From DevOps to NoOps : Is it Worth it?

Anshul Jindal[0000−0002−7773−5342] and Michael Gerndt[0000−0002−3210−5048]

Chair of Computer Architecture and Parallel Systems,
Technical University of Munich, Garching, Germany
anshul.jindal@tum.de, gerndt@in.tum.de

Abstract. With the rise of the adoption of microservices architecture due to its agility, scalability, and resiliency for building the cloud-based applications and their deployment using containerization, DevOps were in demand for handling the development and operations together. However, nowadays serverless computing offers a new way of developing and deploying cloud-native applications. Serverless computing also called NoOps, offloads management and server configuration (operations work) from the user to the cloud provider and lets the user focus only on the product developments. Hence, there are debates regarding which deployment strategy to use.
This research provides a performance comparison of a cloud-native web application along with three different function benchmarks in terms of scalability, reliability, and latency when deployed using DevOps and NoOps deployment strategy. NoOps deployment in this work is achieved using Google Cloud Function and OpenWhisk, while DevOps is achieved using the Kubernetes engine. This research shows that neither of the deployment strategies fits all the scenarios. The experimental results demonstrate that each type of deployment strategy has its advantages under different scenarios. The DevOps deployment strategy has a huge performance advantage (almost 72% lesser 90 percentile response time) for simple web-based requests and requests accessing databases while compute-intensive applications perform better with NoOps deployment. Additionally, NoOps deployment provides better scaling-agility as compared to DevOps.

Keywords: Microservices, Serverless, DevOps, NoOps, Cloud-native Applications, Cloud Computing.

1 Introduction

Cloud computing providing a "pay-as-you-go" model, enables cheap and easy access to the data processing and storage resources. Nowadays most enterprises have migrated or refactored their existing monolithic-based applications into the microservices architecture and deployed it on the cloud [13]. Microservices architecture offers higher agility since it decouples a big application service into smaller microservices and each microservice is then deployed separately either on a virtual machine or in a container where the resources can be scaled on-demand. Developers are now not only assigned the task for the development of

the microservices but also include the operations task like deployments, therefore are called *DevOps*. DevOps is the fusion of development and operations. It drives the services lifecycle, from the design to the delivery. Besides many advantages, microservices architecture also has some disadvantages in software development. For instance, each service communicates through the network via REST API endpoints, which can pose data security concerns during the communication. Also, network latency and load balancing do arise. Furthermore, research shows that the development team with a strong DevOps culture may get benefit from the microservices architecture, therefore the effort to establish DevOps culture is another consideration for adopting a microservices architecture [38].

On the other hand, serverless computing has gained higher popularity and more adoption in different fields since the launch of AWS Lambda in 2014 [18]. Serverless computing is a cloud computing model that abstracts server management and infrastructure decisions away from the users [43]. In this model, the allocation of resources is managed by the cloud service provider rather than by the team of application developers. In other words, DevOps are free from operations work and can purely focus on development and is therefore called *NoOps*. Also, in serverless computing cost is charged on the number of requests received to the functions and the time it takes for the code to execute [16]. This pricing model is much simpler as compared to the traditional instance pricing model which is based on the number of instances and their diverse types. Therefore, application owners in this model are also free from the decisions of choosing instance types and several instances. Function-as-a-Service (FaaS) is a key enabler of serverless computing [43]. In FaaS, an application is decomposed into simple, standalone functions that are uploaded to a FaaS platform for execution. These functions are stateless, i.e., the state is not kept across function invocations. Functions can be invoked by a user's HTTP request or by another type of event created within the FaaS platform. The FaaS platform is responsible for deploying and facilitating resources to the application functions. Currently, there exist many open source and commercial FaaS platforms [30]. All of the large cloud providers have FaaS platforms available based on a *container orchestration platform* such as Kubernetes. In this work, we have used OpenWhisk and Google Cloud Function as the FaaS platforms for deploying application functions.

Both *DevOps* and *NoOps* methodologies have their advantages and disadvantages and the decision to adopt a design pattern depends on the team capability and project requirements. In this research, we have analyzed a cloud-native web application along with 3 function benchmarks refactored into both microservices and FaaS deployment models from the aspect of scalability, reliability, and latency. The experimental results demonstrate that the *DevOps* deployment strategy has a huge performance advantage (almost 72% lesser percentile-90 response time) for simple web-based requests and requests accessing databases while compute-intensive applications perform better with *NoOps* deployment. Additionally, *NoOps* deployment provides better scaling-agility as compared to *DevOps*.

The main contribution of this paper are as follows:

- Performance comparison between *DevOps* and *NoOps* deployment methodologies using two different methods in terms of scalability, reliability, cost, and latency. In our previous work [15], we compared microservices to serverless using AWS Lambda as the FaaS platform and a native cloud web application but in this work, we have extended it by using OpenWhisk and Google Cloud Functions as FaaS platforms. Furthermore, we have added additional function benchmarks and an application for the evaluations.
- Performance comparison between OpenWhisk and Google Cloud Functions is presented as part of this work and we are the first one to do so.
- We highlight different use cases recommendations where *DevOps* and *NoOps* deployment methodologies can be used based on the type of load, scenario, and the amount of the requests.

The rest of this article is composed as follows. Section 2 discusses the background knowledge required for this paper in brief. Section 3 provides the overall methodology used for the evaluation including the different methods and load test settings. Section 4 showcase the results of the conducted analysis. Section 5 summarizes the discussion of the results and in section 6 we describe some of the previous works in this domain. Lastly, section 7 concludes the paper.

2 Background

In this section, we first present an overview of the *DevOps* and *NoOps* deployments cloud model. Following this, we describe the architecture and high-level workflow of the two FaaS platforms used in this work.

2.1 DevOps-based Cloud Model

DevOps cloud model is based on the microservices architecture where the developers are responsible both for the development and operations task. Microservices consists of a suite of modules, and each module is dedicated to a specific business goal and communicates via a well-defined interface. The principle of *DevOps* cloud model architecture is loose-coupling, which requires multiple service instances for an application [36]. The deployment of *DevOps* cloud model can be achieved either by deploying each microservice on a separate virtual machine instance or deploying microservices per container or one can even combine multiple microservices per virtual machine and container. The containerization deployment benefits from the higher deployment speed, agility, and lower resources consumption [37]. This strategy also allows each microservice instance to run in isolation on a host. This enables the guaranteed quality of service for each microservice at the cost of idle resources. Container orchestration tools like Kubernetes[1] and Google Kubernetes Engine (GKE)[2] can be used for managing the microservices containers.

[1] https://kubernetes.io/docs/
[2] https://cloud.google.com/kubernetes-engine

The benefits of this model are improved fault tolerance, flexibility in using technologies and scalability, and speed up of the application [32]. However, there are also some disadvantages such as the increase of development and deployment complexity, implementing an inter-service communication mechanism, and challenging to conduct end-to-end testing [11].

2.2 NoOps-based Cloud Model

NoOps cloud model is based on the serverless computing where Function-as-a-Service (FaaS) platform facilitates application development and the user does not have to worry about the infrastructure management, but only about the code being deployed. The pricing is charged based on the number of requests to the functions and the duration, the time it takes for the function code to execute [1]. The latter varies according to the number of resources such as memory and CPU cores allocated to the function, and are automatically adapted to deliver the best performance. Instead of developing application logic in the form of services and managing the required resources, the application developer implements fine-grained functions connected in an event-driven application and deploys them into the FaaS platform [43]. The platform is responsible for providing resources for function invocations and performs automatic scaling depending on the workload. The functions can be closely integrated with other services, e.g., cloud databases, authentication and authorization services, and messaging services. These services are called Backend-as-a-Service (BaaS). BaaS are the third-party services that replace a subset of functionality in a function and allow the users to only focus on the application logic [25]. In FaaS, function invocations are handled by using containers. Since functions are stateless, the state of the application is stored in databases. In comparison to *DevOps* model, *NoOps* has three advantages (1) no continuously running services are required, (2) functions are only charged when they are executed, and (3) the function abstraction increases the developer's productivity.

