Information Technology Course
Module Cloud Computing
by Damir Dobric / Andreas Pech

Implementation of Serialization of Spatial Pooler in Microsoft Azure Cloud

Anik Biswas
anik.biswas@stud.fra-uas.de

Raghavendra Yarlagadda
raghavendra.yarlagadda@stud.fra-uas.de

Mounika Kolisetty
lakshmi.kolisetty@stud.fra-uas.de

Manash Chakraborty
manash.chakraborty@stud.fra-uas.de

***Abstract*—Cloud Computing is the technological capability to use IT infrastructure and services which are not deployed in local PC or server and usually is accessed by an internet connection. In recent years Cloud Computing has seen tremendous growth due to high availability of data, enhanced mobility, flexibility and efficient resource allocation and cost-saving. Cloud Technology is expected to be a major driving factor of future computing. This project is an aim to get familiarities with various components of Microsoft Azure. In order to cater that, the software engineering project Serialization of Spatial Pooler will be deployed in Cloud environment with the help of Microsoft Azure and docker.**

Keywords- Cloud Computing, Server, Microsoft Azure, Spatial Pooler, Serialization, Docker.

INTRODUCTION

Cloud computing can be described as a model that enables global, convenient and on-demand network access to a shared pool of configurable computing resources that can be efficiently accessed and released with minimal user-side management effort and minimal service provider engagement. Cloud computing is deemed to be the evolution of a variety of technologies that have come together to transform the organization's framework for developing its IT infrastructure. In this paper we discuss about the Microsoft Azure environment, storages, and services in order to deploy and run the Software Engineering project on Cloud. The software engineering project was to integrate feasible serialization technique in Spatial pooler module in .NET environment. The same project is used and reworked in this project to be deployed on Microsoft Azure Cloud.

In the first part of our paper we discuss briefly about the project "Serialization of Spatial Pooler" and significance of implementing the same in Cloud environment. In the subsequent sections, the requirement of different Microsoft Azure Storages and the general workflow for this project will be explained. In the final section, the result obtained from the experiment will be shared and discussed.

***Overview of Serialization of Spatial Pooler**:* Hierarchical temporal memory (HTM), inspired by the architecture and functionality of Neocortex, is designed to learn sequences and make predictions. Vast amount of information is fed to our brain through our sensory systems which remodels raw external data from light, sound pressure, skin deformations etc. In order to organize these data and constitute our perceptibility cortical neurons forms synaptic connections to a subset of the presynaptic neurons. This learning mechanism by cortical neurons to specific input patterns and collective input specific representation of a population of neurons inspires the algorithmic property of Hierarchical temporal memory. Two primary algorithms of the current version of HTM are Spatial Pooler (SP) and Temporal Memory (TM). The SP takes a binary input and produces a new Sparse Distributed Representation (SDR) of the input, here an SDR is a binary vector typically having a sparse number of active bits, i.e., a bit with a value of "1". Hence SP works as a mapping function that transforms input domain to a feature domain. The mapped output from Spatial Pooler in the form of SDR Data is then subsequently fed to Temporal Memory [1].

Serialization is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The HTM Spatial Pooler software model incorporates complex data handling constituted by different type of properties. In the previously implemented Software Engineering Project, JSON serialization technique was used to serialize and deserialize the properties of Spatial Pooler object. Special calibrations were done in JSON serializer settings to handle complex type serialization. Member Serialization attributes had to be used in Spatial Pooler class and its constituent property class Connections. The serialization model was successfully tested to store all the properties of a trained Spatial Pooler object model into a file (of JSON,.txt or relevant extensions) and subsequently was capable of recreating the same Spatial Pooler object from that file through the deserialization model.

