Microservices vs Serverless: A Performance Comparison on a Cloud-native Web Application

Chen-Fu Fan, Anshul Jindal[a] and Michael Gerndt
Chair of Computer Architecture and Parallel Systems, TU Munich, Garching, Germany
{jeff.fan, anshul.jindal}@tum.de, gerndt@in.tum.de

Keywords: Microservices, Serverless, Performance Comparison, Cloud-native Applications, Cloud Computing.

Abstract: A microservices architecture has gained higher popularity among enterprises due to its agility, scalability, and resiliency. However, serverless computing has become a new trendy topic when designing cloud-native applications. Compared to the monolithic and microservices, serverless architecture offloads management and server configuration from the user to the cloud provider and let the user focus only on the product development. Hence, there are debates regarding which deployment strategy to use.
This research provides a performance comparison of a cloud-native web application in terms of scalability, reliability, cost, and latency when deployed using microservices and serverless deployment strategy. This research shows that neither the microservices nor serverless deployment strategy fits all the scenarios. The experimental results demonstrate that each type of deployment strategy has its advantages under different scenarios. The microservice deployment strategy has a cost advantage for long-lasting services over serverless. On the other hand, a request accompanied by the large size of the response is more suitably handled by serverless because of its scaling-agility.

1 INTRODUCTION

With the wide adoption of cloud computing, enterprises have migrated or refactored their existing monolithic-based applications into the microservices architecture (Di Francesco et al., 2018). Microservices architecture based applications have higher agility since it decouples a monolithic-based application into smaller services and each service can then be deployed separately either on a virtual machine or in a container where the resources can be scaled on-demand. This migration has not only affected the application's architecture but also the team's structure within an organization (Mazlami et al., 2017).

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 some data security concerns during communication. Also, research shows that the development team with a strong DevOps culture may indeed get benefit from the microservices architecture, therefore the effort to establish DevOps culture is another consideration for adopting a microservices architecture (Schneider, 2016).

On the other hand, serverless computing has gained higher popularity and more adoption in different fields since the launch of AWS Lambda in 2014 (Handy, 2014). Compared to the monolithic or the microservices architecture, a serverless architecture releases the effort of server management from the application developers where they now just have to focus on the application logic (Castro et al., 2019). In other words, DevOps are free from operations work and can purely focus on development. In addition, 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 (Gancarz, 2017). 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 a number of instances.

Both microservices and serverless computing 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 refactored into both microservices and serverless deployment models from the aspect of scalability, reliability, cost,

[a] https://orcid.org/0000-0002-7773-5342

and latency. The experimental results demonstrate that the microservice deployment strategy has a cost advantage for long-lasting services over serverless. On the other hand, a request accompanied by the large size of the response is more suitably handled by the serverless because of its scaling-agility.

The main contribution of this paper is the performance comparison between microservices and serverless deployment in terms of scalability, reliability, cost, and latency. Further, we provide architecture recommendations based on the type of load, scenario, and the request size.

The rest of this paper is composed as follows. Section 2 discusses the background knowledge required for this paper in brief. Section 3 studies the related works. Section 4 provides the overall system design. Section 5 provides the evaluation strategy and experimental configuration details and, also showcase the results of the conducted analysis. Section 6 summarizes the discussion of the results and lastly, Section 7 concludes the paper.

2 BACKGROUND

2.1 Microservices-based Architecture

Monolithic architecture is one of the most widely used design patterns for enterprise applications. From a modularization abstraction perspective, the characteristics are hardware resource-sharing, and the internal executable is mutually dependent (Kratzke, 2018). Developers could efficiently conduct end-to-end testing on this type of architecture with automation tools. On the other hand, the maintenance, bug-fixing, technology refactoring, and scaling specific resources are the drawbacks of this architecture (Kazanavičius and Mažeika, 2019). In contrast to monolithic architecture, microservices architecture design is a more loose-coupled style (Bhojwani, 2018). 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 benefits of a microservices architecture are improved fault tolerance, flexibility in using technologies and scalability and speed up of the application (Novoseltseva, 2017). 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 (Bhojwani, 2018).

2.1.1 Microservices-based deployment strategy

Deploying a monolithic architecture is relatively straight-forward than other architectures where developers deploy the whole application on a single physical or virtual server (Richardson, 2019b). If an application is required to be deployed on multiple servers, a common way is to deploy the same application multiple times on each server and then load balance it using a load balancer. The principle of microservices architecture is loose-coupling, which requires multiple service instances for an application (Richardson, 2015). This can be achieved either by deploying each service on a separate virtual machine instance or deploying a service per container or one can even combine multiple services per virtual machine and container. The containerization deployment benefits from the higher deployment speed, agility, and lower resources consumption (Richardson, 2019a). 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 AWS Elastic Container Service[2] can be used for managing the containers.

2.2 Serverless-based Architecture

Compared to the monolithic and microservices architecture, serverless architecture does not require the management of underlying infrastructure (Amazon, 2018). Although serverless architecture still requires infrastructure to execute programs, all the tasks, including infrastructure management and operation, autoscaling, and maintenance, shift to the cloud service providers. Since there are no reserved instances for the serverless, the cost is charged on the number of requests received to the functions and the time it takes for the code to execute (Gancarz, 2017). In general, one of the advantages of serverless architecture is the less total cost of ownership by paying as you run.

