Bachelor Degree Project

A comparison between serverless and Docker container deployments

- In the cloud

Author: Andras Balla
Supervisor: *Johan Hagelback*
Semester. VT/HT 2017
Subject: Computer Science

Abstract

Designing an application from top to bottom is a challenge for any software architect. Designing an application to be deployed in the cloud adds extra complexity and a variety of questions to the task. One of these questions is how to deploy an application? The most popular choices at this time is either Docker containers or serverless functions. This report presents a comparison between the two deployment methods based on cost and performance. The comparison did not yield a conclusive winner but it did offer some key pointers to help with the decision. Docker containers offer a standardized deployment method for a low price and with good performance. Before choosing Docker the intended market needs to be evaluated given that for each region Docker needs to serve, the price also increases. Serverless functions offer auto scaling and easy global deployments but suffer from high complexity, slower performance and an uncertain monthly price tag.

Preface

You can have a preface in the report if you want, but it is not necessary. In this you can write more personal reflections on your degree project. In the preface you can also take the opportunity to thank the people who have been particularly helpful during the report writing, for example if you had any contact with a company that helped with the project, people that guided or helped you during the project, or your family and friends that supported you during the project. The preface shall not be longer than half a page.

Contents

1 Introduction

This chapter gives a brief summary of what cloud providers have to offer customers and explains why it is advantageous to spend some time and plan out which services suit your application before deploying in the cloud.

1.1 Background

Cloud providers offer a service where companies and developers can easily deploy software without needing to worry about configuring, buying and maintaining the infrastructure needed to do so. Examples of popular cloud providers are Amazon Web Services(AWS), Google Cloud Platform and Microsoft Azure. Originally they offered *Infrastructure as a Service[1]* but quickly evolved past that stage to accommodate customer demand. It is hard to classify a cloud provider using IaaS, Paas or FaaS AWS and Google Cloud offer all of the above in one way or another. This rapid expansion of services has made cloud providers a viable deployment method where companies chose to deploy most or all of their systems in the cloud. Many services offer similar functionality but have different payment models or configuration requirements. Today the services offered by cloud providers will satisfy most of the needs a project could have. There are a number of ways to deploy a project whether it is a simple VM that needs to be fully configured, a pre-configured VM, a Docker container with auto-scaling or as serverless functions. It is up to the customer to pick and choose the services that best suits them. A cloud provider offering a VM is expected but they also offer configurable networking[2], various database solutions[3], queuing tools[4], messaging tools, monitoring and much more.

In the context of this experiment performance is measured in response time(ms) and the load is determined by hits per second on an HTTP endpoint.

Serverless functions[5] also known as *Function as a Service* is the newest deployment method offered by cloud providers. One shouldn't take the name literally, the servers running the code are still there but they are fully managed by the cloud provider. Serverless functions offer load balancing, autoscaling, security maintenance, compute infrastructure and high availability. The business model is pay only for what you use, meaning that you pay per request and processing time. This approach offers a potentially great way for customers to lower their costs but also a potential pitfall where the increase in users results in a price which is no longer acceptable for the

customer. Serverless functions need to be monitored and hidden behind an API gateway to protect from unexpected costs. Additional costs do come with this type of deployment like an API gateway, data transfer etc. The pay for what you use business model really does mean that everything will be billed. These aspects need to be taken into consideration. Serverless is only great for use cases that do not have a long execution time, heavy data transfers or public API endpoints that handles heavy traffic.

Managed Docker containers are a great way to deploy in the cloud. A service that manages Docker containers can easily reduce the work required to deploy in the cloud. A great example is AWS Elastic Container Service(ECS). ECS offers a repository to store Docker images, functionality to manage/scale clusters and will take care of the deployment and creation of compute instances in which the Docker container will be executed. The business model is simple you pay for what you use i.e any and all VMs that ECS will create to run the cluster of Docker containers.

Amazons VMs are called EC2[6] (Elastic Compute 2) they offer a virtual machine that is fully configurable or with a base configuration using predefined templates. The business model is based on the size of the VM (number of cores, RAM and storage) which is then billed by the hour while the VM is running.

Deploying to the cloud might sound simple but in reality, there are numerous questions that a cloud architect needs to ask and answer before doing so. Questions like: What managed services do we choose? How do we handle load? Do we deploy to Containers or do we use serverless? Most of these questions boil down to cost since customers try to find a collection of services with the lowest monthly cost for running their application.

1.2 Related work

Rafal Gancarz [7] wrote an interesting article about the economics of serverless computing. The comparison made between the serverless approach and a provisioned instance points out that serverless does not fit a compute-intensive application given its price model. Gancarz instead points to the strength of serverless applications handling unpredictable load at a much lower cost instead of having provisioned instances on standby. Gancarz does point out the perils of running a pay per use application in an environment where an automatic retry might try to contact your application for days causing a massive increase in cost. Gancarz concludes that migrating

some workloads to serverless will offer substantial financial gains.

Adam Eivy [8] also offers an article about the economics of serverless cloud computing. The article offers an in-depth description of what serverless is and the pitfalls of its business model. Even though the article is about serverless, parallels are drawn between serverless and reserved compute instances which makes it a great article for this experiment. The difference between serverless and reserved compute is simply explained by Eivy: “It’s like the difference between a rental car and a taxi: you will be charged for the rental car even if you park it for a week, unlike a taxi.”[9]. Eivy then continuous with a breakdown of the serverless business model. The example used in the article is AWS Lambda where a function receives 150 hits per second and an execution time lower or equal to 100 ms. Eivy calculates that this function would cost 167$/month and 200$/month as a reserved compute instance. Serverless is indeed cheaper in this case. Eivy also calculates the cost of the same function but with 30k hits per second which would cost 55.000$/month or 18.000$/month as reserved compute. This calculation shows the importance for cloud architects to perform cost analysis with both initial load and predicted load for an application before making a decision. Eivy concludes that analyzing the projected load (hits per second) and the execution time of an application is key to determining whether or not it is suited as a serverless function.

Edwin F. Boze, Christina L. Abad and Monica Villavicencio[10] author the third article used in this experiment. The goal was to create a tool for calculating cloud computing costs. The article uses three methods. First, the authors ran a survey to shed light on the adoption rate of cloud computing of Ecuadorian organizations. The survey was successful with a total of 92 responses from organizations active in various sectors. In these sectors the Information and communication group makes up 34.78 % of the responses. The majority of the organizations answer that they opt for fixed monthly payments instead of using on-demand, offering the justification that they need to adhere to fixed annual budgets. Note that this survey was conducted in Ecuador from May through June 2017. Secondly, a case study was conducted on an existing application, namely an invoice compression service which receives invoices every day and compresses them for long term storage. A comparison was made between serverless deployment and reserved VMs for said service. They found that for this specific case serverless offered a slower average response time of 1.7s vs 0.36s but a

significant decrease in yearly cost 7.54$ vs 297.84$. This is a great finding and the response time is not an issue for said application given that the required time frame for the total work order execution is set at a maximum of 1 hour. Lastly, the authors present a tool that can be used to calculate the cost of cloud computing based on the three models Reserved VMs, On-Demand VMs and Serverless Computing. The tool will take response times, execution times and cost into consideration in order to predict cost based on the needs of the user. The tool was successfully developed and released but is sadly no longer available at this point in time 25th of March 2018. The findings in this article are very interesting seeing that bigger organizations are slower to adapt to new business models and the case study showing a great example where provisioning VMs for peak load which in this case is 3 hours a day will cost a lot more money than an on-demand serverless functions.

