6/23/2024

ML-Powered Auto-Scaling in AKS

Individual: FestivalConnect

Name: Lucas Jacobs
Class: S-A-RB06
PCN: 490692
Student number: 4607368
Technical teachers: Felipe Ebert, Bartosz Paszkowski
Semester coach: Gerard Elbers

Table of Contents

Introduction

Horizontally scaling your application is a must in enterprise software applications. An important factor that comes to mind is automatically scaling your application based on the demand of the usage. The use of autoscaling of pods will allow clusters in Kubernetes to instantly react based on resource specifications to scale up which will lead to more elastically and efficiency. How can this be done as efficiently as possible, with the usage of pre-build features in Kubernetes? Or by advancing it by using a predictive model? This document will cover how horizontal autoscaling is possible and how machine learning can play a role in optimizing the application's demand for resources. (Autoscaling Workloads, n.d.)

Background

Automatic Scaling In Kubernetes

With the current state of my application, I am using two different types of automatic scaling:

- **Horizontal Pod Autoscaling**: Specify based on utilization of CPU usage of a certain Pod, how the application should autoscale
- **Autoscaling using CronJob**: Based on a certain period, predefine how many replicas of a service are needed.

More information is specified on how this works in my implementation document (Jacobs, Azure Monitoring Deployment).

Advantages and Challenges with these types

Using HPA has benefits such as optimizing resource utilization with the use of adjusting the number of replicas that are needed for a service based on demand, which will reduce waste and limit costs. Nevertheless, the optimal balance between resource utilization and application performance requires careful decisions and proper monitoring. Furthermore, HPA offers flexible scaling, supporting various metrics that will enable custom scaling based on your requirements.

There are also disadvantages to HPA. HPA makes use of metrics such as CPU or memory usage to scale up your pods. This can lead to delays between the time demand increases and the time that the extra pods are ready to meet this demand. Moreover, it will lead to slowing down response times, which will reduce performance. What can also happen is that when more services scale up at the same time, they can start competing for CPU and memory resources.

With the use of CronJobs, we will not be able to properly scale to the actual demand since it will have a predefined set of replicas, which can lead to either over-utilizing resources or vice versa.

(Singh, n.d.)

Proposed Solution

So by looking at scaling using CPU or memory, it can often be not enough when certain surges or demands can overwhelm the servers, which leads to Kubernetes not keeping up with this, leading to crashes and dissatisfied users since autoscaling in Kubernetes can be slow. The metrics server has a default interval check of 15 seconds to check utilization and there will be still a delay between actually creating the pod. (Why Is HPA Scaling Slow, 2023)

Therefore, I propose to scale up the application based on the anticipated load before the actual load happens. This will be done by using machine learning to make a model that will predict load based on a specific period. To give an example. When having a period of one month, each week on the weekends there is an increase in usage during the evenings. The model will be trained to anticipate this and will predict to scale up during this time.

Introduction to Prophet and KEDA

Keda is a Kubernetes Event Driven Autoscaler. It provides scale pods in Kubernetes based on the number of events that are needed to be processed. It is easy to add components that can be added to my Kubernetes cluster. (Kubernetes Event-driven Autoscaling, n.d.)

Facebook's Prophet is an open-source algorithm that generates time-series models. Time series is the collection of data that can track metrics over time, the more data you have the more accurate you can predict and train your model. It is easy to use and can be used to forecast business results without needing an expert. (Resende, 2023)

So in practice, with the use of these two software applications, we can predict the future workload demands using time-series forecasting and trigger autoscaling in Kubernetes. Prohet’s predictions can inform KEDA when to scale up or down based on anticipated demand, which leads to more efficient resource management.

Hypothesis

Hypothesis Statement

I suppose that the use of machine learning for predictive autoscaling in Kubernetes, will significantly improve resource management more efficient and reduce latency to traditional autoscaling methods.

Expected Benefits

Improve the efficiency by predicting the scaling on the anticipated load, making sure that the application is available, and reducing latency. Also with scaling on predictions on previous historical events, resource wastage is limited. Finally, it will improve performance by having faster response times and reduced errors when there is high demand.