One of the biggest differences between other forms of cloud models and the *NoOps* model is scalability [21]. The application automatically scales up or down based on the resource usage (with scaling down to zero number of instances as well) and developers do not have to specify any scaling parameters. The infrastructure of the cloud service provider starts up ephemeral instances of each function on-demand. In general, the total cost of ownership decreases.

NoOps based functions can be invoked by a user's HTTP request or by another type of event created within a FaaS platform. The FaaS platform is responsible for providing resources for function invocations and performs automatic scaling. Currently, a significant number of open source and commercial FaaS platforms are available [30]. FaaS platforms implementations are based on starting containers for function invocations on top of a *container orchestration platform* such as Kubernetes. Applications are defined via a *deployment specification* that describes the functions, APIs, permissions, configurations, and events that make up a serverless application. The specification can be given via a command-line or web interface, or by using some frameworks like Serverless [39] and Architect [7].

We introduce in the following subsections two FaaS platforms which are used as part of this work.

2.2.1 OpenWhisk (OW)

Apache OpenWhisk is a serverless open source cloud platform that was originally developed by a research group at IBM in 2015 and was released in December 2016. It was later donated to the Apache Software Foundation [33]. It powers IBM's serverless offering, IBM Cloud Functions, and implements FaaS on top of Kubernetes as the container orchestration platform. Functions in OpenWhisk are called actions and the execution of an action is called an invocation. Actions and rules can be created through the command-line interface (CLI) (`wsk` [5]), user interface (UI), or SDK. Created actions can then be invoked either manually through the same methods or by event triggers. Events can originate from multiple sources including timers, databases, message queues, or websites like Slack.

OpenWhisk consists of multiple components under the hood and all the components are packaged inside their individual docker containers when OpenWhisk is deployed [6]. Each function invocation is translated into an HTTP request to the Nginx server [35]. The Nginx server is a single point of entry and its main purpose is to implement the support for the HTTPS secure web protocol. On receiving a request, the Nginx server forwards it to the controller where the controller is responsible for authenticating and authorizing the requests. The controller keeps track of the availability of the invokers, i.e., the workers that run the code and chooses one of them for the invocation. The controller publishes the messages to Kafka addressed at a chosen invoker and once the message delivery is confirmed by the invoker, an HTTP request is sent back to the user with an *ActivationId*, which can be used for retrieving the results of this function call. Invokers set up a new docker container for each action, inject the code into them, execute the code, obtain the results, and then destroy it. These containers are run inside Kubernetes pods. There can be an invoker per Kubernetes worker node or an invoker can be responsible for managing multiple Kubernetes worker nodes. Functions can also be chained together into sequences where chained functions use the output of the preceding function as input.

2.2.2 Google Cloud Functions (GCF)

Google Cloud Functions is a serverless execution environment for building and connecting services in a cloud-based application [2]. With Google Cloud Functions, developers do not need to provision any infrastructure or worry about managing any servers, the whole environment including infrastructure, operating systems, and runtime environments are managed by Google. Currently, Cloud Functions supports JavaScript, Python 3, Go, and Java runtimes. Cloud Functions are simple, single-purpose functions that are attached to events emitted from the cloud infrastructure and services. The function is triggered when an event being watched is fired. These events can be things like changes in a database, files added to a storage system, or a new virtual machine instance is created. A response to

an event is created using a trigger which can then be attached to a function to capture and act on events.

Each Cloud Function runs in its own isolated secure execution context, scales automatically, and has a lifecycle independent from other functions [17]. Cloud Functions handles incoming requests by assigning them to instances of function. Depending on the volume of requests, as well as the number of existing function instances, Cloud Functions may assign a request to an existing instance or create a new one. Each instance of a function handles only one concurrent request at a time. Thus the original request can use the full amount of resources (CPU and memory) that you requested. In cases where inbound request volume exceeds the number of existing instances, Cloud Functions start multiple new instances to handle requests. This automatic scaling behavior allows Cloud Functions to handle many requests in parallel, each using a different instance of the function.

3 Methodology

For understanding the performance differences between the *DevOps* and *NoOps* deployment strategies, we consider a range of benchmarks. These benchmarks are evaluated for both the deployment strategies using two different methods in each case. In this section, we first present the details about the benchmarks used for evaluating the deployment strategies and then describe the four different deployments (Kubernetes hosted by Google Kubernetes Engine and self-hosted Kubernetes cluster for DevOps, and OpenWhisk based functions deployment and Google Cloud functions for NoOps) methods used in this work. We also present the load testing infrastructure and details used for the evaluation in this section.

3.1 Benchmarks

We have considered a microservices application matching with the real world applications along with 3 function benchmarks which include compute-intensive, simple API endpoint, and image processing functions for evaluating the deployment strategies. Below subsections present the details about the microservices application and the benchmarks.

3.1.1 Cinema microservices application

For demonstrating the performance differences between deployment of an application using the DevOps methodology (microservices-based deployment) and NoOps methodology (function-based deployment), we use an opensource microservices application: *cinema*[3], which contains movies information, show timings of the movies, users information, and movie bookings made by the user. The overall architecture, its services, interaction between them, and the API endpoints in each of the services are shown in Fig. 1. All the data required by each service is stored inside the Mongo database. The application is developed in Python and consists of the following four services:

[3] https://github.com/umermansoor/microservices

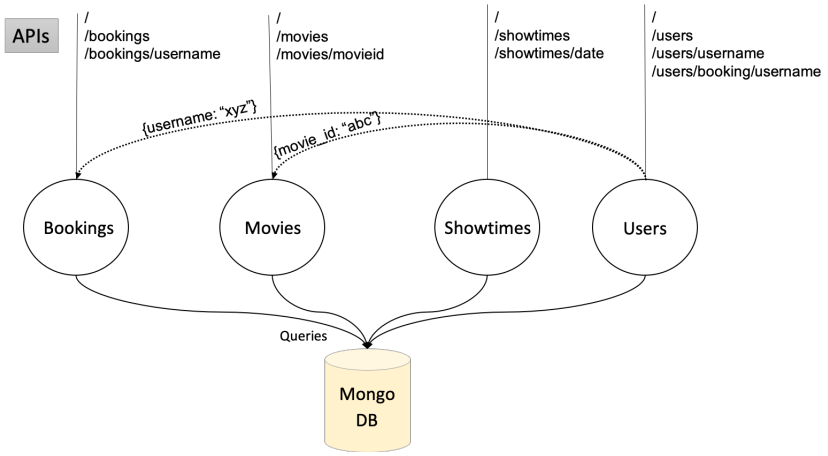

Fig. 1: Overall architecture of the microservices *cinema* application along with the interaction between its services and the API endpoints supported by each of its services.