2.2.1 Serverless-based deployment strategy

Most of the configuration regarding physical servers, containers, and scalability for the deployment of a serverless application does not require the developer's attention (Amazon, 2018). A serverless platform accepts the application source code as an input along with a deployment specification to describe the functions, APIs, permissions, configurations, and events

[1]https://kubernetes.io/docs/

[2]https://aws.amazon.com/ecs/

that make up a serverless application through an interface which can be a CLI or web interface or using some frameworks like Serverless[3] and Architect[4]. This specification is then further used by the framework to build and deploy the application on the serverless platform. Almost all cloud service providers provide cloud-based IDE or plugins to the popular IDEs for the development and deployment of the serverless applications. For instance, AWS provides Serverless Application Model (AWS SAM), an open-source framework that is used to build serverless applications on AWS (AWS, 2020b). It also provides a local environment that allows the developers to test and debug their applications locally before deploying it to the cloud. Each serverless execution platform also provides function execution logs which further can be used by the developers for debugging. In addition, there is user request tracing tools like AWS X-Ray which enables developers to trace the requests and debug potential problems (AWS, 2020a; Lin et al., 2018).

3 RELATED WORK

We present here the related work in threefolds, firstly, on the performance evaluation of microservices, secondly on the performance evaluation of serverless and lastly on the architectural decisions on selecting microservices or serverless deployment. Casalicchio and Perciballi (Casalicchio and Perciballi, 2017) 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. (Jindal et al., 2019) 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 (Kozhirbayev and Sinnott, 2017) 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.

Nowadays serverless computing has become a hot topic in the research (Lloyd et al., 2018; Eivy, 2017; Baldini et al., 2017; Jonas et al., 2017). Baldini et al. (Baldini et al., 2017) presents the general features of serverless platforms and discuss open research problems in it. Lynn et al. (Lynn et al., 2017) discuss the feature analysis of enterprise based serverless platforms, including AWS Lambda, Microsoft Azure Functions, Google Cloud Functions and OpenWhisk. Lee et al. (Lee et al., 2018) evaluated the performance of public serverless platforms for CPU, memory and disk-intensive functions. Similarly, Mohanty et al. (Mohanty et al., 2018) 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. Pinto et al. showcased the use of serverless in the field of IoT by dynamically allocating the functions on the IoT devices (Pinto et al., 2018).

With the rise of serverless computing, microservices architecture is not the only choice when modernizing a monolithic application. There are debates about architecting decisions when it comes to choosing serverless or microservices. Jambunathan et al. (Jambunathan and Yoganathan, 2018) 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.

However, none of the works specifically address the comparison of microservices and serverless deployment from the aspects of scalability, reliability, cost, and latency on a cloud-native web application.

4 SYSTEM DESIGN

4.1 Application Overview

The application used as part of this research is an employee time-sheet management portal where the time an employee has worked on a project is recorded. It is developed in Node.js and consists of following 3 main modules:

4.1.1 Favourite Projects

It consists of the projects which are marked favourite by the user. When a user visits this module, it triggers a GET request to fetch all the projects and display

[3]https://serverless.com/framework/docs/

[4]https://arc.codes/

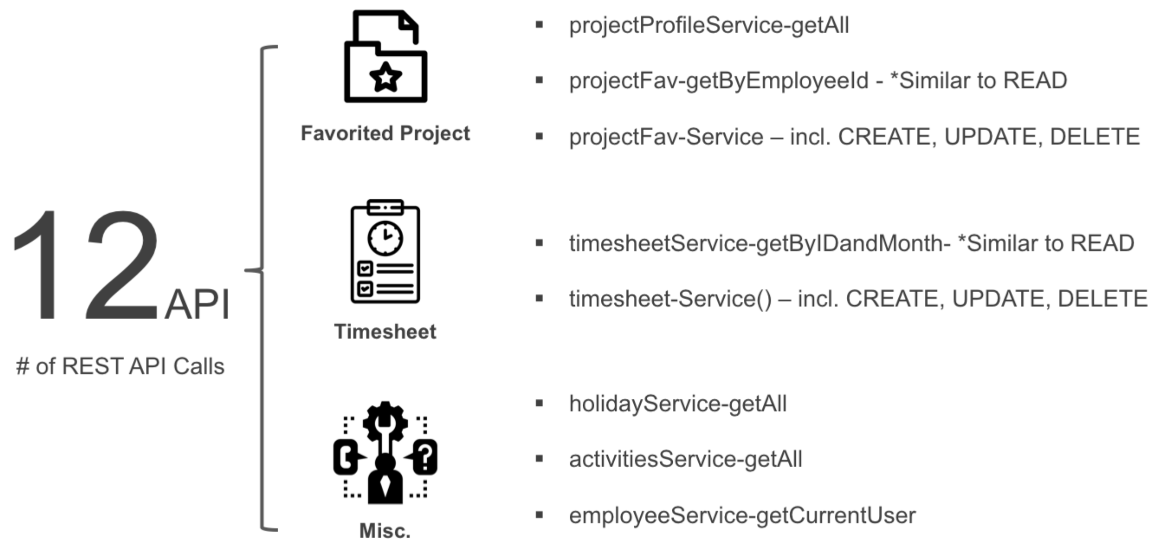

12 API
# of REST API Calls
Favorited Project
projectProfileService-getAll
projectFav-getByEmployeeId - *Similar to READ
projectFav-Service – incl. CREATE, UPDATE, DELETE
Timesheet
timesheetService-getByIDandMonth- *Similar to READ
timesheet-Service() – incl. CREATE, UPDATE, DELETE
Misc.
holidayService-getAll
activitiesService-getAll
employeeService-getCurrentUser

Figure 1: Overview of APIs present in the application.

them in a table view. Then the system invokes another POST request with the user's ID to get the user's favourite projects. Afterward, the system displays all the projects and the favourites are marked in a different color for distinction. A user can conduct CRUD (Create, Read, Update, and Delete) operations on each of the project entity for adding, reading, updating and deleting the favourite projects.

