Nice Piano

Cloud Computing 2020/2021

Matteo Orsini 1795119
Fabrizio Rossi 1815023

Sapienza
Università di Roma

Contents

Chapter 1

Introduction

In this project we realized an online web application, called **NicePiano**, which enables users to play a virtual piano together with other users, in real time. The application can be used via a connected keyboard (or digital piano) which supports the **Musical Instrument Digital Interface (MIDI)** protocol, which sends messages to the browser according to the standard specification. When a message is received, it is parsed and the corresponding action is executed, for example, playing or stopping a note.

Moreover, the system supports multiple "virtual sessions" called **rooms**, that enable users to send and receive events only to/from users in the same room, creating a logical separation.

Another feature of NicePiano is the possibility for a user to **record** its performance which is going to be saved in his account, and can be played back in any other room where the user is currently connected to. There is also the option to **loop** the played recording, meaning that when it ends, it will be started again from the beginning without the need of any user interaction.

Finally, when a user is connected to a room, he can check which users are currently connected to the same room and there is also an indicator to show the **latency** of each user with respect to the server, which is a fundamental metric to play together with other users in time.

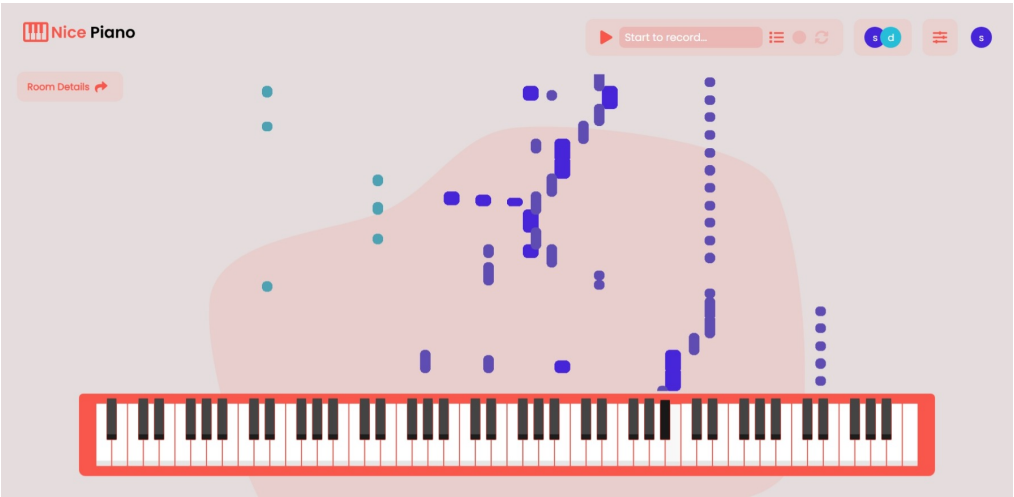

Figure 1.1: UI of a NicePiano room

Chapter 2

Design

In this chapter we describe the design of the system, defining a general architecture and how the different components interact with each other.

In Figure 2.1 we show the connections between the components defining the **architecture**, indicating which protocol they use to communicate.

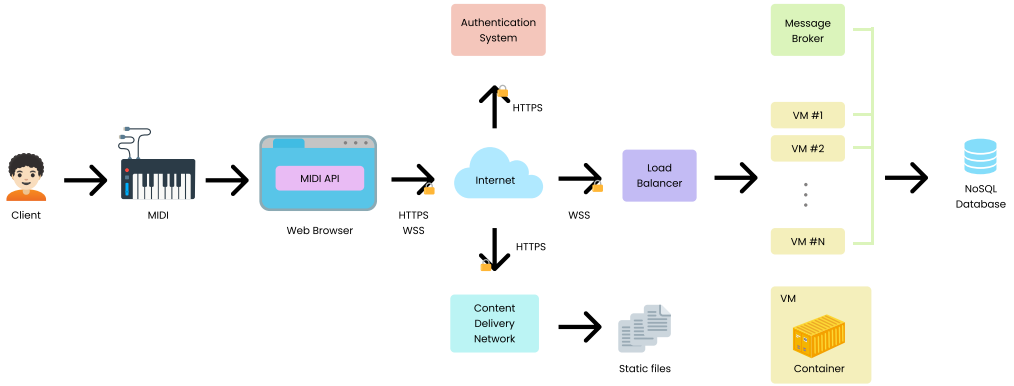

Figure 2.1: Architecture of the system

As we can see, we start from the **MIDI-enabled physical device** owned by the user, which using a USB cable is going to send MIDI messages to the **browser**, integrating a MIDI API to be able to receive and parse them. One important factor to consider is that current browsers require **HTTPS** connections in order to access to the MIDI feature, throwing errors otherwise and making the application not work as intended.

