CLOUDS Azure Lab 2

This assignment is a series of labs using the Azure cloud. In the last assignment, you covered material from lectures 1 and 2. In this assignment, you will carry out a series of tasks that cover lectures 3 and 4.

NOTE: All the code you write in this assignment should be stored in a git repository. I suggest you one repo for all your assignments with separate folders per assignment. You can use gitlab.eurecom.fr. You will need to submit your gitlab link.

1. Create a microservice that does numerical integration

1. We saw how to do numerical integration in the class. You first task is to create a program that can do numerical integration with the function abs(sin(x)). Given an interval *lower* and *upper*, your code should break up the interval into *N* subintervals, compute the area of the rectangle at each subinterval and add them all up. For example, if you give as input 0 and 3.14159 (which is approximately π), you should get 1.99… which is close to 2 ($\int \sin x = -\cos x$, ignoring aspects of continuity, which when evaluated in the range gives you -$\cos \pi + \cos 0 = 2$). Your program should loop and repeatedly compute numerical integral for N = 10, 100, 100, 1000, 10k, 100k, 1M. You will end up getting 7 values, one for each value of N. You will see that as N increases, result converges to 2.This will also make the integral computation time consuming which will be useful for load testing later.

2. Now take the python program from above and convert it into a *microservice*. Read section 19.1 if you want to know what a microservice is if you have not read it already. You can use any framework you want (python (flask), javascript (nodejs), C# (.net)). I recommend python & flask. The microservice, when run, will take the *lower* and *upper* as arguments. These could be URL route or parameters (http://localhost:5000/numericalintegralservice/0/3.14159). What you are doing is instead of invoking the program via python interpreter, you are invoking it via a HTTP call.

3. Test your microservice locally on your computer. If you cant use a local computer, you can get a VM from Azure and do your development there. Use curl or a browser to verify that it works. Next, load test your microservice with Locust (https://locust.io/). The goal is to create many clients that all invoke the service over HTTP simulatenously. Test with 1 client. If it works, test with multiple clients. Here's a reference guide to run locust without a UI

(**https://docs.locust.io/en/stable/running-without-web-ui.html#running-without-web-ui**). You can use a simple locustfile.py like the following:

```
import time
from locust import HttpUser, task, between

class QuickstartUser(HttpUser):
    @task
    def hello_world(self):
        self.client.get("/integral/0/3.14159")
```

4. **For your deliverable, run Locust for 3 minutes and save its output. You will have stable performance. You will use the output to plot a graph later.**

2. Improving availability with scalesets

1. Read AzureMol book chapter 5 to know about Networking (vnics mainly and ssh-agent). Read chapter 6 to know about ARM templates, chapte 7 about availability, and chapter 8 about load balancing. You don't have to do the exercises if you don't want to. Now read chapter 9 with particular focus on VM scaling. You will see how all the above come together.
2. Create a VM scaleset with 2 VMs. Manually configure both VMs so that you the microservice you built in part 1 is deployed on both VMs. Note that the VMs will be behind a load balancer this time. You will be tunneling (all of this is explained in the book). Here's a reference for a quick and dirty way to configure your VMs to deploy a flask app with NGINX (https://krishansubudhi.github.io/webapp/2018/12/01/flaskwebapp.html)
3. Use your development VM or your computer to now perform a load testing on the scale set you just deployed. While the load tester is running, check Azure Insights or VM directly to see which VM is serving the load. Note the latencies reported by your load tester also.
4. While the load tester is running, manually shutdown the VM that is servicing the load. Notice what happens with the load tester – does throughput drop? Does it go back up again? Notice how load balancer redirects load to the other VM.
5. **NOTE: You will need to run Locust for 3 minutes and save its output. In the middle of this run, you will turn off the VM. You will use the output to plot a graph later.**
6. Delete the scaleset and free up all resources

3. Scaling with Azure webapps

1. In part 2, you improved availability with scaleset. If you create a custom VM image, you can configure Azure to automatically scale your VM scalset. But that's a lot of work in creating the image. In this task, you will use Azure Webapps to scale your microservice
2. First, read the part in Chapter 9 about scaling webapps (9,3 and 9.4). Deploy your microservice on Azure Webapps. Azure learn is a fantastic resource for a quick tip on how to deploy an microservice to webapps (https://learn.microsoft.com/en-us/azure/app-service/quickstart-python)
3. Configure your webapp to do auto-scaling. Start with 1 instance and scale to 3. You can use CPU load as the metric and you can use avg CPU usage is > 50% as a rule. You should use 1 minute monitoring time window so that azure reacts quickly to changes.
4. Now perform a load testing with locust again, this time with the webapp URL. Notice how performance improves after each minute. Go on Azure website and use AppInsights to see how many instances were deployed, CPU usage etc.
5. ***Run Locust for 3 minutes and save the output. Stop the webapp (don't deallocate – You will enable it before you submit the report)***

