Courier: Delivering Serverless Functions Within Heterogeneous FaaS Deployments

Anshul Jindal
Technical University of Munich
Garching (near Munich), Bavaria, Germany
anshul.jindal@tum.de

Julian Frielinghaus
Technical University of Munich
Garching (near Munich), Bavaria, Germany
julian.frielinghaus@tum.de

Mohak Chadha
Technical University of Munich
Garching (near Munich), Bavaria, Germany
mohak.chadha@tum.de

Michael Gerndt
Technical University of Munich
Garching (near Munich), Bavaria, Germany
gerndt@in.tum.de

ABSTRACT

With the advent of serverless computing in different domains, there is a growing need for dynamic adaption to handle diverse and heterogeneous functions. However, serverless computing is currently limited to homogeneous Function-as-a-Service (FaaS) deployments or simply FaaS Deployment (FaaSD) consisting of deployments of serverless functions using a FaaS platform in a region with certain memory configurations. Extending serverless computing to support Heterogeneous FaaS Deployments (HeteroFaaSDs) consisting of multiple FaaSDs with variable configurations (FaaS platform, region, and memory) and dynamically load balancing the invocations of the functions across these FaaSDs within a HeteroFaaSD can provide an optimal way for handling such serverless functions.

In this paper, we present a software system called **Courier** that is responsible for optimally distributing the invocations of the functions (called delivering of serverless functions) within the HeteroFaaSDs based on the execution time of the functions on the FaaSDs comprising the HeteroFaaSDs. To this end, we developed two approaches: Auto Weighted Round-Robin (AWRR) and Per-Function Auto Weighted Round-Robin (PFAWRR) that use functions execution times for delivering serverless functions within a HeteroFaaSD to reduce the overall execution time. We demonstrate and evaluate the functioning of our developed tool on three HeteroFaaSDs using three FaaS platforms: 1) on-premise OpenWhisk, 2) AWS Lambda, and 3) Google Cloud Functions (GCF). We show that **Courier** can improve the overall performance of the invocations of the functions within a HeteroFaaSD as compared to traditional load balancing algorithms.

CCS CONCEPTS

• **Computer systems organization** → *Cloud computing*.

KEYWORDS

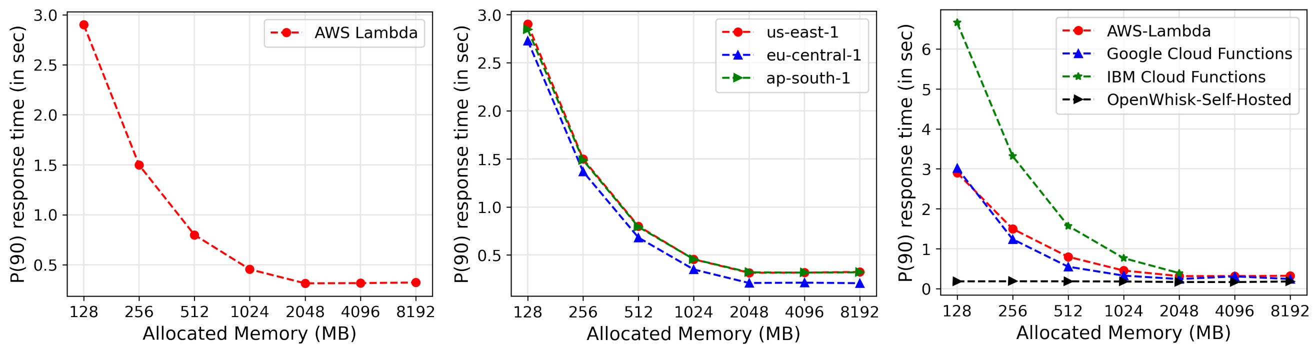
serverless computing, Function-as-a-Service, functions delivery

ACM Reference Format:
Anshul Jindal, Julian Frielinghaus, Mohak Chadha, and Michael Gerndt. 2021. Courier: Delivering Serverless Functions Within Heterogeneous FaaS Deployments. In *2021 IEEE/ACM 14th International Conference on Utility and Cloud Computing (UCC'21), December 6–9, 2021, Leicester, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Function-as-a-Service (FaaS) and serverless computing, in general, have received a growing interest since the launch of AWS Lambda to the general public in 2014 [5]. Developers benefit from various aspects of FaaS computing such as no infrastructure management, automatic scalability, and faster deployments [14]. From an economic point of view, FaaS can reduce the cost of operation due to fine-grained on-demand automatic scaling. Additionally, the lack of server management can decrease the time-to-market for an application [33]. The FaaS paradigm can be used for building a myriad of applications such as web applications, IoT, BigData workloads, Chatbots and Amazon Alexa, as well as IT Automation [10, 16].

Currently, homogeneous FaaS deployments dominate the landscape of serverless computing. We refer to a *Homogeneous FaaS Deployment* or simply *FaaS Deployment (FaaSD)* as the deployment of serverless functions on a FaaS platform (e.g. AWS Lambda, Google Cloud Functions (GCF) or OpenWhisk) with a certain memory configuration in a specific region. The FaaS platform is responsible for providing resources for function invocations and performs automatic scaling. This is done by creating an execution environment that provides a secure and isolated runtime for the function. The amount of resources for an execution environment are typically decided based on the maximum amount of memory and execution time (timeout) statically specified by the user on function creation [17]. The amount of memory configured is important since some commercial FaaS providers increase the amount of compute available to the function when more memory is assigned [11, 24]. If a function invocation violates these constraints, the FaaS platform immediately terminates the invocation. Therefore, a function invocation might get prematurely terminated if it requires high computing power and is executed in an execution environment with low compute capabilities. However, the resource (storage, memory,



(a) Memory-based **(b) Region-based** **(c) Platform-based**

Figure 1: Three different ways of achieving Heterogeneous FaaS Deployment (HeteroFaaSD).

compute, and network) requirements for serverless functions are very dynamic and can differ vastly [26].

Heterogeneous FaaS Deployment (HeteroFaaSD) is the deployment that consists of multiple FaaSDs with each of these deployments differing from another on a particular configuration parameter (FaaS platform, memory, or region) and keeping the other parameters fixed. In the following paragraphs, we highlight different ways by which HeteroFaaSD can be achieved by using a serverless function (deployed with the initial configuration of 512MB memory in `Europe-Central-1` region on AWS Lambda):

Memory-based HeteroFaaSD: A HeteroFaaSD where only the memory configuration is changed between the FaaSDs making it. Figure 1a shows the 90^{th} percentile execution time of the function when deployed with different memory configurations on AWS Lambda. It first decreases with the increase in memory, and after a point (2048MB), it becomes constant. Therefore, dynamically using a lower memory FaaSD within a *Memory-based HeteroFaaSD* based on the workload may reduce the overall cost at a similar Service-Level Objective (SLO) [4].

Region-based HeteroFaaSD: Here the FaaSDs forming the HeteroFaaSD are deployed in the different regions. Figure 1b shows the variations in the 90^{th} percentile execution time of the same function when deployed in three different regions on AWS Lambda. Since executing functions geographically near the end-user would reduce bandwidth delays [6], we can observe from Figure 1b that the function executing in a FaaSD closer to the client (Europe-Central-1 region) takes less time as compared to the other two FaaSDs.

