**Assignment 2**

**Szymon Pawlica**

**R00187226**

**Part 1**

In the following report we will determine the appropriate ratio/percentage of containers for each microservice, this will be done by examining each of their features and requirements; assuming the non-implemented microservices are ignored.

**Achievement Service:**  
The achievement tracking microservice needs to ensure that rewards are granted in a timely and accurate manner. This microservice will likely have high traffic and require frequent updates, so it may benefit from a higher number of containers. A ratio of about 30% maxSurge, 20% maxUnavailable and 20% of the total containers in the Kubernetes cluster may be appropriate for this microservice.

**Game Download Service:**  
The game download microservice requires high availability and scalability to accommodate many concurrent downloads. This microservice will likely have spikes in traffic during game launches and updates, so it may benefit from automatic scaling capabilities. A ratio of about 25% maxSurge, 15% maxUnavailable and 30% of the total containers in the Kubernetes cluster may be appropriate for this microservice.

**Purchase Authentication Service:**  
The purchase authentication microservice requires secure and reliable communication with payment gateways and third-party services. This microservice may have lower traffic compared to other microservices, but it requires high availability and reliability to ensure that players can purchase and access game content without interruption. A ratio of about 20% maxSurge, 10% maxUnavailable and 25% of the total containers in the Kubernetes cluster may be appropriate for this microservice.

**Registration Service:**  
The registration service requires persistent storage and reliable data processing capabilities to ensure that player data is stored correctly. This microservice may have low to moderate traffic, but it requires high availability and scalability to accommodate a growing number of players. A ratio of about 25% maxSurge, 10% maxUnavailable and 25% of the total containers in the Kubernetes cluster may be appropriate for this microservice.

**Client Service:**

The client service is responsible for providing the application with simulated user traffic. It doesn’t require reliable communication, data processing, or persistent storage. However, it requires the other services to have these capabilities. The application will start with a small number of replicas; this number can be gradually increased to achieve the desired level of performance. This approach is known as a "ramp-up" strategy and can help identify any bottlenecks or performance issues in the application.

Considering all the suggested microservice the ratio would look something like this. Assuming certain content is going to be cached on the users’ devices, more replicas can be used in more demanding areas.

* **Achievement Service:** 10%
* **Game Download Service:** 15%
* **Purchase Authentication Service:** 15%
* **Registration Service:** 10%
* **Login Service:** 15%
* **Game Catalogue Service:** 2%
* **Points Service:** 10%
* **Profile Service:** 2%
* **Discovery Service:** 5%
* **News Service:** 5%
* **User Content Service:** 10%

**Part 2**

**Functional Test**

In both screenshots below the starting from the left, the python servers represent the purchase\_auth service | game\_download service | achievement service.

Below, using postman, I test purchasing a new game, the “transaction” goes through, the game is available for download and the achievement tracking is updated.

Graphical user interface, text

Description automatically generated

Once again, using postman, I test purchasing a new game, however, this time the transaction fails. Therefore, there are no messages passed to the game download and achievement services.

A screenshot of a computer

Description automatically generated with medium confidence

**Part 3**

A serverless achievement tracking function would fit into the backend architecture of this application, it would handle events generated by the game and update the achievement status where appropriate.

Input: The game application would generate events in response to player activities, such as finishing a level, beating a boss, or achieving a goal.

Processing: The serverless achievement tracking function would be triggered by these events and would process them, extracting relevant information about the player's progress and updating their achievement status accordingly.

Output: The function would then store this updated information in a database, which in this case is a TinyDB database.

Here is the pseudocode for this function:

A screenshot of a computer

Description automatically generated with medium confidence

When triggered the function updates the ‘completed’ status of the specified achievement\_id in the TinyDB database.

Additional functionality of this function would include updating the players individual achievement lists, where they would be able to track their achievement progress. The profile service, however, hasn’t been implemented.