Significance of Implementing the Software Engineering project in Cloud Technology: From the general context, using a cloud based software comes on with many advantages; some of that being cost-saving, improved collaboration,better scalability, mobility, flexibility, efficient resource allocation etc. In view of this particular Software Engineering project Serialization of Spatial Pooler, it can be particularly highlighted that using Cloud Service to deploy the project would enhance mobility and flexibility. The Serialized properties of an instance of Spatial Pooler, especially with Trained Model, contains a lot of data. If this Software is deployed in Cloud then the benefit of Azure Storage can be leveraged by storing Serialized Properties of any number of instances of Spatial Pooler and subsequently the same storage can be used to recreate Spatial Pooler object with the deserialization function anywhere and anytime. Thus, it will save

significant amount of computing and storage resources of the local environment.

Prerequisite for deployment of any project to Cloud: Comprehension of the following features is needed to deploy the Spatial Pooler Serialization project in the Microsoft Azure Cloud.

a) Containerization of an App using Docker: In conventional software development, codes developed in one computing environment often comes across with bugs and errors when deployed in a different environment. This issue can be solved by deploying the Software through container in the cloud. Containerization helps development teams to work rapidly, deploy applications efficiently and perform on an enhanced rate. Containers provide a logical packaging mechanism in which applications can be segregated from the environment in which they currently work. This decoupling enables quick and consistent deployment of container-based applications, regardless of whether the target environment is a private data center, public cloud, or even the personal laptop of a developer. Containerization offers a clean separation of concerns, with developers focusing on the application logic and dependencies while, IT operations teams can focus on deployment and management without bothering with application details such as specific software versions and configurations specific to the app. All the above requirements can be best served by Docker. Docker enables developers to easily pack, ship, and run any application as a lightweight, portable, self-sufficient container, which can run virtually anywhere. The app first needs to be package up as Docker image, which can be test locally by using Docker for Windows, or on cloud by using Microsoft Azure Container Registry and Instance service [2] [3].

b) Azure Storage: The Azure Storage platform is Microsoft's cloud storage solution for modern data storage scenarios. Core storage services offer a massively scalable object store for data objects, disk storage for Azure virtual machines (VMs), a file system service for the cloud, a messaging store for reliable messaging, and a NoSQL store. The Azure Storage platform includes the following data services [4].

c) Azure Blobs: Azure Blob storage is Microsoft's object storage solution for the cloud. Blob storage is optimized for storing massive amounts of unstructured data, such as text or binary data. Objects in Blob storage can be accessed from anywhere in the world via HTTP or HTTPS. Users or client applications can access blobs via URLs, the Azure Storage REST API, Azure PowerShell, Azure CLI, or an Azure Storage client library. The storage client libraries are available for multiple languages, including .NET, Java, Node.js, Python, PHP, and Ruby [5].

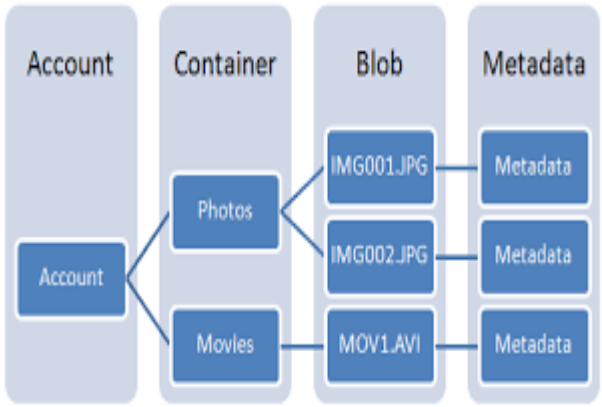

Figure 1. Hierarchy of Blob storage

d) Azure Queue Storage: The Azure Queue service is used to store and retrieve messages. Queue messages can be up to 64 KB in size, and a queue can contain millions of messages. Queues are generally used to store lists of messages to be processed asynchronously [4].

