TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Caner Gür 165512IASM

A COMPARISON OF MICROSERVICE AND SERVERLESS ARCHITECTURES FOR BACKEND SERVICES

Master's Thesis

Supervisor: Lembit Jürimägi
Lecturer

Tallinn 2018

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Caner Gür

25.12.2018

Abstract

The main object in this thesis is to give a comparison between microservice and serverless architectures for implementations of the REST APIs. Before, making the experimental implementations and testing, an historical overview given about these architectures. In addition to that some information given about the usages of these architectures in current market and the biggest cloud service providers which gives support to these architectures on their platforms.

In experiments section, there are two experimental applications created using C# programming language on .NET platform. One of these implementations is for microservice and the other one is the serverless architecture. Both implementations are using common code base to make a better comparison between these two architectures. These experimental applications deployed to Microsoft Azure cloud environment for testing.

An open source tool Locust is used for creating load tests against two implementations and outputs from this tool represented in graphics to explain and compare the results.

All the results acquired from tests and development are compared from three perspectives as cost, performance and development.

This thesis is written in English and is 56 pages long, including 7 chapters, 32 figures and 8 tables.

List of abbreviations and terms

API	*Application Program Interface*
AWS	*Amazon Web Services*
BaaS	*Backend as a Service*
CDN	*Content Delivery Network*
CPU	*Central Processing Unit*
CRM	*Customer Relationship Management*
DB	*Database*
FaaS	*Function as a Service*
IOT	*Internet of Things*
JSON	*JavaScript Object Notation*
MVC	*Model View Controller*
OTT	*Over the Top*
RAM	*Random Access Memory*
REST	*Representational State Transfer*
RPS	*Requests per Second*
SDK	*Software Development Kit*
SOA	*Service Oriented Architecture*
UI	*User Interface*

Table of contents

List of figures

List of tables

1 Introduction

From beginning of software development history, there are various architectures and methodologies which created within the time until now. All those are created because of the changing requirements of the services and technologies. Nowadays, computers and software are all around us. They are in most of the devices that we use in our daily lives such as mobile phones, watches, cars, air conditioners, even washing machines and much more. Also, with IOT, these devices are connecting to internet and communicate each other, so it makes the systems even more complex.

Developers are constantly working on solving problems with the systems and trying to make development and maintenance easier. If the systems get bigger and complex, development and maintenance also get difficult. To solve that kind of issues, there are some architectural approaches for software development which they will be mentioned about in this study.

Service oriented architecture is one of the most important of these architectures. It has been introduced in 90s and started to be used widely in 2000s. With the rapid technological developments after 90s to today, this architecture is extended with new approaches like Microservices.

Microservices approach tries to divide the system by small logical pieces which have different purposes and can work independently from each other and they contain all the necessary resources like DB, cache etc. and communicate with each other using external messaging methods. Even if, it makes development easier, with the latest developments sometimes it can be hard to maintain this kind of services as well. So, a new architecture called Serverless Architecture is introduced recently.

Serverless architecture is a new approach which still in early stages of its support by cloud providers and common usage in the software development area. This approach gives the flexibility to developers to write the code and deploy it directly without thinking any maintenance and performance issues caused by server limitations. That’s a great idea for the first look, but we’ll take a look in this study to see if it’s really good solution or not.

2 Cloud Computing Architectures

In this study, as it mentioned in the introduction there will be a comparison between Microservice and Serverless architectures. In this section, it'll be mentioned about the history and some specifications of these architectures.

2.1 Microservice Architecture

Microservice architecture is an architecture which extends the Service Oriented Architecture. So, firstly SOA will be briefly explained to give a brief history about evaluation on these architectures.

2.1.1 Service Oriented Architecture

Software architectures has changing over the years to design the software systems in a better, cheaper and more efficient way. To achieve this goal lots of different architectures are developed in time. Therefore, the service-oriented architecture is one of them [1].

Service-oriented architecture is dividing a big system to logical subsystems that have [2]different purposes, as services. These services encapsulate some specific logic behind for specific purpose. These services can be reused by other systems and also can communicate with other services via messaging to work cooperatively [1].

In service-oriented architecture, services are divided according to business activities. Each service needs to have clearly defined responsibilities. In that way model-driven implementation, message-based communication., service composition and discovery methods can be applied to the system. Therefore, more flexible and convenient systems can be developed in less time [1, 3].

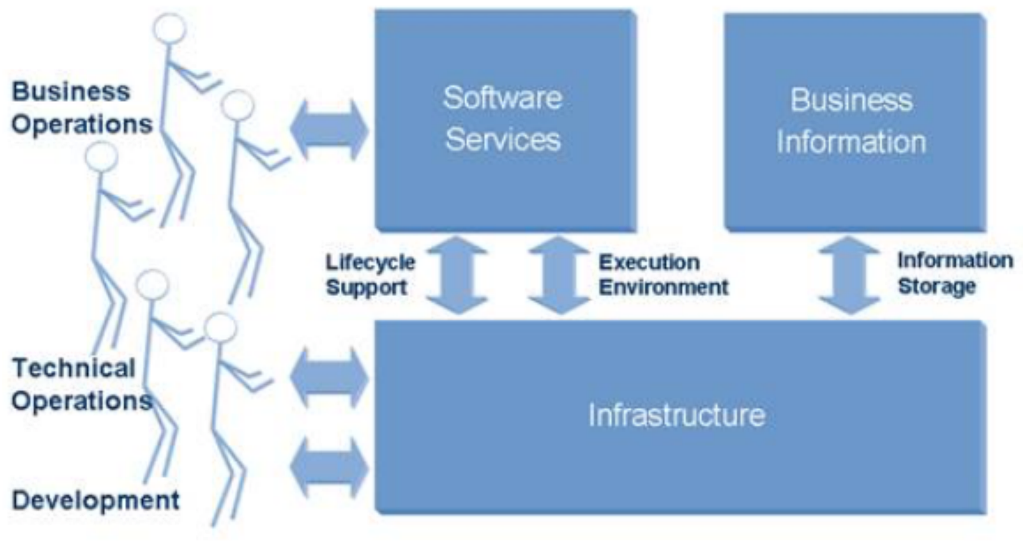


Figure 1 Overview of SOA [4]

Block explanations according to Figure 1 [4].

- Software services are interfaces between the users from business operations of enterprise and computers. They can be web applications or some other user interfaces like control panel of some device.

- Infrastructure is an environment to execute software services. It has an operating system to run the services and also provides networking to communicate other services via messaging. This part of the system managed by technical staff via provided user interfaces.

- Business information can be stored and transferred through infrastructure between services inside the system.

2.1.2 Microservice Architecture

Microservice architecture is a style of designing a system using small independent subsystems which have different business roles, and this is a type of service-oriented architecture. Microservice architecture consists of the microservices as primary building blocks. A single microservice a has single purpose, not dependent to other services in the system and also self-contained [5, 3]. As seen on the Figure 2 microservices are self-contained and independent from each other, but also can communicate among other services and the user interfaces.

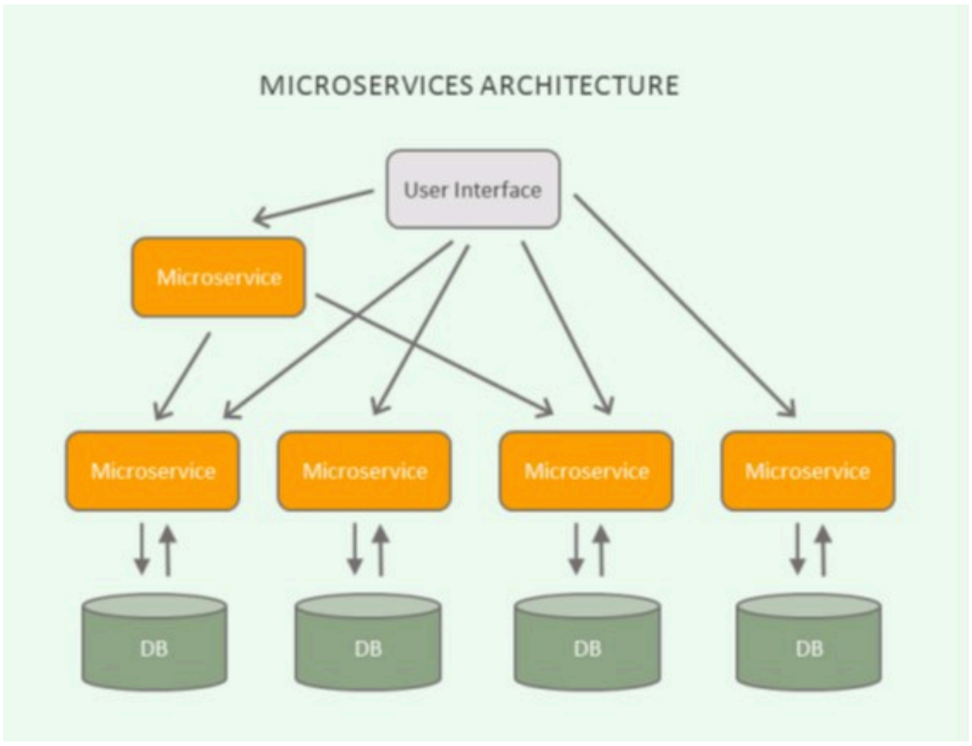


Figure 2 Microservice Architecture Representation [6]

Microservice architecture [5];

- Reduces the complexity development and operation. Only needed microservices could be changed in a system while development and doesn't affect other parts. Operation of the system will be also easier for scaling and easy to manage some disruptions in services.

- Simplifies the system functionally. Each microservice has own functionality and own purpose. So, it's easy to debug and it prevents the services being functionally complex.

- Reduces time to market. It allows faster development and deployment. Especially for smaller coding changes can be applied very fast comparing to a monolithic system. Also, it's easier to deploy services because it requires less resources than a monolithic system and it can be scaled as microservice based when it's needed for lower prices.

- Is more convenient to use DevOps approach with the methods like Continuous Deployment and Integration.

- Gives better user experience. It’s easy to recover services in case of a failure and prevent other services to fail such kind of scenarios. It also helps to improve the uptime rate for system.

2.1.3 Backend as a Service

Backend as a Service is a development approach to use cloud services linked with mobile and web applications. This approach can be used in applications such as social media platforms, OTT media platforms, online games etc. via some software development kits or application programming interfaces. It’s described as “turn-on infrastructure” by technology analysts and it’s a cloud computing category that used by companies who tries to make setup easier for developers and connect all web, mobile and tablet applications with backend cloud resources easily [7]. Backend as a service architecture can be represented like in Figure 3 with a cloud service used by different kind of client applications.

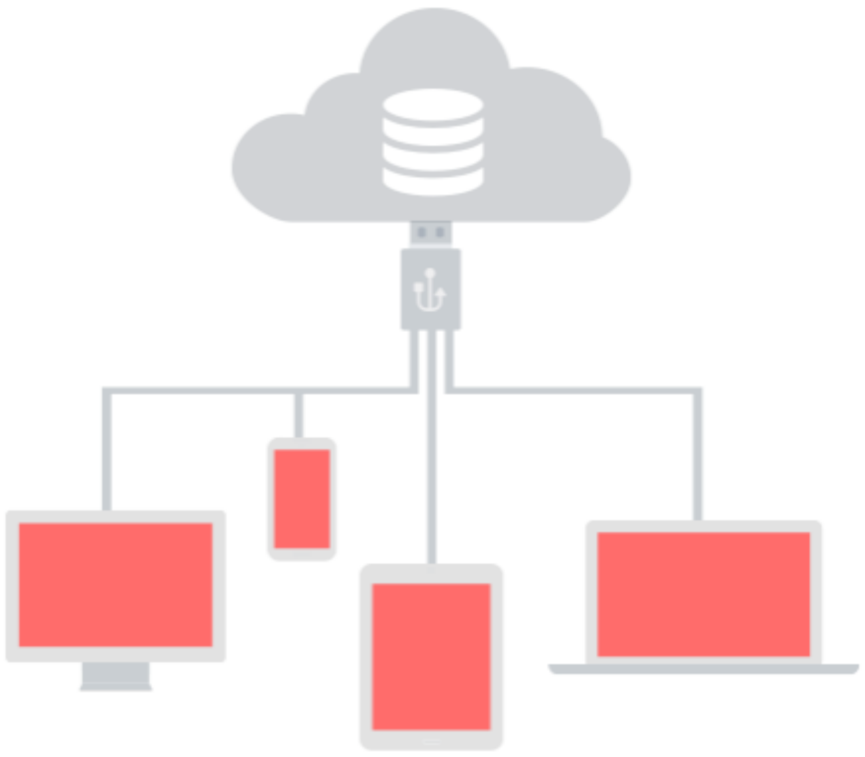


Figure 3 Logical Representation of BaaS [2]

Most of the mobile and web applications uses similar features like instant messaging, social media integration etc. All these services need an application programming interface to be connected to application and more services complicates the development process for developers. Therefore, the BaaS provides a bridge between these services and frontend applications as an SDK or API. API-based BaaS approach provides some third-party services as backend functions which can be reusable in different applications. These functions are available for all frontend applications regardless their platforms like mobile, web or tablet. BaaS has several advantages comparing to regular frontend implementations like an additional layer of security for information exchange and also, quick application development using existing backend functions and features. Here are a couple of benefits that BaaS provides [7, 8].

