Spring 2019

A user-friendly and cost-effective system for automatically configuring AWS EC2 instances for web servers

Xiaochen Yang
Iowa State University

A user-friendly and cost-effective system for automatically configuring AWS EC2 instances for web servers

Abstract

Cloud computing has becoming more popular and mature in the recent years. However, the migration from on premise to cloud platform is still considerably difficult for many organizations. For instance, a simple task to create a virtual machine instance within a cloud platform involves many steps and in-depth of understanding of the cloud infrastructure. It is a huge burden to learn the hundreds of concepts about cloud computing. One of the major parts of a business is to develop and maintain the company website. We consider the easiness of creating or managing cloud resources hugely affects the migration from on premise to cloud platform. Therefore, we attempt to design a user-friendly and cost effective platform. Our goal is to help organizations take advantages of cloud computing to deploy websites without spending much time and money on researching how to build web server using cloud resources. In our current platform, a user only needs to provide budget and machine images. Furthermore, the platform continually monitors the usage of cloud resources and dynamically adjusts the consumption of cloud resources. Compared to the common strategy of building web servers on cloud, our platform is able to reduce the cost and increase the performance with collecting minimal inputs from users.

Background

Cloud computing is the delivery of computing services, including computing power, storage, database, network, and more, over the internet. Due to the unique pay-as-you-go model, the cloud computing provides flexible and scalable computing resources in a cost effective and infrastructure-independent manner [1]. Therefore, cloud computing can be an efficient alternative to owning and maintaining computer resources and applications for many organizations, especially the small- or medium-sized organizations [1]. Many organizations recognized the benefits of cloud computing and started to migrate to cloud from premise. By year 2020, it is expected that more than one trillion dollar of revenue will be generated directly or indirectly through cloud computing systems [2]. Although the cloud computing paradigm has becoming more and more popular, the adoption is still considerably difficult. Many companies that are providing cloud-based services have dedicated teams for building and maintaining the infrastructure using cloud resources. However, such dedicated infrastructure teams is an heavy investment and could be huge overhead for a small- or medium-sized company.

The primary need of cloud services for most small-or medium-sized companies is to host their web applications. The most flexible and cheapest way to build infrastructure for a web application is using the bare-metal level service, infrastructure as a service (Iaas), provided by cloud providers. The users need to configure the web application from the basic virtual private cloud (VPC) to the actual deployment of the application. There are tens of concepts and different choices even for deploying a simple web application on cloud. It might require a dedicated team or personnel working on the building and maintaining this infrastructure, which is a large investment for a relative small part of a business. Therefore, the burden of learning and building infrastructure for hosting a web application put a huge barrier for small businesses from using cloud computing services.

Many cloud providers have services for users that simplify the process of hosting a web application in the cloud. For example, AWS's Elastic Beanstalk is a service for users to deploy, run, and manage web applications [3]. The process of hosting a website using Elastic Beanstalk is very simple. The users just need to upload their code and files to an S3 bucket (an AWS storage service). The Elastic Beanstalk handles the infrastructure configuration and deployment of the application. The Elastic Beanstalk also manages the application in many aspects, such as capacity, provisioning, load balancing, auto-scaling, and health monitoring [3]. The Elastic Beanstalk is an easy-to-use service. However, the architecture of Elastic Beanstalk is fixed by AWS and this may not suitable for all use cases. In addition, the price of the Elastic Beanstalk is relatively high compared to the strategy of building insfrastructure using Iaas. Chieu et al. proposed a novel architecture for dynamically scaling web applications using cloud resources [4]. The authors developed an approach with front-end load-balancer and dynamically adjust cloud resources based on active sessions [4]. This architecture is very similar to AWS's Elastic Beanstalk. The authors derived a detailed algorithm for dynamic scaling instances and the solution is on IaaS level which potentially has lower cost and more control of the server configurations compared to Elastic Beanstalk. However, This paper only focused on the scaling of web application and response to sudden change in requests, but not on costs or suitability for different use cases.

A web application may need to process network-heavy requests (like viewing webpage and downloading files) or computation-heavy requests (like encryption and decryption) or both. Selecting instance types that matching the use case is a major challenge when configuring a web server in cloud. There are many types of instances with different properties and prices, including computation optimized, network optimized, and memory optimized. To achieve high performance, careful selection of instance type plays significant role. However, such selection is not easy, since it requires the users carefully researching the differences of instance types and taking into consideration of cost versus performance. Davatz et al. provided an approach to select instances for web servers hosted on cloud and also proposed a method to benchmark the performance and cost of multi-tier web applications hosted on IaaS cloud. They performed a case study using AWS and Google cloud platform. Their results showed that carefully choose machine type for a web application can significantly reduce the cost of cloud resources [5]. However, to be able to choose an optimal machine type, it involves in-depth of knowledge of the web application, network traffic pattern, and the cloud resources. In most organization, such talents may not be available and therefore put another burrier for many businesses from migrating to cloud if the cost is their top priority.

One of the most important aspect of web application performance is the latency of requests. The traditional strategy to meet high quality of service is by auto-scaling of instances, such as Elastic Beanstalk. However, auto-scaling strategy can introduce high cost by using many instances to achieve low latency. Iqbal et al. developed a strategy to reduce response time and minimizing resource utilization in cloud by finding the bottle neck of a web application and automatically resolving it. The bottle neck detector uses the profiling of the CPU, memory, and I/O resource usage with a combination of real-time and historical log data. The preliminary results showed that this strategy significantly reduce the latency with minimal usage of cloud resources [6]. This research is focused on resolving bottle neck caused by different tiers of a web application. Within each tier, they did not optimize the usage of cloud resources. This research provides an potential method to smartly organizing cloud resources. With the assistance of the real-time and historical log data, we might be able to find the traffic pattern and choose the right cloud resources to further optimize the cloud resources within each tier.

Based on the investigation of AWS services, for a typical web application, the monthly cost table is shown as below. As shown in the table the two major costs are EC2 instances and the RDS service. The most expensive and most customizable component in this table is the EC2 instances. Therefore, in our research, we focused on reducing the cost of EC2 instances while maintaining high performance. AWS provides many instance types which can be computation-optimized, network-optimized, or memory-optimized instances. Different types of instances have very different properties and capacities. Elastic Beanstalk or other models usually try to find the best fit instance type and only use one type of instance. As mentioned above, such a selection is difficult and sometimes might not be available. For example, a t2.micro instance can be twice as powerful as an a1.medium instance in terms of computation, but the bandwidth of t2.micro is one-tenth of the bandwidth of a1.medium. For web applications that needs to handle both computation and communication requests, it might not achieve good performance with just t2.micro or a1.medium. However, by carefully adjusting the percentage of these two instance types, it is possible to achieve lower cost and better performance at the same time.