Platform-based HeteroFaaSD: Here the FaaSDs forming the HeteroFaaSD are deployed using different FaaS platforms. These different platforms can be commercial offerings or private self-hosted open-source FaaS platforms. Figure 1c shows the variations in the 90^{th} percentile execution time of the same function when deployed with three public providers: AWS Lambda, GCF, and IBM Cloud Functions, and one private self-hosted OpenWhisk. In OpenWhisk, compute resources are not proportional to the amount of memory configured. Hence, it can process the same workload much faster than others at lower memory configurations. However, after 2048MB memory, all FaaSDs behave the same. Including non-public FaaSDs can help with data protection and privacy laws and can also mitigate the effects of vendor lock-in [30].

The examples above highlight three main ways to form a HeteroFaaSD and show the different advantages offered by them. Since Function-as-a-Service (FaaS) workloads can incorporate several different languages and SDKs, these workloads can benefit from different aspects of the underlying FaaSDs in a HeteroFaaSD. For instance, workloads that are heavily based on video processing can operate more efficiently if the executing FaaSD has a GPGPU, or running the functions in a FaaSD closer to the user would reduce the latency. Hence, distributing serverless functions invocations (referred to delivering of serverless functions in this paper) across FaaSDs within a HeteroFaaSD can provide an optimal way for handling serverless functions with high dynamism and heterogeneity.

Towards this, we develop **Courier** a tool for delivering serverless functions across FaaSDs within a HeteroFaaSD. Our key contributions are:

- We develop and present a novel system called **Courier** that can deploy and orchestrate serverless functions within HeteroFaaSD(s) based on execution time. To the best of our knowledge, this is the first work that extends serverless computing to multiple FaaS platforms using HeteroFaaSDs.
- We propose and implement two function delivering algorithms: 1) Auto Weighted Round-Robin (AWRR), and 2) Per-Function Auto Weighted Round-Robin (PFAWRR) to reduce the overall execution times of the invocations of the functions without requiring manual intervention of developers or clients.
- We evaluate our developed tool and algorithms against two frequently used algorithms (Round-Robin (RR), and Weighted Round-Robin (WRR)) by creating three HeteroFaaSDs using different combinations of three FaaS platforms: 1) AWS Lambda, 2) GCF and 3) Private self-hosted OpenWhisk.

Paper organization. Section 2 describes some of the prior works in this domain. Section 3 gives a general overview of the existing load balancing algorithms and describes the two proposed algorithms. In Section 4, the system architecture of **Courier** and its components are explained. In Section 5, we describe the performance evaluation setup along with the various benchmarks. In

Section 6, performance evaluation results are presented and Section 7 summarizes the discussion of those results. Finally, Section 8 concludes the paper and presents an outlook.

2 RELATED WORK

Connecting multiple cloud services has been actively researched in the community. We present previous works from two aspects:

2.1 Connecting multiple cloud platforms

Brogi et al. present a software system called SeaClouds that tries to simplify distribution, monitoring, and migration of Platform-as-a-Service (PaaS) software across multiple heterogeneous platforms [7]. The SeaClouds system takes the approach of deploying the different modules to the optimal platform, i.e., the platform that fulfills the requirements of the specific module. Additionally, it monitors the platform to ensure that it meets the requirements in the future. Our system consists of similar components like SeaClouds. However, due to the nature of FaaS, we developed additional components like a load balancer.

Apache Brooklyn is software to control deployment, monitoring, and management of applications in cloud environments [8]. It works by connecting different APIs and SDKs to provide a single software interface. It can conduct complex actions such as deploying a new webserver instance and configuring the load balancer afterward. The developer can specify such actions via pre-defined policies and rules described in so-called *blueprints* in YAML syntax. Unfortunately, Apache Brooklyn has no direct support for FaaS.

2.2 Connecting multiple FaaS platforms

Aske et al. present a software system that helps developers define custom scheduling strategies [1]. In their work, they connected two commercial FaaS offerings (AWS Lambda and IBM Bluemix) and one local OpenWhisk cluster to their service. They implemented a low-latency scheduling algorithm that forwards the requests to the cluster with the lowest Round Trip Time (RTT). Based on that algorithm, they can reduce the overall computation time drastically. However, their system does not distinguish between homogeneous or heterogeneous deployments. We use a similar approach to leverage the benefits of HeteroFaaSDs by developing a scheduling algorithm for delivering the serverless functions.

In our previous research, we present the concept of a Function Delivery Network (FDN) consisting of a network of multiple heterogeneous target platforms orchestrated by a control plane capable of placing functions into several FaaS platforms [20, 21]. The FDN allows combining FaaS platforms with different software and hardware characteristics. Doing so allows to reduce the overall energy consumption and provide better response times and optimized data placement. To the best of our knowledge, there has been no implementation of FDN in literature. In this work, we present **Courier** which is the first implementation of FDN but with higher granularity by using HeteroFaaSDs incorporating heterogeneous platforms as one of its subset under *Platform-based HeteroFaaSD.*

Google Cloud Platform (GCP) has introduced load balancing of user requests to a serverless network endpoint group (NEG) that consists of a Cloud Run, App Engine, or Cloud Functions service [32]. The load balancer serves as the frontend and proxies traffic to the specified serverless endpoint in this service. If the backend service contains multiple serverless NEGs, the load balancer balances traffic between these NEGs, thus minimizing request latency. However, serverless NEGs can point only to Cloud Functions residing in the same region where the NEG is created, and it is only restricted to their infrastructure. This is not the case with our implementation; **Courier** can work with multiple regions and multiple public cloud providers.

Furthermore, the load balancer to serverless NEG cannot detect if the underlying serverless resource (such as an App Engine, Cloud Functions, or Cloud Run (fully managed) service) is working as expected. This means that if a function deployed on GCF in one region is returning errors, but the overall infrastructure is operating normally, then the load balancer will not automatically direct traffic away to other regions. This is mitigated in our implementation by redirecting the traffic to other FaaSDs in different regions depending on their response times. Additionally, a serverless NEG can only represent a group of services sharing the same URL pattern. This will not work if we use multiple platforms, whereas our developed system, **Courier** can work with multiple URL patterns.

3 FUNCTIONS DELIVERING ALGORITHMS

Since delivering of functions can be at the primary level compared with load balancing functions invocations problem. Therefore, we first describe the existing approaches that address the load balancing problem. Following this, we present our proposed algorithms.

3.1 Existing Approaches

Existing approaches can be categorized into two categories:

Static load balancing algorithms derive their decisions based on pre-defined parameters which do not get updated. Static algorithms offer low computational overhead with decent results [35]. However, most static algorithms do not include overload protection [13]. Popular load balancers such as the AWS Elastic Load Balancer [3] and the NGINX load balancer rely on static algorithms [2, 29]. The most prominent static algorithms are the Round-Robin (RR) and Weighted Round-Robin (WRR) algorithm. Round-Robin (RR) distributes the requests equally to all available targets (in our case FaaSDs within a HeteroFaaSD) whereas Weighted Round-Robin (WRR) assigns requests to targets in a rotating manner by respecting pre-defined weights for each target. Other examples are *Min-Min* or *Max-Min* algorithms. These algorithms assign application tasks with the shortest (or longest) execution time to the target with minimal load. In general, static load balancing algorithms have the disadvantage that they do not directly react to changes in the runtime behavior of functions or the load on the target [13].