- It allows more accessibility using the same base backend for all platforms and it makes easy to develop cross platform applications represented in Figure 3. Also, it provides better communication between frontend apps and cloud applications and resources.

- Provides ability to create different platforms using same base model. If we consider the base BaaS model as a class, users of this class can create their custom objects using this base model. Using this unified backend gives better and stronger user base.

- It prevents to make unnecessary development for each application for different platforms. BaaS can provide most of the processing logic for applications via provided SDKs and APIs. Therefore, there will be no need to make them on client device and most of the functions can be executed by cloud applications.

- It reduces time-to-market thanks to ease of frontend development by keeping the application logic in the backend services.

2.2 Serverless Architecture

Serverless architecture is a new way to design software applications without handling any server-side operation, maintenance, scaling and all kind of server related issues. It provides a clear separation between coding and hosting environment. All functions need to be invoked by a trigger. There are plenty of trigger options such as http request, time schedule, queue messages etc. depends on the cloud service provider which the functions deployed on [9].

Software architecture evolves in a way that, there are functions in an atomic form that have single purpose like add a new record to database, send a message to queue, process an image etc. All the resources for these processes are managed by cloud service provider and developer doesn't need to think about provisioning or scaling the resources. For each function call, needed resources are allocated from cloud servers and freed after function completion. So, it prevents to use unnecessary resources [9, 10].

2.2.1 Function as a Service

Function as a Service as known as FaaS is the way to implement serverless application as services. The key point of FaaS is to develop the code and leave the rest to cloud service providers as a feature of serverless architecture. This approach gives a more granular structure than Microservices.

As it's mentioned earlier, microservices have specific purposes to make some business logic behind the scene, but the functions have only one logic behind that they can achieve. So, evolution from monolithic applications to functions can be visualized like in the Figure 4.

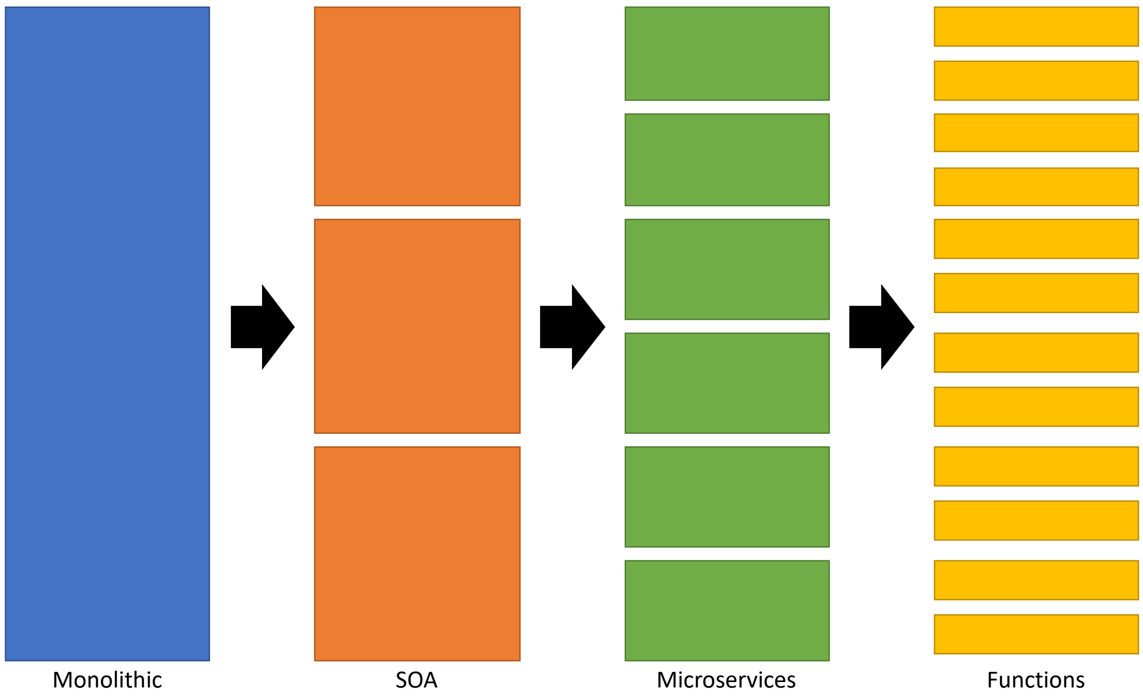


Figure 4 Comparison of Monolithic, SOA, Microservice and Serverless Architectures

Here are some features of FaaS [10];

- FaaS gives an opportunity to code without any provisioning and managing the servers and run the functions only they triggered and freeing the resources after process completion. So, there is no need to keep resources running all the time.

- There is a wide range of support for different programming languages depending on the cloud service providers. So, it gives flexibility to developers to use the programming language that they already familiar with.

- Deployment of the code is also different than the traditional architectures. So, it's only needed to push the codes to cloud service provider and they'll handle the rest of the deployment and new version of code will be running for next executions.

- There is no need to think about the performance for high load scenarios. Namely, all the auto scaling is performed by cloud service provider and more resources is created for high loads. So, lots of parallel executions can be supported without any issues.

- Functions can be triggered by different type of triggers such as http requests, file uploads, queue messages, time schedules and many more of them. So, this gives a flexibility and ease of development for developers to use these kinds of triggers without using any custom trigger handling implementations.

Also using FaaS approach with the features that mentioned above has some benefits such as [11];

- It's easy to maintain and operate. The serverless architecture provides us a clear separation between infrastructure and applications. Auto scaling feature helps to reduce computing cost as well as the maintenance and operation overheads. Comparing to traditional services, it requires almost no effort to make a deployment of the code and it doesn't require any containerization or continuous delivery configurations. It means a completely serverless application doesn't require any system administration.

- It gives an opportunity to innovate faster. Software architects and developers can focus only to innovate and develop new features regardless of thinking the maintenance, deployment, scaling and performance issues. So, it gives more time to think about the actual product and functionalities.

- It reduces the costs of operation and maintenance. This is one of the most important benefit of the FaaS approach. The cost for the serverless solution is managed servers, databases and application logic. So, cost of usage will be based on running functions in solution. It means that if usage increases then cost will be increased on the contrary application owners need to pay for dedicated servers regardless of any usage.

It can be seen obviously that cost is going high depends on usage, unlike traditional services running on dedicated servers in Figure 5.

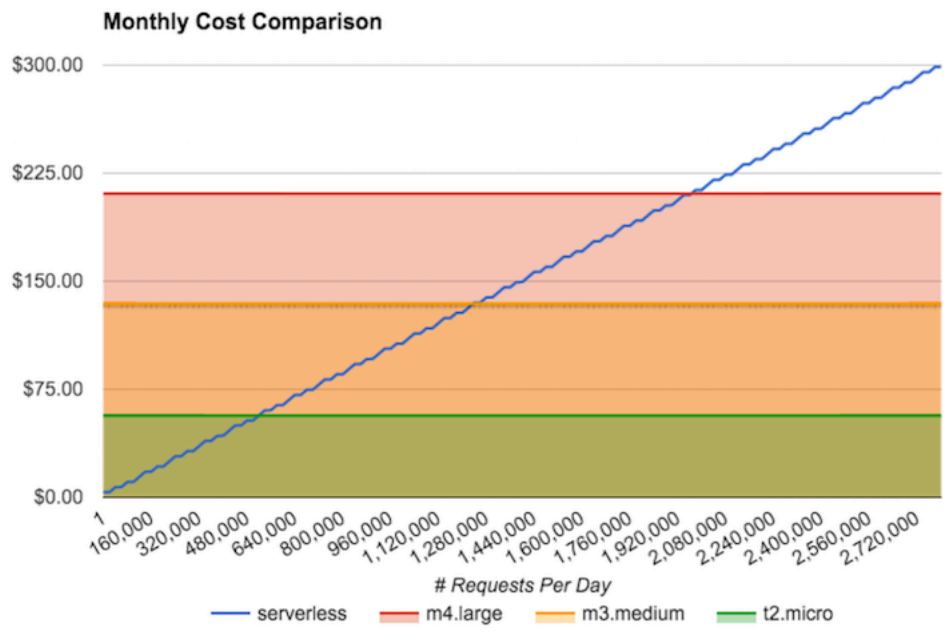


Figure 5 Sample cost comparison of dedicated servers and serverless from AWS [12]

3 Infrastructures of the Common Services in the Market

There are various types of services in our daily lives which using different architectures and resources. In this study, it will be tried to give some overview about some actual services that we use commonly such as social media, video on demand and e-commerce services.

3.1 Social Media Services

Social media services are commonly used services by many people, and it contains lots of user data, some contents like photos posts, videos etc. Therefore, to handle all these content and requests coming from lots of users a solid architecture need to be used. Simply we can describe an architecture with microservices for social media like in the Figure 6.

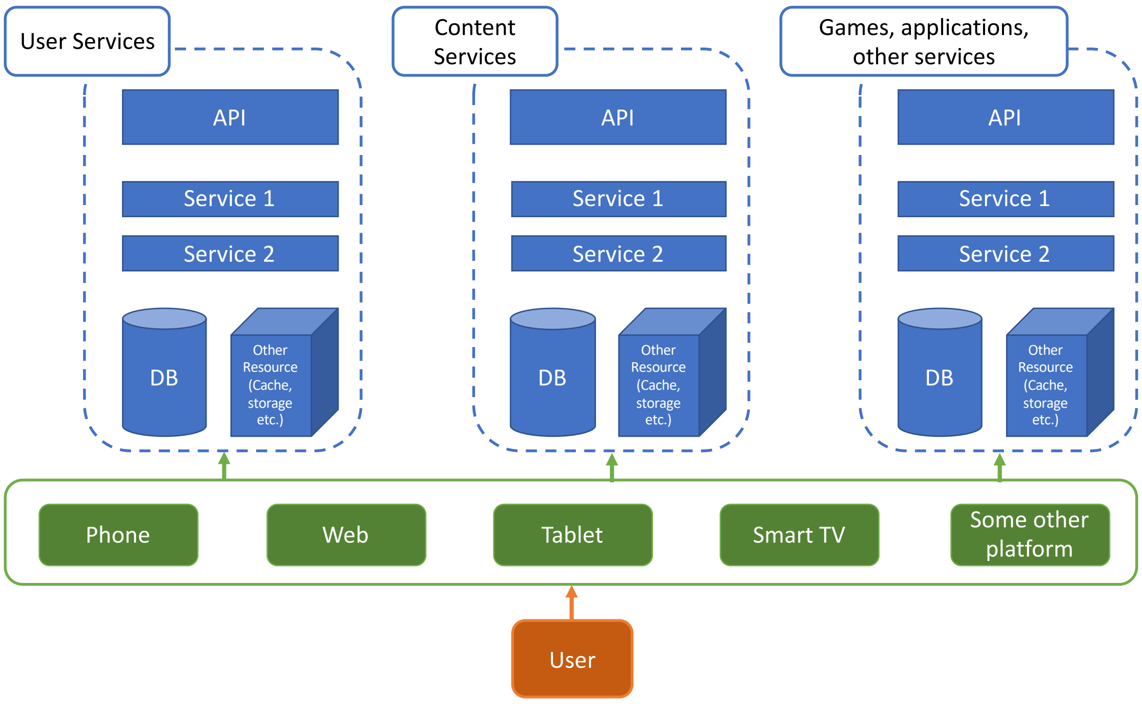

User Services
API
Service 1
Service 2
DB
Other Resource (Cache, storage etc.)
Content Services
API
Service 1
Service 2
DB
Other Resource (Cache, storage etc.)
Games, applications, other services
API
Service 1
Service 2
DB
Other Resource (Cache, storage etc.)
Phone
Web
Tablet
Smart TV
Some other platform
User

Figure 6 Social Media Service Sample Microservice Infrastructure

According to Figure 6;

- User management service does the registration, login, keeping user data like profile information.

- Content management service manages the posts, pictures, videos etc. and keeping them in storage or some file databases. It can be the biggest microservice among others, it'll keep lots of data and take lots of requests. So, it can be divided some small functions for simple duties like adding posts, removing posts, like, dislike kind of actions and many more. This can be a good sample usage of serverless architecture for social media.

- CRM service can be also a microservice manage customer services.

3.2 Video on Demand Services

Video on demand services are the platforms which users can subscribe to some plans to watch shows, movies etc. These kinds of services can give some content set based on subscription of users and also can give one-time purchases to watch only one content like a movie. In these services, there are some subservices like user, subscription, content and streaming services and also, they like described in Figure 7.

