ECE 422

Reliable Secure

Systems Design

Reliability Project

Submitted by

Sangaré, Ibrahima Séga

Gauk, Lucas

Sunday April 14^{h}, 2019

Contents

Abstract

As part of the ECE 422 course (Reliable and Secure Systems Design), students are required to participate in a project based around reliability. The project covered the second half of the semester and covered many concepts discussed in the classroom. The objective of this project was to provide a design and implementation of reactive auto-scaling engine for a cloud microservice. Essentially, the auto-scaler would monitor the response time of user submitted client requests. Based on this metric, the auto-scaler would thus adjust the number of replicas of the web app responsible of handling client requests. The expected scaling process should be addressing horizontal scalability. This suggest that only the number of replicas are affected in order to manage a growing or shrinking amount of client requests with increasing and decreasing response times. In this report, we cover important notions related to this project, technologies and explanations behind the provided implementation, design artifacts, deployment instructions, as well as a detailed user guide. We conclude the report with takeaways and knowledge acquired through this project by summarizing the different aspects of the project.

I – Introduction

Theoretical Considerations

In computer science, reliability is an ever-growing aspect of all computer systems in their effectiveness in terms of being useful to all the users implicated. This concept is consistent with computer security which has been covered in the first project of this course. They are related topics since the reality of computer science is such that distributed systems are more and more needed to provide the different services in daily life. Reliability can be defined as an attribute of assets involved in a computer system, that allowing the whole structure to consistently defined according to its specifications [1]. In that sense, it goes hand in hand with computer security since a case can be made for consistency in protecting assets, especially in an environment based in the cloud. Reliability can be further restricted a specific context such as cloud computing. As more and more services are migrating to the cloud, the context of this project serves to illustrate the main aspects that are at play here. Having many clients to supply within an acceptable time frame suggests that there are one or more components in charge of monitoring health and operational status of microservices and components .

This monitoring process most often requires an algorithmic approach to the adjusting the number or size of resources dedicated to providing services. Scalability is therefore the essential part of any algorithm that is used to monitor an application. As mentioned earlier, scalability can refer to an increase in the amount of operations that any single resource can handle before getting flooded. This is known as vertical scalability and also covers cases where a given resource is under utilized, in which case, it can be downsized. Another form of scalability is horizontal scalability. It implicates that the number of instances providing a service can vary in number based on determined upper and lower bounds that will dictate a change, or lack thereof, in number of replicas. Both forms of scalability are influenced by factors like rapid spike in demand, outages, and variable traffic patterns [2].

A great reason behind an algorithmic approach to scalability is to counter issues that arise in a system when it operates at the boundaries of acceptable behavior, as it can be apparent in a rapid spike in demand. If this not taken into account, a phenomenon that can be defined as a “ping-pong” effect can occur, in which case rapid changes in instance counts can affect the optimal use of resources. The scalability can be automated via many effective algorithms. Predictive-reactive models can serve as example to this auto-scaling process. For instance, Scryer, the predictive-reactive auto-scaling engine used by Netflix uses patterns in traffic and average load in order to predict to a certain degree how many instances would be required in advance [3]. This method is very useful in attempting to optimize resource usage and also react to unusual circumstances by keeping track of current ongoing demands imposed by clients. Other elaborate methods also make use of neural networks to predict the required amount of instances [4].