1.3 Problem formulation

Deploying software to the cloud is widely used and a growing deployment method for software. A common problem for an architect is determining which services can be used to fulfill requirements, reduce maintenance cost, deployment costs, operational costs, handover costs and increase performance.

Finding the right combination of services for a specific application is no easy task. Currently, there is no simple way to compare services based on the application to be deployed. What services to use all comes down to the use case of said application. A service that fits a banking application will not necessarily be the best choice for a social media site. In this paper, the relationship between the aforementioned variables will be investigated for two services namely *serverless deployment[11]* and *managed Docker containers[12]*. The comparison will be based on cost-effectiveness and performance under different load patterns. Both serverless and Docker services will deploy the same web application for consistency. The experiment will produce a document describing their relationships as a matrix. The whole document can be used by architects as a guideline for making decisions between these two services. The investigation will be conducted on the platforms Amazon Web Services and Microsoft Azure to achieve a general interpretation.

1.4 Motivation

Choosing infrastructure at the beginning of a software project is an important step and is crucial to the success of the application. The initial decisions made at the start of the project will set the boundaries for the application. These decisions can be costly to change at a later date, therefore, architects strive to set up a long-term environment for the application. Creating a document for architects to aid in these decisions will save both time and money for the projects that decide to make use of it.

1.5 Objectives

O1	Implement and deploy a AWS Lambda (Serverless function)
O2	Implement and deploy a Azure Function (Serverless function)
O3	Implement and deploy a Docker container to each cloud provider
O4	Find and document load patterns for popular web services i.e Web Store, Social media site etc
O5	Run performance tests mimicking the load patterns found above
O6	Analyze the incurred cost of the performance tests
O7	Conduct interviews with System/Cloud architects from Jayway

The expected result is that the document will show that a serverless approach will, in most cases, be more cost-effective and perform better with a reduced development time if implemented as microservices.

However, there should be a breaking point at a high number of incoming requests per second where containers will be more cost effective and offer better performance under continuous high load. There should also be a minimum traffic level where serverless is always better. The relationships for these services are complex and the two different development methods offer a different level of complexity for the developers themselves, where code complexity alone might rule out the serverless approach.

1.6 Scope/Limitation

The comparison will only be done on AWS and Azure and out of all

the various managed services that can be used to deploy an application only the serverless and Docker container deployments will be evaluated. The serverless functions will be configured to have 128MB memory thus the prices for this tier will apply. Additional data for cost comparison will be gathered by performing cost calculations using official pricing calculators. The data will be generated by applying the user patterns for five different web services: Social media, E-Commerce, Banking, forums and web services offering a service like Spotify, Google Search etc.

1.7 Target group

The primary target group is Cloud architects and developers that are conducting a deployment or migration to the cloud.

1.8 Outline

The next chapter describes the method that has been used during the project. After that follows Chapter 3 where the implementation is covered. This is then followed by Chapter 4 where the results are presented and Chapter 5 presents the analysis of the result. The last part of the report ties everything together with a discussion found in Chapter 6 and the conclusion in Chapter 7.

2 Method

A controlled experiment has been conducted where a web application is deployed to AWS and Azure in a Docker container and as a serverless function. The web application was then tested under different load patterns that mimic traffic on different websites.

After the experiment, a few interviews were conducted with industry professionals to shed light on the complexity of working with a serverless application versus a dockerized application.

2.1 Data

The experiment produced two types of quantitative data from the load testing. Firstly there is the response times of the web applications that show whether or not the different approaches are feasible performance wise. Secondly, there is the incurred cost of the performance test which is used to compare the operational costs of the two services. The data comparison was done in three ways: AWS Lambda vs AWS Container Service, Azure Functions vs Docker container, and the comparison of the average results from both platforms.

Extra data was gathered using cloud pricing calculators to achieve a more precise cost comparison including different levels of load. The cost of serverless functions depends heavily on three factors. Firstly the allocated memory to the function. The allocated memory has a sequential effect on the price. Meaning if 128 MB costs 20 $ then 256 MB will cost 40$ and 384 MB will cost 60$ and so on. The number of handled request is a fixed price where 1 million requests cost 0.2$ and lastly, the execution time is billed by the second.

The cost-performance comparison is important given the different price models. A serverless application is charged by the number of incoming requests and execution time while Docker containers deployed on a VM are charged by the hour, regardless of traffic.

2.2 Tools

The web application has been developed using Node.js. The reason behind the choice of programming language is due to the fact that Node.js can be used to create both serverless functions and dockerized applications. To make the load test as accurate as possible the web application needs to be

as similar as possible for all deployments.

Since the experiment compares response times Node.js was chosen given its shorter startup time. It is important that the comparison compares the two services and not the startup time of different programming languages, hence the choice of the language with the shortest start time.

The load testing has been conducted using Flood IO and BlazeMeter. To be able to conduct the experiment a testing tool is needed that can test a remote HTTP endpoint and perform a load test using a predefined pattern i.e can send requests in intervals, include delays, perform rapid simultaneous requests to mimic user activity. The tool also needs to create a report and visualize the results so that they can be included in this report.

2.3 Load Patterns

The use of different load patterns during the experiment is to provide insight into the needs that different web applications have. As for example, an e-commerce site would greatly benefit from the serverless auto-scaling during the night and work hours. Both AWS and Azure have a monthly free tier on their serverless service so the number of requests used must be higher than the limit. For this experiment, 3 million requests will serve to produce data to be analyzed.

The load patterns will also offer insight into the auto-scaling capabilities of serverless functions. Specific load patterns can trigger a series of new functions given their implementation. Serverless functions do not have a built-in queue system, therefore 10 simultaneous requests will result in the creation of 10 new instances all of which have to start a web server to execute the request. This works as intended but this scenario means that all of the incoming 10 requests will suffer from cold starts. A scenario that fits serverless perfectly is in turn 10 requests coming in over the course of 1 minute with an execution time of 90 ms. The first request will suffer a cold start while the following 9 requests all get handled by the already running instance, therefore, the total response time is lower.

The experiment will use a few different load patterns. First many simultaneous users to show peak load. Few simultaneous to show low load. These two patterns can then be combined to show a pattern that has high load during the day and then low for the rest of the day. Many users growing over a period of time to show the ramp up just before peak load. Few users with a long delay between them to simulate constant low load that many utility

services may encounter. Combined these patterns can show even more complicated load patterns with user activity peaking and dropping multiple times a day or regularly.

2.4 Interviews

The interviews were used to shed light on other costs besides operations like development costs, handover costs and maintenance costs. The development process needs to be taken into consideration to achieve a meaningful result. Therefore the interviews were conducted with industry professionals who have experience in both development processes and who work at Jayway AB.

2.5 Reliability and Validity

The experiment is conducted on two specific platforms, other platforms might have different prices or price models for these services. The price for running these services might also change over time making the result harder to reproduce. This also offers validity to the experiment since it is performed on two competing platforms it gathers data from multiple sources and better reflects reality.