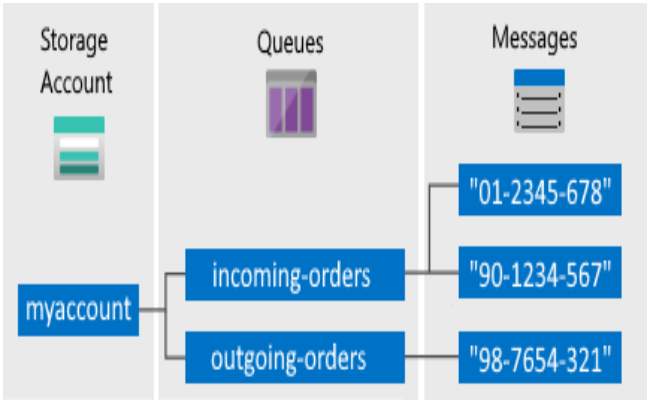

Figure 2 Hierarchy of Queue storage

e) Azure Table Storage:

Azure Table storage is a service that stores structured NoSQL data in the cloud, providing a key/attribute store with a schema less design. Because Table storage is schema less, it's easy to adapt your data as the needs of your application evolve. Access to Table storage data is fast and cost-effective for many types of applications and is typically lower in cost than traditional SQL for similar volumes of data. Azure Table storage stores large amounts of structured data. The service is a NoSQL datastore which accepts authenticated calls from inside and outside the Azure cloud [6].

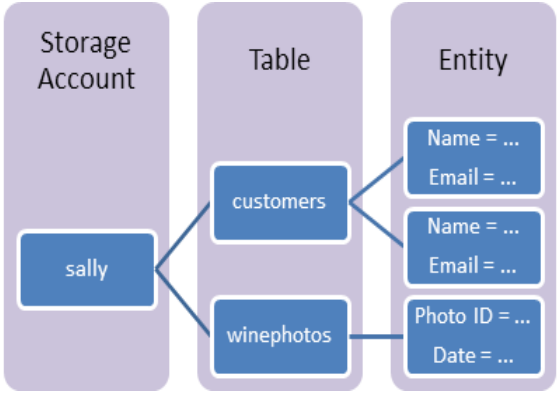

Figure 3 Azure Table Storage Structure

II GENERAL WORKFLOW

The primary objective of this project is to run the already developed code for the Serialization of Spatial Pooler on Cloud using Microsoft Azure cloud for which the general workflow are as follows:

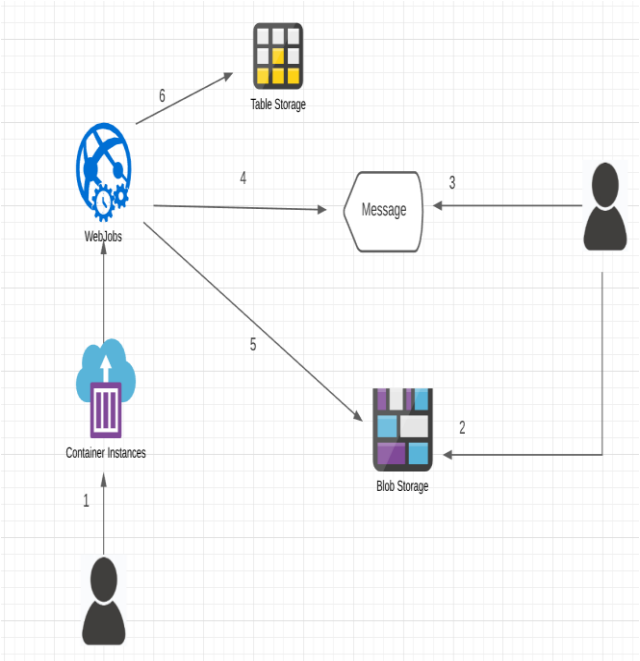

Figure 4 Project's Workflow

In the following sections the description of the implemented code is explained which is followed by the results section after the deployment of Serialization of Spatial Pooler application on Azure Cloud

Develop the code as an Azure Function and publish it as a Docker Container: At First, the code of the Serialization of the Spatial Pooler is built up as a docker image and then this image is pushed to the Azure Container Registry and the application then can be deployed using the Azure Container Instance service.

