avoid cold start accumulation. SAND does not need our approach; however, as a research prototype, it is far from being production-ready so that we still have to deal with today's FaaS services.

Overall, the key difference is that our approaches can be used with multi-platform deployments and do not require cooperation or action of the platform provider.

7.3 Optimizing Cold Starts in the Application

There are a number of approaches that can be used to optimize cold starts within the application:

Manner et al. [14] identified factors that affect cold start latency. Among others, they showed that there is a correlation between the cold start latency and the specified memory size. While users with little cost constraints can simply specify the maximum memory size, choosing the optimal size requires to consider a cost/performance tradeoff [12] and is, thus, often not realistic.

Puripunpinyo et al. [17] evaluated how to reduce the deployment package size of Java-based serverless functions which, as described above, reduces cold start latency. All these approaches focus on cold start latency and, thus, complement our approach.

Lloyd et al. [12] proposed a simple mechanism to reduce the number of cold starts: clients periodically invoke all functions to avoid unprovisioning of unused instances. Overall, this corresponds to our hinting mechanism but can only assert availability of a single function container, i.e., it helps for workloads with single periodic requests but not for a growing load.

An alternative to our approach would be composition by fusion [2]. This, however, is only possible for single-platform processes and when the resulting artifact does not violate platform limits (execution duration and executable size). Preferably, this fusion would also be done automatically to maintain code clarity for developers – we are not aware of any approach in this regard.

Overall, there are only a few approaches that actually aim to reduce the number of cold starts instead of reducing the cold start duration. Of these, our approach is, to our knowledge, the first one to leverage insights on application processes and function composition to reduce the number of cold starts.

8 CONCLUSION

In the last few years, "serverless" FaaS has become very popular. In FaaS, developers deploy individual functions while the platform provider manages the infrastructure, e.g., scaling in or out as needed. FaaS-based applications often incur so-called cold start latency which is the extra delay caused by provisioning a new function instance (typically a container). Recent work has mostly focused on the reduction of cold start latency, e.g., through smaller container sizes [15] or the use of unikernels instead of containers [22].

In this paper, we proposed a complementary client-side approach which reduces the number of cold starts using application knowledge on function compositions. Intuitively, once such a "process" is started, we know the approximate number and time of requests to later functions in that process. For this purpose, we presented a lightweight multi-platform choreography middleware that can be co-deployed with serverless functions. We then proposed three approaches, implemented as part of said middleware, which reduce the number of cold starts while considering the FaaS platform a black box. Through a large number of experiments, we could show that our approaches remove an average of about 30-40% and in some cases up to 80% of all cold starts at low monetary cost.

REFERENCES

[1] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. {SAND}: Towards High-Performance Serverless Computing. In *Proc. of ATC*. USENIX.

[2] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. 2017. The Serverless Trilemma: Function Composition for Serverless Computing. In *Proc. of Onward!* ACM.

[3] D. Bermbach, F. Pallas, D. García Pérez, P. Plebani, M. Anderson, R. Kat, and S. Tai. 2017. A Research Perspective on Fog Computing. In *Proc. of ISYCC*. Springer.

[4] Rodrigo N Calheiros, Enayat Masoumi, Rajiv Ranjan, and Rajkumar Buyya. 2015. Workload prediction using ARIMA model and its impact on cloud applications' QoS. *IEEE Transactions on Cloud Computing* 3, 4 (2015), 449–458.

[5] Sadjad Fouladi, Dan Iter, and Shuvo Chatterjee. 2017. A thunk to remember: make-j1000 (and other jobs) on functions-as-a-service infrastructure. *Preprint, https://web.stanford.edu/~sadjad/gg-paper.pdf* (2017).

[6] P. Garcia Lopez, M. Sanchez-Artigas, G. Paras, D. Barcelona Pons, A. Ruiz Ollobarren, and D. Arroyo Pinto. 2018. Comparison of FaaS Orchestration Systems. In *Proc. of WoSC*. IEEE.

[7] Jonathan Hasenburg, Sebastian Werner, and David Bermbach. 2018. Supporting the Evaluation of Fog-based IoT Applications During the Design Phase. In *Proc. of M4IoT*. ACM.

[8] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *Proceedings of CIDR*.

[9] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with openLambda. In *Proc. of HotCloud*. USENIX.

[10] J. Kuhlenkamp and S. Werner. 2018. Benchmarking FaaS Platforms: Call for Community Participation. In *Proc. of WoSC*. IEEE.

[11] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara. 2018. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. In *Proc. of IC2E*. IEEE.

[12] Wesley Lloyd, Minh Vu, Baojia Zhang, Olaf David, and George Leavesley. 2018. Improving Application Migration to Serverless Computing Platforms: Latency Mitigation with Keep-Alive Workloads. In *Proc. of WoSC*. IEEE.

[13] Maciej Malawski, Adam Gajek, Adam Zima, Bartosz Balis, and Kamil Figiela. 2017. Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions. *FGCS* (2017).

[14] Johannes Manner, Martin Endreß, Tobias Heckel, and Guido Wirtz. 2018. Cold Start Influencing Factors in Function as a Service. In *Proc. of WoSC*. IEEE.

[15] E. Oakes, L. Yang, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. 2017. Pipsqueak: Lean Lambdas with Large Libraries. In *Proc. of WoSC*. IEEE.

[16] Tobias Pfandzelter and David Bermbach. 2019. IoT Data Processing in the Fog: Functions, Streams, or Batch Processing?. In *Proc. of DaMove*. IEEE.

[17] H. Puripunpinyo and M. H. Samadzadeh. 2017. Effect of optimizing Java deployment artifacts on AWS Lambda. In *Proc. of DCPerf*. IEEE.

[18] Simon Shillaker and Peter Pietzuch. 2018. A Provider-Friendly Serverless Framework for Latency-Critical Applications. In *Doctoral Symposium of Eurosys*. ACM.

[19] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. *Google Tech. Rep.* (2010).

[20] Wil MP Van der Aalst and AJMM Weijters. 2004. Process mining: a research agenda. *Computers in Industry* (2004).

[21] Erwin van Eyk, Alexandru Iosup, Simon Seif, and Markus Thömmes. 2017. The SPEC Cloud Group's Research Vision on FaaS and Serverless Architectures. In *Proceedings of the International Workshop on Serverless Computing*. ACM.

[22] Madhuri Yechuri. 2017. Unikernels and event-driven serverless platforms. *https://serverless.com/blog/madhuri-yechuri-unikernels-event-driven-serverless-emit-2017/ (accessed Sep 20, 2019)* (2017).