Proposal : Create a cloud based python application that can return the sorted input numbers. This application concept would be further enhanced into a full fledged SaaS application.

Keywords: Elastic Load Balancer, Auto Scaling Group, Python, Flask, Amazon EC2, ip address, DNS Name, json, VPC, HTTP.

1. Brief:

The idea of this project is to create a python application that communicates with the consumer via api calls. Once the api is hit, it must fetch the sorted order of the input numbers to the consumer. The complexity of the application with the architecture of the cloud the server is hosted in.

2. Requirements:

Modules: Python3, pip3;
Python Libraries: json, literal_eval, flask, jsonify and request;
Servers: httpd, wsgi.

3. Inputs :

Since the apis communicate through json, the inputs i.e., numbers are received from the client through json via the server ip address that is provided to the client.
Input format could be as follows:

a. { “x_1”: “num_1”,“x_2”: “num_2”,.......“x_n”: “num_n” }
b. { “x_1”: “[num_1,num_2,.......num_n]”}
c. { “x_1”: “[num_1,num_2,.......num_n]”,“x_2”: “num_2”,.......“x_n”: “num_n” }

4. Output:

After the parsing the consumed json, the object is sorted, converted into a json object so that the client could fetch the response.

5. Proposed Architecture:

The server resides in the environment with enable Auto Scaling Groups with Elastic Load Balancer. The server is only accessible via the ip address of the server or a DNS Name(it would be optimal to provide the client with DNS Name rather than ip addresses; since in the AWS ip addresses change for the instance once it is stopped and started again). The consumer would communicate to the server via HTTP method requests. Also, if ever, the infrastructure must be shifted to a non-default VPC, DNS hostname and resolution of the instance, that is hosting server, must be configured.

6. Implementation:

Cloud infrastructure and Development could be the segments of the implementation phase.

a. Cloud Infrastructure:
The architecture of the project from top-down approach would be the -Elastic Load Balancer in the public

subnet. That means, ELB accepts HTTP requests from any addresses from the web. With the target group as ip of the instances in the following layers.
-Auto scaling group of EC2 instances, over which the server resides, placed in multiple AZs. These are placed in private subnet of the VPC; only allowed to communicate with ELB, using security groups.
-Ec2 instances are configured with apache servers. Detailed implementation of infrastructure will be explained further in this document.

b. Development:
In this phase, two python files are created; one with the main method and other contains the api that is to be hit by the client's request
i. api.py - an api object is instantiated and a base index route, with HTTP methods 'GET' and 'POST' is initialized. Under which an index() method could be invoked. This index() method, once invoked by the client's request will be converted the json format of the input into the python dictionary and calls the sorted.py's method to further work with the input .

ii. sorted.py - this file receives the formatted input from the api.py, checks if the numbers are given sequentially or in a list together; and appropriately sorts the input.

7. