Project Ruby Acorn - Autoscaling a gaming service

Candidate number: 365

OsloMet

ACIT4410: Agile Service Delivery and Developer Operations

1. Discussion

The Ruby Acorn gaming company is currently struggling with managing the cloud-based gaming servers they host. The company has explored two solutions; dedicated engineers who manually scale the gaming servers, based on their player count. This solution was discarded, as engineers cost money and having them spend their time on tasks like this is not a good way for a company to spend their money. Dedicating engineers to simply watching the number of game servers could in some cases be useful, as it would allow the game servers to closely follow the trend of player count. However, as the goal of companies is to reduce inefficiencies and increase profits, other methods need to be explored. Another solution was to set a fixed number of servers, based on the time of day. Again, this solution was discarded. The number of players is usually dynamic, and, in many cases, this would not be optimal in cases where the player count would be unexpectedly high, or low. This would have great consequences for players, as there would be a lack of servers in some cases, and too many servers in other cases. There should be a continuous feedback loop when it comes to determining the number of game servers. Dedicated engineers could be considered this, where they constantly watch player count and determine an outcome, however, the inefficiency is too great for it be a legitimate solution.

Ruby Acorn must therefore explore other options when it comes to managing their game servers. As the company does not want to spend money on dedicated engineers, or having a fixed number of servers, finding a way to automatically scale the number of servers based on the player count could potentially be a solution. It would allow game servers to be scaled automatically, responding to increases and decreases in player count, while not having a large number of servers that are empty, or cases where there are too few servers. The company will allow a pilot implementation to be investigated, and asks three crucial questions: how much money could the company save by automatically scaling the game servers? How would the autoscaling method react to changes in player count? How would you measure if the autoscaling was successful?

There are a few important properties that the implementation should include. As the goal of any company is to create profit, the cost efficiency of the autoscaling method is an important aspect that should be focused on. Server instances cost money, so ensuring that there aren't a lot of unused instances when scaling up or down is important. The costs of the autoscaling method should at least be cheaper than the cost of using a fixed number of instances.

There should not be a great difference in the number of players and the number of server instances. The servers should be scaled dynamically, based on the current player count. For example, if the player count is 1000 at some point, with a capacity of 200 per game server, there should ideally be 5 server instances, not more or less. However, player numbers fluctuate often, and it should be acceptable to have a few servers too many, or a few short.

The autoscaling method should be able to scale the number of server instances without any hiccups or lag. It is not uncommon for games to have spikes in player counts when popular YouTubers or streamers play a game. The method should be quick in its autoscaling, so that players do not have to wait for long periods of time.

Ensuring that the autoscaling does not crash during the autoscaling would be crucial, as a crash would cause all players to not be able to play, and/or cause extra expenses. There shouldn't be lag or negative consequences for the players, as a result of the autoscaling.

For an autoscaling implementation to work, there are some technical milestones that should be followed. Firstly, data for the player counts needs to be gathered, so that we can use it in real time. A decision engine needs to be developed, which would decide if the server instances should be scaled up or down. There also needs to be ways for the player count and the number of server instances to be evaluated.

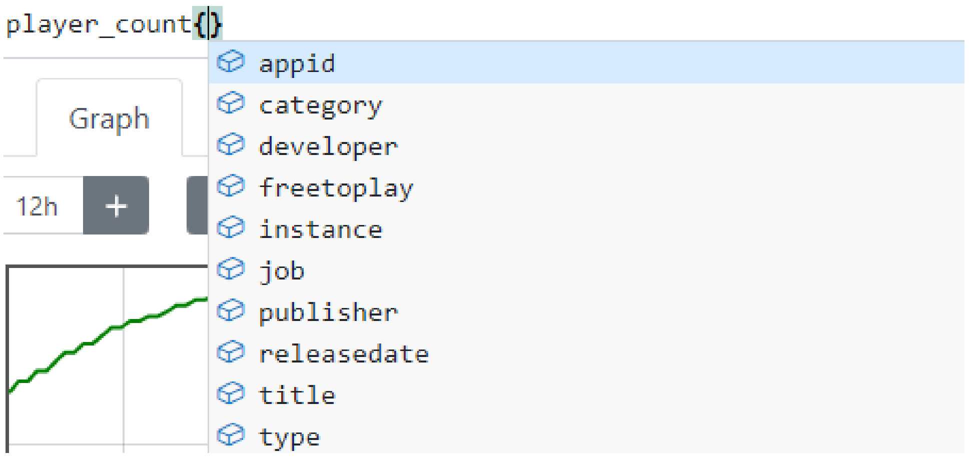
Autoscaling is a commonly used method within many sectors of industry, for automatically allocating or removing resources, based on the workload of a service. It is used for websites, where the number of active users often fluctuates depending on the day and time. For media streaming platforms such as Netflix, a predictive model is used in combination with autoscaling to determine the right number of instances in a time period [1]. It is essentially done to minimize costs and optimize a service, while allowing a company to focus their resources on more important aspects of their company. The inclusion of predictive models in the autoscaling could allow for more accurate allocation of resources compared to the number of activate users, while reducing inefficiencies such as downtime, ultimately resulting in greater user experience.

2. Technical design

In this section, we will go over the technical design of our implementation. It includes a description of methods used and why these were chosen. We will describe the actual pilot implementation in the next section.

In this project, we will use the OsloMet OpenStack cloud, which allows us to create up to 8 instances. Instances are virtual machines which are running on the cloud, which we can customize based on the use case of the project. We will also be using the OsloMet VPN so that we can connect to the virtual machines without being at campus.

Let us analyze the metrics data for player counts given at [2]. Each game has its corresponding line of data and is constructed as "player_count" with key-value pairs, such as "title" which states the title of the game, "appid" stating the Steam ID for each game, "category" indicating if the game is in the top 1000, or bottom 1000, depending on its player count. The data is structured, and it allows us to use different query options, depending on our needs. In this project, we will use the "title" key the most, as it provides a simple and precise way to pick the games we want to use.



```
player_count{}
```

```
appid
category
developer
freetoplay
instance
job
publisher
releasedate
title
type
```

Figure 0: Metrics for player_count

Data for player counts needs to be gathered in real time, which can be done with Prometheus. Prometheus is a useful tool for gathering time-series data. It would scrape the target, in this case, the game metrics, in desired intervals, such as every 30 seconds or every minute. Data for the virtual machine system such as CPU usage and number of Docker containers can also be scraped. Prometheus allows us to also use queries for the data, such as how long back we want to go for player count. This would allow us to use historical data, which could be useful for predictive models, as generally, there are patterns in the number of player count. However, it is also possible to simply use the data from the [2], as it will provide the current player count, refreshing every 5 minutes. In this project, we will look at solutions with both these methods.

To easily analyze player count and other features related to the project, a tool such as Grafana could be used to visualize the player counts and create dashboards and graphs with many customizations such as showing data from the last week, or month. The data used for Grafana is gathered by Prometheus and can easily be integrated. We can use query options to specify the games to display, such as displaying 3 games with different player counts, from the past month. Grafana also allows the user to view the cloud's virtual machines memory/CPU usage, and display alerts if the usages are too high, which could be useful in the moment the instance scaling happens.

We will use a Prometheus-Grafana stack that runs on Docker services. The stack consists of Prometheus, Node-exporter, Alertmanager, Cadvisor and Grafana. Prometheus allows us to get information about services going on, which we can specify depending on what we want to scrape. In this case, Prometheus will be used to scrape data for player count and various system information. Node-exporter shows information about the system, which can be scraped by Prometheus. Alertmanager displays alerts for when various systems and processes, such as CPU/memory usage, become too high or if processes stop working. Cadvisor displays information about the Docker containers, such as their CPU/memory/size which can be scraped by Prometheus. Grafana, as mentioned, is the tool we will use to display graphs of system processes and services that are used for the project.

A decision engine must be developed, which would use the player count data to decide if the number of server instances should be scaled up or down. A simple algorithm for deciding the number of server instances needed could be to divide the current player count by the server instance capacity, and round the number up. For example, if the player count is 14000 with a capacity of 3000, the number of server instances required would be 4.667, rounded to 5. This method would ensure that there are enough servers when the servers are scaled. However, there are some flaws with this decision algorithm. First of all, it simply reacts to the current player count. If the player count increases or decreases greatly over a period of time, there could be wasted resources because there are either not enough servers, or too many servers.

Another problem is that it doesn't account for large fluctuations in player count. Whenever the player count spikes, the algorithm would scale all the servers at once. This could potentially cause lag and instability. To counter these issues, other decision algorithms should be explored, such as predictive models that would use historic data to better understand how player count changes. Some models that could be used are regression models, which predict a target based on some independent variables. Predictive models could be useful, as games tend to have patterns. It is common for player count to be higher on Fridays and the weekends, and at certain time periods.

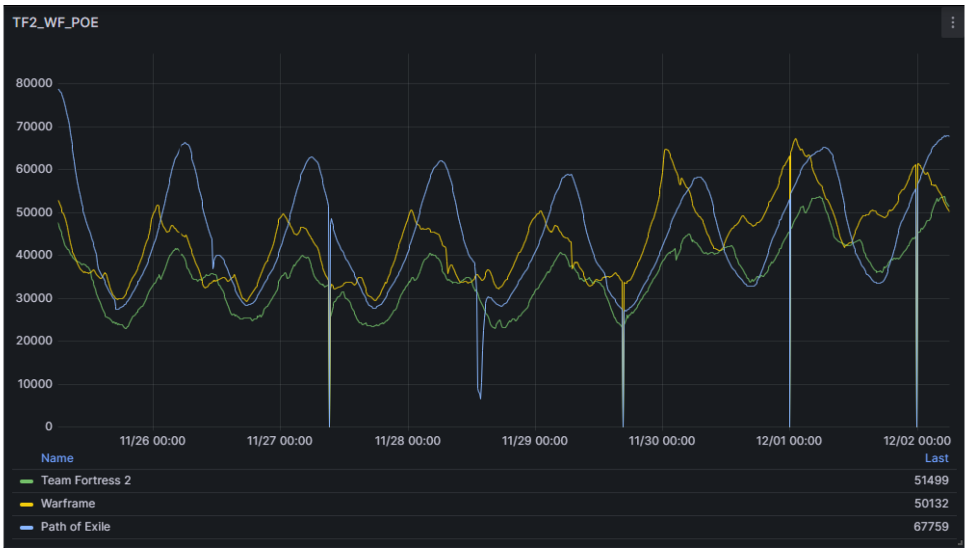


Figure 1: Player count for 3 different games, one week span

As shown in Figure 1, all 3 games seem to have higher player counts for the mentioned days. However, the greatest indicator of player count is the time of day, with all games having spikes at roughly the same time periods. This indicates that predictive models could be useful for this project. A decision algorithm with predictive methods would create the game servers before the actual player count reaches the player count that was predicted.