Methodology

Data Collection

I am going to be using a preset of data for testing purposes, since in real life you would have a dataset of a much larger timestamp. This is only for testing purposes to try and predict the future using a certain dataset with timestamps and values.

Model Training

The model needs to be trained in a way that it gives an input ‘Time’ and gives an output of the corresponding replicas that are needed. This means we need a time series model that gives replicas over time. Give the following example for more explanation.

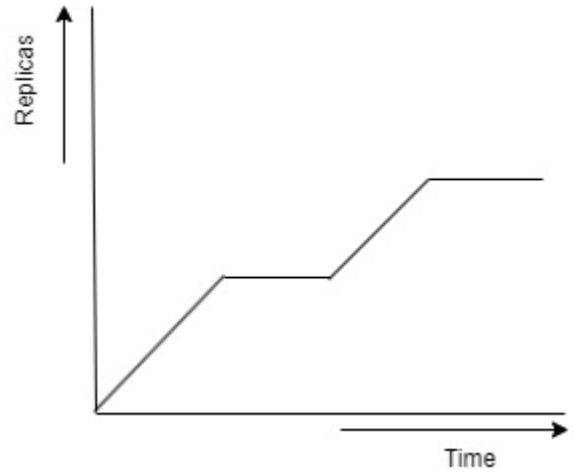

Figure 1: Model Replica over Time

Over time your model performance decreases and becomes inaccurate, therefore training your model is a key aspect of this.
When using the number of replicas (pods) as input for retraining the model, it means that we are using the predictions that were previously made by the model itself. This leads to biased inputs because a model’s errors get thrown back into itself, causing it to become less accurate over time. To avoid this, it is better to use actual observed data instead of only the models’s predictions. Therefore in this case I am going to be using the “Request Throughput”. The final model will look like the following (Codes, 2023)

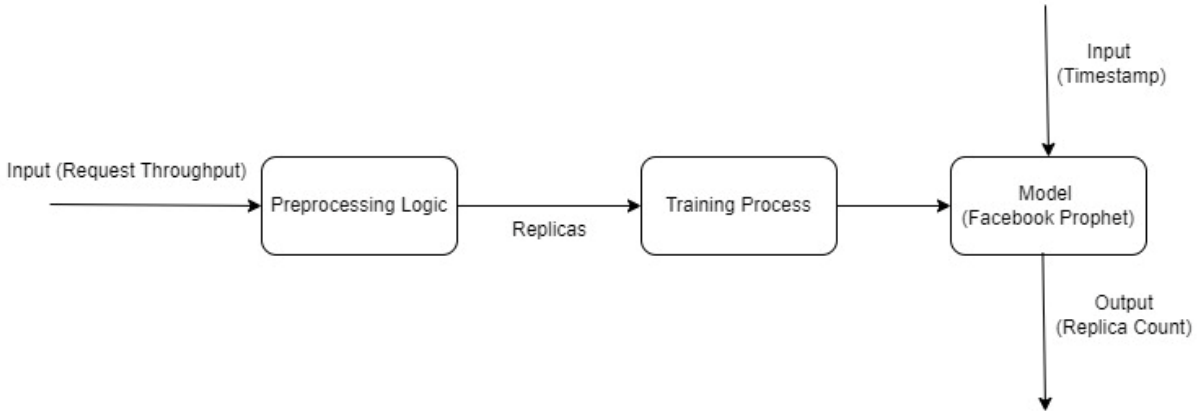

Figure 2: Model Training

Integration with KEDA

Once the model is trained, we will import the results into a CSV file, which will be imported into the Azure MySQL database. A ScaledObject will be created that can be used by KEDA which will use the data in the database to make scaling decisions.

Experiment Design

Note: I used a project's information on how to work with prophets quickly. This was to speed up the process and show the automatic scaling for Keda. All results are made by me, with usage of how to use prophet from the project. (contributes to Scaling Architecture and Cloud-Native learning outcomes). Source: https://github.com/YourTechBud/ytb-practical-guide/tree/master/predictive-autoscaling/k8s

Setup

I am using predefined values from a dataset online that gives a defined range of values over time, in such a way that there is easily a pattern found. This can be used with any data. I did change the timestamps. We will use Jupyter Notebook to display the results of the test easily.

