Xanadu: Mitigating cascading cold starts in serverless function chain deployments

Nilanjan Daw
Department of Computer Science
and Engineering
Indian Institute of Technology Bombay
nilanjandaw@cse.iitb.ac.in

Umesh Bellur
Department of Computer Science
and Engineering
Indian Institute of Technology Bombay
umesh@cse.iitb.ac.in

Purushottam Kulkarni
Department of Computer Science
and Engineering
Indian Institute of Technology Bombay
puru@cse.iitb.ac.in

Abstract

Organization of tasks as workflows are an essential feature to expand the applicability of the serverless computing framework. Existing serverless platforms are either agnostic to function chains (workflows as a composition of functions) or rely on naive provisioning and management mechanisms of the serverless framework—an example is that they provision resources after the trigger to each function in a workflow arrives thereby forcing a setup latency for each function in the workflow. In this work, we focus on mitigating the cascading cold start problem— the latency overheads in triggering a sequence of serverless functions according to a workflow specification. We first establish the nature and extent of the cascading effects in cold start situations across multiple commercial server platforms and cloud providers. Towards mitigating these cascading overheads, we design and develop several optimizations, that are built into our tool Xanadu. Xanadu offers multiple instantiation options based on the desired runtime isolation requirements and supports function chaining with or without explicit workflow specifications. Xanadu's optimizations to address the cascading cold start problem are built on speculative and just-in-time provisioning of resources. Our evaluation of the Xanadu system reveals almost complete elimination of cascading cold starts at minimal cost overheads, outperforming the available state of the art platforms. For even relatively short workflows, Xanadu reduces platform overheads by almost 18x compared to Knative and 10x compared to Apache Openwhisk.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Theory of computation** → **Distributed computing models**; • **Software and its engineering** → **Scheduling**.

Keywords: Serverless workflows, Speculative deployment, Just-in-time scheduling

ACM Reference Format:
Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2020. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *21st International Middleware Conference (Middleware '20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3423211.3425690

1 Introduction

Serverless computing, the new-kid-on-the-block in cloud services, obviates the need to rent and manage runtime environments such as virtual machines or containers [18, 23, 41]. Its commercial avatar, Function as a Service (FaaS), accepts autonomous and stateless compute tasks (aka *functions*)—along with specifications of triggers used to invoke functions. The provider provisions a runtime environment for functions to execute in, dispatches the requests, and deals with the elasticity of resource provisioning and other management tasks as needed. In exchange, the provider charges for resources consumed per instance of the function execution. Typically functions are small blocks of code needing fine-grained resources to execute for short periods of time. The flexibility that this model affords to the provider to manage resources allows for higher resource utilization and efficiency while offering inexpensive pay-as-you-use models to clients deploying applications in the FaaS mode. The attractiveness of such a proposition has made primary cloud service providers support serverless platforms such as AWS Lambda [6], Google Cloud Functions [12], and Azure Functions [9] coexisting alongside open-source efforts such as Knative [16] and Apache OpenWhisk [5]. However, this flexibility comes at the cost of provider-side resource management complexity - the typical trade-off is between the competing factors of having to keep pre-provisioned resources ready to offer low latency for function responses (thereby reducing efficiency for the provider) vs. allowing client requests to suffer high latency (while it waits for resources to be provisioned on the fly before the function can execute).

Real-world applications are often complex [18, 57] involving interactions between smaller tasks—MapReduce based data processing [25, 44], algebraic operations at scale [36, 55], video analytics [22, 66]. In the serverless paradigm, this requirement translates to control and data flow between functions, resulting in complex workflows of these functions (also referred to as *function chains*). Recognising the need for such function chains, service providers support explicit specification of workflows involving functions [7, 8, 13]. Providers allow clients to use conditional branching, parallel execution with synchronisation barriers, and simple sequencing of functions as part of their workflow specification.

The latency VS efficiency trade-off that the provider must juggle depends on the level of isolation guarantees of runtime environments [10, 19, 23, 27, 28, 46] used to execute functions. Examples of these runtime sandboxes include containers and lightweight virtual machines. A key performance parameter is the startup latency—the duration between the occurrence of the user-specified trigger and the start of function execution. Studies have established that a cold start (where the execution environment for the function to run when the trigger arrives does not exist and one is provisioned on

Middleware '20, December 7–11, 2020, Delft, Netherlands

ACM ISBN 978-1-4503-8153-6/20/12…$15.00
https://doi.org/10.1145/3423211.3425690

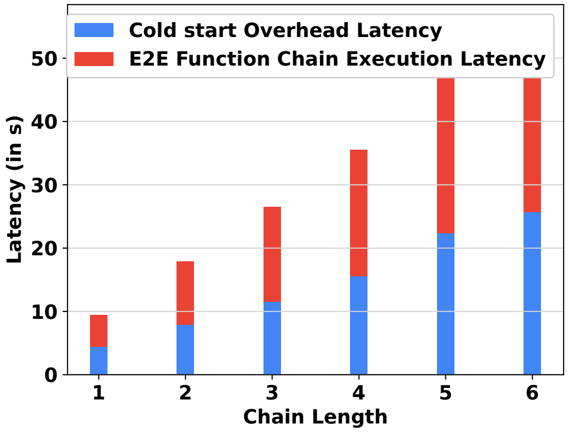


Figure 1. Cascading cold start overheads for a linear chain of functions instantiated using Docker containers.

demand) can take up to 70% of a function’s lifetime [32]. Subsequent triggers can reuse existing execution sandboxes (warm start) only if they arrive within a platform-specific interval. The interval between the arrival of these triggers represents provisioned resources not yielding any returns for the provider and hence represent wasted resource allocation. Reducing the startup latency while minimising resource wastage and under-utilisation is an active area of work [23, 50, 51, 64]—–our paper deals with this tricky trade-off in the context of function chains.

Function chains amplify the cold start problem since the impact cascades through the depth of the chain—the initial trigger starts the first function which in turn causes an internal trigger to start the next function(s) and so on till the workflow terminates. The severity of this cascading cold start is due to the fact that current FaaS platforms treat functions as autonomous entities for provisioning and resource management purposes and hence are chaining agnostic (despite support for workflow of functions). Figure 1 shows the impact of increasing the chain length of a simple linear chain on cold start latency. The execution duration of a function chain with cold start is dominated primarily by the environment provisioning latency, library download and setup latency, and process startup latency, together constituting the cold start latency. The execution duration also consists of other minor delays like the networking and signalling delays. However, since these are orders of magnitude lower as compared to the cold start latency, the difference between the end-to-end execution duration of a function chain and the cumulative raw function execution duration is considered as the cold start latency for workflows. The provisioning overhead of a chain increases linearly with the chain length (**Observation 1**), and for a function execution duration of 5s each, a cascading cold start can account for 46% of the total workflow duration for a chain length of 6 and for smaller functions with execution times of 500 ms, the cascading cold start overhead increases to up to 90% for a workflow of the same length.

Further, the impact of cascading cold starts depends on the sandboxing environment employed to meet isolation requirements. The requirements of robust isolation are met by employing lightweight VMs and containers (cold start latency ~3000ms), which have higher startup latency compared to processes and threads (cold start latency ~1000ms). Correspondingly, the impact of cascading cold start is more dramatic with stronger isolation mechanisms (**Observation 2**).

The naive approach of provisioning all the resources of a function chain at the start is very resource-inefficient, Furthermore, this inefficiency is proportional to the length and structure of the chain. Further, *implicit* chains, where workflows are not explicitly specified but are coded into the functions themselves are not even considered for optimizations in existing solutions (**Observation 3**).

Towards mitigating the cascading cold start overheads with function chains, we propose Xanadu. Xanadu employs a just-in-time resource provisioning technique to prevent cascading cold starts while at the same time maintaining high resource usage efficiency. The key ideas stem from the observation that function dependencies are either known a priori (with explicit specification) or can be inferred by the FaaS platform (**Observation 4**). We employ a probabilistic model of the function chain execution path to speculatively deploy resources just before they are required for progressing the execution of the chain. Xanadu offers a speculation aggressiveness parameter to control the extent of proactive deployment, which is tuned dynamically based on function chain characteristics and deployment costs. We measure the performance of Xanadu under multiple chaining scenarios and real-world applications and compare them with both open source platforms as well as with commercial proprietary cloud offerings. Our evaluation shows that Xanadu eliminates cascading cold starts, reducing the linear growth of the overhead latency to a constant overhead. This results in up to 5x-7x overhead reduction in emulated real-world workloads.

The main contributions of our work are:

1. A detailed analysis of cascading cold starts for workflows instantiated using function chains on popular platforms.
2. A novel platform for providing Function as a Service offering, that has speculative and just-in-time resource provisioning to eliminate the linearly increasing cascading cold start overheads as well as techniques to detect implicit chains and apply the provisioning optimizations.
3. Comprehensive evaluation with multiple chaining scenarios, platforms and real-world applications to demonstrate the efficacy of our solution.

The rest of the paper is structured as follows. We provide a brief background into function chains and cascading cold starts in Section 2, and introduce the key ideas behind Xanadu in Section 3. We discuss Xanadu’s implementation in Section 4 and performance evaluation in Section 5, followed by discussing related work in Section 6 finally concluding in Section 8.

2 Background & Motivation

We now formally define function chains and establish the severity of the problem empirically by looking into the effects of cascading cold starts, under different conditions like depth of the chain, request arrival pattern & the choice of sandbox technique. We also present evidence of cascading cold starts in public cloud infrastructure. Based on the evidence, we motivate the need for speculative resource provisioning in function chains.