The actual autoscaling should be done on Docker containers, to replicate the environment of actual server instances, and scaled on a couple of virtual machines. This is done because it is usually expensive to launch server instances. Docker is a great way to test your program before eventually doing a real implementation with actual servers. Docker has many features that fit well for this project, such as the services and swarm. We will use Docker swarm to run the service, where the Docker containers are seen as the game server instances. Using Docker containers to test the autoscaling is a simple and effective method, as it allows you to test it with various decision algorithms with no expenses or a system breaking.

We use Docker swarm which is a Docker feature that allows for manager and workers nodes. Manager being the node “in charge”, that can start/remove Docker services and manage them. We add all the cloud servers/virtual machines as workers and run a service where we can scale the number of Docker containers based on the output of the decision algorithm. When we run the algorithm, the containers are automatically scaled, and because all cloud virtual machines are in a manager/worker relationship, the containers are automatically load balanced. For example, if there are currently 20 server instances, with 4 cloud virtual machines connected together, the load balancing would balance it to 5 Docker containers each. This is an efficient method that allows for more Docker containers to be scaled, and when there are heavy CPU and memory loads on the virtual machines.

When the scaling starts, it should preferably not scale all the Docker containers instantly, as this can be quite resource intensive, and it wouldn’t present the ideal scenario where there are little to no wasted resources. For example, if the player count is 1000 at hour 1, and the prediction model predicts the player count to be 2000 at hour 2, then instantly scaling all the containers at hour 1 would mean that some of them would be empty until the player count reaches its predicted value at hour 2. To solve this problem, we could implement a method so that the containers are scaled gradually, such as a quarter or a third of all the containers are scaled every 5 minutes, for example. The Docker containers do not actually run any code or program inside them, so we need a way to prevent them from exiting, such as an infinite loop.

The autoscaling should be done for games with different player count profiles, so that we can test how the autoscaling reacts to different profiles. If we look at the player count graphs implemented in Grafana, we can see that certain games have more dynamic player counts than others. This would allow us to easily determine how well the autoscaling works. If the game servers do not follow the player count, such as there being too many, or too few servers, this should be addressed.

Because we want to automatically scale the number of Docker containers, there should be some method implemented that would run the script at a regular interval. The interval chosen is quite important, as it will directly impact the cost of the entire autoscaling setup, and the server instances themself. If we have a really fast interval, such as 5 or 10 minutes, the server instances could be more accurately aligned with the player count, because they are more frequently being added, or removed. However, scaling more frequently could affect player experience negatively, and the cost of the scaling itself. A longer interval such as 30 minutes could be a better choice, as the scaling will become less frequent, reducing potential costs and issues such as high CPU and memory usage.

The program should be able to handle any and all errors, or unexpected behavior, such as the website being down, the Docker service/API not working, automatically executing the script not working, etc. Many things can go wrong during autoscaling, and for a successful 24 hours to happen, it is crucial that everything works as it should, and if it does not, there should be back-up plans just in case. For example, if there is a predictive model implemented that starts predicting with massive errors, there could be another scaling option that would look at the actual player count and determine the number of game servers. If this method would also not work anymore, there could be a "final" method where the number of game servers is simply set to a number, using the average number of players for a period.

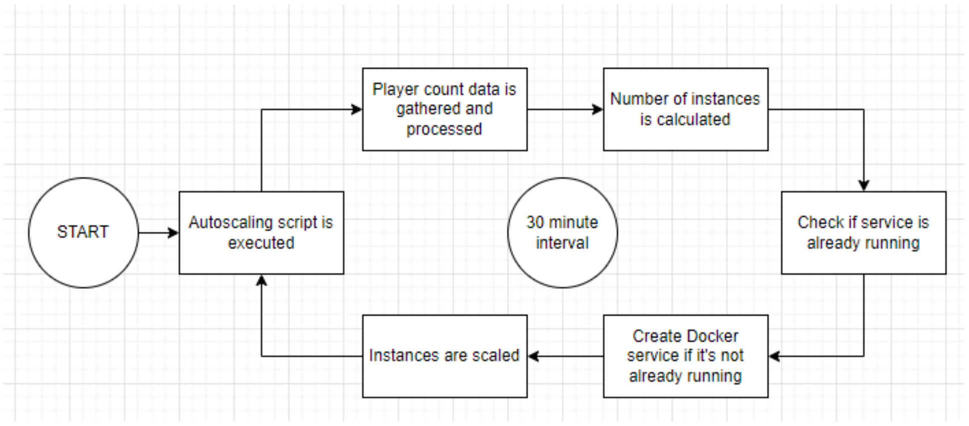
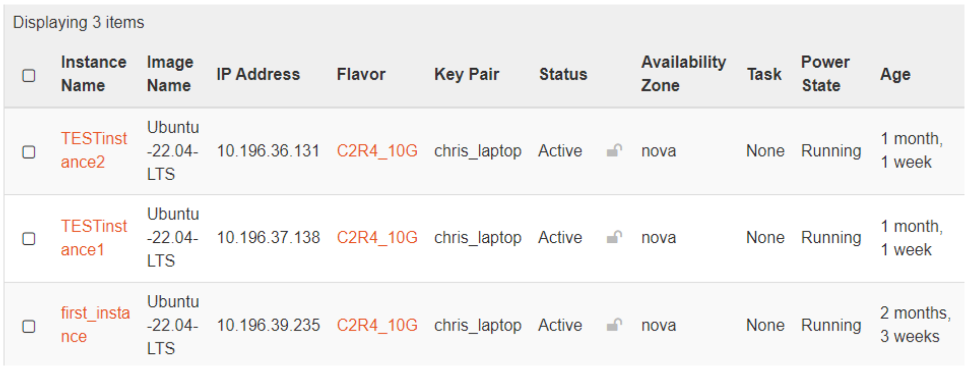

START
Autoscaling script is executed
Player count data is gathered and processed
Number of instances is calculated
Check if service is already running
Create Docker service if it's not already running
Instances are scaled
30 minute interval

Figure 2: Diagram of autoscaling implementation

3. Pilot implementation

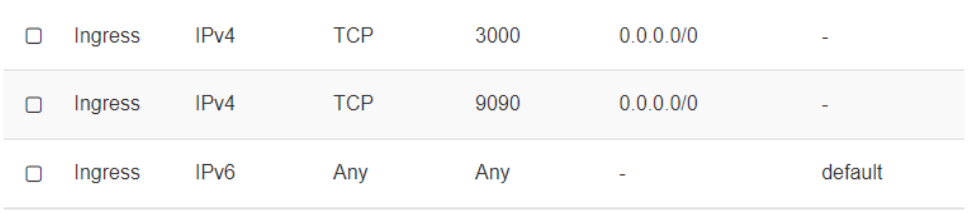
The implementation will be running on a cloud virtual machine, which is provided by OsloMet. Originally, we had only one virtual machine running, but we created two more to enable the use of load balancing for the Docker containers. The virtual machines are created through the OpenStack dashboard, which allows the user to create and manage the virtual machine instances, change network configuration and ports, access API features, among other things. We have a total of 3 virtual machines running, with the "first_instance" being the main VM used to implement the project. The VMs image is, Ubuntu 22.04 LTS which is a commonly used Linux distribution. Image is essentially the operating system which the virtual machine will be running. The key pair indicates the ssh key used, which is chris_laptop, generated from the local Linux environment. The same key pair is used for all 3 servers, which would not be ideal in real world scenarios, as a compromised key could be used to gain access to all the servers with the key, however, it is assumed that the OpenStack cloud is secure. Additionally, the VMs do not contain any sensitive information that someone could exploit. The two remaining VMs do not contain many files; their main purpose is to load balance the Docker containers.


Displaying 3 items

	Instance Name	Image Name	IP Address	Flavor	Key Pair	Status		Availability Zone	Task	Power State	Age
☐	TESTinstance2	Ubuntu-22.04-LTS	10.196.36.131	C2R4_10G	chris_laptop	Active		nova	None	Running	1 month, 1 week
☐	TESTinstance1	Ubuntu-22.04-LTS	10.196.37.138	C2R4_10G	chris_laptop	Active		nova	None	Running	1 month, 1 week
☐	first_instance	Ubuntu-22.04-LTS	10.196.39.235	C2R4_10G	chris_laptop	Active		nova	None	Running	2 months, 3 weeks

Figure 3: Display of running instances

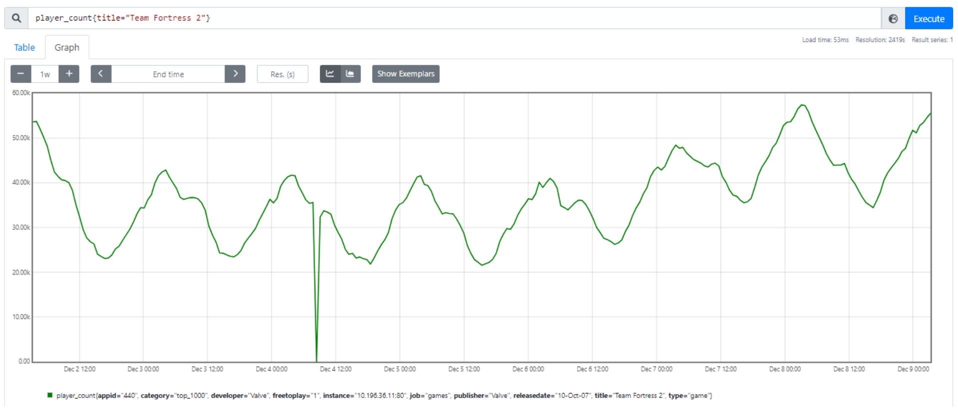
Security groups are responsible for providing a layer of security to the network of the cloud servers. For this project, the use of Grafana and Prometheus is important, and the ports for which these services are on must be added to the security group. On port 3000, the Grafana services can be viewed, and is essential in the implementation of this project. Port 9090 allows us to view Prometheus' services, such as viewing the current targets that are up and all the metrics which Prometheus has scraped.



☐	Ingress	IPv4	TCP	3000	0.0.0.0/0	-
☐	Ingress	IPv4	TCP	9090	0.0.0.0/0	-
☐	Ingress	IPv6	Any	Any	-	default

Figure 4: Ports for Grafana and Prometheus

The service also has a Graph feature, allowing the user to view graphs with historical data for several metrics, such as the game metrics. The project could technically be done with the Prometheus services only, as the graphs will display all the metrics which Grafana will use. However, Grafana allows for far greater customization where all the metrics and information can be displayed at the same location without having to enter multiple queries and values. As shown in the figure below, the query is entered 'player_count, with its game title, displaying player count the last week.



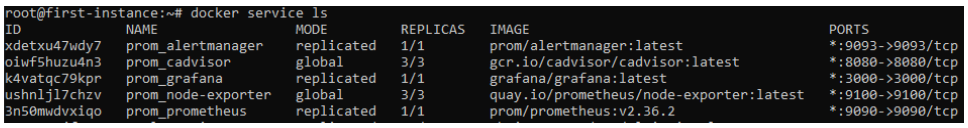
```
player_count{title="Team Fortress 2"}
```

```
player_count{appid="440", category="top_1000", developer="Valve", freetoplay="1", instance="10.196.36.11:80", job="games", publisher="Valve", releasedate="10-Oct-07", title="Team Fortress 2", type="game"}
```

Figure 5: Graph of historical data over player count

We are using Windows Subsystem for Linux, which allows us to run Linux on a Windows machine. The ssh key was generated from the subsystem, which is the ‘chris_laptop’ key. When connecting to a particular virtual machine, we use ssh @ubuntu VM IP. It will require a password to connect to the VM, which is the password of the ssh key. Once connected to the VM, we can use ‘sudo su’ to become root superuser, allowing us to use administrative commands on the VM.

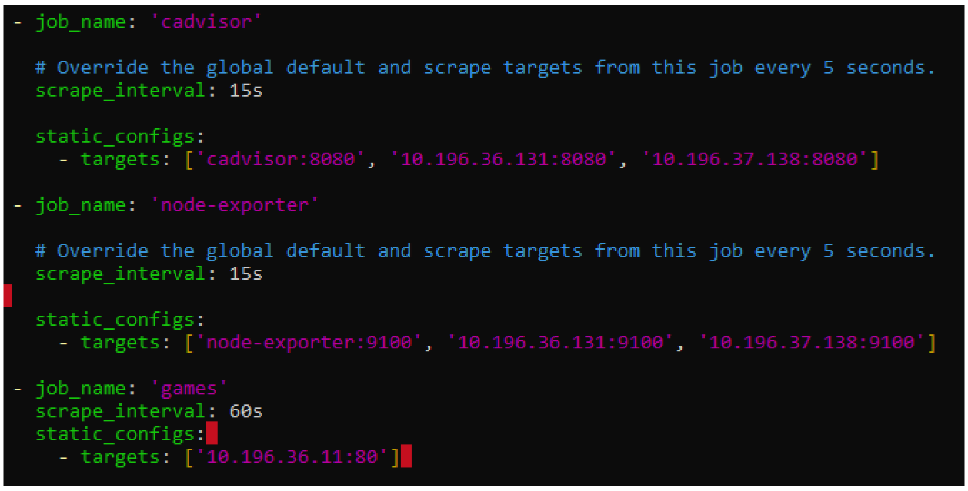
Firstly, we need to set up the Prometheus-Grafana stack cloned from [3]. Once the Prometheus Grafana stack has been set up, their status can be displayed as shown in the figure. Originally, the services only contained a single replica each, however, our implementation is running 3 nodes, which is why Cadvisor and Node exporter has 3 replicas instead of 1. On the worker nodes, only Docker was installed.



```
root@first-instance:~# docker service ls
ID             NAME               MODE         REPLICAS   IMAGE                                     PORTS
xdetxu47wdy7   prom_alertmanager  replicated   1/1        prom/alertmanager:latest                  *:9093->9093/tcp
oiwf5huzu4n3   prom_cadvisor      global       3/3        gcr.io/cadvisor/cadvisor:latest           *:8080->8080/tcp
k4vatqc79kpr   prom_grafana       replicated   1/1        grafana/grafana:latest                    *:3000->3000/tcp
ushnljl7chzv   prom_node-exporter global       3/3        quay.io/prometheus/node-exporter:latest   *:9100->9100/tcp
3n50mwdvxiqo   prom_prometheus    replicated   1/1        prom/prometheus:v2.36.2                   *:9090->9090/tcp
```

Figure 6: Prometheus-Grafana stack services

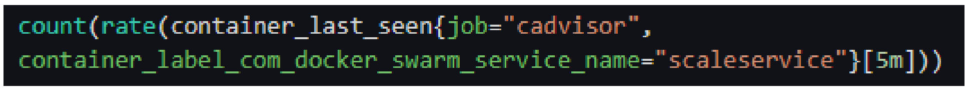
As mentioned, we are running our service in a Docker swarm, which allows for multiple nodes to be connected. Figure 7 shows the inclusion of our two additional VM nodes in the “targets” section for both Cadvisor and Node-exporter. This allows Grafana to use data across all nodes, which is important as we want to have an accurate representation of the number of containers. In addition to the previously mentioned services, a job is added, “games” for Prometheus to scrape data from the game metrics.



```
- job_name: 'cadvisor'

  # Override the global default and scrape targets from this job every 5 seconds.
  scrape_interval: 15s

  static_configs:
    - targets: ['cadvisor:8080', '10.196.36.131:8080', '10.196.37.138:8080']

- job_name: 'node-exporter'

  # Override the global default and scrape targets from this job every 5 seconds.
  scrape_interval: 15s

  static_configs:
    - targets: ['node-exporter:9100', '10.196.36.131:9100', '10.196.37.138:9100']

- job_name: 'games'
  scrape_interval: 60s
  static_configs:
    - targets: ['10.196.36.11:80']
```

Figure 7: Prometheus.yml configuration for Cadvisor and Node-exporter

In Grafana, we create dashboards for the games we want, such as “Warframe”, “Team Fortress 2”, “Path of Exile”. We can also include the number of Docker containers and how much CPU/memory is used by all the containers. To display the number of Docker containers for our main service “scaleservice”, we can use the following code in Grafana’s metric browser. It uses the job Cadvisor, which scrapes the data on the running Docker containers, for the last 5 minutes. This allows us to view how the autoscaling works, by looking at the number of Docker containers and seeing if it fits with the current player count. When creating the dashboard, we create it as a time series graph, as this will allow us to see how the number of containers changes based on the time of day.

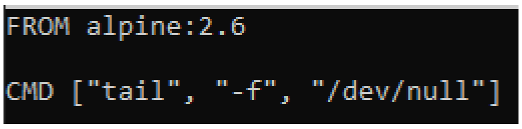


```
count(rate(container_last_seen{job="cadvisor",
container_label_com_docker_swarm_service_name="scaleservice"}[5m]))
```

Figure 8: Line of code to show containers

The entire project is in a folder “finalproj” which consists of a Dockerfile and the app-folder which contains the Python scripts, “scaledocker.py” for the first scaling method and “histscale.py” which will use historical data and a predictive model. The Dockerfile is very simple; it only contains a command which makes the Docker containers do not exit, Figure 9. We do this because the Docker containers are not supposed to run anything to begin with, and Docker containers are made so that they exit when they have finished the commands given to it. The command ensures that the Docker containers will run infinitely, and the only way to remove them is by removing them individually or removing the entire service.

The image chosen is alpine, which is a lightweight image, just under 5 MB in size. The version chosen is 2.6, which is the oldest version. Newer versions are slightly larger, which would increase the computational resources required, which could cause issues when scaling a large number of Docker containers. To build the image, we use ‘docker build’, and then use the image in the containers where they will execute the commands given to it. Because we are running the Docker swarm with multiple nodes, we can use the Docker Hub repository to store the built image. Then, when the scaling script starts, it will automatically pull the image from the repository where it is stored. This way, we don’t need to create individual images for all the nodes, as all nodes will have the same image. This is useful for cases where images are changed frequently and the number of nodes on the Docker swarm is high, as the user does not have to manually change the images.

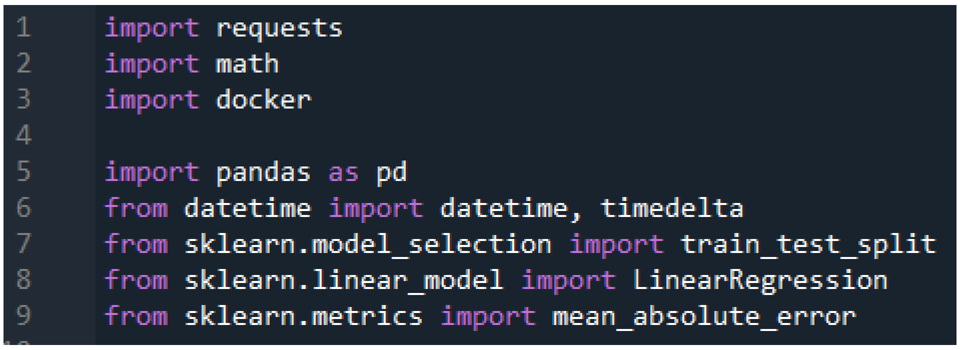


```
FROM alpine:2.6

CMD ["tail", "-f", "/dev/null"]
```

Figure 9: Dockerfile for containers

We use many libraries in this project, as they are great tools and allow us to write more efficient code. We will use the requests library to get the data from the source for our initial implementation. Math library will be used for calculating the number of server instances required. Logging helps us find any errors which have occurred when running the code, which is useful for troubleshooting if something goes wrong. The Docker library allows us to use Docker API in Python, which is useful because we must be able to manage containers and services through the Python script.

Additionally, we will use pandas and sklearn to process the historical data which will be used for our second autoscaling method. The pandas library is a useful tool for analyzing data and processing it through data frames. Finally, we import various functions from the sklearn, as we will use a linear regression model to predict player count.

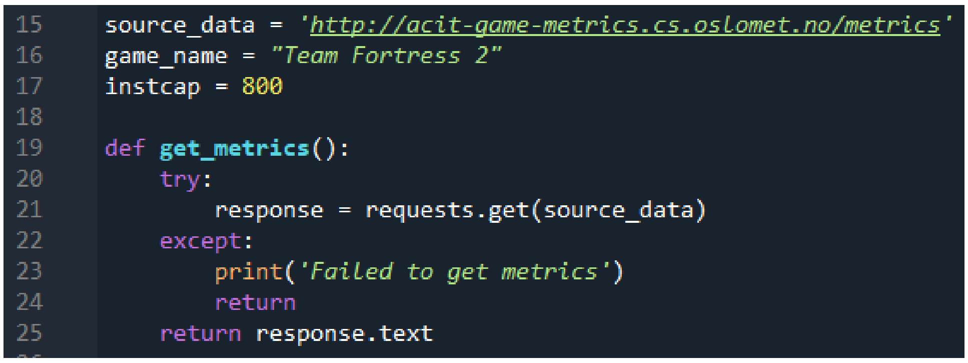


```python
import requests
import math
import docker

import pandas as pd
from datetime import datetime, timedelta
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error
```

Figure 10: Libraries used

We need to get the data from the game metrics target that is being scraped by Prometheus. The data is being scraped every 60 seconds, and it is considered “live” despite being from 2017, because it changes every 5 minutes.