Then, the browser requests the user interface to our **CDN**, which is going to serve static files containing the necessary elements to interact with the system from the outside. In order to be able to use the platform, a user needs to create an account and log in, procedures which are both carried out by sending requests to the **authentication system**. Once a user gets access to the system, he is going to be placed into a private virtual room (or the one indicated by the URL) and then, he gets connected to the **back end** using a **Web Socket**. This connection is handled by a **load balancer**, which is going to forward it to one of the multiple VMs running a containerized version of the server logic.

When a server receives a message from a client connected to a certain room, like play or stop a particular note, it is going to **broadcast** it to all the clients which are connected to the same room. However, different users can join the same room, while being connected to different server instances, since connections are distributed by the

load balancer. This means that the two users will not be able to play together since even if they are connected to a room with the same name, the two instances will not communicate the activity of the users to each other. For this reason, we needed a mechanism to deliver messages across instances, which is what the **Message Broker** accomplishes using a publish-subscribe model for events.

At the end, we have a **NoSQL database** which stores the tracks recorded by each user, in order to be able to play them in a room at a later time. For the scope of this project, the database contains just a single table, whose schema is shown in Table 2.1.

Attribute name	**Description**
username	Username of the user that recorded the track
recordingTime	Timestamp of the beginning of the recording
endTime	Timestamp of the end of the recording
name	Name of the recording
notes	List of timed events composing the track

Table 2.1: Recordings table schema

Chapter 3

Implementation

In this chapter we talk about how we implemented the whole system, focusing on which libraries and technologies have been used. Both the front end and the back end have been implemented using the **JavaScript** language, leveraging some well-established libraries.

For the front end we opted to use the famous React.js framework since we were already quite confident with it, and was powerful enough to fulfill our needs.

Some modern browsers like Chrome, Edge or Opera, support MIDI connections using the built-in API, but they don't offer a way to reproduce sounds out of the box. For this reason, we used the Tone.js library which is a **Web Audio framework**, and using the @tonejs/piano package, we were able to playback realistic piano sounds when a user plays a note on the real hardware device. This is possible, since MIDI messages are composed as shown in Table 3.1.

Name	Size (bits)	Description
Channel	4	Number indicating one of the 16 possible MIDI channels
Event type	4	Describes an action, like hold down a note, release a note, hold down a pedal
Data	8	Generally indicates the pitch of a note or the kind of pedal
	8	Generally indicates the velocity of the event

Table 3.1: Structure of MIDI messages

In order to handle authentication we used the aws-amplify library which can make users access to the system without requiring us to run a separate server for authentication purposes, since it is all handled by **Amazon Cognito**.

Regarding the back end, the server logic has been written in JavaScript thanks to the Node.js runtime, and then we created a `Dockerfile` in order to specify how to build a Docker image, and run it using a container on each virtual machine.

In the **Dockerfile**, besides the standard instructions, we defined the name of the DynamoDB table as an environment variable and specified the port which has to be exposed.

To handle real time message exchange, we opted to use **Web Sockets** which provide

a full-duplex communication between a client and a server, using a reliable transport protocol (TCP). In this case they are the perfect technology, since they reduce latency, not requiring to establish a connection for each requests, and they also support to send messages from the server to the client, without requiring the latter to initiate a communication. To ease the implementation, we decided to use the SocketIO library, both for the front and the back end, since it abstracts numerous aspects of the standard Web Socket implementation, providing also the possibility to define custom namespaces to send messages, which we used to implement **rooms**.

Moreover, as we said in Chapter 2, we needed to make servers exchange messages since users in the same room can be connected to different machines. So, we used the Redis key-value store as a message broker, in order to store and retrieve messages from all the server instances. This mechanism is greatly simplified by SocketIO, since it already implements a Redis adapter, which allowed us to focus more on the application logic instead of the message passing one.