4.1.2 Timesheet

This module is used for recording and viewing the time the user has spent on a project. When a user visits this the page, the system invokes a POST request with the user's ID to get a list of favorite projects along with the time the user has spent on each of them. This information is then displayed in a calendar style's grid matrix to present the user's working hours in each cell. Users can conduct CRUD(Create, Read, Update, and Delete) operation on each project entity which is then executed with the database.

4.1.3 Miscellaneous

Both Favorite Projects and Timesheet modules require the user's information. Also, each region's holiday information is necessary when displaying a calendar style's grid matrix. Meanwhile, a list of activities that represent what type of role the user has played when doing projects is also required. All this information is handled by this module.

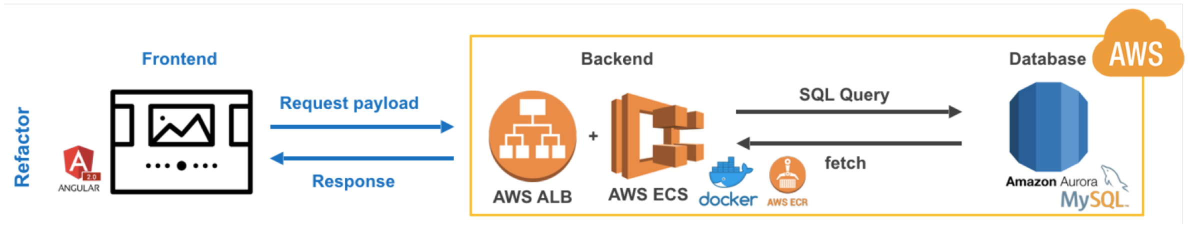
To sum up, there are 12 API calls which are invoked in the application and studied in this research, these APIs are shown in Fig. 1.

4.2 Deployment Strategies

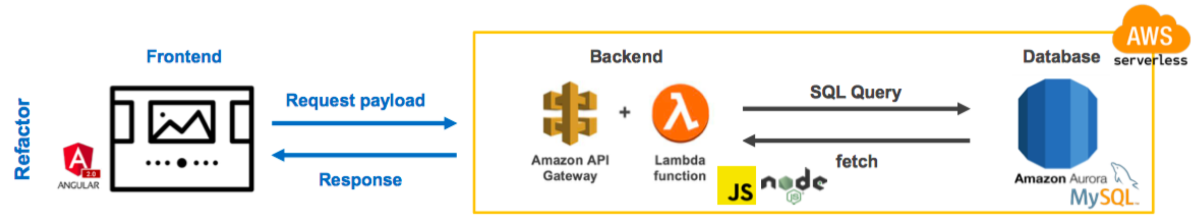
The above-discussed application is deployed using the two strategies: Microservices and Serverless. For both of these strategies, the front-end user interface has remained the same, the AWS Relational Database System with the Aurora cluster database is used as the main database for storing the data. However, the back-end server is refactored accordingly for each of the strategies. The overview of the system when deployed using each of the strategies is shown in Fig. 2. Below subsections describe the back-end deployment for each strategy in more detail.

4.2.1 Microservices

The microservices back-end is deployed using containerized instances and leverages Amazon Elastic Container Service(ECS) to orchestrate all of the container instances as shown in Fig. 2a. Amazon ECS is a highly scalable, high-performance container orchestration service that supports Docker containers and allows the users to run and scale containerized applications on AWS. It is integrated with Amazon Elastic Container Registry(ECR) where the updated containerized-images are uploaded and ECS service automatically pulls the latest images from it. In order to handle multiple requests, the ECS cluster is integrated with an application load balancer(ALB) that distributes the traffic to the containers. Fig. 3a shows a further detailed overview of the deployed back-end

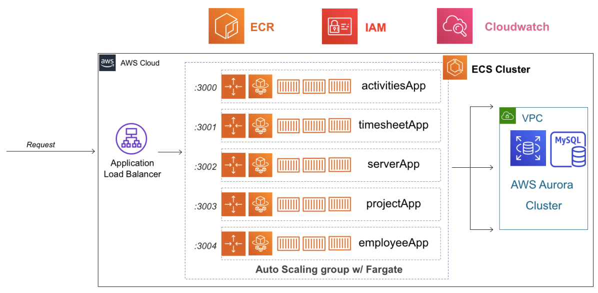


(a) Overview of system when deployed using microservices strategy using AWS Application Load Balancer and Elastic Container Services.

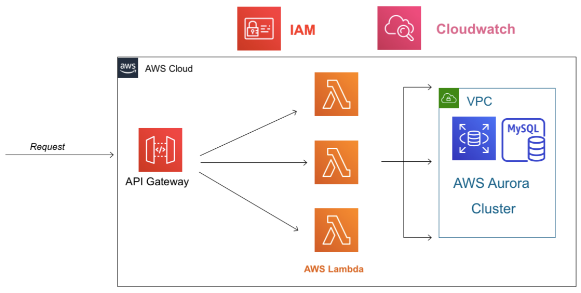


(b) Overview of the system when deployed using serverless strategy using AWS API gateway and Lambda functions.

Figure 2: Overview of the system when deployed using different strategies.



(a) Deployed back-end in microservices.



(b) Deployed back-end in serverless.

Figure 3: Detailed overview of the deployed back-end architecture in each of the strategy.

architecture in this strategy. It has 5 containers for 5 main cases (project, timesheet, activities, server, employee). Table 1 shows the per API call the container used and its port configuration.