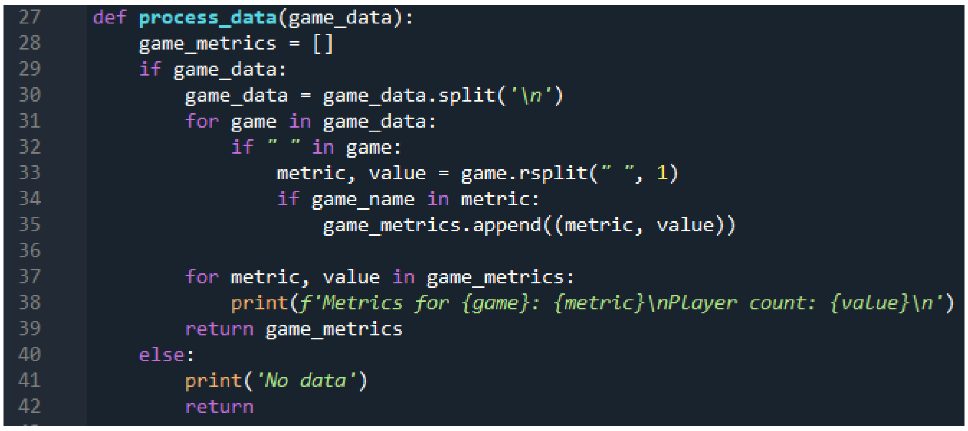
Firstly, the source is defined, which is the address to the game metrics. We define a variable for instance capacity, which was set to 800. This was chosen so that there aren’t hundreds of containers running at the same time, as this seemed to slow down the program. The requests.get sends a get request to the source and if the connection is established to the target, we return the text found on the source. This would return the entire page and will therefore need to be processed so that we can get the value we want, which is the player count of the chosen game.



```python
source_data = 'http://acit-game-metrics.cs.oslomet.no/metrics'
game_name = "Team Fortress 2"
instcap = 800

def get_metrics():
    try:
        response = requests.get(source_data)
    except:
        print('Failed to get metrics')
        return
    return response.text
```

Figure 11: Function to get metrics for games

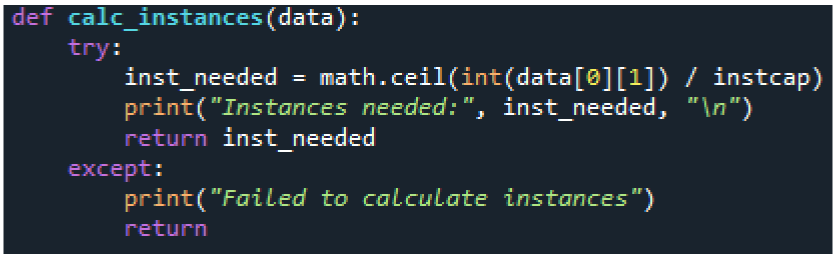
If the game data has been obtained from the previous function, we can start to process it. The function takes the game data as a parameter. Firstly, because each game is on its own line in the game metrics data on [2], the lines must be split into individual items in a list, to more easily process it further. A for-loop iterates over all the games in the list, and checks if the game has a space in its line. If this is the case, the line is split into two variables. They are split by using the rsplit function, which splits the line, starting from the right side, based on the space found in the line. If the game name we specified is found in the metric variable, which contains information about the game, including game title, the metric and player count will be stored in a list. The metrics and player count are now separated, allowing for easier calculation of the instances required.



```python
def process_data(game_data):
    game_metrics = []
    if game_data:
        game_data = game_data.split('\n')
        for game in game_data:
            if " " in game:
                metric, value = game.rsplit(" ", 1)
                if game_name in metric:
                    game_metrics.append((metric, value))

        for metric, value in game_metrics:
            print(f'Metrics for {game}: {metric}\nPlayer count: {value}\n')
        return game_metrics
    else:
        print('No data')
        return
```

Figure 12: Function to process game data

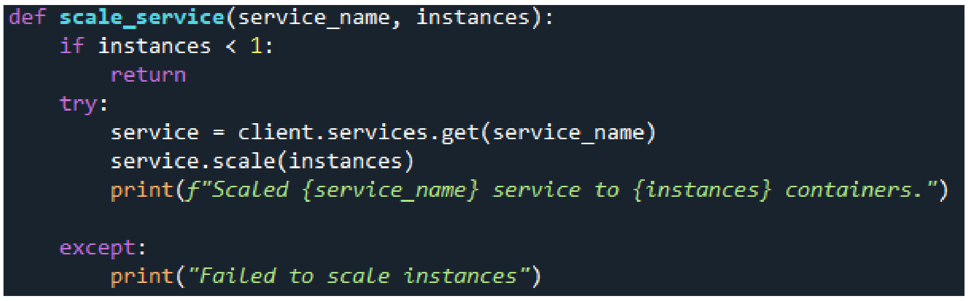
Now that the player count is easily available, the instances required can be calculated. If this data exists, we can calculate the number of instances by dividing the player count by the capacity of each instance, and then applying a ceiling function to it, which rounds the number up. This is necessary because there can't be decimal numbers used for the number of instances. Rounding up is also important in regard to the servers and players, as it will allow other players to join the server until it is full, at which point, the number of instances would increase.



```python
def calc_instances(data):
    try:
        inst_needed = math.ceil(int(data[0][1]) / instcap)
        print("Instances needed:", inst_needed, "\n")
        return inst_needed
    except:
        print("Failed to calculate instances")
        return
```

Figure 13: Function to calculate number of instances

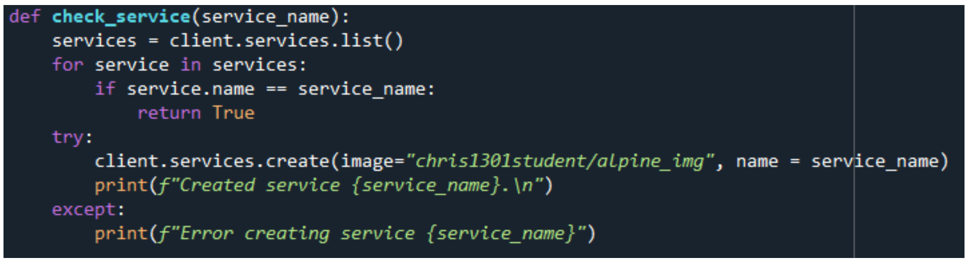
Once the number of instances has been calculated, the scaling of Docker containers can begin. A variable is created which stores information about the service, such as its name, ID, number of containers. The scaling is a simple process, only requiring the number of instances to be scaled to. It scales all the containers at once, which would not be ideal, especially when there are huge spikes in the player count. If the calculated number of instances is equal to the current number of Docker containers, the function will not scale up or down. This means that we do not need to check if the calculated instances are higher or lower than the current containers. We also don’t need to check the current number of Docker containers, as the scale command will simply scale the value given to it. This is a simple and effective method for scaling containers that are in a Docker service.



```python
def scale_service(service_name, instances):
    if instances < 1:
        return
    try:
        service = client.services.get(service_name)
        service.scale(instances)
        print(f"Scaled {service_name} service to {instances} containers.")

    except:
        print("Failed to scale instances")
```

Figure 14: Function to scale the service

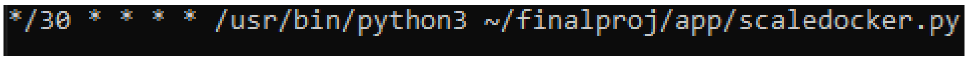
Each time the program runs, it will check the service which runs the Docker containers if it has been created. The function takes the name of the service as the parameter. Using the Docker API, we can get the list of all the currently running services. If the service is already running, nothing will happen to the current service, it will simply proceed to the scaling function. If there is no service running with the specified name, one will be created. The image used is alpine, which is minimal and usable in this project, as the Docker containers will not be running any code themselves.



```python
def check_service(service_name):
    services = client.services.list()
    for service in services:
        if service.name == service_name:
            return True
    try:
        client.services.create(image="chris1301student/alpine_img", name = service_name)
        print(f"Created service {service_name}.\n")
    except:
        print(f"Error creating service {service_name}")
```

Figure 15: Function to check if Docker service is running

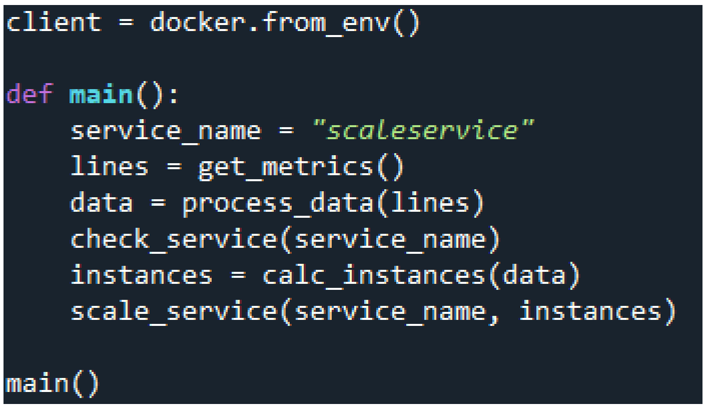
The scripts contain all the necessary functions and code to do the autoscaling, so we can run it at regular intervals. In Figure 16, we specify that the scaledocker.py file should be executed every 30 minutes, indicated by the asterisk, backslash and 30. The remaining asterisks can be used to run scripts at specific hours or days. The path to the Python interpreter must also be specified. This allows the script to be executed even when the computer is off, as it is running on the virtual machine itself, which is always running as well, unless shut down by the user.



```
*/30 * * * * /usr/bin/python3 ~/finalproj/app/scaledocker.py
```

Figure 16: Crontab command to run script at regular intervals

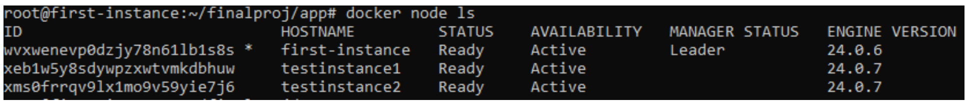
Finally, the main function can be called, which executes all the necessary functions to perform the scaling. Before the main function is run, the client which connects to the Docker daemon must be created so that the Docker related commands can be executed.



```python
client = docker.from_env()

def main():
    service_name = "scaleservice"
    lines = get_metrics()
    data = process_data(lines)
    check_service(service_name)
    instances = calc_instances(data)
    scale_service(service_name, instances)

main()
```

Figure 17: Main function

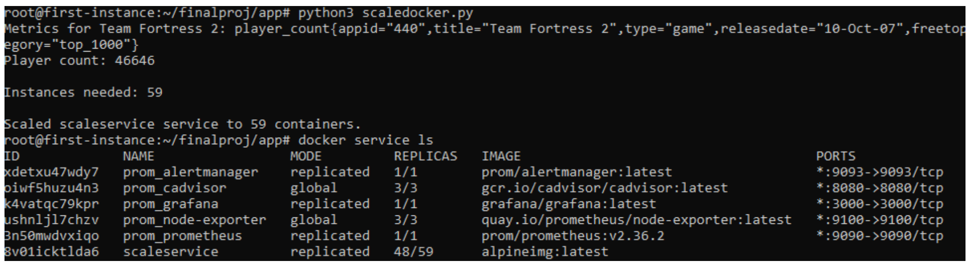
There are 3 nodes in the Docker swarm, with the “first-instance” being the leader and can manage the Docker services. This allows for load balancing between every node, so that more Docker containers can be scaled. Only the leader node can manage the services, the worker nodes can only manage containers on their own node.



```
root@first-instance:~/finalproj/app# docker node ls
ID                            HOSTNAME         STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
wvxwenevp0dzjy78n61lb1s8s *   first-instance   Ready     Active         Leader           24.0.6
xeb1w5y8sdywpzxwtvmkdbhuw     testinstance1    Ready     Active                          24.0.7
xms0frrqv9lx1mo9v59yie7j6     testinstance2    Ready     Active                          24.0.7
```

Figure 17: Overview of Docker swarm nodes

The script can be manually run as in the figure below, which would display the metrics for the game chosen, its player count, and the number of instances required. Note that the message indicating that the service has been scaled is not completely accurate, as the actual scaling of Docker containers can take some time, after the script has finished. The more Docker containers are scaled at once, the longer the process will take. Generally, scaling a handful of containers does not take long, especially since the containers are not running any code except the infinite loop. We can also easily view the number of containers in the scaling service because it is a service. This is useful for testing purposes, as we don’t have to run other scripts or use Grafana to view the current number of Docker containers. In this example, the instance capacity is 800 players.