- **Movies**: This service is responsible for accessing the movie's information (movie title, director, and rating) stored in the "movies" collection of MongoDB. It has two main API endpoints, 1) `/movies` to return all the movies information as JSON, and 2) `/movies/movieid` to return a movie information based on the specified `movieid` parameter.
- **Showtimes**: Movies show timings are managed by this service and are stored in the "showtimes" collection of MongoDB. It also has two main API endpoints, 1) `/showtimes` to return all the shows information for all the dates as JSON, and 2) `/showtimes/date` to return movie shows information for the specified `date`.
- **Bookings**: It manages the movie shows booked by the users and stores them into the "bookings" collection of MongoDB. It also has two main API endpoints, 1) `/bookings` to return all the bookings made by all the users as JSON, and 2) `/bookings/username` to return the movie bookings made by a particular user.
- **Users**: It manages the user information (full name, and address) stored in the "users" collection of MongoDB. It has three main endpoints: 1)`/users` to return all the users information as JSON, 2) `/users/username` to return user information based on the specified `username` parameter and 3) `/users/booking/username` to return the movie bookings information made by the user for the specified `username`; here the service first queries the *Bookings* service to provide the movie reservations made by the user and then based on that information it queries the *Movies* service to get the movies information and return its JSON back to the user.

Additionally, for testing the *NoOps* deployment, the application services are decomposed into different functions. Each of the API endpoints is converted into

Table 1: Overview of the API endpoints, microservices containers, their ports and serverless functions in *cinema-application* and used in this work for evaluation.

Service	**API endpoint**	**Microservices (container:port)**	**Serverless (function)**
Movies	/ /movies /movies/movieid	movies:5001	movies-base movies-all movies-id
Bookings	/ /bookings /bookings/username	bookings:5003	bookings-base bookings-all bookings-username
Showtimes	/ /showtimes /showtimes/date	showtimes:5002	showtimes-base showtimes-all showtimes-date
Users	/ /users /users/username /users/booking/username	users:5000	users-base users-all users-id users-booking

a new function and therefore in total, there are 13 functions and 4 microservices. Summary of each of these functions and microservices is shown in the Table 1.

3.1.2 Functions

To investigate further the performance differences of each deployment strategy, we used a subset of the benchmarks provided with the FaaSProfiler [40] and modified them for our use case. Furthermore, we developed microservices implementations of the chosen functions to enable their execution on the Kubernetes engine. The OpenWhisk action container generally includes code for the function along with its language runtime. OpenWhisk processes the incoming HTTP requests for the function invocation with any number of arguments and sends the results back to the user or caller. For most of the functions, we have used the default runtime environment provided by OpenWhisk depending on the language that the function is written in. If a function uses some extra packages which are not part of their default language runtimes, we created a Docker runtime for it based on their default docker runtime. While on google cloud functions the package requirements along with the code are specified. They automatically create a runtime environment based on it.

The functions used as part of this work are summarized in the Table 2 along with their description and language runtimes. The `nodeinfo` function exposes an HTTP endpoint and provides the user with basic information about the system such as hostname, underlying architecture, number of CPUs, etc. We utilize this function to test the general performance of each strategy and get an overall idea of their performance on a basic web application. The compute-intensive `primes-python` function is used for comparing their performances on compute-intensive applications and study the scalability aspects. Finally, for demonstrating

Table 2: List of functions we developed or modified for demonstrating performance differences of each deployment strategy.

Function Name	**Description**	**Language runtime**
nodeinfo	Gives basic characteristics of node like CPU count, architecture, uptime.	Node.js
primes-python	Calculates prime numbers till 10000000.	Python3
image-processing	Reads an image from object storage (here google object storage) and performs basic operations	Python3

the access latency each strategy adds up when accessing an object (in our work an image) stored on google cloud storage `image-processing` function is used where after getting the image basic operations like flipping, rotating, conversion to grayscale, and resize are performed.

3.2 Deployment Strategies

The above-presented application and function benchmarks are deployed using the two strategies: *DevOps* and *NoOps*. *DevOps* deployment is achieved by deploying the microservices on a Kubernetes cluster and configuring the scaling and other parameters manually, whereas *NoOps* deployment is achieved by deploying the functions (of the applications and function benchmarks) on a FaaS platform. In this work, we have used our self hosted OpenWhisk platform and Google Cloud Functions as FaaS platforms. Additionally, we have hosted the mongo database externally therefore it remains common to all the methods along with the google storage bucket for the *image-processing* function.

3.2.1 DevOps

The *DevOps* deployment strategy is achieved by deploying the microservices on a Kubernetes cluster with a manual configuration of the scaling parameters. In this work, this is achieved using two methods:

1. **Google Cloud Platform hosted Kubernetes Engine (devops-gcp-hosted-kubernetes)**: In this deployment method, a Google Kubernetes Engine (GKE) cluster is created with three nodes each with a configuration of 4 vCPU and 16GB of memory. The Autoscaling of nodes is not enabled. Once the cluster is created then each of the microservices of the application and the converted microservices of the functions are deployed on it. Additionally, each of the microservices are exposed externally using a Load Balancer. Also, horizontal pod autoscaling is enabled for each of the microservices as well with the CPU utilization threshold set to 80%, and minimum replicas to 1 and maximum as 5. The overall deployment architecture is shown in Fig. 2. Clients through the command-line interface (CLI) or REST APIs

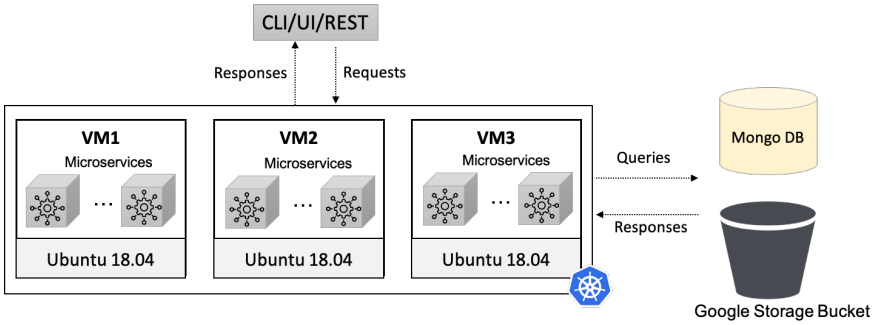

Fig. 2: Overall architecture of *DevOps-Deployment* model along with the interaction between its components using Kubernetes as the underneath container orchestration. The Kubernetes engine is created using the Google Kubernetes Engine (GKE) on Google Compute Platform (GCP).

send requests to the external addresses of the exposed microservices and get a response in return. Each of the microservices of the application can access the externally hosted mongo database and google storage bucket.

2. **Self-hosted Kubernetes Engine (devops-self-hosted-kubernetes)**: In this deployment method, instead of using the Kubernetes engine hosted by GKE, we created our own using the kubeadm[4] tool with three nodes each with the same configuration as the previous method (4 vCPU and 16GB memory). The cluster has three nodes with one master and two worker nodes but to have the same configuration as the one hosted by GKE we tainted the master node as well so as to allow the pods to be scheduled on it. Other configurations: deployment of the services, exposing them externally, and enabling autoscaling were done in a similar manner as the above method. Each of the service here also can access the externally hosted mongo database and google storage bucket. The overall deployment architecture is similar to the one shown in Fig. 2.