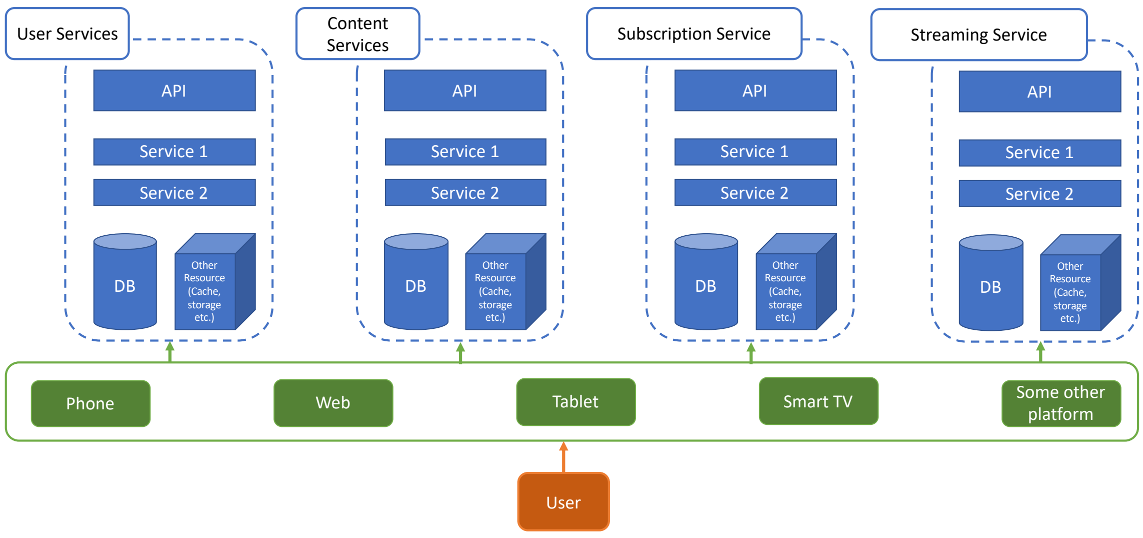


Figure 7 Video on Demand Service Sample Microservice Infrastructure

According to Figure 7;

- User service provides login, registration, user profile etc. features just like the social media services. In user service there can be some features like keeping watch history, adding favorite shows and more. These features can be implemented as functions for these actions like add to favorite, update watch history and so on.

- Subscription service provides the ability for user to subscribe different plans for different needs and also gives entitlement to user to get content according to subscribed plans. In this case this entitlement requests will be very high because each watch attempt will create an entitlement request so these endpoints also can be changed as functions and they will be not a problem to handle those even for the peak hours.

- Content service provides the metadata for the content like movies, tv shows and all kind of video on demand content. This part also can be divided to smaller pieces to get content with small queries such as get movie, get show so on.

- Streaming service is a service that provides the video or music streaming according to metadata acquired from content service. It contains storage and CDN services inside.

3.3 E-Commerce Services

E-Commerce services can be various, selling services, goods, some digital contents and so on. For these kinds of services, we also need similar structure with other services like in Figure 8.

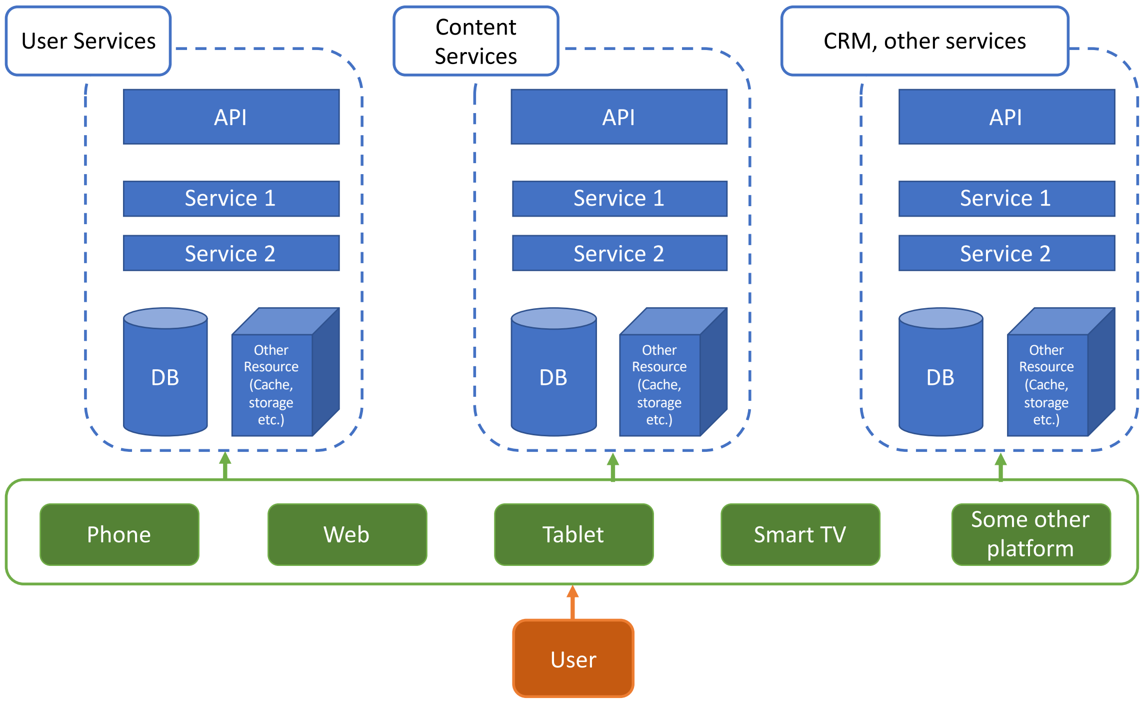


Figure 8 E-Commerce Service Sample Microservice Infrastructure

According to Figure 8;

- User service provides register, login or profile management features for users.
- Content service provides the metadata of the products on sale. This part can be divided to functions to get benefits of auto-scaling feature since it's the part of the system that will create most of the load.
- CRM services gives ability to service provider to manage the customers and give customer service.

4 Cloud Service Providers

There are plenty of cloud service providers which they can support serverless architecture. In this study, it'll be mentioned the first three biggest cloud service providers;

- Amazon Web Services
- Google Cloud Platform
- Microsoft Azure

All these providers have similar services, application and resources that they provide to users. Users can deploy and maintain application, use virtual machines, store data in storage spaces and databases and they can use many more features. In this study it'll be used resources for serverless and microservices for comparison.

Serverless solutions for these providers are very similar about various aspects. All platforms have equivalent triggering options such as database triggers, schedulers, queue triggers, http triggers and so on. Also, they're similar in aspect of scaling. Therefore, the using programming languages and familiarity for this kind platforms make an effect on choosing the provider.

4.1 Amazon Web Services

Amazon Web Services as known as AWS is the cloud service platform that created by Amazon in 2006. AWS is started to offer infrastructure services for businesses in a form of web services as known as cloud platforms. Cloud platforms gives ability to businesses to use servers and resources instantly without any procurement of servers or any other infrastructure elements. Now AWS offers scalable, reliable and low-cost cloud infrastructure solutions in 190 countries around the world with the datacenters in U.S., Europe, Brazil, Singapore, Japan, and Australia. [13]

4.1.1 Microservice Solution

Microservice solution which can be used in this study that provided by Amazon is the AWS Elastic Beanstalk. It's a web service platform which can be used as a REST API or any other web application for both frontend and backend. These web applications use some resources from other type infrastructure elements basically virtual machines and storage. Virtual machine solution is EC2 and storage is S3 Bucket in AWS. [14]

- Supported languages are Go, Java, .NET, Node.js, PHP, Python, and Ruby [14].
- Pricing is done according to pricing of EC2 instances and the S3 Buckets which contains runtime of the EC2 instance and storage used in S3 Bucket [15].

4.1.2 Serverless Solution

The solution for serverless architecture of Amazon is AWS Lambda platform [16].

- It supports Java, Node.js, C#, and Python code currently and many other languages will be supported in the future.
- It's billed depending on the compute time to run the code for per 100 milliseconds.

4.2 Google Cloud Platform

Google Cloud Platform is the platform which released by Google in 2008 to provide cloud services to customers. It provides various cloud-based services around the world with 18 datacenters. [17]

4.2.1 Microservice Solution

Microservice solution provided by Google is Google App Engine. It can be used to deploy frontend web applications or backend APIs [18].

- Supported languages for App Engine are Java, PHP, Node.js, Python, C#, .Net, Ruby, Go [18].
- App Engine pricing is done based on the resource consumption such as run time, storage, network traffic according to selected configuration plans [19].

4.2.2 Serverless Solution

Google supports serverless architecture by Google Cloud Functions [20].

- It supports only JavaScript (Node.js) and Python as programming languages. So, this a disadvantage of Google platform against AWS.

- It’s also billed in execution time of function per 100 milliseconds.

Google also have another cloud service platform called Firebase and it has a serverless solution named Cloud Functions for Firebase. It supports JavaScript and TypeScript for as programming languages for function implementation [21].

4.3 Microsoft Azure

Azure is the cloud services platform which created and released in 2010 by Microsoft as Windows Azure and it’s renamed as Microsoft Azure in 2014. It provides wide range of cloud services worldwide as the other service providers that’s mentioned before. There are 54 datacenters around the world which provides most of the services that Azure platform supports. [22]

4.3.1 Microservice Solution

Microservice web application infrastructure which provided by Microsoft Azure is the Azure Web Apps [23].

- It supports .NET, Java, Node.js, PHP, and Python on Windows or .NET Core, Node.js, PHP or Ruby on Linux as programming languages [23].

- Billing is calculated according to run time of the application based on hourly price of the selected tier [24].

4.3.2 Serverless Solution

Azure Functions is the equivalent solution of Microsoft like the other providers.

- Programming language options are pretty much better than the other providers since it supports C#, JavaScript, F#, Java as main languages. Also, there is support for Python, TypeScript, PHP, Batch, Bash and PowerShell languages for experimental use [25].

- It’s billed depending on the compute time to run the code for per 100 milliseconds.

4.4 Comparison

For the solutions and specifications for the microservice and the serverless platforms which provided by Amazon, Google and Microsoft can be compared like in the Table 1 and Table 2.

Table 1 Microservice platforms comparison by service providers

Specifications	**AWS Elastic Beanstalk**	**Google App Engine**	**Azure Web App Service**
Pricing calculation	Runtime of EC2 instance per hour Storage consumption for S3 Bucket per GB	Sum of run time, network traffic, storage according to selected configuration	Runtime of instance per hour
Supported languages	Go, Java, .NET, Node.js, PHP, Python, Ruby	Java, PHP, Node.js, Python, C#, .Net, Ruby, Go	.NET, .NET Core, Java, Ruby, Node.js, PHP, Python
Visual Studio Integration	Yes (with extension)	Yes (with extension)	Yes

Table 2 Serverless platforms comparison by service providers

Specifications	**AWS Lambda Functions**	**Google Cloud Functions**	**Azure Functions Apps**
Pricing calculation	Execution time per 100 ms Resource consumption per 128 MB	Sum of number of invocations, compute time, network traffic according to selected configuration	Execution time per 100 ms Resource consumption per 128 MB
Supported languages	Java, Node.js, C#, Python	JavaScript (Node.js), Python	C#, JavaScript, F#, Java (Python, TypeScript, PHP, Batch, Bash, PowerShell for experimental use)
Visual Studio Integration	Yes (with extension)	No	Yes (native)

5 Experiment Setup

For a complete project such as social media, shopping, game applications it's needed to use various infrastructure resources like database, messaging queues, storage etc. All resources which is used it the system may have bottlenecks on high loads. For example, application server resources like CPU, RAM supports to make 1000 requests per second for implemented application, but database may support only 100 simultaneous queries per second, so it can be problem to check actual effect of the used architecture that serverless or microservice.

Therefore, a new sample benchmark project solution created independent from other infrastructural resources like database or storage. Thus, it's possible to compare pure architecture effect on this sample application.

In this study, it has been chosen to create sample applications for microservice and serverless approaches using C# programming language. The reason for this choice is familiarity for me to use C# language.

Microsoft Azure is selected platform to make experiments and benchmarks for microservices and serverless services because it supports the C# language and .NET applications and it gives a better development experience with Visual Studio and Azure integrations.

For microservice deployment, Azure Web Apps is used and all the deployment processes for both implementations are done by the Visual Studio and Azure publishing integration tools.

Both projects created in one .Net solution in structure given in Figure 9 and Microsoft Visual Studio for Mac Community Edition used as IDE for development.