4.2.2 Serverless

The serverless back-end is deployed using AWS Lambda and AWS API Gateway, as shown in Fig. 2b. AWS API Gateway serves as an API access point that directs the incoming requests to the right AWS Lambda function. AWS Lambda function then executes a customized business logic after parsing the request and fetching the data from the database resources. As part of this deployment, it is not necessary to assign specific ports since a specific path and resource are designated. AWS Lambda is then equipped with an auto-scaling policy for scaling on-demand with the workload. Fig. 3b shows a further detailed overview of the deployed back-end architecture in this strategy. In the current implementation, we have 6 lambda functions and the Table 1 shows the lambda functions with their API calls.

5 Evaluation

We have evaluated the performance of both the deployment strategies. The following subsections elaborate more detail about the infrastructure settings, evaluation system, and performance metrics used for the evaluation.

5.1 Infrastructure setting

For the AWS ECS setting on the microservices strategy, we allocated each container instance 0.25 CPU core (equal to 256 CPU units) and 512 MB of memory. For maintaining the desired state of the containers, the auto-scaling of ECS is also enabled. For each container's scale-out policy, the system automatically adds two more instances into the cluster when the

Module	**API Call**	**Microservices (Container:port)**	**Serverless (Lambda function)**
Favorite Projects	projectProfileService-getAll	project:3003	projectProfileService-getAll
	projectFavService-Create	project:3003	projectFavService
	projectFavService-Read	project:3003	projectFavService
	projectFavService-Update	project:3003	projectFavService
	projectFavService-Delete	project:3003	projectFavService
Timesheet	timesheetService-Create	timesheet:3001	timesheetService
	timesheetService-Read	timesheet:3001	timesheetService
	timesheetService-Update	timesheet:3001	timesheetService
	timesheetService-Delete	timesheet:3001	timesheetService
Miscellaneous	activityService-getAll	activities:3000	activitiesService-getAll
	holidayService-getAll	server:3002	holidayService-getAll
	employeeService-getCurrentUser	employee:3004	employeeService-get

Table 1: Overview of the API calls, microservices containers, their ports and serverless functions which are part of each module.

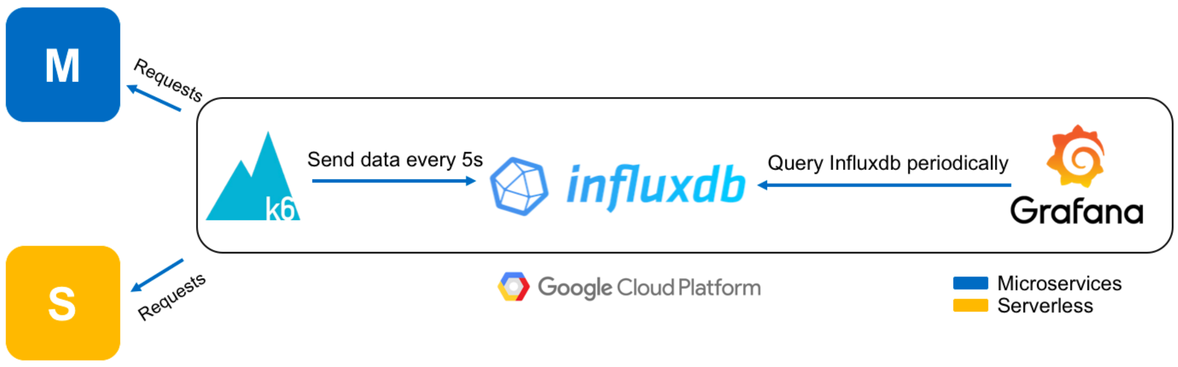


Figure 4: Overall evaluation system design for evaluating each of the strategies using the k6 as the load generator, influxdb for storing the data and grafana for real time visualization deployed on the Google Cloud Platform.

average CPU utilization goes above 50% for a consecutive 1 minute. In contrast, the scale-in policy is configured to remove two instances from the clusters once the average CPU utilization is lower than 30% for a consecutive 1 minute. The cooldown period is set to 30 seconds for both the scaling policies to prevent over-provisioning. For both scaling scenarios, to distribute the incoming traffic to a dedicated container group for achieving load balancing a load balancer is also used. It automatically registers or deregisters the newly joined containers into their target group. The Microservices strategy consists of five different container entities. Each of them is responsible for different API calls and is designated to dedicated ports.

The serverless strategy is also deployed on AWS and leverages AWS Lambda and API Gateway. The business logic of each Lambda function is written in Javascript and is run with Node.js version 10.10. Each function is allocated with 512MB memory space. We have not set a specific concurrency number for each Lambda function, however, AWS set a concurrency quota of 1000 for each user account as a soft limitation for this service. The AWS API Gateway acts as the entry point to navigate the incoming request to the right Lambda function.

In order to minimize the experiment's performance deviation, both microservices and serverless are connected to the same database cluster deployed on AWS. The database used is the AWS Aurora cluster with the MySQL engine (version. 5.7.12).

5.2 Evaluation system

Our evaluation strategy is implemented via a load testing tool - k6[5] and Fig. 4 shows the overall evaluation system design. *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, serverless, containers, and websites. *k6* generates different patterns of the user workload to the deployed microservices and serverless system. It is deployed on the

[5]https://k6.io/

Google Compute Platform and the testing results from *k6* are ingested into the InfluxDB[6], which is an open-source time-series database. Furthermore, Grafana[7], 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.

5.3 Performance testing metrics

The performance is evaluated by calculating the HTTP-request-duration (min, mean, median, max, and percentile95), the total number of requests *k6* has sent, the number of total requests served successfully and cost. The cost is referenced from AWS billing estimation and derived from the pricing model of AWS ECS and Lambda. In this research, the free-tier quota of each building block is not taken into consideration.