	Cost for a month ($)
Elastic load balancer	18.30
EC2	408.04
S3	0.78
Route 53	0.90
CloudFront	33.37
RDS service	336.72
DynamoDB	80.89
AWS data transfer in/out	42.21

In this research, we aim to build an easy-to-use and cost-effective platform for building web servers using cloud resources. In addition, we put the performance as one of the top considerations in our design. We designed a system that collecting minimal information from users and automatically build the infrastructure of web servers. The system builds the infrastructure using multiple machine types (network-optimized and computation-optimized machine). By continually monitoring the usage of cloud resources (CPU and network), we identify the bottle neck of the web servers and adjust usage of different machines to meet the current workload with high performance and low cost. In this research, we used AWS as an example for the proof of concept, but our platform is not relied cloud provider and is possible to be used in hybrid cloud to even reduce the cost. We chose computation-optimized t2.micro and network-optimized a1.medium machine types for the experiments. The system collects CPU utilization and network throughput by querying AWS services every five minutes. Based on the historical and current data, we calculated the optimal ratio of t2.micro and a1.medium. Our results showed that our platform is able to achieve better performance with lower cost compared to auto-scaling with one machine type or two machine types.

Proposed System

Our system is aimed to reduce the overhead of learning many concepts of cloud services and researching for the best strategies for building web servers. Our targeted users are small- or medium-sized businesses. We consider that primary goal of the small- or medium-sized businesses is not to build the most efficient and cost effective web servers by themselves. A simple to use and cost-effective system would increase their tendency for migrating to cloud computing services. In this way, we can save cost for users and increase the revenue of cloud providers, which is a win-win situation for both users and cloud service providers. After careful investigation, in our current design, we only need the users to provide their machine images and a budget. Our system automatically create the infrastructure and start to monitor and manage the usage of cloud resources. Essentially, the only work users need to do is to build the machine images and they do not need to know the many concepts of the cloud resources, such as VPC, security groups and more. Therefore, we reduce the learning overhead for users down to how to build a machine image in cloud.

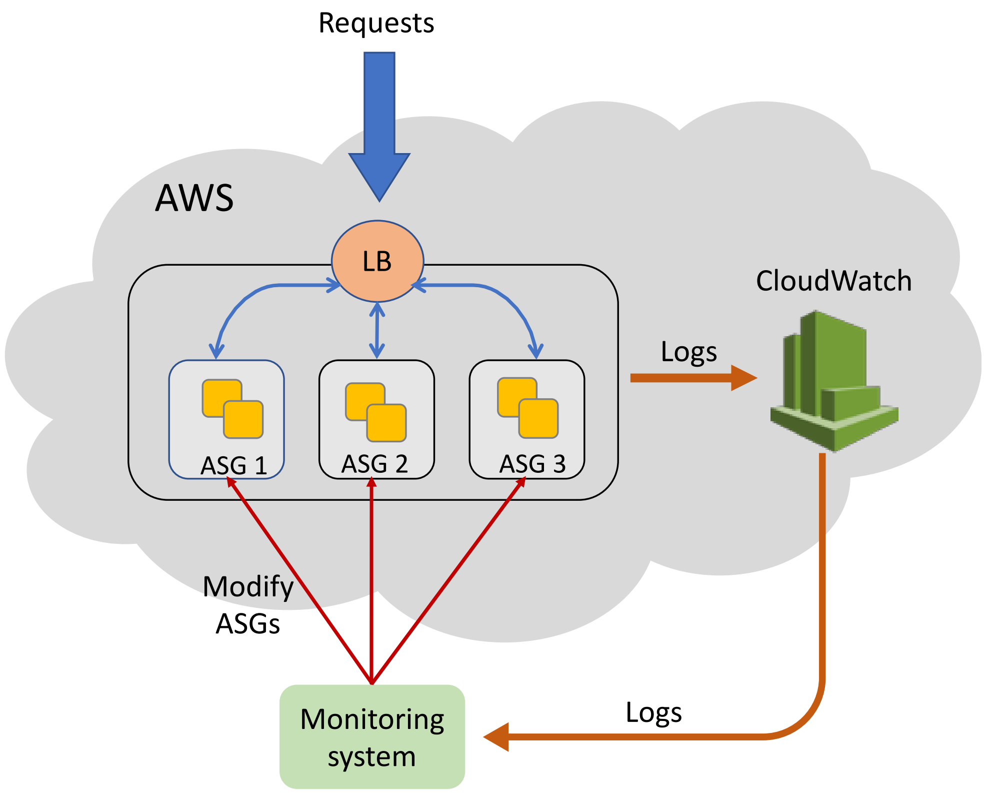

Figure 1. Diagram of proposed system.

Our proposed system is summarized and shown in Figure 1. The system built one virtual private cloud (VPC) to host all the cloud resources. The system creates proper security groups for VPC, which provide security for the VPC. In this VPC, the system creates and configures three subnets, one internet gateway, and the associated route tables. The system creates one load balancer (LB) which receives requests and passes them to the running instances in a default manner (round-robin). We create auto-scaling groups (ASG) for each type of instance provided by users and the instance number of each type is controlled by the designed capacity of ASG. All the ASGs are connected to the load balancer. The system enables the AWS CloudWatch service to collect detailed logs from ASGs, which include CPU utilization and network throughput. Our monitoring system sends queries and acquires the logs from CloudWatch in every 5 minutes and calculates the optimal number of instances for each ASG based on the historical and current workload. In the current implementation of the system, we only use the AWS services as an example to show the proof-of-concept and therefore the certain definitions or service names are only available in AWS. However, based on our investigation of services provided by other cloud providers, we can find very similar services. Therefore, our system is potentially easy to include other cloud providers.

We consider that the historical and current information of cloud usage should weight differently in the calculation of the optimal number of instances. We used an exponential average function to calculate the CPU utilization and network throughput, which take into consideration of both historical and current data. Taking CPU utilization as an example:

$$avg\_CpuUsage = previous\_CpuUsage \times \alpha + current\_CpuUsage * (1 - \alpha)$$

In the exponential average function, α is the weight of previous average CPU utilization or network throughput. In our monitoring system, we assign α to be 0.7 which means we care more about current workload, because the current workload may be a sudden burst of network queries or computation queries that we need to compensate. $previous\_CpuUsage$ is the calculated exponential average of CPU utilization in previous round (5 minutes window). $current\_CpuUsage$ is the CPU utilization from current round (5 minutes window) collected from CloudWatch logs.