3.2.2 NoOps

The *NoOps* deployment strategy is achieved by deploying the functions on a FaaS platform. Functions can then be invoked by a user's HTTP request or by another type of event created within the FaaS platform. The FaaS platform is responsible for providing resources for function invocations and performs automatic scaling. In this work, we have used two FaaS platforms for the evaluation:

1. **OpenWhisk (noops-openwhisk)**: We started an OpenWhisk platform on top of a Kubernetes cluster hosted using Google Kubernetes Engine (GKE) with three nodes each with a configuration of 4 vCPU and 16GB

[4] https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/

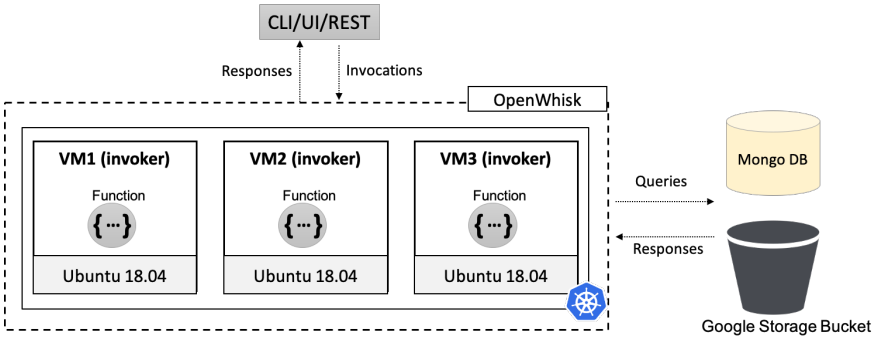

Fig. 3: Overall architecture of *NoOps-Deployment* model using OpenWhisk with Kubernetes as its container factory. The Kubernetes engine is created using the Google Kubernetes Engine (GKE) on Google Compute Platform (GCP).

of memory. Additionally, we increased the default limits on the number of concurrent invocations and invocations per minute which can be served in OpenWhisk to `99999`, and the memory allocated to the invoker is set to `2048` MiB. Once the OpenWhisk is deployed, then the *wsk* command-line interface is used for deploying the functions onto the OpenWhisk. Each function is allocated memory of 256MB. The overall deployment architecture is shown in Fig. 3. Clients through the command-line interface (CLI) or REST APIs send function invocation requests to the exposed OpenWhsik Nginx external address and get responses in return.

2. **Google Cloud Functions (nops-google-cloud-functions)**: In this deployment method, instead of using the self-hosted FaaS platform, we have used the Google Cloud Function for deploying the functions. As part of it, we only specified the function runtime and the python code of the functions. Google Cloud automatically handles the operational infrastructure. Each function is allocated memory of 256MB along with Python 3.7 as the function runtime and used HTTP as the trigger type. Each of the function here is exposed externally and also access the externally hosted mongo database and google storage bucket. The overall deployment architecture is shown in the Fig. 4.

3.3 Load Test Settings and Infrastructure

Our evaluation strategy is implemented via a load testing tool - *k6* [3]. *k6* is a developer-centric open-source load and performance regression testing tool for testing the performance of the cloud-native backend infrastructure, including APIs, microservices, functions, containers, and websites. *k6* generates different patterns of the user workload to the deployed system. *k6* uses a script for running the tests where the HTTP(s) endpoint along with the request parameters are specified. HTTP(s) endpoint represents the deployed microservice or function

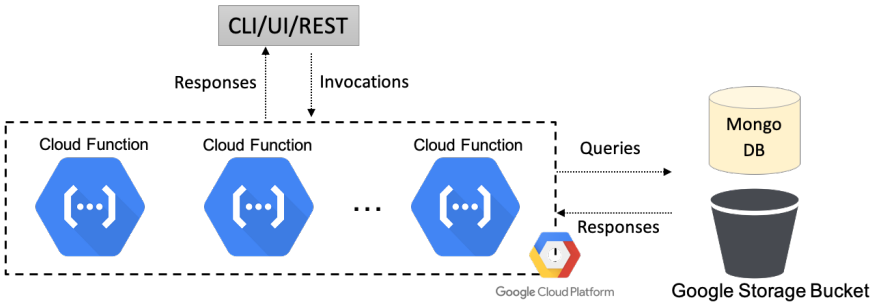

Fig. 4: Overall architecture of *NoOps-Deployment* model using Google Cloud Functions on Google Cloud Platform (GCP).

endpoint. Two of the other k6 parameters which are configured as part of each test are:

- **Virtual Users (VUs)**: Virtual Users (VUs) are the entities in k6 that execute the test and make HTTP(s) or websocket requests. VUs are concurrent and will continuously iterate through the request endpoint until the test ends.
- **Duration**: A string specifying the total duration a test will run. During this time each VU will execute the script in a loop.

The number of requests per second generated by k6 depends on the number of VUs and the time taken by each request to complete. For example, if there are 10 VUs with total test duration set to 10 minutes and each request from a VU took 30 seconds to complete, then from each VU, there will be 2 requests per minute and 20 requests per minute from 10 VUs with a total of roughly 200 requests completed in the whole duration.

k6 is deployed on the Google Compute Platform and the testing results from *k6* are ingested into the InfluxDB[5], which is an open-source time-series database. Additionally, Grafana[6], an open-source analytic & monitoring solution, visualizes the queried data from the InfluxDB and presents it in real-time in a user-defined dashboard style.

The performance for each deployed strategy is evaluated by calculating the HTTP-request-duration (90-percentile), and the number of total requests served successfully. The response time for a HTTP request below which 90% of the response time values lie is called the 90-percentile (P90) response time, which means 90 percent of the requests are processed in a 90-percentile response time or less. This metric is important from the SLA point of view, where one wants to have most of the requests (90% in this case) completed before a certain time. We have evaluated each of the DevOps and NoOps strategies through different microservices and functions API endpoints across linearly increasing (continuously increasing) and random (random number of requests) user workload patterns.

[5] https://docs.influxdata.com/influxdb/v1.7/
[6] https://grafana.com/docs/grafana/latest/

Each test of an API endpoint is executed for 30 minutes. The total duration for which the metrics data is collected is set to `31 minutes` and the sampling rate is set to `10 seconds`, i.e, metrics values are aggregated for 10 seconds. The term unit time refers to the sampling interval in Section 4.

4 Results

To compare the performance of both *DevOps* and *NoOps* strategies, we focused on the 90-percentile response time of requests along with the number of requests served. The x-axis for each of the graph represents the unit time duration (each point corresponds to aggregated value over 10 seconds). In the section, we present our findings from different tests across different deployment strategies.

4.1 Cinema application

As part of this application, we only show the results from 2 API endpoints: 1) `/movies`, and 2) `/users/booking/username` for both the workload patterns, 2 deployment strategies and their 2 ways of achieving it. The rest of the API endpoints in this application are similar to `/movies` and hence are not presented.

4.1.1 /movies API Endpoint

The `/movies` API endpoint which is used to get all the movies stored in the Mongo database is used for demonstrating the evaluation. Fig. 5 shows the comparison results for this API endpoint for different deployment strategies along with two different methods by which these two can be achieved on two workload patterns for two metrics: number of requests served and P90 response time. Overall *DevOps* strategy compared to *NoOps* showcased the best performance results. *devops-gcp-hosted-kubernetes* along with *devops-self-hosted-kubernetes* achieved almost the similar performance on both types of workloads showcasing approximately a response time of `4.1ms` and average number of requests served per second as `13.65` on linearly increasing workload and `3.77ms` on random workload pattern with serving `9.98` number of requests per second. However, in case of the *NoOps* strategy, *noops-openWhisk* showcased the worst performance with the P90 response time of *1322ms* for linearly increasing workload and `780ms` on random workload pattern. Also, *noops-google-cloud-functions* compared to *DevOps* methods has a huge performance drop but is better than the *noops-openWhisk-functions* showcasing approximately a response time of `344.74ms` and average number of requests served per second as `10.52` on linearly increasing workload, and `339.94ms` on random workload pattern with serving `7.69` number of requests per second.