Talking about messages, we also implemented a simple mechanism to measure the **latency** between client and server messages, using web sockets: every x seconds the server is going to broadcast a "ping" message to all the clients, which are going to respond with a "pong" message as soon as they receive it. In this way the server can collect the time at which the "pong" message has been received and compute the difference with the ping message sending time to measure the client latency.

At the end, in order to interact with the **NoSQL database**, we used the aws-sdk package to send queries to **Amazon DynamoDB**, enabling us to perform CRUD operations on the stored recordings of each user.

Chapter 4

Deployment

In this chapter, we report how we deployed our web application and how we configured the various services used. First of all, we chose to use the **Amazon Web Services (AWS)** infrastructure to deploy our platform, as it is the one we used during the Labs. We used the **AWS Educate account** provided by the course, but due to its limitations, we weren't able to use the initially planned technologies, so we used some alternative strategies as explained in the next part of this chapter.

We tried to recreate an infrastructure as close as possible to a real production environment, leveraging AWS services in order to keep our architecture as server-less as possible. So, as said in Chapter 3, we used **Amazon Cognito** to handle authentication, creating a User Pool for our application and configuring the security parameters. Moreover, in order to use the Amplify API and authenticate users from the front end, we added a new app client which is used to obtain an ID to grant the access to the requests from our application.

Concerning **persistent storage**, as we said previously, we used **DynamoDB**, creating a table containing the recorded tracks for each user, and we used the on-demand option, in order to scale the read and write capacity units automatically according to the current load.

For the **front end**, we planned to use a **CI/CD pipeline** using **AWS CodePipeline** and **CodeBuild**. In fact, the first one can detect changes from a GitHub repository, and pull the code into an **S3** bucket. Then, it is possible to define a build action that will save the output artifact again on S3. Unfortunately, at the time of writing this report, CodeBuild is not available anymore for AWS Educate accounts, so we skipped this step.

To serve the produced **static files**, we initially used the S3 option to host static websites, but since we categorically needed HTTPS to make MIDI work, we should have used a **CloudFront** distribution with a TLS certificate generated by the **AWS Certificate Manager**, but both of them are not available for the Educate accounts, so for the scope of our tests, we hosted it by ourselves with a self signed certificate.

For the **back end**, instead we were able to setup the **CI/CD pipeline** thanks to **AWS CodePipeline**, because we didn't need the build step, since the code is directly deployed using **ElasticBeanstalk**, which is also going to create the Docker image. This last one, is a service used to easily deploy web applications, which can launch EC2 instances, explicitly running the **Docker engine**, to launch and manage containers. During the setup of the service, it is possible to define a "high availability" configuration, which enables to use an auto scaling policy for the EC2 instances which are going to be deployed. For our tests, we set the minimum number of instances to 1, and the maximum to 3.

For the **scaling policy**, instead, we targeted the **CPU usage**, since we experimentally assessed that is the most used resource in a typical execution of our application, and we defined two policies: terminate one instance if the average CPU usage of all the VMs is below 20% for 1 minute, and launch a new one if it exceeds 66% for 1 minute.

Moreover, it is also possible to define the EC2 instance type deployed, and we used one of the less powerful machines, the **t2.micro**, which is also eligible for the Free Tier. In addition, in order to make the EC2 instances access to DynamoDB without specifying credentials in the code, we created a **service role** in the IAM console, gave it full DynamoDB permissions, and associated it from the ElasticBeanstalk configuration to the EC2 instances.

Another section in the configuration of ElasticBeanstalk is the **load balancer** settings. Since we needed to establish a secure Web Socket connection with the front end, we uploaded a **custom signed certificate** to AWS IAM, and used it with the Application Load Balancer, which is going to perform the TLS handshake with the client, and then forward the requests to the target instance, using a standard connection. In order to do that, inside our code directory, we needed to create a folder named `.ebextensions`, and the configuration file for the Application Load Balancer, `securelistener-alb.config`, with the following content:

```
option_settings:
  aws:elbv2:listener:443:
    ListenerEnabled: 'true'
    Protocol: HTTPS
    SSLCertificateArns: <our certificate ARN>
```

The above settings specify to the load balancer of ElasticBeanstalk that it needs to listen for HTTPS connections on the port 443 using the certificate indicated by the ARN, which uniquely identifies our certificate uploaded via IAM.