Upload the Input File to blob storage: The input files are uploaded to Azure blob storage in a JSON format to start the application.

Send a Trigger message in Queue: A particular message is sent in the Queue to trigger the training process.

Perform Deserialization: When the message from Queue is received, deserialization is performed, and the code gets triggered. The process of deserialization is needed here .

Download input file from blob storage: Once the code gets triggered it will start downloading the input file from the Azure blob storage.

Run the testcases of Serialization of Spatial Pooler: Since the required input file has already been downloaded the testcases should successfully executed taking that input as an argument.

Download first output file from blob storage: The output file will be downloaded in order to subsequently check if it works with the deserializer function of spatial pooler and recreate the same spatial pooler object.

Use the downloaded file in Spatial Pooler Deserialization function: The downloaded file will be fed to deserializer function.

Same instance of Spatial Pooler will be recreated.

Upload Result/Output file to table and blob storage: The output of the testcases will be uploaded to both table and blob storage.

Delete message from Queue: At last the queue message which we sent earlier to trigger the training process will be deleted.

III CODE DESCRIPTION

The entire projects are contained in the MyCloudProject solution. The focal point of this project according to the workflow described above is Experiment.cs class. It is located in the MyExperiment directory in the project folder and this is called by the main Program.cs class for execution.

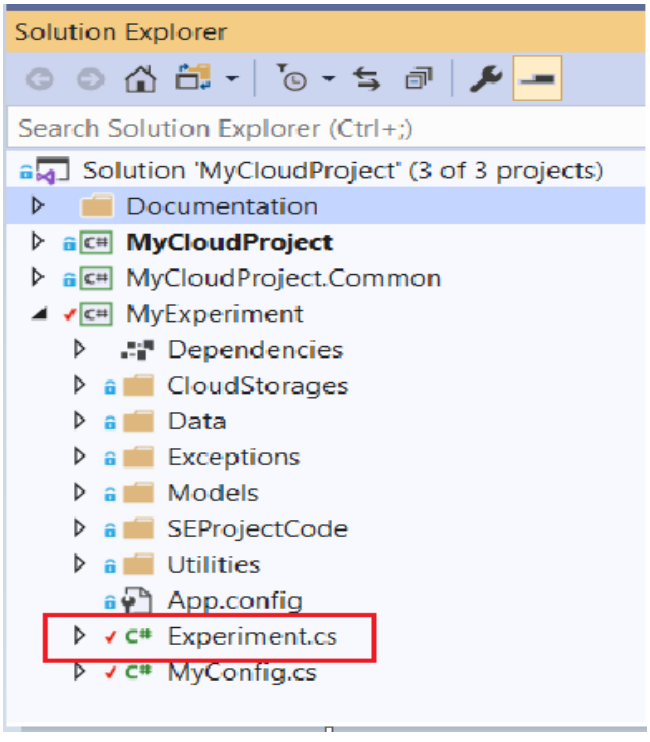

Figure 5 Orientation of Experiment.cs Class

The aim of the experiment class is to locally create the folder path where files will be downloaded from blob storage and executed and uploaded back to azure (Blob & Table). Subsequently the uploaded Blob will again be downloaded, extracted, and fed to test the accuracy of the deserialization method of Spatial Pooler and thus would demonstrate the advantages of the Cloud Computing functionalities specific to this project. The program will then be executed via a cancelation token until it is signalled to cancel. The program's outcomes are then uploaded to the storage blob

and table. In several approaches discussed below, this chain of activities is illustrated.

The input data which is uploaded to the blob storage and the queue message which are triggered to the queue storage are given below.