Dynamic algorithms take the state of the target systems into account when scheduling [13]. By respecting the system state, the algorithms try to prevent the overloading of the target systems. This comes at the cost of increased overhead and network communication [38]. *Least-connection*-based algorithms count the number of open connections to a target. They then try to keep these in balance (e.g., according to a pre-defined weight in the *Weighted Least Connection* algorithm [38]). Another dynamic approach is the prediction of the future workload, as shown by Lavanya et al. [25].

Algorithm 1: AWRR Algorithm

```
Input: $avg\_exec\_times$ = [], $max\_sum\_weights$, number of
       FaaSDs $D$
Output: $W$ = [] weights for each FaaSD
min_weight = 1, weights_sum = 0;
W = [1,1,..1] ;        // equal weights for each FaaSD
$max\_exec\_time$ = Max($avg\_exec\_times$)
for $i \in D$ do
    $t_i = avg\_exec\_times_i$ ;
    $w_i = \frac{max\_exec\_time}{t_i}$;
    $weights\_sum = weights\_sum + w_i$;
end
for $i \in D$ do
    $w_i$ = max(floor(($\frac{w_i}{weights\_sum}$) $\times$
     $max\_sum\_weights$), $min\_weight$)
end
return $W$
```

This prediction can also help to improve the overall energy efficiency as unused machines could be turned off. Tong et al. present an algorithm that calculates the *residual load rate* of each server [38]. This rate indicates the remaining capacity of that specific target. The algorithm groups targets with similar *residual load rates* and distributes the requests in a WRR fashion between them.

3.2 Our Approach

For our algorithm design, we combine a static algorithm with the aspects of a dynamic algorithm. We chose the Weighted Round-Robin (WRR) algorithm as it offers better performance than *Min-Min*-based algorithms while having a very low overhead [35]. We do not consider *Least Connection*-based algorithms, as they have a significantly higher overhead. In the following subsections, we provide more details on the two designed algorithms.

3.2.1 *Auto Weighted Round-Robin (AWRR).*

The pseudocode for the AWRR algorithm is shown in Algorithm 1. The algorithm represents a dynamic version of a WRR algorithm as it adapts its weights according to the functions execution time in the target FaaSDs within a HeteroFaaSD. It initially assigns equal weights to all the target FaaSDs and periodically updates them to reflect changes in the target FaaSDs. The average execution time (measured in milliseconds) of the functions within a FaaSD is used as the main metric for the weight estimation. We use this metric to prevent overloading of the target FaaSDs similar to the *Least Connection*-based algorithms. The algorithm builds a mean value for the metric across a specific time delta δ for each FaaSD within a HeteroFaaSD. It then determines the maximum execution time from the average execution times of FaaSDs (Line 3). Based on this, it calculates the weight of each FaaSD by dividing the maximum execution time by the FaaSD's average execution time (Line 5-7). Then, the weights of each FaaSD are normalized by using a maximum sum weight provided by the user (Line 9-11).

Algorithm 2: PFAWRR Algorithm

```
Input: $avg\_func\_exec\_times$ = [[]], $max\_sum\_weights$,
       number of functions $F$, number of FaaSDs $D$
Output: $W$ = [[]] weights per function per FaaSD
min_weight = 1;
$W$ = [[]] ;  // a $F \times D$ 2-d matrix initialized to 1
for $f \in F$ do
    weights_sum = 0 ;
    $max\_time$ = Max($avg\_func\_exec\_times[f][:]$) for
     $i \in D$ do
        $t_{fi} = avg\_func\_exec\_times_{fi}$;
        $W_{fi} = \frac{max\_time}{t_{fi}}$;
        $weights\_sum = weights\_sum + W_{fi}$;
    end
    for $i \in D$ do
        $w_{fi}$ = max(floor(($\frac{W_{fi}}{weights\_sum}$) $\times$
         $max\_sum\_weights$), $min\_weight$)
    end
end
return $W$
```

3.2.2 *Per-Function Auto Weighted Round-Robin (PFAWRR).*

The second algorithm, PFAWRR, consists of $F+1$ instances of AWRR balancers, where F is the number of known functions that have been executed within a specific time delta δ. These instances take over the functions delivering decisions if a client requests the respective function. The computation of the weights for these individual balancing instances per function is computed in the same way as in the AWRR algorithm, and its pseudocode is shown in Algorithm 2. In this case, we use the average execution time per function within each FaaSD ($avg\_func\_exec\_times$). This is a 2-dimensional matrix with size $F \times D$, where D is the number of FaaSDs within HeteroFaaSD. In the first step, the weight matrix W of the same size as $avg\_func\_exec\_times$ is initialized to 1 (Line 2). Then for each function, FaaSDs weights are calculated in a similar manner as in the AWRR algorithm (Line 3-12). In the first step, we find the maximum execution time for a function in all the FaaSDs (Line 5). We calculate the weight of each FaaSD for that particular function by dividing the maximum execution time for the function by the time taken within that FaaSD (Line 6-9). Afterward, weights are normalized (Line 10-12). This procedure is repeated for all the functions, and the 2-D weight array is returned.

This algorithm is designed to leverage the heterogeneity of FaaS functions for different FaaSDs and is designed to find optimal FaaSD within a HeteroFaaSD, i.e. the FaaSD with the shortest execution time, for each function invocation.

Both algorithms try to avoid overloading FaaSDs within a HeteroFaaSD by estimating the weights as accurately as possible. For handling overload situations, we have also implemented a component called *circuit breaker* [15] as part of **Courier**.

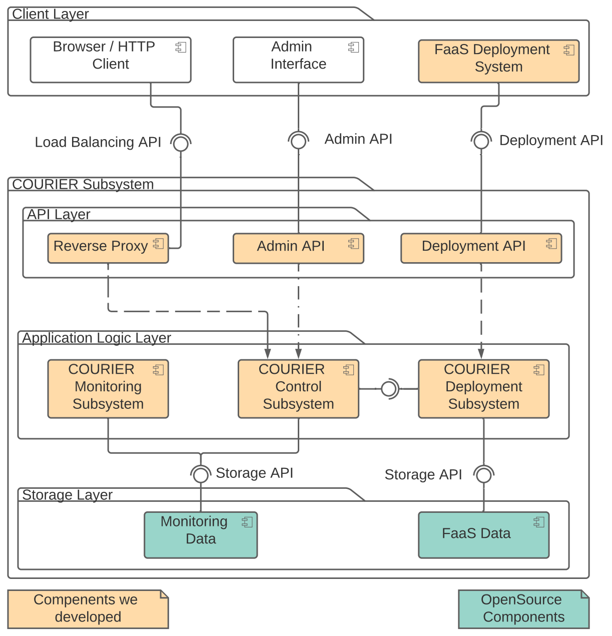


Figure 2: Courier overall subsystem decomposition.

4 SYSTEM ARCHITECTURE

In this section, we give a high-level overview of the general architecture of the **Courier** system. It uses the functions delivering algorithms (§3.2) to achieve its primary goal: efficiently delivering serverless functions within a HeteroFaaSD. It also handles the deployment of the functions. Based on the principle to "keep functionally related objects together" [9], we created a 4-layer overview of the different subsystems shown in Figure 2 and described next.

4.1 Client Layer

The Client Layer offers three perspectives to distinguish between three types of clients: developers, administrators, and client applications that invoke the FaaS functions. Developers use the deployment API of **Courier** to register and deploy new functions into the system. Administrators can decide the functions delivering algorithm via the Admin API. The client applications send requests to the **Courier** system using the functions delivering API, and it delivers the functions based on the chosen algorithm.