Metrics

In the dataset, there are two columns.

- Timestamp: each row has a time interval of 5 minutes, lasting for 10 days.
- Value: This is the value throughput.

To give an example, here are the first rows of the file.

	timestamp	value
2	15/06/2024 00:00	248
3	15/06/2024 00:05	242
4	15/06/2024 00:10	236
5	15/06/2024 00:15	232
6	15/06/2024 00:20	225

Figure 3: dataset

Scenarios

The scenario is simple. We have a dataset that contains a total of 10 days of data, with intervals of 5 minutes in between. This load varies a lot and has a standard wave of usage. See it as in the morning there is a small spike, and during the evening there is a huge spike.

Observation Space

Timestamp: represents the time interval

Value (Throughput): represents the throughput of each timestamp

The observation space consists of the current state of the system, containing the timestamp with the corresponding throughput value.

Action Space

Number of Replicas (Pods): The number of output of the model, will be the amount of pods that are needed for each time stamp.

Action space is what the model can take in terms of actions, which means observing the throughput over time, based on the amount of replicas that are needed.

Results and Analysis

So first we want to visualize the dataset. So read the CSV file, convert it to readable data for the prophet algorithm, and specify some values. For the usage of 'throughput_per_replica' is the number where the performance of the pod will start to drop off, in our scenario will use the predefined one since we are using dummy data. See graph.

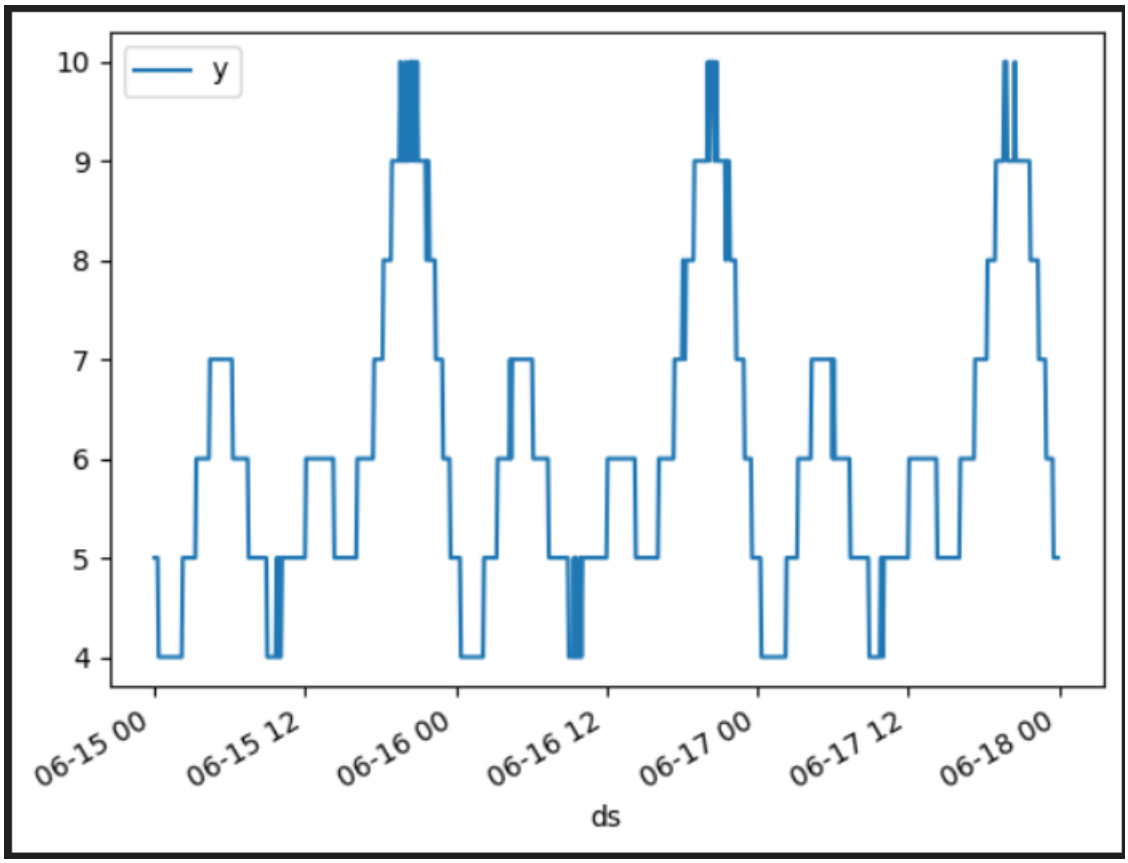

Figure 4: Dataset visualization

After this, we could just put in the data of the prophet object and start training it.

```
# Create and fit a prophet model
model = Prophet()
model.fit(df_train)
```

Figure 5: Triggers the Prohet model

After this, we can plot some results for the coming days.