Given that the experiment is run on two different platforms the choice of region is important. When choosing a region one decides the physical data center to be used. Both cloud providers have a region in Ireland, therefore, the experiment will be conducted exclusively in that region.

Currently both AWS and Azure have a free tier on their serverless service which will be a part of the result and comparison. This might change or be removed in the future. The free tier for serverless functions is identical on both platforms.

When deploying the Docker containers one chooses the VM that they will be deployed on. The choice of VM directly dictates the hourly rate. In the future, the chosen VM might be removed or the price might change.

The interviews shed light on costs incurred during development projects at Jayway AB these costs might differ at different companies and even at Jayways future projects.

2.6 Ethical Considerations

The data collected from the interviews will be handled according to the Swedish personuppgiftslagen (PuL) to protect the participants.

Sensitive information about Jayway AB that might occur during the interview will be removed from the transcribed document.

3 Implementation

In this chapter, the implementation of the web services, their deployment methods, and the load tests will be explained in detail.

3.1 Web Services

The web services used in this experiment are simple Hello world applications deployed on AWS and Azure as both serverless functions and Docker containers. The goal of the experiment was to do a comparison between two deployment methods, therefore, the web service is unimportant. The sole purpose of these applications is to have a running application that the tests can talk to and receive a response from. The language used was Node.js and both the serverless functions and dockerized applications implement the same functionality for consistency.

3.2 Serverless functions

The serverless functions are both deployed in the same region, EU Ireland. They also have the same allocated memory, which is 128 MB. There is, however, one slight difference between them. AWS functions need an API Gateway to receive HTTP requests while Azure functions have built-in HTTP triggers. This minor difference will affect the price of the serverless functions since the AWS API gateway will incur an extra cost of 3.50 $ per million incoming requests. This cost will be included in the test results as part of the cost for running a serverless function on AWS since it is the only way for an AWS serverless function to receive external HTTP requests.

3.4 Docker Containers

The Docker containers are also deployed in the same region EU Ireland. On AWS, the service ECS was used to deploy the Docker container. This service does not add any extra costs on top of running the VM for the application. The underlying instance running on AWS is a t2.small which is the smallest instance available when deploying with ECS. T2.small was chosen since it has 2GB of RAM and 1 vCPU which is identical to the instance used on Azure. Smaller instances did not match either CPU or RAM. On Azure, a single VM is deployed and used to host the Docker container. This VM has the same size as the AWS one and works in the same way.

3.5 Load tests

The load tests are the method the experiment uses to gather most of its data

on cost and performance. The load patterns were implemented using two tools: flood.io and blazemeter.com. These are both online tools and are very similar to each other and function in the same way. They both generate load with variable inputs. In order to simulate the different load patterns, the tests are configured in different ways. The number of simulated users defines the number of simultaneous requests sent. The duration of the test dictates the total amount of requests sent. This first static test is done with flood.io where the load patterns with stable constant load is executed. BlazeMeter can be configured to deploy a ramp-up time and a delay between the request. This will be used to execute load patterns where requests are sent with a set delay simulating user patterns where requests come in with a significant delay, i.e low load.

3.6 The specific configuration for the load patterns

Given the nature of serverless functions, there is little reason to load tests them for more than 20 minutes. Once the optimal amount of functions are created to handle the incoming load, the functions will have the same response time and performance will stabilize. Therefore, the load patterns have been broken down into several pieces to simulate more complex load patterns. During the experiment, the execution of the test has a cool down period of 15 minutes to ensure that all serverless functions have scaled down. The dockerized applications do not have any inbuilt auto-scaling capabilities, and given that they have an uptime of 100%, the short duration of the load tests does not have a negative effect on the outcome.

The duration of 20 minutes is due to some tests having a ramp-up capability so the duration needs to include data where the results show serverless functions scaling up as the load ramps up.

A total of 13 tests were run with different configurations. Table 3.1 shows the configurations used during testing. The medium and high load tests were run 4 times, once for each web service. The rest of the tests were run once calling all four web services at the same time. The high and medium load tests were conducted separately due to two reasons: firstly it is costly to simulate 500 and 1000 users calling a web service at the same time and secondly it is better to gather stress test data with minimum outside interference. The rest of the tests had a low amount of simulated users, therefore, there was no problem generating the required load and performing the test simultaneously gathers good data for comparison given that the results suffer from the same outside interferences like network latency, noisy neighbors in the data center an so on. These tests cover a few typical load patterns. Constant load over a period of time is covered by Constant load, low load, Medium load and High load. The rest cover load patterns where load increases over time.

Type	**Users**	**Duration** (min)	**Ramp-up** (sec)	**Delay** (ms)
Low load	10	20	5	500
Constant load	40	20	60	1000
Peak load	20	20	1080	1000
2 sec delay	20	20	1080	2000
Medium load	500	20	30	500
High load	1000	20	30	500
1 min delay	40	20	1080	10000

Table 3.1: Load test configuration.

4 Results

The results of the experiment consists of two datasets, the data gathered from the load tests found in Appendix A and the cost of Docker containers and serverless functions produced by calculations found in this chapter.

4.1 Load tests

Table A.1 shows the results where 40 simulated users sent requests with a one minute delay between the requests. The data shows that serverless functions have the highest maximum response times. Serverless functions also have double the response time of their Docker counterparts. This is due to the ramp-up period of 18 minutes triggering multiple cold starts during the load test as shown in Figure 4.1.

Figure 4.1 shows the average response times of the four web services during the 1 minute delay test. The blue line is the amount of simulated users which constantly increases due to the 18 minute ramp up period and peaks out at 40 users on the 18 minute mark. The yellow and green line show the two serverless functions where yellow is a Azure function and green is the AWS Lambda. The Azure function graph clearly shows the effect cold starts have on response times by peaking periodically. The red and purple lines represent the two Docker deployments where red is Azure Docker and purple is AWS Docker. The Docker graphs show a lower average response time and less fluctuation in response time during the tests.

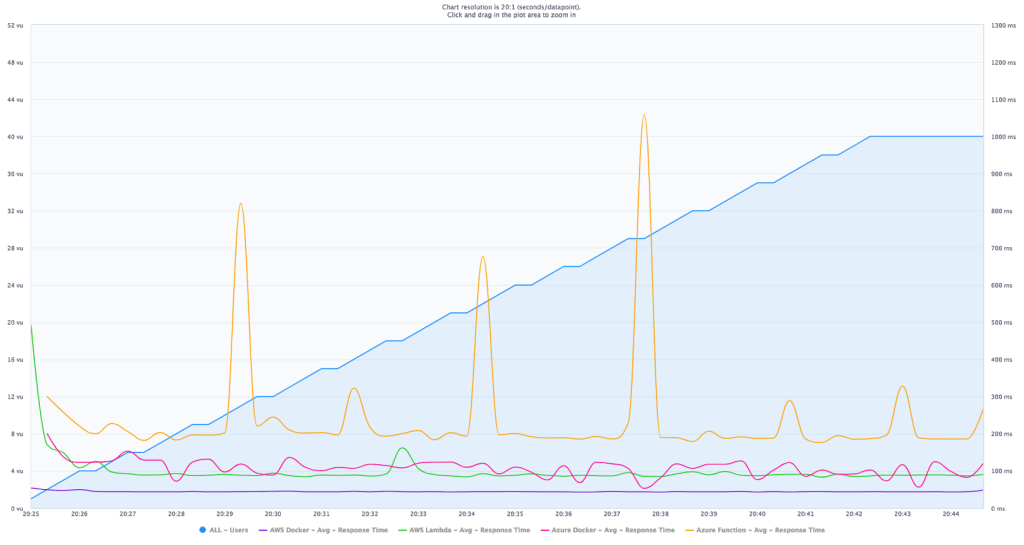

Figure 4.1: Response times of the 4 web services during the 1-minute delay test and simulated users[1].