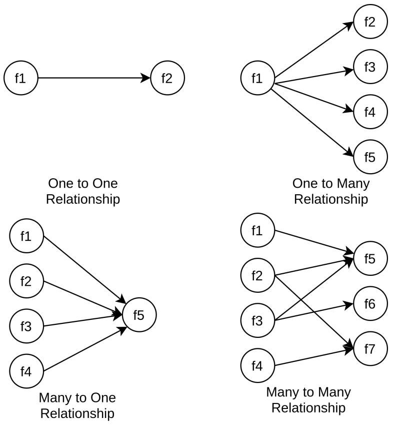


Figure 2. Types of inter-function relationships in function chains.

2.1 Function Chaining

A function chain is a workflow of functions meant to deliver comprehensive functionality of an application. While, in the most general case, there may be cycles/loops in this ordering of functions, we assume the workflow to be a directed acyclic graph (DAG) for the purposes of this effort. Control flow in such graphs allow all or a subset of one-to-one (1:1), one-to-many (1:m), many-to-one (m:1) and many-to-many (m:n) relationships between functions of a chain, as shown in Figure 2. The direction of arrows in the figure refer to the direction of flow of control for the functions; for example in the 1:1 scenario, `f2` must wait till the completion of the function `f1`, while in the 1:m case, the completion of the function `f1` acts as a multi-cast trigger to functions `f2` to `f5`. A variant of the $1 : m$ relationship is one where only one the of m downstream children (we refer to this behaviour as *XOR cast*) is invoked based on the output of the parent function. In the m:1 relationship, `f5` must wait till all its dependencies complete, effectively behaving as a barrier for multiple parallel functions `f1` to `f4`. The m:n relationship combines the features of 1:m and m:1 relationships with a subset of functions acting as multi-cast triggers while some others are acting as synchronization barriers. Based on how these relationships are expressed, function chains can be classified into, explicit and implicit function chains.

2.1.1 Explicit Function Chains. Explicit function chains are those which have a developer specified workflow schema in the form of a processable descriptor that is interpreted by a platform to orchestrate the workflow. AWS Step Functions allow such explicit workflows using a JSON based structured states language called the Amazon States Language [1], IBM Cloud Functions uses a pseudo-programming language-based API [13] to generate the workflow schemas. Azure Durable functions provide APIs to implement custom function chaining logic.

2.1.2 Implicit Function Chains. The presence of an externalised workflow schema opens up avenues to optimise resource allocation needed for the functions of the workflow. However, functions can themselves invoke other functions specified as part of the function's body, which can cascade (synchronously or asynchronously), building an *implicit chain*. With implicit function chains, each function in the chain behaves as a part of a distributed orchestrator, without the platform playing any role in the orchestration. Implicit chains are challenging to detect and optimise and can lead to unexpected performance degradation in seemingly standalone functions.

Regardless of whether the schema is externalised, function chains can also, be classified into deterministic and conditional function chains based on their behaviour of following a deterministic path on every run or dynamically selecting a path based on the inputs and outputs of each function in the workflow.

2.2 Cascading cold starts in Function Chains

Cold start latency can account for a significant portion of a function's runtime overhead. It has been shown that for short functions, cold start latency can occupy up to 90% of a function's total response time [52]. This latency overhead aggravates in case of function workflows. A single trigger to a workflow can generate multiple cold starts, one per function that needs to be invoked as part of the workflow. In the worst case, this overhead latency is proportional to the depth of the chain.

2.3 Empirical Analysis of FaaS Platforms

We divide our investigation into the impact of cascading cold starts into three parts. First, we investigated the extent to which the performance of a workflow degrades when it suffers from cascading cold starts. Cold start *overhead* is the latency suffered by a function beyond its actual execution time, due to resource provisioning delays, networking delays & userspace setup and orchestration delays. For a workflow, cascading cold start latency is the aggregate latency of the slowest control flow branch of the workflow. A function workflow F, with overall runtime R_F consisting of n linearly ordered functions $i \in \{1 \dots n\}$, each with individual runtime r_i, has an overhead latency C_D defined as,

$$C_D = R_F - \sum_{i=1}^{n} r_i \tag{1}$$

We deployed a simple linear function chain with individual function runtime (r_i) of 500ms on the AWS and Azure serverless platforms. In both cases, we use the workflow management tools of the corresponding platforms, AWS Step Functions (ASF) and Azure Durable Functions (ADF) to execute the chains and report latency. We varied the size of the chains from 1 to 5 functions, and executed the chains under both cold start and warm start conditions to contrast the effect of cascading cold starts on the chain's performance. As shown in Figure 3, both AWS Step functions and Azure Durable functions show strong linear behaviour with R^2 value of 0.993 and 0.953, respectively. On an average, cold start overheads accounted for 48.5% of the total runtime on ASF and 41.2% on ADF compared to 13.2% and 13.8% for warm start on the respective platforms.

To ensure that the buildup of latency is due to cascading cold starts and not because of other platform-specific factors, we repeated our tests on two open-source platforms, Knative, initially developed by Google and Apache OpenWhisk, which forms the basis for IBM's serverless platform. Both these platforms exhibit the same behavioural patterns of linearly increasing cold start latency, with even more overhead compared to ASF and ADF (Figure 4). The open-source solutions use general-purpose Docker containers to isolate workers, with high cold start latency, instead of optimised solutions (such as lightweight VMs). We also observed that OpenWhisk in standalone mode keeps a limited number of containers

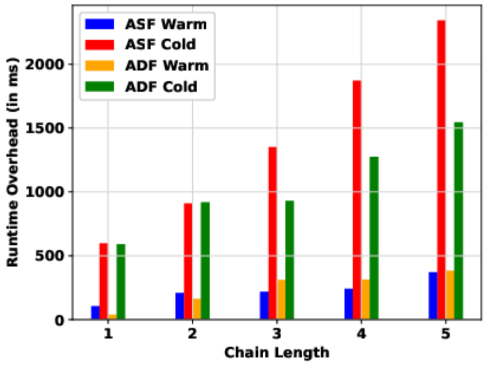


Figure 3. Cascading cold starts in AWS Step Functions (ASF) and Azure Durable functions (ADF).

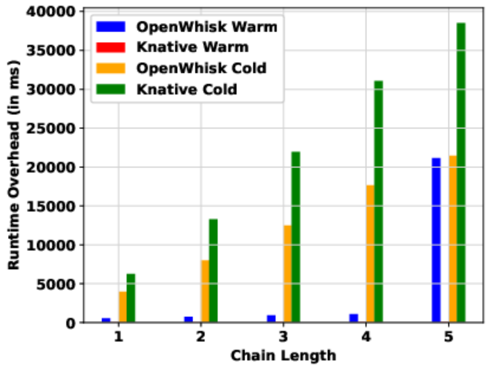


Figure 4. Cascading cold starts with Knative Implicit Functions and OpenWhisk Sequences.

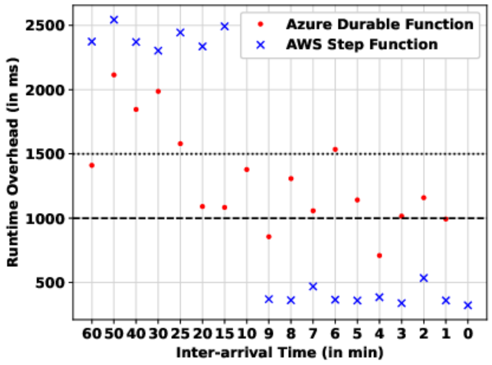


Figure 5. Cascading cold start profiles for function chains with decreasing request intervals.

warm, even for consecutive requests, which explains the sudden increase in cold start latency for chain length 5.

Next, we explored the frequency with which the ASF and ADF serverless platforms reclaim function (isolation sandbox) resources that result in cascading cold starts in *subsequent* requests to the chain. We started two linear no-op function chains, of depth 5 using AWS Step functions and Azure Durable Functions. Requests were sent at a decreasing arithmetic progression of intervals starting at 60 minutes with a common difference of 10 minutes. To increase the granularity of investigation, we decreased the common difference to 5 minutes when the interval of sending requests came down to 30 minutes and further decreased the common difference to 1 minute when the interval of request came down to 10 minutes. We repeated the experiment 5 times with a cumulative experimentation time of about 20 hours. As shown in Figure 5, we found that the ASF platform reclaims workflow resources after ~10 minutes of idle time. As can be seen, the overhead time drops noticeably (from close to 2.5 seconds to 0.5 seconds) for request inter-arrival times of less than 10 min. With the ADF platform, a similar drop in overhead can be observed after inter-arrival times less than ~20 minutes.

Further, analysis of logs collected from the Azure cloud platform [58] shows a high probability of cascading cold starts. The study reports ~45% of all functions being invoked once or less per hour—a significant proportion of the workload being invoked infrequently. The request inter-arrival time in such a scenario is expected to be larger than a platform's keep-alive time which results in frequent cascading cold starts by a large proportion of these serverless workflows. With this workload characterisation as a reference, we simulated a lightly loaded function workflow (~2 requests/hour) on both the platforms with requests for the two workflows generated at random intervals drawn from a uniform distribution between 0 minutes and 60 minutes ($U(0, 60)$) and ran the experiment for about 16 hours. A timeline diagram is provided in Figure 6. We threshold a latency of 1000ms as the highest warm start latency for the workflow deployed on ASF, and 1500ms as the corresponding threshold for the ADF workflow. We observed about 62.5% of the requests to ADF suffering from cascading cold starts, while on ASF about 78.1% of the requests suffered cascading cold starts. This demonstrates that both the platforms suffer from a high frequency of cascading cold starts with an average overhead of 1400ms on ADF and 1800ms on ASF.

The cold start profile of both the platforms remained relatively stable over the entire duration of the experiment, further revealing that neither of the platforms employs any learning optimisations to mitigate cascading cold start effects. A sub-note from the experiment is that the performance metrics obtained from ASF were more stable compared to that obtained from ADF, Our findings are in line with the findings of [24] for parallel workloads, which also reported variability in performance for Azure Durable Functions.

Lastly, we empirically determined the impact of the choice of execution environment (isolation sandbox) on cascading cold starts in function chains. Most public clouds use a homogeneous isolation environment such as a container or a lightweight VM to execute functions, settling on a trade-off between performance and security. However, depending on a function's security sensitivity, picking the right isolation sandbox can lead to better performance and when required better security. We investigate the effects of isolation environments used by different platforms — V8 isolates [10], processes and containers [5, 16]. Figure 7 shows the runtime overhead of a linear chain whose length has been varied from 1 to 5. As expected, the overheads for container-based environments are higher than

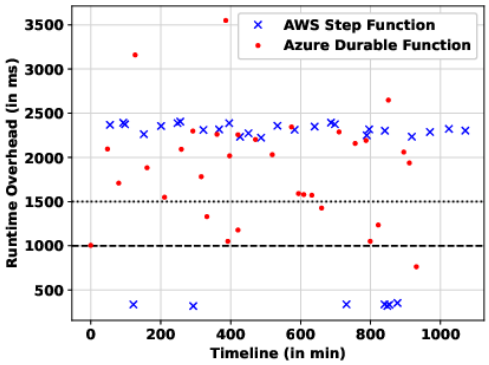


Figure 6. Runtime overhead profile of an emulated lightly-loaded function workflow depicting a large concentration of cascading cold starts.

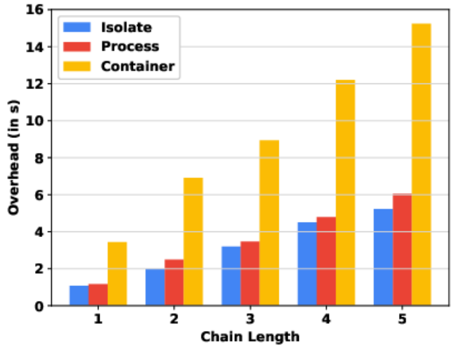


Figure 7. Runtime overhead of different isolation environments.

that with processes or isolates, with container-based chains exhibiting up to a 2.5x to 2.9x increased overhead compared to processes and isolates, respectively. For latency-sensitive function workflows, these execution specific environments need to avoid the high cold start overheads and provide required isolation requirements (and Xanadu provides these optimizations).

2.4 Metrics of Goodness and Cost

Our empirical studies of cascading cold start on the cloud platforms show significant performance degradation for function chains compared to what we would expect if resources had been pre-allocated for each instantiation of the chain. To formally quantify the cascading cold start penalty, we introduce two overhead cost factors, *Latency overhead cost* C_D, and *Resource provisioning overhead cost* C_R. Cost C_D is the additional latency which function chains pay (beyond that needed to execute the functions in the longest path of the workflow) due to cascading cold starts (Equation 1). Cost C_R captures the costs of resources (CPU and memory) provisioned and locked before the actual function execution begins and the resources are put to use. CPU resource cost or $C_{R_{CPU}}$ is the aggregate CPU time slices (measured in `seconds`) spent by a worker, across all allocated CPU cores before the worker starts executing a request, while the memory resource cost or $C_{R_{memory}}$ is the aggregate product of the memory allocated to each worker and the time before being put to use (measured in `MBs`). Formally, for a function chain of depth `n` the memory cost with is defined as,

$$C_{R_{memory}} = \sum_{i=1}^{n} memory_i * (r_i^{total} - r_i^{exec}) \tag{2}$$

where $memory_i$ is the memory allocated to function worker `i` whose total runtime and actual function execution time are r_i^{total} and r_i^{exec} respectively. We combine the notions of latency overhead and resource allocation overhead into a single penalty factor, which is the product of the cost factors C_D and C_R, again subdivided into ϕ_{cpu} (defined as $C_{R_{CPU}} * C_D$), measured in s^2, and ϕ_{memory} (defined as $C_{R_{memory}} * C_D$), measured in MBs^2.

The goal of any FaaS platform must be to minimise ϕ to reduce latency overhead without infrastructure expenditure overruns. Current systems suffer from high C_D costs while minimising C_R, due to lack of workflow level cold start optimisations. On the other hand, naively over-provisioning resources to mitigate C_D, such as pre-crafted resource pools [45, 52, 54] increases C_R. Either of these approaches increases the joint factor ϕ.

We propose Xanadu, *a speculative, just-in-time, multi-granular isolation* based function orchestration system which uses function profiling to eliminate linearly increasing cascading cold starts in function workflows. Xanadu adopts a combination of techniques to eliminate cascading cold starts, it profiles functions to predict implicit chains in correlated functions. predicts the Most Likely Estimated Path (*MLP*) taken by a function workflow when invoked and proactively deploys resources on the *MLP* reducing C_D. To reduce resource idle time, workers are deployed just ahead of a function invocation, providing warm start latency at minimal cost overhead, limiting C_R for an overall low ϕ_{cpu} and low ϕ_{memory}.

3 Xanadu: Key Ideas

In this section, we discuss the building blocks of Xanadu that enable it to eliminate cascading cold start starts while keeping resource provisioning cost overheads minimal. We first describe how Xanadu detects the most likely path a function workflow will execute, followed by how to perform Just-in-time deployment of resources, and finally how Xanadu detects implicit chains and pre-provision resources just in time for them as well.

3.1 Inferring the Most Likely Path

Availability of workflow information in explicit chains lends itself to pre-deployment of resources for all the functions of the workflow. While this succeeds in getting rid of the cold start overhead for deterministic chains, it would lead to resource wastage in case of conditional workflows. Consider the case of a multicast workflow, as shown in Figure 8. With such workflows, deploying the entire function chain implies provisioning of worker sandboxes for functions along all possible paths, while actual resource utilization will occur along a single path of execution. To prevent such over-deployment of resources, we estimate the most likely path (MLP) that may be taken by a workflow and only provision sandboxes along the MLP. Xanadu uses a generative model to estimate the most likely path in the workflow DAG. $\rho(Cj|Pi)$ denotes the probability of executing child node C_j, given that the parent node P_i executes. The likelihood that child node C_j executes next is

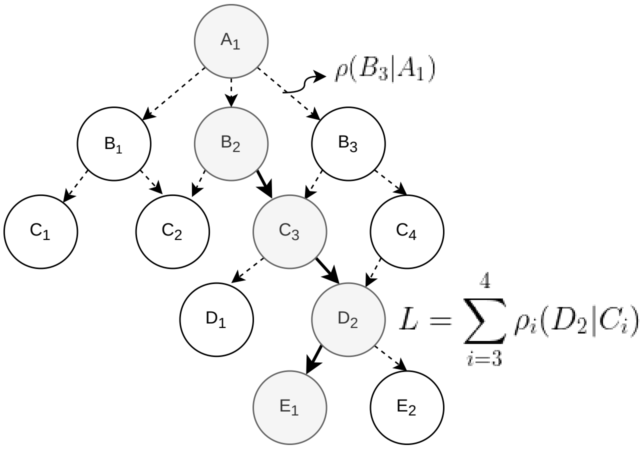


Figure 8. A XOR cast DAG with solid arrows indicating a 70% probability of being triggered. All other siblings at each level are equally likely to be triggered.

dependent on possible parent nodes P_i, $(i \in 1 \dots n)$ and is estimated using a likelihood factor L which is the summation across all parent nodes as follows:

$$L_j = \sum_{i=1}^{n} \rho_i(C_j|P_i) \tag{3}$$

The function with the maximum likelihood factor among all its siblings is appended to the MLP. In the case of `1:1` and XOR cast relationships, the factor is L is upper bounded by 1 and behaves like a probability factor. However, the upper bound does not hold for multicasts and `m:n` relationships. Algorithm 1 lays out the steps to estimate the MLP. The probability of each node being in the path depends on the probability that one of its parent nodes executes and the probability of it executing next. The MLP behaviour of a function can change over time, to account for it, we use exponential averaging for function related metrics (start times, runtimes, and branch probabilities), with the metrics being updated after every fixed interval of time. This procedure lets the MLP algorithm adapt to changes in a workflow's path likelihood while being tolerant of outlier behaviour.

Algorithm 1: Generating the MLP

```
Input: workflow
Output: mlp
parents ← workflow.root;
while parent in parents do
    siblings ← parent.children;
    foreach sibling ∈ siblings do
        sibling.prob ← parent.branch [sibling ] ∗
          parent.prob;
        parents.Append (sibling);
    end
    mlp.Append (Max (siblings));
end
```

To test the effectiveness of the algorithm, we deployed a function chain structured as a conditional branching DAG as shown in Figure 8. We triggered the workflow with 20 requests to observe the behaviour of the algorithm. The branch detection and MLP algorithms ran in parallel to detect the workflow branches and their probabilities, while at the same time, generating an MLP based on the available data. The algorithm was able to infer the entire function workflow within 8 triggers to the function chain, visiting each function the tree at least once.

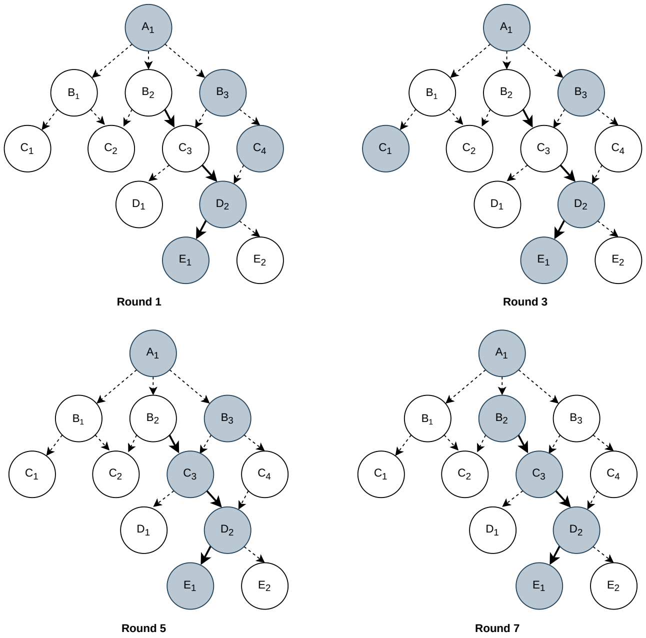


Figure 9. Stages of most-likely-path (MLP) estimation, inferred based on fraction of tree discovered.

We show the evolution stages of the MLP in Figure 9. As can be observed, in the initial stages, the MLP does not represent the actual MLP as the branch probabilities are not accurate, and the entire workflow has not been mapped. During the first 3 triggers, for example, C1 was chosen as a possible candidate for MLP, even though its likelihood factor L_{C_1} is around 0.15. As such, after the first 3 requests (Round 3), D2 and E1 were the only functions correctly identified as part of the MLP. However, with more triggers to the workflow, the MLP starts converging, with 80% of the MLP functions being correctly detected after Round 5. The inference algorithm was able to converge to the MLP within 7 triggers to the chain. We also observed that after the convergence, there was no oscillation in the MLP, and it stayed the same at the end of the 20 triggers. We provide a more detailed evaluation of the efficacy of the MLP detection algorithm in Section 5.

On invocation of a particular workflow, Xanadu speculatively deploys all the nodes (functions) identified to be part of the function chain at the onset of the chain execution. This results in chained requests experiencing warm starts since the workers are provisioned and ready.

3.2 Minimizing Pre-provisioning Overheads

Speculatively provisioning sandboxes at the onset of a function workflow can be expensive, especially for extended function chains with large depths. Sandboxes provisioned for functions at the tail of

	Speculation ON (in s)	Speculation OFF (in s)	Average #function miss per request	Average #of workers per request
Average Case	7.62	15.65	0.6	5.6
Worst Case	17.7	17.17	3	8
Best Case	4.8	14.12	0	5

Table 1. Cold start latency and resource cost with different speculation scenarios.

the MLP of such chains incur large idle times resulting in resource wastage, which while mitigates cascading cold start overheads (C_D), is counterproductive with respect to resource utilisation (increasing the resource cost C_R). Furthermore, speculative provisioning on conditional chains can suffer from prediction misses, which occurs when the function chain under consideration deviates from its expected path of execution. Deploying the entire MLP upfront can lead to considerable resource wastage in such cases of prediction misses. The speculatively deployed resources have to be discarded, and new isolation sandbox instances have to be provisioned to serve the actual path taken by the workflow. This results in both increased latency and resource wastage increasing both the costs, C_D and C_R. Table 1 shows the effect of prediction misses in a function chain of depth 5 with 3 conditional points, for 10 cold start triggers. We observe that the worker count (and hence the resource cost) and request latency increases considerably when the chain suffers from prediction misses. For example, in the worst case of 3 prediction misses, the performance degradation, compounded by Docker's concurrent scalability issues [52, 54], lead to a cold start performance worse than that observed with no optimisation.

We propose two techniques to reduce the impact of prediction misses in speculative provisioning.

3.2.1 Deployment Aggressiveness.

The first of these is a provider-side continuous-scale tunable parameter to control the aggressiveness with which resources are provisioned. This parameter controls the fraction of the total depth to which execution sandboxes for functions should be provisioned or how much should the pre-provisioning algorithm look ahead. This prevents cost overrun due to eager pre-deployment and reduces miss penalty in case the workflow deviates to a different path than that predicted by the MLP estimation algorithm. The aggressiveness parameter allows service providers to juggle the trade-off between cost overruns and cold start overheads.

3.2.2 Just In Time Deployment.

In case the predicted MLP is incorrect, there will arise a situation in which we will speculatively have deployed a sandbox that will not be used. The platform recovers from this prediction miss by shutting down the incorrectly deployed container and starting up a new one for the path now being taken by the workflow. The deployment aggressiveness parameter does not overcome the double provisioning in case of a prediction miss. We propose *Just in Time (JIT) deployment* to tackle this issue. The idea is to provision an isolation sandbox just in time for its use as opposed to provisioning it ahead of time speculatively. We profile the runtime characteristics of the functions comprising a workflow and estimate their cold-start time, worker startup time and warm-start runtime using an exponential moving average function. For implicit functions, we also measure the delay after which a parent node invokes its child (next function in the chain). We use these function profiles to speculatively provision sandboxes on the MLP just-in-time for their expected invocation. This JIT deployment is done by adding a planning phase to the orchestration component. When a request arrives for a workflow execution, the orchestrator starts invocation of the function at the start of the DAG while starting the planning phase in parallel. The planning phase generates a timeline estimating the time of deployment of the function nodes based on startup times of the child sandbox (S_c) and expected time of invocation of the child.

For explicit chains, the invocation delay for a child depends upon the lifetime of the parent(s), since a child can only be invoked by the orchestrator upon completion of its parent(s) on whom it depends. In case of a `m:1` barrier function, this translates to be the lifetime of the slowest parent, which acts as the barrier bottleneck. The complete algorithm to generate the JIT path is given in algorithm 2. Since the algorithm works transparently, without hooks into the function runtime, it is not possible to estimate the exact lifetime of any function. Instead, we use the warmstart time as a reasonable estimate of a functions lifetime. In case of implicit functions, however, child nodes are invoked directly by their parents' runtime, which invalidates our previous proposition of using the parents' *warmstart* time as a measure of the delay in invoking the child. Alternatively, Xanadu maintains an internal list of requests and their arrival timestamps. We assume that parent to child requests maintain a chronological ordering, i.e. parent requests arriving earlier invoke their child functions earlier, thus maintaining a one-to-one mapping between parent and child requests based on their arrival timestamps. Even though this assumption might not hold for every request, it is statistically sound for a large number of requests. The mapping is used to infer the call delay between a parent node and a child node. Algorithm 2 thus needs to be slightly updated for implicit chains, where *coldstart* time and *warmstart* time used in line 5 and line 10 respectively needs to be replaced with the invoke time calculated as discussed.

To tackle the problem of wasteful resource provisioning in case of prediction misses, JIT deployment stops all planned proactive provisioning as soon as it detects a prediction miss. Thus, even though we pay the price of cascading cold starts due to miss penalty (increase in C_D), we avoid the double cost of wasteful speculative provisioning (by controlling C_R).

3.3 Detecting Implicit Chains

The absence of a workflow schema with implicit chains makes it challenging to identify workflow paths and pre-deploy resources. We observe that all requests originating from functions in a chain have their point of origin and the destination function as worker nodes managed by Xanadu. Xanadu exploits this phenomenon to identify function branches belonging to a chain. We patched the HTTP request libraries to include a unique identifier as part of the request header, identifying the function initiating such a function request. A request received by the function dispatcher identifies the caller and callee functions to build function chain. We calculate the conditional probability of a (child) function C being invoked provided the parent function P was invoked $\rho(C|P)$ as a ratio between

Algorithm 2: Algorithm to Generate JIT deployment plan for explicit workflows

```
Input: mlp
Output: jitPlan
while node in mlp do
    if node.dependency is empty then
        // root nodes without dependencies are
           called immediately
        jitPath.Append (node, 0);
        node.maxDelay ⟵ node.coldtstart;
    else
        delay ⟵ Max (node.parents.maxDelay);
        delay ⟵ delay- node.starttime;
        jitPath.Append (node, delay);
        delay ⟵ delay + node.warmtstart;
        node.maxDelay ⟵ delay;
    end
end
```

Algorithm 3: Algorithm to detect function chain branches in implicit chains

```
Input: request, branchTree
Output: branchTree
if request.header.Has (parentID) then
    parent ⟵ branchTree.get(request.header);
    child ⟵ parent.branches.get(request.dest);
    child.probability ⟵
        $\frac{(\text{child}.probability*\text{child}.requestCount)+1}{(\text{child}.requestCount{++})}$;
    siblings ⟵ parent.branches;
    foreach sibling ∈ siblings do
        if sibling ≠ child then
            sibling.probability ⟵
                $\frac{\text{sibling}.probability*\text{sibling}.requestCount}{(\text{sibling}.requestCount{++})}$;
        end
    end
else
    parent ⟵ branchTree.get(request.header);
    parent.requestCount ⟵ parent.requestCount + 1;
end
```

the total requests to the child to that of the parent and use it to weigh the branches of function chain tree. For each request received by C, we increase the request count of C by one and update the probability of C and all its siblings. Each function in the workflow thus begins to shadow the actual probability distribution of the workflow as more requests are received. We iteratively update the model and the workflow map based on the outcome and behaviour of requests as further detailed in Algorithm 3.

This generative probabilistic model of the workflow's runtime branching behaviour is treated as the workflow map for an implicit chain used for proactive, ahead of time, pre-provisioning of resources, as discussed in the preceding sections.

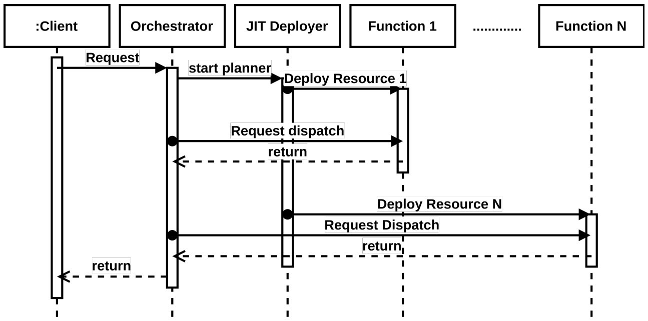


Figure 10. Sequence of operations with Xanadu employing just-in-time speculative deployment.

3.4 Putting It All Together

Xanadu's strategy for eliminating cascading cold starts consists of:

(i) Proactively detecting (implicit) function workflows,
(ii) Speculatively deploying resources ahead of time to reduce cold starts for functions in the workflow &
(iii) Delaying proactive provisioning of resources to minimise runtime overhead costs due to idle workers.

The typical sequence of operations with Xanadu are as shown in Figure 10, and starts with a workflow execution request arriving at the orchestrator. The orchestrator asynchronously invokes the JIT deployer while executing in parallel forwarding requests to functions whose runtime dependencies are met. The JIT deployer generates a deployment plan consisting of function deployment timelines expected to eliminate cold start with minimal execution overhead. The JIT deployer then performs proactive ahead of time deployment of function resources such that when a function request arrives, a runtime is already executing and it encounters a warm start.

4 Implementation

We implemented Xanadu, using `Node.js`, as a testbed to enable quick prototyping of the proposed optimizations. The architecture of Xanadu is shown in Figure 11. The Dispatch Daemon(DD) runs on individual host machines and performs resource provisioning and maintenance of Xanadu workers. Workers are the encapsulation sandboxes within which functions execute. The Dispatch Manager(DM) is the central orchestration component of Xanadu, comprising of two main sub-modules, the Function Resource Allocator which instructs the Dispatch Daemon about the size and the nature of the resource to be provisioned and a Reverse Proxy, forwarding incoming requests to respective host workers. The DM also runs a metrics engine and a branch detector engine in parallel to the reverse proxy which gathers function profile data and detects implicit branches in incoming requests. We use Apache CouchDB [2] to store metrics and function branch-related metadata. CouchDB supports native JSON data support, allowing us to efficiently migrate structured data to and from the backend server without the need for conversion of data across formats. We use Apache Kafka [4] for internal communication between the Dispatch Manager and the Dispatch Daemon and also for state management of Xanadu workers.

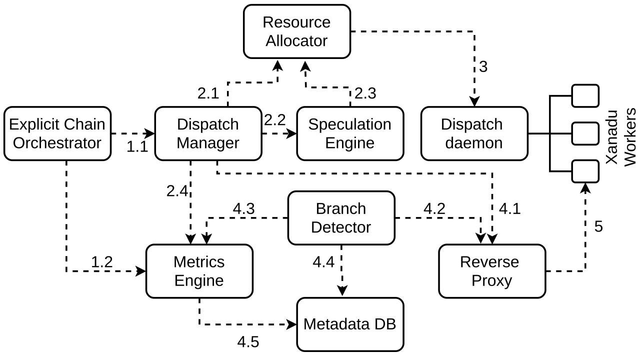


Figure 11. The Xanadu architecture (dotted arrows represent asynchronous calls).

Xanadu workers support multi-granular isolation, with function isolation levels pegged at thread level isolation implemented using JavaScript V8 Isolates [15], process level isolation and container level isolation implemented using Docker [11] Containers. Users specify the level of isolation to be used for each function as a request parameter during function deployment or as part of workflow description for explicit function chains. Explicit chaining support is provided using a state definition language we developed based on JSON. An example of a state specification for an explicit conditional chain is as shown in Listing 1. Functions are defined within blocks of type `function`, with parameters like memory size allocation, isolation sandbox to use and its dependency list. Blocks of type `Conditional` are used to define conditional branching in the workflow, with their `success` and `fail` parameters pointing to the respective branch to invoke.

When an explicit workflow request arrives, the explicit chaining map for that workflow is interpreted by the Explicit Chain Orchestrator module which coordinates with the Xanadu DM for deploying functions and their resources and dispatching function requests, while for an implicit chain, the request is handled directly by the DM. Either way, the arrival of a request at the DM, sets off parallel invocations to the Resource Allocator, Speculation Engine and the Metrics Engine (Communications 2.1 through 2.4 in Figure 11). The Speculation Engine generates the pre-deployment plan and deploys resources as required while the metrics engine collects data related to the workflow's runtime and behaviour. When the resources are ready, the Reverse Proxy forwards the function chain requests to the Xanadu workers. The Branch detector works in parallel to the Reverse Proxy, improving the workflow behaviour model and updating the MLP as required, while backing everything up on the Metadata DB for persistence.

```
"f1":{
    "type":"function", "memory": 512,
    "runtime":"container",
    "wait_for":[],
    "conditional":"condition1"
},
"condition1":{
    "type": "conditional",
    "wait_for":["f1"],
    "condition":{ "op1":"f1.x", "op2": 7, "op":"
        lte" },
    "success":"branch1", "fail":"branch2"
},
"branch1":{
    "type": "branch",
    "f3":{
        ...
    }
},
"branch2":{
    ...
    }
}
```

Listing 1. An explicit conditional chain defined by Xanadu's state definition language

5 Evaluation

This section discusses the evaluation of Xanadu's speculative provisioning of resources for function chains. Xanadu was setup on an x86-64 Linux machine (kernel v4.15.0) with 64 core Intel Xeon 2.1GHz processor and 128GB of memory. The workload was generated from other machines on the same network using JMeter[3]. We compare Xanadu's cascading cold start performance against Knative (Commit *#f87352b*) and OpenWhisk (Commit *#ffef0a9*). We deployed Knative on a single node Kubernetes deployment and deployed OpenWhisk in standalone mode with Docker as a runtime backend for both. Both Knative and OpenWhisk were deployed with default settings without any custom fine-tuning, and with resource provisioning identical to that of Xanadu.

Our evaluation answers the following questions:

(i) What is the rate of increase of latency with speculative deployment?
(ii) What is the cost overhead of speculative deployment?
(iii) How effective is the generative model and how long does it take the model to converge to the MLP?
(iv) What is the average cost overhead and worst case cost and runtime overhead for conditional chains?
(v) What is the effect of JIT deployment on cascading cold start?
(vi) What is the joint penalty factor for no optimization vs speculative deployment vs JIT deployment?
(vii) How does the choice of sandboxing technique, affect the performance of function workflows?
(viii) How does the Speculative and JIT deployment improve performance of real-world applications?

5.1 Speculative Provisioning

We discussed speculative provisioning of execution sandboxes in Xanadu, where we detect the MLP in a function workflow and proactively deploy functions to eliminate cascading cold starts. To evaluate the effectiveness of Speculative Provisioning, we deployed 10 linear chains of depths ranging from chain depth 1 to chain depth 10 (Node.js runtime) with Docker containers as the execution environment. Each function in the workflow has a runtime of 5s. For each chain, we triggered 10 requests in cold start condition. We ran Xanadu first in the no optimisation mode (Xanadu Cold), then with only speculative provisioning (Xanadu Speculative) and finally in the JIT mode (Xanadu JIT) and compare our results with OpenWhisk and Knative.

Figure 12a shows the overhead latency (in logarithmic scale[1]) as we vary the chain length. We observe that as expected OpenWhisk, Knative and Xanadu Cold shows linearly increasing cold start latency due to consecutive cold starts in the workflow. However, Xanadu Speculative shows a stable cold start profile remaining almost constant with increasing chain length. At chain length 10, it had a latency overhead of 4.85s, compared to 76.34s for Knative and 44.38s for OpenWhisk, a 1.11x increase in overhead as opposed to 10.5x and 10.14x increase in overhead in Knative and OpenWhisk respectively.

Takeaways: Xanadu Speculative has an almost constant latency compared to linear growth on other platforms.

5.2 JIT Deployment

Speculative deployment shows an impressive improvement in C_D costs. However, proactive provisioning can be resource expensive, leading to an increase of C_R costs. We measured the cumulative idle CPU time (in ms), the amount of CPU time consumed while the worker remains idle waiting for a request, and the cumulative memory used, as a product of memory used and the worker idle time(measured in MBs). Figure 13a and 13b shows that Speculative Deployment can be up to 15.6% more expensive on the CPU usage parameter and 250 times more expensive than Xanadu Cold concerning memory usage. This can become prohibitively large for practical use case purposes.

Just-in-deployment performs a tight ahead of time resource deployment, significantly reducing C_R costs. We observe that compared to Xanadu Cold, Xanadu JIT is only 0.9% more CPU expensive on an average and 2.18x more memory expensive. A more than an order of magnitude cost improvement compared to Xanadu Speculative.

Improvements in resource utilisation, while keeping performance close to Xanadu Speculative results in reduced penalty factor ϕ as evident in Figures 12b and 12c. Xanadu JIT has an average of 5.8x reduction in ϕ_{cpu} and 1.7x reduction in ϕ_{memory} penalties compared to Xanadu Cold, which means not only does it eliminate Cascading Cold starts, but it does that without significant cost overruns. We further notice that Xanadu JIT has a better overhead latency compared to Xanadu Speculative, on an average showing up to 10% improvement in C_D. This can be attributed to Docker's concurrent scalability bottlenecks as noted earlier. Xanadu Speculative starts all MLP workers at the same time at the commencement of the workflow, and this increases the initial cold start suffered by the chain, compared to Xanadu JIT which spreads resource deployment across the workflow's lifetime alleviating the startup bottleneck—cumulatively leading to an overhead improvement of 16.7x and 9.29x compared to Knative and OpenWhisk respectively.

Takeways: Xanadu JIT eliminates Cascading Cold starts with negligible increase in resource provisioning costs leading to improved penalty factors ϕ_{cpu} and ϕ_{memory}.

5.3 Inferring the MLP

Fast, effective and accurate detection of the MLP path is necessary to perform speculative deployment of resources. The convergence time of the MLP path is a function of multiple factors like the depth of the workflow, the number of conditional points in the workflow and their probabilistic biases. To evaluate the effectiveness of the algorithm and its convergence speed, we deployed 100 randomly generated binary trees with 1 to 10 nodes each with random biases at conditional points. Each tree was explored ten times to learn the workflow behaviour and infer the MLP. Figure 14a and 14b shows the MLP convergence behaviour across two different parameters; the size of the workflow and number of conditional branches. On average, workflows with up to 4 functions needed around two requests to converge, which increased to an average 5.3 requests for workflows with more than eight functions. Similar trends are shown by variance in the number of conditional points in the workflow. Functions with a maximum of 1 conditional branch needed an average of 2 requests to converge, increasing gradually to above 5.2 requests when the number of conditional branches increases to 3. We note that there is considerable variance in either of plots. This can be attributed to the probabilistic biases at each conditional points. A sharp bias expresses itself strongly, which helps the inference algorithm to converge faster, compared to weaker biases, where it might oscillate between equiprobable paths. We also note that barring 1 instance, the inference algorithm was able to converge to the actual MLP. The outlier case had conditional probabilities extremely close to 0.5, which caused it to oscillate between two parallel branches.

Takeaways: The MLP inference algorithm's convergence rate varies as a function of workflow behaviour; even then empirical evaluation shows, under a diverse set of conditions the inference algorithm has a fast convergence rate while maintaining a high degree of accuracy.

5.4 Chains with conditional branching

Conditional branches in chains pose challenges in pre-provisioning resources due to their non-deterministic nature. Execution sandboxes provisioned on the MLP may remain unused when the chain deviates from its expected behaviour. We evaluate our algorithm's performance on the 100 randomly generated trees discussed previously. Each tree was evaluated with 10 requests for a total of 1000 requests. Figure 15a provides a comparative analysis of the average latency overhead for Xanadu's enhancement modes versus Xanadu Cold.

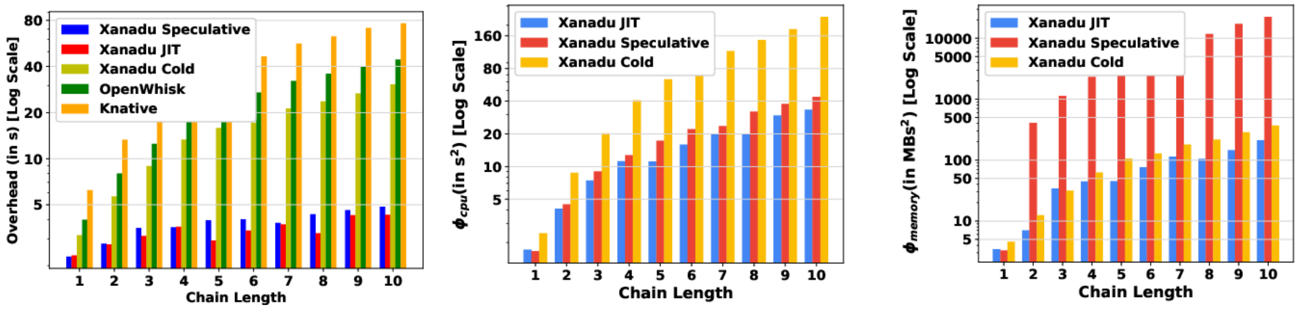
On an average, overhead latency gains in speculative and JIT mode range from 29% to 45% with an average gain of 37% and 34% respectively for chain lengths of more than two. This shows that even under prediction misses, Xanadu provides better cascading latency compared to other platforms. CPU overheads for Speculative deployment remain within 11.9% on average of Xanadu Cold, further reducing to 1% in case of Xanadu JIT (Figure 15b). The average C_R costs for memory stood at 5.8x of Xanadu Cold for Speculative deployment but improved to 2.7x in the JIT mode, as shown in Figure 15c.

Takeaways: Even under prediction misses, Xanadu provides significant performance benefits with slight increase in resource provisioning costs.

5.5 Impact of Sandboxing Mechanisms

We had earlier reviewed the effects of isolation sandboxes on cold start latencies in function chains. Lightweight sandboxes like V8 isolates and processes have shorter startup time leading to lower cold start latencies. Figure 16 details our evaluation of how speculative deployment can improve overhead latencies in irrespective of sandboxing choices. We note that for a linear chain of depth `10`,

[1] All logarithmic scale plots are in base e

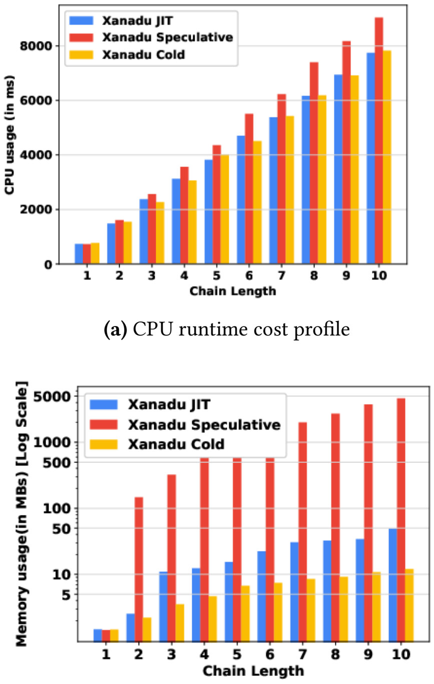


(a) Comparative

(b) ϕ_{cpu}

(c) ϕ_{memory}

Figure 12. Figure 12a: Cascading cold start profiles(C_D) of Xanadu (Cold, Speculative, JIT), OpenWhisk and Knative. Figure 12b and 12c: ϕ_{cpu} and ϕ_{memory} profiles of different Xanadu modes with varying chain lengths.



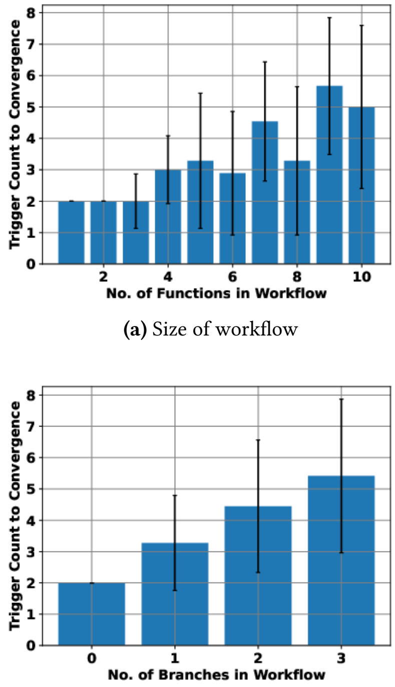
(a) CPU runtime cost profile

(b) Memory runtime cost profile

Figure 13. CPU($C_{R_{CPU}}$) & memory($C_{R_{memory}}$) runtime cost profiles of different Xanadu modes.

with individual function lifetimes of 5000ms, isolate based sandboxes with Speculative deployment show an end-to-end overhead latency of only 1289ms, resulting in a mere 2.5% increase in end-to-end latency. This is a significant improvement over baselines and an acceptable penalty in most latency-sensitive scenarios.

Takeaways: Lightweight sandboxes coupled with Xanadu's pre-deployment strategy are ideal for latency sensitive workloads.



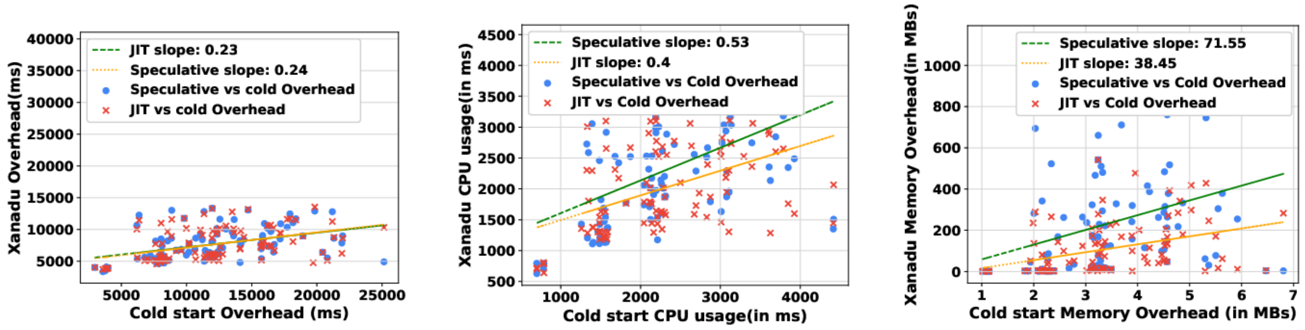
(a) Size of workflow

(b) Branching factor

Figure 14. Time to converge to the MLP (in terms of number of triggers to the workflow) for different sized workflows (Figure 14a) and the number of conditional branches (Figure 14b).