4.2 API Layer

It handles the delivery of functions for clients with a reverse proxy. This allows the system to detect unavailable clusters without additional network overhead. Client applications (like web browsers) use the HTTP Reverse Proxy component to invoke functions. The Reverse Proxy uses the *Control Subsystem* to decide which FaaSD within a HeteroFaaSD should execute the function and forwards the network communication to the client and the invoking target FaaSD. Configuration of the **Courier** system happens via the Admin API. Administrators use the Admin API to set the functions delivering strategy. This can be: 1) Direct FaaSD strategy where the delivery of functions happens to only the specified FaaSD within a HeteroFaaSD, 2) RR, 3) WRR with manually set weights, 4) AWRR, and 5) PFAWRR. The *Deployment Subsystem* offers deployment information to the developer through the Deployment API.

4.3 Application Logic Layer

It handles the actual invocation of the different requests. It operates across the following three independent services:

4.3.1 Control Subsystem. This is the heart of the **Courier** system. It implements the functions delivering strategies and decides where the functions should be invoked. It also can trigger the deployment of a function, e.g., if it detects that a function is not available on one of the targets FaaSDs. It then advises the *Deployment Subsystem* to handle the deployment. We implemented the *Control Subsystem* in Java. It is called *Controlplane*. We chose Jetty [36] as the HTTP Server since it is a stable project with an existing reverse proxy implementation that can be extended easily. The build system produces one single JAR file as an artifact.

4.3.2 Monitoring Subsystem. It periodically collects available performance data metrics for the functions delivering algorithms from all FaaSDs within a HeteroFaaSD and processes them if necessary. Performance data metrics include:

- *active_instances*: The number of active function instances.
- *invocations*: Number of times the function is executed.
- *execution_time*: The time spent by function in processing.
- *memory_usage*: Function's memory usage during execution.
- *allocated_memory*: Memory allocated to the function.

It uses the *Monitoring Data* database to store the metrics. Its subcomponent, the *Syncworker* periodically gets a list of available FaaSDs and starts the individual data collection, and uses the so-called data writers to store the performance data.

4.3.3 Deployment Subsystem. It handles the deployment of a FaaS function to the different FaaSDs within a HeteroFaaSD. It gets invoked by the *Control Subsystem*. It uses the *FaaS Data* storage to retrieve the function source code and the deployment specification. We implemented this using Node.js. It is called the *Deployment Agent*. As the *serverless framework* [37] (which we use for deployment) is written in Node.js, we chose Node.js to connect with the framework directly.

4.4 Storage Layer

It provides services to store persistent data. The **Courier** system stores monitoring data and FaaS functions persistently. Persistent storage keeps the performance stable in case of restarts. We use InfluxDB, a time-series database [18] as the *Monitoring Data* database and a MinIO object storage [27] to handle the FaaS data.

5 PERFORMANCE EVALUATION

We first introduce the different benchmarks, i.e., FaaS functions, along with the developed application that we use to evaluate our system in §5.1. Following this, we describe the testing process and evaluation infrastructure in §5.2 and §5.3 respectively.

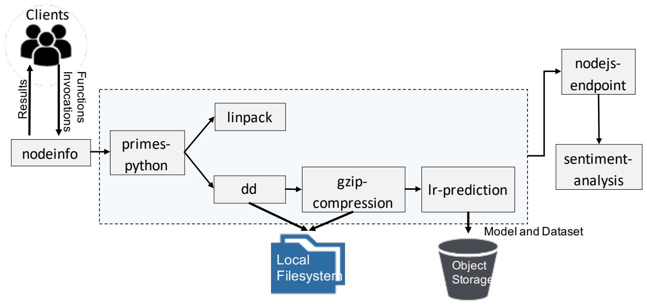


Figure 3: High level workflow of the evaluated application.

5.1 Benchmarks

To investigate the performance of **Courier**, we used a subset of the microbenchmarks provided with the FaaSProfiler [34]. Also, we created an application with those microbenchmarks shown in Figure 3 for our use case. The microbenchmarks used are summarized in Table 1 along with their description and language runtimes.

The application flow starts with the `nodeinfo` function, which exposes an HTTP endpoint and provides the user with basic information about the system such as hostname, underlying architecture, number of CPUs, etc. This then invokes the compute-intensive `primes-python` function, which further invokes `linpack` and `dd` asynchronously, and waits for their response to come back. `dd` invokes `gzip-compression`, which further invokes `lr-prediction` in a sequence. `lr-prediction` queries the model and data from the google cloud storage (created in GCP in the `Europe-west3` region, AWS Lambda functions also use this storage bucket) and then performs prediction. Once the response is available to the `primes-python` function from both invocations, it sends back the response to the `nodeinfo` function, which invokes the `nodejs-endpoint` function. Following this, `nodejs-endpoint` invokes the `sentiment-analysis` function.

5.2 Testing Process

The testing process consist of three phases as follows:

Pre-warm phase: We pre-warm the FaaSDs within a HeteroFaaSD to reduce the effect of cold starts. It is executed for a minute.

Testing phase: In this phase, we execute the test against one functions delivering strategy (§4.2). This phase is run for 10 minutes.

Cooldown phase: Here the K6 engine finishes open computations and free worker resources. It is executed for 1 minute.

To unify the phase execution, we developed a testing system via several bash scripts. These scripts use the K6 [22] load testing framework. We collect several metrics automatically reported by K6 [23] during load testing.

To evaluate the performance of **Courier** in different scenarios, we define two different load tests. Each test is executed via multiple so-called Virtual Users (VUs). VUs are the entities in K6 that execute the test and make HTTP(s) requests.

T1 **One function test**: Here, *nodeinfo* function is invoked from `20` VUs with each VU executing `750` invocations for the test duration and results in a total of about `15000` invocations.

T2 **Application test**: In this test, created serverless *application* (§5.1) is invoked from `20` VUs with each VU executing `750` invocations for the duration of the test. This results in a total of about `15000` invocations for the entire application.

With test T1, we evaluate the overhead of the **Courier** in delivering functions as this function has the lowest processing overhead. With test T2, we test a more complex scenario, simulating the workload a cloud provider like AWS could potentially be faced with.

5.3 Environment

Our evaluation infrastructure consists of three different FaaS platforms, two from the public cloud providers: 1) AWS Lambda, 2) GCF, and one privately-hosted OpenWhisk platform on a single-node dual-socket system, with each socket containing an Intel Cascade Lake processor with 22 cores. We further used three different regions for AWS lambda and three memory configurations for GCF. Overall there are seven different FaaSDs, and these seven deployments are divided into three HeteroFaaSDs as explained below.

H1 **Platform-based HeteroFaaSD**: We use three FaaSDs consisting of three different platforms: 1) Private OpenWhisk, 2) AWS Lambda, and 3) GCF. All the three FaaSDs are created in the `Europe` region and all the application functions are configured with a memory of `512MB`.

H2 **Region-based HeteroFaaSD**: Three FaaSDs are created using the `GCF` platform in three different regions: 1) Europe-West3, 2) US-East1, and 3) Asia-South. All the application functions are configured with a memory of `512MB`.

H3 **Memory-based HeteroFaaSD**: Here, we created 3 FaaSDs with the fixed FaaS platform as `AWS Lambda` and the deployed region as `Europe-Central-1`. But in these FaaSDs, application functions are deployed with three different memory configurations: 1) `256MB`, 2) `512MB`, and 3) `1024MB`.

We use one VM to host a **Courier** instance and one VM for the K6 load testing engine. Both machines have Ubuntu 18.04, 2.4 GHz Xeon Skylake Processor, 2vCPU cores, and 4 GB Memory.

6 RESULTS

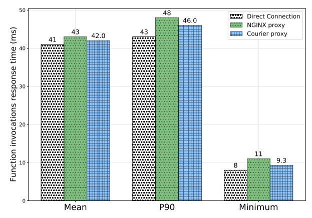
In this section, we first present the overhead induced by the **Courier**. After this, we present the performance of the individual functions (§6.2) within the serverless application (§5.3). Following this, we present the performance of the functions delivering algorithms (§6.3). Finally, we present the results of the algorithms under a high workload (§6.4). We repeat the tests five times for all our experiments and report metrics averaged over those five runs.

6.1 Courier System Overhead

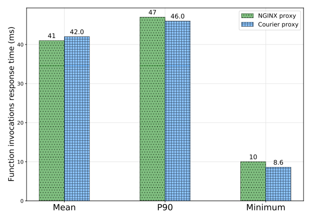
As a comparison counterpart for the system's overhead performance, we choose NGINX. As of November 2020, NGINX is one of the most widely used web servers. It serves about 24% of websites [28]. The developers advertise their fast performance and scalability as one of its crucial benefits. NGINX performs load balancing via a reverse proxy. This is the same way as **Courier** handles the redirect. We evaluate the performance in two aspects:

Table 1: Description of microbenchmarks used as part of the application for evaluation.

Microbenchmark	Description	Runtime
`nodeinfo`	Gives basic characteristics of node like CPU count, architecture, uptime	`Node.js 14`
primes-python	Calculates prime numbers till 100000.	`Python 3.7`
`linpack`	It solves a dense linear system of equations in double precision and returns the results in GFlops. Problem size (number of equations) is fixed to 100.	`Python 3.7`
`dd`	It is based on Unix dd command-line utility for converting and copying files. 128bytes as a block size and 5 times conversion is used as parameters.	`Python 3.7`
`gzip-compression`	Creates a file with random numbers of size 1MB and compresses it using gzip compression.	`Python 3.7`
`lr-prediction`	It first downloads a linear regression model trained on user reviews data from the storage bucket along with the test data and performs prediction on it.	`Python 3.7`
`nodejs-endpoint`	It is a REST endpoint which sends the current time along with the time zone.	`Node.js 14`
`sentiment-analysis`	Analyzes the sentiment of a provided string using the Python TextBlob library	`Python 3.7`



(a) System overhead



(b) Scheduling overhead

Figure 4: Evaluation of the Courier system overhead.

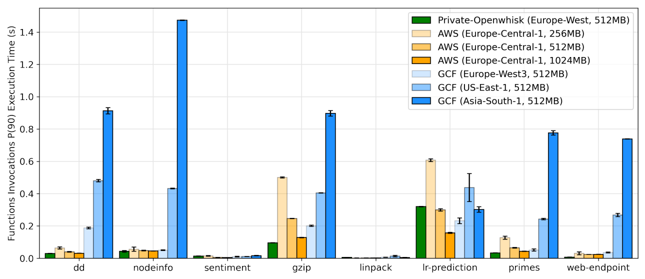


Figure 5: P90 response times of the sandboxed functions.

6.1.1 System Overhead. For this evaluation, we use the test T1 (§5.2). The function deployed using `AWS Lambda` in `Europe-Central-1` region with `512MB` memory is used as FaaSD for evaluation. We compare the benchmark on three different scenarios:

- Direct connection (we perform the requests directly to the FaaSD). This scenario corresponds to the baseline.
- NGINX reverse proxy.
- **Courier** reverse proxy with *Direct FaaSD strategy* (§4.2).

Figure 4a shows, the mean, 90th percentile, and minimum function invocations response time for the three scenarios. Across all response-time-based metrics, the **Courier** proxy performs better than the NGINX proxy. Although the 90th percentile response time of the **Courier** proxy is higher as compared to the direct connection's, for other metrics, its almost equivalent. Therefore we conclude that the system overhead is negligible for the **Courier**.

6.1.2 Scheduling Overhead. We execute the test T1 (§5.2) against the round-robin strategy. Here, we only evaluate against NGINX. Since NGINX does not support load distribution between multiple FaaS platforms, we could only make it work with H2 HeteroFaaSD (§5.3) where only region is changed between the FaaSD endpoints.

From Figure 4b we observe that, across most response-time-based metrics, the **Courier** performs slightly better than the NGINX proxy. Only the mean response time of NGINX is slightly better. The difference in the total amount of requests processed by each proxy is insignificant (below 0.1%), and no errors occurred. Hence, the scheduling overhead for the **Courier** is also negligible.

6.2 Individual Function's Performance

Usually, a serverless application consists of multiple functions, and the performance of one function could affect the others depending on it. Therefore, to determine the individual function's performance from the benchmark application (§5.1) on each FaaSD (§5.3), we need to sandbox them [19]. We use the **Courier** proxy to isolate each function and substitute its direct neighbors with dummy functions accepting the requests and sending the responses in the same format, but without any additional processing, allowing us to measure the performance of that function purely. We replace the neighboring function calls with a proxy function whose inputs are the originally called function names and the input payload to them. Each function invocations and response goes through the proxy function. This dummy proxy function will invoke the next function based on the input received, and at the same time, copies of these requests and responses are stored in the MongoDB database deployed together with **Courier**. Following this, each function receives its own sandboxed deployment where dummy functions replace the direct neighbors. These dummy functions will respond with the response stored in MongoDB. As a result, the time taken by the dependent functions to respond becomes negligible. This allows us to measure the relatively pure performance of the functions. Figure 5 shows the 90th percentile response time of the sandboxed functions when executed with test T2 (§5.2) on 7 FaaSDs (§5.3). We make the following observations:

6.2.1 Decrease in response time with the increase in memory: In Figure 5, we observe that for most of the functions in `AWS Lambda`

FaaSDs, the 90^{th} percentile response time decreases with an increase in the memory. This can be attributed to an increase in the number of allocated CPU clock cycles with increasing memory.

6.2.2 Function execution in a region closer to the user has a lower response time than the far ones: For the three GCF FaaSDs in Figure 5, we observe that the function in region Europe-West3 performed far better than the other two regions due to lower communication latency as the client was located in that region. Also, functions like *nodeinfo*, *dd*, and *gzip* etc. which are calling other functions (now dummy function deployed in Europe-West3 region) incur longer response times compared to the ones executing in the same region where the dummy function is deployed due to higher communication latency. Therefore, **Courier** proxy must handle all these scenarios when delivering functions.

6.2.3 Functions behave differently for different FaaSDs: All the functions when executed in the Private-OpenWhisk FaaSD performed better than the other FaaSDs due to the availability of higher resources. Additionally for functions like **dd** we can see a significant performance increment when using AWS Lambda based FaaSDs (maximum 0.08s response time) as compared to GCF based FaaSDs (minimum 0.19s response time). On the other hand, **lr-prediction** performed better on GCF based FaaSD with 512MB memory in Europe region (0.22s response time) as compared to AWS based FaaSD with the same memory and in the same region (0.3s response time). The heterogeneity in the performance of functions for the different FaaSDs is used by PFAWRR algorithm when delivering functions.