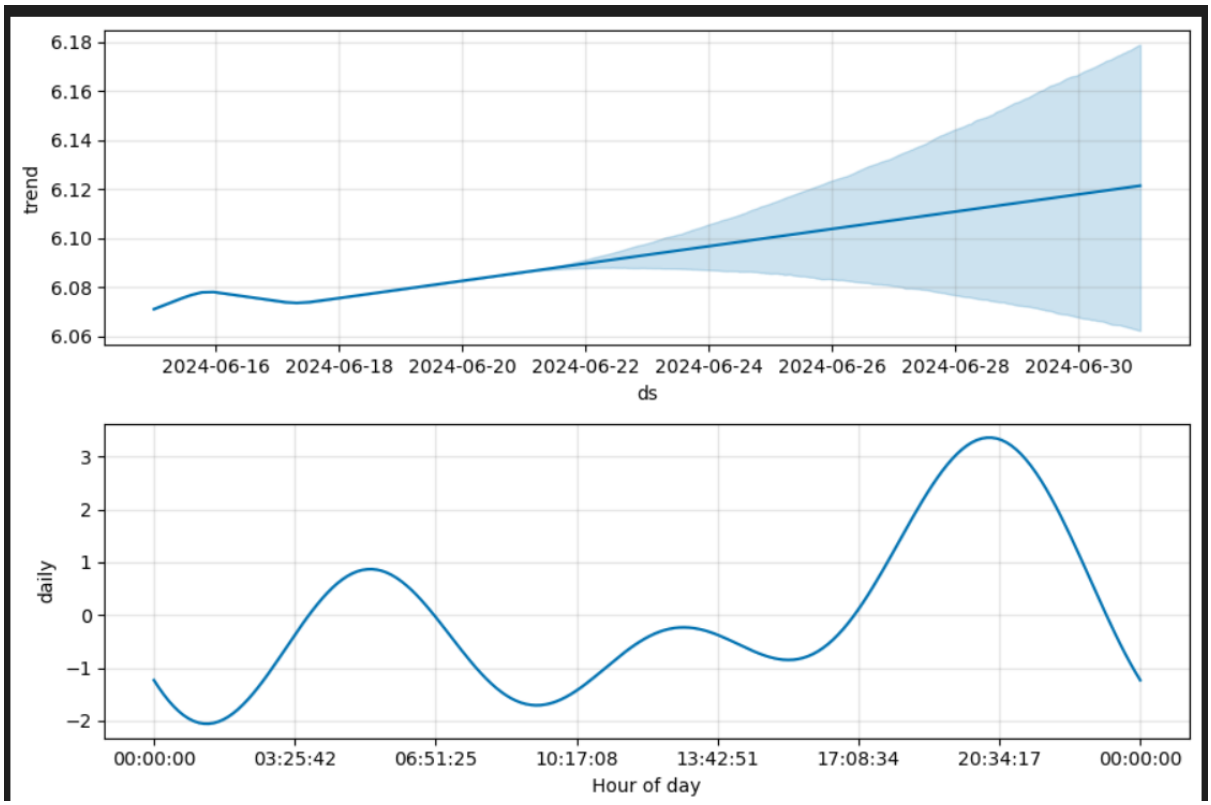

Figure 6: Plots from the outcomes of the trained model

In the first graph:

- X-Axis: ranging the date.
- Y-Axis: Trend component of the time series.

We have two things, the trend line, and the uncertainty interval.

- Trendline: The blue line shows the general trend of the data over the period. The trend is slowly increasing.
- Interval: The shaded blue area is the uncertainty level. This interval increases in wideness as we move further into the future, meaning there is more uncertainty in the trend prediction.