Instead, as **message broker** among the server instances, we opted for **Redis**, since it is available in the **AWS ElastiCache** service as an in-memory key-value store. Unfortunately, also this time, even if the service is listed in the AWS Educate offered services, we haven't been able to use it, obtaining various permission errors. We also tried to contact AWS support, but they confirmed us that at this time it is not possible to use it with an AWS Educate account. For this reason, for the tests, we used one of our machine in order to host a Redis server.

Chapter 5

Validation

In this chapter we describe the validation methodology that we followed in this project, which has been **more complex** due to the fact that ours is not a normal web application, but it uses the Web Socket technology. In fact, load balancing is more difficult to perform, since when a user connects to the server, the session is going to be active until he disconnects: this means that after the first connection, all the requests must be delivered to the same server, and cannot be distributed among other VMs, making the load balancing less effective.

For this reason, to test the scalability of our system, we designed a **custom Node.js script**, which tries to simulate a large amount of users compared to the computational power of our provisioned EC2 instances. To model the users behavior, we sent random socket **events at a fixed interval** of 500ms. We defined two types of events which are sent with a certain probability which is representative of the average user interactions with the system: 90% of them are **note events**, while 10% are **database requests**. In Figure 5.1 is depicted the requests behavior and we can see that at the beginning we have a **"ramp up"** phase, where we start making users connect at regular intervals: in particular every 10 seconds, 5 users will join a random room. When the total number of connected users reaches a prefixed value, which we set to 175, the connected users will **keep sending** requests for a specified amount of time, 10 minutes in our tests. After that, a **"ramp down"** phase begins, in which users start to disconnect at the same pace at which they connected, in order to validate the scaling down policy.

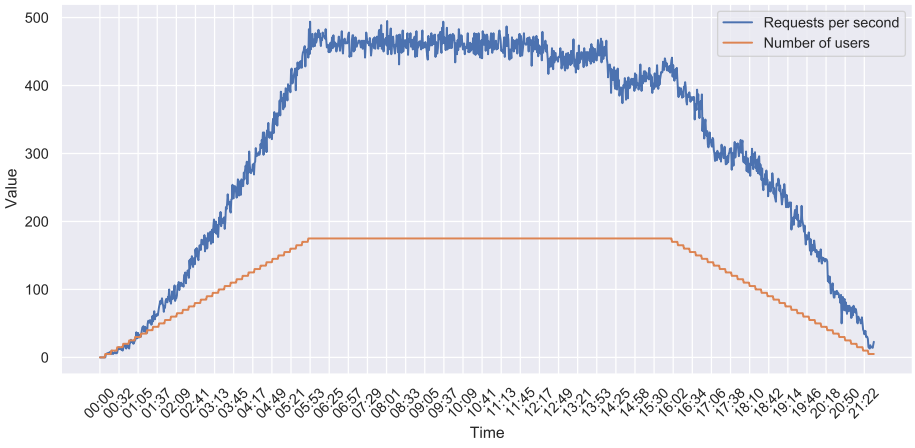

Figure 5.1: Requests per second during the test

When we ran our first tests, though, we found out that due to how Web Sockets work, once we reached the scaling up threshold (66% of CPU usage), a new EC2 would be launched, but in the meanwhile, the old instance would clog up given that new users will still connect to it, since it takes some time for a new instance to warm-up. However, when the new instance transitioned to a running state, the old one, would still be at 100% CPU usage, since new requests would still be redirected to the same instance due to the **Web Socket stickiness**. For this reason, in the user behavior we added a **disconnection probability**: every second, there is a 1% change that the user will disconnect and reconnect right away. In this way, the new connection will be established through the Application Load Balancer, which can decide to forward it to another instance with a lower CPU usage balancing the load among the instances.

Finally, in order to collect metrics about our system, we used the **CloudWatch** service, which enables us both to see real time plots to check the resources usage of the application, and also enables us to download csv files containing metrics datapoints, which we used to produce the plots shown in the next chapter. As a marginal note, we also turned on EC2 **"detailed monitoring"**, which enables a higher metrics accuracy, measuring the resource usage every minute, instead of every 5 minutes.

Chapter 6

Results

In this chapter we describe the results obtained from the validation phase conducted as described in the previous chapter.