For both microservices and serverless strategies, we have evaluated each of them through 12 API calls across incremental (continuously increasing), random (random number of requests), and triangle (continuously increasing till half time and then continuously decreasing for the remaining time) user workload patterns. As a result, there are 72 (12 APIs * 3 load patterns * 2 deployment strategies) API tests executed for the evaluation. Each test of an API is executed for 15 minutes. In addition, there is a 3 minutes cool down period between each API's test and between each workload pattern's test there is a 30 minutes pause to make sure that the testing infrastructure achieves a normal state.

6 Experiment Results

To compare the performance of microservices and serverless strategy, we focused on the percentile95 response time of requests for both the deployment strategies. The x-axis represents the testing duration from 0 to 15 minutes, and the y-axis is the percentile95 duration of HTTP-request in milliseconds.

6.1 Favorite Projects

The *projectProfileService-getAll* API call which is used to get all the projects is a relatively static request which is always invoked when a user visits this module. From the Fig. 5, The serverless strategy has a better performance in terms of performance stability and scaling-agility. For the microservices strategy, we found that the response time starts to rise with an increase in the workload. However, after the scale-out, the response time has decreased. In the case of the serverless strategy, there is also high response times at the beginning of each test due to the cold-start problem but afterward, it becomes stable. But the serverless has a lesser duration of high request response time as compared to microservices.

The *projectFav-Read* is the consecutive API call that fetches user's favorite projects and displays them in the front-end. It is apparent that the serverless strategy here also provides better performance with respect to fast responsiveness and performance stability. Moreover, we immediately identify that the microservices have a regular peaking response time, which often happens on the dot of a minute. One of the potential explanations would be its monitoring granularity, a minute-level monitoring period. So every time it starts to scale with more container instances, the redistribution of traffic results in particular high peaking response time as shown in Fig. 5

Table 2 shows the summary results for the Favourite Projects module. It shows that the cold start problem of serverless resulted in a significantly higher *minimum response time* than microservices. Nevertheless, the serverless strategy still has *cost* advantages as compared to microservices. Also, it still provides a higher *average number of requests served per second (Avg. RPS)* than microservices in all the scenarios.

6.2 Timesheet

For the create and update operations in the case of the Timesheet module, the serverless strategy provides better performance in terms of responsiveness and performance stability as shown in Fig. 6. In contrast, the microservices strategy provided an unstable performance on the incremental and triangle workload scenarios. However, the microservices outperforms serverless when executing the *delete* operation. The serverless strategy as in the previous case suffers from cold-start during the first 2 minutes of the evaluation period.

Table 3 shows the summary results for the *Timesheet* module. It shows that the serverless strategy provides better response time in *create, read, and update* operations. But for the *delete* operation, microservices performs better than serverless. Even though both strategies have similar performance with respect to the average number of requests served per second, it shows that in this case, microservices are more cost-effective than the serverless. It could be due to the fact that the minimum cost required for setting up serverless is more than what is required in the case for setting

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

[7]https://grafana.com/docs/grafana/latest/

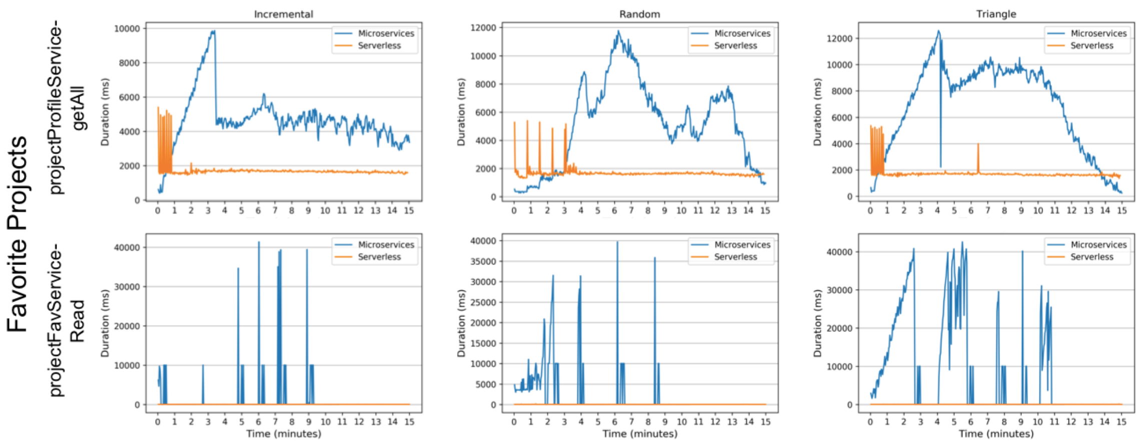


Figure 5: Comparison of percentile95 response time in Favorite Projects for both microservices and serverless deployment at different workload patterns.

API	Metrics	Min		Mean		Max		Avg. RPS		Cost	
	Type	M	S	M	S	M	S	M	S	M	S
projectProfileService -getAll	Incremental	**21.2**	1160.0	2930.0	**1410.0**	10060.0	**5680.0**	13.8	**26.8**	0.0062	**0.0054**
	Random	**23.0**	1110.0	2990.0	**1400.0**	11900.0	**5390.0**	11.0	**23.1**	0.0093	**0.0047**
	Triangle	**23.1**	1150.0	4100.0	**1430.0**	12580.0	**5350.0**	9.4	**24.7**	0.0062	**0.0050**
projectFavService -Read	Incremental	**20.1**	26.7	582.1	**32.7**	41400.0	**252.3**	27.5	**42.1**	**0.0062**	0.0076
	Random	**20.1**	25.9	503.1	**29.7**	39730.0	**248.8**	23.6	**35.0**	0.0093	**0.0063**
	Triangle	**20.1**	27.2	976.5	**33.8**	42810.0	**267.0**	19.6	**40.0**	0.0093	**0.0072**

Table 2: Summary results for Favorite Projects module showcasing HTTP-request-duration (minimum, mean, median, and maximum) the average number of requests served successfully and cost for both microservices and serverless deployment at different workload patterns.

up microservices deployment.