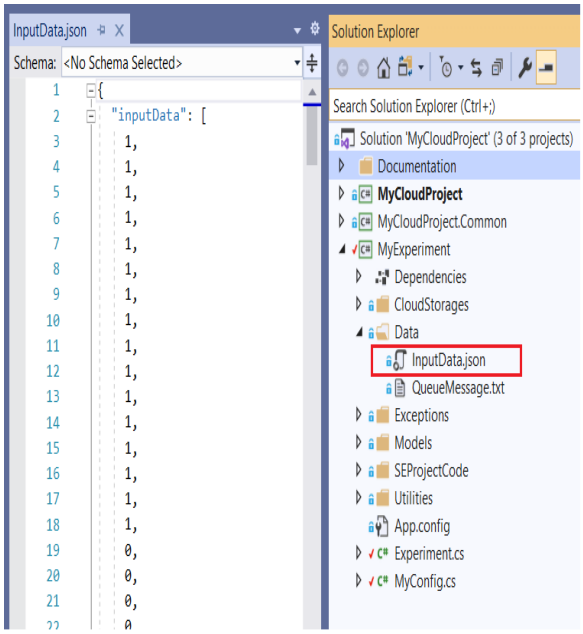

Figure 6 Input data


```
{
  "ExperimentId" : "ML 19/20 - 5.8",
  "InputFile" : "https://aniklogu.blob.core.windows.net/anikblobcontainer/InputData.json",
  "Name" : "Testing input1.json",
  "Description" : "project review"
}
```


Figure 7 Queue Message

The main three objects which are declared in the Experiment.cs are explained below and to be used throughout the project.

```
public class Experiment : IExperiment
  {
     public static string DataFolder { get; private set; }
     private IStorageProvider storageProvider;
     private ILogger logger;
     private MyConfig config;
```

The three objects are storageProvider, logger, and config. *storageProvider* deals connections with the blob storage i.e. uploading and downloading files in and from the blob whereas *logger* concerns appending structured messages regarding instant at hand throughout the code called logs and the *config* depicts the configuration selected for the project, associates with container names, queue, and group ID etc.

```
public Experiment(IConfigurationSection configSection,
IStorageProvider storageProvider, ILogger log)
     {
        this.storageProvider = storageProvider;
        this.logger = log;
         config = new MyConfig();
        configSection.Bind(config);

DataFolder=Path.Combine(Environment.GetFolderPath
(Environment.SpecialFolder.LocalApplicationData),
            config.LocalPath);
        Directory.CreateDirectory(DataFolder);
          }
```

A constructor is defined after the object declarations. It is noteworthy that *configSection* is binded with the config. This is useful in order to read configuration properties from the appsettings.json file of the project.

Moving ahead a *RunQueueListener* method is created after the constructor. This reads the message from the queue for the program to be executed.

The arguments of this method basically receive a token and this token must be cancelled in the further time of execution with type *CancellationToken*. At first a queue is generated with the information provided in the appsettings.json file. This trigger queue is generated in the storage account of the user. After this the status of the token is continuously being checked. At a cancel state (subject to the user pressing the key), the process is stopped, and an appropriate log is registered and at a non-cancel state the *RunQueueListener* method takes 6 steps which are described below.

```
public async Task RunQueueListener(CancellationToken
cancelToken)
 {
CloudQueue queue = await AzureQueueOperations.
CreateQueueAsync(config, logger);
 while (cancelToken.IsCancellationRequested == false)
 {
var message = await queue.GetMessageAsync
(cancelToken);……
```

Step I: The queue message which is triggered is read and if the message is not null it is deserialized or converted from byte stream to object in memory. A log is then logged on the console which displays the details in the queue message such as Experiment ID, InputFile, etc.