As shown above in the monthly cost table of a web application, the major part is from the cost of instances. Therefore, we simplified the total cost to be the sum of cost of all instances. The cost calculation algorithm takes the current CPU utilization, network usage, the price of each instance type, and the budget provided by user as inputs. Within the budget boundary, the system needs to calculate instance number of each machine type that provide enough computation power and network bandwidth with the lowest cost. We defined this problem as an optimization problem as shown below. $price_i$ is the price of instance i. m_i is the number of instance i and is an integer between 1 and n. n is the total number of instance types. The output of this optimization algorithm is the number of each instance type. If the current numbers of each instance type are different from the optimal numbers, the system sends requests to AWS to modify the auto-scaling group sizes. Since the instance number of each instance type is an integer, the complexity of this algorithm is O (n^2) in the worst case.

$$totalCost = minimize \sum_{i=1}^{n} price_i \times m_i ,$$

$subject\ to$:

$$totalCost < budget,$$

$$\sum_{i=1}^{n} bandWidth_i \times m_i \geq avg\_NetworkUsage,$$

$$\sum_{i=1}^{n} Cpu_i \times m_i \geq \text{avg\_CpuUsage},$$

$$\text{m} \in \{1, 2, \dots, \text{n}\}$$

Experimental Design

After researching the AWS EC2 instance types, we decided to use t2.micro and a1.medium. The t2.micro instance type has high computation power and low bandwidth and the a1.medium instance type has low computation power and high bandwidth. In order to perform experiments to test our proposed system, we first designed and implemented a simple web application. The web application can take four types of queries. For query 0, the web server generates and sends a string with length of 25,000 Bytes back to client, which is used as the network-intensive query and needs very little computation. For the computational intensive queries, we designed three queries that requires three level of computation. Three level of computation requirements provide more dynamics for our tests. For query 1, the web server generates a string with length of 25,000 Bytes and encrypts the string. Finally, the server sends the hash of encryption to client which is very short and needs very little network traffic. Query 2 and query 3 are similar as query 1, with string lengths 100,000 and 500,000 Bytes respectively. As shown in the table below, these queries requires different processing time when we tested them against a t2.micro instance or a1.medium instance, which indicate different levels of computation requirements.

	Response time (s)	
	t2.micro	a1.medium
query 0	0.00676	0.0148
query 1	0.0189	0.0453
query 2	0.0761	0.1792
query 3	0.3773	0.8949

We then build two amazon machine images (AMI), a t2.micro AMI and an a1.medium AMI, to host our simple web application. As shown in the table below, the t2.micro instance has high computing power and low price, but very low bandwidth. The a1.medium instance has very high bandwidth, but relative high price and low computing power [8]. Based on processing time shown in the table above, we can also tell that the t2.micro instances have more computation power compared a1.medium instances. In our experiments, we use t2.micro as computation-optimized machine type and a1.medium as network-optimized machine type.

	Bandwidth (Mbps)	CPU clock speed (GHz)	# of cores	Price ($/hour)
t2.micro	70	3.3	1	0.0138
a1.medium	7000	2.0	1	0.0225

To test our system, we implemented a program that automatically sends queries to the web servers. This program is able to send different percentages of queries in a random manner. This is achieved by generating random numbers between 1 and 100 and limits certain numbers to send one type of queries. We can also control the frequency of queries using multithreading program. An important part of the testing is the latency of queries. Latency is one of the key performance measures of a web application. We also record the response time for this query and write it to a file, which is then used to analyze the response time and performance of the system.

Results

User-friendly system

We started off with the goal to build a very simple system for users to build their web server infrastructure. After carefully investigating the AWS services and researching the strategies for building web servers, we found many common services we need to build for a web server no matter big or small web server, such as VPC, subnets, and internet gateways. The virtual machines are the unique parts for each server. Therefore, we built our system that only takes inputs that are specific to their web server and then build the entire infrastructure automatically.

As shown in Figure 1, the system takes the AMI id and budget as the inputs, which are the only inputs we need from users. The system configures the networks, load balancer, necessary security, auto-scaling groups, and connections between all cloud resources. Without our system, users have to learn all these services and build them by themselves. Furthermore, for the purpose of proof of concept, our system is relatively simple and uses a small number of cloud resources. With larger and more complicated systems, the involved cloud services will significantly increase and the connections between them become more sophisticated, which requires very deep understand of the cloud services and significant amount efforts for maintenance. With our system, the learning task is cut down to how to prepare virtual machine images, which is also required for premise system. We assume preparation of virtual machine images is a relative familiar task for most IT managers and developers. Therefore, we believe, with this user-friendly system, more users are likely to migrate from premise system to cloud. This can save money for the users and increase the profits of cloud providers.