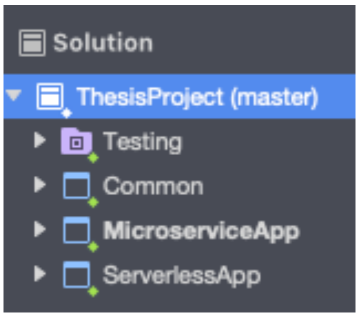


Figure 9 Example Benchmark Solution Structure

A Common .NET Standard Library project is created to be referenced as a common service interface to use the same code base for both services to make a better comparison by reducing coding difference. There is one *ICommonService* interface and *CommonService* class as implementation along with the *Success*, *Error* and *SampleModel* response model classes in the structure given in Figure 10.

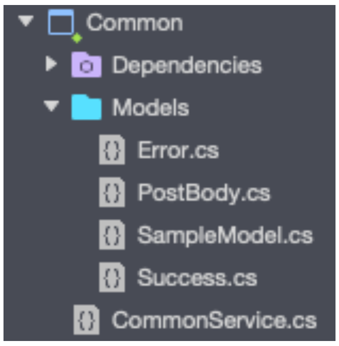
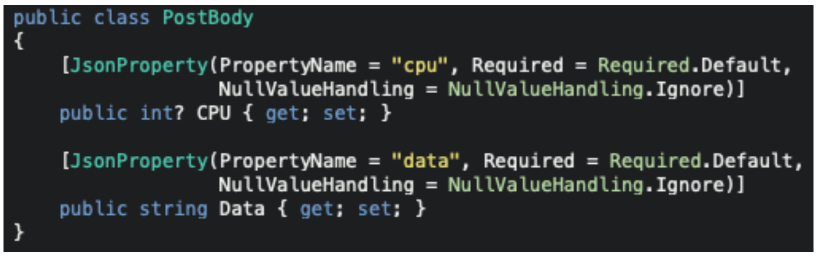


Figure 10 Common Service Project Structure

There are two endpoints defined for both serverless and microservice projects as;

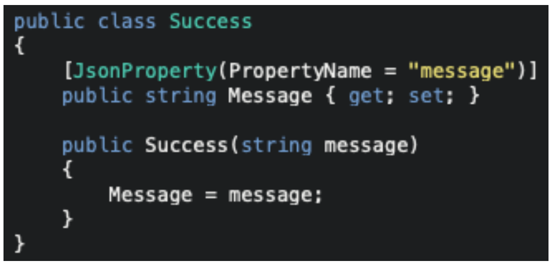
- “POST /api/test/” : Gets the body as in Figure 11 from the request as a JSON object with “data” and “cpu” values and returns *Success* response with a message contains the value of “data” given in Figure 12 and Figure 13.



```
public class PostBody
{
    [JsonProperty(PropertyName = "cpu", Required = Required.Default,
                  NullValueHandling = NullValueHandling.Ignore)]
    public int? CPU { get; set; }

    [JsonProperty(PropertyName = "data", Required = Required.Default,
                  NullValueHandling = NullValueHandling.Ignore)]
    public string Data { get; set; }
}
```

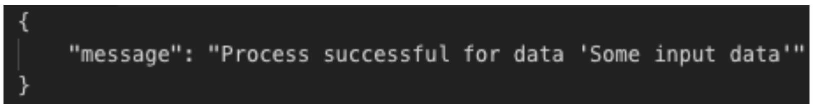
Figure 11 Post Request Body



```
public class Success
{
    [JsonProperty(PropertyName = "message")]
    public string Message { get; set; }

    public Success(string message)
    {
        Message = message;
    }
}
```

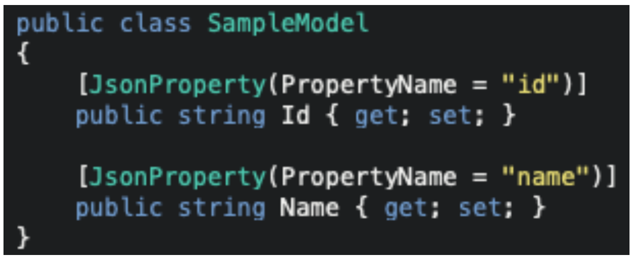
Figure 12 Success Class Definition



```
{
    "message": "Process successful for data 'Some input data'"
}
```

Figure 13 Sample response for "POST /api/test"

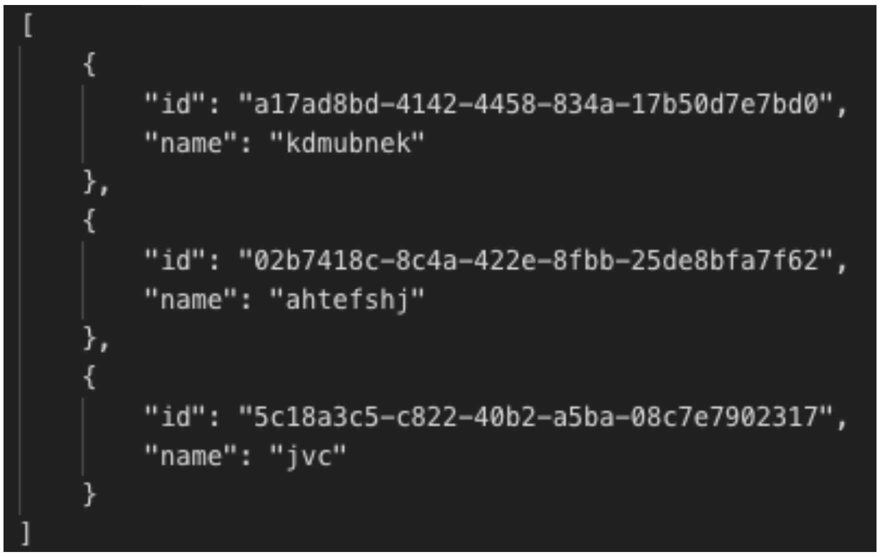
- “GET /api/test/” : Gets integer values “input” and “cpu” as query parameters and returns number of items same as input value as a list of *SampleModel* given in Figure 14 and Figure 15.



```
public class SampleModel
{
    [JsonProperty(PropertyName = "id")]
    public string Id { get; set; }

    [JsonProperty(PropertyName = "name")]
    public string Name { get; set; }
}
```

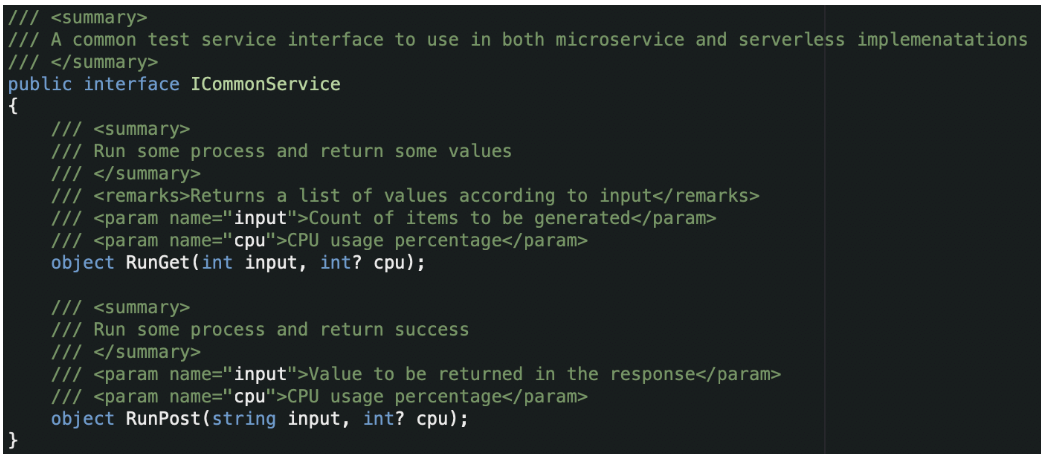
Figure 14 SampleModel Class Definition



```
[
    {
        "id": "a17ad8bd-4142-4458-834a-17b50d7e7bd0",
        "name": "kdmubnek"
    },
    {
        "id": "02b7418c-8c4a-422e-8fbb-25de8bfa7f62",
        "name": "ahtefshj"
    },
    {
        "id": "5c18a3c5-c822-40b2-a5ba-08c7e7902317",
        "name": "jvc"
    }
]
```

Figure 15 Sample response for "GET /api/test"

There are two methods defined in interface for *CommonService* to be used for GET and POST endpoints like in Figure 16.



```
/// <summary>
/// A common test service interface to use in both microservice and serverless implemenatations
/// </summary>
public interface ICommonService
{
    /// <summary>
    /// Run some process and return some values
    /// </summary>
    /// <remarks>Returns a list of values according to input</remarks>
    /// <param name="input">Count of items to be generated</param>
    /// <param name="cpu">CPU usage percentage</param>
    object RunGet(int input, int? cpu);

    /// <summary>
    /// Run some process and return success
    /// </summary>
    /// <param name="input">Value to be returned in the response</param>
    /// <param name="cpu">CPU usage percentage</param>
    object RunPost(string input, int? cpu);
}
```

Figure 16 ICommonService Interface Definition

- *RunGet* method runs a process to consume some CPU resources according to given CPU usage percentage for a random duration between 500 and 1500 milliseconds then generates a list of *SampleModel* according to given number in the request and returns the list.

- *RunPost* method runs a process to consume some CPU resources according to given CPU usage percentage for a random duration between 500 and 1500 milliseconds and returns *Success* result with a message contains input value.

5.1 Backend as a Service (BaaS) Implementation

BaaS implementation *MicroserviceApp* is created as ASP .NET Core Web API project in the structure given in Figure 17.

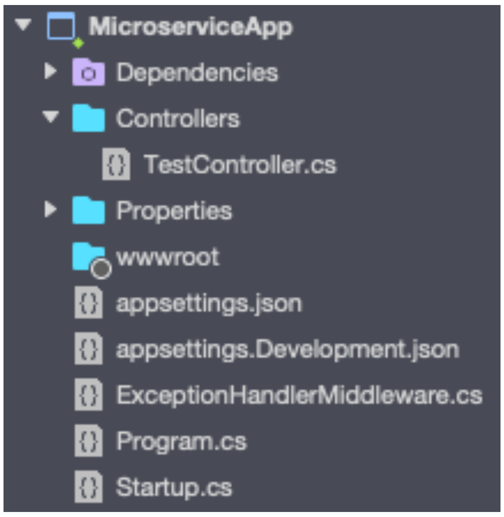
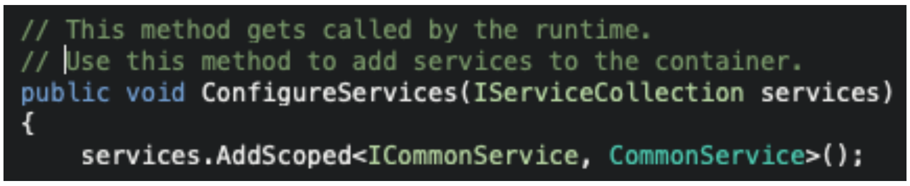


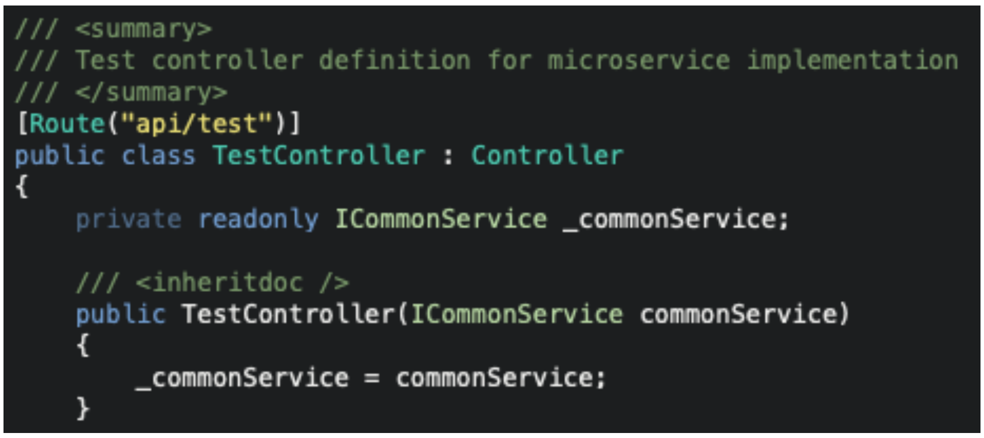
Figure 17 BaaS Implementation Project Structure

CommonService is initialized in *Startup* like in Figure 18 to be injected as a dependency for *TestController* in runtime like in Figure 19 as a common approach for using service classes [26]. It has been chosen to use dependency as *Scoped* to create an instance of *CommonService* for each request as well as in *ServerlessApp*.



```
// This method gets called by the runtime.
// Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<ICommonService, CommonService>();
```

Figure 18 Initialization of CommonService in Startup



```
/// <summary>
/// Test controller definition for microservice implementation
/// </summary>
[Route("api/test")]
public class TestController : Controller
{
    private readonly ICommonService _commonService;

    /// <inheritdoc />
    public TestController(ICommonService commonService)
    {
        _commonService = commonService;
    }
```

Figure 19 Dependency Injection for TestController

5.2 Function as a Service (FaaS) Implementation

FaaS implementation *ServerlessApp* is created as Azure Functions project in the structure given in Figure 20.