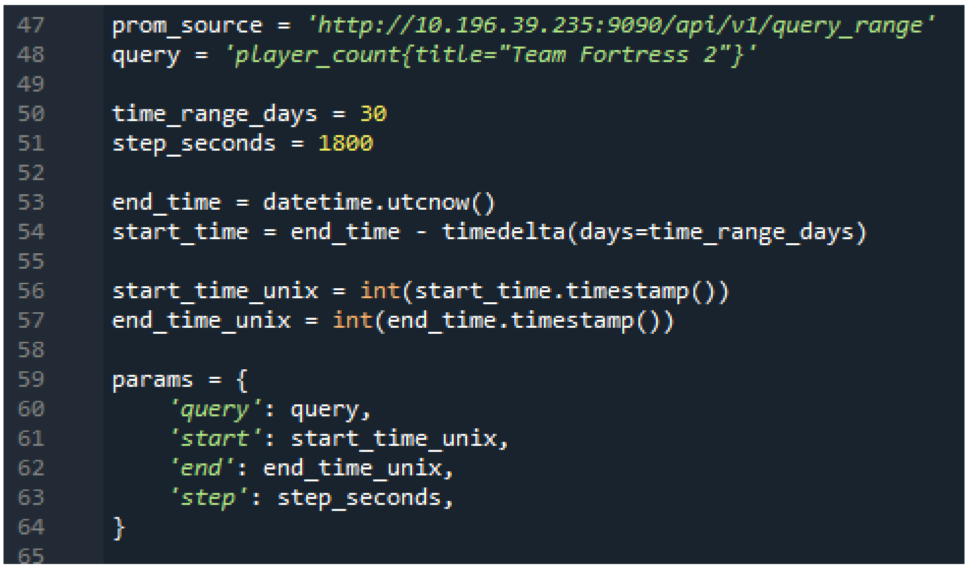
```
root@first-instance:~/finalproj/app# python3 scaledocker.py
Metrics for Team Fortress 2: player_count{appid="440",title="Team Fortress 2",type="game",releasedate="10-Oct-07",freetop
egory="top_1000"}
Player count: 46646

Instances needed: 59

Scaled scaleservice service to 59 containers.
root@first-instance:~/finalproj/app# docker service ls
ID             NAME                 MODE         REPLICAS   IMAGE                                     PORTS
xdetxu47wdy7   prom_alertmanager    replicated   1/1        prom/alertmanager:latest                  *:9093->9093/tcp
oiwf5huzu4n3   prom_cadvisor        global       3/3        gcr.io/cadvisor/cadvisor:latest           *:8080->8080/tcp
k4vatqc79kpr   prom_grafana         replicated   1/1        grafana/grafana:latest                    *:3000->3000/tcp
ushnljl7chzv   prom_node-exporter   global       3/3        quay.io/prometheus/node-exporter:latest   *:9100->9100/tcp
3n50mwdvxiqo   prom_prometheus      replicated   1/1        prom/prometheus:v2.36.2                   *:9090->9090/tcp
8v01icktlda6   scaleservice         replicated   48/59      alpineimg:latest
```

Figure 18: Manual execution of autoscaling script

A general issue with only using the current player count as a way to determine the number of game servers to scale to is that it cannot account for what the player count will look like in the future. Historical data is especially important for games, as player count often has patterns, as shown earlier. As such, implementing a predictive model could be able to preemptively allocate game servers so that players can instantly start playing, instead of having to wait. There is a plethora of predictive models that could be used. A simple and commonly used model is linear regression, which calculates a dependent variable, which would be the player count in this case, based on some independent variables, which would be the time on time and the day of the week. There could be more variables implemented as well, such as a holiday feature or some special event. However, we will only be using time of day and day of week for our implementation, as these are obvious and strongly correlated features, shown earlier in Figure 1, on historical player count in the games.

In Python, we can get the historical data from Prometheus by defining variables which will determine how far we want to go back for the data, and how frequently the data will be shown. As we are using historical data, we want to have as much data as possible, as this will give the model more information on how the player count behaves, which would create a model that can predict future player count better, based on the independent variables. Data on the player count for all games seems to only be available from 2 weeks ago until the current time, which is not a lot of data.

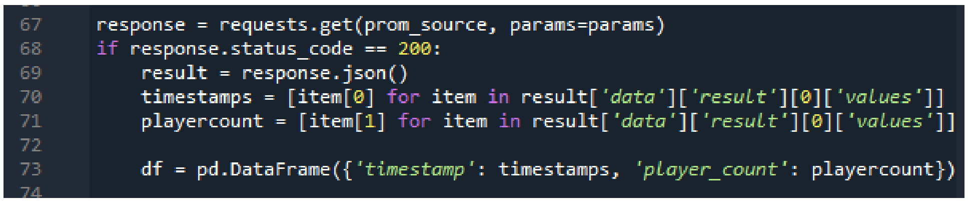
The Prometheus metrics are shown at the localhost with the port which the Prometheus Grafana stack has configured. The query was determined by testing it in the Prometheus endpoint, as it will display a graph if the query is valid. When using queries to get historical data, the time range is necessary. Prometheus uses the Unix time format, so the time ranges must be converted to this format for the query to work. We want to use historical data, including the most recent data, so the end time will be the current time. The start time will be 30 days before today, however, only data from roughly 2 weeks ago will be available. These time values are both converted to the Unix time format. With these values, the query can get the data from Prometheus.



```python
prom_source = 'http://10.196.39.235:9090/api/v1/query_range'
query = 'player_count{title="Team Fortress 2"}'

time_range_days = 30
step_seconds = 1800

end_time = datetime.utcnow()
start_time = end_time - timedelta(days=time_range_days)

start_time_unix = int(start_time.timestamp())
end_time_unix = int(end_time.timestamp())

params = {
    'query': query,
    'start': start_time_unix,
    'end': end_time_unix,
    'step': step_seconds,
}
```

Figure 19: Preparing Prometheus for query

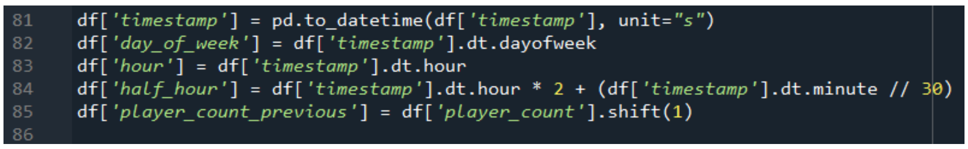
We can use the get function, similarly to how we did it for the first decision algorithm. If the status code is 200, which would indicate that a connection could be made, the data is converted from JSON into usable data that Python can deal with, so that we can get the timestamps and the player count. The result variable becomes a dictionary, with multiple nested lists and dictionaries. The timestamps are player counts are extracted from the data and a data frame is created based on these values. The pandas data frame allows us to analyze the data, create additional columns based on the feature engineering, and provides a good way to create a predictive model with data.



```python
response = requests.get(prom_source, params=params)
if response.status_code == 200:
    result = response.json()
    timestamps = [item[0] for item in result['data']['result'][0]['values']]
    playercount = [item[1] for item in result['data']['result'][0]['values']]

    df = pd.DataFrame({'timestamp': timestamps, 'player_count': playercount})
```

Figure 20: Processing data

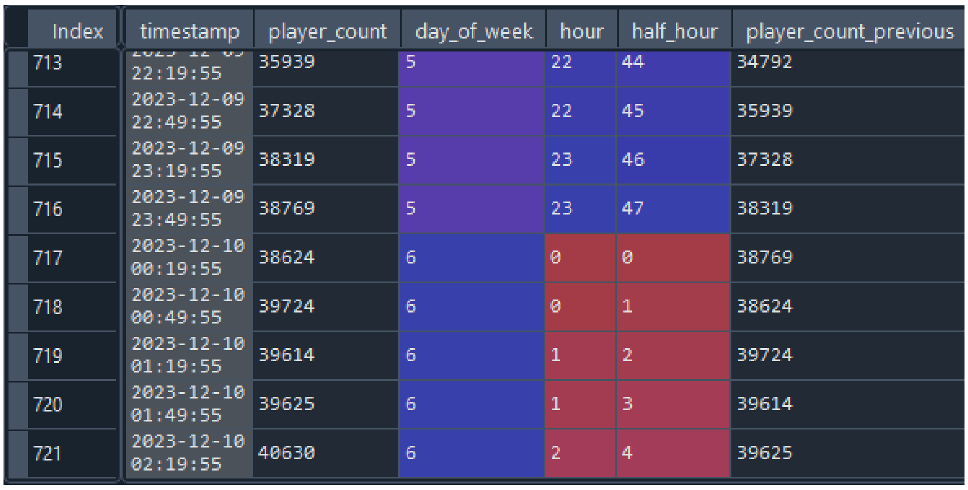
The timestamp in the data frame needs to be converted into usable format for the data frame so that we can add time-dependent features. It is converted from a Unix time format into a standard time format, and features for the day of week, hour, and the half hour are created. Pandas is useful for creating categories and features based on a timestamp, by using the date time functions. The half hour feature for the data is created by multiplying the hour by 2 and adding 1 if the half hour is from 30-60 minutes. A new column is created for the previous player count, which is done because the model should use previous data so that we can evaluate accuracy by comparing the actual player count with the predicted value.



```python
df['timestamp'] = pd.to_datetime(df['timestamp'], unit="s")
df['day_of_week'] = df['timestamp'].dt.dayofweek
df['hour'] = df['timestamp'].dt.hour
df['half_hour'] = df['timestamp'].dt.hour * 2 + (df['timestamp'].dt.minute // 30)
df['player_count_previous'] = df['player_count'].shift(1)
```

Figure 21: Feature engineering of processed data

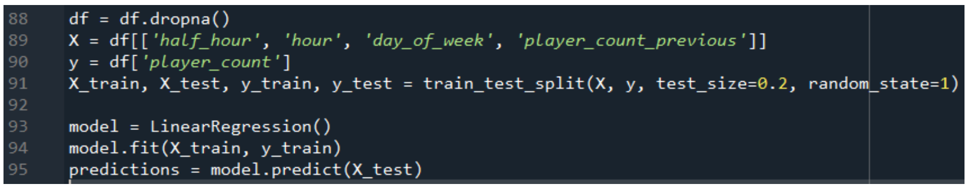
Once the data has feature engineered, it can be used to train a model. The dataset contains around 700 rows with data, with each row being 30 minutes between each other. We will also be using a time interval of 30 minutes between each execution of the scaling script for our testing, as this is the time interval for the dataset, it should provide a reasonable prediction based on historical data.