The first results are shown in Figure 6.1, which shows the **average CPU usage** of the system, and Figure 6.2, which instead shows the **CPU usage of each EC2 instance** separately. As we can see from the first figure, the system gets to almost 100% CPU usage after about 6 minutes, with a total amount of 175 active users. After reaching the scale up threshold and waiting 1 minute, a new instance is launched, but is still not ready to handle new requests. This can be seen from the second figure, where at 09:00, we start to see the activity of the new instance. From this same plot we can also note that the first instance CPU usage is going to decrease, while the one of the second instance is going to increase, since users will randomly disconnect and connect again, and the load balancer is going to redirect new connections to the instance with the lowest CPU usage. Instead, after the 16:00 minute, it starts the ramp down phase, which can be noted from the decrease of the CPU usage in both instances. Finally, at 21:00 we can note that the scale down policy is working, since the first instance is shut down due to the fact that the CPU load is below 20%.

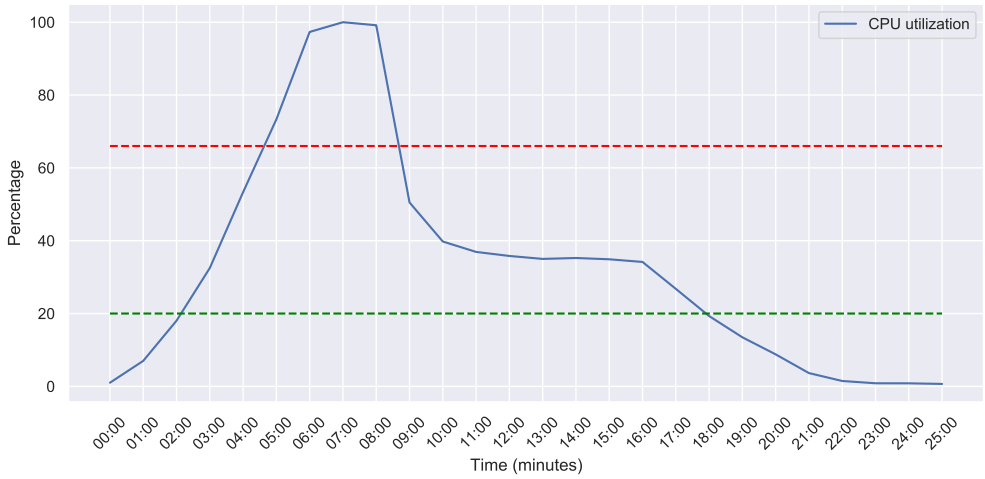

Figure 6.1: Average CPU usage during the test

One important observation to note, is that the increase of the CPU usage is **exponential** instead of being linear in the number of users connected and events received.

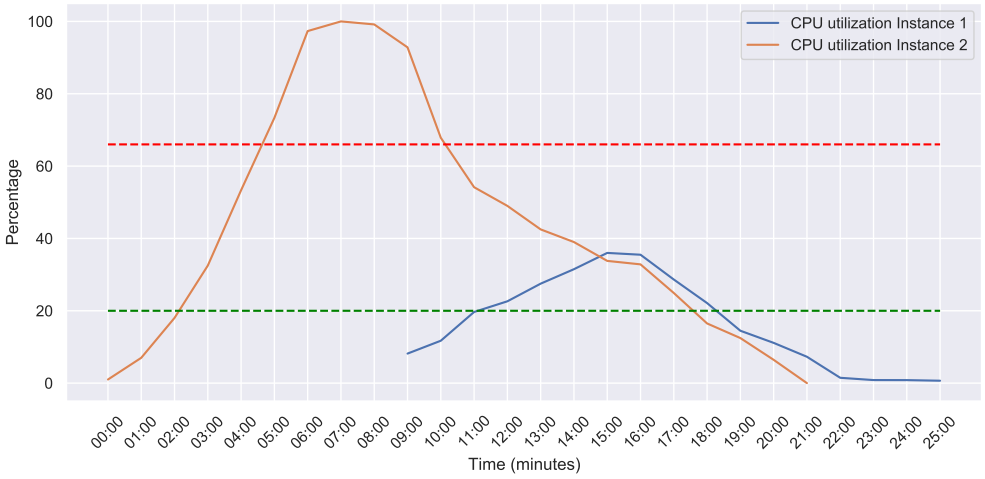

Figure 6.2: CPU usage of each instance during the test

This is because when a user event is received, it needs to be broadcasted to all the users in the room, and this can really pose a challenge to the system. For this reason, in our test we distributed the 175 evenly in 10 rooms, getting an average of 17 users in a room, which we expect to be a reasonable number in a real platform.