6.3 Algorithms performance for HeteroFaaSDs

Now we present the performance results of the function delivering strategies (§4.2) on three different HeteroFaaSDs:

6.3.1 Platform-based HeteroFaaSD (H1). Figure 6a shows the aggregated mean, 90^{th} percentile, and 95^{th} percentile response times of the individual FaaSDs as compared to the collaborative execution when using different functions delivering algorithms. For the individual FaaSDs, the Private-OpenWhisk FaaSD performs the best (with P(90) response time as 0.61s), then comes the AWS Lambda FaaSD (with P(90) response time as 0.90s), followed by the GCF FaaSD (with P(90) response time as 1.20s) across all the 3 metrics.

The algorithms here have a challenge of distributing the invocations to the three different FaaS platforms to provide high performance. RR and WRR with manually assigned weights of: 3 for Private-OpenWhisk FaaSD, 3 for AWS Lambda FaaSD and 2 for GCF FaaSD, perform similar to AWS Lambda FaaSD on mean response time metric but the P(90) (1.13s for RR and 1.09s for WRR) and P(95) (1.16s for RR and 1.13s for WRR) response times are more closer to the performance of GCF FaaSD. Clearly, our manually assigned weights are not good enough to get the overall performance closer to the AWS Lambda FaaSD or even less than it. However, AWRR and PFAWRR have the advantage of automatically updating the weights every δ time (in our case one minute), hence perform better than AWS Lambda FaaSD on mean (0.73s for AWRR and 0.71s for PFAWRR) and P(90) (0.85s for AWRR and 0.97s for PFAWRR) response time metric. In general for all the metrics, PFAWRR is able to perform better than the other algorithms since its adaptability at the function level.

Figure 7a shows the weights assignment for the three FaaSDs within the Platform-based HeteroFaaSD during the course of the load test for AWRR. We can observe that for a majority of the time, Private-OpenWhisk FaaSD had the highest weight since it can process the function invocations faster than the others. This provides a clear advantage when new FaaSDs join the HeteroFaaSD, i.e., the admin operators do not need to provide weights for them manually. The AWRR algorithm can automatically determine the weights for them based on the execution time of the functions.

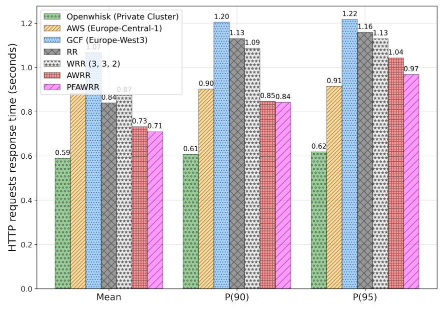
6.3.2 Region-based HeteroFaaSD (H2). The aggregated mean, 90^{th} percentile, and 95^{th} percentile response times of the individual FaaSDs as compared to the collaborative execution when using different algorithms is shown in Figure 7b. For the individual FaaSDs, the GCF FaaSD closer to the client (Europe-West3 region) performs the best (with 1.29s P(90) response time), then comes the GCF FaaSD in the US-East1 region (with 3.20s P(90) response time), followed by the GCF FaaSD in the Asia-South1 region (with 8.08s P(90) response time) for all the three metrics.

The algorithms here distribute the invocations across the three regions FaaSDs within a HeteroFaaSD and we can see a significant improvement in the performance alone with the RR algorithm (5.67s P(90) response time) as compared to the GCF FaaSDs in the Asia-South1 region. Manually introducing weights of <3, 3, 2> to the WRR algorithm for the FaaSDs did not have a significant impact on performance in terms of 90^{th} percentile and 95^{th} percentile response times but the mean response time (3.39s P(90) response time) got closer to the GCF FaaSD in US-East1 region. AWRR and PFAWRR performs better than the two GCF FaaSDs in US-East1 and Asia-South1 regions on mean (2.39s for AWRR and 2.23s for PFAWRR) and P(95) (4.05s for AWRR and 3.78s for PFAWRR) response time metric. This shows that the algorithms can adapt the distribution of invocations across FaaSDs within Region-based HeteroFaaSD. PFAWRR in general performs better than AWRR on mean P(95) metric but falls behind slightly on P(90) metric.

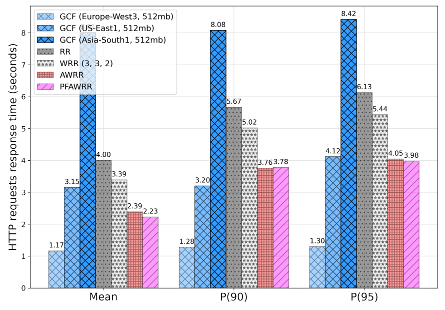
When looking at the the weights assignment for the three FaaSDs during the course of the test for AWRR in Figure 7b, we see for majority of the time, GCF FaaSD in Europe-West3 region had the highest weight since it is able to process the function invocations faster than the others, while if it gets overloaded the load automatically shifts to other FaaSDs.

6.3.3 Memory-based HeteroFaaSD (H3). For the FaaSDs in Memory-based HeteroFaaSD in Figure 6c, we observe that the AWS FaaSD with 1024MB memory configuration performs the best (with P(90) response time as 0.63s), followed by the FaaSD with 512MB memory configuration (having P(90) response time as 0.89s), and finally the FaaSD with 256MB memory configuration (with P(90) response time as 1.61s) in all the 3 metrics. This is due to AWS Lambda allocating CPU compute proportional to the memory provisioned.

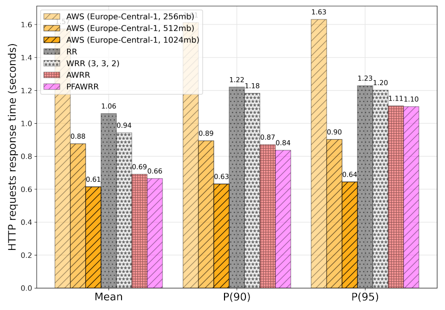
Similar to the last two HeteroFaaSDs, RR and WRR algorithm's 90^{th} percentile, and 95^{th} percentile response time metrics are higher than the second most performant FaaSD, i.e. AWS FaaSD with 512MB memory configuration, but the mean response time (1.06s for RR and 0.94s for WRR) got closer to it (0.88s). The AWRR and PFAWRR algorithms due to their dynamic adaptability perform better than the other algorithms and the two AWS FaaSDs with 256MB



(a) Platform-based HeteroFaaSD (H1)

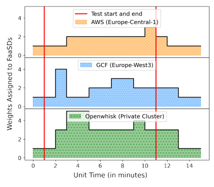


(b) Region-based HeteroFaaSD (H2)

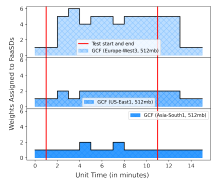


(c) Memory-based HeteroFaaSD (H3)

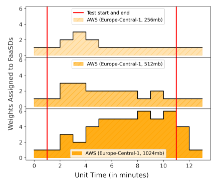
Figure 6: Individual FaaSDs performance comparison with the developed algorithms performance in three HeteroFaaSDs.



(a) H1

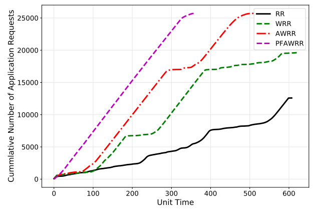


(b) H2

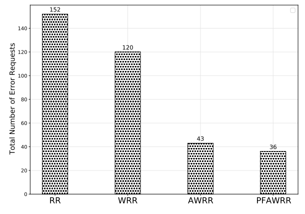


(c) H3

Figure 7: Weights distribution among FaaSDs during evaluation test for AWRR algorithm in three HeteroFaaSDs.



(a) Algorithms Performance



(b) Failed number of requests

Figure 8: Performance results under high user workload.

and 512MB memory configurations on mean (0.69s for AWRR and 0.66s for PFAWRR) and P(90) (0.87s for AWRR and 0.84s for PFAWRR) response time metric. PFAWRR works better than AWRR on all metrics. This shows that the designed algorithms can adapt the distribution of functions invocations across FaaSDs distributed over heterogeneous memory configurations. Figure 7c shows the weights assignment for the three FaaSDs for AWRR. We observe that most of the load is directed to the FaaSD with the highest memory since it is processing the invocations faster than the others.

6.4 Performance under high workload

To evaluate the performance of the algorithms under high load, we performed the testing with 50 VUs for the test *T2* with H1 HeteroFaaSD (§5.3, §5.2) where the number of requests were dynamically adjusted based on the response time [22].

The performance in this test is shown in Figure 8a. We observe that not all the functions delivering algorithms could manage to serve a total of 25716 requests within the 10 minutes testing time which AWRR and PFAWRR algorithms manage to complete. However, the PFAWRR algorithm completes them faster. In this scenario, the AWRR algorithm outperforms both the RR and WRR algorithms by a significant number of requests and time. The PFAWRR strategy can leverage individually generated weights and thus performs the best. On the other hand, AWRR can adapt dynamically to the overloaded FaaSDs and send the invocations to other FaaSDs. During this test, we also started seeing failing requests which indicate that FaaSDs are overloaded. Figure 8b shows the number of failed requests. Most of the failures reported above were timeouts. Since only AWRR and PFAWRR were able to complete all requests, therefore based on the failed requests we calculated the percentage availability of the FaaSDs within a HeteroFaaSD when using the two algorithms as: AWRR - 99.84%, and PFAWRR - 99.86%. Both the algorithms provide high availability.

7 DISCUSSION

In this section, we discuss our results from two aspects:

7.1 System Performance

As shown in §6.1, the **Courier** system can efficiently execute scheduling decisions and delivery of functions. Compared to NGINX, **Courier** performs better in delivering functions within the H2 HeteroFaaSD (§5.3). In our testing scenario, we tested the current implementation of **Courier** (§4) by deploying it to one single virtual machine. This implementation is somewhat limited in the number of requests that it can handle, as after a certain number of requests, the **Courier** system would be overloaded. However, since **Courier** is designed using a microservices architecture and made of tools (for example, MinIO, InfluxDB, etc.) that support deployment across multiple nodes, it can be easily extended to be deployed on a multiple node cluster as well. For example, the reverse proxy within **Courier** does not carry any stateful information. Therefore, reverse proxy instances are easily scalable on-demand without additional development effort.

7.2 Algorithms Performance

Courier already provides benefits for the cluster administrators. Generally, our algorithms made better functions delivering decisions as compared to traditional load balancing algorithms (§6.3). This means that administrators can easily add new FaaSDs within a HeteroFaaSD and the **Courier** functions delivering algorithm automatically profiles these FaaSDs while end-users are already using them. We analyze the two designed algorithms in more details :

7.2.1 Auto-Weighted Round Robin. The Auto Weighted Round-Robin (AWRR) algorithm itself does provide good results while delivering functions within the HeteroFaaSDs (§6.3). In all the testing scenarios, we observed that the weight generation based on average execution times of all the functions running in a FaaSD could provide good insight into the computational capabilities of the FaaSD. Furthermore, the automatic dynamic changes in weights for each FaaSD within a HeteroFaaSD (Figure 7) during the testing scenarios shows the algorithm's adaptability if a FaaSD is overloaded. However, the frequent collection of metrics from each FaaSD can induce an additional overhead on the FaaSD, which can be avoided by setting the optimal weight update δ time. Additionally, if the differences in runtime are not big, then all the FaaSDs within a HeteroFaaSD will be assigned with the same weights. This algorithm can be easily extended to work on other metrics as well.

7.2.2 Per-Function Auto Weighted Round Robin. The Per-Function Auto Weighted Round-Robin (PFAWRR) algorithm was able to perform better or equivalent to AWRR algorithm when delivering the functions within the HeteroFaaSDs (§6.3). However, it performed much better under heavy load and outperformed the other algorithms. This algorithm leverages the heterogeneity of the functions and the underneath capabilities of FaaSD by assigning higher weights to functions for FaaSDs where they can operate more efficiently or to FaaSDs closer to the location of the user so that the overall latency is reduced. A drawback of the PFAWRR algorithm is that it requires all the functions to be invoked once on all the FaaSDs within a HeteroFaaSD. Once that is done, it can provide good functions delivering decisions. If all the functions have not been invoked once, the PFAWRR strategy would have to "warm-up," i.e., learn from previous invocations by providing some historical data or manual feeding of information. During that time, the PFAWRR strategy will perform similarly to the RR strategy.

8 CONCLUSION

Due to the current limitations of serverless computing for highly dynamic applications in their structure and computational requirements, we introduced the **Courier** enabling the automatic delivering of serverless functions within HeteroFaaSDs based on the execution time of functions on FaaSDs within the HeteroFaaSDs. We evaluated **Courier** on three HeteroFaaSDs using three FaaS platforms (§6.3). Furthermore, we introduced two new algorithms: AWRR and PFAWRR, purposefully built for delivering functions and compared them against the traditional load-balancing algorithms. We found that collaborative execution of the functions between the FaaSDs within a HeteroFaaSD using AWRR and PFAWRR can lead to higher performance as compared to scenarios where functions are exclusively invoked on most of the individual FaaSDs.

The **Courier** can be extended to further increase the adoption of FaaS in HeteroFaaSDs. The function delivering decision making can be improved to include various aspects such as cold starts, scientific workflows [12], and parallel executions as well [31].

ACKNOWLEDGMENTS

This work was supported by the funding of the German Federal Ministry of Education and Research (BMBF) in the scope of the Software Campus program. Google Cloud credits in this work were provided by the *Google Cloud Research Credits* program with the award number NH93G06K20KDXH9U.

REFERENCES

[1] Austin Aske and Xinghui Zhao. 2018. Supporting Multi-Provider Serverless Computing on the Edge. In *Proceedings of the 47th International Conference on Parallel Processing Companion* (Eugene, OR, USA) *(ICPP '18)*. Association for Computing Machinery, New York, NY, USA, Article 20, 6 pages. https://doi.org/10.1145/3229710.3229742

[2] AWS. [n.d.]. AWS Elastic LoadBalancer Limits. $https://docs.aws.amazon.com/de_de/elasticloadbalancing/latest/userguide/how-elastic-load-balancing-works.html$. Accessed: 2021-01-10.

[3] AWS Elastic Load Balancing. [n.d.]. https://aws.amazon.com/elasticloadbalancing/. Accessed 09/24/2020.

[4] AWS Lambda Pricing. [n.d.]. https://aws.amazon.com/lambda/pricing/. Accessed 09/24/2020.