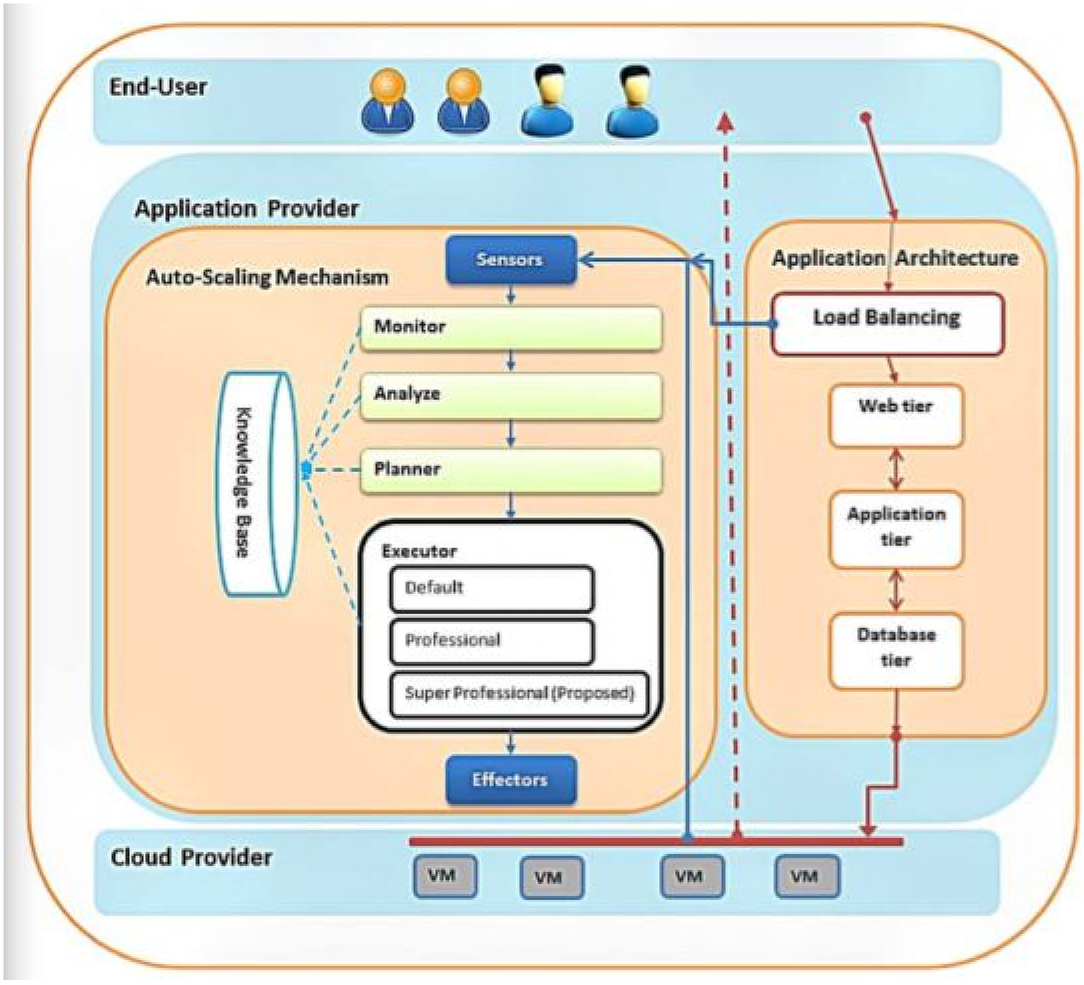

Figure 1: Illustration of auto-scaling in a cloud based application (MAPE-K model). Source: Journal of Network and Computer Applications 95 (2017)

The figure above summarizes the thought process behind regulating instance count in a cloud based applications. In a general sense, a monitor is collects information about ongoing operations in a web applications (response time to a request for example). The collected data is analyzed to develop a plan of action. This plan ultimately indicates potential changes in terms of scalability the application.

II - Technologies

Elastic stack

We needed a way to log workload and transaction metrics in order to display trend lines and make decisions about when to scale our web service. The decision to pick elasticsearch was based entirely on which other adjacent technologies were available along with the NoSQL database to help accomplish our goal. Elasticsearch has a great dashboarding tool, as well as a fantastic Flask library that handles transaction metrics for each instance of our Flask app. This was really the deciding factor, as there was no great way to export our response times to other databases. Kibana is elastics in house dashboarding server, and it is fairly robust. The graphing is not quite as good as something like Grafana, but it works well with Elasticsearch straight out of the box and was enough for the purposes of this project so we didn't look elsewhere to replace it. In the future, something like Grafana could be plugged on top of Elasticsearch instead of Kibana for more control and better graphing capabilities. APM server was needed in order to get data from the Flask apps. The libraries provided by elastic sent their data to APM server which acted as an intermediary for our Elasticsearch database. APM clients like our web app could stream their data straight to an endpoint provided by the server, and it would turn it into Elasticsearch documents.