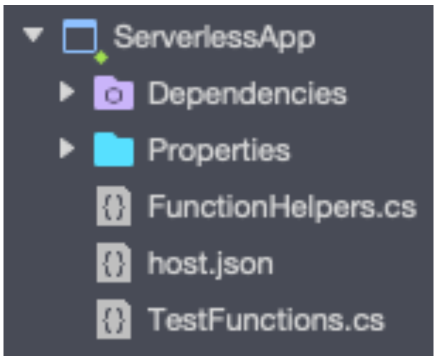
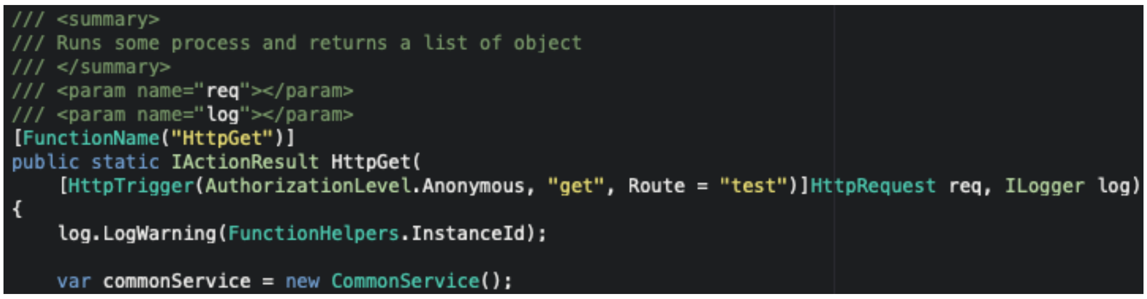


Figure 20 FaaS Implementation Project Structure

Since functions are static methods, all the resources like used class instances, database connections etc. must be initialized for each function call. For this case only *CommonService* class need to be instantiated manually in function like in Figure 21.



```
/// <summary>
/// Runs some process and returns a list of object
/// </summary>
/// <param name="req"></param>
/// <param name="log"></param>
[FunctionName("HttpGet")]
public static IActionResult HttpGet(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route = "test")]HttpRequest req, ILogger log)
{
    log.LogWarning(FunctionHelpers.InstanceId);

    var commonService = new CommonService();
```

Figure 21 Instantiation of CommonService in Function

To observe scaling for Azure Functions extra logging is added to see instance ids like in Figure 21. Thus, it'll be possible to see more different instances created when load goes high.

It's not possible to get query parameters or body parsed directly by function infrastructure. So, some custom function helper extensions created to get query parameters and parse request body.

5.3 Testing

Http load testing should be done for comparison of the serverless and microservice implementations and for observation of scaling for serverless functions. There are various options to make load testing for REST APIs but many of them are commercial. So, an open source testing tool Locust [27] decided to be used for load testing in this study. Because it's free and easy to customize by changing the source codes since it's an open source tool.

Locust is a load testing tool created using Python programming language and it uses Python scripts to test API endpoints. It also has a web user interface while it's running to check the logs and statistics.

There is new folder is created for testing in solution. It contains Python script for endpoint testing and commands to run testing tool for serverless and microservices in structure in Figure 22.

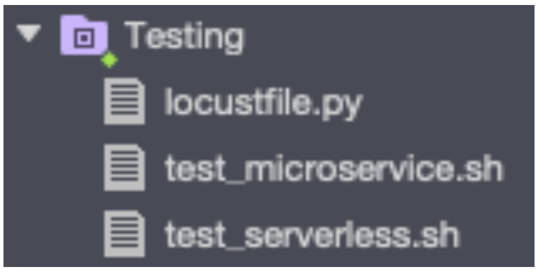
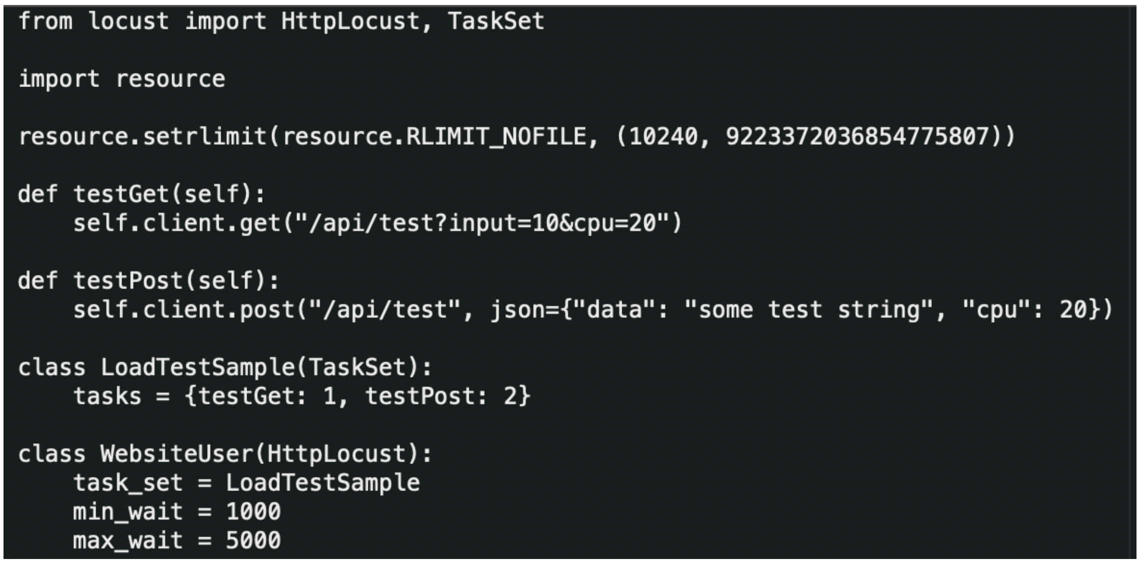


Figure 22 Testing Folder Structure

It’s possible to create lots of various test scenarios using the locustfile.py. For this study, a scenario is created for each endpoint like in Figure 23. Both “GET” and “POST” scenarios are configured to consume 20% of CPU resources while running the process using the “cpu” parameter as input for the endpoints.



```
from locust import HttpLocust, TaskSet

import resource

resource.setrlimit(resource.RLIMIT_NOFILE, (10240, 9223372036854775807))

def testGet(self):
    self.client.get("/api/test?input=10&cpu=20")

def testPost(self):
    self.client.post("/api/test", json={"data": "some test string", "cpu": 20})

class LoadTestSample(TaskSet):
    tasks = {testGet: 1, testPost: 2}

class WebsiteUser(HttpLocust):
    task_set = LoadTestSample
    min_wait = 1000
    max_wait = 5000
```

Figure 23 Locust Test Scenarios

Locust testing tool can be run simply using the command “*locust --host=<host_url> --port <port_number>*”. After running the command, test run can be started using the web interface on “http://localhost:<port_number>” like in Figure 24.

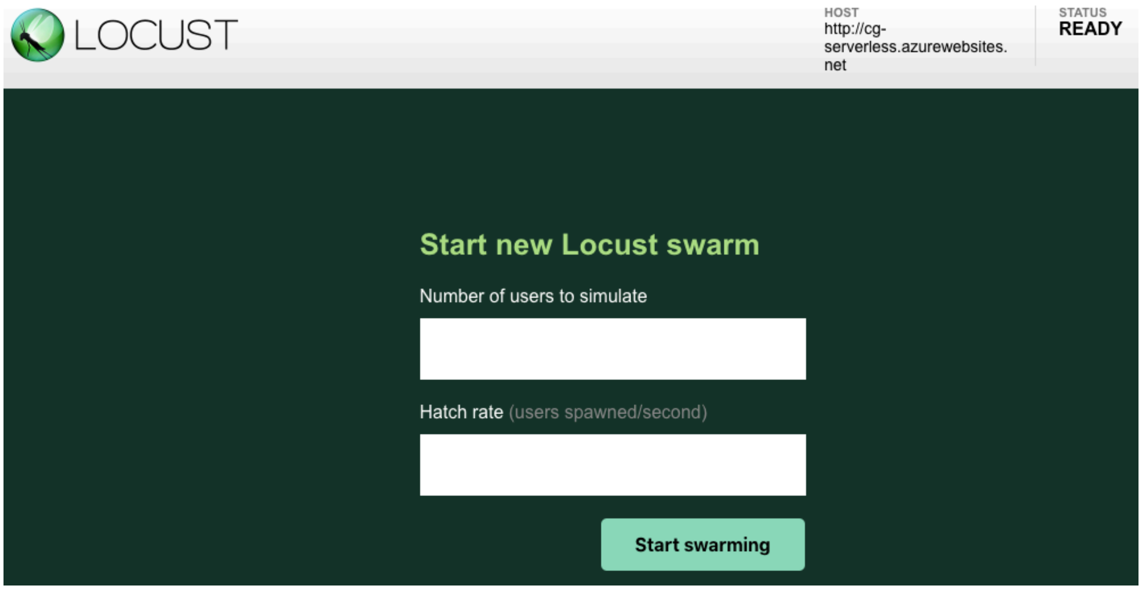

LOCUST
HOST
http://cg-serverless.azurewebsites.net
STATUS
READY
Start new Locust swarm
Number of users to simulate
Hatch rate (users spawned/second)
Start swarming

Figure 24 Locust Load Test Web UI Main Page

The value number of users to simulate is for defining maximum number of users to be simulated and hatch rate is to add number of users per second until it reaches to maximum number of users.

Locust runs a load test using the request definitions in the locust files and the parameters given from web interface. But the hatch rate doesn't support different periods than seconds so it's hard to observe performance. So, it's created custom testing scripts using bash script language to run tests directly from command line.

There are two scripts created to run custom test scenario as "test_microservice.sh" and "test_serverless.sh". Both scripts get a parameter as "start" or "stop".

- *"start*" : Starts the testing process on deployed microservice starting with 50 users and increases the number of users per minute by 50 until it reaches to 400 users. Then it runs extra one more minute for 400 users and stops running process. Also, it opens the default browser to check statistics immediately after starting the process on "localhost:8073" for microservice and "localhost:8072" for serverless.

- "*stop*" : Stops all the running testing processes

Test results can be seen from web user interface as graphics like in Figure 25.

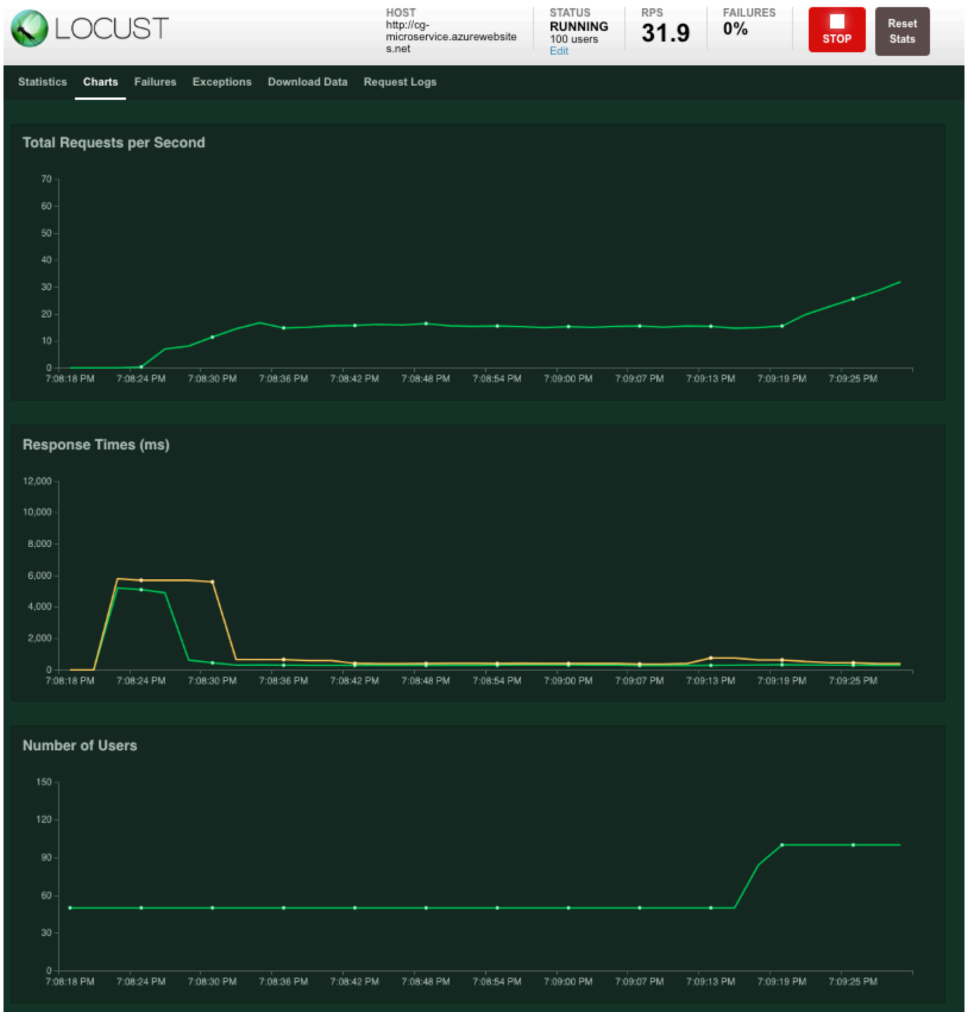
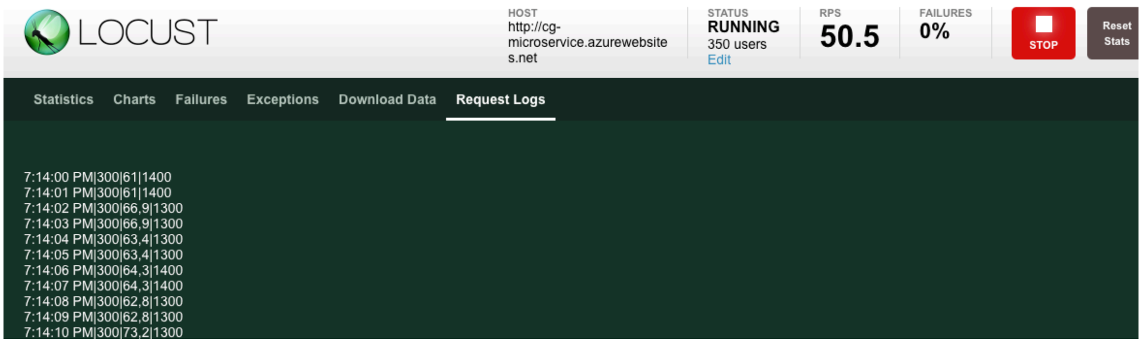