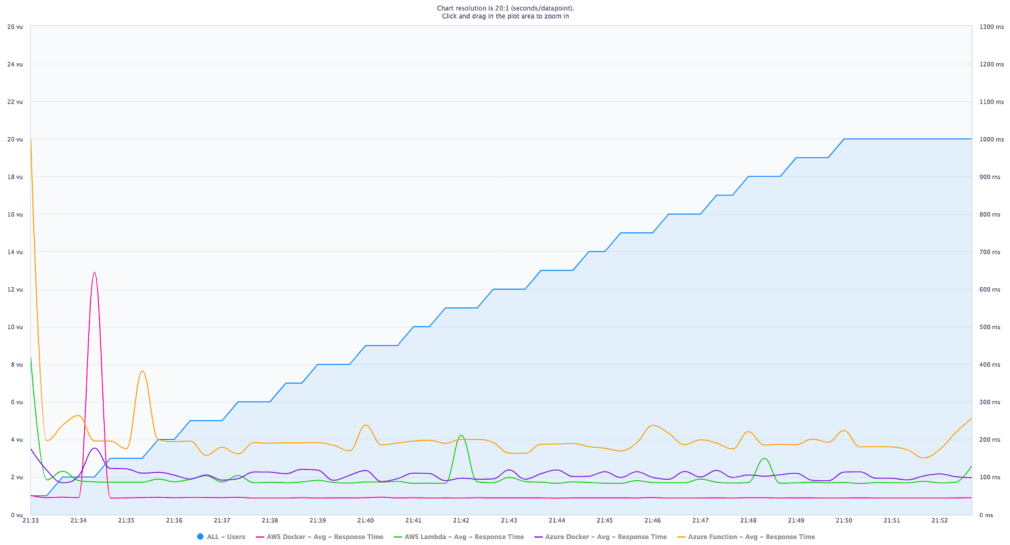

Figure 4.2: Response times of the 4 web services during the 2-second delay test and simulated users[2].

Figure 4.2 shows the average response time of the four webservices during the 2 second delay test. The blue line representing simulated users increasing to 20 concurrent users. The graph shows that all four web services stabilize with response times under 300 ms after 3 minutes and show limited variation. AWS Docker showed by the purple graph is the most stable showing only a single peak and Azure function the most unstable with multiple peaks.

Table A.3 and Figure 4.3 shows that serverless functions are more stable under constant load compared to increasing load as shown in previous tests. The fluctuations in response times shown in Figure 4.3 are much smaller than the ones seen in Figures 4.2 and 4.1.

Comparing the results of Table A.4 to Table A.3 shows that these tests have produced very similar results. The amount of simulated users does

[1] Link to 1 minute delay graph picture:
https://drive.google.com/open?id=1WkHqS1ncWTag8rxE4NoGMIS1Q3eQ-YrP

[2] Link to 2 second delay picture:
https://drive.google.com/file/d/1JbqH9plBBisx4Bp9GdAmSh79dCTEmIVJ/view?usp=sharing

not have a major effect on serverless functions performance.

Overall the tests show that under most circumstances the average response time of all the web services is under 200 ms which is widely considered as adequate performance[13].

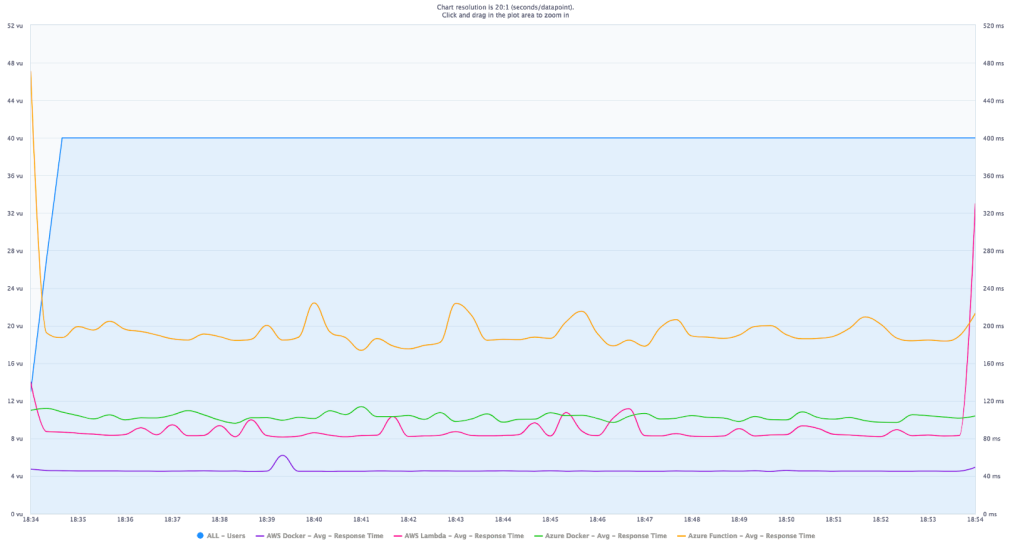

Figure 4.3: Constant load test[3]

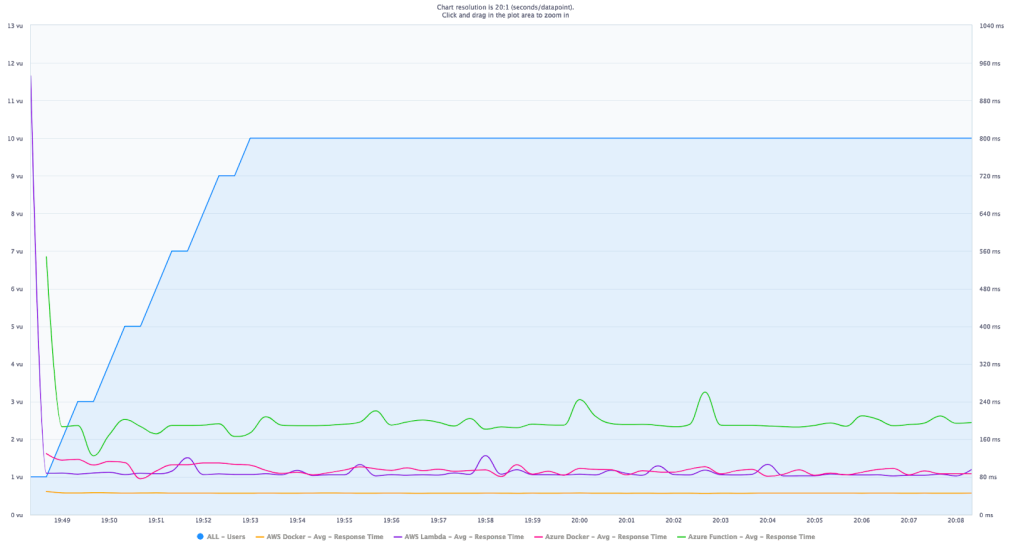

Figure 4.4: Low load test[4]