Index	timestamp	player_count	day_of_week	hour	half_hour	player_count_previous
713	2023-12-09 22:19:55	35939	5	22	44	34792
714	2023-12-09 22:49:55	37328	5	22	45	35939
715	2023-12-09 23:19:55	38319	5	23	46	37328
716	2023-12-09 23:49:55	38769	5	23	47	38319
717	2023-12-10 00:19:55	38624	6	0	0	38769
718	2023-12-10 00:49:55	39724	6	0	1	38624
719	2023-12-10 01:19:55	39614	6	1	2	39724
720	2023-12-10 01:49:55	39625	6	1	3	39614
721	2023-12-10 02:19:55	40630	6	2	4	39625

Figure 22: Snippet of data set

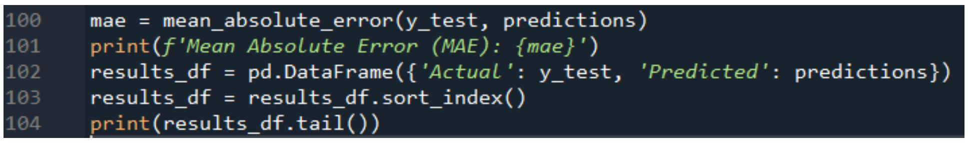
We drop any missing values if they exist so that the predictive model does not encounter any error which would prevent it from executing properly. When training and testing a model, the data set is usually split into a train data set, which is X, and contains all features and columns which the predictive model should use to train. These are the independent variables which were created by analyzing the player count trends. The real player count is not included in this data set, as it will be the dependent variable and used to evaluate how well the predictive model performed. The test size chosen is 20%. There isn’t a lot of data available, so going higher would not be ideal as it would reduce the amount of data which the model is trained on, which could result in the model not picking up on the patterns in the data. A random state is chosen so that the data set will be the same for every iteration of the program, which would be useful for comparison with other predictive models. A linear regression can then be fitted and trained on the data, as well as create a list with the predictions so that it can be evaluated.



```
df = df.dropna()
X = df[['half_hour', 'hour', 'day_of_week', 'player_count_previous']]
y = df['player_count']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)

model = LinearRegression()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
```

Figure 23: Creating linear regression model

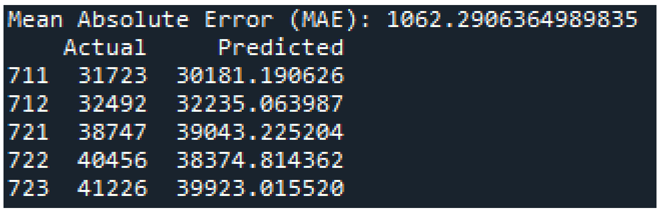
The predictive model can be evaluated by calculating the mean absolute error for the testing data set and the predicted values. The mean absolute error is a measure of the average error, for both underprediction and overprediction. We also create a data frame for the results, where we add the real player count as one column, and the predicted count as another. The data frame is also sorted so that it displays the rows in a chronological order.



```python
mae = mean_absolute_error(y_test, predictions)
print(f'Mean Absolute Error (MAE): {mae}')
results_df = pd.DataFrame({'Actual': y_test, 'Predicted': predictions})
results_df = results_df.sort_index()
print(results_df.tail())
```

Figure 24: Showing MAE and results of prediction

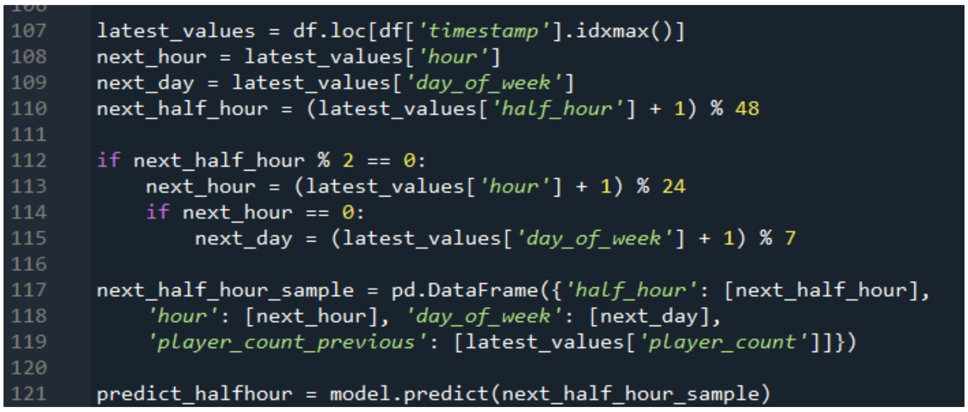
In this test, the mean absolute error was just over 1000, while other tests had a slightly higher MAE, anywhere from 1500-2000. However, these values could be considered acceptable since the player count in this particular game (Team Fortress 2) ranges from around 20000 to 60000. With the predictive model showing acceptable value for player count, it can be implemented into the scaling algorithm.



```
Mean Absolute Error (MAE): 1062.2906364989835
    Actual      Predicted
711  31723   30181.190626
712  32492   32235.063987
721  38747   39043.225204
722  40456   38374.814362
723  41226   39923.015520
```

Figure 25: MAE and result data frame is shown

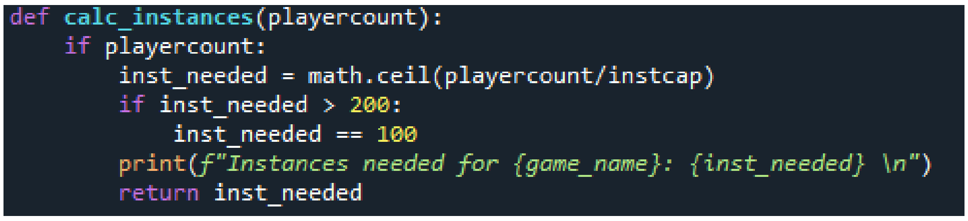
To predict the future player count, we must create features that reflect the values they will be in future. This involves getting the latest row of data and adding half an hour. The features need to be checked for so that it does not increase the value indiscriminately. The half hours will range from 0 to 47, so if the half hour is an even number, the hour must increase. Additionally, when the hour is increased, the day must increase, since the hours range from 0 to 23. The modulo operator ensures that the values are calculated correctly. For example, if the hour is increased from 23 to 24, the remainder becomes 0, which would increase the day. With these features calculated, the model can predict the future player count.



```python
latest_values = df.loc[df['timestamp'].idxmax()]
next_hour = latest_values['hour']
next_day = latest_values['day_of_week']
next_half_hour = (latest_values['half_hour'] + 1) % 48

if next_half_hour % 2 == 0:
    next_hour = (latest_values['hour'] + 1) % 24
    if next_hour == 0:
        next_day = (latest_values['day_of_week'] + 1) % 7

next_half_hour_sample = pd.DataFrame({'half_hour': [next_half_hour],
    'hour': [next_hour], 'day_of_week': [next_day],
    'player_count_previous': [latest_values['player_count']]})

predict_halfhour = model.predict(next_half_hour_sample)
```

Figure 26: Creating variables for next half hour prediction

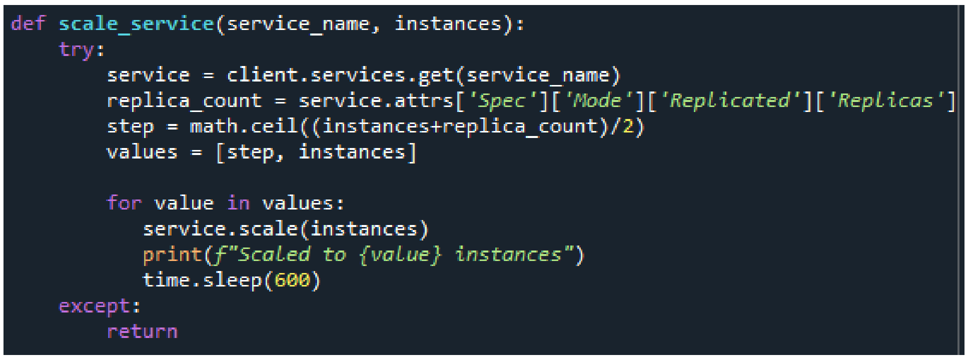
Once the model has predicted the future player count, the instances must be calculated. This is done similarly to the first scaling method we implemented. We will also check if the number of instances is higher than a threshold, just in the predictive model would predict incorrectly.



```
def calc_instances(playercount):
    if playercount:
        inst_needed = math.ceil(playercount/instcap)
        if inst_needed > 200:
            inst_needed == 100
        print(f"Instances needed for {game_name}: {inst_needed} \n")
        return inst_needed
```

Figure 27: Function to calculate instances for predictive model

The scaling process is slightly different than the first method. In this method, we decided to not scale all the servers at once. This was done because the number of instances will most likely be greater or lower than the current number of instances, and scaling all of them at once would mean that they either be empty at the time of scaling, or all instances would be full. Therefore, implementing a delay should solve this issue. However, knowing exactly at what time the instances will become populated or depopulated is difficult, so the time delay is set to 600 seconds. The number of currently running Docker containers is obtained easily, since we are using Docker services. A variable is created for the value which the service will first be scaled to. This value is simply the average of the current number of containers and the predicted number of containers. This works for both scaling up and down, as the first step is the average of these values, while the second step is the predicted value.



```
def scale_service(service_name, instances):
    try:
        service = client.services.get(service_name)
        replica_count = service.attrs['Spec']['Mode']['Replicated']['Replicas']
        step = math.ceil((instances+replica_count)/2)
        values = [step, instances]

        for value in values:
           service.scale(instances)
           print(f"Scaled to {value} instances")
           time.sleep(600)
    except:
        return
```

Figure 28: Function to scale service

4. Evaluation

From the figure below, the autoscaling for the first method, only scaling the number of Docker containers based on the current player count, ran for over 20 hours for the game Team Fortress 2, and it seems to follow the player count of the chosen game quite well. The instance capacity was 800 players for this test. In this time period, the player count goes from roughly 35000 at the minimum to its peak during the end at 55000. The player count hits lows at roughly 17:00, with the containers following the same trend. There is also a time period where the period hits 0, which could be many reasons, such as Steam or the game itself being down, or there were issues with

Prometheus scraping data for this time period. However, the implementation is not affected by this, as the code checks for errors with getting the player count. Overall, the implementation of the first autoscaling method was successful. Further testing on different types of games would be ideal, but due to time constraints, this was not possible.