Metricbeat is another tool provided by elastic. It is, like the Flask app, another APM server client that streams information about the docker containers running on any instance to the Elasticsearch database. This information can then be seen in Kibana.

Autoscaler

The autoscaler is a fairly simple python script, enabled by the Elastic stack described above. Elasticsearch provides an HTTP api to perform CRUD operations. We simply query the response time average seen in the last 30 seconds and scale according to a simple algorithm found in the Kubernetes Horizontal Pod Autoscaler. The equation is as follows:

desiredReplicas = ceil[currentReplicas * (currentMetricValue / desiredMetricValue)]

We set our desired metric value to the speed seen by the webapp with a single client, which was around 2 seconds per transaction. We limited the amount of replicas to 24, which was the highest number of replicas we could have while still seeing improvements in performance. We sample the database every 20 seconds to allow for our scaling to settle, and for the averages we query to be meaningful.

III - Design Artifacts

The following is an accurate representation of the Elastic stack we use to monitor our Flask app and docker instances. In our case, our APM agents are the Flask applications and the Metricbeat instances on each node.

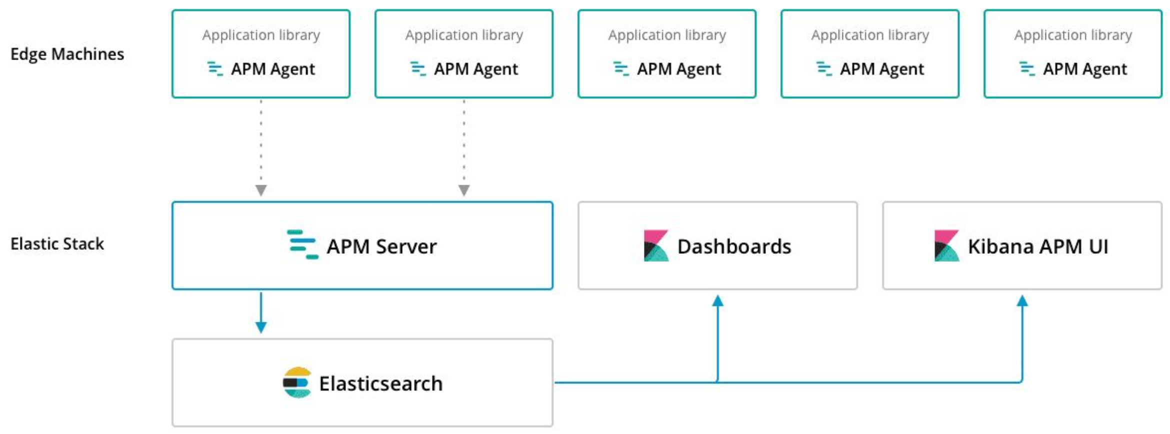

Figure 2: High-level architectural view of the application

In terms of the Autoscaler, there are two classes involved.