[5] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. 2017. Serverless Computing: Current Trends and Open Problems. In *Research Advances in Cloud Computing*. Springer Singapore, 1–20. https://doi.org/10.1007/978-981-10-5026-8_1

[6] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog Computing and Its Role in the Internet of Things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing* (Helsinki, Finland) *(MCC '12)*. Association for Computing Machinery, New York, NY, USA, 13–16. https://doi.org/10.1145/2342509.2342513

[7] Antonio Brogi, Ahmad Ibrahim, Jacopo Soldani, José Carrasco, Javier Cubo, Ernesto Pimentel, and Francesco D'Andria. 2014. SeaClouds. *ACM SIGSOFT Software Engineering Notes* 39, 1 (2 2014), 1–4. https://doi.org/10.1145/2557833.2557844

[8] Apache Brooklyn. [n.d.]. The Theory behind Brooklyn. https://brooklyn.apache.org/learnmore/theory.html. Accessed: 2020-01-10.

[9] Bernd Bruegge. 2010. *Object-oriented software engineering : using UML, patterns, and Java.* Prentice Hall, Boston.

[10] Mohak Chadha, Anshul Jindal, and Michael Gerndt. 2020. Towards Federated Learning Using FaaS Fabric. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing* (Delft, Netherlands) *(WoSC'20)*. Association for Computing Machinery, New York, NY, USA, 49–54. https://doi.org/10.1145/3429880.3430100

[11] Mohak Chadha, Anshul Jindal, and Michael Gerndt. 2021. Architecture-Specific Performance Optimization of Compute-Intensive FaaS Functions. *CoRR* abs/2107.10008 (2021). arXiv:2107.10008 https://arxiv.org/abs/2107.10008

[12] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. [n.d.]. A Review of Serverless Use Cases and their Characteristics. ([n. d.]). arXiv:2008.11110 [cs.SE]

[13] Youssef Fahim, Hamza Rahhali, Mohamed Hanine, El-Habib Benlahmar, El-Houssine Labriji, Mostafa Hanoune, and Ahmed Eddaoui. 2018. Load Balancing in Cloud Computing Using Meta-Heuristic Algorithm. *Journal of Information Processing Systems* 14, 3 (2018).

[14] Chen-Fu Fan., Anshul Jindal., and Michael Gerndt. 2020. Microservices vs Serverless: A Performance Comparison on a Cloud-native Web Application. In *Proceedings of the 10th International Conference on Cloud Computing and Services Science - CLOSER,*. INSTICC, SciTePress, 204–215. https://doi.org/10.5220/0009792702040215

[15] Martin Fowler. 2014. Circuit Breaker. https://martinfowler.com/bliki/CircuitBreaker.html. Accessed: 2021-01-10.

[16] Geoffrey C. Fox, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. [n.d.]. Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research. ([n. d.]). https://doi.org/10.13140/RG.2.2.15007.87206 arXiv:1708.08028 [cs.DC]

[17] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless Computing: One Step Forward, Two Steps Back. *ArXiv* abs/1812.03651 (2018).

[18] Influxdata. [n.d.]. Influxdb. https://www.influxdata.com/products/influxdb/. Accessed: 2021-05-27.

[19] Anshul Jindal, Mohak Chadha, Shajulin Benedict, and Michael Gerndt. 2021. Estimating the Capacities of Function-as-a-Service Functions. In *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion* (Leicester, United Kingdom) *(UCC '21 Companion)*. Association for Computing Machinery, New York, NY, USA, 8 pages. https://doi.org/10.1145/3492323.3495628

[20] Anshul Jindal, Mohak Chadha, Michael Gerndt, Julian Frielinghaus, Vladimir Podolskiy, and Pengfei Chen. 2021. Poster: Function Delivery Network: Extending Serverless to Heterogeneous Computing. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. 1128–1129. https://doi.org/10.1109/ICDCS51616.2021.00120

[21] Anshul Jindal, Michael Gerndt, Mohak Chadha, Vladimir Podolskiy, and Pengfei Chen. 2021. Function delivery network: Extending serverless computing for heterogeneous platforms. *Software: Practice and Experience* 51, 9 (2021), 1936–1963. https://doi.org/10.1002/spe.2966 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2966

[22] k6. [n.d.]. API load testing. https://k6.io/docs/testing-guides/api-load-testing/. Accessed: 2021-05-25.

[23] K6. [n.d.]. K6 Results output. https://k6.io/docs/getting-started/results-output/. Accessed: 2021-05-27.

[24] D. Kelly, F. Glavin, and E. Barrett. 2020. Serverless Computing: Behind the Scenes of Major Platforms. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. 304–312. https://doi.org/10.1109/CLOUD49709.2020.00050

[25] M Lavanya and V Vaithiyanathan. 2015. Load prediction algorithm for dynamic resource allocation. *Indian J Sci Technol* 8 (2015), 35.

[26] Sulav Malla and Ken Christensen. 2019. HPC in the cloud: Performance comparison of function as a service (FaaS) vs infrastructure as a service (IaaS). *Internet Technology Letters* 3, 1 (12 2019), e137. https://doi.org/10.1002/itl2.137

[27] MinIO. [n.d.]. MinIO Object Storage. https://min.io/product/overview. Accessed: 2021-05-27.

[28] Netcraft. [n.d.]. November 2020 Web Server Survey. https://news.netcraft.com/archives/2020/11/30/november-2020-web-server-survey.html. Accessed: 2021-05-25.

[29] NGINX. [n.d.]. NGINX Load Balancing Algorithms. https://www.nginx.com/blog/choosing-nginx-plus-load-balancing-techniques/. Accessed: 2020-01-10.

[30] E. D. Nitto, M. A. A. d. Silva, D. Ardagna, G. Casale, C. D. Craciun, N. Ferry, V. Muntes, and A. Solberg. 2013. Supporting the Development and Operation of Multi-cloud Applications: The MODAClouds Approach. In *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 417–423.

[31] Klaithem Al Nuaimi, Nader Mohamed, Mariam Al Nuaimi, and Jameela Al-Jaroodi. 2012. A Survey of Load Balancing in Cloud Computing: Challenges and Algorithms. In *2012 Second Symposium on Network Cloud Computing and Applications*. IEEE. https://doi.org/10.1109/ncca.2012.29

[32] Google Cloud Platform. [n.d.]. Serverless network endpoint groups overview. https://cloud.google.com/load-balancing/docs/negs/serverless-neg-concepts. Accessed: 2021-05-27.

[33] Mike Roberts. 2018. Serverless Architectures. https://martinfowler.com/articles/serverless.html. Accessed: 2020-04-17.

[34] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 1063–1075.

[35] G.Siva Shanmugam and N.Ch.S. N. Iyengar. 2016. Effort of Load Balancer to Achieve Green Cloud Computing: A Review. *International Journal of Multimedia and Ubiquitous Engineering* 11, 3 (3 2016), 317–332. https://doi.org/10.14257/ijmue.2016.11.3.30

[36] The Eclipse Jetty Project. [n.d.]. https://www.eclipse.org/jetty/. Accessed 09/24/2020.

[37] The Serverless Framework. [n.d.]. https://www.serverless.com/. Accessed 09/24/2020.

[38] Ruixia Tong and Xiongfeng Zhu. 2010. A Load Balancing Strategy Based on the Combination of Static and Dynamic. In *2010 2nd International Workshop on Database Technology and Applications*. IEEE. https://doi.org/10.1109/dbta.2010.5658951