5.6 Case Studies

Analysis of function instantiations on the Azure platform [58] reveals a large variance in function runtimes, ranging from a few milliseconds to tens of seconds. We present two end to end real-world case studies from [32, 34] to illustrate how Xanadu can improve performance of function chains in the wild. Functions of the case studies were implemented using `Node.js`. The first case study, an e-commerce processing pipeline, aims to demonstrate Xanadu's feature of handling variability in function runtimes within a workflow.



(a) Latency Overhead **(b)** CPU Overhead Cost **(c)** Memory Overhead Cost

Figure 15. Scatter plot profiles for Xanadu in JIT and Speculative mode vs Xanadu cold for 100 randomly generated trees, sorted in increasing order of their function count.

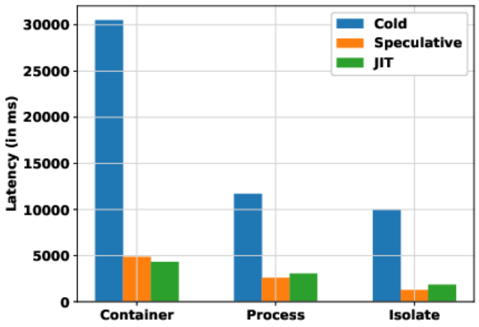
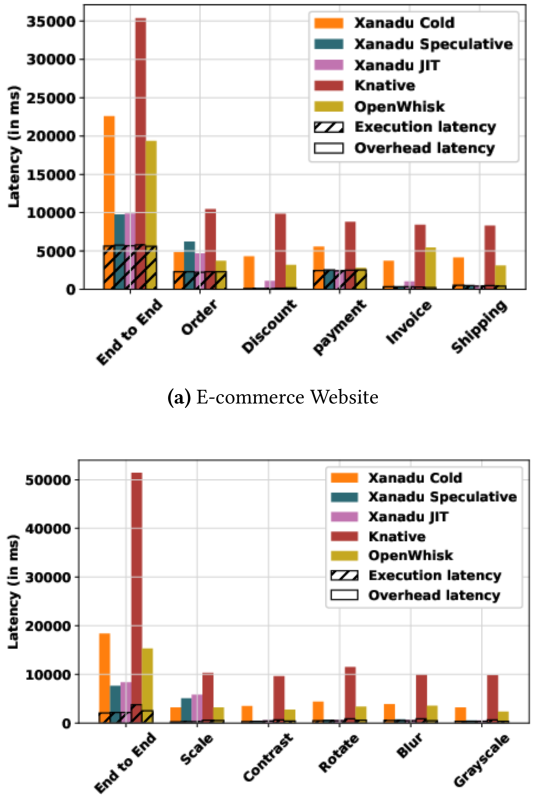


Figure 16. Impact of sandboxing environments (Function chains of depth 10).

The second case study, Image Processing Pipeline demonstrates how Xanadu reduces overheads for pipelines with short but homogeneous runtimes. In each case, we report the execution and overhead latency and compare the performance of Xanadu with Knative and OpenWhisk.

5.6.1 E-Commerce Website. We emulate the checkout process of an E-commerce website as an implicit chain. The chain is triggered by a customer placing an Order(~2000ms), which upon completion invokes the Discount(~100ms) module. The customer is then taken to the Payment (~2500ms) gateway. On successful payment, an Invoice(~300ms) is generated before culminating with product Shipping(~500ms). The results are shown in Figure 17a. Both Knative and OpenWhisk exhibit significant overhead latencies, with cascading cold start latency being 520% and 130% of the end to end execution latency. Xanadu improves the overhead latency to 70% of end to end execution latency, which is a considerable improvement compared to the baselines.

5.6.2 Image Processing Pipeline. Data and signal processing pipelines are ideal candidates for explicit workflows owing to their pre-structured control flows. We implement a five function image processing pipeline using the JIMP [14] image processing library. The pipeline accepts an image file, Scales(~400ms) it, adjusts its Contrast(~350ms), Rotates(~600ms) it by 180°, Blurs (~500ms) and converts it into Grayscale(~300ms). It uses an FTP server to store intermediate stage results running locally on the same machine. Figure 17b shows the execution result of the pipeline. As expected, cascading cold starts dominates the pipeline execution latency. Xanadu again shows significant performance improvements with



(a) E-commerce Website

(b) Image Processing Pipeline

Figure 17. Performance analysis of different Xanadu modes against Knative and Openwhisk for an E-commerce and an image processing pipeline.

overhead latencies reducing by 5x and 2x compared to Knative and OpenWhisk, respectively.

6 Related Work

Workflow based Serverless Applications. Serverless computing has grabbed considerable attention in different scenarios, with its promise of bringing low-cost computation [62]. Active research is being done to find novel applications of this new paradigm in diverse fields like IoT and industrial processes [21, 35, 38], scientific workflows [31, 37, 49, 60, 65]. Extensive use of workflow-based function interactions is a recurrent theme in almost all solutions. Tools like [40, 43, 56, 59] have been designed to aid in developing highly parallel applications as serverless workflows. Solutions have been developed to enable machine learning using serverless computing [29, 30, 33, 39, 61, 63] and exploit parallelism in serverless computing to enable video analytics [22]. All of these solutions show that both implicit and explicit serverless workflows are essential to present generation of hybrid solutions, making it critical to mitigate cascading cold start latency.

Lopez et al. [47] compared function workflow orchestration services of Amazon, IBM and Microsoft under different loads; however, their experiments were conducted under warm start conditions disregarding the effects of cascading cold starts. [24] reported exponential growth in overhead latency for AWS Step Functions, while IBM's composer had a more linear growth in latency. However, the reports lacked information regarding cascading cold start latency.

Managing Cold start in Serverless Computing. Notable work has gone into mitigating side effects of cold start latency, with solutions looking into possible bottlenecks in the cold start pipeline and ways to reduce them. Mohan et al. [52] removed bottlenecks in the networking namespace construction pipeline using Pause Container pools, while Oakes et al. [53] looked into ways for sharing user-level libraries. [54] formulated lean containers for faster startup and enabled library sharing utilising a Zygote based library management system. There have also been some scheduler level optimisations[17, 48] enabling better function to host mapping for reduced cold starts, using package affinity and function runtime profiles as scheduling criteria. However, almost none of the solutions address the problem of cascading cold start latency, which is crucial because even if individual cold start latency is reduced, they can quickly aggregate to be a significant fraction of a function's workflow.

Managing Cascading Cold starts in Function Chains. Akkus et al. [20] mentions that existing platforms manage components of a function chain in isolation from one another, which leads to significant latencies since they must traverse the entire end to end call graph. Their solution, SAND employs multi-level fault isolation, with applications running in separate sandboxes while functions belonging to the same application share the same isolation sandboxes. SAND is based around the assumption that functions comprising an application can be trusted to co-exist. Nevertheless, this assumption can be hazardous for co-tenants, where an attack on one of the functions can compromise other functions in the workflow housed inside the same sandbox. Xanadu makes no such assumptions about the security model of functions, providing optimisations in the temporal domain as opposed to optimisations in the spatial domain provided by SAND. Bermbach[26] et al. discusses a cascading cold start mitigation approach while maintaining functions as a black box. An external Hint Manager prevents cascading cold starts by pre-starting workers using hints regarding possible invocations. However, the applicability of such a black-box approach is limited to linear chains. Xanadu , on the other hand, uses function runtime information to prevent cascading cold starts in both linear and conditional chains. Lin et al. [45] suggest the use of pre-crafted worker pools as a measure to reduce cold start latency. Even though this can be used to mitigate cascading cold starts, the overhead running costs of a long-running pool can be significant. Kijak et al[42] uses a Deadline-Budge heuristic-based scheduling algorithm to schedule DAG serverless workflows. Instead, Xanadu proposes a Just-in-Time deployment procedure to reduce cascading cold starts without paying for runtime overheads.

7 Future Work

- **Reducing prediction miss penalties**
  Currently, prediction miss penalty mitigation is limited to stopping the Speculation process. In future, Prediction misses would cause the MLP path to be re-evaluated and functions on the new path to be deployed speculatively based on the updated MLP. Also, since individual workers may already have been deployed ahead-of-time, they can be reused by functions in the alternate branch, provided they are of similar architectures.
- **Reducing Function keep-alive time**
  Xanadu's Speculative deployment prevents a significant amount of cold starts. This eliminates the need for workers with long keepalive period. As part of future work, we plan to take advantage of this to reduce the Keepalive time of workers from tens of minutes to a few seconds, enabling us more significant resource savings.
- **Integrating Xanadu with other platforms**
  Xanadu uses function profiling to enable speculation; thus, the complexity of integrating with commercial platforms would depend on the availability of such data. We are working on integrating the Speculation engine with open-source platforms like OpenWhisk along with the addition of necessary profilers.