Moreover, in Figure 6.3, is depicted the trend of the **response time** measured by the load balancer, which is the time elapsed in seconds, after the request leaves the load balancer until a response from the target is received.

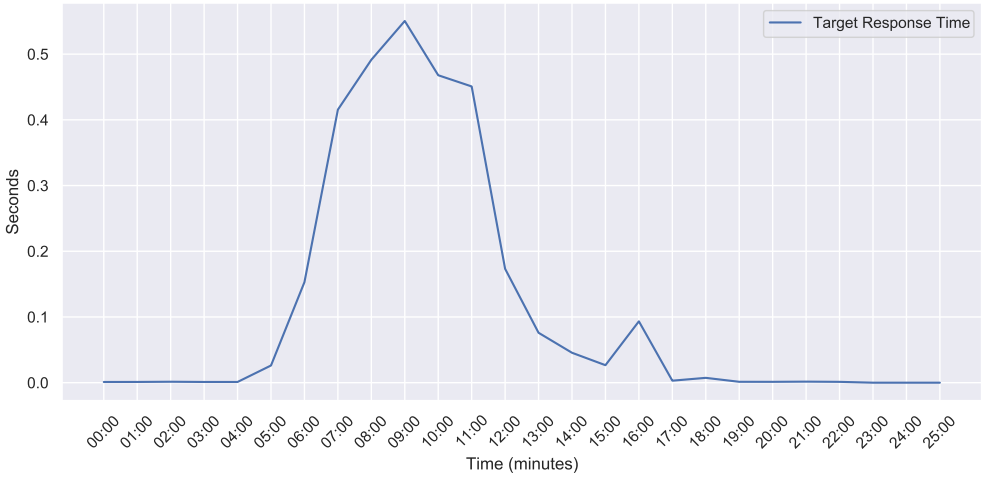

Figure 6.3: Target response time measured by the load balancer

As we can see from the plot, the response time is always good also when the system reaches 100% CPU usage, remaining always under 600ms, which means that the users playing performance are going to be very reasonable even when the system is in a critical status.

Instead, in Figure 6.4, we plotted the **number of read capacity units** which are dynamically provisioned by DynamoDB, to support the requests coming from the users.

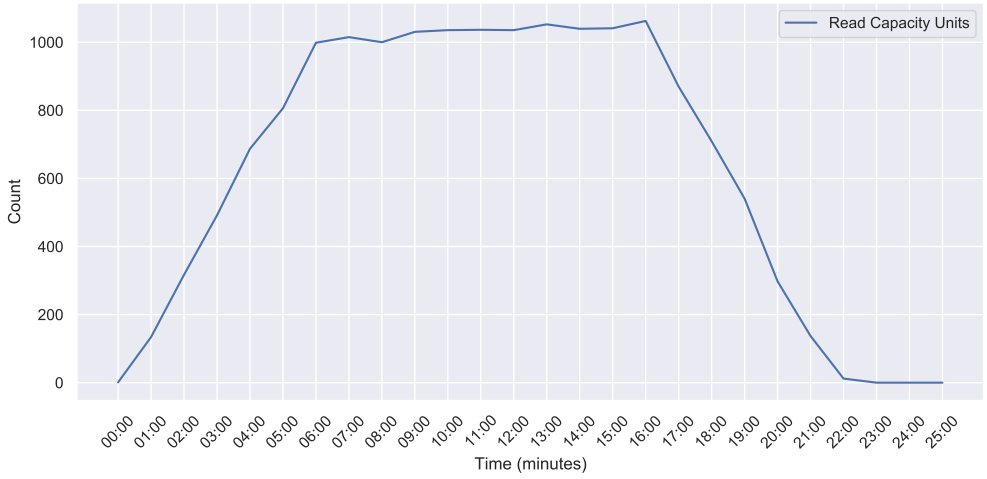

Figure 6.4: Read capacity units count of DynamoDB

As we can see, the number of units starts to increase as the number of users increases, indicating that the automatic scaling works correctly, keeping up with the load. When the number of requests decreases, instead, we can note that also the number of provisioned units decreases, meaning that we can save money thanks to the autoscaling, reducing them when they are not needed.