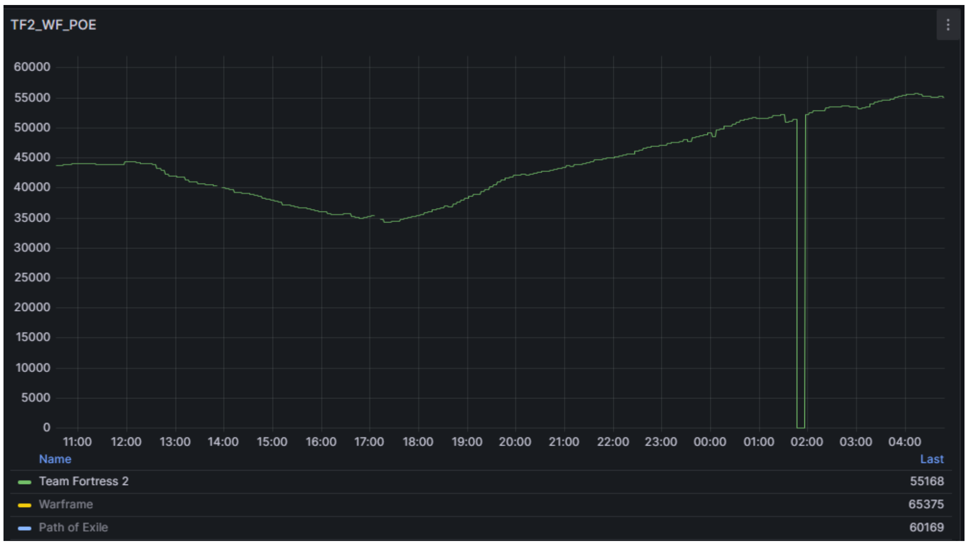
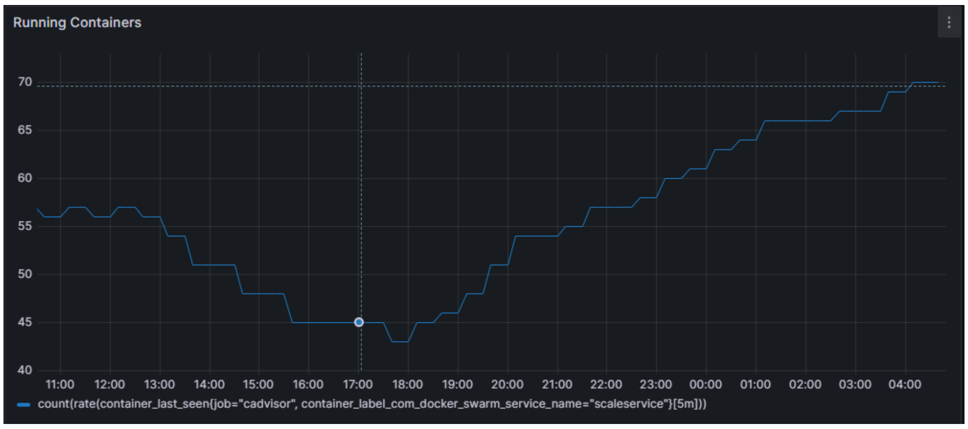


Figure 29: Player count for Team Fortress 2



count(rate(container_last_seen{job="cadvisor", container_label_com_docker_swarm_service_name="scaleservice"}[5m]))

Figure 30: Number of instances for Team Fortress 2, first scaling method

The autoscaling ran without encountering any issues, such as the program stopping or scaling the number of instances with a massive error. Most of the instances are within one or two instances below or above the expected number of containers. This is as expected, since the number of instances is calculated and scaled instantly, without accounting for the future player count. In general, the player count for this time period does not have unexpected spikes or dips, and the increase in player count is not rapid.

The scaling seems to be responsive as well, not having a long time period between each increase or decrease in the number of server instances. Having a small Docker image in the containers helps with this, as the computational resources required to launch the containers are not very high.

Ruby Acorn had previously tested their game hosting services with a fixed number of server instances. If we assume that the fixed number should at least be able to hold the maximum number of players in a time period, it will require roughly 69 server instances with 800 players on each instance, where the maximum number of players is at 55000 as seen in the figure below. This would naturally mean that many server instances would be empty for a long time period, which would increase costs. In this scenario, our autoscaling would have much lower costs, since the number of instances is only around 70 at its peak, with it going down to 45 instances when the player count hits its lows. The fixed number of instances would be at a constant 70 instances, which would mean that our autoscaling method would save at maximum 25 containers, which is a considerable amount.

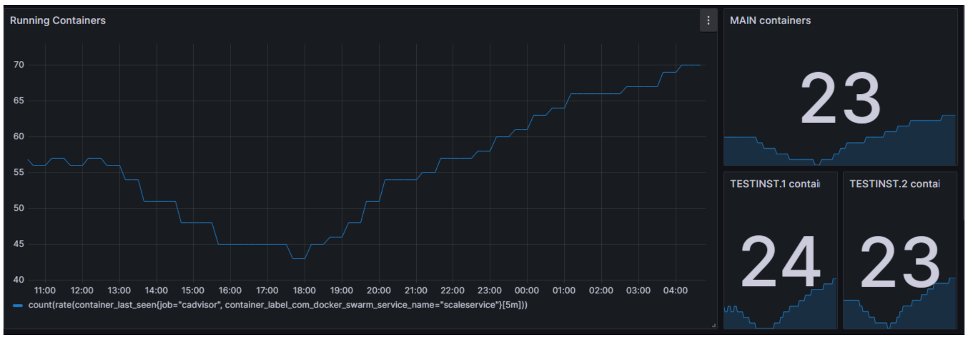
However, a fixed number of instances should not be completely dismissed, as one way they could improve costs is by having a fixed number of instances depending on the time period. As shown previously, player count is generally dependent on the time of day, and the day of week. One way this could be implemented is by using historical data to set a fixed number of instances. For example, player count for Team Fortress 2 is low at 15:00 through 18:00, so a low number of instances should be allocated at this period. However, at 23:00 through 04:00, the player count increases, and the number of instances could be increased. This method would work for games where a consistent and predictable player count is shown, however, this is the best-case scenario for a fixed number method, and an autoscaling method would overall perform better, as it could react to unexpected player count spikes and dips, something a fixed number wouldn’t be able to do.

An autoscaling implementation based only on the current player count should be a good starting point in autoscaling projects. It allows for resources to be allocated dynamically, where instances are scaled at a time point or when a condition is met. However, this method is quite simple as it does not require any historical data to work. Despite this, the autoscaling scales without there being periods where there are a large number of players above the expected number of instances at one point. This is most likely due to the capacity for each instance being 800, with the player count in the chosen game not increasing or decreasing by a lot over the time when the autoscaling is idle. If the capacity per instance were to be a lower number, such as 400, the autoscaling would potentially show different kinds of results, such as there being more empty servers during a period when scaling up or down.

The autoscaling occurred every 30 minutes, scaling either up or down depending on if the player count increased or decreased. Other time intervals should be tested, such as 15 minutes or even shorter, which would provide more frequent allocation or removal of server instances if the player count has changed rapidly since the last autoscaling. This would most likely mean that players would not have to wait for long periods of time if there weren’t enough server instances available. However, this could introduce some issues as well. During scaling, CPU and memory usage will usually spike, and having more frequent scaling would need to make sure that the system is not becoming unstable or slowing down due to the scaling. There would need to be more monitoring focused specifically on the system status, with measures in place so that emergencies are handled appropriately. Down time of the autoscaling would most likely introduce more costs, as the issue would need to be handled as well as having other scaling methods implemented while the autoscaling is down. This would ultimately reduce the quality of the game, which could lead to fewer players and profit overall.

In this project, we did not implement any methods which would analyze the status of the autoscaling system. Some methods could be implemented which would analyze the CPU and memory usage of the virtual machines during the scaling. If the scaling were to cause a maximum load on the system, the scaling could be done gradually over the next 20 minutes, for example.

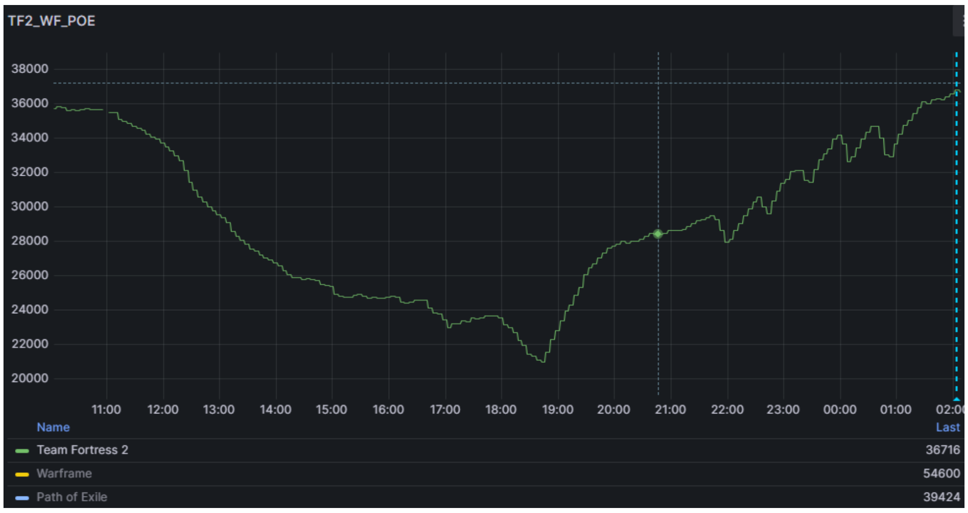
Docker Swarm has automatic load balancing, where it attempts to distribute containers equally amongst the nodes. This is useful for cases where there are many Docker containers being launched at the same time, distributing the resources across the nodes. For the same period, the individual nodes load balance the Docker containers correctly, with the same number of containers on each node. Naturally, some containers will have a container more or less, depending on the total number of containers, but the total number of containers in the 3 nodes equals the total number of containers currently running. More nodes could be added which would reduce the computational resources required per each node, allowing for more containers to be scaled. Load balancing, in combination with a small Docker image, reduces the chances of encountering issues related to having high computational costs.



```
count(rate(container_last_seen{job="cadvisor", container_label_com_docker_swarm_service_name="scaleservice"}[5m]))
```

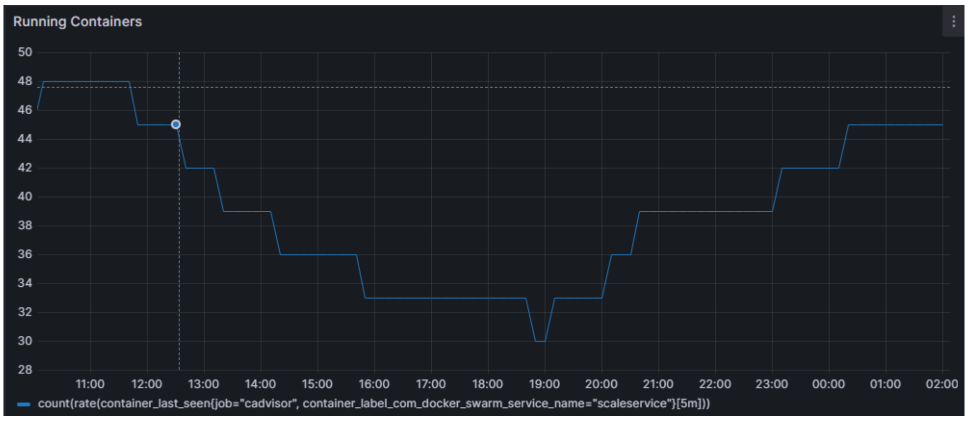
Figure 31: Number of instances and Swarm nodes

The predictive model was also evaluated on the same game to more easily compare the two methods. With a longer project timeline, more games should be tested to show how well it reacts to different kinds of player count. A good predictive model should be able to predict the future player count with little error. Similarly, to the previous method, it was executed every 30 minutes with crontab. The player count for this time period is much more volatile than the previous time period, which should provide a better evaluation of how the predictive model performs.



