

Ruby Acorn Auto Scaling

Candidate 358

April 25, 2024

Contents

1	A discussion about the case	1
2	A technical design	2
2.1	Assumptions	2
2.2	Base-load servers	3
2.3	Auto-scaling	3
2.4	Tech Stack	4
2.4.1	Prometheus	5
2.4.2	Grafana	6
2.5	CI/CD	6
2.6	Architecture	7
3	Pilot implementation	9
3.1	Server Setup	9
3.2	Prometheus and Grafana	10
3.2.1	CI/CD	11
3.2.2	Server status	14
3.2.3	Auto-scaler	19
3.3	Scaling experiments	19
3.4	Scaling solutions overview	24
3.5	Solution 5 with real servers	25
3.6	Scaling solution on other games	25
3.7	Hot swap Scaler Schema	26
3.8	Lessons from development	27

4	An evaluation	28
4.1	Server capacity evaluation	29
4.2	Evaluation of scaling algorithm	30
4.3	Evaluation of added complexity	31
4.4	Conclusion	31
5	Appendix	32
5.1	gitlab-ci.yaml	32
5.2	OpenStack/app.py	34
5.3	OpenStack/fake_servers.py	36
5.4	OpenStack/Dockerfile	37
5.5	Scaling/app.py	38
5.6	Scaling/fakeserver.py	41
5.7	Scaling/openstackutils.py	43
5.8	Scaling/scale.py	45
5.9	Scaling/utils.py	51
5.10	Scaling/Dockerfile	54
5.11	Scaling/.env	55

1 A discussion about the case

Ruby Acorn is a gaming company that host their own gaming servers. From the project description, we can see that they struggle with providing the correct number of servers for the ever-changing number of active players.

Before starting the project, Ruby Acorn has already considered two solutions,

1. Manual scaling - engineers scaling server on demand
2. Fixed adjustment based on time of day

For item 1, it's concluded that this might be even more costly than having too many servers running, and for solution 2, we see that scaling on time of the day do not always work, as influencers, streamers and other events will affect the numbers of players greatly. One option has yet to be tested, namely autoscaling. To lower costs, this solution should work without human intervention, and provide the engineering team with a dashboard to track the status of servers and players.

From the project description, the goal of this project is to answer:

1. How much money could be saved compared to a fixed number of instances?
2. How well would such a solution react to any changes in player numbers?
3. What would be a good way to measure a "successful 24 hours" using automated scaling?

To answer these questions, it is needed to define them somewhat more. For question one, we need to decide on what a fixed number of instances mean. To allow the game to grow without the servers being pushed to the max, I suggest that we define fixed number of instances to the handle the last 3 months max players + 25% for each game. Here is it important to remember that renting servers for a year is cheaper than renting on-demand. To get a fair comparison, we will add a 25% discount to servers rented for the fixed number of servers.

Question two is well-defined, and the object here would be to scale as efficiently as possible to keep costs down and players happy. By running some experiments and closely monitor the performance of the server status, we can get a good estimate of how well these solutions react to changes in players.

Question three can be looked at in a couple of different ways. We can define it as the servers never being full, meaning that players always get to log

in when they want, but we can also delve into the psychology of Social Proof. In a paper exploring if Social Proof exist online as well, they investigated the online ticket store of the Dutch National Opera. If their findings can be translated into the behavior of gamers, one could expect that, a login queue, stemming from server scarcity, can induce a sense of urgency in players to join. However, this perceived scarcity might deter them from making in-game purchases and seasoned gamers, familiar with such tactics, might be less influenced by restricted server access [1]. Even if this approach might be accepted during new releases or during extraordinary events, I suggest we try to always keep some free capacity on the servers. How we calculate the capacity can change between the scaling algorithms, and will in this project use both free space in terms of percentage, and free space in terms of free player spots. We need to explore as many options as possible to provide gamers with the best experience.

It is also worth noting that this pilot project only looks at the number of players online, and not how they actually use resources on the server. This would need to be explored further if we decide to move forward with the solution.

In the world of gaming, changing player counts are influenced by various external factors, including social media trends, twitch streamers and game updates. Ruby Acorn face the complex task of efficiently adjusting server resources to these ever-changing demands. Balancing the need between always having enough servers for smooth gameplay, while keeping costs down, is the core idea behind this project.

For Ruby Acorn, the essence of the problem lies in the real-time allocation of server resources. The optimal approach would be one that's largely automated, minimizes human intervention, quickly responds to the change in player numbers, and ensures good use of resources to reduce costs.

2 A technical design

2.1 Assumptions

Some assumptions need to be in place to describe a good technical design. First, we will split the problem of autoscaling into two parts. I think we will benefit from having some static servers. These servers will serve a low number of players, but will also ensure that even if the scaling solution collapse, the game will still be playable for some players. Therefor will the first part of the solution consist of defining a base-load. The discussion and design for this is found in 2.2. After a base-load is defined, we need to scale servers based

on number of players. Due to delays in the data pipeline, we only receive information about the numbers of players every five minutes. The data delay makes us "blind" for five minutes, only giving us a static view of the players, and a sudden surge in players between two data points can lead to under provisioning of servers. We will however look at ways to combat this issue by including the derivative of player change, and add that as a factor to the scaling solution.

2.2 Base-load servers

Opting for long-term server rentals can lead to significant cost savings, as many providers offer discounts for commitments over fixed time periods. Common rental plans include 1-year and 3-year duration's, with some providers also offering monthly options. For this pilot, we assume that the savings on a 1-year option is 10% and a 3-year option is 20%. Based on this, I recommend reserving a certain number of servers on a monthly or yearly basis to manage the base load of the game effectively.

In our context, the base number of servers should correspond to the minimum number of active players observed in our historical data. This approach ensures that the infrastructure is always primed to handle the lowest expected player load. In a practical scenario, this number should be defined by doing some more analysis, as the lowest number of recorded players could be due to a server outage or other unknown issues. A better approach might be setting the base load to the average number of player - 40%.

2.3 Auto-scaling

During auto-scaling we need to fetch the necessary data. In our case, this will be the number of servers running and number of gamers online. The server status is updated in real time, while the player count has a 5-minute update cycle. Because of this delay, it might be beneficial for us to try to predict the future when scaling the servers. There are a lot of auto-scaling techniques, and ideas taken from several domains.

In [2], Jawaddi and Ismail preformed a taxonomy on autoscaling in serverless computing. When discussing scaling techniques in Section 5.3, they mention that one of the biggies challenges when it comes to scaling is the cold start problem. This issue comes into play since a server don't start instant. There will always take time from a server is started until it's ready to be used. To get around this problem, we chose in this project to use a rule-based technique, that ensures enough free server capacity to keep the game running smooth even in the time delay.

Furthermore, the paper describes scaling the rule-base technique as a commonly implemented algorithm used in Cloud Service Providers, because of its straightforward approach in scaling decisions, which are triggered when system performance crosses predefined thresholds. This technique involves estimations, where resources are added or removed in specific amounts or percentages, and does not require precise resource estimation. There are two types of thresholds used in this technique: static and dynamic[2]. We will implement our own version of both the static and semi-dynamic solutions in this project.

The paper also discusses additional scaling techniques, such as AI-based techniques, analytical modeling, control theory, application profiling, and hybrid methods. Each of these techniques offers unique advantages and approaches to tackle the complexities of scaling in serverless computing environments. AI-based techniques, for instance, might leverage machine learning algorithms for more predictive and adaptive scaling, while analytical modeling could provide a more theoretical and systematic approach to understanding resource needs[2].

If we choose to go forward with the auto-scaling solution, a hybrid rule-based and AI-based technique would from my experience be an interesting solution. This is due to the repetitive pattern in player numbers, as can be seen in figure 1. Here we see that there is a clear pattern in when we have a lot of players online, and it's consistent throughout the entire period.

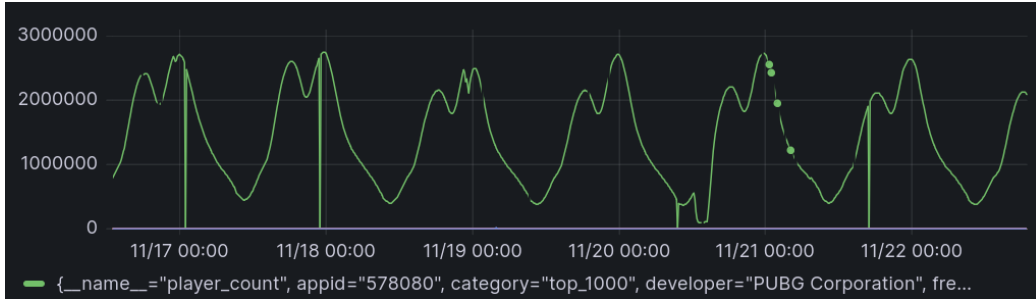


Figure 1: Number of players for PUBG over a 5-day period

2.4 Tech Stack

The autoscaling system utilizes a centralized technology stack hosted on a single virtual machine (VM) instance. This provides a streamlined and self-contained platform for managing all components of the scaling process.

The core services deployed on the management VM are:

1. **GitLab Runner** - Automates running CI/CD pipelines to deploy new code versions. It builds updated container images and orchestrates rolling out changes.
2. **Prometheus** - An open source time-series monitoring and alerting toolkit. It collects, stores, and allows querying all metrics required to drive scaling decisions.
3. **Grafana** - Visualizes the Prometheus metrics data. Provides dashboards and graphs for real-time monitoring of system health and performance.
4. **Server Status Container** - Polls game servers frequently to gather updated data on current load and capacity. Exposes this as metrics to Prometheus.
5. **Autoscaler Container** - Houses the core scaling algorithm logic. Consumes live metrics from Prometheus, calculates needed capacity changes, and interfaces with Kubernetes to spin up or terminate servers accordingly.

GitLab Runner, Prometheus, and Grafana are static services installed and configured only once on the management VM. In contrast, the Status and Autoscaler containers are integrated with the CI/CD pipeline for automated deployment of updates.

The Status container is optimized to poll game servers frequently, collecting current load data. It exports this as metrics to Prometheus, establishing the foundation for the Autoscaler's calculations.

The Autoscaler container houses the core scaling algorithm logic. It consumes live metrics from Prometheus, applies predefined rules, and determines appropriate capacity increases or decreases. It then interfaces directly with Kubernetes to provision or decommission servers accordingly.

By leveraging Docker and automation via CI/CD, the stack minimizes operational overhead for the user. New code can easily be deployed by pushing changes to the Git repository. Centralized metric collection in Prometheus provides transparency into system performance. Overall, the technology choices aim to deliver maximum automation, scalability, and observability.

2.4.1 Prometheus

Prometheus is a critical component of the stack. It enables aggregating all the time-series metrics required to make data-driven scaling decisions. It

uses a pull model to scrape exposed metrics from the Status and Autoscaler containers. The custom timeseries database provides powerful querying and alerting capabilities.

As an open-source project created by SoundCloud, Prometheus, a monitoring and alerting toolkit, is pivotal for our system’s reliability and scalability. Adopting a multidimensional data model, Prometheus focuses on time series data, identified using specific metric names complemented by labels. It employs a pull-based approach to fetch metrics from instrumented jobs and introduces a versatile query language, PromQL, to enhance data interaction [3]. Prometheus is used to handle all metrics and data points needed for the autoscaling solution.

2.4.2 Grafana

Grafana, an open-source analytics and interactive visualization web application, is integral to our project for monitoring and analyzing cloud-native environments. It excels in data visualization, providing a versatile platform for constructing a variety of dashboards to track different metrics. A key feature of Grafana is its smooth integration with Prometheus, a specialized time-series database geared towards efficient data collection and storage. This integration is vital as it not only supports data collection and basic alerting but also enhances Grafana’s visualization capabilities. It allows users to craft dashboards that query and exhibit data from Prometheus, making Grafana a comprehensive tool for monitoring, alerting, and visualizing data in diverse applications and systems¹. In our project, Grafana’s role is to create dashboards that effectively monitor the performance of our auto-scaling solution. This will help the employees in charge of the scaling solution in their day-to-day operations.

2.5 CI/CD

CI/CD (Continuous Integration/Continuous Deployment) is a great asset to our development process. It makes us able to implement changes to the scaling application swiftly and safely, significantly reducing the risk of disruptions or failures.

As shown in Figure 2, our CI/CD pipeline is structured in a way to catch errors before they disrupt the application. The initial step involves testing the code. This is a crucial phase where potential issues are identified before getting pushed into the production code. Following this, we proceed

¹<https://grafana.com/docs/grafana/latest/getting-started/getting-started-prometheus/>

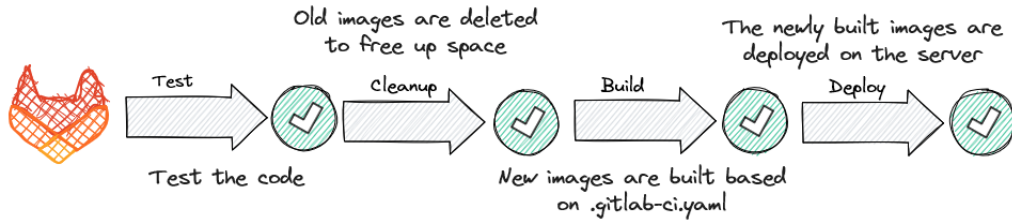


Figure 2: High level CI/CD design

to remove old images, freeing up space and make the instance ready for the introduction of updated code.