Figure 25 Locust Graphical Interface

Also, with some small changes in testing tool, a new tab “Request Logs” added in scope of this study to get the status for each one second to create combined graphs like in the Figure 26. It generates a “|” separated list and it can be divided into columns using Excel and any graphics can be created using the generated data.



```
7:14:00 PM|300|61|1400
7:14:01 PM|300|61|1400
7:14:02 PM|300|66,9|1300
7:14:03 PM|300|66,9|1300
7:14:04 PM|300|63,4|1300
7:14:05 PM|300|63,4|1300
7:14:06 PM|300|64,3|1400
7:14:07 PM|300|64,3|1400
7:14:08 PM|300|62,8|1300
7:14:09 PM|300|62,8|1300
7:14:10 PM|300|73,2|1300
```

Figure 26 Locust Request Logs Tab

6 Benchmarks for BaaS and FaaS

Benchmarks for BaaS and FaaS will be evaluated in perspectives as performance, cost and development according to experimental application which is developed for this study.

6.1 Performance

From performance perspective, three different results will be compared with BaaS and FaaS. These are requests per second, response times and the number of users.

Same load testing scenario is run on the both platforms. Number of users started from 50 and increased by 50 per minute until it reaches to 400 users and load continued on 400 one more minute.

6.1.1 BaaS Test Results

As seen on the Figure 27, response time starts to increase after second 180 with 200 users. Then there is a dramatic increase on the response times when the user count continues to increase. On the other hand, request per second stops increasing after second 180 with 200 users. So, it's obvious that if the load is increased on the one instance of a service it affects response time and performance and it has a limit to serve without problems. In this case the limit is approximately 60 requests per second.

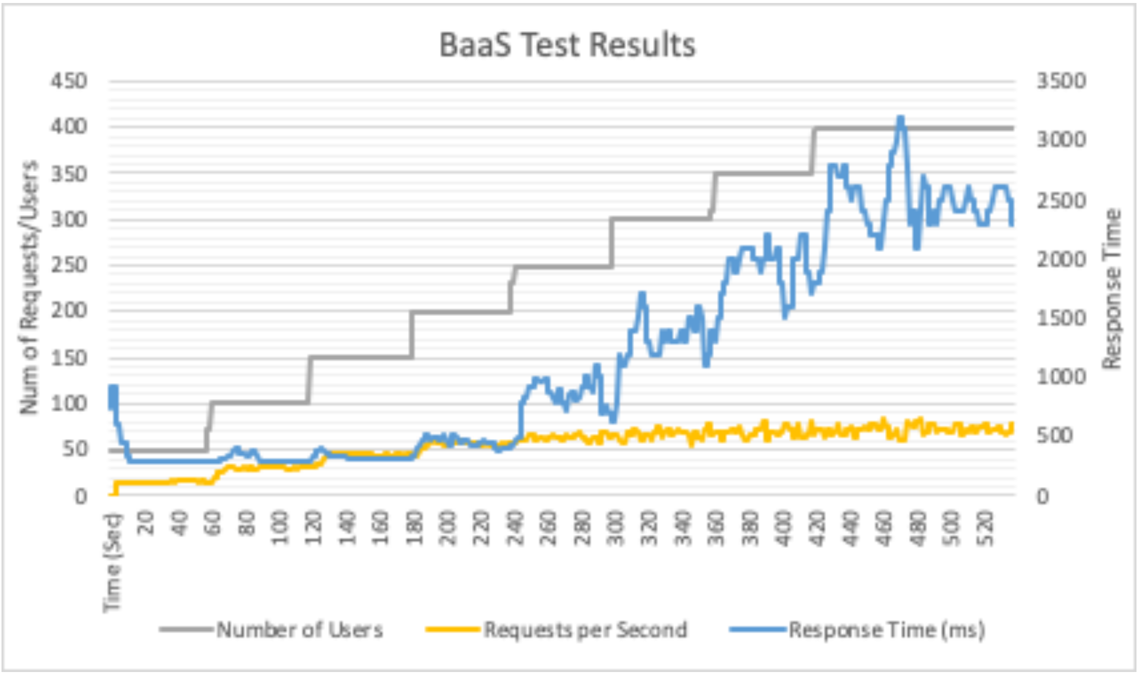


Figure 27 BaaS Test Results

6.1.2 FaaS Test Results

As seen in the Figure 28, request per second is increased with user count increase. Response times are not changed significantly except some peaks on increases. It can be seen after 240 there are new instances created and response time is back to normal again. So, scaling is worked nicely to handle more requests and there are no issues with response times and it can support up to 120 requests per second with 17 function instances.

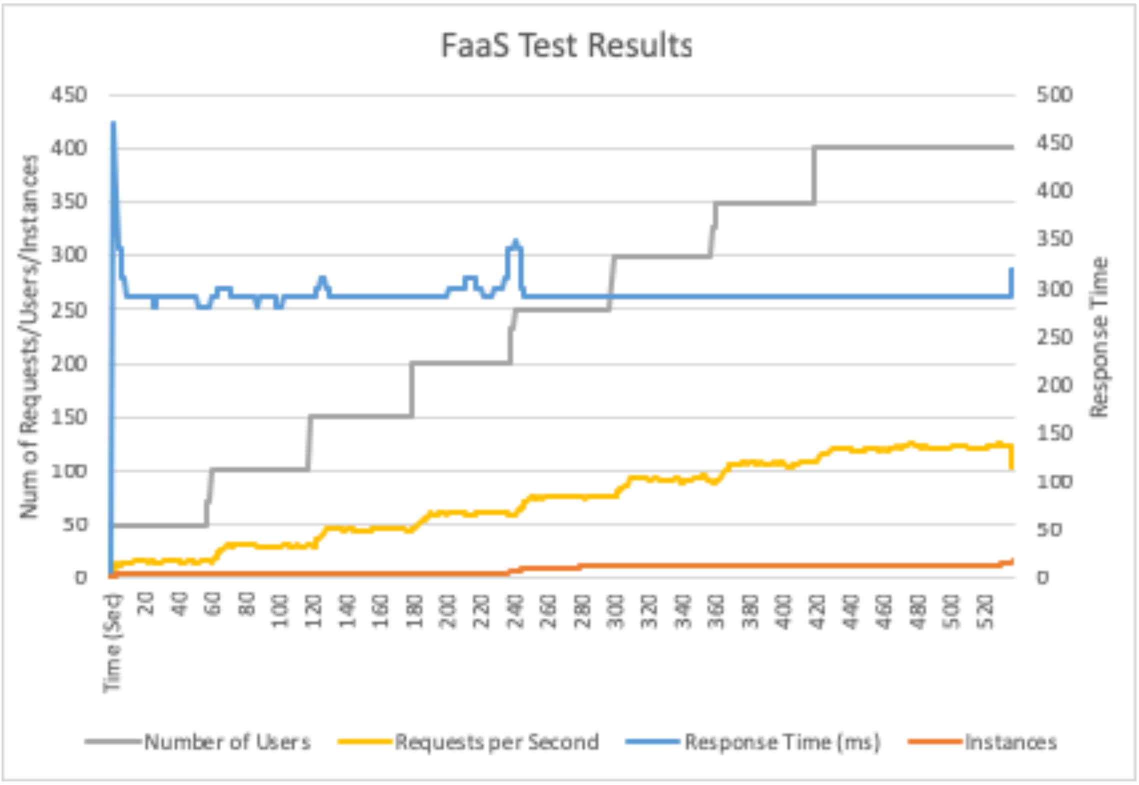


Figure 28 FaaS Test Results

6.1.3 Comparison

Both implementations give the same throughput until 200 users like in the Figure 29. After 200 users;

- Microservice implementation stays same around 60 requests per second. We can understand that the service reached the maximum capacity after 200 users.
- Serverless implementation could handle the all requests and reached to 120 RPS thanks to automatic scaling feature.

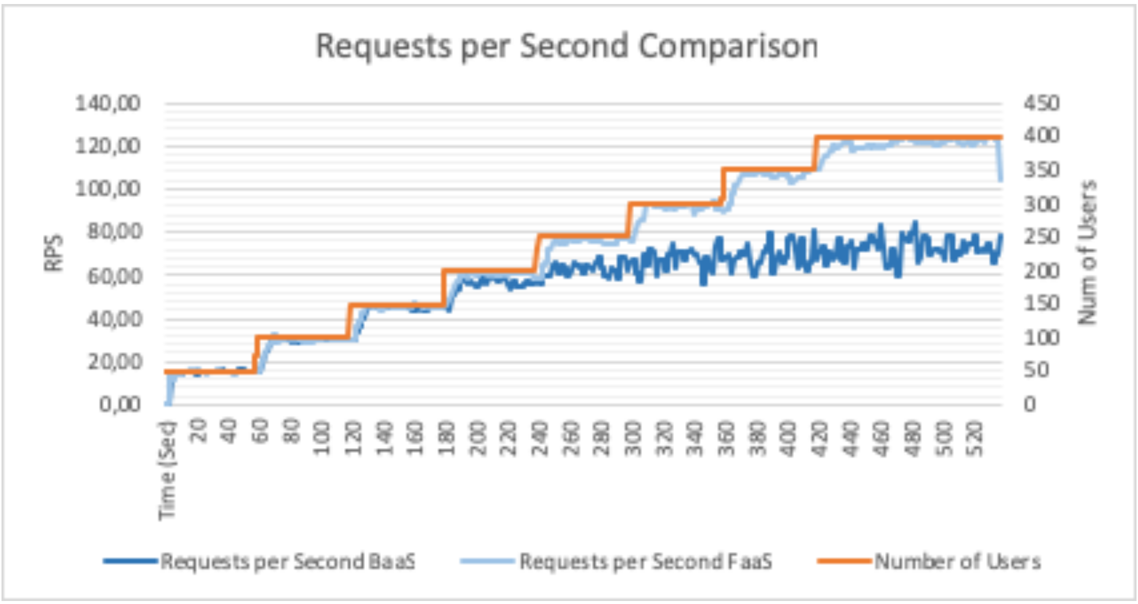


Figure 29 Requests per Second Comparison

If we compare the response times in Figure 30, both services provide same response times until the user count is increased to 200. After 200 users,

- Serverless implementation has a small peak at second 240 when the users count increased from 200 to 250. At that time scaling worked and response times back to normal again.
- But for microservice implementation, response times started to increase dramatically to the end of testing and it reached to 3 seconds.