Reading message from Queue and deserializing

var experimentRequestMessage = JsonConvert. DeserializeObject<ExerimentRequestMessage>(message. AsString);

logger?.LogInformation($"Received message from the queue with experimentID: " +$"{experimentRequestMessage.ExperimentId}, " +

$"description: {experimentRequestMessage.Description}, " +

$"name: {experimentRequestMessage.Name}");

Step II: The training data file present in the queue message is downloaded from the blob and is stored in the local path of the file and this local path is stored in a variable *localStorageFilePath*. The *DownloadInputFile* method which is implemented in *AzureBlobOperations.cs* class and is used to download the input file. The definition of this method is shown in Fig 3.8. In the arguments, the filename to be downloaded is provided. The local repository path is generated first, and the name of the file is added. Next, using the storage account connection string an instance blob is initialized by using the credentials of the account and the URI of the file name in the storage blob. The contents in the blob are then downloaded in the file using the instance with *DownloadToFileAsync.*

Downloading the input file from the blob storage

var fileToDownload = experimentRequestMessage. InputFile;

var localStorageFilePath = await storageProvider. DownloadInputFile(fileToDownload);

logger?.LogInformation(

$"File download successful. Downloaded filelink: {localStorageFilePath}");

Step III: As the training data file is in the local repository, the path of the file stored is passed as an argument to the *Run* method which is shown in Fig 3.9. It first deserializes the data from the file and it is passed to the method called *RunSoftwareEngineeringExperiment.* The result of the program returned by `Run` is serialized or converted into byte stream for further processing.

var result = await Run(localStorageFilePath);

result.Description = experimentRequestMessage. Description;

var resultAsByte = Encoding.UTF8.GetBytes (JsonConvert.SerializeObject(result));

Step IV: The zip file is uploaded to the blob storage with the help of a method *UploadResultFile* in class ***AzureBlobOperations****.cs.* A decent log is added, and the blob URI is stored in result. A blob service client object is created to create an object of container client. The file is then uploaded to the blob storage and the URI of the file in blob is returned.

Uploading result file to blob storage

var uploadedUri =

await storageProvider.UploadResultFile("ResultFile-" + Guid. NewGuid() + ".txt", resultAsByte);

The result is uploaded to the table storage using UploadExperimentResult method which is in AzureBlobOperation.cs class as shown below With the help of CreateTableAsync method in AzureTableOperations.cs class a table is created using the storage connection string and the results are inserted or merged as entities in the table. This is done with the help of InsertOrMergeEntityAsync method which is in the class AzureTableOperations.cs.

Uploading result file to table storage

await storageProvider. UploadExperimentResult(result);

Subsequently the message of the queue is deleted. The cancellation token and the message are passed in the method DeleteMessageAsync. This is to observe the cancellation token which the delete is taking place. If the cancel token signals to stop the process the delete will not occur

Deleting the message from the queue

await queue.DeleteMessageAsync(message, cancelToken);

Integrating Software Engineering experiment : The software engineering experiment is referenced and run under the method *RunSoftwareEngineeringExperiment().*

using MyExperiment.SEProjectCode;

private async Task<string> RunSoftwareEngineering Experiment (

SeProjectInputDataModel seProjectInputDataList)

{

// Implementation and reference of our Unittests

var inputVector = seProjectInputDataList.inputData;

SpatialPoolerSerializeTestCloud serializetest = new SpatialPoolerSerializeTestCloud();

output=serializetest.SerializationTestWithTrainedData (inputVector);

/*Writing Output data to the specific file*/

```
FileUtilities.WriteDataInFile("SpatialPoolerSerialization
Output.txt", inputVector,
            output);
```

As per the code above, a reference of the existing Software Engineering project is made. The Unit Test for the Software Engineering project is rewritten under the SEProject Folder.

Within the task **RunSoftwareEngineeringExperiment** (), an object of our SE Project *SpatialPoolerSerializeTestCloud* is created. Input parameters are passed through inputData from class SeProjectInputDataModel; output is generated through *SerializationTestWithTrainedData*() method from the SE Project code and the output is subsequently written to the file SpatialPoolerSerializationOutput.txt.

The output file is then Zipped to a file SPOut1.zip and uploaded to Blob through UploadResultFile () method called from AzureBlobOperations class. The method ***UploadResultFile*** () uses Bob service client object to create a container client object. The filename of the result file is added as per the local path. The URI of the file in Blob is returned. The Result is then uploaded to Table Storage using ***UploadExperimentResult***() method implemented in the same AzureBlobOperations class.