The idea behind testing the code prior to the cleanup of old images is to ensure continuous functionality of the scaling application. If the cleanup occurred prematurely, it could result in a dysfunctional pipeline, where no new code is deployed while the previously functioning code is already discarded.

During the build phase, we focus on constructing new containers directly on the server. These containers are then launched in the deployment phase. This systematic approach not only ensures the smooth and efficient updating of our application but also maintains its stability and reliability throughout the process.

2.6 Architecture

In a best case scenario, we should have a dedicated scaling service running for each game we need to scale. The service need to be able to get the number of online players in a game, have access to start, stop and count servers, and do the necessary calculations to scale servers correctly.

First we can design a couple of docker services, such as the Prometheus and Graphana Stack as described in section 3.2, and the two containers containing our code, namely the Scaler and Openstack Exporter. The setup is illustrated in figure 3. The internal communication between the Scaler and OpenStack Exporter is mainly due to the need for running "Fake Servers" during setup and development phases.

Figure 4 illustrate how the docker services from figure 3 integrate with the rest of the system, including the GitLab CI/CD pipeline.

The process begins with deploying the code through the CI/CD pipeline in GitLab. This crucial step involves setting up two key components: the Scaler and the OpenStack Exporter. It's assumed that both Prometheus and Grafana, along with the GitLab runner service, are already installed and operational on the server at the time of its creation.

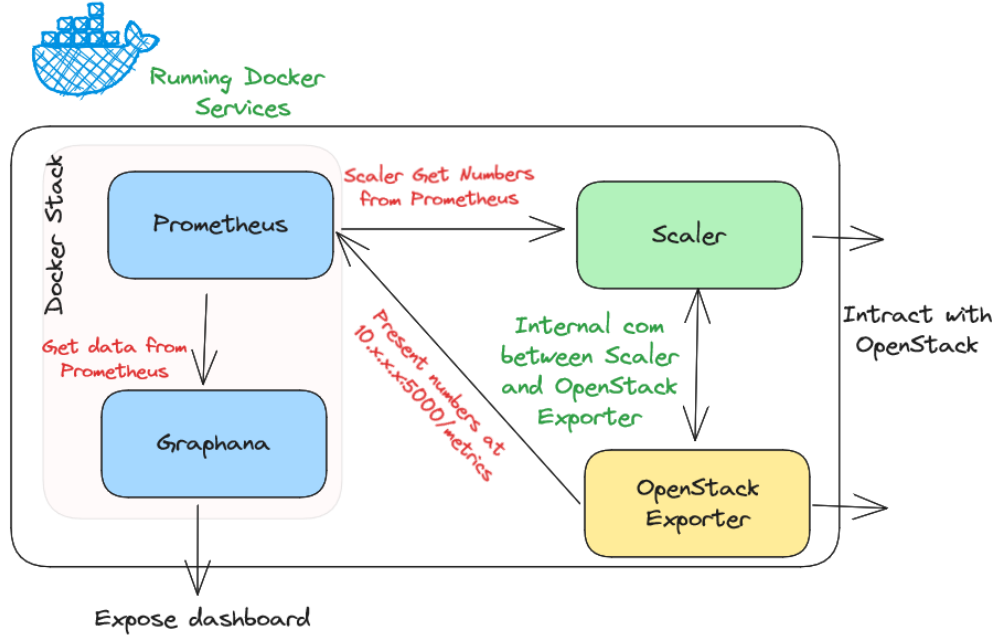


Figure 3: Running docker services

The OpenStack Exporter retrieves the current status of servers from our cloud infrastructure. The data it gathers is formatted in a way that is compatible with Prometheus. In addition to this, Prometheus is tasked with monitoring the current number of online players in the game, which is a critical factor for our scaling operations.

Based on the data collected, the scaler service then evaluates the situation and decides on the necessary scaling actions. These decisions are communicated to OpenStack through the OpenStack SDK, ensuring a responsive and dynamic scaling process.

To enhance visibility and ease of monitoring, the effectiveness and current status of the scaler is displayed on a dashboard in Grafana. This visualization helps in keeping a real-time track of the scaler's performance and the overall system health, making it easier for our team to manage and optimize the scaling process effectively. For scaling proposes, I suggest that the number of running servers are multiplied with the capacity per server, and baseload added after that. That way, it will be easy to do a visual verification of the solution. This do however call for some manual work when setting up a new game.

One solution to handle changes in user habits, is the ability to hot swap algorithms. This makes it possible to change algorithms due to time of day,



Figure 4: Architecture of the scaler server

or planned holidays.

3 Pilot implementation

To verify our ideas about the scaling solutions, I decided to deploy a pilot implementation. The pilot implementation is just a proof of concept, and some changes should be expected to make it run in a production environment. The code created for this implementation is found in section 5.

3.1 Server Setup

Some initial server configuration is required to enable the key components of this system, including installing the GitLab runner, Grafana, and Prometheus.

Once set up, the CI/CD pipeline will automatically deploy code changes from the main GitLab branch to the live servers. This following sections will step through the server setup process.

A few key points on the server architecture:

- In a production deployment, servers would likely be provisioned in multiple regions to minimize player latency. However, during this project implementation, region-specific player data was not available. Supporting geo-distributed servers is straightforward to add by creating region-aware scaling instances, similar to how different scaling algorithms are handled in Section 3.2.3.

- The GitLab runner is installed on the servers to enable automated deployment of code changes via the GitLab CI/CD pipelines. The pipelines are configured to rebuild and redeploy the application whenever the main branch is updated.

- Prometheus and Grafana are deployed to provide metrics collection and visualization capabilities for analyzing the scaling algorithms. Prometheus scrapes and stores time-series data exposed by the application and hardware monitors. Grafana reads data from Prometheus and presents customizable dashboards.

- Experimenting with different scaling algorithms is enabled by the "fake server" approach described in Section 3.2.2.

3.2 Prometheus and Grafana

To speed up development, I chose to use a pre-built Prometheus & Grafana docker-compose stack, published by Vegasbrianc[4]. The installation process is described in listing 1. At line two, we clone the pre-built stack from GitHub. Then I changed the admin password for Graphana using the nano editor from the console. On line seven, we start the stack. Once it is up and running, we need to do some additional setup.

Listing 1: Prometheus and Grafana installation

```
1 # Make sure you are in project root
2 git clone https://github.com/vegasbrianc/prometheus
3 # Change Graphana admin password
4 nano prometheus/grafana/config.monitoring
5 # Change the password at line: GF_SECURITY_ADMIN_PASSWORD=foobar
6 # Start the docker stack
7 docker stack deploy -c prometheus/docker-stack.yml prom
```

To connect Prometheus to our data sources, I edited the prometheus.yml file, found in prometheus/prometheus/ to include two new sources². On line 1–4 we define the endpoint that serves us the numbers of players for each

game, updated every five minutes. The scraping interval on line 2 is set to 60s, but could of course been set to 300s to save on network traffic.

On line 6-8, we define the endpoint that provides information about the numbers of servers we got running. The scraping interval for this service is defaulted to 15 seconds. This interval could also be set closer to 300s if we need to save on network traffic. It is worth noting that this service need more setup before Prometheus can pull data from it. This will be described in Section 3.2.2

Listing 2: Lines added to prometheus.yaml

```
1 - job_name: 'games'
2   scrape_interval: 60s
3   static_configs:
4     - targets: ['10.196.36.11:80']
5
6 - job_name: 'openstack'
7   static_configs:
8     - targets: ['server_ip:5000']
```

3.2.1 CI/CD

To achieve continuous integration and deployment (CI/CD) we utilize GitLab runners. The runner will, based on our pipeline script, build, test and deploy all necessary containers to the VM, ensuring that changes can be deployed fast and effortless. This service significantly reduces the manual work required, thereby accelerating the development cycle and enhancing the productivity of the development. The first step to use GitLab Runner, is to install it onto our server. This is done with script 3 and 4.

Listing 3: GitLab runner installation on Ubuntu

```
1 curl -L "https://packages.gitlab.com/install/repositories/runner/gitlab-↵
   runner/script.deb.sh" | sudo bash
2 sudo apt-get install gitlab-runner
3
4 # Check the status of the runner
5 service gitlab-runner status
```

After installing the service, there are some need for configuration before it will work with our system. To get the necessary values, we can go to Settings - Project runners in GitLab and copy the URL and key shown in figure 5.

Listing 4: GitLab runner init

```
1 gitlab-runner register
2 # URL - https://git.cs.oslomet.no/
3 # gitlab-ci token - [TOKEN FROM GITLAB]
4 # Executer - shell
5
6 # Add gitlab runner to the correct usergroups for building and testing
```

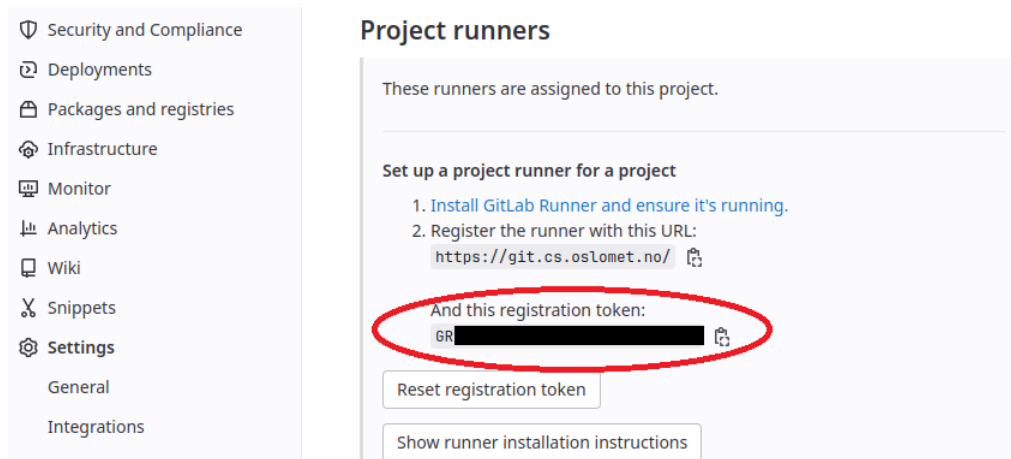


Figure 5: URL and token for GitLab - runner

```
7 usermod -aG docker gitlab-runner
8 sudo -u gitlab-runner -H docker info
```

Once the service is running on the server, we can verify that it runs by revisiting the runner page in GitLab, and we should see the status as shown in figure 6.

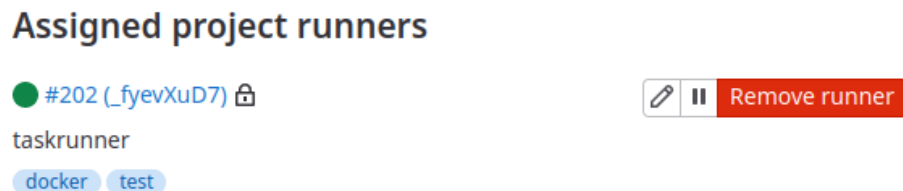


Figure 6: Verify status of the GitLab runner

In the development of our CI/CD pipeline, as detailed in the `gitlab-CI/CD.yaml` file (refer to script 15), a methodical approach was employed to ensure efficient and secure deployment of our services. The process begins with a cleanup stage, where old Docker containers are stopped and removed to maintain a pristine state on the server. This step is crucial for preventing potential conflicts or storage filling up from previous deployments. Please note that in a production environment, we would preform a test stage to ensure that the images builds and runs as expected before running the cleanup stage.

Then, we proceed to build new Docker containers. This phase is critical as it implements the changes to the docker images in charge of the scal-

ing service and the Prometheus reporting. The reason for splitting these services into two distinct containers is driven by the principle of separation of tasks. The scaling service (Auto-Scaler), tasked with dynamically managing resources, operates independently, allowing for targeted updates and maintenance without influencing the monitoring container. In contrast, the Prometheus reporting container (OpenStack-Exporter) is exclusively focused on monitoring and reporting. Because of the nature of the reporting container, this will not see as many code changes during development as the Scaler container, and it was therefor a natural choice for me to split the tasks between two containers.

Also, by isolating this service, we ensure its operations are streamlined and more secure, unaffected by the complexities inherent in the scaling service. Since the reporting container is build with fastApi, it also allows us to communicate with the scaling service by evoking endpoints. This is further described in section 3.7.

An important component of our pipeline is the incorporation of environment variables in the build process. These variables include sensitive information such as login credentials to our OpenStack server and are essential for the operation of our Docker containers. By securely storing them in GitLab and integrating them only during the image-building process, we effectively shield these sensitive details from exposure. This approach ensures our CI/CD pipeline is both functional and secure.

As the project scales, the significance of this secure management of environment variables escalates, and I recommend storing all sensitive data, like passwords, internal URLs, and API keys, in a centralized and secure manner, as it is done in this pilot. This practice not only safeguards critical information but also facilitates ease of management and oversight.

For a visual understanding of this management, refer to figure 7, which displays the GitLab interface used for configuring these environment variables. This interface offers a straightforward and centralized way to manage the essential variables, thereby streamlining the process of updating and handling access to sensitive data.

The architecture of our CI/CD pipeline, with its emphasis on modular design and security, forms the foundation of an efficient and robust deployment process. The division of services into separate containers, combined with the strategic handling of environment variables, ensures that our infrastructure is not only scalable but also secure and resilient against potential vulnerabilities.

As an added service, I created a POC dashboard for the GitLab Runner as well. Here we can verify that the runner is working as expected, and get information about failed runs and builds. In this project, the runner only

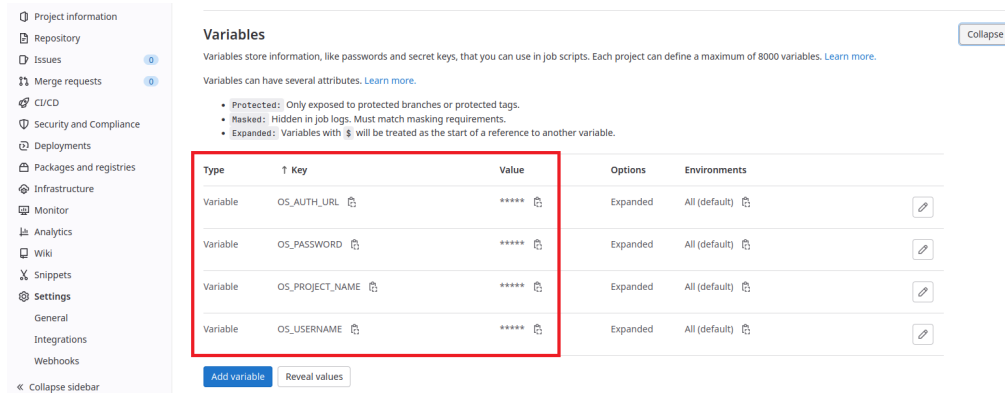


Figure 7: GitLab Variables - For storing secrets

starts when we get new code pushed to the main branch. One can therefore argue that a dashboard for the runner is redundant, as you would expect the user pushing code would monitor the deployment directly from the GitLab CI/CD site. However, this is just a stepping stone into making the system easier to monitor and use.

3.2.2 Server status

The docker service, OpenStack-Exporter, is the service that will monitor the number of servers running. In a production environment, we would be interested in metrics like CPU, RAM and GPU use, but for our design we only monitor the numbers of servers running. This service is essential as it provides a real-time count of active servers, which can be crucial for both load balancing and understanding the current load the infrastructure is under. During the POC stage of this project, I got seven servers to test the scaling solution. Due to the shortage of servers, they are set up to handle a huge number of players for the games we are testing. This leads to slow scaling, and making it hard to evaluate the solution. To combat this, a "Fake Server" method was implemented, making it possible for the Auto-Scaler to scale up and down fictional servers and report them to the same endpoint as the real servers are reported. From there the numbers are reported to Prometheus.

The backbone of managing servers on OpenStack is shown in listing 5. On line three to 17, we initialize a server, and pull the auth variables from GitLab. On line 20, the function to create a new instance begins, where we spin up a new server using the OpenStack SDK. On line 41, we have the reverse process, namely deleting a server.

One could argue that it would be more beneficial to create the full amount of servers once, and then just start and stop them on demand. However, since

these are empty test servers, I decided to start new instances each time to introduce an additional time delay when scaling. However, in hindsight, this time delay do not create an issue for the scaling solution, and starting and stopping VM's instead of building new one would be good enough for the pilot project.

Listing 5: Openstackutils.py code

```

1  class OpenStackManager:
2
3      def __init__(self, game):
4          # Add game name
5          self.conn = connection.Connection(
6              auth_url=os.getenv("OS_AUTH_URL"),
7              project_name=os.getenv("OS_PROJECT_NAME"),
8              username=os.getenv("OS_USERNAME"),
9              password=os.getenv("OS_PASSWORD"),
10             user_domain_id='default',
11             project_domain_id='default'
12         )
13         self.image_name = os.getenv("IMAGE_NAME")
14         self.flavor_name = os.getenv("FLAVOR_NAME")
15         self.network_name = os.getenv("NETWORK_NAME")
16         self.instance_base = os.getenv("INSTANCE_BASE")
17         self.instance_base = f"{game}-{self.instance_base}"
18     ...
19
20     def create_instance(self, instance_name):
21         """Create a new VM instance."""
22         try:
23             image = self.conn.compute.find_image("Ubuntu-20.04-LTS")
24             flavor = self.conn.compute.find_flavor(self.flavor_name)
25             network = self.conn.network.find_network(self.network_name)
26
27             server = self.conn.compute.create_server(
28                 name=instance_name,
29                 image_id=image.id,
30                 flavor_id=flavor.id,
31                 networks=[{"uuid": network.id}],
32             )
33
34             server = self.conn.compute.wait_for_server(server)
35             logger.info(f"Created server {server.name} with id {server.id}↵")
36             return server.name
37         except Exception as e:
38             logger.error(f"Error starting new VM {e}")
39             return None
40
41     def delete_instance(self):
42         """Delete an instance and its attached volumes."""
43         try:
44             # Fetch all servers
45             servers = list(self.conn.compute.servers())
46
47             # Filter servers with the specified instance base name
48             filtered_servers = [server for server in servers if self.↵
49                               instance_base in server.name]
50
51             # If no matching servers, nothing to delete

```

```

51         if not filtered_servers:
52             logger.warning(f"No instances found with base name {self.↵
                    instance_base}!")
53             return
54
55         # Sort filtered servers by created date
56         instance_id = sorted(filtered_servers, key=lambda s: s.↵
                    created_at, reverse=True)[0]
57         instance = self.conn.compute.get_server(instance_id)
58         # Now delete the instance
59         self.conn.compute.delete_server(instance_id)
60         logger.info(f"Deleted instance {instance_id}")
61
62     except Exception as e:
63         logger.error(f"Error in deleting instance or its volumes: {e}"↵
                    )

```

The fake servers implemented in the OpenStack exporter also open up an exciting possibility - testing different auto-scaling algorithms against the same game workload. Since the fake servers are simply tracked as an integer metric exposed to Prometheus, we can easily spin up a large number of auto-scaling control loops side by side and monitor their performance. The results of these experiments will be described in detail in Section 3.2.3.

The API endpoint implementation for the OpenStack exporter is shown in Listing 6. Lines 1-8 define a function to report the number of running servers, aggregating both real and fake servers.

Lines 10-36 then define the API endpoints used to control the fake servers. As can be seen, the implementation is quite basic but provides the key capabilities needed - creating servers, scaling up/down, and querying the number of running servers.

The first endpoint on lines 1-8 is a GET method, meaning it just returns current stats when hit. The remaining endpoints are POST methods which take parameters like a game name and perform operations like scaling up/down or querying server counts.

This simple but flexible API provides the foundation needed to make the auto-scaling algorithms dynamic and extensible. It opens the door for easy integration of new control mechanisms going forward.

Listing 6: Code snippet from the FastApi app

```

1  @app.get("/metrics")
2  def metrics():
3      response_str = "running_servers_total {}\n".format(get_running_servers↵
                    ())
4
5      for gamename, fake_server in fake_servers.items():
6          response_str += 'fake_servers_total{{gamename="{}}"}} {}\n'.format(↵
                    gamename, fake_server.running_servers)
7
8      return Response(content=response_str, media_type="text/plain")
9
10 @app.post("/fakeserver/create/{gamename}")

```

```

11 def create_fakeserver(gamename: str):
12     if gamename not in fake_servers:
13         fake_servers[gamename] = FakeServer(gamename)
14     return fake_servers[gamename].getdata()
15
16 @app.post("/fakeserver/scaleup/{gamename}")
17 def scale_up(gamename: str):
18     if gamename in fake_servers:
19         fake_servers[gamename].scale_up(1)
20     return fake_servers[gamename].getdata()
21     raise HTTPException(status_code=404, detail="Fake server not found")
22
23 @app.post("/fakeserver/scaledown/{gamename}")
24 def scale_down(gamename: str):
25     if gamename in fake_servers:
26         fake_servers[gamename].scale_down(1)
27     return fake_servers[gamename].getdata()
28     raise HTTPException(status_code=404, detail="Fake server not found")
29
30 @app.post("/fakeserver/{gamename}")
31 def get_fakeserver(gamename: str):
32     # Return the number of running servers for the given game
33     if gamename in fake_servers:
34
35         servers = fake_servers[gamename].get_running_servers()
36         #Return the number of running servers as a number
37         return {"running_servers": servers}
38     raise HTTPException(status_code=404, detail="Fake server not found")

```

When calling the metrics endpoint of this service, you will get a response as shown in listing 8. On line 1 we can see the number of real servers running on OpenStack. This refers to the number of servers not included the "TaskRunner" server. At the time of the requests, I also took a screenshot of the dashboard in OpenStack, shown in figure 8. Here, we can see that the game PLAYERUNKNOWN'S BATTLEGROUNDS has two virtual machines running, which is the same number the metrics endpoint returns.

The code for checking the number of running servers on OpenStack, use the same initialization method as shown in listing 5, and uses the code found in listing 7. It is worth noting that on line six, we only query for active servers. Doing this, forces the server to be functioning before they count. Another interesting part is that we choose to return active servers - 1 on line nine. This is because we use one server to run the code documented here.

Listing 7: OpenStack server status

```

1 def get_running_servers():
2     try:
3         # Use the existing connection to OpenStack
4         active_servers_count = 0
5         for server in conn.compute.servers():
6             if server.status == 'ACTIVE':
7                 active_servers_count += 1
8
9         return active_servers_count - 1
10

```

```

11     except Exception as e:
12         logger.error("Error in get_running_servers: {}".format(e))
13         return -1

```

Lines 3, 4, and 5 show the same game being run with different auto-scaling solutions and different maximum server limits. This highlights one of the key benefits of implementing the "fake server" approach - it allows testing various scaling algorithms side-by-side without consuming actual resources. Notably on line 2, we see a starting server count of 39 for this game. This is because the algorithm on that line was configured with a total server limit of 250, while the algorithms on lines 3 and 4 had a limit of 25 servers.

On line 5, we also see metrics for a different game - CS:GO. Including this shows how the same auto-scaling solutions can be applied across different games with minimal modification. The core algorithms do not need to be tailored specifically to a certain game's workload patterns.

This flexibility to easily simulate and experiment with many combinations of algorithms and configurations is a major advantage of the fake server approach. It allows rapid iteration to find optimal scaling policies before committing real resources. The ability to scale experiments horizontally by simulating a large server fleet is also very valuable for emulating production environments.

Listing 8: Resonse from server exporter

```

1 running_servers_total 2
2 fake_servers_total{gamename="PLAYERUNKNOWN'S BATTLEGROUNDS 2"} 39
3 fake_servers_total{gamename="PLAYERUNKNOWN'S BATTLEGROUNDS 3"} 4
4 fake_servers_total{gamename="PLAYERUNKNOWN'S BATTLEGROUNDS 4"} 4
5 fake_servers_total{gamename="Counter-Strike: Global Offensive 1"} 2

```

Instances

Instance ID ▾

Filter

Launch Instance

Delete Instances

More Actions ▾

Displaying 3 items

<input type="checkbox"/>	Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Age	Actions
<input type="checkbox"/>	PLAYERUNKNOWN'S BATTLEGROUNDS-GameServer-2	GameServer	10.196.38.8	C1R1_5G	-	Active	nova	None	Running	7 minutes	Create Snapshot
<input type="checkbox"/>	PLAYERUNKNOWN'S BATTLEGROUNDS-GameServer-1	GameServer	10.196.38.234	C1R1_5G	-	Active	nova	None	Running	18 minutes	Create Snapshot
<input type="checkbox"/>	TaskRunner	Ubuntu-22.04-LTS	10.196.37.200	C2R4_10G	ThinkPad	Active	nova	None	Running	4 weeks, 1 day	Create Snapshot

Figure 8: OpenStack Compute Dashboard

3.2.3 Auto-scaler

During the project phase, I tested a lot of different scaling methods, and it was therefor important to let the scaling service handle this requirement. At the same time, I felt it was important to make the service as generic as possible, and the .env file was therefore designed to handle different games, number of servers, scaling schema and the server type.

On line 1 in listing 9 the game to be scaled is listed. In this example, we have six instances of POGB. On line two, we define the max numbers of servers for the game. This number is used to calculate the capacity of the servers. In a production environment, the capacity would be pre-defined, which would need a slight rewrite of Scaling/app/app.py, and include the capacity in the env file. On line three, we define the scaling schema, and on line four, we define if we want to scale servers on OpenStack, or just create fake servers for testing.

Line five to eight, describe the setup for the real server. The script would need some adaption to handle different images / flavors if we wanted to do "real" scaling of more than one game.

Listing 9: Scaler service .env file

```
1 GAME_NAME=PLAYERUNKNOWN'S BATTLEGROUNDS,PLAYERUNKNOWN'S BATTLEGROUNDS,↵
    PLAYERUNKNOWN'S BATTLEGROUNDS,PLAYERUNKNOWN'S BATTLEGROUNDS,↵
    PLAYERUNKNOWN'S BATTLEGROUNDS,PLAYERUNKNOWN'S BATTLEGROUNDS
2 NUMBER_OF_SERVERS=7,250,250,250,250,250
3 SCALING_SCHEME=1,1,2,3,4,5
4 FAKE = 0,1,1,1,1,1
5 IMAGE_NAME=Ubuntu-20.04-LTS
6 FLAVOR_NAME=C1R1_5G
7 NETWORK_NAME=acit
8 INSTANCE_BASE=GameServer
```

3.3 Scaling experiments

For the scaling experiments, I mainly focused on the game PUBG, since this is one of the games with the greatest changes in players during a 48 hour period. To force the algorithm to actively scale, I defined the numbers of servers to 250. This setup gave a base-load = 60 000 players (this is the capacity when we have 0 servers running), and a capacity per server = 12179

Scaler solution 1

I decided to start off with one of the most basic scaler schemas to create a baseline scaler.

Scale up when: $U \geq 0.80$

Scale down when: $U \leq 0.40$

Where U is how full the server is (from 0 to 1) On each calculation step, the algorithm gets the current number of players, current numbers of servers and calculate a server capacity percentage. The interesting part of the implementation of this solution, can be seen in listing 10.

Listing 10: Scaling solution 1

```

1  while capacity_percentage > 80:
2      required_servers += 1
3      if required_servers + current_servers >= self.↵
         number_of_servers:
4          break
5      capacity_percentage = self.calc_capacity(current_servers + ↵
         required_servers, current_players)
6
7      # Scale down if below 40% capacity but ensure that we do not scale↵
         down below 1 server
8      while capacity_percentage < 40 and current_servers + ↵
         required_servers > 1:
9          required_servers -= 1
10         if required_servers + current_servers <= 1:
11             break
12         new_capacity_percentage = self.calc_capacity(current_servers +↵
         required_servers, current_players)
13         # If scaling down once would bring us above 40%, we check to ↵
         see if we should scale down
14         if new_capacity_percentage >= 40:
15             break # We've reached an optimal number of servers
16         capacity_percentage = new_capacity_percentage
17
18     return required_servers

```

While this approach works fine when scaling up, it does not scale down efficiently. As seen in figure 9 there are several issues with this scaling method. However, ill describe the issue I decided to fix for the next scaling method.

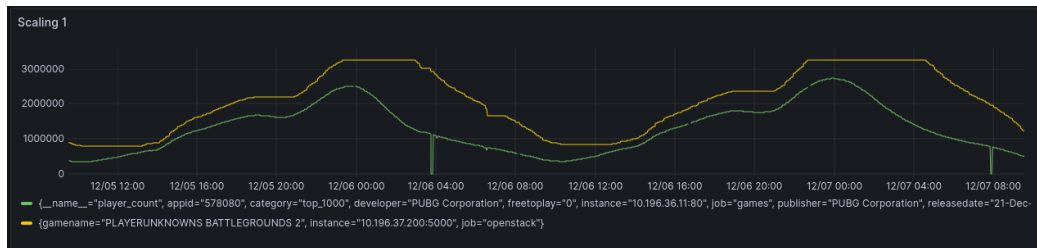


Figure 9: 48hr run with scaler 1

From figure 9 one can see that the numbers of servers won't decrease when the server load decrease. While too fast scaling can be an issue, I was not expecting the numbers of servers being constant over six hours from 22:00 to 04:00. Even worse, this will always happen at max capacity, since the rules for this scaling method are much more sensitive when it comes to up scaling. The issue is how the rules were defined. Since we scale up on 80% capacity

usage, and down on 40% there will always be a delay during the scale down phase. And on top of that, during the scale down phase, we will still have more servers than needed online, because of the 40% constraint. To fix this, I started to work on the algorithm for scaler 2.

Scaler solution 2

This scaling solution is a copy of solution 1, with one crucial change. While calculating the server percentage, we do no longer care about the 40% limit. What the algorithm does is that it calculates what is the minimum number of servers needed to have a usage percentage below 80%. As can be seen from figure 10 this allows for a faster down scaling phase, and not only that, during the down scaling phase, the server count is the same as it would be for the upscaling phase. This is one major improvement from scaler solution 1. To implement the change, the code from line 8 to 16 in listing 10 was changed with the code presented in listing 11. This code recalculate the server capacity with one less server and check if we still are within the desired capacity.

Listing 11: Scaling solution 2

```

1      while capacity_percentage < s_down and current_servers + <←
2          required_servers > 1:
3              required_servers -= 1
4              new_capacity_percentage = self.calc_capacity(current_servers + <←
5                  required_servers, current_players)
6              if new_capacity_percentage >= s_up:
7                  required_servers += 1 # Go one step back up
8                  break # Stop scaling down if we reach or exceed the s_up <←
9                      threshold
10             elif current_servers + required_servers < 1:
11                 required_servers += 1 # Ensure we don't go below 1 server
12                 break
13             capacity_percentage = new_capacity_percentage

```

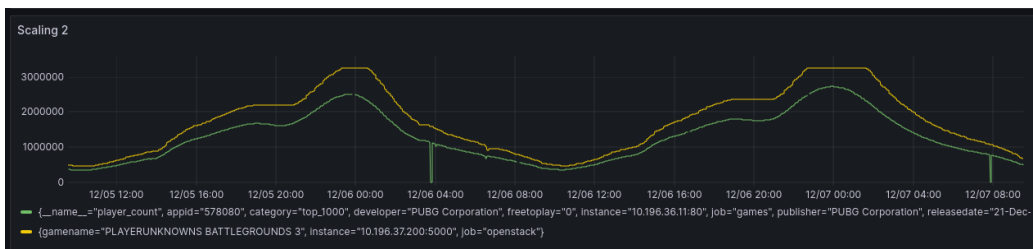


Figure 10: 48hr run with scaler 2

However, even with this new scaling approach, there are some issues that need to be solved. First on my list was to build in some more "predicting" features.

Scaling solution 3

This scaling solution is based on the improvements done in solution 2, but adds a rate of change variable to the calculation. The idea was that since we only get data every five minutes, we could benefit from predicting what the player count would be at the next data point. This is done with a simple calculation, where we take:

$$playerChange = (current_p - current_{p_{last_point}}) / time_{delta}$$

To implement this in code, the changes introduced in solution 2, were replaced with the code in listing 12. This value is then added to the current player's count before doing the capacity calculations. The solution works well, but it reinforces the over provisioning seen during peak hours, as seen in Figure 11.

Listing 12: Scaling solution 3

```

1      while capacity_percentage > s_up:
2          required_servers += 1
3          if required_servers + current_servers >= self.↵
4              number_of_servers:
5              break
6          capacity_percentage = self.calc_capacity(current_servers + ↵
7              required_servers, predicted_player_count)
8
9          # Scale down if below s_down capacity, but not below s_up ↵
10         capacity
11     while capacity_percentage < s_down and current_servers + ↵
12         required_servers > 1:
13         required_servers -= 1
14         new_capacity_percentage = self.calc_capacity(current_servers +↵
15             required_servers, predicted_player_count)
16         if new_capacity_percentage >= s_up:
17             required_servers += 1 # Go one step back up
18             break # Stop scaling down if we reach or exceed the s_up ↵
19             threshold
20         elif current_servers + required_servers < 1:
21             required_servers += 1 # Ensure we don't go below 1 server
22             break
23
24     capacity_percentage = new_capacity_percentage

```

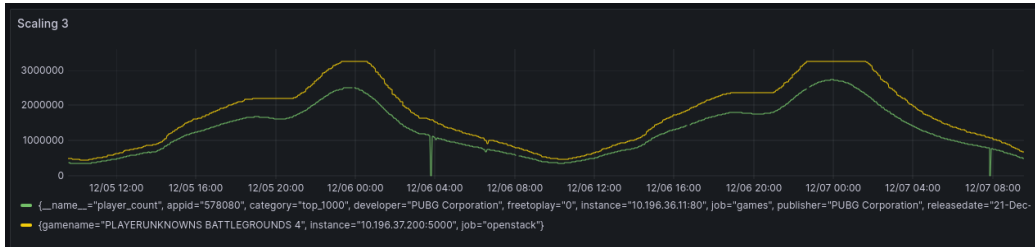


Figure 11: 48hr run with scaler 3

After analyzing this issue, I came to the conclusion that the root cause of this is the percentage calculation. At minimum capacity, 20% free capacity

on the server might add up to room for another 1000 players, while at peak capacity, it could add up to 100 000 players. So instead of working on a fix for solution 3, I started to prototype a solution that would use the number of free server spots instead of using the capacity percentage as the deciding factor.

Scaling solution 4

This solution mimics solution 2, with one change, it scales from a fixed number of free server spots, instead of a percentage. This avoids the over provisioning during peak hours, and greatly reduce the need for servers. However, the solution is also a bit more risky in terms of not being able to scale for a huge increase in players over a time step.

From figure 12 we can immediately see that we scale servers much closer to the actual needs.

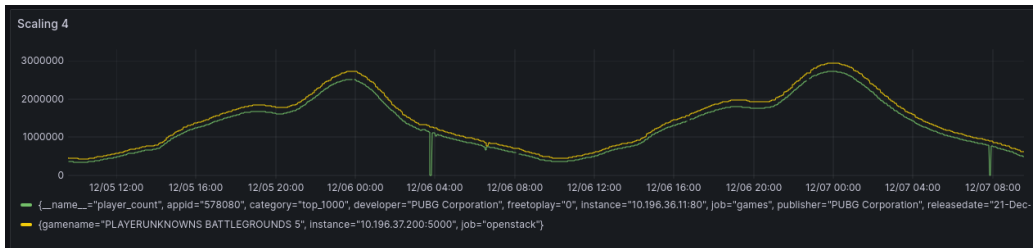


Figure 12: 48hr run with scaler 4

In the field of programming, you can often see a function grow and grow while you try to implement changes and improve on the code. This is not always a good thing, and it is not the case with solution 4. Here we see that the entire function in listing 13 is written in the same amount of lines as the changes introduced in listing12. The reason for this is that we can calculate the needed amount of servers, purely based on how many users we have at the moment, with the added free spots.

Listing 13: Scaling solution 4

```

1  def scaling_4(self, current_servers, current_players, free_spots=1000)↵
2      :
3      """
4      Scale on free player spots instead of server capacity %
5      """
6
7      required_servers = 0
8      # Calculate the current server status
9      current_capacity = (int(current_servers) * self.↵
10         capacity_per_server)
11      current_free_spots = current_capacity - current_players
12      logger.debug(f"Current free spots: {current_free_spots}")
13      # Calculate the target capacity
14      target_capacity = current_players + free_spots

```

```

13         # Calculate the required servers
14         required_servers = (target_capacity - current_capacity) / self.capacity_per_server
15         # Round up to the nearest integer
16         required_servers = math.ceil(required_servers)
17         logger.debug(f"Required servers: {required_servers}")
18         return required_servers

```

Scaling solution 5

Scaling solution 5 tries to take the improvements from solution 4, and add a dynamic layer to it, namely the change in players from the last data points. It will then assume a linear growth in players, and add this number in before scaling. This solution seems to work well, even if can provide one or two more servers than solution 4. As seen on line nine in listing 14, the only change in solution 5 is that we adjust the free_spots to include the player_change rate, before running solution 4. This will make the solution more robust when facing periods of rapid change of online players.

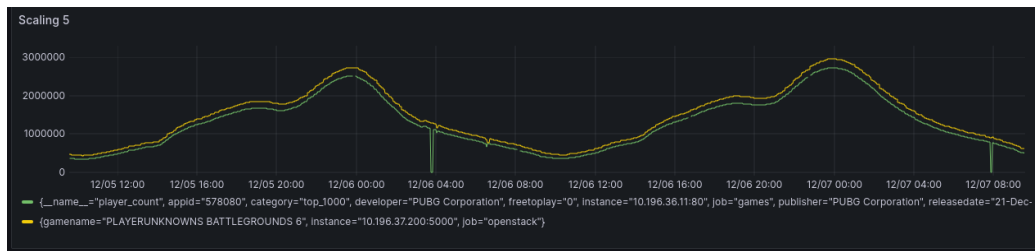


Figure 13: 48hr run with scaler 5

Comparing figure 12 and 13 show that they performed more or less the same. In table 1 we confirm that this is more or less the case as well, but that solution 5 is a bit more aggressive while scaling up.