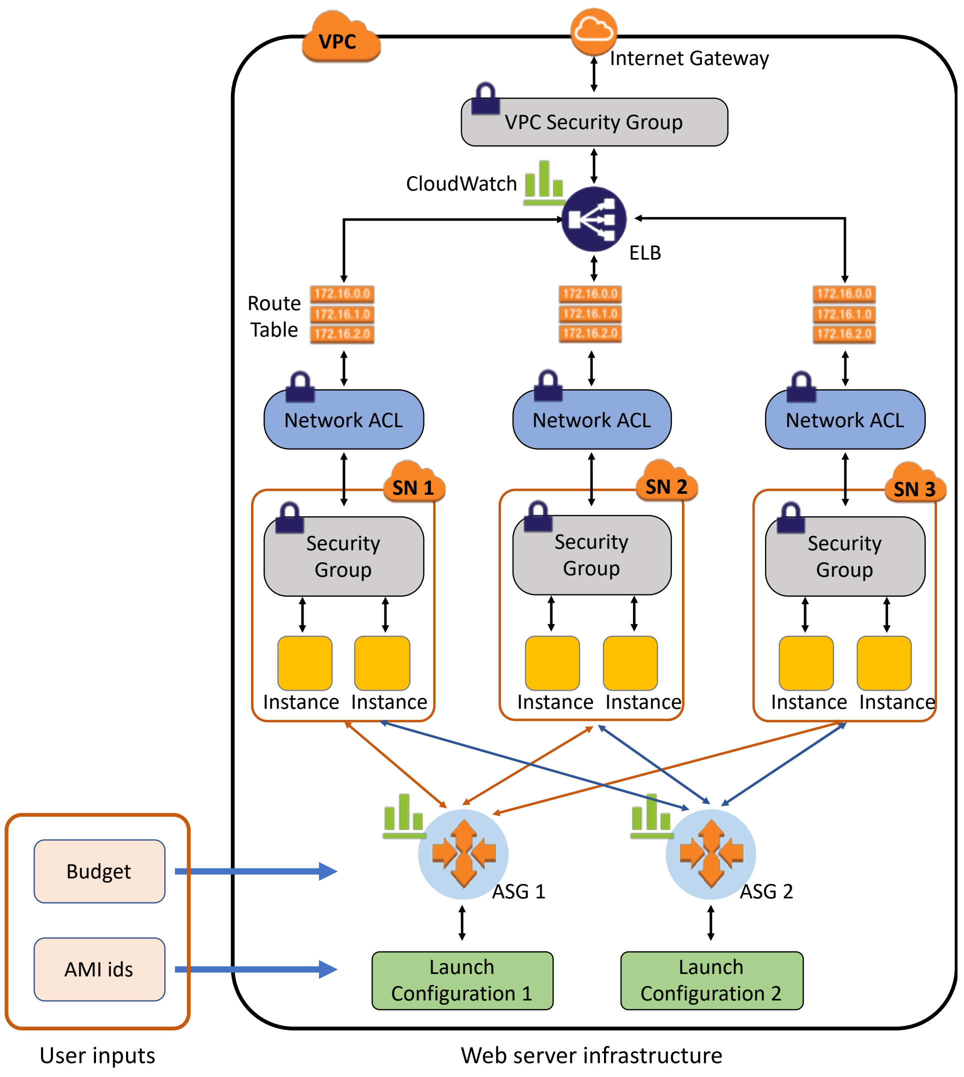

Figure 2. Complicated web server infrastructure with simple inputs from users.

Web application test with single t2.micro or a1.medium instance

According to the information from AWS, the t2.micro and a1.medium instances have very different properties. To show the different properties of these two instance types in terms computation and network bandwidth, we first performed experiments with one instance and one machine type each time.

We first performed experiments of these two machine types with 100% network queries (query 0) to test the difference in the capacities of bandwidth. The testing program is configured to send only query 0 with different frequencies ranging from 2 queries/second up to 30 queries/second. In our analysis, we consider a query with response time more than 1.5 seconds as a bad response. As mentioned above, in our design, we monitor and adjust the cloud resources in a 5 minutes window. So we did this set of experiment also with a 5 minutes window. As shown in the table below, with very low network traffic, both machine types can handle queries with no bad response in 5 minutes. When the frequency increase to 20 queries/second, the t2.micro instance start to have bad responses. This indicates that the t2.micro reached its bandwidth capacity and some queries are queue up because of no available bandwidth. The t2.micro instance cannot response to the queries as soon as possible. However, a1.medium did not have any bad response, which indicates the current network load is below its bandwidth capacity. With even 30 queries/s, the a1.medium instance can still handle requests with no slow response. On the other hand, the t2.micro instance has 17 bad responses in 5 minutes. This means that 17 users of this web application might get frustrated because of the slow response, which does not meet a high quality of service. This result showed, as expected, the a1.medium instance can handle significantly more network queries than t2.micro instance with no bad response happening.

Frequency	t2.micro	a1.medium
2/s	0	0
5/s	0	0
10/s	0	0
20/s	6	0
30/s	17	0

We then performed experiments to test the difference in computation power between t2.micro and a1.medium instances. In order to generate different computation requirements, we adjust the percentage of query 3 (high computation cost) range from 0% to 100%. The testing program sends queries with different frequencies, range from 2 queries/second to 10 queries/second. As shown in figure 2, with a frequency of 2 queries/second, the t2.micro can handle up to 90% of query 3 with no bad response, in contrast, a1.medium can only handle 15% of query 3 with no bad response. With a frequency of 5 queries/second, the t2.micro instance can handle 20% of query 3, but a1.medium can only handle 2%. When the frequency is set to 10 queries/second, a1.medium cannot handle any requests with no bad response time, but t2.micro can still hand 2% of query 3. This results indicates that the t2.micro instance has significantly more computation power compared to a1.medium instance. In summary, the t2.micro instance is more suitable for handling computation-intensive queries and a1.medium is better in handling network-intensive queries.

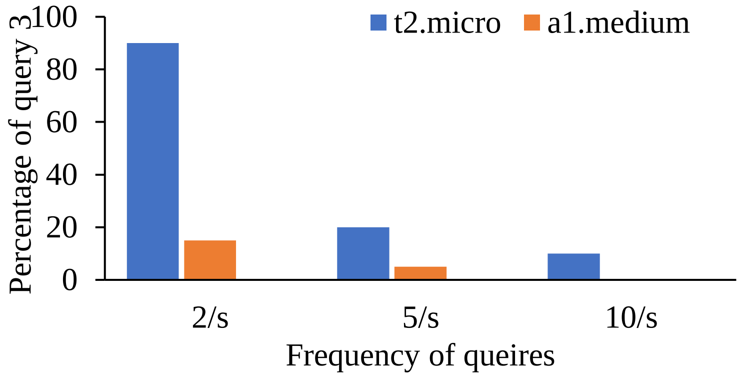

Figure 3. Difference of t2.micro and 1.medium instances in computation power.

Traditional auto-scaling strategy versus our system with low-level computation queries

With single instance tests, we can clearly see the different properties of t2.micro and a1.medium instances. However, it is rare that a web application receives and handles network or computation queries. Therefore, auto-scaling with one instance type might not cover all use cases, such as a mix of computation and network queries. In many cases, the performance is even worse when using the wrong type of instance. We configured the test condition to be 15% computation queries and 10 queries/second which is an computation-intensive test. The traditional auto-scaling strategy with only a1.medium instance type will increase the number of instances to be 5, but we still get bad response. However, with t2.medium instance type, auto-scaling increase the number of instances to be 2 and there is no bad response. Based on these observations, we proposed that a careful combination with these two instance types would increase the performance and reduce cost compared to traditional auto-scaling strategy. We then performed experiments to compare traditional autoscaling with one instance type or two instance types with our system with two instance types. Based on the experiments with single instances, we determined to set the threshold of average CPU utilization to be 70% and average network usage to be 70 Mb for the traditional autoscaling, which add or remove instances to prevent bad response happening.