6.3 Miscellaneous

Fig.7 represents *activityService-getAll*, *holidayService-getAll* and *employeeService-getCurrentUser* API calls respectively in the *Miscellaneous* module. All of them are relatively simple, small and static requests. Here, the microservices strategy outperforms the serverless in most of the cases. In addition, microservices deployment strategy provides faster response times than serverless, this could again be due to the minimum overhead of the virtualization stack required in serverless deployment. Serverless strategy was only better in *employeeService-getCurrentUser* with the incremental load pattern.

The summary table 4 also shows that the microservices strategy has somewhat better or equivalent performance as compared to the serverless with respect to the average number of requests served per second. However, again microservices strategy is more cost-effective than serverless and serverless has higher minimum response time due to cold start.

7 Discussion

To summarize, we have drawn four points of discussion mentioned below :

1. **Serverless strategy suffers from the cold-start problem** Whenever a function is triggered or invoked by a user request the AWS lambda function 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. This phenomenon is referred to as the *cold-start* problem. There already have been many researches to decrease the cold start time like using pre-warmed containers (Thömmes, 2017), periodic warming consisting of submitting dummy requests periodically to induce a cloud service provider to

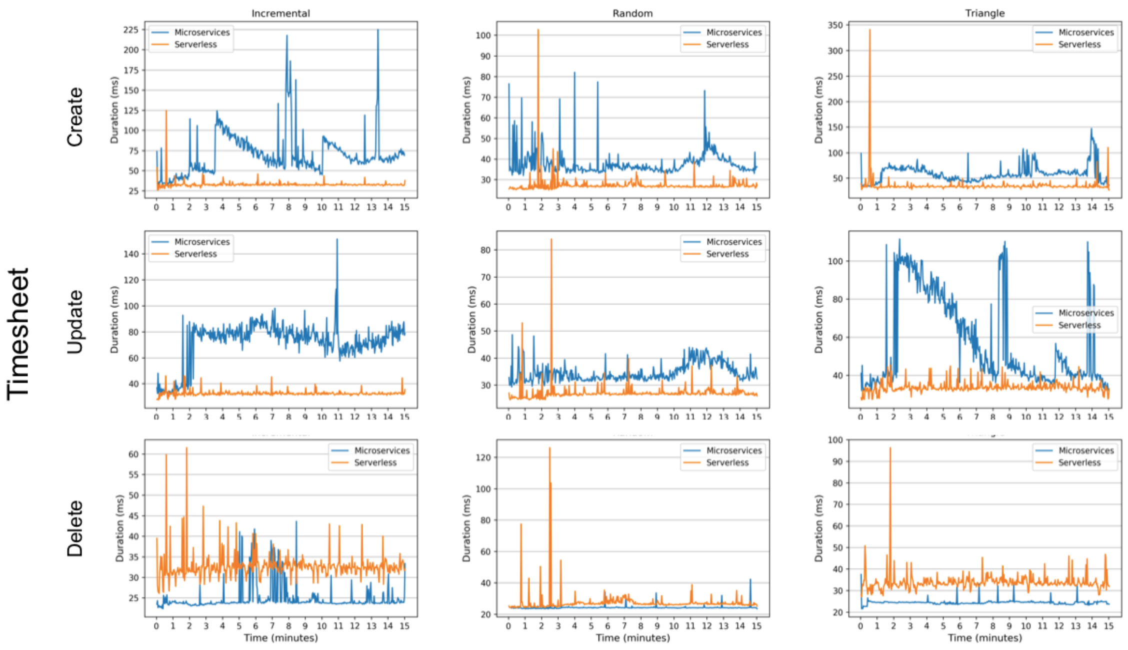


Figure 6: Comparison of percentile95 response time in Timesheet module for both microservices and serverless deployment at different workload patterns.

API	Metrics	Min		Mean		Max		Avg. RPS		Cost	
	Type	M	S	M	S	M	S	M	S	M	S
timesheetService -Create	Incremental	27.3	**23.7**	46.0	**28.1**	279.1	**244.2**	42.1	42.1	**0.0062**	0.0076
	Random	27.2	**22.6**	32.6	**25.3**	286.8	**262.4**	35.0	35.0	**0.0031**	0.0063
	Triangle	27.7	**23.7**	40.5	**28.9**	496.2	**340.8**	40.0	40.0	**0.0031**	0.0072
timesheetService -Update	Incremental	25.3	**23.5**	48.0	**27.9**	**300.6**	310.3	42.1	42.1	**0.0062**	0.0076
	Random	24.5	**22.8**	29.9	**25.5**	**249.7**	323.4	35.0	35.0	**0.0031**	0.0063
	Triangle	26.3	**24.0**	36.2	**29.0**	**262.2**	300.4	40.0	40.0	**0.0062**	0.0072
timesheetService -Delete	Incremental	**20.6**	23.8	**22.5**	28.1	246.5	**233.6**	42.1	42.1	**0.0031**	0.0076
	Random	**21.3**	22.6	**23.1**	25.3	**254.1**	294.5	35.0	35.0	**0.0031**	0.0063
	Triangle	**20.5**	24.6	**22.4**	29.2	**247.3**	481.2	40.0	40.0	**0.0031**	0.0072

Table 3: Summary results for Timesheet module showcasing HTTP-request-duration (minimum, mean, median, and maximum) the average number of requests served successfully and cost for both microservices and serverless deployment at different workload patterns.

keep containers warm (Şamdan, 2018) and pause containers (Mohan et al., 2019). DevOps 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.