Second Graph:

This will show per day how the trend is going. So we can see in the morning there is a bit of action, and during the evening there is peak action.

After this we can view the predicted model.

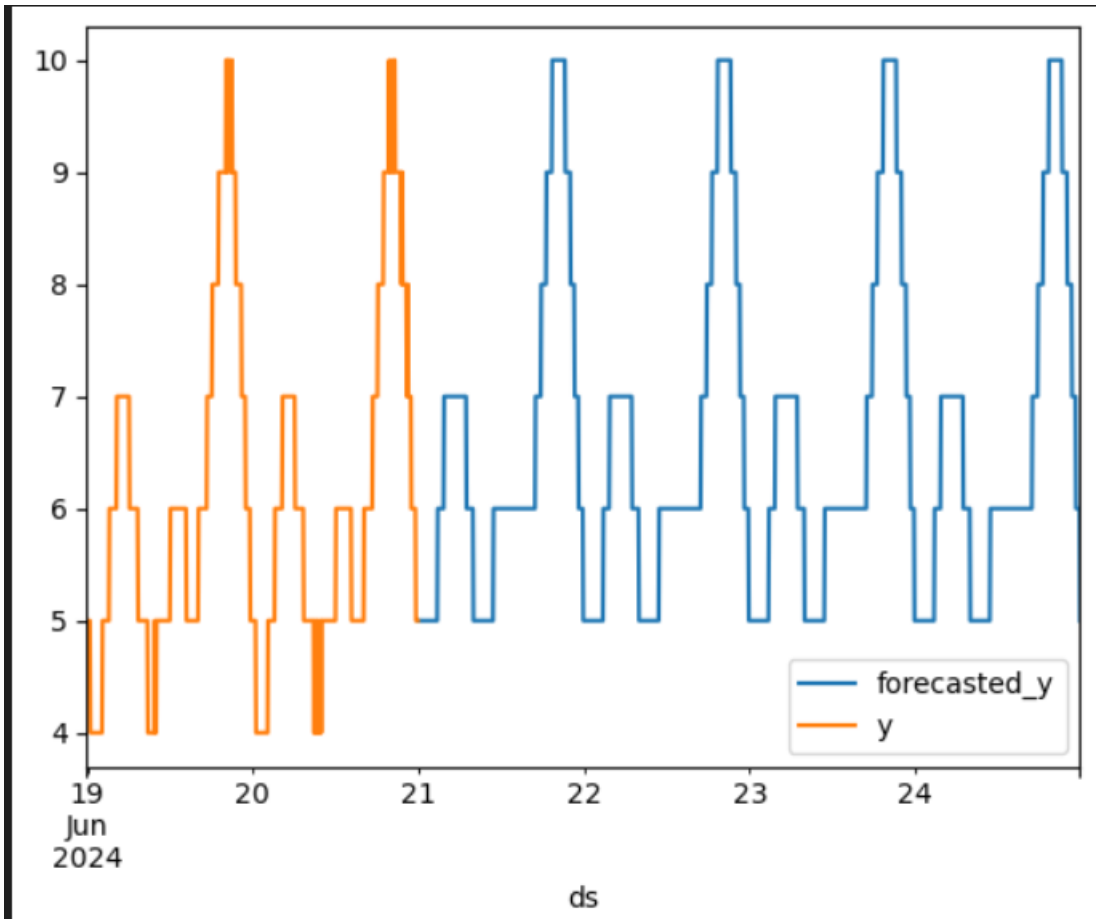

Figure 7: Predicted Results of Model

This shows the amount of pods needed over the forecasted days. As you can see it seems pretty accurate. After this, we want to calculate the root mean squared error.

```
RMSE: 0.72; 7.18%
```

Figure 8: Root Mean Squared Error Result

It indicates a value of 0.72 or 7.18% of the maximum actual value. This value summarized this model's performance. It will measure the differences between the predicted values by the model and the actual observed values. The best value is 0, meaning these predicted values can still vary from the actual results. Having the value 0.72 means that we are around one replica (pod) off. (Root Mean Squared Error (RMSE), n.d.)