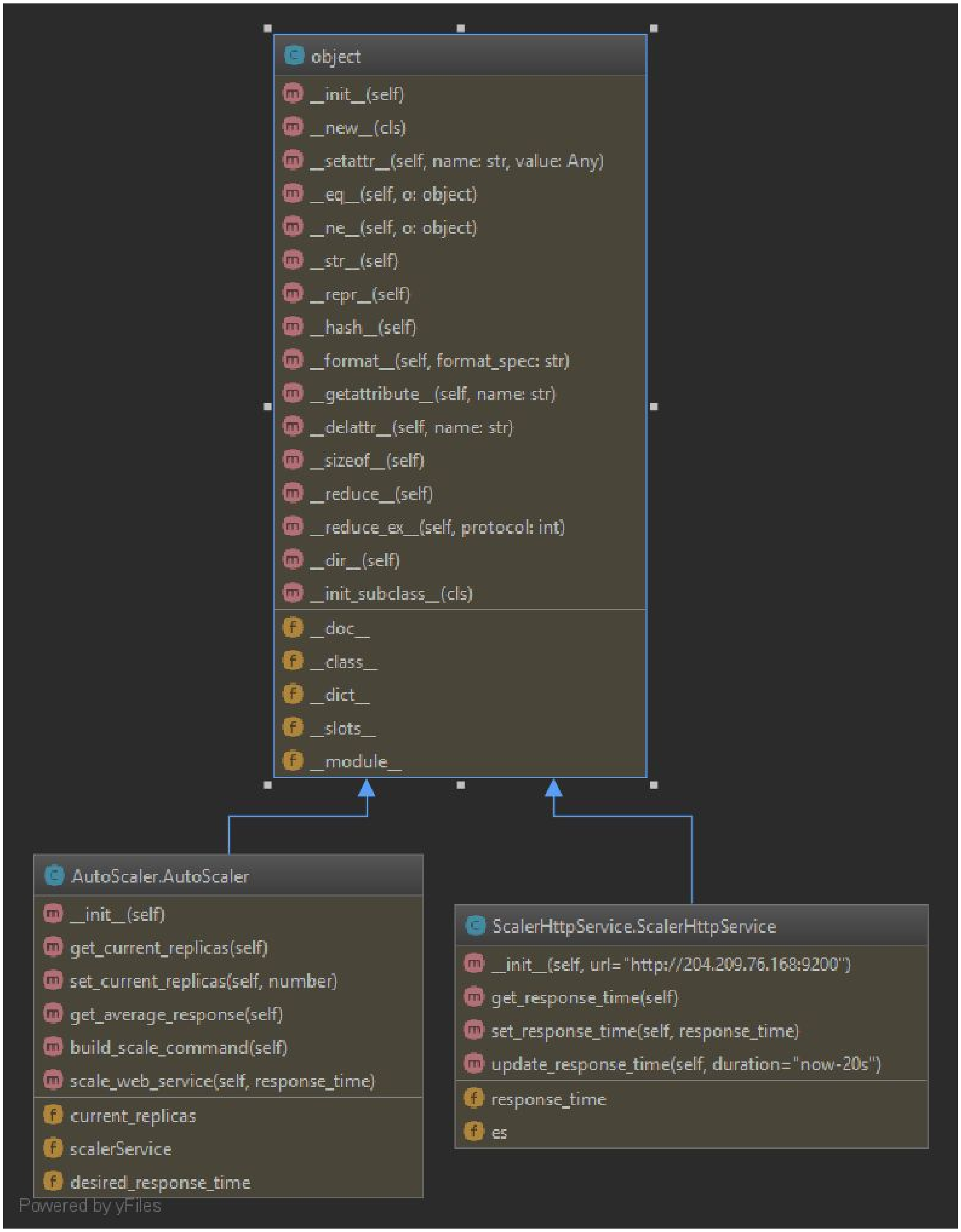
object
__init__(self)
__new__(cls)
__setattr__(self, name: str, value: Any)
__eq__(self, o: object)
__ne__(self, o: object)
__str__(self)
__repr__(self)
__hash__(self)
__format__(self, format_spec: str)
__getattribute__(self, name: str)
__delattr__(self, name: str)
__sizeof__(self)
__reduce__(self)
__reduce_ex__(self, protocol: int)
__dir__(self)
__init_subclass__(cls)
__doc__
__class__
__dict__
__slots__
__module__
AutoScaler.AutoScaler
__init__(self)
get_current_replicas(self)
set_current_replicas(self, number)
get_average_response(self)
build_scale_command(self)
scale_web_service(self, response_time)
current_replicas
scalerService
desired_response_time
ScalerHttpService.ScalerHttpService
__init__(self, url="http://204.209.76.168:9200")
get_response_time(self)
set_response_time(self, response_time)
update_response_time(self, duration="now-20s")
response_time
es
Powered by yFiles

Figure 3: Class diagrams of the Autoscaler and HttpServive.

The AutoScaler class does what the names implicates: it adjust the number of replicas of the web app based on an average response time given by the HttpService. The HttpService gathers the average response time over the last 20 seconds. This information is collected via request to APM. The scaler periodically gets the updated average response and update the number of replicas through a system call, by using the formula described earlier (Kubernetes Horizontal Pod Autoscaler).

The following represent a state diagram displaying the events and actions occurring in the auto-scaler.