[3] Link to Constant load graph: https://drive.google.com/file/d/1Auattah7U-qT5cm6tHg_52ndheJs6W05/view?usp=sharing

[4] Link to Low load graph: https://drive.google.com/file/d/1QtmFzHwlswHxjt6Y5W20UbsRcNe8kjK1/view?usp=sharing

Figure 4.4 shows the same behaviour as Figure 4.3. During constant load serverless functions have much lower fluctuations in response times. In Figure 4.4 the two serverless functions are presented by the green and purple lines. Docker is shown by the yellow and red lines. As we can see after the first minute all four web services stabilize and the variation in response time is not higher than 100 ms.

Label	**AWS Docker** (ms)	**AWS Lambda** (ms)	**Azure Docker** (ms)	**Azure function** (ms)
1 minute delay	44.13	90.92	104.4	232.46
2 second delay	46.569	91.737	103.843	193.753
Constant load	45.562	87.087	102.737	192.121
Low load	45.356	88.162	93.28	194.42
Peak load	43.536	89.849	102.115	186.421
Medium load	44	84	98	520
High load	46	87	98	522
Average times	45.02	88.4	100.34	291.6

Table 4.1 Average response times from all the tests.

Table 4.1 shows the average response times gathered from the tests. Azure functions have longer response times for the High and Medium load tests while other tests and web services achieve very similar average response times regardless of circumstances. These results show that on average the response times of the different deployment methods do not vary significantly across the different tests.

4.2 Cost

In this experiment, Docker has a fixed cost due to no auto scaling, therefore the cost can easily be calculated and presented per month as shown in table 4.2

Vm Size	**AWS Name**	**AWS Price**	**Azure name**	**Azure Price**
1 Core 2GB	t2.small	16.83$	B1MS	17.74$

2 Core 4 GB	t2.medium	33.97$	B2S	35.48$
2 Core 8 GB	t2.large	67.93$	B2MS	70.96$
4 Core 16GB	t2.xlarge	135.86$	B4MS	141.91$
8 Core 32GB	t2.2xlarge	271.72$	B8MS	283.82$

Table 4.2: Monthly price of a few popular reserved instances on AWS and Azure

Table 4.3 shows examples of what a serverless function can cost. The prices in the table include the free tier which is why the first three are so cheap. For this experiment, the AWS serverless function needed an AWS API gateway to expose the function as an HTTP endpoint which is why the price for the gateway is included in the data.

Comparing Table 4.2 and 4.3 the data shows that a single Docker container is cheaper to run than a serverless function.

Total Requests	**Requests per second**	**Execution time (ms)**	**Memory size (mb)**	**Azure Price**	**AWS Price**	**AWS Gateway Price**
2,592,000	10	100	128	0.4$	0.32$	9.072$
2,592,000	10	500	128	0.4$	0.32$	9.072$
2,592,000	10	1000	128	0.4$	0.32$	9.072$
2,592,000	10	2000	128	4.37$	4.45$	9.072$
129,600,000	50	200	128	76.36$	73.063$	453600$
129,600,000	50	1000	128	278.60$	263.39$	453600$
259,200,000	100	100	128	103$	98.98$	907200$
259,200,000	100	200	128	155.2$	152.99$	907200$
259,200,000	100	500	128	307.8$	315.03$	907200$
259,200,000	100	1000	128	563.800$	585.44$	907200$

Table 4.3: A few examples of the cost of serverles functions

5 Analysis

The results clearly show that Docker is far superior in both performance and monthly cost but in reality it is not that simple.

5.1 Performance comparison

Performance-wise the Docker configuration used in the experiment has a hard limit since it cannot scale automatically. This limit is dictated by the underlying VM so for the experiment it is 1 CPU and 2GB of RAM, this gives serverless the advantage since it can scale up infinitely. If the web service would have a longer execution time, the load would increase beyond the scope of this experiment. Eventually a limit will be reached and the Docker instance will start dropping requests trying to cope with the incoming traffic.

Both Figure 4.1 and 4.2 offer a visual representation of the effect of cold starts. When comparing the Docker response time to serverless the results show that Docker has very little variation in the response times while serverless varies wildly as load increases which can be seen in Tables A.1, A.2, and A.5

After analysing the average response time across all the tests shown in Table 4.1, Docker seems to be superior based on average response time. Dockers response time takes half the time than the serverless functions do, compared on the same cloud platform. There are probably many factors that play into this, one major factor is that the serverless functions rely on API Gateways to do half of the work for them, namely security and routing while Docker simply listens on an open port. Another factor is the lack of a queue for serverless functions if a new request arrives at the API Gateway and there are no available functions a new one is created, this adds to an increase in the average response time of serverless functions.

Analyzing Figure 4.4 with a steady flow of incoming traffic Docker is still superior response time wise. The traffic is steady but serverless response times are still uneven since any of the requests can trigger a new function if it comes in 1ms to soon. The test shown in Table 5.1 had ten simulated users with a delay of 500ms which is higher than the average response time and should give the serverless functions plenty of time to execute the previous round of requests. This test is where serverless should have outperformed Docker under near perfect circumstances for a serverless application but that

was not the case.

5.2 Cost

The data gathered shows performance when the web service is deployed in Europe and the requests originate from Europe. If the web service needs to be available globally, then more instances of Docker needs to be deployed to cover different regions thus increasing the price. Serverless functions can be made available globally for no extra cost. If the target region for a web service can be covered by a single deployment, then the data shows that it is more cost effective to deploy as a Docker container.

The interview in Appendix B.1 shows that larger projects tend to provision Docker for peak load thus increasing the actual monthly cost for Docker deployments, the interview also states that performance is more important than costs at large companies.

The cost analysis of serverless functions shows a major increase in cost for AWS Lambda due to the extra cost of the API Gateway. This extra cost makes AWS Lambda less cost effective when dealing with load over 50 requests per second. To put this in perspective Twitter has about 6000 tweets per second[14] and Google.com handles on average 40.000 searches per second[15]. Removing the API Gateway from AWS Lambda removes the possibility to deploy with a HTTP endpoint at this time.

If a web service would be deployed globally in Docker containers and provisioned for peak load using two containers per region that would increase the deployment price presented in Table 4.1 fourteen times. Therefore when analyzing the price of Docker containers one must always consider the intended market of the application to account for extra instances in different regions and provisioning extra instances to handle regular peak load if auto-scaling is unavailable.

6 Discussion

To answer the question of which deployment method is better suited for deploying an application in the cloud my results show that it all depends on the requirements of said application. Serverless deployment has a higher response time while maintaining a lower cost under low load when compared to Docker containers.