Finally, the results can be saved in a prediction CSV file

	timestamp	value
2	24/06/2024 23:59	5
3	25/06/2024 00:04	5
4	25/06/2024 00:09	5
5	25/06/2024 00:14	5
6	25/06/2024 00:19	5

Figure 9: Predicted values.

Implementation Into Kubernetes

Now that we have the prediction values, we can use Keda to scale our application based on the current time we are in. First, we need to import the data to storage that can be accessed. I will import my data to MySQL Azure Database. There is a new database added with a table to my server called the following.

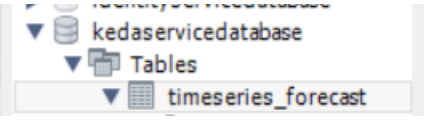

Figure 10: Database for Prediction values

Then I will import my CSV file into the table and all the data will now in the database.

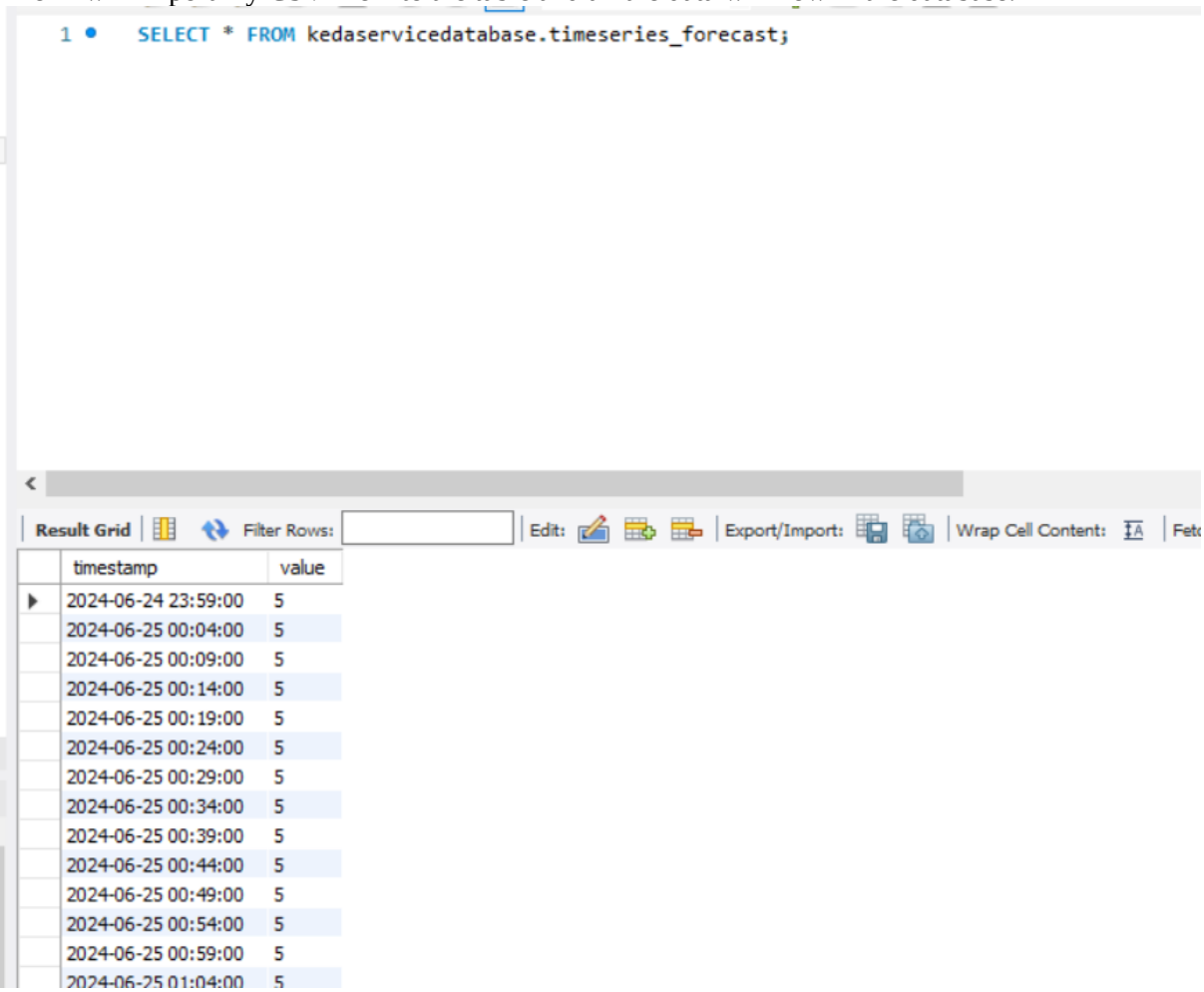

Figure 11: Data Import

After this, we can make use of ScaledObject in Kubernetes, which will define a job for Keda.

```yaml
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: identityapi-scaler
  namespace: default
spec:
  scaleTargetRef:
    name: identityapi
  pollingInterval: 10
  cooldownPeriod: 30
  maxReplicaCount: 10
  triggers:
    - type: mysql
      metadata:
        host: az-mysql-server-festivalconnect.mysql.database.azure.com
        port: "3306"
        dbName: kedaservicedatabase
        query: "SELECT MAX(value) FROM (SELECT value FROM timeseries_forecast WHERE timestamp > CURRENT_TIMESTAMP ORDER BY timestamp LIMIT 3) t;"
        queryValue: "1"
        targetQueryValue: "1"
        tagetValue: "1"
      authenticationRef:
        name: mysql-auth
```

Figure 12: ScaledObject Configurations

So it will scale the deployment of the "Identity Service" based on the following:

- We will connect to the MySQL database, where the data is stored of the prediction values.
- This will execute a query that will select the maximum forecasted value from the future timestamps in the 'timeseries_forcast' table, enabling Kubernetes to dynamically scale based on the workload.
- With the use of Limit 3, we will return at most 3 rows that meet the specified conditions. We will look a bit into the future because we do the condition where the timestamp in the MySQL database is greater than the current timestamp. Which will get forecasted values of Pods and adjust them in time, to change on the anticipated load.
- The polling interval is set to check the metrics or conditions are checked for autoscaling decisions.
- Cool down period is the delay that once an object has been scaled, it will not immediately change, but in this case, wait 30 seconds before any evaluation can happen again.

So then I deployed my ScaledObject which looks like this:

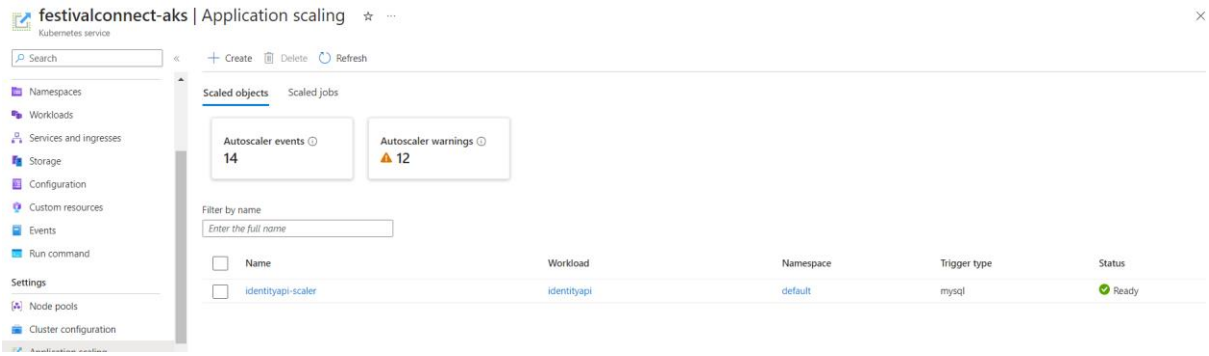

Figure 13: Scaled Object in Keda Inside Kubernetes Cluster

To further specify the current specification of the scaled object.

Figure 14: Scaled Object Details

This will further show the details of the current metrics and other details.

The result is that it will make 5 replicas (pods) based on the current metrics in time. So this is what Keda did, and it created my pods.