8 Conclusion

General-purpose workflow support for serverless platforms presents unique opportunities and challenges for performance, resource utilisation and cost. We delved into the challenges of cold start aggregation in function chains and introduced Xanadu, a speculative resource provisioning based serverless platform to eliminate cascading cold starts. Combined with our implicit chain detection mechanism, Xanadu reduces overhead latencies even in standalone correlated functions. The comparison with Knative and OpenWhisk shows that Xanadu provides significant performance improvements, limiting cascading cold starts to a single event while avoiding cost overruns usually associated with resource pre-deployments.

Acknowledgments

We thank the reviewers, our shepherd Mina Sedaghat, and our colleagues Nikhila Somu and Naman Dixit for their helpful comments. This work was partially supported by VMware Inc. via a research grant (grant number 17DON003).

References

[1] 2020. Amazon States Language. https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html. Accessed: 2020-04-12.

[2] 2020. Apache CouchDB. https://couchdb.apache.org/. Accessed: 2020-04-22.

[3] 2020. Apache JMeter. https://jmeter.apache.org/. Accessed: 2020-05-27.

[4] 2020. Apache Kafka. https://kafka.apache.org/. Accessed: 2020-04-12.

[5] 2020. Apache OpenWhisk. https://openwhisk.apache.org/. Accessed: 2020-04-12.

[6] 2020. AWS Lambda. https://aws.amazon.com/lambda/. Accessed: 2020-04-11.

[7] 2020. AWS Step Functions. https://aws.amazon.com/step-functions/. Accessed: 2020-04-11.

[8] 2020. Azure Durable Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview. Accessed: 2020-04-11.

[9] 2020. Azure Functions. https://azure.microsoft.com/en-in/services/functions/. Accessed: 2020-04-11.

[10] 2020. Cloudflare Workers. https://workers.cloudflare.com/. Accessed: 2020-04-12.

[11] 2020. Docker Container. https://www.docker.com/. Accessed: 2020-05-08.

[12] 2020. Google Cloud Functions. https://cloud.google.com/functions. Accessed: 2020-04-11.

[13] 2020. IBM Cloud Functions Composer. https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-pkg_composer. Accessed: 2020-04-22.

[14] 2020. JavaScript Image Manipulation Program. https://www.npmjs.com/package/jimp. Accessed: 2020-05-27.

[15] 2020. Javascript V8 Engine. https://v8.dev/. Accessed: 2020-04-12.

[16] 2020. Knative. https://knative.dev/. Accessed: 2020-04-22.

[17] Cristina L Abad, Edwin F Boza, and Erwin Van Eyk. 2018. Package-aware scheduling of FaaS functions. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 101–106.

[18] Gojko Adzic and Robert Chatley. 2017. Serverless computing: economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 884–889.

[19] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 419–434.

[20] 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 *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 923–935.

[21] Eyhab Al-Masri, Ibrahim Diabate, Richa Jain, Ming Hoi Lam Lam, and Swetha Reddy Nathala. 2018. A Serverless IoT Architecture for Smart Waste Management Systems. In *2018 IEEE International Conference on Industrial Internet (ICII)*. IEEE, 179–180.

[22] Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter. 2018. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*. 263–274.

[23] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. 2017. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer, 1–20.

[24] Daniel Barcelona-Pons, Pedro García-López, Álvaro Ruiz, Amanda Gómez-Gómez, Gerard París, and Marc Sánchez-Artigas. 2019. FaaS Orchestration of Parallel Workloads. In *Proceedings of the 5th International Workshop on Serverless Computing*. 25–30.

[25] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. 2019. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the 20th International Middleware Conference*. 41–54.

[26] David Bermbach, Ahmet-Serdar Karakaya, and Simon Buchholz. 2020. Using application knowledge to reduce cold starts in FaaS services. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. 134–143.

[27] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. 2018. Putting the" Micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 645–650.

[28] Tyler Caraza-Harter and Michael M Swift. 2020. Blending containers and virtual machines: a study of firecracker and gVisor. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 101–113.

[29] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2018. A case for serverless machine learning. In *Workshop on Systems for ML and Open Source Software at NeurIPS*, Vol. 2018.

[30] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: a Serverless Framework for End-to-end ML Workflows. In *Proceedings of the ACM Symposium on Cloud Computing*. 13–24.

[31] Ryan Chard, Tyler J Skluzacek, Zhuozhao Li, Yadu Babuji, Anna Woodard, Ben Blaiszik, Steven Tuecke, Ian Foster, and Kyle Chard. 2019. Serverless Supercomputing: High Performance Function as a Service for Science. *arXiv preprint arXiv:1908.04907* (2019).

[32] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 467–481.

[33] Lang Feng, Prabhakar Kudva, Dilma Da Silva, and Jiang Hu. 2018. Exploring serverless computing for neural network training. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 334–341.

[34] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.

[35] Xinzhou Geng, Oiuzhe Ma, Yunman Pei, Zhibo Xu, Wenjing Zeng, and Jing Zou. 2018. Research on early warning system of power network overloading under serverless architecture. In *2018 2nd IEEE Conference on Energy Internet and Energy System Integration (EI2)*. IEEE, 1–6.

[36] Rong Gu, Yun Tang, Chen Tian, Hucheng Zhou, Guanru Li, Xudong Zheng, and Yihua Huang. 2017. Improving execution concurrency of large-scale matrix multiplication on distributed data-parallel platforms. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2539–2552.

[37] Ling-Hong Hung, Dimitar Kumanov, Xingzhi Niu, Wes Lloyd, and Ka Yee Yeung. 2019. Rapid RNA sequencing data analysis using serverless computing. *bioRxiv* (2019), 576199.

[38] Razin Farhan Hussain, Mohsen Amini Salehi, and Omid Semiari. 2019. Serverless edge computing for green oil and gas industry. In *2019 IEEE Green Technologies Conference (GreenTech)*. IEEE, 1–4.

[39] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2018. Serving deep learning models in a serverless platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 257–262.

[40] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*. 445–451.

[41] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[42] Joanna Kijak, Piotr Martyna, Maciej Pawlik, Bartosz Balis, and Maciej Malawski. 2018. Challenges for scheduling scientific workflows on cloud functions. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 460–467.

[43] Youngbin Kim and Jimmy Lin. 2018. Serverless data analytics with flint. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 451–455.

[44] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding ephemeral storage for serverless analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC '18)*. 789–794.

[45] Ping-Min Lin and Alex Glikson. 2019. Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach. *arXiv preprint arXiv:1903.12221* (2019).

[46] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. 2018. Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 159–169.

[47] Pedro García López, Marc Sánchez Artigas, G Parıs, Daniel Barcelona Pons, AR Ollobarren, and David Arroyo Pinto. 2018. Comparison of production serverless function orchestration systems. *CoRR, vol. abs/1807.11248* (2018).

[48] Nima Mahmoudi, Changyuan Lin, Hamzeh Khazaei, and Marin Litoiu. 2019. Optimizing serverless computing: introducing an adaptive function placement algorithm. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*. 203–213.

[49] Maciej Malawski. 2016. Towards Serverless Execution of Scientific Workflows-HyperFlow Case Study.. In *Works@ Sc*. 25–33.

[50] Johannes Manner, Martin Endreß, Tobias Heckel, and Guido Wirtz. 2018. Cold start influencing factors in function as a service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 181–188.

[51] Garrett McGrath and Paul R Brenner. 2017. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 405–410.

[52] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile cold starts for scalable serverless. In *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*.

[53] Edward Oakes, Leon Yang, Kevin Houck, Tyler Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Pipsqueak: Lean lambdas with large libraries. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 395–400.

[54] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. *SOCK*: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC'18)*. 57–70.

[55] Zhengping Qian, Xiuwei Chen, Nanxi Kang, Mingcheng Chen, Yuan Yu, Thomas Moscibroda, and Zheng Zhang. 2012. MadLINQ: large-scale distributed matrix computation for the cloud. In *Proceedings of the 7th ACM european conference on*

Computer Systems. 197–210.

[56] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. 2018. Serverless data analytics in the ibm cloud. In *Proceedings of the 19th International Middleware Conference Industry*. 1–8.

[57] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. 2020. Serverless Computing: A Survey of Opportunities, Challenges and Applications. (2020).

[58] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[59] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. 2018. Numpywren: Serverless linear algebra. *arXiv preprint arXiv:1810.09679* (2018).

[60] Josef Spillner, Cristian Mateos, and David A Monge. 2017. Faaster, better, cheaper: The prospect of serverless scientific computing and hpc. In *Latin American High Performance Computing Conference*. Springer, 154–168.

[61] Zhucheng Tu, Mengping Li, and Jimmy Lin. 2018. Pay-per-request deployment of neural network models using serverless architectures. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*. 6–10.

[62] Mario Villamizar, Oscar Garces, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, et al. 2016. Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 179–182.

[63] Hao Wang, Di Niu, and Baochun Li. 2019. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 1288–1296.

[64] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 133–146.

[65] Sebastian Werner, Jörn Kuhlenkamp, Markus Klems, Johannes Müller, and Stefan Tai. 2018. Serverless Big Data Processing using Matrix Multiplication as Example. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 358–365.

[66] Miao Zhang, Yifei Zhu, Cong Zhang, and Jiangchuan Liu. 2019. Video processing with serverless computing: a measurement study. In *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*. 61–66.