The high requests response times in the case of the *NoOps* strategy at the beginning of each test is due to the cold-start[] problem but afterward, it becomes stable. The performance drop in *noops-openWhisk-functions* compared to *noops-google-cloud-functions* can be attributed to the no availability of the

Fig. 5: `/movies` API endpoint performance comparison results for different deployment strategies along with two different methods by which these two can be achieved on two different workload patterns for two metrics: number of requests served and P90 response time.

resources underneath as virtual machines scaling was not enabled which might be possible in google cloud functions. Furthermore, finding optimal configuration parameters for each workload is required. One potential explanation of the huge performance difference between *NoOps* and *DevOps* strategies can be due to the big virtualization stack (FaaS platform) added inside *NoOps* for decreasing the operational tasks, which does not exists in *DevOps*.

4.1.2 /users/bookings API Endpoint

`/users/booking/username` API endpoint returns the movie booking information made by the user. This service is querying two other services. It takes `username` as the parameter and first queries the `/bookings/username` API endpoint to get the movie reservations made by the user and then based on that information it queries the */movies/movieid* service to get the movies information and return its JSON to the user. Fig. 6 shows the comparison results for this API endpoint for different deployment strategies along with two different methods by which these two can be achieved on two workload patterns. Overall as in the previous result, *DevOps* strategy compared to *NoOps* showcased the best performance results.

Although *devops-gcp-hosted-kubernetes* along with *devops-self-hosted-kubernetes* achieved almost the similar performance on both types of workloads but opposite to previous results they have a small difference of approximately `5ms` (*devops-self-hosted-kubernetes* taking a longer time to process requests) between the P90

Fig. 6: `/users/bookings` API endpoint performance comparison results for different deployment strategies along with two different methods by which these two can be achieved on two different workload patterns for two metrics: number of requests served and P90 response time.

response times for both the workloads while serving an almost equal number of requests per second. On the other hand, *noops-openWhisk-functions* could not handle the load after a certain number of requests and start taking more than a minute to process the requests which can also be seen in the Fig. 6 (3rd row, 1st column). However, *noops-google-cloud-functions* showcased better performance than *noops-openWhisk* but again like previous results have a huge performance overhead almost having `72x` more P90 response time than *Devops* deployments. This again can be attributed to the huge virtualization stack embedded into FaaS platforms which increases up the response times of the requests. Also, *NoOps* deployments have some initial high response times due to the cold start problem and we can also in between the test *noops-google-cloud-functions* in the case of linear workload a few spikes which most probably are due to the more function replicas being getting created.

Summary results from both these APIs are showcased in the Tablet 3.

4.2 Primes-python function

This function calculates prime numbers till 10000000 and is used for demonstrating the behavior of each of the deployment strategies on a compute-intensive microservice and function. We have converted this function to a microservice as well using the Python Flask framework and deployed it to each of the Kubernetes clusters in case of *DevOps* strategy. Fig. 7 shows the comparison results for this

Table 3: Summary results for *cinema* application showcasing HTTP P90 response time and the average number of requests served successfully for all the deployments methods in *DevOps* and *NoOps* at two user workload patterns.

API	Metrics	P90 Response Time (ms)				Avg. RPS			
	Type	DGCP	DSELF	NOW	NGCF	DGCP	DSELF	NOW	NGCF
/movies	Linear	**4.15**	4.20	1322.21	**344.74**	13.65	**13.66**	8.58	**10.52**
	Random	**3.77**	3.77	780.7	**339.94**	9.98	**9.98**	7.29	**7.69**
/ub*	Linear	**16.47**	22.04	59999.12	**1192.89**	13.46	**13.46**	1.65	**6.53**
	Random	**16.44**	20.90	60021.0	**1167.74**	9.78	**9.85**	2.080	**4.83**

function for different deployment strategies. It is clear from the figure that there is no winner. In the case of *DevOps* strategy, *devops-gcp-hosted-kubernetes* initially performed better by serving more number of requests with lesser response time but with the increase in the number of requests the scaling of replicas kicks in and the requests start to take longer time to respond. *devops-self-hosted-kubernetes* is not able to cope up with the high requests and larger compute requirements for the microservice, hence not able to scale properly and the number of requests served is much lesser than the one completed by *devops-gcp-hosted-kubernetes*. This points towards the load balancing and traffic re-distribution problem for *devops-gcp-hosted-kubernetes*.

On the other hand, in the case of *NoOps* strategy, both of the deployments suffer from the cold start problem and hence have longer response times in the beginning. However, *noops-google-cloud-functions* shows a constant request response time even with the increase in the number of requests for the rest of the test duration in both the workload patterns. This indicates that the *NoOps* deployment is more agile in terms of scalability and can provide a constant response time baring the initial cold starts. Furthermore, with the increase in time *noops-google-cloud-functions* was able to serve more number of requests than any other deployment strategy for this compute-intensive function.

4.3 Image-processing function

For demonstrating the access latency each deployment strategy adds up when accessing an object (in our work an image) stored on google cloud storage, `image-processing` function is used. Google cloud storage is an object store in buckets, which can store unstructured data such as photos, videos, log files, backups, and container images. This function first gets an image from the bucket and then basic operations like flipping, rotating, conversion to grayscale, and resize are performed. Like the previous function, this function as well is converted to microservice for deployment on the Kubernetes engine. Fig. 8 shows the comparison results for this function for different deployment strategies. *noops-google-cloud-functions* can serve a higher number of requests and at a lower

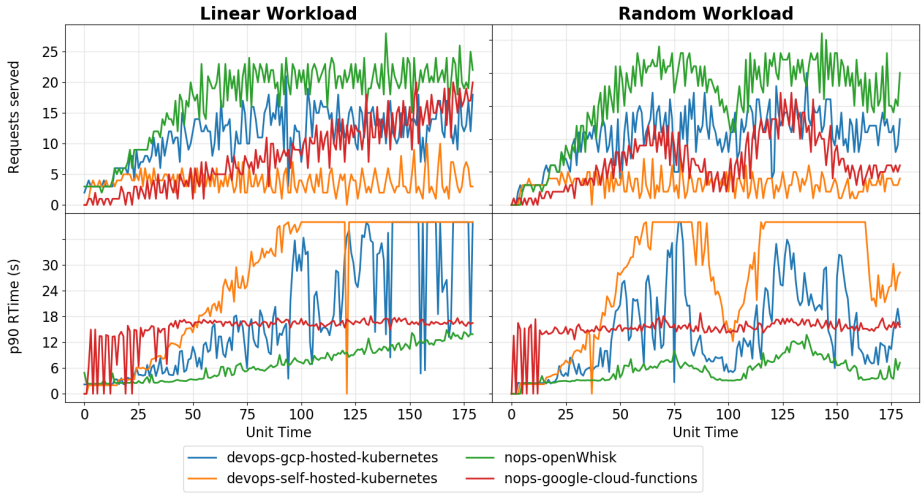

Fig. 7: `primes-python` function performance comparison results for different deployment strategies on two different workload patterns for two metrics: number of requests served and P90 response time.

response time for both the workloads therefore this deployment method is the clear winner in this case. *devops-gcp-hosted-kubernetes* has varied response times with the increase in the number of requests which again points towards the load balancing and traffic re-distribution problem in it. In the case of *devops-self-hosted-kubernetes*, the response time was continuously increasing with the number of requests which also suggests the scaling issues in it, while the *devops-gcp-hosted-kubernetes* can provide a constant requests response times even when the number of requests is increasing. This again indicates that the *NoOps* deployment is more agile in terms of scalability for this scenario as compared to *DevOps*. Therefore, *NoOps* deployment can be used for the applications requiring the constant requests latency. Though the *DevOps* strategy could also provide a constant request latency but finding the optimal scaling and configuration parameters is difficult and requires a prior load testing of the application.