4. Scaling microservice with functions

1. So far you did scaling with webapps. You saw how much easier it is to autoscale. For the next two parts, you will explore Azure functions. In this part, you will deploy the numericalintegral code as a function and you will perform autoscaling as before. This time you will see how functions make it even easier to scale apps.
2. First, read Azuremol chapter 21 about serverless and functions. You don't have to do the labs. An even better resource is azure learn. Go ahead and read: *https://learn.microsoft.com/en-us/azure/azure-functions/create-first-function-vs-code-python?pivots=python-mode-configuration*. It is way easier to develop functions with VSCode. So follow the steps in the link above, implement a simple function on VScode, test it locally, deploy it, and test it on Azure.
3. Now, turn your numericalintegrap app into a function based on what you learned in the previous step. Deploy the function on azure. Configure the function so that it scales automatically (you can leave max instances to 200).
4. Now, repeat the locust load testing with the function endpoint. As locust runs, you can use AppInsights to see how functions are scaling and your locust performance is improving.
5. ***Run Locust for 3 minutes and save the output. Stop the function. You will enable it before***

you submit the report)

5. Implement MapReduce using Azure Durable functions

- For your final task, you will be implementing MapReduce and using it to do word count on a bunch of documents!! In doing so, you will learn about Azure Durable function and Azure Blob store.
- First, read Durable functions: https://learn.microsoft.com/en-us/azure/azure-functions/durable/quickstart-python-vscode?tabs=windows. Use the examples there to locally implement and test it locally and in Azure. Read https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-sequence?tabs=python to learn about various durable fn patterns (particularly function chaining and fan-in/fan-out). You can test the code if you want.
- Now, use the examples to implement a simple MapReduce framework. You should have **(i) MasterOrchestrator function** which will call (ii) **Mapper activity functions** first to do mapping and then **(ii) shuffler activity function** that does shuffling, and **(iii) Reducer activity functions** to do reduce, and a **(iv) HTTP client trigger function** that invokes the orchestrator.
    - The mapper should take as input a <key, value> pair, where key is the line number and value is the line string, and produce as output a list of key—value pairs, where key is word and value is 1. The mapper tokenize the line into words and reduce <word, 1> pairs.
    - The reducer should take as input a key-value pair, where key is a word and value is a list (which will be all 1s). The reducer should add up all the values in the list and produce as output a key—value pair where key is word and value is total count. This is what we saw in lecture 4 – look at the slide deck for pseudocode.
    - The shuffler should take map outputs (i.e a list of k-v pairs) and produce the input to reduce fn (<key, [list of values]>).
    - TIP: you can implement the orchestrator so that it yields on each call to mapper or reducer for your first implementation. This will simplify things. But later, you should have it run mappers and reducers in parallel. So you should check the fan-out/fan-in pattern to see how you can do parallel task invocations with deferred wait.
    - Test the above implementation locally. Feed in fake lines to Mapper from MasterOrchestrator and see if everything works.
- Now, we have given you a few input files (mrinput-[1,2,3,4] in moodle) that contain paragraphs from the Saving Mr. Banks song played in class. Create a storage account and upload these files on Azure

blobstore in a container. You can do this via the portal and you can find info about blobstore here: https://learn.microsoft.com/en-us/azure/storage/blobs/storage-quickstart-blobs-portal

- Now, read how to use the python client library to get blobs from the Azure blobstore using their connection string (you can use passwordless if you want, but connection string is easier): https://learn.microsoft.com/en-us/azure/storage/blobs/storage-quickstart-blobs-python?tabs=connection-string%2Croles-azure-portal%2Csign-in-azure-cli
- Implement a new activity function **GetInputDataFn** that uses the blob store API and connection string to pull down the files from Azure. It should then read each file, break it into lines, and build the overall input to mapreduce which you faked earlier – a list of [<offset, line string>, …] key-value pairs. You can see code for reading from blob store here : https://learn.microsoft.com/en-us/azure/storage/blobs/storage-quickstart-blobs-python?tabs=managed-identity%2Croles-azure-portal%2Csign-in-azure-cli.

- Modify MasterOrchestrator to call the GetInputDataFn first to get all data. If you have implemented Map and Reduce correctly, everything else should just work like magic! Notice the power of pure functions and clean abstractions.
- Deploy the entire solution to azure and test it to make sure that it works. See how many functions got invoked in parallel and marvel at the beauty of Azure Durable functions!

Deliverable: at the end of this assignment, you have to fill and submit on moodle a file called "Azure-Lab2-Deliverable.docx", which is available in the assignment 2 section on moodle.