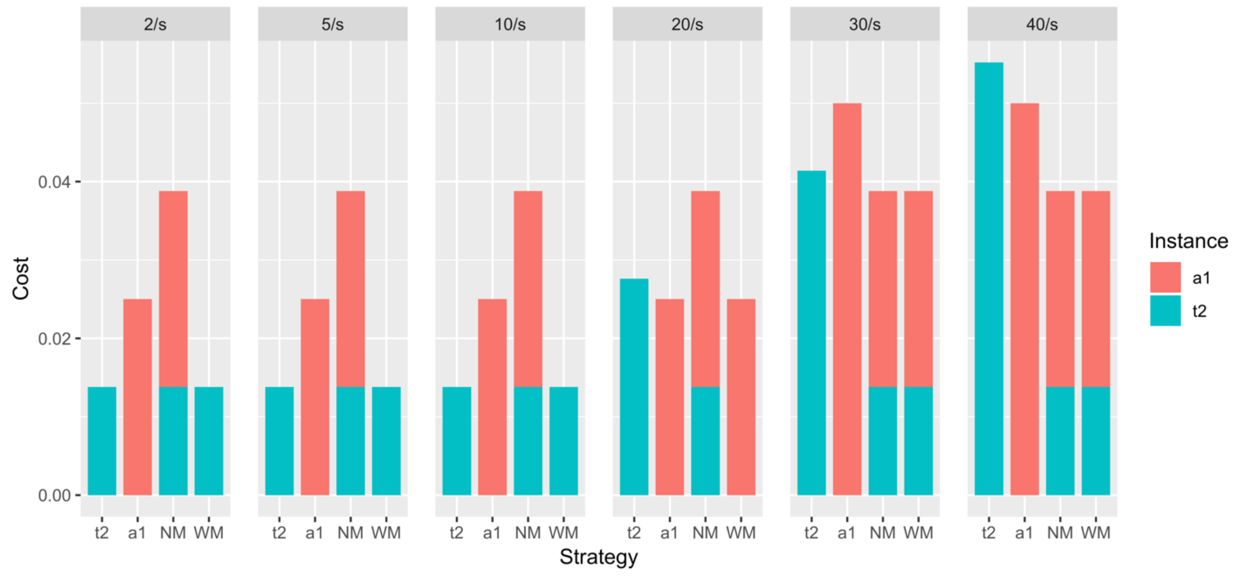

Figure 4. Network intensive queries with different frequencies.

Firstly, we performed experiments with only network intensive queries (query 0) to compare the cost of our system with auto-scaling strategy using one machine type or two machine types. We configured the test program to send 100% query 0 with a frequency ranging from 2 queries/second to 40 queries/second. As shown in figure 3, with long frequency of queries (2, 5, and 10 queries/second), our system and autoscaling with t2.micro only has the lowest cost. In fact, our system only uses t2.micro for these three cases. When network usage increased (20, 30, and 40 queries/second), our system remains to be the lowest cost option. With the 40 queries/second frequency, auto-scaling with t2.micro instance is the highest cost option. This indicates that with high network demands, auto-scaling with computation-optimized instances is not good selection of instance type due to high cost. With these results, we conclude that with high network usage, our system is always the cheapest option.

Then, we performed experiments with slightly increase of computation. Query 1 is designed to be a low-level computation query. When we mix the query 1 with query 0, overall computation is increased about three folds according to the computation time. The experiments are performed with a mixture of query 1 and query 0. The percentage of query 1 ranges from 10% to 40%. The frequency of the queries is ranged from 2 queries/second to 40 queries/second. As shown in figure 4, with 10% of query 1, the result is same as that with 100% query 0. With 20% of query 1, we got the same result as 10% of query 1. We can see with relative low computation demand per request, the network usage is the bottleneck of t2.micro. With low network traffic, t2.micro instance type works better. However, with high network traffic, adding the network-optimized instance type, a1.medium, can significantly reduce the cost. With 40 queries/second, our strategy can save 29.7% compared to only using t2.micro instance and 22.4% compared to only using a1.medium instance.

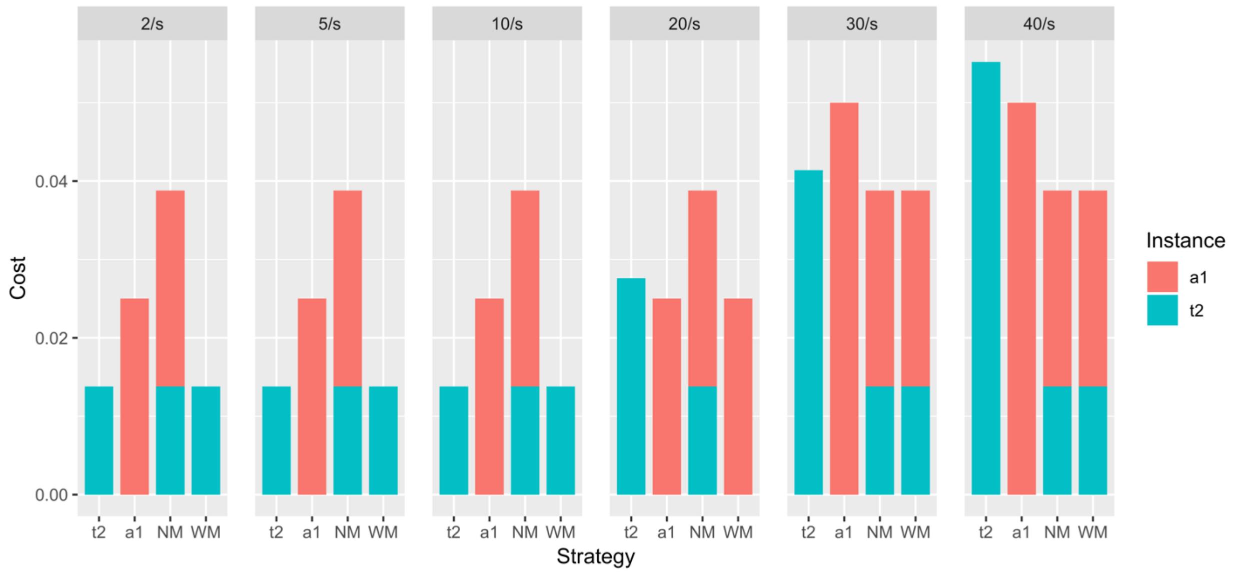

Figure 5. Mixed queries with 90% network intensive and 10% low-level computation.