Serverless applications are best suited when there is a need for global auto scaling while experiencing low or sporadic traffic. There are many use cases that fit this description like a login service, a checkout system for an e-commerce site etc. These are services that need to be highly available thus making them a perfect fit for serverless but they are also just a part of a bigger system. I would not recommend implementing an ecommerce site solely as serverless based on my cost analysis. Serverless is harder to deploy and test due to its cloud only limitation and natural microservice design.

Docker applications have a lower response time but when deployed across multiple regions the extra VMs increase the cost. The main advantage of Docker is that it is a standardized system that can be deployed on any platform and most operating systems thus making it highly flexible. A web service can be deployed in the cloud and on premise while running the exact same application. The development environment is also identical to the production environment thus making both testing and deploying much easier.

The best deployment method for most applications will most likely be a service deployed as microservices. Each service can then be deployed in the most suitable way to meet its specific requirements.

Cost and performance are important factors but complexity is also something that needs to be considered when designing an application. The interviews in Appendix B have shown that serverless applications are more complex due to them being natural microservices. This increases the complexity of the design and the natural flow of events of the application. There is a difference between programmers that makes working with serverless difficult. There are programmers that work mostly with maintaining existing applications. This long term work on one system makes them less flexible and usually means that they have limited experience working in the cloud. The mindset of a maintenance programmer is to polish an existing application and fixing issues. On the other hand there are programmers that regularly work with new projects and are used to working

with new technology and techniques thus making them more flexible. The mindset of these programmers is to deliver the application as fast as possible.

Considering the different roles programmers fall into an architect must consider the long maintenance period of the application and try to reduce the complexity to ensure a longer lifetime.

The previous research mentioned in Chapter 1.2 had serverless functions as their focus while this experiment focused on a comparison between serverless functions and dockerized applications. The previous articles do however draw parallels between serverless and reserved instances and their conclusions match the experiments results. Both this experiment and the previous articles conclude that serverless is indeed cheaper for use cases with periodic load and short execution times.

7 Conclusion

This report presented an experiment where two different deployment methods were compared based on cost and performance. The experiment has shown that there is no clear winner. The deployment methods excel at different things. Docker containers excel at performance and are most suitable for applications with long execution times. Serverless functions excel where applications have unpredictable load, short execution times and offer high availability.

The best deployment method is most likely both. An application can be deployed as microservices and the best suited deployment method can be chosen for each microservice thus getting the benefit of both methods. This shows that architects should deploy applications as microservices and deploy each microservice based on its specific needs and requirements.

The results could have been extended with data where Docker reaches its VMs limit and starts dropping requests. An attempt was made to achieve this with the test that had 1000 requests per second but it was unsuccessful in doing so.

The web application could have had a built in delay to simulate execution time or a number of built in delays to gather better data regarding response times given different execution times.

The results are helpful for architects that intend to design an applications deployment in the cloud.

7.1 Future work

Given the results the natural progression of this experiment would be to deploy a fully functioning REST API with one or two databases in three ways: only serverless, only kubernetes with functioning autoscaling and as a mix where the application is split between kubernetes and serverless functions. This test would provide much better data on the pros and cons of the two deployment methods and offer data on mixed deployment as well.

Secondly future work can use the base price difference found during this experiment and analyze the other costs incurred during the development process. The different methods have different costs for: development, maintenance, external services needed to achieve the requirements etc.

References

[1] M. J. Fork, “What is infrastructure as a service (IaaS)? - Cloud computing news,” *Cloud computing news*, 14-Feb-2014. [Online]. Available: https://www.ibm.com/blogs/cloud-computing/2014/02/what-is-infrastructure-as-a-service-iaas/. [Accessed: 25-Feb-2018].

[2] “Networking Products – Amazon Web Services (AWS),” *Amazon Web Services, Inc.* [Online]. Available: https://aws.amazon.com/products/networking/. [Accessed: 23-Mar-2018].

[3] “Cloud SQL - MySQL & PostgreSQL Relational Database Service | Google Cloud,” *Google Cloud*. [Online]. Available: https://cloud.google.com/sql/. [Accessed: 23-Mar-2018].

[4] “Amazon Simple Queue Service (SQS) | Message Queuing for Messaging Applications | AWS,” *Amazon Web Services, Inc.* [Online]. Available: https://aws.amazon.com/sqs/. [Accessed: 23-Mar-2018].

[5] “Website.” [Online]. Available: https://serverless.com/framework/docs/providers/aws/guide/intro/https://serverless.com/framework/docs/providers/aws/guide/intro/. [Accessed: 23-Mar-2018].

[6] “Amazon EC2,” *Amazon Web Services, Inc.* [Online]. Available: https://aws.amazon.com/ec2/. [Accessed: 23-Mar-2018].

[7] R. Gancarz, “The economics of serverless computing: A real-world test | TechBeacon,” *TechBeacon*, 23-Mar-2017. [Online]. Available: https://techbeacon.com/economics-serverless-computing-real-world-test. [Accessed: 18-Mar-2018].

[8] “Be Wary of the Economics of ‘Serverless’ Cloud Computing - IEEE Journals & Magazine.” [Online]. Available: http://ieeexplore.ieee.org. [Accessed: 25-Mar-2018].

[9] A. Eivy, “Be Wary of the Economics of ‘Serverless’ Cloud Computing - IEEE Journals & Magazine.” [Online]. Available: http://ieeexplore.ieee.org. [Accessed: 25-Mar-2018].

[10] “Reserved, on demand or serverless: Model-based simulations for cloud budget planning - IEEE Conference Publication.” [Online]. Available: http://ieeexplore.ieee.org. [Accessed: 25-Mar-2018].

[11] “AWS Lambda – Serverless Compute - Amazon Web Services,” *Amazon Web Services, Inc.* [Online]. Available: https://aws.amazon.com/lambda/. [Accessed: 25-Feb-2018].

[12] “Docker Basics for Amazon ECS - Amazon Elastic Container Service.” [Online]. Available: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/docker-basics.html. [Accessed: 25-Feb-2018].

[13] “Improve Server Response Time | PageSpeed Insights | Google Developers,” *Google Developers*. [Online]. Available: https://developers.google.com/speed/docs/insights/Server. [Accessed: 07-May-2018].

[14] “Twitter Usage Statistics - Internet Live Stats.” [Online]. Available:

http://www.internetlivestats.com/twitter-statistics/. [Accessed: 29-Apr-2018].
[15] “Google Search Statistics - Internet Live Stats.” [Online]. Available: http://www.internetlivestats.com/google-search-statistics/. [Accessed: 29-Apr-2018].

A Appendix 1 - Test result

The following tables show the test results from the seven load tests conducted during the experiment.