Next a table is created with the ***CreateTableAsync***() method implemented in AzureTableOperations class. The results are inserted or merged based on the presence of entity in the table with the help of ***InsertOrMergeEntityAsync***() method from AzureTableOperations class.

In order to test the Deserialization functionality of Spatial Pooler, some subsequent operations are performed.

1) The result file SPOut1.zip uploaded to the Blob storage is again downloaded through the method downloaded_from_blob() which is implemented in AzureBlobOperations class. This method downloaded_from_blob() is separately created to independently download the particular blob file without requiring of any Trigger queue.

```
public async Task download_from_blob()
    {
      string connectionString =
Environment.GetEnvironmentVariable("AZURE_STORAG
E_CONNECTION_STRING");

BlobServiceClient blobServiceClient = new
BlobServiceClient(connectionString);

string containerName = "result-files";

BlobContainerClient containerClient =
blobServiceClient.GetBlobContainerClient(containerNam
e);

 string localPath = @".\downloadlocation\";

string fileName = "SPOutput1.zip";

string localFilePath = Path.Combine(localPath,fileName);

BlobClient blobClient =
containerClient.GetBlobClient("SPOutput1.zip");

BlobDownloadInfo download = await
blobClient.DownloadAsync();

using (FileStream downloadFileStream =
File.OpenWrite(localFilePath))
      {
await download.Content.CopyToAsync
(downloadFileStream);

        downloadFileStream.Close();
      }
    }
```

In order to accommodate the above Blob Downloading functionality it is important to save the AZURE connection settings in the local system as an environment variable priorly. Following are the ways to update environment variable in different OS:

Windows :
setx AZURE_STORAGE_CONNECTION_STRING "<yourconnectionstring>"

Linux**:**
export AZURE_STORAGE_CONNECTION_STRING ="<yourconnectionstring>"

macOS:
export AZURE_STORAGE_CONNECTION_STRING="<your connectionstring>"

After the ZIP result file is downloaded from the Blob it is extracted to an appropriate local path with the help of a custom built *ExtractMethod*() function. The serialized result file SpatialPoolerSerializationOutput.txt is obtained after extraction and used in Spatial Pooler deserializer function. Hence the same object of Spatial Pooler as obtained earlier is recreated. The accuracy of the deserialization process is verified by the following process: The deserialized instance of Spatial Pooler is then again serialized and the result is stored in a string and subsequently compared with previous output of Serialization with string comparison function.

```
SpatialPoolerSerializeTestCloud spDeserializer = new
SpatialPoolerSerializeTestCloud();

 var sp=spDeserializer. SpatialPoolerCloudDeserializerTest
("SpatialPoolerSerializationOutput.txt");

 sp.Serializer("LocalSerializedFile.txt");

 string des1 = File.ReadAllText("LocalSerializedFile.txt");
 int comp = string.Compare(output, des1);

if(comp==0)
      {
        Console.WriteLine("Spatial Pooler Object
deserialized");
      }
```

```
else
        {
            Console.WriteLine("Not all the properties are serialized/deserialized properly");
        }
```

III RESULT

The entire process of Serialization and Deserialization in Cloud completes very quickly. Once the process begins a Queue is created as per Fig 8.

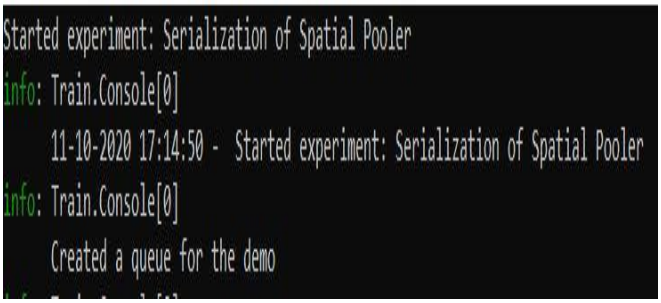

Figure 8 Creation of Queue

Once the Queue is updated in the Azure Portal with specific message and Input URL of the blob storage, the Input File is downloaded, Software Engineering Experiment is executed, and result is uploaded to Blob and Table storage.