2. **Microservices deployment strategy suffers from the load balancing and traffic redistribution problem** Despite the cold start problem in the serverless deployment, it performed stably after the initial period. In contrast, microservices deployment had a high peak of duration scattered randomly during each test. One potential explanation is that these peaks coincide with scaling out or scaling in time of the 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 serverless computing.

3. **Microservices deployment strategy outperforms when fetching small size and repetitive**

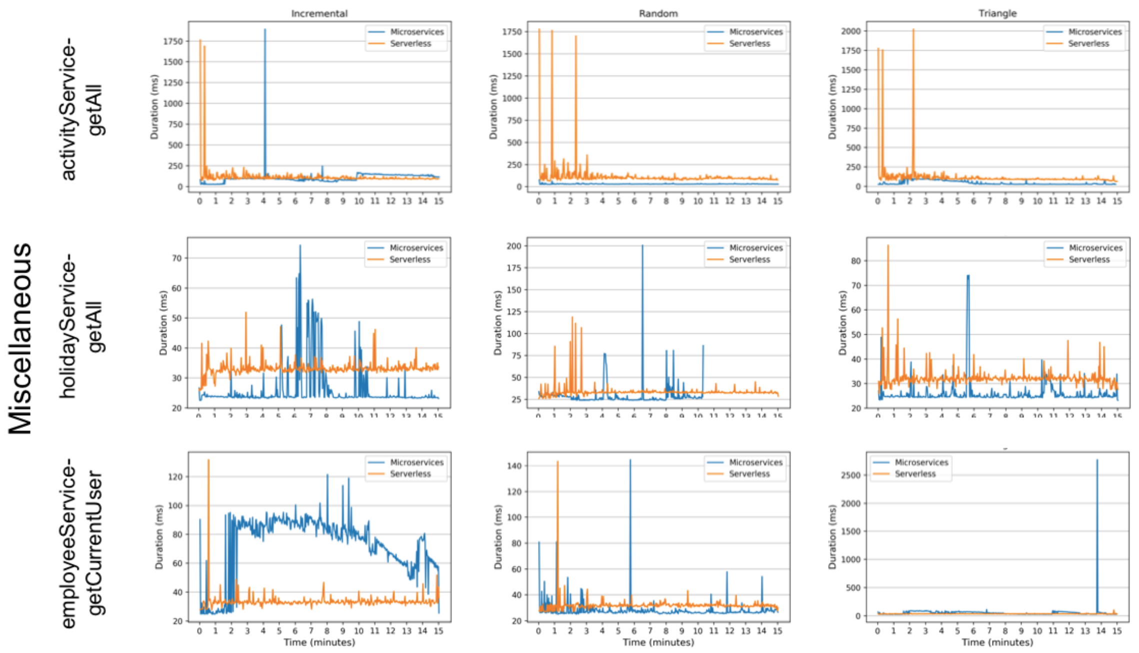


Figure 7: Comparison of percentile95 response time in Miscellaneous module for both microservices and serverless deployment at different workload patterns.

API	Metrics	Min		Mean		Max		Avg. RPS		Cost	
	Type	M	S	M	S	M	S	M	S	M	S
activityService -getAll	Incremental	**22.8**	36.4	**69.0**	69.7	1900.0	**1790.0**	42.1	42.1	**0.0031**	0.0076
	Random	**21.2**	37.4	**26.4**	65.4	**255.0**	1800.0	35.0	35.0	**0.0031**	0.0063
	Triangle	**20.6**	35.4	**28.7**	**67.4**	228.9	2330.0	40.0	**39.9**	**0.0031**	0.0072
holidayService -getAll	Incremental	**20.5**	24.0	**22.6**	28.6	**252.1**	299.4	42.1	42.1	**0.0031**	0.0076
	Random	**21.3**	24.6	**24.0**	28.7	367.9	**334.7**	35.0	35.0	**0.0031**	0.0063
	Triangle	**21.4**	23.4	**23.4**	28.2	**249.5**	3530.0	40.0	40.0	**0.0031**	0.0072
employeeService -getCurrentUser	Incremental	**20.8**	23.8	**37.4**	28.6	**246.7**	382.0	42.1	42.1	**0.0062**	0.0076
	Random	**20.8**	23.0	**24.8**	27.2	270.4	**215.9**	35.0	35.0	**0.0031**	0.0063
	Triangle	**22.0**	23.9	**29.2**	28.5	2700.0	**260.0**	40.0	**39.9**	**0.0031**	0.0072

Table 4: Summary results for Miscellaneous module showcasing HTTP-request-duration (minimum, mean, median, and maximum) the average number of requests served successfully and cost for both microservices and serverless deployment at different workload patterns.

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 microservices deployment due to cost advantages and no cold-start problem. Serverless deployment has some minimum overhead due to either the virtualization stack or the different involved components which is more than what these cases need as a result for such cases microservices deployment should be preferred.

4. **Serverless deployment is more agile in terms of scalability**

   As we compare the scalability and agility of both the deployments, serverless is better than microservices. Since the microservices deployment 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. In the end, the granularity of monitoring set in minute-level limits the agility of the microservices scalability which is not the case with the serverless deployment. 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.