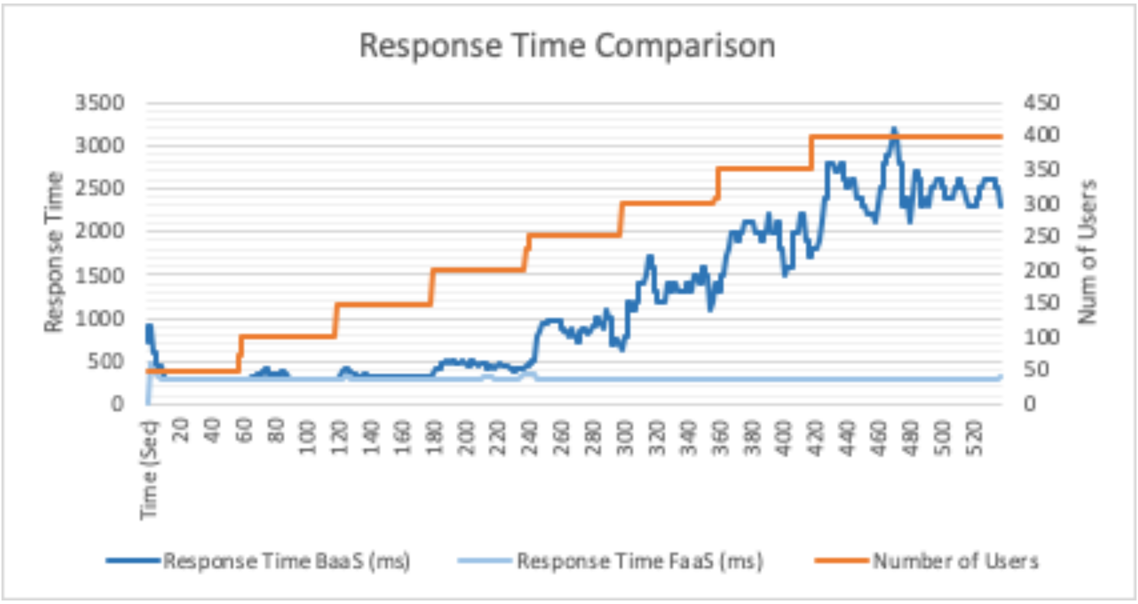


Figure 30 Response Time Comparison

If we use more powerful server for the microservice, response times will be decreased and also it can support more requests, but it comes with a price. Also, it needs to be considered that, these response times are only for this experimental implementation. If there are more dependencies and more resources like database, cache etc. response time will be increased for serverless application since it’s all the resources need to be initialized while running the functions. But that’s not the case for microservice, most resources are created when start-up of application and it’ll respond quickly using the resources that are already initialized.

As a result for this comparison, serverless applications gives more stable response while instant load increases thanks to auto-scaling. But both implementations give the same performance for some average loads as around 50 RPS for this experiment.

6.2 Cost

Microservice applications are deployed on a dedicated instance which is billed hourly per instance [24]. Also, there are ways to scale these kinds of services using load balancing and additional scaling options by creating new instances. For this experiment Standard S2 Service Plan used as a tier.

Serverless applications are deployed on function apps which is billed by per request and the execution time for consumption plans [28] and scaling of the application done by automatically. For this experiment consumption plan is used.

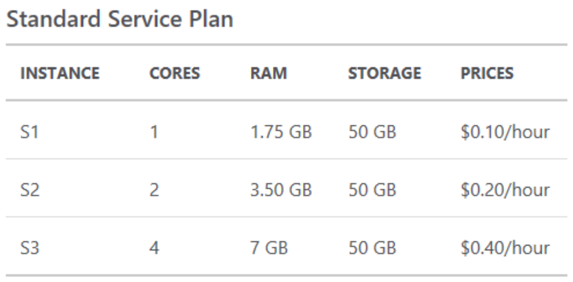
Test results can used to calculate the cost of microservice and serverless implementations like following.

- In microservice implementation total requests per second approximately could reach up to 75 RPS for 400 users and it caused 100% CPU usage on deployed application instance and it made huge impact on response times. So last stable response times and RPS was around 65 RPS with 350 users.

- In serverless implementation total requests per second approximately could reach up to 120 RPS for 400 users and there is no impact on the response times thanks to scaling of function apps.

6.2.1 Microservice Cost Calculation

As we use Standard S2 plan for service application, hourly prices are like in the Figure 31.



Standard Service Plan

INSTANCE	CORES	RAM	STORAGE	PRICES
S1	1	1.75 GB	50 GB	$0.10/hour
S2	2	3.50 GB	50 GB	$0.20/hour
S3	4	7 GB	50 GB	$0.40/hour

Figure 31 Standard service plan pricing for Service App [24]

Microservice cost calculation can be done by multiplying cost and duration like the Equation (1).

$$HPM * CPH = TCPM \tag{1}$$

Input and output values are explained in **Error! Not a valid bookmark self-reference.**.

Table 3 Microservice cost calculation parameters

Parameter	**Description**	**Type**	**Value**
HPM	Hours per month (Total hours for 30 days)	Input	720
CPH	Cost per hour	Input	0,20 $
TCPM	Total cost per month	Output	**144 $**

6.2.2 Serverless Cost Calculation

Calculation is a bit more complex for serverless. Both request count and process time need to be considered in calculation.

If the max RPS is considered as 65 for both implementations to calculate the total number of requests per month Equation (2) can be used.

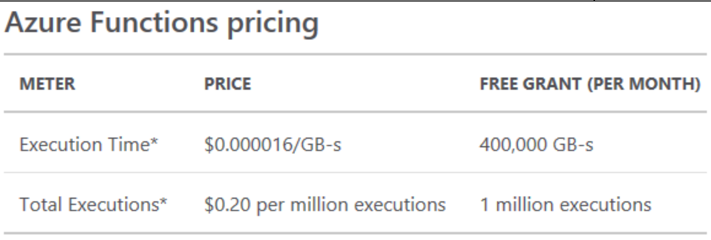
$$RPM = RPS * SPM \tag{2}$$

According to Equation (2) input and output parameters are explained in Table 4.

Table 4 Request count calculation parameters

Parameter	**Description**	**Type**	**Value**
SPM	Seconds per month (Total seconds for 30 days)	Input	2592000
RPS	Number of requests per second	Input	65

RPM	Total number of requests per month	Output	**168480000**



Azure Functions pricing

METER	PRICE	FREE GRANT (PER MONTH)
Execution Time*	$0.000016/GB-s	400,000 GB-s
Total Executions*	$0.20 per million executions	1 million executions

Figure 32 Pricing for Serverless [28]

This experimental application is designed to consume CPU only so there is no significant memory usage for these both applications. But we can make an assumption for usage as 128 MB since it's the minimum memory assigned for a single function instance.

So, resource cost can be calculated according to memory usage per second by subtracting the free grant like in the Equation (3)

$$(MUPR * \mathrm{RTPR} * RPS * SPM - FGFR) * CPGPS = RCPM \tag{3}$$

According the Equation (3) parameters are defined in the Table 5 with the given values and the result.

Table 5 Serverless resource cost calculation parameters

Parameter	**Description**	**Type**	**Value**
MUPR	Memory usage per request	Input	128 MB (0,125 GB)
RTPR	Response time per request (obtained from test results)	Input	300 ms (0,3s)
CPGPS	Cost per GB per second	Input	0,000016 $
FGFR	Free grant for resources	Input	400000
RCPM	Resource consumption cost per month	Output	**94,69 $**

Execution cost can be calculated by multiplying the cost per million requests with the total requests subtracted by free grant like in the Equation (4)

$$(RPM - FGFE) * \frac{CPMR}{1000000} = ECPM \tag{4}$$

Parameters are explained and input/output values are given in the Table 6 for the Equation (4).

Table 6 Serverless execution cost calculation parameters

Parameter	**Description**	**Type**	**Value**
CPMR	Cost per million requests	Input	0,20 $
FGFE	Free grant for execution	Input	1000000
ECPM	Execution cost per month	Output	**33,49 $**

Total cost should be the sum of the resource and execution costs like in the Equation (5).

$$TCPM = RCPM + ECPM \tag{5}$$

Parameters explained in Table 7 for Equation (5).

Table 7 Serverless total cost calculation parameters

Parameter	**Description**	**Type**	**Value**
TCPM	Total cost per month	Output	**128,18 $**

6.2.3 Comparison

With whole month full load for the microservice users can get maximum 65 RPS throughput for used tier in the experiments. Serverless implementation can provide more than 120 RPS within the tests also, it can support more than that with the scaling. But of course, the costs will increase with the number of requests.

Table 8 Microservice and Serverless Monthly Cost Comparison

Microservice Monthly Cost	**Serverless Monthly Cost**
144 $	128 $

According to full load cost calculations comparison will be like in the Table 8. So serverless implementation seems slightly better than microservice implementation for this load. But for a system generally it’s not a case to work on full load for whole month even whole day. There should be some peak hours or peak days. So, if the system gets 10 RPS as average, in the peak hours it can be 50 RPS for example. To support such scenarios, application need to run on a server which can support 50 RPS, in this case resources will be wasted for other times than the peak times. As a result, if we calculate 30 RPS cost for serverless, it will be nearly half of the calculated result above but also it can support 65 RPS for peak times. For microservice we need to keep the same service tier to support 65 RPS and need to pay the same amount.

As contrary, if the number of requests very high for the system such as a social media application running worldwide. Then it will be cheaper to use a dedicated server or microservice application for that. Because the load will be steadier and very high. For such systems it is also an option to use multiple instance for same servers or applications with the load balancers but it's also additional cost and need more maintenance.

6.3 Development

In this experiment, the programming language used is C# for both microservice and serverless applications. MVC approach is used for implementing the microservice application. Serverless applications has their own structures with a static function class and handler method in it. Unlike the microservices, serverless application has one entry point and runs only one process from there. It runs with a static method and all the needed processes need to be completed within that method.

Since serverless functions are static, all the connections to DBs, services used, and all kind of resources needed for the process need to be initialized from each function handler method. But for microservices, all the needed resources can be initialized on start-up of the application using dependency injection and all the initialized services, connections or resources can be used in runtime easily.

As a result, functions are not suitable for complex processes with lots of other resource dependencies. It can be used perfectly for some database queries, add, delete, update actions, sending notifications or such. But it's not suitable for creating reports using different sources or making some long running processes like data migrations on databases or such.

7 Conclusion

To summarize this study, divide an conquer method is also valid for the software systems namely, to divide big systems to small subsystems like microservices and functions -as the subject for this study- makes development and management easier for us. These architectures will be in continuous development in the future with the new requirements coming from the technological developments.

As a result for the experiments in this study, it can be seen there are benefits of serverless on microservices with no maintenance and auto-scaling features, on contrary there are disadvantages of serverless as well such as stateless development which can cause long response times because of initialization of all resources for each function call. Therefore, it can’t be said that the serverless architecture can replace the microservice architecture a least for a near future.

In my opinion, for a better architecture for a good and robust system, serverless and microservice architectures should be combined to have all the advantages of both approaches.

References

[1] The Open Group, “SOA Reference Architecture,” [Online]. Available: http://www.opengroup.org/soa/source-book/soa_refarch/index.htm.

[2] Backendless, “Backend as a Service,” [Online]. Available: https://backendless.com/platform/backend-as-a-service/.

[3] M. Richards, Microservices vs. ServiceOriented Architecture, O’Reilly Media, 2015.

[4] The Open Group, “Service-Oriented Architecture,” [Online]. Available: http://www.opengroup.org/soa/source-book/soa/index.htm.

[5] The Open Group, “Microservices Architecture,” [Online]. Available: http://www.opengroup.org/soa/source-book/msawp/index.htm.

[6] Edvantis, “Microservice Architecture,” [Online]. Available: https://www.edvantis.com/blog/101-series-edvantis-software-benefits-microservices-architecture/.

[7] BitHeads Custom Software Development, “A BaaS Overview,” [Online]. Available: https://telusdigital-marketplace-production.s3.amazonaws.com/iot/user-content/product/818d-o.pdf.

[8] Integrove, “ Backend as a Service (BaaS),” [Online]. Available: http://www.integrove.com/backend-as-a-service-baas/.

[9] M. Fowler, “Serverless Architectures,” [Online]. Available: https://martinfowler.com/articles/serverless.html.

[10] K. Chowhan, Hands-On Serverless Computing, Packt Publishing, 2018.

[11] Maruti Techlabs, “Serverless Architecture the Future of Business Computing,” [Online]. Available: https://www.marutitech.com/serverless-architecture-business-computing/.

[12] Stelligent, “Serverless Delivery: Architecture (Part 1),” [Online]. Available: https://stelligent.com/2016/03/17/serverless-delivery-architecture-part-1/.

[13] Amazon Corporation, “Amazon Web Services - About AWS,” [Online]. Available: https://aws.amazon.com/about-aws/.

[14] Amazon Corporation, “What is Elastic Beanstalk,” [Online]. Available: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/Welcome.html.

[15] Amazon Corporation, “Elastic Beanstalk Pricing,” [Online]. Available: https://aws.amazon.com/elasticbeanstalk/pricing/.

[16] Amazon Corporation, “AWS Lambda,” [Online]. Available: https://aws.amazon.com/lambda/features/.