When we increased the percentage of query 1 to 40%, the results is very different compared to figure 4. With 40% of query 1, the overall computation cost is significantly increased. As shown in figure 5, our system, together with auto-scaling with only t2.micro, are the lowest cost options for all cases from low frequency to high frequency. This indicates that when an web application has a lot of computation tasks, autoscaling using computation-optimized instance type is the best option. Our system is able to smartly make the choice of using t2.micro instances.

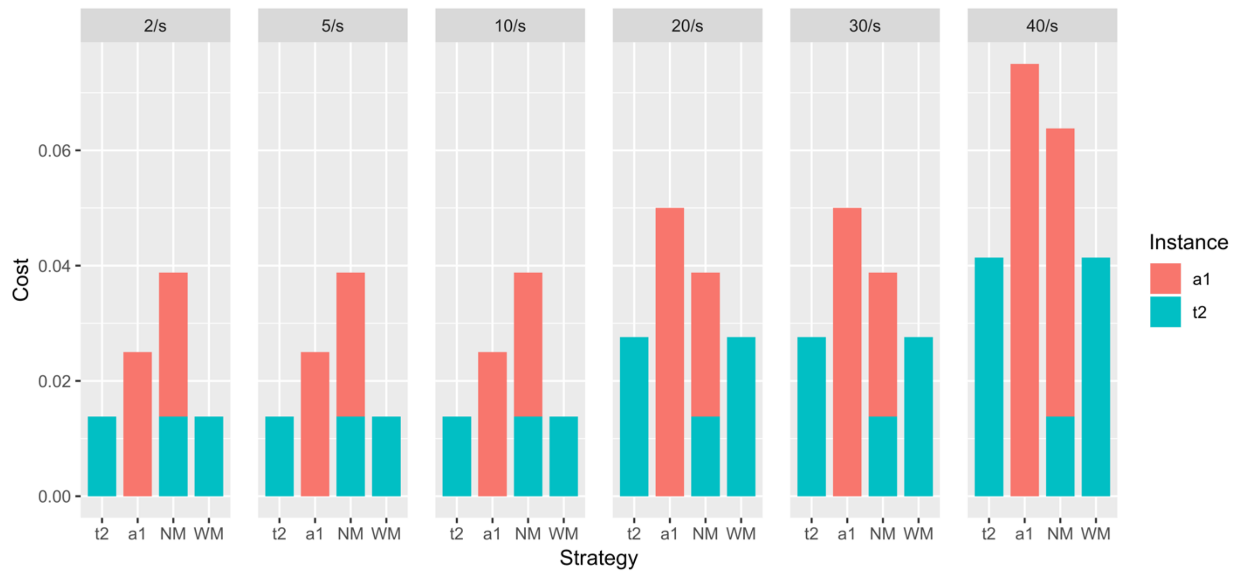

Figure 6. Mixed queries with 60% network intensive and 40% low-level computation.

Traditional auto-scaling strategy versus our system with high-level computation queries

In the last section, we saw that our system is able to make the right choice with low-level computation queries and high frequency,. In order to test if our system can still perform well with high-level computation queries, we further test our system with the other two other computation queries, query 2 and query 3. Based on the processing time, query 2 requires 5 times more computation power and query 3 requires 20 times more computation power compared to query 1.

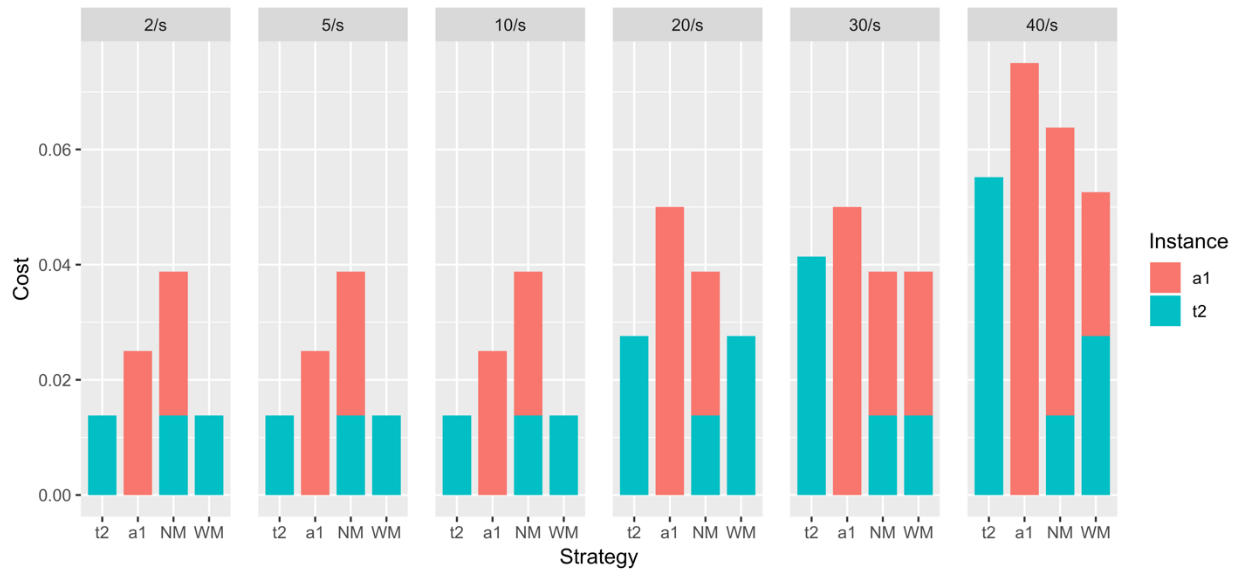

Figure 7. Mixed queries with 90% network intensive and 10% medium-level computation.

We first tested our system with query 2. The experiments are performed with a mixture of query 2 and query 0. The percentage of query 2 ranges from 10% to 40% and with a frequency ranging from 2 queries/second to 40 queries/second. As shown in figure 7, with 10% of query 2, our system is the cheapest option with different frequencies among four strategies. Similar as in figure 4 and figure 5, t2.micro instance is the better choice with low frequency of queries. With high frequency (high network usage), our strategy with these two instance types is the cheapest option (figure 6). This confirmed that adding the network-optimized instance is able to resolve the network bottleneck of t2.micro.

With 10% of query 2 and 40 queries/second frequency, The difference between our system and auto-scaling with only t2.micro becomes smaller compared to 100% network or 10% of query 1 with same frequency. This suggests that the computation-optimized instance type becomes more important in handling requests. We then further increased the computation cost per query by using query 3. With only 5% of query 3, auto-scaling with t2.micro and our system are the cheapest options for all the different frequencies (figure 7).