Label Name	**Avg. Response Time** (ms)	**90% line** (ms)	**99% line** (ms)	**Min Response Time** (ms)	**Max Response Time** (ms)
ALL	116.88	190	298	33	2524
AWS Docker	44.13	48	54	42	66
AWS Lambda	90.92	99	160	75	508
Azure Docker	104.4	128	184	33	292
Azure Function	232.46	225	1988	76	2524

Table A.1: 1 minute delay test results

Label Name	**Avg. Response Time** (ms)	**90% line** (ms)	**99% line** (ms)	**Min Response Time** (ms)	**Max Response Time** (ms)
ALL	108.824	190	267	33	3328
AWS Docker	46.569	48	49	43	3046
AWS Lambda	91.737	93	138	74	3328
Azure Docker	103.843	126	141	33	323
Azure Function	193.753	207	371	75	2054

Table A.2: 2 second delay test results

Label Name	**Avg. Response Time** (ms)	**90% line** (ms)	**99% line** (ms)	**Min Response Time** (ms)	**Max Response Time** (ms)
ALL	106.84	190	273	32	3103
AWS Docker	45.56	47	51	44	3047
AWS Lambda	87.09	91	138	70	3103
Azure Docker	102.74	124	138	32	1248
Azure Function	192.12	204	389	75	2582

Table A.3: Constant load test results

Label Name	**Avg. Response Time** (ms)	**90% line** (ms)	**99% line** (ms)	**Min Response Time** (ms)	**Max Response Time** (ms)
ALL	105.29	191	243	28	3302
AWS Docker	45.36	49	50	28	62
AWS Lambda	88.16	92	141	35	3302
Azure Docker	93.28	123	135	32	236
Azure Function	194.42	203	347	74	2538

Table A.4: Low load test results

Label Name	**Avg. Response Time** (ms)	**90% line** (ms)	**99% line** (ms)	**Min Response Time** (ms)	**Max Response Time** (ms)
ALL	105.41	186	238	32	2354
AWS Docker	43.54	47	48	42	57
AWS Lambda	89.85	94	144	72	2354

Azure Docker	102.12	124	138	32	659
Azure Function	186.42	199	311	72	2168

Table A.5: Peak load test results

Label Name	**Avg. Response Time** (ms)	**90% line** (ms)	**99% line** (ms)	**Min Response Time** (ms)	**Max Response Time** (ms)
ALL	186.5	206.5	281	490	2563
AWS Docker	44	48	74	42	77
AWS Lambda	84	90	136	78	2269
Azure Docker	98	99	131	92	185
Azure Function	520	589	783	490	2563

Table A.6: Medium load test results

Label Name	**Avg. Response Time** (ms)	**90% line** (ms)	**99 % line** (ms)	**Min Response Time** (ms)	**Max Response Time** (ms)
ALL	188.25	201.75	264	41	2873
AWS Docker	46	50	105	41	161
AWS Lambda	87	104	124	80	2193
Azure Docker	98	99	121	93	142
Azure Function	522	554	707	497	2873

Table A.7 High load test result

B Appendix - Interviews

B.1 Interview Jayway Project lead

1. Do you have experience deploying applications to one of the major cloud platforms ?

Yes. I have exclusively worked with AWS in two separate projects.

The first project was a migration of a on premise setup to Amazon. Amazon was the obvious choice at the time due to them always being ahead of their competitors. Obviously when migrating an old on premise solution the existing structure is a limiting factor thus it ended up being a lot of virtual machines deployed on Amazon to avoid rebuilding much of the existing structure.

The second project which is my current project is a brand new system built from scratch. Deploying the service to Amazon was a requirement from the customer who have adopted a cloud first approach. Working from scratch is very different than migrating an existing system you have much more freedom and you can tailor your system to the managed systems on offer at Amazon.

2. Are you familiar with Serverless Functions and Docker containers ?

The project I am currently working on is deployed as a Docker cluster using AWS ECS. I have limited experience with serverless functions but I know what they are and how they work.

3. Have you recently taken part in the planning phase of a new application ?

Yes. The project I am currently working on is one built from scratch with the requirements presented by the customer.

   a. Did you choose to deploy to the cloud ?
      Yes.
   b. Why ?
      The customers has a cloud first policy.
   c. Can you give a broad description of the application ? ie 2 APIs one Default Gateway and 1 Database
      Load balancer, APIs and a number of databases. We are considering adding queuing and caching as well.

4. Do you consider a deployment method to be permanent ?
   a. If yes why

b. If No how do you plan to develop an application without a specific deployment method in mind

No. I believe that projects need to be more agile with their deployment and configurations and not only the software implementation.
My current project is a great example. We currently use ECS that means that we need to manage the VMs actually running our software ourselves. We are probably going to go over to use AWS Fargate which is a managed service meaning that we only need to worry about our own code and not the underlying infrastructure.

‘ c. In your experience is it easier to change your infrasture in the cloud then on premise ?

Yes I believe it is. Amazon in particular does a great job with their Managed services. They will manage all the infrastructure and configuration needed to run a tool while the user can only use it with their software without needing to worry about setup, configuration and maintenance. AWS Aurora is a great example. Aurora is a fully managed database solution. All the user needs to do is place an order and in 5 minutes you have a working database all you need to do then is to add a schema to it. You can't really do that on premise. Someone will need to have the knowledge of how to set everything up so that the project can later use it.

5. Do you consider load an important factor when choosing your application stack ?
    a. How far ahead do you calculate probable load

Yes. I believe it is important to design your infrastructure for expected load from the beginning. We are already designing our infrastructure in a way that it is scalable and for it to have the ability to be deployed in multiple regions around the globe since our customer is active on the global market.
Historically not considering load when designing your infrastructure will most likely result in difficulties in handling high load speaking from my own experience.

6. Is High availability important for you ? Or your application ?

Yes. High availability and performance are the cornerstones for my current project. The customer has high expectations on the services uptime. High availability has been an important factor when planning our infrastructure. When deploying our software the core functionality will be deployed across multiple availability zones and multiple regions thus providing fallback if any one zone goes down. Half of Amazon's Data Centers would need to go down before our service would go down due to load which is very unlikely.

7. What are your thoughts on Serverless vs Docker Containers

I personally do not believe serverless to be a good option at this time. Your

application ends up being locked down into serverless. Lets take an API for example implemented as serverless. Half of your logic i.e routing will end up being implemented in a AWS API Gateway which you can not easily move and your API logic will rely on Amazon events which is again something you can not simply remove. I do not like being locked down into one specific infrastructure. On the other hand you receive a lot for free with serverless functions when it comes to autoscaling out of the box which is helpful.
I am also worried about actually designing a complex system using only functions. How do you keep track of them ? How do you deploy updates ? Version management. I have little professional experience with this but I see potential problems if a project has and relies upon multiple functions due to the extra complexity that comes with serverless functions.

Docker on the other hand has become a industry standard and is well defined. The main advantage is that you can move them around freely and your testing environment is a exact replica of your production environment which is a great help when performing tests. Using services like Kubernetes and AWS Fargate will also mitigate the amount of maintenance required by the underlying infrastructure and allow for reliable auto-scaling.

In my current project we rely solely on Docker due to a number of reasons but one of them is that the customer wants to be able to deploy the service on-premise in their own data centers if needed for special markets.

8. When deploying an application do you provision for peak load or implement auto scaling ?

In the past I have provisioned for peak load. We always consider auto-scaling but we never get enough time to actually implement it so it falls by the sidelines. It really depends on the project, my current project will be deployed globally in multiple regions so provisioning for peak load and letting it run is not an option, it would cost to much. Our client has a deployment process which includes 5 environments which would also cost a lot of money to keep going considering that test environments idle more than they are used.