8 Conclusion

Based on the experimental evaluation for microservices and serverless deployments, it proves that no single type of deployment could fit all kinds of applications. For example, a POST request which fetches a response body of large fixed size may not work well with a microservices deployment due to the latency in auto-scaling execution. On the other hand, a microservices deployment may outperform serverless deployment in some scenarios, For example, the GET, POST, and DELETE requests with a simple payload can result in a lower duration and cost when used with microservices as compared to using serverless. In addition, serverless strategy provides immediate scalability and prompt response when handling random spike traffic, but the microservices architecture still is the best cost-effective when facing regular traffic patterns.

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 microservices as well the serverless deployment strategies. A deployment strategy is selected dynamically based on the workload pattern.

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. The authors also thank the anonymous reviewers whose comments helped in improving this paper.

REFERENCES

Amazon (2018). Serverless. `https://aws.amazon.com/serverless/`. [Online; Accessed: 4-Feburary-2020].

AWS (2020a). What is aws x-ray? `https://docs.aws.amazon.com/xray/latest/devguide/aws-xray.html`. [Online; Accessed: 4-Feburary-2020].

AWS (2020b). What is the aws serverless application model (aws sam)? `https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/what-is-sam.html`. [Online; Accessed: 4-Feburary-2020].

Baldini, I., Castro, P. C., Chang, K. S., Cheng, P., Fink, S. J., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R. M., Slominski, A., and Suter, P. (2017). Serverless computing: Current trends and open problems. *CoRR*, abs/1706.03178.

Bhojwani, R. (2018). Design patterns for microservice-to-microservice communication - dzone microservices. `https://dzone.com/articles/design-patterns-for-microservice-communication`.

Casalicchio, E. and Perciballi, V. (2017). 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)*, pages 207–214.

Castro, P., Ishakian, V., Muthusamy, V., and Slominski, A. (2019). The rise of serverless computing. *Commun. ACM*, 62(12):44–54.

Di Francesco, P., Lago, P., and Malavolta, I. (2018). Migrating towards microservice architectures: An industrial survey. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 29–2909.

Eivy, A. (2017). Be wary of the economics of ”serverless” cloud computing. *IEEE Cloud Computing*, 4:6–12.

Gancarz, R. (2017). The economics of serverless computing: A real-world test. `https://techbeacon.com/enterprise-it/economics-serverless-computing-real-world-test`. [Online; Accessed: 23-March-2020].

Handy, A. (2014). Amazon introduces lambda, containers at aws re:invent. `https://sdtimes.com/amazon/amazon-introduces-lambda-containers/`. [Online; Accessed: 4-Feburary-2020].

Jambunathan, B. and Yoganathan, K. (2018). Architecture decision on using microservices or serverless functions with containers. In *2018 International Conference on Current Trends towards Converging Technologies (ICCTCT)*, pages 1–7.

Jindal, A., Podolskiy, V., and Gerndt, M. (2019). Performance modeling for cloud microservice applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE ’19, page 25–32, New York, NY, USA. Association for Computing Machinery.

Jonas, E., Pu, Q., Venkataraman, S., Stoica, I., and Recht, B. (2017). Occupy the cloud: Distributed computing for the 99In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC ’17, page 445–451, New York, NY, USA. Association for Computing Machinery.

Kazanavičius, J. and Mažeika, D. (2019). Migrating legacy software to microservices architecture. In *2019 Open Conference of Electrical, Electronic and Information Sciences (eStream)*, pages 1–5.

Kozhirbayev, Z. and Sinnott, R. O. (2017). A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems*, 68:175 – 182.

Kratzke, N. (2018). A brief history of cloud application architectures. *Applied Sciences*, 8.

Lee, H., Satyam, K., and Fox, G. (2018). Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 442–450.

Lin, W.-T., Krintz, C., Wolski, R., Zhang, M., Cai, X., Li, T., and Xu, W. (2018). Tracking causal order in aws lambda applications. *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 50–60.

Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., and Pallickara, S. (2018). Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 159–169.

Lynn, T., Rosati, P., Lejeune, A., and Emeakaroha, V. (2017). 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)*, pages 162–169.

Mazlami, G., Cito, J., and Leitner, P. (2017). Extraction of microservices from monolithic software architectures. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 524–531.

Mohan, A., Sane, H., Doshi, K., Edupuganti, S., Nayak, N., and Sukhomlinov, V. (2019). Agile cold starts for scalable serverless. In *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'19, page 21, USA. USENIX Association.

Mohanty, S. K., Premsankar, G., and di Francesco, M. (2018). An evaluation of open source serverless computing frameworks. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 115–120.

Novoseltseva, E. (2017). Benefits of microservices architecture implementation. `https://dzone.com/articles/benefits-amp-examples-of-microservices-architectur`. [Online; Accessed: 23-March-2020].

Pinto, D., Dias, J. P., and Sereno Ferreira, H. (2018). Dynamic allocation of serverless functions in iot environments. In *2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 1–8.

Richardson, C. (2015). Introduction to microservices. `https://www.nginx.com/blog/introduction-to-microservices/`. [Online; Accessed: 25-January-2020].

Richardson, C. (2019a). Microservices pattern: Microservice architecture pattern. `https://microservices.io/patterns/microservices.html`. [Online; Accessed: 28-January-2020].

Richardson, C. (2019b). Microservices pattern: Monolithic architecture pattern. `https://microservices.io/patterns/monolithic.html`. [Online; Accessed: 25-January-2020].

Schneider, T. (2016). Achieving cloud scalability with microservices and devops in the connected car domain. In *Software Engineering*.

Thömmes, M. (2017). 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`. [Online; Accessed: 14-Feburary-2020].

Şamdan, E. (2018). Dealing with cold starts in aws lambda. `https://medium.com/thundra/dealing-with-cold-starts-in-aws-lambda-a5e3aa8f532`. [Online; Accessed: 14-Feburary-2020].