Name	Last
Team Fortress 2	36716
Warframe	54600
Path of Exile	39424

Figure 32: Player count for Team Fortress 2

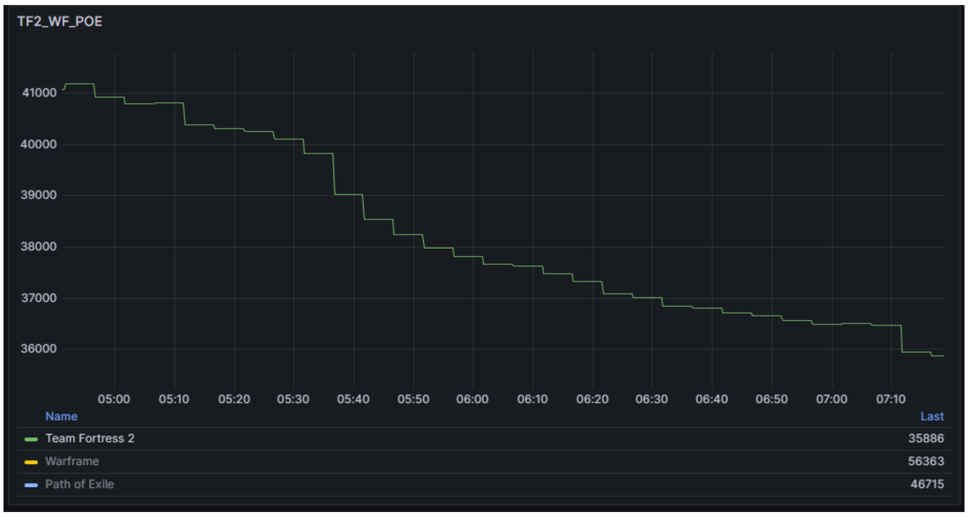


count(rate(container_last_seen{job="cadvisor", container_label_com_docker_swarm_service_name="scaleservice"}[5m]))

Figure 33: Number of instances for Team Fortress 2, predictive method

Generally, the current implementation of a linear regression follows the trend of the data, however, when it comes to its cost-efficiency, it seems to not perform as well. Firstly, at the beginning of the time period at around 11:00, Figure 33, the number of instances is 48, while the player count is roughly 36000. With an instance capacity of 800 players, this would result in 3 instances being empty for the time period. Additionally, from around 16:00 through 18:00, the number of instances stays at the same number, 33, which would hold at maximum 27200 players, while the real player count goes down to 23000. However, during roughly 21:00 through 23:00, the number of instances is acceptable, as the player count climbs from roughly 29000 to 31000 with the instances being at 39, meaning that they can hold up to 32000 players. Overall, the number of instances is greater than the number of instances required at a time, which means that players do not have to wait for long.

We reduced the number of players per instance from 800 to 400 to see how it performs when there are more instances to scale. From roughly 05:00 to 05:30, it seems to predict relatively well the number of players and instances to allocate. 102 instances would handle around 41000 players, and the real player count is around 40000. For the next hour, it is off by about 3 containers. Based on both of the tests conducted, it seems to struggle with predicting when there is rapid change in player count.



Name	Last
Team Fortress 2	35886
Warframe	56363
Path of Exile	46715

Figure 34: Player count for Team Fortress 2

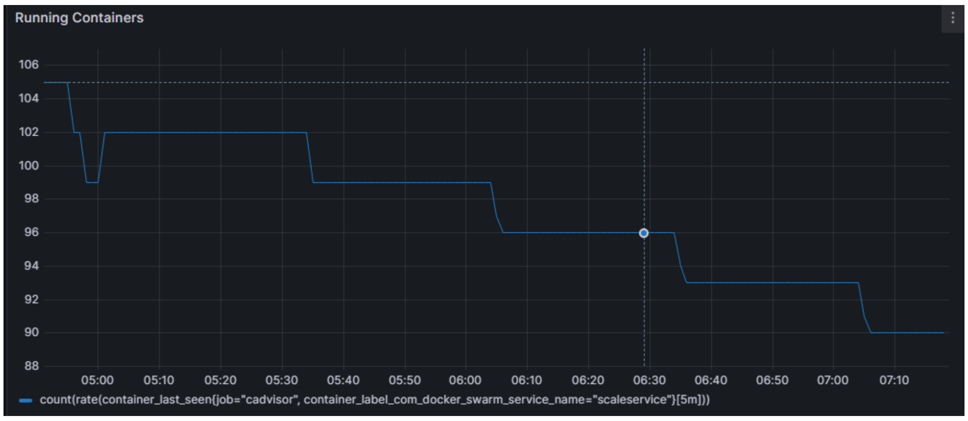
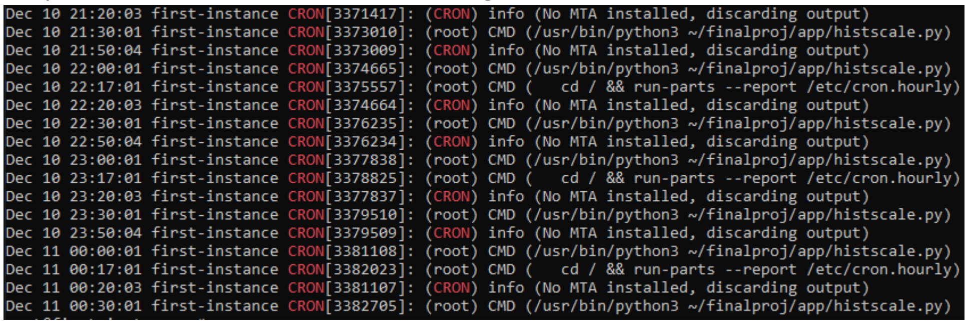


Figure 35: Number of instances, predictive method, 400 instance capacity

The results were mostly as expected, as the mean absolute error on the predictive model ranged from 1000-2000. The implementation of a linear regression model would require further testing to be considered a good choice for an autoscaling application. Having more data available could be useful, as it could highlight additional features in the data set. Using other predictive models should also be considered, as linear regression assumes that the relationship is linear. For the most part, player count has strong correlations with the time period as shown earlier, however, many factors can affect the true player count, such as servers being slow which leads to players leaving, causing a dip in the player count which a linear model would not recognize.

The autoscaling using historical data was done with a different script, “histscale.py” than the first method. The log shows that the script starts at the full and half hours, with the first script ending 20 minutes after the execution. This is due to the gradual scaling where the first scaling starts when the script is executed, and the second scaling is done 10 minutes later.



```
Dec 10 21:20:03 first-instance CRON[3371417]: (CRON) info (No MTA installed, discarding output)
Dec 10 21:30:01 first-instance CRON[3373010]: (root) CMD (/usr/bin/python3 ~/finalproj/app/histscale.py)
Dec 10 21:50:04 first-instance CRON[3373009]: (CRON) info (No MTA installed, discarding output)
Dec 10 22:00:01 first-instance CRON[3374665]: (root) CMD (/usr/bin/python3 ~/finalproj/app/histscale.py)
Dec 10 22:17:01 first-instance CRON[3375557]: (root) CMD (   cd / && run-parts --report /etc/cron.hourly)
Dec 10 22:20:03 first-instance CRON[3374664]: (CRON) info (No MTA installed, discarding output)
Dec 10 22:30:01 first-instance CRON[3376235]: (root) CMD (/usr/bin/python3 ~/finalproj/app/histscale.py)
Dec 10 22:50:04 first-instance CRON[3376234]: (CRON) info (No MTA installed, discarding output)
Dec 10 23:00:01 first-instance CRON[3377838]: (root) CMD (/usr/bin/python3 ~/finalproj/app/histscale.py)
Dec 10 23:17:01 first-instance CRON[3378825]: (root) CMD (   cd / && run-parts --report /etc/cron.hourly)
Dec 10 23:20:03 first-instance CRON[3377837]: (CRON) info (No MTA installed, discarding output)
Dec 10 23:30:01 first-instance CRON[3379510]: (root) CMD (/usr/bin/python3 ~/finalproj/app/histscale.py)
Dec 10 23:50:04 first-instance CRON[3379509]: (CRON) info (No MTA installed, discarding output)
Dec 11 00:00:01 first-instance CRON[3381108]: (root) CMD (/usr/bin/python3 ~/finalproj/app/histscale.py)
Dec 11 00:17:01 first-instance CRON[3382023]: (root) CMD (   cd / && run-parts --report /etc/cron.hourly)
Dec 11 00:20:03 first-instance CRON[3381107]: (CRON) info (No MTA installed, discarding output)
Dec 11 00:30:01 first-instance CRON[3382705]: (root) CMD (/usr/bin/python3 ~/finalproj/app/histscale.py)
```

Figure 36: Crontab logs for predictive model

There is much room for improvement when it comes to improving the autoscaling implementation. Currently, both methods scale the number of instances by analyzing the player count every 30 minutes. Both of these methods could be combined into one solution, where a predictive model could predict the player count for the next half hour, and the first method being executed if the player count is far greater or lower than what the predicted player count was. There is a huge selection when it comes to predictive models, and more advanced machine learning models could be explored. Linear regression is commonly used for prediction with linear relationships between the target and the independent variables, and it might not be the best method for player count as there are often many factors that can affect it, including ones with a non-linear relationship. Linear regression was chosen for its popularity and ease of use, but more advanced machine learning models could potentially find patterns more effectively. Training and testing such a model would take more resources and time, as there are generally parameters to tune, and the general computational resources should be higher than a linear regression model. We only used 3 virtual machines in this project, when the cloud allows up to 8 instances. This would allow us to test the models in different games simultaneously, which could be useful, as the testing of an autoscaling method takes a long time if we want to get a good understanding of how well it performs. Having more data on the player count would be useful, as we could include features related to holidays and time periods where the player count is not only affected by the time of day or the day of the week.

Overall, the current implementation provides a good foundation, and it is recommended that Ruby Acorn pursue an autoscaling method, as they do not currently have any way of dynamically adding or removing server instances.

5. References

[1] Jacobson, D., Yuan, D., & Joshi, N. (2013, November 5). Scryer: Netflix's Predictive Auto Scaling Engine. Netflix TechBlog. https://netflixtechblog.com/scryer-netflixs-predictive-auto-scaling-engine-a3f8fc922270

[2] acit-game-metrics.cs.oslomet.no/metrics

[3] vegasbrianc. (2023, February 1). Prometheus. GitHub. https://github.com/vegasbrianc/prometheus