4.4 Nodeinfo function

The `nodeinfo` function is a function to replicate a simple HTTP server endpoint. We utilize this function to test the base performance of each strategy and get an estimate of their performance on a basic web application. Fig. 9 shows the comparison results for this function for different deployment strategies.

Overall *DevOps* strategy compared to *NoOps* showcased the best performance results. *devops-gcp-hosted-kubernetes* along with *devops-self-hosted-kubernetes* achieved almost the similar performance on both types of workloads showcasing approximately a response time of `13ms` and average number of requests served per

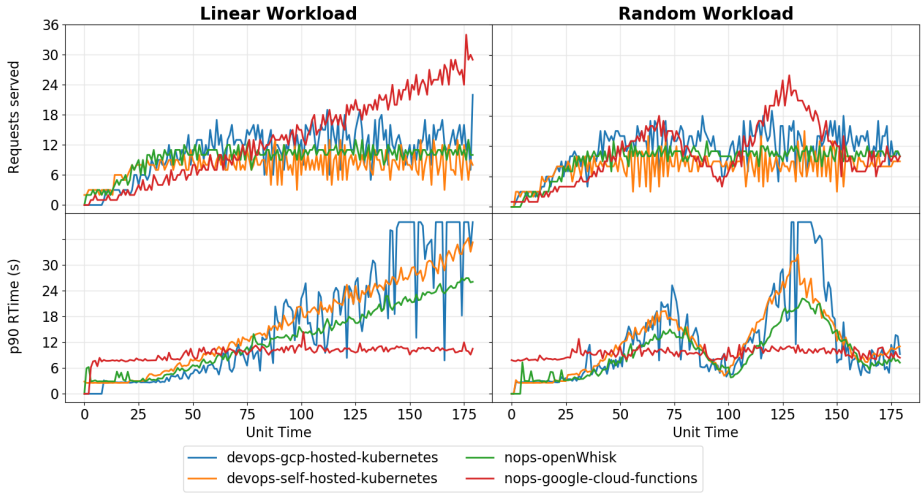

Fig. 8: `image-processing` function performance comparison results for different deployment strategies on two different workload patterns for two metrics: number of requests served and P90 response time.

second as `13.46` on linearly increasing workload and `12ms` on random workload pattern with serving `10` number of requests per second. However, there is a big performance difference when the function is executed using *NoOps* strategies, taking longer time to process the requests. From the initial P90 response times of the requests for both the workloads, it can be inferred that *noops-google-cloud-functions* has a big virtualization stack in comparison to the our own hosted *noops-openWhisk* FaaS platform as the requests took times more time to complete when executed on *noops-google-cloud-functions* than on *noops-openWhisk*. However, when the number of requests increases, *noops-google-cloud-functions* is able to cope up with scalability better than *noops-openWhisk*, which can also be seen from the P90 response time of *noops-google-cloud-functions* as `66.9ms` as compared to `789.42ms` for *noops-openWhisk* in case of linear workload. Similar behaviour can also be seen for the random workload pattern.

Summary results from both the above-discussed functions evaluations are showcased in the Tablet 4 with the best ones highlighted.

5 Discussion

To summarize, we have drawn five points of discussion mentioned below.

NoOps deployment strategy which is based on serverless computing suffers from the cold-start problem: From all the evaluations conducted above, one can see that in the case of *NoOps* whenever a function is triggered or invoked by a user request in the beginning there are always significant-high response times as compared to later. This problem is referred to as the *cold-start*

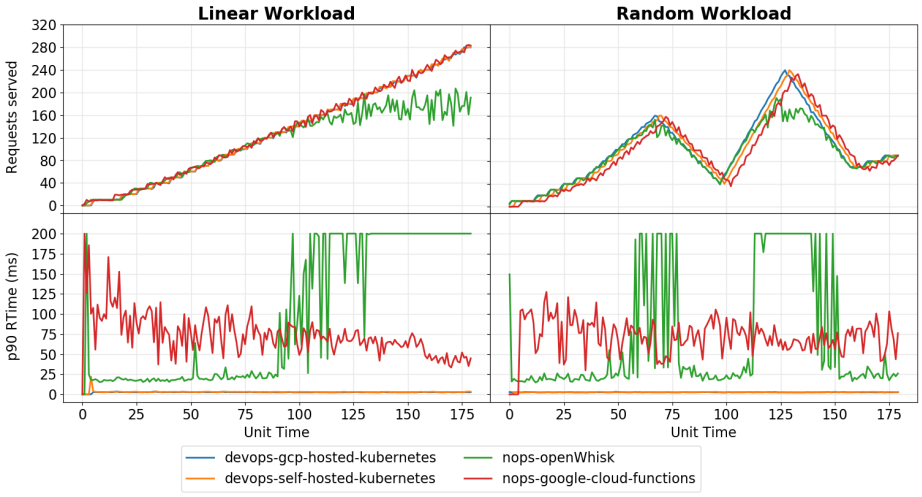

Fig. 9: `nodeinfo` function performance comparison results for different deployment strategies on two different workload patterns for two metrics: number of requests served and P90 response time.

problem. Whenever a function is invoked, it is deployed in a newly-initiated container and there is always a certain small period that a request needs to wait until the container is ready to serve. This wait is usually taken by the container to initialize the environment and pull the function source code which results in high response times. There already have been many kinds of research to decrease the cold start time like using pre-warmed containers [42], periodic warming consisting of submitting dummy requests periodically to induce a cloud service provider to keep containers warm [44] and pause containers [29]. Application developers need to keep this in consideration when deploying an application and decide based on the use case whether this deployment strategy is beneficial or not.

Google Cloud Functions can provide a stable latency baring the initial cold-starts: From the evaluations conducted, it can be seen that even when the workload is constantly increasing or randomly increasing and decreasing Google Cloud Functions were able to provide a near-constant P90 response time for the requests while the others could not. This can useful for the applications which need constant latency. However, if the function instances are not invoked for a certain time period then they will scale down to zero and the new requests will suffer from a cold-start. To avoid that, a dummy application can be created which will just send dummy requests to the functions for keeping them always warm.

DevOps deployment strategy outperforms when fetching simple web-based and database query requests: For the API calls where the requests are with the simple payload and invoked repetitively having the static or small size response then they should leverage a *DevOps* deployment due to high

Table 4: Summary results for function benchmarks showcasing HTTP P90 response time and the average number of requests served successfully for all the deployments methods in *DevOps* and *NoOps* at two user workload patterns.

Function	Metrics	P90 Response Time (ms)				Avg. RPS			
	Type	DGCP	DSELF	NOW	NGCF	DGCP	DSELF	NOW	NGCF
primes	Linear	26417.4	59999.2	**15810**	16810.8	1.13	0.38	**1.72**	0.84
	Random	21062.6	59999.12	**8590.0**	16299.5	1.06	0.32	**1.66**	0.65
imageP*	Linear	27694.4	29231.9	23650.1	**10429.8**	1.14	0.79	0.951	**1.38**
	Random	15710.0	21314.9	17570	**10123.1**	**1.05**	0.79	0.96	**1.15**
nodeinfo	Linear	**2.61**	2.92	789.42	66.9	**13.68**	13.67	11.29	13.24
	Random	**2.58**	2.64	260.45	72.7	**9.99**	**9.99**	9.19	9.66

performance, cost advantages and no cold-start problem. In the evaluations, we can see the decrements in P90 response times to almost 72%. *NoOps* deployment has some minimum overhead due to either the virtualization stack or the different involved components which is much more than what these usecases need as a result for such cases *DevOps* deployment methods should be preferred.