Name	Ready	Status	Restart count	Age	Pod IP	Node
identityapi-79758bc6f5-cdlml	1/1	Running	0	1 hour	10.224.0.65	aks-agentpool-21526954-vmss000000
identityapi-79758bc6f5-799dv	1/1	Running	0	1 minute	10.224.0.111	aks-agentpool-21526954-vmss000000
identityapi-79758bc6f5-mtkdh	1/1	Running	0	1 minute	10.224.0.18	aks-agentpool-21526954-vmss000000
identityapi-79758bc6f5-nljg5	1/1	Running	0	1 minute	10.224.0.69	aks-agentpool-21526954-vmss000000
identityapi-79758bc6f5-x2p62	0/1	Pending	0	1 minute		

Figure 15: Pods Creation Using Keda

It will now constantly check based on the data that is provided in the MySQL database. Therefore, we need constant information by training the model with new data, to eventually have a more accurate understanding of the amount of load that is needed. So in this case we need overtime deployments to the MySQL database and have proper monitoring to check if the model is getting trained correctly.

Performance Comparison

So by looking into event-driven autoscaling (Keda) with the usage of predicted data, I gained a new view of how scaling is also possible. With this approach we can scale the application before the application increases in load, preventing possible crashes. However, to accurately calculate this we need a lot of data over time to scale to demand.

With further analysis, if this is reliable we could do various comparisons to HPA, by comparing response times, and scalability, which one is more reliable,e and whether is it worth it to use this predictive model with the number of extra resources and time that is needed.

Conclusion

From the model training using Prophet on the dataset, we can conclude that Prophet can make accurate predictions based on previous events. Also by applying a future look into the predicted data, we can have enough pods to handle the incoming traffic. Nevertheless, the model needs to be frequently trained to keep it up-to-date. This can be done by pulling data from my monitoring applications such as Prometheus, to train the model, which we can then deploy data to MySQL database if the predicted data from the trained model is under a certain error margin. This all can be specified inside a CronJob. With the hypothesis statement, I can not make a concrete conclusion on this since I don’t have explicit values that say if one or the other is better, with for example using a p-value. Furthermore, what I can say is that, if used correctly, significantly improves the application performance. Which will benefit the application's availability and performance.

Future Work

Besides frequently updating the model and automatizing it using a CronJob. We can train different models with various data from specific services. So to make scaling more accurate for individual services, we use data from each service, to independently train them. This will improve the accuracy and limit wasted resources.

References

Autoscaling Workloads. (n.d.). Retrieved from kubernetes.io: https://kubernetes.io/docs/concepts/workloads/autoscaling/#:~:text=With%20autoscaling%2C%20you%20can%20automatically,the%20current%20demand%20of%20resources.

Codes, Y. (2023, 11 14). *I Fixed Kubernetes Autoscaling using Machine Learning | ft. Keda & Prophet*. Retrieved from youtube: https://www.youtube.com/watch?v=MhlkAivKkCw&t=1s

Kubernetes Event-driven Autoscaling. (n.d.). Retrieved from keda: https://keda.sh/#:~:text=KEDA%20is%20a%20single%2Dpurpose,functionality%20without%20overwriting%20or%20duplication.

Resende, D. (2023, 05 02). *Time-Series Forecasting With Facebook Prophet*. Retrieved from zerotomastery: https://zerotomastery.io/blog/time-series-forecasting-with-facebook-prophet/#:~:text=What%20is%20Facebook%20Prophet%3F,expert%20in%20time%20series%20analysis.

Root Mean Squared Error (RMSE). (n.d.). Retrieved from help.sap: https://help.sap.com/docs/SAP_PREDICTIVE_ANALYTICS/41d1a6d4e7574e32b815f1cc87c00f42/5e5198fd4afe4ae5b48fefe0d3161810.html

Singh, V. (n.d.). *What You Need to Know About Kubernetes Autoscaling*. Retrieved from deploy.equinix: https://deploy.equinix.com/blog/what-you-need-to-know-about-kubernetes-autoscaling/#:~:text=Cons%20of%20HPA,reducing%20performance%20for%20end%20users.

Why Is HPA Scaling Slow. (2023, 11 16). Retrieved from midbai: https://midbai.com/en/post/why-hpa-scale-slowly/

Jacobs, L. (2024). Monitoring Azure Deployment (Unpublished manuscript), FontysICT.