Figure 9 Download from Blob storage and Result Upload

Also, the Console Message confirms the accuracy of the serialization and deserialization process. The table storage confirms the Experiment result updated with appropriate Timestamps.

PartitionKey	RowKey	Timestamp	Description	DurationSec	EndTimeUtc
GroupSW	80e21066-6e65-45c2-b4c9-5be0c6374f09	Thu, 08 Oct 2020 14:31:53 GMT	project review	00000000000000048	Thu, 08 Oct 2020 14:31:43 GMT
GroupSW	ddb3c127-ef2a-42c6-b165-22ed583a8c12	Thu, 08 Oct 2020 14:40:26 GMT	project review	00000000000000019	Thu, 08 Oct 2020 14:40:24 GMT
GroupSW	6e0daebe-24c6-4302-a29b-f8e8fb46f68f	Thu, 08 Oct 2020 14:46:24 GMT	project review	00000000000000029	Thu, 08 Oct 2020 14:46:01 GMT
GroupSW	c4c1d7d6-f6a4-478e-9b49-b786713d412d	Thu, 08 Oct 2020 17:18:16 GMT	project review	00000000000000056	Thu, 08 Oct 2020 17:18:11 GMT
GroupSW	dd40ed48-f903-4366-8d24-5e4ee4a2e575	Thu, 08 Oct 2020 17:31:53 GMT	project review	00000000000000048	Thu, 08 Oct 2020 17:31:49 GMT
GroupSW	466a4e3f-3027-418f-bd49-86b26186d559	Sun, 11 Oct 2020 15:00:28 GMT	project review	00000000000000048	Sun, 11 Oct 2020 15:00:18 GMT
GroupSW	d21da694-fab4-4122-b176-1e9a60ff72d3	Sun, 11 Oct 2020 15:02:18 GMT	project review	00000000000000021	Sun, 11 Oct 2020 15:02:16 GMT
GroupSW	e689c08c-a214-4226-9ca3-741877889650	Sun, 11 Oct 2020 15:08:45 GMT	project review	00000000000000048	Sun, 11 Oct 2020 15:08:43 GMT
GroupSW	3c987658-67b4-46f4-9bbc-3974e1ce928e	Sun, 11 Oct 2020 15:16:05 GMT	project review	00000000000000005	Sun, 11 Oct 2020 15:16:04 GMT

Figure 10 Updated Table Storage with Experiment Result

Hence, from the above experiment it can be inferred that Cloud Computing provides an excellent platform to access and deploy Software anywhere and anytime; thus, enhancing mobility,flexibility and efficient resource allocation.

References

[1] Y. Cui, A. Subutai and J. Hawkins, "The HTM Spatial Pooler—A Neocortical Algorithm for Online Sparse Distributed Coding,," *Front. Comput. Neurosci*, 2017.

[2] "What is docker and why it is so darn popular," [Online]. Available: https://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/.

[3] "CONTAINERS AT GOOGLE," [Online]. Available: https://cloud.google.com/containers.

[4] "Introduction to Azure Queue," [Online]. Available: https://docs.microsoft.com/en-us/azure/storage/queues/storage-queues-introduction.

[5] "Introduction to the core Azure Storage services," [Online]. Available: https://docs.microsoft.com/en-us/azure/storage/common/storage-introduction.

[6] "Azure Table Storage Overview," [Online]. Available: https://docs.microsoft.com/en-us/azure/cosmos-db/table-storage-overview#:~:text=Azure%20Table%20storage%20is%20a,needs%20of%20your%20application%20evolve..

.