DevOps deployment strategy suffers from the load balancing and traffic re-distribution problem: Despite the cold start problem in the *NoOps* deployment, it performed stably after the initial period. In contrast, microservices deployment had a high peak of duration scattered randomly during different tests. One potential explanation is that these are due to the scaling out or scaling in of the pods triggered by horizontal pod autoscaling which resulted in the increase in the response time. If one needs a stable latency over the whole time, then one could choose deployment using *NoOps* deployment method.

NoOps deployment is more agile in terms of scalability: As we compare the scalability and agility of both the deployments, *NoOps* is better than *DevOps*. Since in the *DevOps* deployment, microservices starts to auto-scale only after the system has reached the defined criteria for at least one minute, there is always a delay of responsiveness to re-balance the current workload. As a result, there is an increase in response time with the increasing workload, then it drops after the new containers have been launched. Furthermore, if the underneath resources are not scaled than the requests will start to timeout as was in our evaluations. The granularity of the monitoring set can also limit the agility of the microservices scalability which is not the case with the *NoOps* deployment method. However, this disadvantage can be resolved by configuring a proper caching mechanism to store repetitive content but the user has to deal with more than what is required.

6 RELATED WORK

We present here the related work in threefolds, firstly, on the performance evaluation of *DevOps* based microservices deployment, secondly on the performance evaluation of *NoOps* based serverless deployment and lastly on the architectural decisions on selecting *DevOps* or *NoOps* deployment.

In [8], they introduced a four-step approach for the quantitative assessment of microservice architecture deployment alternatives. They found that in autoscaling cloud environments, careful performance engineering activities should be executed before additional resources are added to the architecture deployment configuration otherwise can result in significant performance degradation. In [4], three microservices design patterns practiced in the software industry are evaluated from the aspects of query response time, efficient hardware usage, hosting costs, and packet loss rate. They concluded that there is no single microservices pattern that is better than the others. Each design pattern performs better in different scenarios. Casalicchio and Perciballi [12] analyze the effect of using relative and absolute metrics to assess the performance of autoscaling. They have deduced that for CPU intensive workloads, the use of absolute metrics can result in better scaling decisions. Jindal et al. [22] addressed the performance modeling of microservices by evaluation of a microservices web application. They identified a microservice's capacity in terms of the number of requests to find the appropriate resources needed for the microservices such that, the system would not violate the performance (response time, latency) requirements. Kozhirbayev and Sinnott [24] present the performance evaluation of microservice architectures in a cloud environment using different container solutions. They also reported on the experimental designs and the performance benchmarks used as part of this performance assessment.

NoOps based on serverless computing topic has been researched recently to a great extent [27,14,10,23]. Baldini et al. [10] presents the general features of serverless platforms and discuss open research problems in it. Lynn et al. [28] discuss the feature analysis of enterprise based serverless platforms, including AWS Lambda, Microsoft Azure Functions, Google Cloud Functions, and OpenWhisk. Lee et al. [26] evaluated the performance of public serverless platforms for CPU, memory, and disk-intensive functions. They concluded that AWS Lambda outperforms other public cloud solutions. Similarly, Mohanty et al. [31] compared the performance of open-source serverless platforms Kubeless, OpenFaaS, and OpenWhisk. They evaluated the performance of each in terms of the response time and success ratio for function when deployed in a Kubernetes cluster. Shillaker [41] evaluates the response latency on OpenWhisk at different levels of throughput and concurrent functions. They proposed a way for improving startup time in serverless frameworks by replacing containers with a new isolation mechanism in the runtime itself. Pinto et al. [34] showcased the use of serverless in the field of IoT by dynamically allocating the functions on the IoT devices. Furthermore, researchers have identified the limitations of current serverless platforms, such as no control over specifying additional hardware resources like the required number of CPUs, GPUs, or other types of accelerators for the functions [19,9].

With the rise of *NoOps* based on serverless computing, *DevOps* based on microservices architecture is not the only choice when developing an application. There are debates about architecting decisions when it comes to choosing between these two. Jambunathan et al. [20] elaborated on the aspects of architecture decisions on microservices and serverless. From the service deployment's perspective, serverless has infrastructure restrictions that need native cloud service support and must be hosted by cloud service providers. In contrast, a microservices architecture could deploy on either the private data center or public cloud. However, the benefits of auto-scaling without considering complex server configuration is a deployment advantage on serverless than microservices.

In our previous work [15], we have compared microservices to serverless using AWS Lambda as the FaaS platform and a native cloud-native web application from the aspects of scalability, reliability, cost, and latency. We showcased the use cases where each of the deployment strategies can be used. But in this work, we have extended it by using OpenWhisk and Google Cloud Functions as FaaS platforms. Furthermore, we have added additional function benchmarks and applications for concluding the decisions.

7 Conclusion

Based on the experimental evaluations for *DevOps* and *NoOps* deployments, it can be concluded that no single type of deployment can fit all kinds of applications. For example, a GET and POST request to API endpoints fetching a response body by querying a database has a huge overhead in *NoOps* as compared to *DevOps*. On the other side, compute-intensive functions invocations were well served *NoOps* deployments due to its scaling agility as compared to *DevOps* where the user has to find the optimal scaling parameters and can suffer from latency in auto-scaling execution. Also, *NoOps* strategy provides immediate scalability and prompt response when handling random traffic, by which it can offer a constant latency.

In the end, this research derived a future research direction towards optimizing the deployment in terms of cost, performance, and application domain by building a hybrid deployment environment consisting of both the *DevOps* and*NoOps* deployment strategies. A deployment strategy is selected dynamically based on the workload pattern or load-balanced between the two.

8 Availability

The used applications, functions and the conducted evaluations are publicly available on GitHub under the link https://github.com/ansjin/devops_to_noops.git.

ACKNOWLEDGEMENTS

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 were provided by the Google Cloud Platform research credits. The authors also thank the anonymous reviewers whose comments helped in improving this paper.