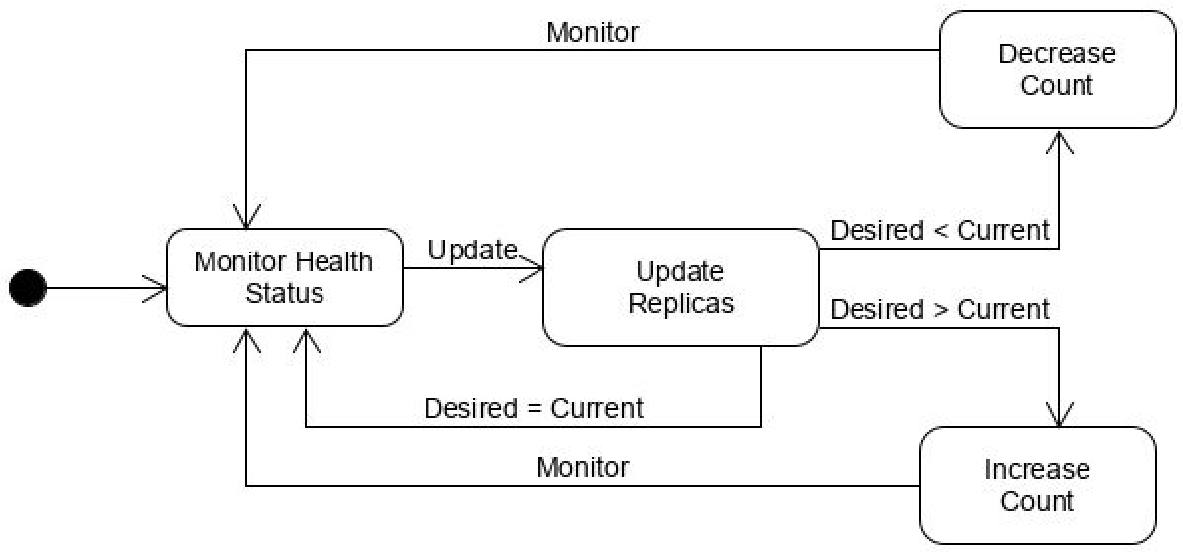

Figure 4: State diagram of the auto-scaler.

As already explained, the autoscaler monitors the health status of the web app. It can scale out, scale in or keep the same number of replicas based on the response time as the desired metric value.

IV - Deployment instructions

The monitoring stack is fairly complicated, but the generalized instructions for how to set it up on a single node are as follows.

Elasticsearch, Kibana and APM-server

Elasticsearch handles all the transaction logging. Kibana is an elastic product that does dashboarding and information retrieval for our Elasticsearch database. APM server handles requests to store data in Elasticsearch and correctly formats the documents To install all three:

1. Install Elasticsearch using APM.
2. Go to /etc/elasticsearch/ and edit the elasticsearch.yml. Replace any instances of localhost with 0.0.0.0.
3. Install Kibana using APM.
4. Like before, go to /etc/kibana/ and edit the kibana.yml file. Replace all instances of localhost with 0.0.0.0.
5. If kibana is installed, you can go to <your IP>:5601/app/apm to get instructions on how to install apm-server.
6. Again, go to /etc/apm-server/ and edit the apm-server.yml file. Replace all instances of localhost with 0.0.0.0.
7. Enable apm kibana dashboards with apm-server setup --dashboards.
8. Use the tools at <your IP>:5601/app/apm to verify that the apm server is correctly installed.
9. Use systemctl to restart all three services (elasticsearch.service, kibana.service, apm-server.service)

Web App

Our demo web app is nearly the same app as was provided for the assignment with a few small modifications that logs transaction information and sends it to the apm-server.

1. Clone the https://github.com/lucasgauk/elasticsearch-flask repository.
2. Edit the python app to point to the correct apm-server instance.
3. Run the following commands to build and publish the webapp.

```
sudo docker build --tag=elasticwebapp .
sudo docker tag elasticwebapp <docker username>/elasticwebapp:v1
sudo docker push <docker username>/elasticwebapp:v1
```

Edit the docker-compose.yml file to locate the right webapp to run. Run the following to start the webapp, Redis, and a simple visualizer to view your docker containers.

```
sudo docker stack deploy <stack name> --compose-file <path to docker compose.yml>
```

Verify that Redis, your Flask webapp, and the visualizer are all running properly.