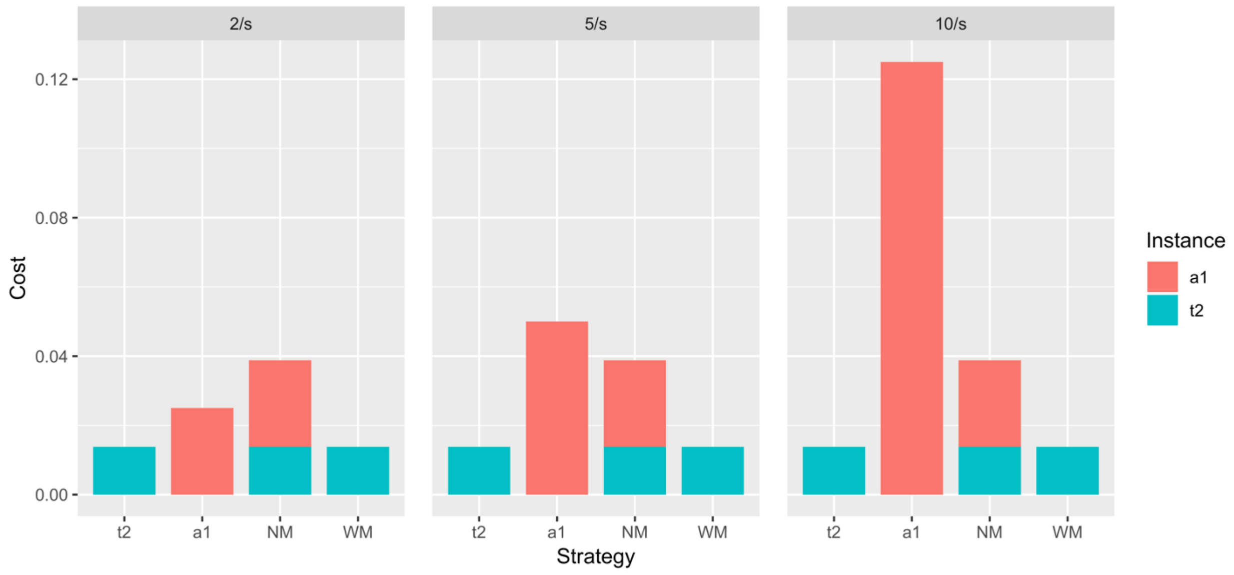

Figure 8. Mixed queries with 90% network intensive and 10% heavy-level computation.

With all the above results, we concluded that our system can smartly configure the cloud resources and reduce cost while maintaining high performance. Our system works well with many different use cases, such as high network usage, high computation cost caused by low-level computation queries, and high computation cost caused by high-level computation queries. This is an extreme valuable feature for many users with complicated web applications. They do not have to find all possible use cases and then optimize the infrastructure for all of them. With our system, they only need to provide machine types that work well for different features of a web server, like CPU usage, memory usage, or I/O operations.

Fast reaction to sudden burst of network or computation queries

One of the drawbacks of traditional auto-scaling strategy is that only certain number of instances can be added or removed a time. This might not work well with sudden change of traffic pattern. For example, adding one instance of t2.micro to the auto-scaling group might not be able to compensate for a large increase of computation queries and result in bad performance. On the other hand, with gradually decreasing of instances will cause unnecessary cost with sudden decrease of computation queries. Our system calculates the optimal number of instances for each auto-scaling group and adjusts the size of them directly. Therefore, we hypothesize that our system can perform better with sudden change of traffic due to the faster reaction. At the beginning, the two instance types are set to have 1 instance each. We then configured the test program to send queries with 60% query 1 and 40 queries/second. As shown in table, our monitoring system can reduce the bad responses with one step of scaling and the traditional auto-scaling strategy required two step of scaling to achieve 0 bad response.

	1st 5min	2nd 5min	3rd 5min
Traditional ASG	20	13	0
Our system	20	0	0

Better performance and lower cost with real-world-like traffic pattern

In all the previous experiments, the test program is fixed with visiting certain query with a fixed frequency. To mimic a real-world web application with changes in traffic pattern, we implemented a test program that mimic the network traffic pattern happened over one day. The experiment is set up as shown in the table.

	Frequency	Query 0 (%)	Query 1 (%)	Query 2 (%)
8 - 10 am	30/s	60	40	0
10 - 12 pm	30/s	90	0	10
12 - 2 pm	10/s	60	0	40
2 - 4 pm	20/s	100	0	0
4 - 6 pm	20/s	80	20	0
> 6 pm	5/s	90	0	10

We tried to mimic this application as a stock market tracking and trading application. In between 8am to 10am, a lot of clients come into office and first need to login to their account. The login requests required some computation to verify user credentials. After login, users start to look at the stock market. We mimic this period of time by configure the test program to send 30 queries/second with 60% query 0 and 40% of query 1. In between 10am to 12am, more users login to the application and check the price of stocks. They might decide to sell some stock and make requests for trading, which are the queries require more computation. We mimic this period of time by configuring the test program to send 30 queries/second with 90% query 0 and 10% query 2. In between 12pm to 2pm, some users go to lunch and some users continue to work and decided to make some stock trading. We mimic this period of time by configuring the test program to send 10 queries/second with 60% query 0 and 40% query 2. In between 2pm to 4pm, users might just tracking the market after trading. We mimic this period of time by configuring the test program to send 20 queries/second with 100% query 0. In between 4pm to 6pm, users start to logout from the application and prepare to go home. We mimic this period of time by configuring the test program to send 20 queries/second with 80% query 0 and 20% query 1. After 6pm, some users might work late and do some final minutes trading. We mimic this period of time by configuring the test program to send 5 queries/second with 90% query 0 and 10% query 2. We performed experiments with this test configuration against our system and traditional auto-scaling with one machine type and two machine types. As shown in the table below, our system can save 17.8 % compared to auto-scaling with only t2.mcro, 58.9% with only a1.medium, and 34.6% with two instance types. In addition, our system only have 24 bad responses (response time > 1.5 second), which indicates better performance. We believe this is the result of faster reaction to sudden changes of traffic pattern compared the traditional autoscaling strategies.

	Bad response	Hourly Cost ($)
t2.micro	37	0.0303
a1.medium	218	0.0608
Two instance types	120	0.0381
Monitoring with two types	24	0.0249

With this real-world-like test, we showed the our system can smartly make combinations of two instance types to have higher performance and lower cost. We achieved this by analyzing two pieces of information from logs, which are readily available from AWS services as well as other cloud providers. We believe that the system can achieve better performance and even lower cost with analysis of more information from logs and smarter analysis programs.

Conclusion

With all the results above, we concluded that our system can achieve both high performance and low cost compared to the traditional auto-scaling strategy with one or two machine types. Our system only requires users to provide a budget and the machine images they want to use for the web application. Our system builds the infrastructure for the web application using the infrastructure as a service (Iaas), which is the lowest level of service. Iaas provides more flexibilities of the usage of cloud resources and can save cost while achieving better performance. With our monitoring system, we are able to achieve up to 64% saving without getting any bad response. In addition, our system is suitable for any use cases, such as network only requests, network and light computation requests, and network and heavy computation requests. This is a huge benefit for users that have complicated web applications. With our system, they do not need to spend much efforts to find the most suitable images for their application. The users only need to configure images that performs well with certain use cases and provide them to our system. Our system can smartly find the best combinations to meet both good performance and low cost. Compared to traditional auto-scaling strategy, our system can dynamically add or remove instances without the limitation of a certain number in one step of scaling. This feature allows faster reaction to the sudden change of traffic pattern, increase performance, and reduce unnecessary costs.

Discussion

The existing cloud services for building a web application are either too complicate or expensive. Furthermore, many existing cloud services for web applications hide the implementation details and are hard to adjust for certain use cases. In order to achieve high performance, the most common strategy is auto-scaling [7]. However, the traditional auto-scaling only associate with one type of instances. In many use cases, simply scaling-in or scaling-out a certain type of instance can achieve good performance, but may introduce very high cost. These options might not suitable for many web applications. Building infrastructure for a web application with the bare-metal (like virtual machines) resources from AWS can significantly reduce the cost and also provide more flexibilities. However, for average users, the effects for learning and researching the cloud computing could be too much and is a huge barrier for migration to cloud computing paradigm. Our research demonstrated a possibility of building web applications using Iaas and only needing minimal effects from user.

We achieved both good performance and low cost and we did this by only acquiring the CPU utilization and network throughput from AWS. These two pieces of information are easily available through CloudWatch service of AWS, whose cost is neglectable compare to any other cost of cloud resources. However, CloudWatch or other services can provide a lot more logs of almost every aspect of the cloud resources being used. For example, the logs from load balancer include many entries including request processing time, target response time, response processing time, and more. Instance logs provide very detailed information about CPU, memory, and disk i/o processes. If we can leverage these information and build even smarter scaling strategy, we may be able to achieve even better performance and lower cost.

In addition to better scaling strategy, we might get more benefit if we combine a good scaling strategy with a smart load balancer. In our current system, we use the default load balancing strategy, round-robin, for distributing the queries. However, many times, a computation-intensive query is sent to the network optimized instance. If we can avoid this case by directing the computation-intensive queries to computation-optimized instances and network-intensive queries to network-optimized instances, we believe the performance can be even better. In order to achieve this, we may need to develop algorithms to understand the context of queries and differentiate them. One potential method is to train an machine learning algorithms to cluster the queries into different categories. As mentioned above, the logs from many cloud services can provide many features related a query, such as the target response time, network in and out bytes, and memory usage. With these features, it would be possible to train a machine learning algorithm. However, the trained algorithm might be specific to a specific web application. Therefore, it might require certain amount of efforts to train the algorithm before actual deployment of the web application.

Our system is relatively simple. We build the entire infrastructure within one region and with one VPC. This might work well for small businesses that have limited funds and do not care too much about the performance. In order to serve as a more general system, it needs to take care of users that needs high performance. We propose that the system can be expand to multiple regions and multiple VPC. The system might allow users to choose if they want to deploy their application in multiple regions depending locations of clients. For example, the clients of a web application mainly located in west coast and east coast. If all servers are setup in us-west region, even with the best configuration, the east coast users still suffer from high latency. Therefore, if the system can build infrastructure in both us-west and us-east regions, all clients have servers that are close to them and therefore with low latency. In addition to multiple regions and VPCs, AWS and other cloud providers provide content delivery network, such as CloudFront, which caches contents in AWS global date centers and reduces latency of network requests. For users that have enough funds and have needs for high performance, we might want to add the CloudFront as a option. Therefore, we propose that a more complete system would include different tiers of architectures that satisfy the requirements for small or large businesses. However, with adding these tiers, the system should remain user-friendly. In order to achieve this goal, we need to extract high-level ideas of creating and deploying web server, which can be easily understood by users and are readily available from their previous experience with web servers. We further summarize these high-level ideas as inputs for our application to create and manage cloud resources. The users only need to provide some inputs they are already familiar with.

In this research, we showed that the potential of leveraging logs from AWS to build user-friendly and cost-effective systems for building web application infrastructure. More studies can be performed with other aspects of logs to achieve even better results. It is also beneficial to develop smart load balancing strategies to direct queries to a certain type of instances. This can be possibly achieved by machine learning techniques.

References

[1] Osanaiye, O., Choo, K. K. R., & Dlodlo, M. (2016). Distributed denial of service (DDoS) resilience in cloud: review and conceptual cloud DDoS mitigation framework. *Journal of Network and Computer Applications*, 67, 147-165.

[2] Stamford, C. (2016). Gartner says by 2020 cloud shift will affect more than 1 trillion in it spending. *http://www.gartner.com/newsroom/id/3384720*.

[3] AWS Elastic Beanstalk documentation. *https://docs.aws.amazon.com/gettingstarted/latest/deploy/overview.html*

[4] Chieu, T. C., Mohindra, A., Karve, A. A., & Segal, A. (2009). Dynamic scaling of web applications in a virtualized cloud computing environment. In *E-Business Engineering, 2009. ICEBE'09. IEEE International Conference on* (pp. 281-286). IEEE.

[5] Davatz, C., Inzinger, C., Scheuner, J., & Leitner, P. (2017). An approach and case study of cloud instance type selection for multi-tier web applications. In *Proceedings of the 17th*

IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (pp. 534-543). IEEE Press.

[6] Iqbal, W., Dailey, M. N., & Carrera, D. (2010). Sla-driven dynamic resource management for multi-tier web applications in a cloud. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing* (pp. 832-837). IEEE Computer Society.

[7] AWS Auto Scaling documentation. *https://docs.aws.amazon.com/autoscaling/plans/userguide/what-is-aws-auto-scaling.html*

[8] AWS Elastic Compute Cloud Documentation. *https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html*