I think that one should always implement auto scaling but it is no easy task to do. It's not enough to simply add more instances and hope for the best your hole application stack needs to be able to scale both up and down as needed and defining the rules to achieve this is very difficult.

B.2 Interview Jayway CTO

1. Do you have experience deploying applications to one of the major cloud platforms ?

Yes I have deployed application in the past. Usually however I do not get to deploy application given my role in the company but I do know how to do it just never really get the chance to do it anymore.

2. Are you familiar with Serverless Functions and Docker containers ?

Yes. The first time I had to opportunity to deploy to the cloud was in late 2007. A lot has changed since that time given that back then all you had was VMs with a temporary storage, an external permanent storage solution and a queue system. Docker and Serverless are just one of many iteration on how to deploy to the cloud and I believe we will see new ways in the future.

Docker has the major advantage of being standardized thus making it portable across platforms without much issues. Serverless functions are platform specific thus making them hard to move.

3. Have you recently taken part in the planning phase of a new application ?

   Yes

   a. Did you choose to deploy to the cloud ?

      Yes

   b. Why ?

Customers today think it is obvious to deploy to the cloud and many employ a cloud first strategy. There is a second reason for this as well. Running applications on premise has become more expensive in the last few years due to a difficulty to find the right people to manage it and an increase in cost for those people thus making it less profitable in a world where standardized product and low prices are required to compete.

There are several advantages to deploying in the cloud. An architect is no longer bound by the rules and possibilities found in the on premise data center. They can build an infrastructure solution that fits for the specific application instead of adapting the application to run on pre existing infrastructure.

The open competition between the different cloud providers only encourages them to develop better services and to offer even better prices for their customers making on premise solutions slow and expensive in comparison.

The managed services provide functionality needed by many projects but without the maintenance. AWS offers managed databases, Docker clusters, API Gateways, serverless functions and much more.

    c. Can you give a broad description of the application ? ie 2 APIs one Default Gateway and 1 Database

Node/Express web servers and a database, there is another project that is being deployed fully as serverless functions and many more.

4. Do you consider a deployment method to be permanent ?
    a. If yes why
    b. If No how do you plan to develop an application without a specific deployment method in mind

Definitely no. Things change so fast that projects need to be flexible to meet new requirements or to seize new opportunities. It is good to have a target environment in mind when designing your application but one must be able to adapt to new services and requirements. An application that has an expected lifetime of 10 years will most certainly be deployed to a new platform during its lifetime. This is where Docker really shines since it is a standardized system that can easily be moved to a new environment or deployed as a cluster using Kubernetes, AWS Fargate or any other service.

5. Do you consider load an important factor when choosing your application stack ?

Yes, always. One should consider the expected load before committing to a solution. Maybe that the expected load is very low then designing with low response time in mind is not so important. It is important to understand the scope of the project and the intended audience for the system in order to make calculated decisions on how to prioritize performance.

    a. How far ahead do you calculate probable load

Non functional requirements like probable load always come in late in a projects lifetime. Non functional requirements also tend to be misunderstood. When asked, many clients tend to say that they want 99.9% uptime even do they don't need it. In many cases a uptime of 99.5% is more than enough to meet the clients demands and also allows for a service window where the application can be simply shutdown for an hour a month which greatly reduces the complexity of the deployed application.

Even tho serverless functions have a lot of functionality built in they are still very complex to grasp and to deploy in a production environment. Just the task to deploy all functions in the right order and with all their dependencies is a major undertaking and there is no easy way to test them or to validate the deployed functions. One should not view serverless or Docker in a vacuum however. Serverless will probably be cheaper to run but will it be worth it given the increased development cost due to the added complexity and difficulty of developing an application as serverless functions? Probably yes but it is hard to say. Even if the extra development cost is worth it how do you maintain such a system?

Planning for load is often overlooked during development it comes in towards the end of the development process. Also many projects do not continue running the load tests during the lifetime of the application to validate that the application still meets the original requirements.

6. Is High availability important for you ? Or your application ?

Applications deployed in the cloud often have an expected availability of 100%. This should however not be misunderstood. Very few applications actually need 24/7 availability. Having a service window once a week or once a month of one or two hours will in most cases not have a negative effect on the service if done right. It is easier to achieve high availability in the cloud if the application was designed with that in mind.

7. What are your thoughts on Serverless vs Docker Containers ?

I remember the Amazon conference when serverless functions were announced and it was so impressive to see such a clean solution with independent functions working together. When considering serverless in reality for a major client active on a global market serverless falls short due to a few reasons. The application can be built but then it will be handed over to the client which in turn needs to maintain it for years to come this is much harder to do at this point in time than for example Docker due to the complexity in a fully serverless system. I am yet to be convinced that serverless is feasible for a large scale project. Serverless does win points for being new and interesting making it easier to attract developers to such projects.

Docker is now an industry standard which makes deployment and handover to the client a smooth process.

8. When deploying an application do you provision for peak load or implement auto scaling ?

In the cloud we always provision with auto scaling but this is easier said than done. The problem is often not scaling up your application stack but scaling it down again once load has decreased. It is very difficult to implement the required rule set for autoscaling. If we compare to running on premise even if you implemented auto scaling the application can only scale to the size of available physical servers which leads to projects provisioning for peak load on premise to ensure that they have enough compute power when they need it. Sometimes projects would take over servers from lower prioritized systems in order to handle special days like black friday. This could happen when running on major cloud providers but the chance of that happening is very small.

9. Extra talk about complexity:

An average developer with little to no knowledge about the cloud will have no problems understanding what Docker is and how it works. Kubernetes on the other hand would be considered a kind of magic black box that does all these cool things with auto scaling, cluster management, networking etc. Serverless would end up being incomprehensible since it is such an abstract concept. Just imagine having to build a website using serverless functions. Every page is an independent entity totally decoupled from the rest of the website. It would require a lot of extra effort to build such a website compared to how easy it is to build the same website with Node/Express, React or AngularJs. It will take longer to develop, harder to maintain and in the end if the website does not serve hundreds of millions of requests per month the actual saving on infrastructure is about 10$. On the other hand bigger companies often have at least 3-4 different environments that are up and running 24/7. There is a development environment, Integration testing environment, Pre production environment, production environment, training environment etc. In this case serverless is pure gold since production is really the only environment that is actually in use the rest of them idle more than they are being used.

There is no way to write a general formula to calculate to total cost of ownership since it is very specific to the company and said application. Therefore it is much better to analyze the difference in cost between Docker and serverless regardless of platform and application. The first step in analyzing the potential costs of the two different deployment methods is knowing the base price difference which is then used as a pivoting point for the rest of the analysis. If serverless is half the price of Docker then the analysis can continue by analyzing the other costs incurred by that deployment method. Costs being: development, maintenance, other services needed to achieve the requirements etc.

The main focus of many companies today is time to market. The price tag during the development of a application has taken a backseat priority wise. Companies are willing to pay more if they can decrease time to market thus making small saving on infrastructure costs a very low priority. Once the application is finished reducing monthly costs becomes a higher priority.

Two great use cases for serverless is a search function or a checkout function that is part of an existing website. The website becomes more lightweight and the crucial functionality like search and buy get moved to serverless thus providing the scaling and availability that is expected of such services.