References

1. Aws lambda – pricing. https://aws.amazon.com/lambda/pricing/, (Accessed on 07/30/2020)
2. Cloud functions overview. https://cloud.google.com/functions/docs/concepts/overview, (Accessed on 08/22/2020)
3. What is k6? https://k6.io/docs/, (Accessed on 07/28/2020)
4. Akbulut, A., Perros, H.G.: Performance analysis of microservice design patterns. IEEE Internet Computing **23**(6), 19–27 (2019)
5. Apache: Openwhisk cli (2017), https://github.com/apache/openwhisk/blob/master/docs/cli.md#openwhisk-cli
6. Apache: Openwhisk documentation (2017), https://openwhisk.apache.org/documentation.html
7. Architect: Project philosophy (2020), https://arc.codes/intro/philosophy, [Online; Accessed: 4-Feburary-2020]
8. Avritzer, A., Ferme, V., Janes, A., Russo, B., Schulz, H., van Hoorn, A.: A quantitative approach for the assessment of microservice architecture deployment alternatives by automated performance testing. In: Cuesta, C.E., Garlan, D., Pérez, J. (eds.) Software Architecture. pp. 159–174. Springer International Publishing, Cham (2018)
9. Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., et al.: Serverless computing: Current trends and open problems. In: Research Advances in Cloud Computing, pp. 1–20. Springer (2017)
10. Baldini, I., Castro, P.C., Chang, K.S., Cheng, P., Fink, S.J., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R.M., Slominski, A., Suter, P.: Serverless computing: Current trends and open problems. CoRR **abs/1706.03178** (2017), http://arxiv.org/abs/1706.03178
11. Bhojwani, R.: Design patterns for microservice-to-microservice communication - dzone microservices. https://dzone.com/articles/design-patterns-for-microservice-communication (Dec 2018), https://dzone.com/articles/design-patterns-for-microservice-communication
12. Casalicchio, E., Perciballi, V.: Auto-scaling of containers: The impact of relative and absolute metrics. 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W) pp. 207–214 (2017)
13. Di Francesco, P., Lago, P., Malavolta, I.: Migrating towards microservice architectures: An industrial survey. In: 2018 IEEE International Conference on Software Architecture (ICSA). pp. 29–2909 (April 2018). https://doi.org/10.1109/ICSA.2018.00012
14. Eivy, A.: Be wary of the economics of "serverless" cloud computing. IEEE Cloud Computing **4**, 6–12 (2017)
15. Fan., C., Jindal., A., Gerndt., M.: 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 - Volume 1: CLOSER,. pp. 204–215. INSTICC, SciTePress (2020). https://doi.org/10.5220/0009792702040215

16. Gancarz, R.: The economics of serverless computing: A real-world test. https://techbeacon.com/enterprise-it/economics-serverless-computing-real-world-test (2017), https://techbeacon.com/enterprise-it/economics-serverless-computing-real-world-test, [Online; Accessed: 23-March-2020]
17. GoogleCloud: Cloud functions execution environment. https://cloud.google.com/functions/docs/concepts/exec, (Accessed on 08/22/2020)
18. Handy, A.: Amazon introduces lambda, containers at aws re:invent. https://sdtimes.com/amazon/amazon-introduces-lambda-containers/ (2014), https://sdtimes.com/amazon/amazon-introduces-lambda-containers/, [Online; Accessed: 4-Feburary-2020]
19. Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Serverless computation with openlambda. In: 8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16) (2016)
20. Jambunathan, B., Yoganathan, K.: Architecture decision on using microservices or serverless functions with containers. In: 2018 International Conference on Current Trends towards Converging Technologies (ICCTCT). pp. 1–7 (March 2018). https://doi.org/10.1109/ICCTCT.2018.8551035
21. Jamieson, F.: Losing the server? (2017), https://www.bcs.org/content-hub/losing-the-server/
22. Jindal, A., Podolskiy, V., Gerndt, M.: Performance modeling for cloud microservice applications. In: Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering. p. 25–32. ICPE '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3297663.3310309, https://doi.org/10.1145/3297663.3310309
23. Jonas, E., Pu, Q., Venkataraman, S., Stoica, I., Recht, B.: Occupy the cloud: Distributed computing for the 99Cloud Computing. p. 445–451. SoCC '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3127479.3128601, https://doi.org/10.1145/3127479.3128601
24. Kozhirbayev, Z., Sinnott, R.O.: A performance comparison of container-based technologies for the cloud. Future Generation Computer Systems **68**, 175 – 182 (2017). https://doi.org/https://doi.org/10.1016/j.future.2016.08.025, http://www.sciencedirect.com/science/article/pii/S0167739X16303041
25. Lane, K.: Overview of the backend as a service (baas) space. API Evangelist (2015)
26. Lee, H., Satyam, K., Fox, G.: Evaluation of production serverless computing environments. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). pp. 442–450 (July 2018). https://doi.org/10.1109/CLOUD.2018.00062
27. Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., Pallickara, S.: Serverless computing: An investigation of factors influencing microservice performance. In: 2018 IEEE International Conference on Cloud Engineering (IC2E). pp. 159–169 (April 2018). https://doi.org/10.1109/IC2E.2018.00039
28. Lynn, T., Rosati, P., Lejeune, A., Emeakaroha, V.: A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms. In: 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). pp. 162–169 (Dec 2017). https://doi.org/10.1109/CloudCom.2017.15
29. Mohan, A., Sane, H., Doshi, K., Edupuganti, S., Nayak, N., Sukhomlinov, V.: Agile cold starts for scalable serverless. In: Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing. p. 21. HotCloud'19, USENIX Association, USA (2019)

30. Mohanty, S.K., Premsankar, G., di Francesco, M.: An evaluation of open source serverless computing frameworks. In: 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). pp. 115–120 (Dec 2018). https://doi.org/10.1109/CloudCom2018.2018.00033
31. Mohanty, S.K., Premsankar, G., di Francesco, M.: An evaluation of open source serverless computing frameworks. In: 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). pp. 115–120 (Dec 2018). https://doi.org/10.1109/CloudCom2018.2018.00033
32. Novoseltseva, E.: Benefits of microservices architecture implementation. https://dzone.com/articles/benefits-amp-examples-of-microservices-architectur (2017), https://dzone.com/articles/benefits-amp-examples-of-microservices-architectur, [Online; Accessed: 23-March-2020]
33. Pierre-Louis, M.A.: Openwhisk: A quick tech preview. DeveloperWorks Open, IBM, Feb **22**, 7 (2016)
34. Pinto, D., Dias, J.P., Sereno Ferreira, H.: Dynamic allocation of serverless functions in iot environments. In: 2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC). pp. 1–8 (Oct 2018). https://doi.org/10.1109/EUC.2018.00008
35. Reese, W.: Nginx: The high-performance web server and reverse proxy. Linux J. **2008**(173) (Sep 2008)
36. Richardson, C.: Introduction to microservices. https://www.nginx.com/blog/introduction-to-microservices/ (May 2015), https://www.nginx.com/blog/introduction-to-microservices/, [Online; Accessed: 25-January-2020]
37. Richardson, C.: Microservices pattern: Microservice architecture pattern. https://microservices.io/patterns/microservices.html (May 2019), https://microservices.io/patterns/microservices.html, [Online; Accessed: 28-January-2020]
38. Schneider, T.: Achieving cloud scalability with microservices and devops in the connected car domain. In: Software Engineering (2016)
39. Serverless: Documentation (2020), https://serverless.com/framework/docs/, [Online; Accessed: 4-Feburary-2020]
40. Shahrad, M., Balkind, J., Wentzlaff, D.: Architectural implications of function-as-a-service computing. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. pp. 1063–1075 (2019)
41. Shillaker, S., Pietzuch, P.R.: A provider-friendly serverless framework for latency-critical applications (2018)
42. Thömmes, M.: Squeezing the milliseconds: How to make serverless platforms blazing fast! https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e9951bd0 (2017), https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e9951bd0, [Online; Accessed: 14-Feburary-2020]
43. WG, C.S.: Cncf wg-serverless whitepaper v1. 0 (March 2018), https://gw.alipayobjects.com/os/basement_prod/24ec4498-71d4-4a60-b785-fa530456c65b.pdf, [Online; Accessed: 15-July-2020]
44. Şamdan, E.: Dealing with cold starts in aws lambda. https://medium.com/thundra/dealing-with-cold-starts-in-aws-lambda-a5e3aa8f532 (2018), https://medium.com/thundra/dealing-with-cold-starts-in-aws-lambda-a5e3aa8f532, [Online; Accessed: 14-Feburary-2020]