[17] Google Corporation, “About Google Cloud Platform,” [Online]. Available: https://cloud.google.com/about/.

[18] Google Corporation, “Google App Engine,” [Online]. Available: https://cloud.google.com/appengine/.

[19] Google Corporation, “Google App Engine Pricing,” [Online]. Available: https://cloud.google.com/appengine/pricing.

[20] Google Corporation, “Cloud Functions,” [Online]. Available: https://cloud.google.com/functions/features/.

[21] Google Corporation, “Cloud Functions for Firebase,” [Online]. Available: https://firebase.google.com/docs/functions/.

[22] Microsoft Corporation, “Microsoft Azure Global Infrastructure,” [Online]. Available: https://azure.microsoft.com/en-us/global-infrastructure/.

[23] Microsoft Corporation, “Azure Web Apps,” [Online]. Available: https://azure.microsoft.com/en-us/services/app-service/web.

[24] Microsoft Corporation, “Pricing - App Service,” [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/app-service/windows/.

[25] Microsoft Corporation, “Supported languages in Azure Functions,” [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/supported-languages.

[26] Microsoft Corporation, “Dependency Injection in ASP.NET Web API 2,” [Online]. Available: https://docs.microsoft.com/en-us/aspnet/web-api/overview/advanced/dependency-injection.

[27] Locust, “Locust Documentation,” [Online]. Available: https://docs.locust.io/en/stable/.

[28] Microsoft Corporation, “Pricing - Functions,” [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/functions/.

[29] Microsoft Corporation, “Serverless apps: Architecture, patterns, and Azure implementation,” [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/serverless-architecture/.

Appendix 1 – ThesisProject – Common

Models

```
using Newtonsoft.Json;

namespace Common.Models
{
    public class Error
    {
        [JsonProperty(PropertyName = "message")]
        public string Message { get; set; }

        public Error(string message)
        {
            Message = message;
        }
    }
}
```

```
using Newtonsoft.Json;

namespace Common.Models
{
    public class PostBody
    {
        [JsonProperty(PropertyName = "cpu", Required = Required.Default,
                      NullValueHandling = NullValueHandling.Ignore)]
        public int? CPU { get; set; }

        [JsonProperty(PropertyName = "data", Required = Required.Default,
                      NullValueHandling = NullValueHandling.Ignore)]
        public string Data { get; set; }
    }
}
```

```
using Newtonsoft.Json;

namespace Common.Models
{
    public class SampleModel
    {
        [JsonProperty(PropertyName = "id")]
        public string Id { get; set; }

        [JsonProperty(PropertyName = "name")]
        public string Name { get; set; }
    }
}
```

```
using Newtonsoft.Json;

namespace Common.Models
{
    public class Success
    {
        [JsonProperty(PropertyName = "message")]
        public string Message { get; set; }

        public Success(string message)
        {
            Message = message;
        }
    }
}
```

CommonService

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Threading;
using Common.Models;

namespace Common
{
    /// <summary>
    /// A common test service interface to use in
    /// both microservice and serverless implementations
    /// </summary>
    public interface ICommonService
    {
        /// <summary>
        /// Run some process and return some values
        /// </summary>
        /// <remarks>Returns a list of values according to input</remarks>
        /// <param name="input">Count of items to be generated</param>
        /// <param name="cpu">CPU usage percentage</param>
        object RunGet(int input, int? cpu);

        /// <summary>
        /// Run some process and return success
        /// </summary>
        /// <param name="input">Value to be returned in response</param>
        /// <param name="cpu">CPU usage percentage</param>
        object RunPost(string input, int? cpu);
    }

    /// <inheritdoc />
    public class CommonService : ICommonService
    {
        private char[] _alphabet =>
            "abcdefghijklmnopqrstuvwxyz".ToCharArray();

        /// <inheritdoc />
        public object RunGet(int input, int? cpu)
        {
            ConsumeCpu(cpu ?? 5);
            return GenerateData(input);
        }
        /// <inheritdoc />
        public object RunPost(string input, int? cpu)
        {
            ConsumeCpu(cpu ?? 5);
            return new Success($"Process successful for data {input}");
        }

        private IEnumerable<SampleModel> GenerateData(int itemCount)
        {
            for (var i = 0; i < itemCount; i++)
            {
                yield return new SampleModel()
                {
                    Id = Guid.NewGuid().ToString(),
                    Name = GenerateName()
                };
            }
        }

        private string GenerateName()
        {
            var name = "";
            var numOfChars =  new Random().Next(3, 10);
            for (var i = 0; i < numOfChars; i++)
            {
                var idx = new Random().Next(0, _alphabet.Length - 1);
                name += _alphabet[idx];
            }

            return name;
        }

        private void ConsumeCpu(int percentage)
        {
            if (percentage < 0 || percentage > 100)
                throw new ArgumentException("percentage");

            var duration = new Random().Next(100, 200);
            var mainWatch = new Stopwatch();
            var watch = new Stopwatch();
            mainWatch.Start();
            watch.Start();

            while (mainWatch.ElapsedMilliseconds < duration)
            {
                if (watch.ElapsedMilliseconds <= percentage)
                    continue;

                Thread.Sleep(100 - percentage);
                watch.Reset();
                watch.Start();
            }

            watch.Stop();
            mainWatch.Stop();
        }
    }
}
```

Appendix 2 – ThesisProject – MicroserviceApp

Controllers

```
using Common;
using Common.Models;
using Microsoft.AspNetCore.Mvc;

namespace MicroserviceApp.Controllers
{
    /// <summary>
    /// Test controller definition for microservice implementation
    /// </summary>
    [Route("api/test")]
    public class TestController : Controller
    {
        private readonly ICommonService _commonService;

        /// <inheritdoc />
        public TestController(ICommonService commonService)
        {
            _commonService = commonService;
        }

        /// <summary>
        /// Runs some process and returns a list of object
        /// <param name="input">Item count to be returned</param>
        /// </summary>
        [HttpGet]
        public IActionResult HttpGet(int input, int? cpu)
        {
            var result = _commonService.RunGet(input, cpu);
            return new JsonResult(result);
        }

        /// <summary>
        /// Runs some process and returns success
        /// </summary>
        /// <param name="input">Any string</param>
        [HttpPost]
        public IActionResult HttpPost([FromBody]PostBody body)
        {
            var result = _commonService.RunPost(body.Data, body.CPU);
            return new JsonResult(result);
        }
    }
}
```

Startup

```
using Common;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace MicroserviceApp
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime.
        // Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddScoped<ICommonService, CommonService>();

            services.AddMvc()
                    .SetCompatibilityVersion(CompatibilityVersion.Latest);
        }

        // This method gets called by the runtime.
        // Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app,
                              IHostingEnvironment env)
        {
            app.UseExceptionHandlerMiddleware();
            app.UseHttpsRedirection();
            app.UseMvc();
        }
    }
}
```

Program

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace MicroserviceApp
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateWebHostBuilder(args).Build().Run();
        }

        public static IWebHostBuilder CreateWebHostBuilder(string[] args)
            => WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>();
    }
}
```

Appendix 3 – ThesisProject – ServerlessApp

Functions

```
using Common;
using Common.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;

namespace ServerlessApp
{
    /// <summary>
    /// Test function definitions for serverless implementation
    /// </summary>
    public static class TestFunctions
    {
        /// <summary>
        /// Runs some process and returns a list of object
        /// </summary>
        /// <param name="req"></param>
        /// <param name="log"></param>
        [FunctionName("HttpGet")]
        public static IActionResult HttpGet(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get",
                         Route = "test")]
            HttpRequest req, ILogger log)
        {
            log.LogWarning(FunctionHelpers.InstanceId);

            var commonService = new CommonService();
            var input = req.Query.Get<int>("input");
            var cpu = req.Query.Get<int>("cpu");

            var result = commonService
                .RunGet(input, cpu == 0 ? null : (int?)cpu);
            return new JsonResult(result);
        }

        /// <summary>
        /// Runs some process and returns success
        /// </summary>
        /// <param name="req"></param>
        /// <param name="log"></param>
        [FunctionName("HttpPost")]
        public static IActionResult HttpPost(
            [HttpTrigger(AuthorizationLevel.Anonymous, "post",
                         Route = "test")]
            HttpRequest req, ILogger log)
        {
            log.LogWarning(FunctionHelpers.InstanceId);

            var commonService = new CommonService();
            var body = req.Body.Get<PostBody>();

            var result = commonService.RunPost(body.Data, body.CPU);
            return new JsonResult(result);
        }
    }
}
```

Function Helpers

```
using System;
using System.IO;
using System.Linq;
using Microsoft.AspNetCore.Http;
using Newtonsoft.Json;

namespace ServerlessApp
{
    public static class FunctionHelpers
    {
        /// <summary>
        /// Gets the instance identifier.
        /// </summary>
        /// <value>The instance identifier.</value>
        public static string InstanceId => Environment
            .GetEnvironmentVariable("WEBSITE_INSTANCE_ID",
                                    EnvironmentVariableTarget.Process);

        /// <summary>
        /// Get the specified parameter value from request query.
        /// </summary>
        /// <returns>The get.</returns>
        /// <param name="query">Query.</param>
        /// <param name="param">Parameter.</param>
        public static T Get<T>(this IQueryCollection query, string param)
        {
            var value = (object)query?
                .FirstOrDefault(a =>
                    a.Key.Equals(param, StringComparison.OrdinalIgnoreCase)
                .Value.FirstOrDefault();

            if (typeof(T) == typeof(int))
                value = int.TryParse((string)value, out var intOut)
                           ? intOut
                           : 0;

            return (T) Convert.ChangeType(value, typeof(T));
        }

        /// <summary>
        /// Get the request body as specified type.
        /// </summary>
        /// <returns>The get.</returns>
        /// <param name="body">Body.</param>
        public static T Get<T>(this Stream body)
        {
            using (var reader = new StreamReader(body))
            {
                var strBody = reader.ReadToEnd();
                return typeof(T) == typeof(string)
                    ? (T)Convert.ChangeType(strBody, typeof(T))
                    : JsonConvert.DeserializeObject<T>(strBody);
            }
        }
    }
}
```

Appendix 4 – ThesisProject – Testing

Locust File

```
from locust import HttpLocust, TaskSet

import resource

resource.setrlimit(resource.RLIMIT_NOFILE, (10240, 9223372036854775807))

def testGet(self):
    self.client.get("/api/test?input=10&cpu=20")

def testPost(self):
    self.client.post("/api/test", json={"data": "some test string", "cpu":
20})

class LoadTestSample(TaskSet):
    tasks = {testGet: 1, testPost: 2}

class WebsiteUser(HttpLocust):
    task_set = LoadTestSample
    min_wait = 1000
    max_wait = 5000
```

Microservice Testing Script

```bash
#!/bin/bash

RunSwarm()
{
    curl -X POST http://localhost:8073/swarm \
        -H "Content-Type: application/x-www-form-urlencoded" \
        -d "hatch_rate=$1&locust_count=$2"
}

StopSwarm()
{
    curl -X GET http://localhost:8073/stop
}

AutoSwarm()
{
    n=0
    while [ $(($n * $2)) -lt $3 ]
    do
        n=$(( n+1 ))
        RunSwarm $2 $(($n * $2))
        sleep $1
    done

    sleep $1

    StopSwarm
}

if [ "$1" == "start" ]
then
    locust --host=http://cg-microservice.azurewebsites.net --port 8073 &
    sleep 2
    AutoSwarm 60 50 400 &
    sleep 1
    open "http://localhost:8073"
elif [ "$1" == "stop" ]
then
    ps -ef | grep locust | grep 8073 | awk '{ print $2 }' | xargs kill -9
    ps -ef | grep test_microservice | grep start \
        | awk '{ print $2 }' | xargs kill -9
else
    echo "Undefined command"
fi
```

Serverless Testing Script

```
#!/bin/bash

RunSwarm()
{
    curl -X POST http://localhost:8072/swarm \
        -H "Content-Type: application/x-www-form-urlencoded" \
        -d "hatch_rate=$1&locust_count=$2"
}

StopSwarm()
{
    curl -X GET http://localhost:8072/stop
}

AutoSwarm()
{
    n=0
    while [ $(($n * $2)) -lt $3 ]
    do
        n=$(( n+1 ))
        RunSwarm $2 $(($n * $2))
        sleep $1
    done

    sleep $1

    StopSwarm
}

if [ "$1" == "start" ]
then
    locust --host=http://cg-serverless.azurewebsites.net --port 8072 &
    sleep 2
    AutoSwarm 60 50 400 &
    sleep 1
    open "http://localhost:8072"
elif [ "$1" == "stop" ]
then
    ps -ef | grep locust | grep 8072 | awk '{ print $2 }' | xargs kill -9
    ps -ef | grep test_serverless | grep start \
        | awk '{ print $2 }' | xargs kill -9
else
    echo "Undefined command"
fi
```