Listing 14: Scaling solution 5

```

1         now = datetime.datetime.now()
2         time_delta = (now - self.previous_time).total_seconds() / 60
3         self.previous_time = now
4         # Calculate the rate of change of player count
5         player_change_rate = (current_players - self.previous_player_count
6                                ) / time_delta
7         # New capacity is player count + free spots + rate of change
8         new_capacity = current_players + free_spots + player_change_rate
9         return self.scaling_4(current_servers, new_capacity, free_spots=
                                free_spots)

```

3.4 Scaling solutions overview

To get a better overview of the different solutions, figure 14 show the five different solutions together, along with the current number of servers online

for each solution. From the chart, we can see that there are almost no difference between solution 2 and 3, and the same can be said for solution 4 and 5. I find it interesting that solution 4 and 5 never uses the maximum number of servers, and seems to scale much more conservative.



Figure 14: Scaling services compared against each other

3.5 Solution 5 with real servers

After testing and finding the best scaling solution, a 24 hour test was run using the seven VM's we had for this project. As illustrated in figure 15, the solution works good with fewer servers, we did however have an issue with players disconnecting at around 23:00, causing us to scale down too fast and experiencing a brief moment of being under provisioned. To account for this, it might be wise to add a gliding average or a similar method to account for sudden drops in players.

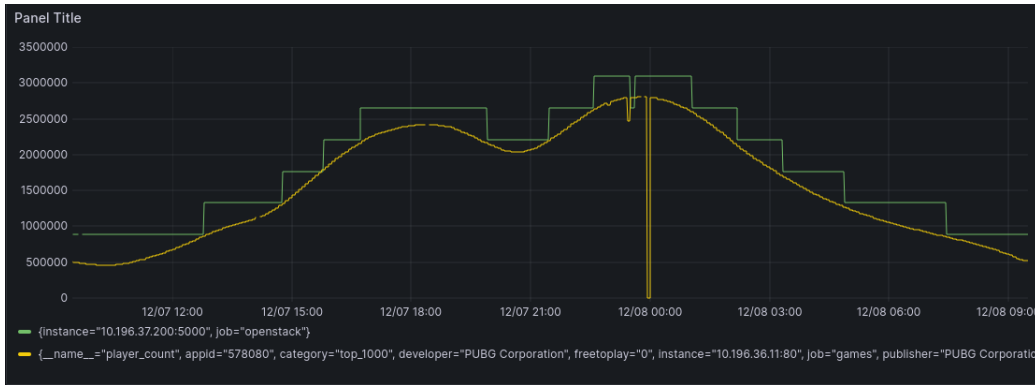


Figure 15: Scaling PUBG with 7 VM's on OpenStack - 24hr

3.6 Scaling solution on other games

Since these algorithms should work regardless of the games they are scaling for, I preformed a test for CS:GO as well. As one can see from figure 16

the scaling solution works well for this game as well. More testing should be done with other games as well to detect potential corner cases, but it seems that the solution is general and will fit whatever games we need to scale for.



Figure 16: CS:GO Scaled with solution 5

It is worth noting that in figure 16, it seems that the free server space is much higher than in figure 9 to 13, but this is due to the scale of the chart, since CS:GO has fewer players than PUBG.

3.7 Hot swap Scaler Schema

In a late stage of the project, the idea of hot swapping scaler schema got introduced. The idea is as follows: During normal weekdays, we run the overall most cost-effective algorithm we have. However, being cost-effective, it might not be capable to tackle fast increase of new players. To achieve this, some minor changes must be done to the current code, creating new endpoints in Listing 6, and connecting those endpoints to functions both in the OpenStack Exporter and Scaler app. First, creating two new endpoints, one for setting the scaling solution and one to retrieve the current solution. Then we need to ensure that we check what scaling method to use each time the scaling functions get invoked.

Automating the process of scaling methods can be realized by implementing a cron script. This script can be set to activate in response to specific triggers, such as an alarm indicating that servers are under-provisioned. Alternatively, the process can also be carried out manually. This dual approach offers a balance between flexibility and robustness, allowing us to adapt to different scaling needs efficiently.

In scenarios where we believe it can come a significant increase in player numbers, especially due to a scheduled event, a strategy can be fine-tuned. By adjusting the scaler solution, we can enable a more rapid scaling up of resources. This proactive adjustment can be important for ensuring that we are prepared to handle the increase in demand. The goal here is to ensure

that the increase in player traffic does not negatively impact the user experience. By scaling up more quickly, we can provide a smooth, uninterrupted service, maintaining high levels of user satisfaction and engagement during peak times. This approach not only helps in managing the server load effectively, but also plays a vital role in maintaining the reputation of our service as reliable and user-friendly, especially during times of high demand.

After a scheduled event ends, there is a couple of options for managing our scaling strategy. One approach is to revert directly back to our original scaling scheme, which is a straightforward method of returning to our standard operational mode. This option is ideal when we expect the user traffic to normalize quickly after the event.

Alternatively, we can go for a more aggressive scale-down strategy initially. This involves temporarily switching to a scheme that rapidly reduces resources to more closely match the decreased demand post-event. This aggressive scale-down is might be useful in scenarios where there is a sharp drop in user numbers immediately after the event, as it allows for more efficient resource management and cost savings.

Once this phase of aggressive scale-down is complete, we can then switch back to our normal scaling scheme. This staged approach to scaling down ensures that we're not maintaining unnecessarily high resource levels, which can be costly, while still being prepared for any unexpected variations in user traffic.

3.8 Lessons from development

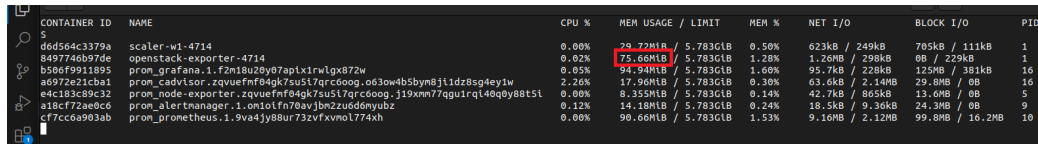
In the course of developing the project, I stumbled upon a particularly challenging error: the server was consistently running out of memory. The root of this problem was traced back to a flaw within the "get_running_servers" method. To provide some context, this function was initially designed at the early stages of the project. Its primary role was to obtain the number of running servers from OpenStack. To do this, it establishes a new connection to OpenStack each time the metrics endpoint received a call. Interestingly, during the initial testing phase and even throughout extended test runs spanning 24 hours, this approach didn't seem to cause any issues. However, a critical point to note is that after the service had been running for a period ranging between 36 and 48 hours, the server would simply crash. After discussing with our cloud engineers, I realized that the server crash was due to running out of memory.

This memory overload was not just a minor inconvenience but a significant problem. The direct consequence of the Out of Memory (OOM) situation was that the TaskRunner Virtual Machine (VM) would cease to

function, essentially freezing up, and necessitating a complete reboot. The first solution was to bring a bigger hammer, scaling the TaskRunner up to the most powerful server we had access to. This however did not solve the problem, but made the service run for longer before it froze up again.

Unraveling the mystery behind this bug was not straightforward and required a substantial amount of investigative work. During this process, I made numerous enhancements to the code. One significant change was transitioning the entire OpenStack exporter application from Flask to FastAPI, which is known for its efficiency and speed. In addition, I developed a more sophisticated method for managing server scaling within OpenStack, coupled with refinements to the algorithm responsible for calculating scaling needs. Despite these extensive modifications, the memory issue persisted.

Ultimately, I turned to a more hands-on approach to diagnose the problem: monitoring the different containers through the "docker status" command. This close observation was probably where I should have started, as it quite quickly lead me to the right spot. It became evident that the issue lay within the open-stack-exporter container. I noticed a rapid and continuous increase in its memory usage, starting from a modest 30MiB and escalating to 75MiB (as highlighted by the red box in figure 17). This increase didn't stop, but kept on growing until it reached the maximum limit allowed. This observation was crucial in identifying and addressing the underlying issue with the memory.



CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PID
5							
d6d564c3379a	scaler-wt-4714	0.00%	29.72MiB / 5.78GiB	0.50%	623kB / 249kB	705kB / 111kB	1
8497746b97de	openstack-exporter-4714	0.02%	75.66MiB / 5.78GiB	1.28%	1.26MB / 298kB	8B / 229kB	1
b5e6f9911895	prom_grafana.1.f2m18u20y07apt1x1rwlgx872w	0.05%	94.94MiB / 5.78GiB	1.60%	95.7kB / 228kB	125MB / 381kB	16
a6972e21cb1	prom_cadvisor.zqvvefn04gk7su5l7qrc6oog.o63ow4b5byn8j1ldz8sg4ey1w	2.26%	17.96MiB / 5.78GiB	0.30%	63.4kB / 2.14MB	29.6MB / 8B	16
e4c183c89c32	prom_node-exporter.zqvvefn04gk7su5l7qrc6oog.j19xnm77aguirql46qy88t5l	0.00%	8.35MiB / 5.78GiB	0.14%	42.7kB / 865kB	13.6MB / 8B	5
a18c772ae8cc	prom_alertmanager.1.onio1ofn78avjbn2zud6dyubz	0.12%	14.18MiB / 5.78GiB	0.24%	18.5kB / 9.36kB	24.3MB / 8B	9
cf7cc6a903ab	prom_prometheus.1.9va4jy88ur73zvfxxmo1774xh	0.00%	90.66MiB / 5.78GiB	1.53%	9.16MB / 2.12MB	99.8MB / 16.2MB	10

Figure 17: Screenshot of output from "docker status"

To solve the issue, all that needed to be done was to ensure that we only connected the app to OpenStack once, so the fix ended up being moving a couple of lines outside a definition.

4 An evaluation

At the beginning of the project, I envisioned an AI algorithm to scale fast and efficient. However, during the design phase, it quickly became clear that for this problem, a rule-based approach would do the scaling to a satisfying level. The choice to not delve into the realm of AI algorithms is therefore

	Solution 1	Solution 2	Solution 3	Solution 4	Solution 5
Min	1	1	1	4	4
Max	250	250	250	227	228
Avg	156	128	128	106	106

Table 1: Data about number of servers from 27.11.2023 11:45 to 04.12.2023 11:45

an active choice to avoid creating additional overhead and complexity to the system.

4.1 Server capacity evaluation

From the data shown in section 3.3 we can see that all servers stay in capacity at all times. This is also true for longer timeframes, and we can therefore conclude that the rule-base algorithm does its job, but we can also see that we over provision servers, especially for some of the algorithms. From table 1 the issues discussed in section 3.3 become clear. Solution 1 to 3, has higher highs and lower lows than Solution 4 and 5. The section where solution 1 to 3 goes down to only one server is due to the player number is below the baseload of 60 000 players. Since we are calculating percentages, the required servers move towards 0, and the calculations have issues caching up when the players join the game again. Therefore, we have a brief moment in time where we are under provisioned. This is not the case with solution 4 and 5, since these solutions will scale with the same factor even if there are few players online.

It is also worth noting that solution 4 and 5 has an average of 106 servers, while solution 2 and 3 has an average of 128 servers and solution 1 has an average of 156 servers during this time span. This gives an improvement of 36% for solution 4 and 5 compared to solution 1 as shown in eq. 1, and an improvement of 17.2% for solution 4 and 5 compared to solution 2 and 3, as shown in eq. 2

$$\frac{156 - 106}{156} \times 100 = 32.05\% \quad (1)$$

$$\frac{128 - 106}{128} \times 100 = 17.19\% \quad (2)$$

When comparing these solutions with our current solution, which is running all the servers all the time we need to make one assumption. Since committing to rent a server for a longer time span often results in a discount, we need to define this discount to make fair comparisons. A rule of thumb is

that you get about 10% discount when renting long term, but for this evaluation, we'll use a discount of 20%. This means that we can calculate the difference between running 200 servers and 106 servers for solution 4 and 5.

$$\frac{200 - 106}{200} \times 100 = 47.00\% \quad (3)$$

From eq. 3 we see that we will save 47% implementing the most optimized scaling solution for the game.

Another interesting path to explore is to provide more base-load servers. From table 1 we can see that the two best solutions, runs on average 106 servers, and they never got below four servers. This might leave some room to add even more longtime servers for an added saving. The drawback with this method, is that we would need to do in depth analyzes for each game we want to scale for.

Creating added overhead might be an issue if we want the scaling service to be quick and easy to deploy, but there is also the possibility that we can automate the calculations for the optimal base-load servers and scaling servers. By integrating this calculation in the pipeline, we can easily decide the number of needed servers when starting to scale for a new game.

4.2 Evaluation of scaling algorithm

The rule-based scaling algorithm used in this project provides satisfactory performance while maintaining simplicity. By relying on predefined rules rather than complex AI methods, it avoids unnecessary overhead and complexity. At the same time, as shown in Section 3.3, it maintains server capacity effectively across all solutions.

Solutions 4 and 5 in particular reduce average servers by 36% and 17% respectively compared to simpler methods. This demonstrates that the algorithm, while straightforward, can optimize resource usage well. The transparency of a rules-based system is also beneficial - behavior is predictable and easy to understand.

In conclusion, the scaling algorithm achieves a good balance. Advanced machine learning techniques could potentially improve efficiency further, but at the cost of complexity and opacity. The current approach delivers solid scaling results for this application while remaining simple and transparent. Ongoing maintenance and refinement of the rules is also straightforward for users. Overall, it meets the key project criteria of automation, efficiency and simplicity effectively given the use case.

4.3 Evaluation of added complexity

One key criterion for this project was to achieve a high degree of automation, as a solution requiring constant manual oversight would not be viable. In this regard, the implemented system succeeds in providing a robust, self-sufficient scaling solution.

The Docker containers and GitLab CI/CD pipeline enable a completely automated workflow. Any code changes pushed to the repository automatically deploy to the live containers, without any manual intervention needed. This allows the system to run independently once initially set up.

The scaling rules themselves are also simple to modify as needed. Users can tune performance by editing the scaling factor values in the `.env` file. No code changes are required. This allows new configurations to be tested and deployed instantly via the CI/CD automation.

Overall, the use of Docker combined with GitLab CI/CD results in a system that can scale on its own with minimal ongoing human input. The only manual task is occasionally changing the `.env` values as desired, which takes seconds. All other processes, from updating code to rebuilding and redeploying containers, happen automatically in the background.

This hands-off automation meets the key project goal of an independent self-managing scaling solution. Once up and running, oversight is minimal to none. The system's autonomy combined with simple configurability via `.env` makes it easy to operate and optimize on an ongoing basis. This keeps complexity low while still delivering automated, efficient scaling tailored to the game use case.

4.4 Conclusion

The aim of this project was to explore the opportunities we have when it comes to auto-scaling servers. In section 4.1 we can see an expected cost decrease of 47% compared to running maximum servers at all time. My personal recommendation is that we explore setting solution 53.3 in production, as it shows more or less the same savings as solution 4, but is a bit more robust as it provides some more servers.

However, it is worth noting that the solution is not complete, and would need more work to be ready for production. Even so, the projected savings is so good that I would recommend that we start planning for this. Since the service also works fine across all games tested during development, we now have a solution that fits all of our needs across the different games.

For the future work, we need to assess the HotSwap schema described in section 3.7, explore if we can tighten up the rules for scaling and create a

monitoring system that can alert us if something goes wrong.

To answer the questions defining the project: The pilot project has shown that it will be possible to scale quite dynamic with the change of player numbers. Given this discovery, we would measure a "successful 24 hours" to be a period with no outage or full servers.

If we compare the scaled solution against the approach to having full capacity at all times, we could save as much as 47% in server costs.

References

- [1] A. Fenko, T. Keizer, and A. Pruyn, "Do social proof and scarcity work in the online context," in *16th International Conferences on Research in Advertising (ICORIA 2017)*, 2017, pp. 1–7.
- [2] S. N. A. Jawaddi and A. Ismail, "Autoscaling in serverless computing: Taxonomy and open challenges," May 2023. [Online]. Available: <http://dx.doi.org/10.21203/rs.3.rs-2897886/v1>
- [3] T. L. F. Prometheus, "Overview: Prometheus," Jan 2014. [Online]. Available: <https://prometheus.io/docs/introduction/overview/>
- [4] Vegasbrianc, "A prometheus & grafana docker-compose stack," 2023. [Online]. Available: <https://github.com/vegasbrianc/prometheus>

5 Appendix

5.1 gitlab-ci.yml

Listing 15: .gitlab-ci.yml

```
1  stages:
2    - cleanup
3    - build
4    - deploy
5  cleanup:
6    stage: cleanup
7    script:
8      # Stop any containers with names starting with 'scaler-w1-'
9      - docker ps --aq --filter "name=scaler-w1-" | xargs --r docker ←
        stop || echo "Failed to stop some containers."
10     - docker ps --aq --filter "name=openstack-exporter-" | xargs ←
        --r docker stop || echo "Failed to stop some containers."
11     # Wait for a few seconds to ensure everything is settled
12     - sleep 10
13     - docker ps --aq --filter "name=openstack-exporter-" | xargs ←
        --r docker rm || echo "Failed to remove some containers."
```

```

14 - docker ps -aq --filter "name=scaler-w1-" | xargs -r docker↵
    rm || echo "Failed to remove some containers."
15
16
17
18 # Remove images with names containing 'scaler' or 'openstack↵
    -exporter'
19 - docker images -q | grep 'scaler' | xargs -r docker rmi || ↵
    echo "Failed to remove some scaler images."
20 - docker images -q | grep 'openstack-exporter' | xargs -r ↵
    docker rmi || echo "Failed to remove some openstack-↵
    exporter images."
21
22
23
24 scaler_build:
25   stage: build
26   script:
27     - echo "Stage build, Dockerfile:"
28     - cd apps/Scaling
29     - cat Dockerfile
30     - docker build -t scaler:$CI_PIPELINE_ID .
31
32 openstack_exporter_build:
33   stage: build
34   script:
35     - echo "Stage build, Dockerfile:"
36     - cd apps/OpenStack
37     - cat Dockerfile
38     - docker build -t openstack-exporter:$CI_PIPELINE_ID .
39
40 openstack_exporter_deploy:
41   stage: deploy
42   script:
43     - docker images | grep openstack-exporter
44     - docker run -d -p 5000:5000 --restart=unless-stopped -e ↵
        OS_AUTH_URL=$OS_AUTH_URL -e OS_PROJECT_NAME=↵
        $OS_PROJECT_NAME -e OS_USERNAME=$OS_USERNAME -e ↵
        OS_PASSWORD="$OS_PASSWORD" --name openstack-exporter-↵
        $CI_PIPELINE_ID openstack-exporter:$CI_PIPELINE_ID
45     - docker ps
46
47 scaler_deploy:
48   stage: deploy
49   script:
50     - docker run -d -p 7474:7474 --restart=unless-stopped -e ↵
        OS_AUTH_URL="$OS_AUTH_URL" -e OS_PROJECT_NAME="↵
        $OS_PROJECT_NAME" -e OS_USERNAME="$OS_USERNAME" -e ↵
        OS_PASSWORD="$OS_PASSWORD" --name scaler-w1-↵
        $CI_PIPELINE_ID scaler:$CI_PIPELINE_ID

```

5.2 OpenStack/app.py

```
1 from fastapi import FastAPI, Response, HTTPException
2 from openstack_check import get_running_servers
3 from fake_servers import FakeServer
4 import logging
5 from typing import Dict
6
7 app = FastAPI()
8
9 fake_servers: Dict[str, FakeServer] = {}
10
11 logging.basicConfig(
12     level=logging.WARNING,
13     filename='server_status.log',
14     format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
15 )
16
17 logger = logging.getLogger('get_running_servers')
18 logger.info("Starting OpenStack app")
19
20 @app.get("/metrics")
21 def metrics():
22     response_str = "running_servers_total {}\n".format(get_running_servers↵
23     ())
24
25     for gamename, fake_server in fake_servers.items():
26         response_str += 'fake_servers_total{{gamename="{}"}} {}{}\n'.format(↵
27             gamename, fake_server.running_servers)
28
29     return Response(content=response_str, media_type="text/plain")
30
31 @app.post("/fakeserver/create/{gamename}")
32 def create_fakeserver(gamename: str):
33     if gamename not in fake_servers:
34         fake_servers[gamename] = FakeServer(gamename)
35     return fake_servers[gamename].getdata()
36
37 @app.post("/fakeserver/scaleup/{gamename}")
38 def scale_up(gamename: str):
39     if gamename in fake_servers:
40         fake_servers[gamename].scale_up(1)
41         return fake_servers[gamename].getdata()
42     raise HTTPException(status_code=404, detail="Fake server not found")
43
44 @app.post("/fakeserver/scaledown/{gamename}")
45 def scale_down(gamename: str):
46     if gamename in fake_servers:
47         fake_servers[gamename].scale_down(1)
48         return fake_servers[gamename].getdata()
49     raise HTTPException(status_code=404, detail="Fake server not found")
50
51 @app.post("/fakeserver/{gamename}")
52 def get_fakeserver(gamename: str):
53     # Return the number of running servers for the given game
54     if gamename in fake_servers:
```

```
53
54     servers = fake_servers[gamename].get_running_servers()
55     #Return the number of running servers as a number
56     return {"running_servers": servers}
57     raise HTTPException(status_code=404, detail="Fake server not found")
58
59 if __name__ == "__main__":
60     import uvicorn
61     uvicorn.run(app, host="0.0.0.0", port=5000)
```

5.3 OpenStack/fake_servers.py

```
1 class FakeServer:
2     def __init__(self, gamename):
3         self.gamename = gamename
4         self.running_servers = 0
5
6     def getdata(self):
7         # Return a dict with the game name and the number of running ↵
8             servers
9             return {"title": self.gamename, "running_servers": self.↵
10                 running_servers}
11
12     def scale_up(self, num_servers):
13         self.running_servers += num_servers
14
15     def scale_down(self, num_servers):
16         self.running_servers -= num_servers
17
18     def get_running_servers(self):
19         return self.running_servers
```

5.4 OpenStack/Dockerfile

```
1 # Use a Python 3.9 slim image as the base
2 FROM python:3.9-slim
3
4 # Install required packages including FastAPI and Uvicorn
5 RUN pip install fastapi uvicorn openstacksdk
6
7 # Copy scripts into the container
8 COPY /app/ /app/
9
10 # Set the working directory
11 WORKDIR /app
12
13 # Expose the port the app runs on
14 EXPOSE 5000
15
16 # Command to run the application with Uvicorn
17 CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "5000"]
```

5.5 Scaling/app.py

```
1  from utils import get_min_max, get_running_servers, get_current_players
2  from scale import Scaler
3  import logging
4  import os
5  import time # import the time module
6  import dotenv
7  import random
8
9
10
11  dotenv.load_dotenv()
12
13  logging.basicConfig(
14      level=logging.INFO,
15      filename="scaler.log", # Set the minimum log level (DEBUG, INFO, ↵
16      format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
17  )
18
19  logger = logging.getLogger("scaler")
20
21  ## INIT
22  # Set up variables
23  games = os.getenv("GAME_NAME")
24  nub_server = os.getenv("NUMBER_OF_SERVERS")
25  scaling_scheme = os.getenv("SCALING_SCHEME")
26  fake_list = os.getenv("FAKE")
27
28
29  # Split the environment variable strings into lists
30  game_list = games.split(",")
31  nub_server_list = nub_server.split(",")
32  scaling_scheme_list = scaling_scheme.split(",")
33  fake_list = fake_list.split(",")
34
35  # Resetting the game_count for the new iteration
36
37  game_count = {}
38  for game in game_list:
39      game_count[game] = game_count.get(game, 0) + 1
40  game_occurrence = {}
41  list_of_games = []
42  for idx, game in enumerate(game_list):
43      # Count the game occurrence for suffixing
44      game_occurrence[game] = game_occurrence.get(game, 0) + 1
45      # Format the game name with occurrence number if more than one, or ↵
46      # with "1" if it occurs only once
47      if game_count[game] > 1:
48          game_name = f"{game} {game_occurrence[game]}"
49      elif game_count[game] == 1: # Check if there's only one game of this ↵
50          type
51          game_name = f"{game} 1"
52      else:
53          game_name = game
```



```

52
53     list_of_games.append(
54         [
55             game_name.strip(), # .strip() to remove leading/trailing ↵
56                 whitespace
57             int(nub_server_list[idx]),
58             int(scaling_scheme_list[idx]),
59             int(fake_list[idx]),
60         ]
61     )
62
63
64
65     logger.info(f"Game: {games}")
66     logger.info(f"Number of servers available: {nub_server}")
67     logger.info(f"Scaling scheme: {scaling_scheme}")
68
69
70     def setup_g(game, nub_server, capacity_per_server, baseload, number = 1, ↵
71         scaler=1, fake=False):
72         """Calculate the baseload for the game"""
73         print(game)
74         p_min, p_max = get_min_max(game=game)
75         try:
76             p_min = int(p_min)
77             p_max = int(p_max)
78         except:
79             raise ValueError(f"p_min or p_max is not an integer: {p_min}, {↵
80                 p_max} for game {game}")
81
82         # Round down to nearest 5000 for easier calculations
83         p_min = p_min - (p_min % 5000)
84         # Round up to nearest 1000 for easier calculations
85         p_max = p_max + (1000 - (p_max % 1000))
86         # Set baseload to p_min
87         if baseload != 0:
88             baseload = p_min
89         # Calculate server capacity
90         fluctuation = p_max - baseload
91         # Lets leave 10% of capacity for growth
92         if capacity_per_server == 0:
93             capacity_per_server = int(fluctuation * 1.1 / nub_server)
94
95         logger.info(f"Game: {game}, Baseload: {baseload}, Capacity per server:↵
96             {capacity_per_server}, ServerNumber: {number}")
97         logger.info(f"Number of servers available: {nub_server}, Fake: {fake}, ↵
98             Scaling scheme: {scaler}")
99
100         scaling_solution = Scaler(
101             game=game,
102             scaling_scheme=scaler,
103             baseload=baseload,
104             capacity_per_server=capacity_per_server,
105             server_suffix = number,
106             number_of_servers = nub_server,
107             fake=fake,
108         )
109         logger.info(
110             f"Game: {game}, Baseload: {baseload}, Capacity per server: {↵
111                 capacity_per_server}"
112         )

```

```

108         return scaling_solution
109
110
111     scaler_list = []
112     for game in list_of_games:
113         logger.info(f"Setting up scaler for {game}")
114         game_name = game[0]
115         # The game name should be Game Name 1, Game Name 2, etc. if there are ↵
116         # multiple games of the same type
117         # Strip the number from the name and add it to a number var
118         if game_name.rstrip().endswith(('1', '2', '3', '4', '5', '6', '7', '8', ↵
119         , '9', '0')):
120             # Split only on the last space to get the number
121             game_name, game_number = game_name.rsplit(' ', 1)
122             game_number = int(game_number)
123         # And remove the number from the name
124         nub_server = int(game[1])
125         scaling_scheme = int(game[2])
126         fake = int(game[3])
127         scaler_list.append(
128             setup_g(
129                 game=game_name,
130                 number = game_number,
131                 nub_server=nub_server,
132                 scaler=scaling_scheme,
133                 capacity_per_server=0,
134                 baseload=1,
135                 fake=fake
136             )
137         )
138
139     ## MAIN LOOP
140     while True: # This will keep running indefinitely
141         # Call the scaler function with the scaling scheme selected, baseload, ↵
142         # and capacity per server
143         for scaler in scaler_list:
144             scaler.scale()
145             time.sleep(10)
146         time.sleep(
147             300
148         ) # Introduce a delay of 240 seconds (4 minutes) before the next ↵
149         iteration

```

5.6 Scaling/fakeserver.py

```
1 import requests
2 import logging
3 logger = logging.getLogger('scaler')
4
5 # Create a fake serverstack for a game
6 class FakeServer:
7     def __init__(self, gamename, max_servers=10):
8         self.max_servers = max_servers
9         self.gamename = gamename
10        self.servers = 0
11        self.server_address = "http://10.196.37.200:5000"
12        self.create_fakeserver()
13
14
15    def scale_up_servers(self, nr_of_servers):
16        logger.info(f"Scaling up {nr_of_servers} fake servers")
17        for i in range(abs(nr_of_servers)):
18            self.scale_up()
19            self.servers += 1
20
21    def scale_down_servers(self, nr_of_servers):
22        logger.info(f"Scaling down {nr_of_servers} fake servers")
23        for i in range(abs(nr_of_servers)):
24            self.scale_down()
25            self.servers -= 1
26
27    def create_fakeserver(self):
28        url = f"{self.server_address}/fakeserver/create/{self.gamename}"
29        response = requests.post(url)
30        logger.debug(response.json())
31        return response.json()
32
33    def scale_up(self):
34        url = f"{self.server_address}/fakeserver/scaleup/{self.gamename}"
35        response = requests.post(url)
36        logger.debug(response.json())
37        return response.json()
38
39    def scale_down(self):
40        url = f"{self.server_address}/fakeserver/scaledown/{self.gamename}"
41
42        response = requests.post(url)
43        logger.debug(response.json())
44        return response.json()
45
46    def get_servers(self):
47        url = f"{self.server_address}/fakeserver/{self.gamename}"
48        response = requests.post(url)
49        data = response.json()
50        number_of_servers = data['running_servers']
51        self.servers = int(number_of_servers)
52
53    return int(number_of_servers)
```


5.7 Scaling/openstackutils.py

```
1  from openstack import connection
2  from utils import get_running_servers
3  import os
4  import logging
5  logger = logging.getLogger('scaler')
6  import dotenv
7  dotenv.load_dotenv()
8
9  class OpenStackManager:
10
11     def __init__(self, game):
12         # Add game name
13         self.conn = connection.Connection(
14             auth_url=os.getenv("OS_AUTH_URL"),
15             project_name=os.getenv("OS_PROJECT_NAME"),
16             username=os.getenv("OS_USERNAME"),
17             password=os.getenv("OS_PASSWORD"),
18             user_domain_id='default',
19             project_domain_id='default'
20         )
21         self.image_name = os.getenv("IMAGE_NAME")
22         self.flavor_name = os.getenv("FLAVOR_NAME")
23         self.network_name = os.getenv("NETWORK_NAME")
24         self.instance_base = os.getenv("INSTANCE_BASE")
25         self.instance_base = f"{game}-{self.instance_base}"
26
27     def scale_up_servers(self, num_servers):
28         """Scale up the number of servers based on the given image."""
29         num_servers = int(num_servers)
30         running_servers = get_running_servers()
31         for i in range(num_servers):
32             instance_name = f"{self.instance_base}-{running_servers+1+i}"
33             self.create_instance(instance_name)
34
35     def scale_down_servers(self, num_servers):
36         """Scale down the number of servers based on the given image."""
37         num_servers = int(abs(num_servers))
38         for i in range(num_servers):
39             logger.info(f"Deleting server {i+1}")
40             self.delete_instance()
41
42
43     def create_instance(self, instance_name):
44         """Create a new VM instance."""
45         try:
46             image = self.conn.compute.find_image("Ubuntu-20.04-LTS")
47             flavor = self.conn.compute.find_flavor(self.flavor_name)
48             network = self.conn.network.find_network(self.network_name)
49
50             server = self.conn.compute.create_server(
51                 name=instance_name,
52                 image_id=image.id,
53                 flavor_id=flavor.id,
54                 networks=[{"uuid": network.id}],
```

```

55         )
56
57         server = self.conn.compute.wait_for_server(server)
58         logger.info(f"Created server {server.name} with id {server.id}↵
59         ")
60         return server.name
61     except Exception as e:
62         logger.error(f"Error starting new VM {e}")
63         return None
64
65 def delete_instance(self):
66     """Delete an instance and its attached volumes."""
67     try:
68         # Fetch all servers
69         servers = list(self.conn.compute.servers())
70
71         # Filter servers with the specified instance base name
72         filtered_servers = [server for server in servers if self.↵
73                             instance_base in server.name]
74
75         # If no matching servers, nothing to delete
76         if not filtered_servers:
77             logger.warning(f"No instances found with base name {self.↵
78                             instance_base}!")
79             return
80
81         # Sort filtered servers by created date
82         instance_id = sorted(filtered_servers, key=lambda s: s.↵
83                             created_at, reverse=True)[0]
84         instance = self.conn.compute.get_server(instance_id)
85         # Now delete the instance
86         self.conn.compute.delete_server(instance_id)
87         logger.info(f"Deleted instance {instance_id}")
88
89     except Exception as e:
90         logger.error(f"Error in deleting instance or its volumes: {e}↵
91         ")
92
93
94 if __name__ == "__main__":
95     osm = OpenStackManager("Test")
96     inp = ""
97     while inp != "Q":
98         inp = input("Scale [U]p or [D]own?")
99         if inp == "U":
100             osm.scale_up_servers(1)
101         elif inp == "D":
102             osm.scale_down_servers(1)
103         else:
104             print("Invalid input")

```

5.8 Scaling/scale.py

```
1 from utils import get_min_max, get_running_servers, get_current_players
2 from openstackutils import OpenStackManager
3 from fakeserver import FakeServer
4 import datetime
5 import math
6 import logging
7 logger = logging.getLogger('scaler')
8
9 class Scaler:
10     """
11     Scaler class to scale the game servers based on the scaling scheme.
12     """
13     def __init__(self, game, scaling_scheme, baseload, capacity_per_server, ↵
14         server_suffix, number_of_servers, fake=False):
15         """
16         scaling_scheme: 1 = Scale based on current players
17                        2 = Scale based on current players and time of day
18                        3 = Scale based on current players and derivative ↵
19                           of player count
20
21         """
22         self.game = game
23         self.scaling_scheme = int(scaling_scheme)
24         self.baseload = baseload
25         self.capacity_per_server = int(capacity_per_server)
26         self.server_name = f"{game} {server_suffix}"
27         self.number_of_servers = number_of_servers
28
29         # Just init to something
30         self.current_players = 1000
31         self.current_players = int(self.get_current_players())
32         self.previous_player_count = self.current_players
33         self.previous_time = datetime.datetime.now()
34
35         self.fake = fake
36         if self.fake:
37             self.VMCONTROLLER = FakeServer(gamename=self.server_name)
38             self.current_servers = self.VMCONTROLLER.get_servers()
39         else:
40             self.VMCONTROLLER = OpenStackManager(game)
41             self.current_servers = int(self.get_running_servers())
42
43     def get_current_players(self):
44         cur_player = get_current_players(self.game)
45         # Fix for the data not being updated
46         if cur_player != 0:
47             self.current_players = cur_player
48         return self.current_players
49
50
51
52     def get_running_servers(self):
```

```

53         # Return the current number of running servers.
54         if self.fake == True:
55             return self.VMCONTROLLER.get_servers()
56         return get_running_servers()
57
58     def create_instance(self, nr_of_servers):
59         self.VMCONTROLLER.scale_up_servers(nr_of_servers)
60
61
62     def delete_instance(self, nr_of_servers):
63         self.VMCONTROLLER.scale_down_servers(nr_of_servers)
64         logger.info(f"Deleted {nr_of_servers} server instances")
65
66
67     def scale(self):
68         current_players = self.get_current_players()
69         current_servers = self.get_running_servers()
70         required_servers = 0
71         logger.info(f"Current players: {current_players}, Current servers:↵
72             {current_servers}, Scaling scheme: {self.scaling_scheme}, ↵
73             server capacity: {self.capacity_per_server*current_servers}, ↵
74             Baseload: {self.baseload}")
75
76         # Scaling decision based on the scaling scheme
77         if self.scaling_scheme == 1:
78             required_servers = self.scaling_1(current_servers, ↵
79                 current_players)
80         elif self.scaling_scheme == 2:
81             required_servers = self.scaling_2(current_servers, ↵
82                 current_players, s_up=80, s_down=75)
83         elif self.scaling_scheme == 3:
84             required_servers = self.scaling_3(current_servers, ↵
85                 current_players, s_up=80, s_down=75)
86         elif self.scaling_scheme == 4:
87             required_servers = self.scaling_4(current_servers, ↵
88                 current_players)
89         elif self.scaling_scheme == 5:
90             required_servers = self.scaling_5(current_servers, ↵
91                 current_players)
92         else:
93             required_servers = self.scaling_1(current_servers, ↵
94                 current_players)
95         logger.info(f"Required servers: {required_servers}")
96
97         self.previous_player_count = current_players
98
99         # Ensure that we do not scale above the maximum number of servers
100         if required_servers + current_servers > self.number_of_servers:
101             required_servers = self.number_of_servers - current_servers
102             logger.warning(f"Max servers :( ): {required_servers}")
103         # Ensure that we do not scale below 1 server
104         elif required_servers + current_servers < 1:
105             required_servers = 1
106
107         if required_servers > 0:
108             self.create_instance(required_servers)
109         elif required_servers < 0:
110             self.delete_instance(required_servers)
111
112     def calc_capacity(self, current_servers, current_players):
113         # Calculate the capacity of the current number of servers

```



```

106         # Calculate the current capacity
107         logger.debug(f"Current players: {current_players}")
108         logger.debug(f"Current servers: {current_servers}")
109         current_players = int(current_players) - self.baseload
110     try:
111         current_capacity = (int(current_servers) * self.↵
            capacity_per_server)
112     except:
113         current_capacity = 0
114     # Based on current players, calculate the target capacity
115     target_capacity = current_players
116     # Calculate the percentage of capacity
117     if current_capacity == 0:
118         current_capacity = 1000
119     capacity_percentage = target_capacity / current_capacity * 100
120     logger.debug(f"Current players above baseload: {current_players}")
121     logger.debug(f"Current capacity: {current_capacity}")
122     logger.debug(f"Target capacity: {target_capacity}")
123     logger.debug(f"Baseline: {self.baseload}")
124     logger.debug(f"Current capacity: {capacity_percentage}")
125
126     # Make sure that the percentage is not above bellow 0% or above ↵
        200%
127     if capacity_percentage < 0:
128         capacity_percentage = 0
129     elif capacity_percentage > 200:
130         capacity_percentage = 200
131
132
133     return capacity_percentage
134
135 def scaling_1(self, current_servers, current_players):
136     """
137     Scale based on current players
138     Scale up if servers are on 80% capacity
139     Scale down if servers are on 50% capacity
140     """
141     logger.debug(f"Scaling scheme 1")
142     required_servers = 0
143     capacity_percentage = self.calc_capacity(current_servers, ↵
        current_players)
144
145     # Scale up if above 80% capacity
146     while capacity_percentage > 80:
147         required_servers += 1
148         if required_servers + current_servers >= self.↵
            number_of_servers:
149             break
150         capacity_percentage = self.calc_capacity(current_servers + ↵
            required_servers, current_players)
151
152     # Scale down if below 40% capacity but ensure that we do not scale↵
        down below 1 server
153     while capacity_percentage < 40 and current_servers + ↵
        required_servers > 1:
154         required_servers -= 1
155         if required_servers + current_servers <= 1:
156             break
157         new_capacity_percentage = self.calc_capacity(current_servers +↵
            required_servers, current_players)
158     # If scaling down once would bring us above 40%, we check to ↵
        see if we should scale down

```

```

159         if new_capacity_percentage >= 40:
160             break # We've reached an optimal number of servers
161             capacity_percentage = new_capacity_percentage
162         logger.debug(f"New capacity: {capacity_percentage}")
163         logger.debug(f"Required servers: {required_servers}")
164         return required_servers
165
166     def scaling_2(self, current_servers, current_players, s_up=80, s_down=
167         =40):
168         """
169         Scale based on current players
170         Scale up if servers are on s_up capacity
171         Scale down if servers are on s_down capacity, but not below s_up ↵
172         capacity
173
174         """
175         logger.debug("Scaling scheme 2")
176         required_servers = 0
177         capacity_percentage = self.calc_capacity(current_servers, ↵
178             current_players)
179
180         # Scale up if above s_up% capacity
181         while capacity_percentage > s_up:
182             required_servers += 1
183             if required_servers + current_servers >= self.↵
184                 number_of_servers:
185                 break
186             capacity_percentage = self.calc_capacity(current_servers + ↵
187                 required_servers, current_players)
188
189         # Scale down if below s_down capacity, but not below s_up capacity
190         while capacity_percentage < s_down and current_servers + ↵
191             required_servers > 1:
192                 required_servers -= 1
193                 new_capacity_percentage = self.calc_capacity(current_servers + ↵
194                     required_servers, current_players)
195                 if new_capacity_percentage >= s_up:
196                     required_servers += 1 # Go one step back up
197                     break # Stop scaling down if we reach or exceed the s_up ↵
198                         threshold
199                 elif current_servers + required_servers < 1:
200                     required_servers += 1 # Ensure we don't go below 1 server
201                     break
202                 capacity_percentage = new_capacity_percentage
203
204         logger.debug(f"New capacity: {capacity_percentage}")
205         logger.debug(f"Required servers: {required_servers}")
206         return required_servers
207
208     def scaling_3(self, current_servers, current_players, s_up=80, s_down=
209         =40):
210         """
211         Scale servers based on the rate of change of player count (↵
212             derivative).
213
214         :param current_servers: The current number of servers.
215         :param current_players: The current number of players.
216         :param time_delta: The time passed since the last measurement in ↵
217             minutes.
218         :return: The number of servers to scale up or down.
219         """
220         required_servers = 0

```

```

210     now = datetime.datetime.now()
211     time_delta = (now - self.previous_time).total_seconds() / 60
212     self.previous_time = now
213     # Calculate the rate of change of player count
214     if current_players - self.previous_player_count == 0:
215         player_change_rate = 0
216     else:
217         player_change_rate = (current_players - self.previous_player_count) / time_delta
218
219     # Predict player count in 5 minutes
220     predicted_player_count = current_players + player_change_rate
221
222     # Calculate the current load percentage
223     capacity_percentage = self.calc_capacity(current_servers, predicted_player_count)
224
225     while capacity_percentage > s_up:
226         required_servers += 1
227         if required_servers + current_servers >= self.number_of_servers:
228             break
229         capacity_percentage = self.calc_capacity(current_servers + required_servers, predicted_player_count)
230
231         # Scale down if below s_down capacity, but not below s_up
232         while capacity_percentage < s_down and current_servers + required_servers > 1:
233             required_servers -= 1
234             new_capacity_percentage = self.calc_capacity(current_servers + required_servers, predicted_player_count)
235             if new_capacity_percentage >= s_up:
236                 required_servers += 1 # Go one step back up
237                 break # Stop scaling down if we reach or exceed the s_up threshold
238             elif current_servers + required_servers < 1:
239                 required_servers += 1 # Ensure we don't go below 1 server
240                 break
241
242         capacity_percentage = new_capacity_percentage
243
244     logger.debug(f"New capacity: {capacity_percentage}")
245     logger.debug(f"Required servers: {required_servers}")
246     return required_servers
247
248 def scaling_4(self, current_servers, current_players, free_spots=1000):
249     """
250     Scale on free player spots instead of server capacity %
251     """
252
253     required_servers = 0
254     # Calculate the current server status
255     current_capacity = (int(current_servers) * self.capacity_per_server)
256     current_free_spots = current_capacity - current_players
257     logger.debug(f"Current free spots: {current_free_spots}")
258     # Calculate the target capacity
259     target_capacity = current_players + free_spots
260     # Calculate the required servers

```

```

261         required_servers = (target_capacity - current_capacity) / self.↵
                capacity_per_server
262         # Round up to the nearest integer
263         required_servers = math.ceil(required_servers)
264         logger.debug(f"Required servers: {required_servers}")
265         return required_servers
266
267     def scaling_5(self, current_servers, current_players, free_spots=5000)↵
        :
268         """
269         Combine scaling 3 and 4
270         Calculate free spots based on the derivative of player count
271         """
272         required_servers = 0
273         now = datetime.datetime.now()
274         time_delta = (now - self.previous_time).total_seconds() / 60
275         self.previous_time = now
276         # Calculate the rate of change of player count
277         player_change_rate = (current_players - self.previous_player_count↵
                ) / time_delta
278         # New capacity is player count + free spots + rate of change
279         new_capacity = current_players + free_spots + player_change_rate
280
281         return self.scaling_4(current_servers, new_capacity, free_spots=↵
                free_spots)

```

5.9 Scaling/utils.py

```
1 import requests
2 from typing import List, Dict, Union, Optional
3 import time
4
5
6 # Base URL for Prometheus server
7 PROMETHEUS_URL = "http://10.196.37.200:9090"
8
9 def fetch_data_from_prometheus(endpoint: str, params: Dict[str, Union[str, ←
    int]]) -> List[Dict]:
10     """Fetch data from Prometheus using given endpoint and parameters."""
11     response = requests.get(f"{PROMETHEUS_URL}/{endpoint}", params=params)
12
13     if response.status_code != 200:
14         raise ValueError(f"Invalid response from server: {response.↵
            status_code} - {response.text}")
15
16     return response.json()['data']['result']
17
18 def get_prometheus_historical_data(query: str, start: int, end: int, step: ←
    int) -> List[Dict]:
19     """Fetch historical data from Prometheus."""
20     params = {'query': query, 'start': start, 'end': end, 'step': step}
21     return fetch_data_from_prometheus("api/v1/query_range", params)
22
23 def get_prometheus_latest_data(query: str) -> List[Dict]:
24     """Fetch the latest data from Prometheus."""
25     results = fetch_data_from_prometheus("api/v1/query", {'query': query})
26
27     # Extract the latest value if available
28     for item in results:
29         if 'value' in item:
30             item['values'] = [item['value']]
31             del item['value']
32     return results
33
34 def server_status(instance: str = "10.196.37.200:5000",
35                   start: Optional[int] = None,
36                   end: Optional[int] = None,
37                   step: Optional[int] = None,
38                   historical: bool = False) -> List[Dict]:
39     """Return the status of a server."""
40     query = f'running_servers_total{{instance="{instance}", job="openstack↵
        }}"'
41     if historical:
42         if not all([start, end, step]):
43             raise ValueError("start, end, and step must be specified for ↵
                historical data")
44         return get_prometheus_historical_data(query, start, end, step)
45     else:
46         return get_prometheus_latest_data(query)
47
48 def game_status(game: str,
49                 instance: str = "10.196.36.11:80",
```

```

50         start: Optional[int] = None,
51         end: Optional[int] = None,
52         step: Optional[int] = None,
53         historical: bool = False) -> List[Dict]:
54     """Return the status of a game."""
55     query = f'player_count{{title="{game}", instance="{instance}", job="↔
56         games"}}'
57     if historical:
58         if not all([start, end, step]):
59             raise ValueError("start, end, and step must be specified for ↔
60                 historical data")
61         return get_prometheus_historical_data(query, start, end, step)
62     else:
63         return get_prometheus_latest_data(query)
64
65 def game_player_stats(game: str, instance: str = "10.196.36.11:80") -> ↔
66     Dict[str, Union[str, int]]:
67     """
68     Get the max and min player counts for the specified game from ↔
69     Prometheus data.
70
71     Args:
72     game: Name of the game.
73     instance: The instance to be queried.
74
75     Returns:
76     A dictionary containing the game's title, minimum, and maximum player ↔
77     count.
78     """
79
80     # Set the start time to a far past value (e.g., 10 years ago)
81     start = int(time.time()) - (10 * 365 * 24 * 3600) # 10 years * 365 ↔
82     days/year * 24 hours/day * 3600 seconds/hour
83
84     # Calculate the range in seconds from the start date to now
85     range_seconds = int(time.time()) - start
86
87     # Convert the range to a more human-readable format for the Prometheus↔
88     query
89     if range_seconds < 60:
90         duration = f"{range_seconds}s"
91     elif range_seconds < 3600:
92         duration = f"{range_seconds // 60}m"
93     elif range_seconds < 86400:
94         duration = f"{range_seconds // 3600}h"
95     else:
96         duration = f"{range_seconds // 86400}d"
97
98     # Fetch all player counts over the specified range
99     query = f'player_count{{title="{game}", instance="{instance}", job="↔
100         games"}}[{duration}]'
101     results = get_prometheus_latest_data(query)
102
103     if not results or 'values' not in results[0]:
104         return {
105             'title': game,
106             'min': None,
107             'max': None
108         }
109
110     # Extract player counts from the results
111     counts = [int(value[1]) for value in results[0]['values']]

```

```

104
105     # Find the maximum player count
106     max_count = max(counts)
107
108     # Find the minimum player count where the count is greater than 0
109     min_count = min([count for count in counts if count > 0], default=None↵
110                      )
111
112     return {
113         'title': game,
114         'min': min_count,
115         'max': max_count
116     }
117
118 def get_min_max(game:str)->List[int]:
119     # Clean the response and return min and max
120     resp = game_player_stats(game=game)
121     return resp['min'],resp['max']
122
123 def get_current_players(game:str)->int:
124     # Clean the response and return min and max
125     resp = game_status(game=game)
126     try:
127         players = int(resp[0]['values'][0][1])
128     except IndexError:
129         # Happens when the endpoint is down
130         players = 0
131     return players
132
133 def get_running_servers()->int:
134     # TODO FIX THIS TO WORK WITH FAKE SERVERS AS WELL
135     # Clean the response and return min and max
136     resp = server_status()
137     print(resp)
138     servers = int(resp[0]['values'][0][1])
139     return servers
140
141
142
143
144 if __name__ == "__main__":
145     print(get_current_players("Team Fortress 2"))
146     x = game_player_stats(game="Team Fortress 2")
147     print("Min:", x['min'], "Max:", x['max'])
148     print(get_running_servers())

```

5.10 Scaling/Dockerfile

```
1 FROM python:3.8-slim
2
3 # Install required packages
4 RUN pip install openstacksdk python-dotenv
5
6
7 # Copy scripts into the container
8 COPY /app/ /app/
9
10 # Set working directory
11 WORKDIR /app
12
13 # Expose the port the app runs on
14 EXPOSE 7474
15
16 # Command to run the application
17 CMD ["python", "app.py"]
```


5.11 Scaling/.env

```
1  GAME_NAME=PLAYERUNKNOWN'S BATTLEGROUNDS,PLAYERUNKNOWN'S BATTLEGROUNDS,↵
    PLAYERUNKNOWN'S BATTLEGROUNDS,PLAYERUNKNOWN'S BATTLEGROUNDS,↵
    PLAYERUNKNOWN'S BATTLEGROUNDS,PLAYERUNKNOWN'S BATTLEGROUNDS,Counter-↵
    Strike: Global Offensive
2  NUMBER_OF_SERVERS=7,250,250,250,250,250,200
3  SCALING_SCHEME=5,1,2,3,4,5,5
4  FAKE = 0,1,1,1,1,1,1
5  IMAGE_NAME=Ubuntu-20.04-LTS
6  FLAVOR_NAME=C1R1_5G
7  NETWORK_NAME=acit
8  INSTANCE_BASE=GameServer
```