Open <your IP>:5601/app/apm again and use the built in tools to verify that your Flask app is sending information to the apm-server.

Metricbeat

Metricbeat is an elastic product that sends information about a nodes docker images and processes to APM server. We can then monitor it using Kibana.

Run the following command to install Metricbeat.

```
sudo apt-get install metricbeat
```

1. Navigate to /etc/metricbeat/ and edit the metricbeat.yml file. In the kibana section, edit the host to match the IP of your kibana instance. In the outputs section, edit the outputs.elasticsearch hosts to point to your elasticsearch IP. Restart the metricbeat.service.
2. Navigate to /etc/metricbeat/modules.d and run cp docker.yml.disabled docker.yml to enable docker exporting.
3. Restart the metricbeat service with sudo systemctl restart metricbeat.service.
4. Enable dashboards for kibana with metricbeat setup --dashboards.
5. Verify that your Kibana instance is receiving information from Metricbeat.
6. Note that any other instances with metricbeat installed will automatically be added to the same dashboard!

You can now monitor the request volume and response time of your Flask app, along with basic information about the docker instance that's running it.

Autoscaler

The autoscaler uses the transaction time information logged by elasticsearch to determine how it should scale the webapp.

1. Clone the https://github.com/segason/ECE-422-Autoscaler repository.
2. Start a virtualenv in the folder created by git.
3. Run sudo pip install -r requirements.txt to install the autoscaler requirements.
4. Modify the ScalerHttpService.py file to point to the correct elasticsearch instance.
5. Run python AutoScaler.py and monitor response times on Kibana.

You now have a Flask web app that reports it’s metrics to a NoSQL database and automatically scales itself depending on the response times, as well as a dashboarding tool to monitor its behaviour.

VI - Conclusion

In conclusion, we were able to design an auto-scaling engine using available and widely used frameworks to put into practice theoretical knowledge from this course. To summarize, this report discussed the reasons and goals that motivated this project. We covered extensively all the details related to the technologies used to design a solution for this particular scenario. Design artifacts described the overall behavior of the auto-scaler. The deployment instructions and user guide should allow an external to setup the project and observe the auto-scaling process. Throughout the project we were able to experiment and design an auto-scaling in a cloud based environment and thus, improve our knowledge in this context.

References

[1] What is reliability? - Definition from WhatIs.com. (n.d.). Retrieved April 13, 2019, from https://whatis.techtarget.com/definition/reliability

[2] Netflix Technology Blog. (2013, November 05). Scryer: Netflix's Predictive Auto Scaling Engine. Retrieved April 13, 2019, from https://medium.com/netflix-techblog/scryer-netflixs-predictive-auto-scaling-engine-a3f8fc922270?fbclid=IwAR3JK0TvpNqEFb_t5V4Z0avqs9stgT0pv8aWhxOK7PdUhPLTlZD231Fom7E

[3] A., A. (2016). Automatic Cloud Resource Scaling Algorithm based on Long Short-Term Memory Recurrent Neural Network. *International Journal of Advanced Computer Science and Applications,7*(12). doi:10.14569/ijacsa.2016.071236

[4] Aslanpour, M. S., Ghobaei-Arani, M., & Toosi, A. N. (2017). Auto-scaling web applications in clouds: A cost-aware approach. *Journal of Network and Computer Applications,95*, 26-41. doi:10.1016/j.jnca.2017.07.012

Elasticsearch: https://www.elastic.co/products/elasticsearch
Kibana: https://www.elastic.co/products/kibana
APM-Server: https://www.elastic.co/guide/en/apm/server/current/overview.html
Metricbeat: https://www.elastic.co/products/beats/metricbeat
Horizontal Pod Autoscaler:
https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

Our Elastic stack: https://github.com/lucasgauk/elasticsearch-flask
Our Autoscaler: https://github.com/segason/ECE-422-Autoscaler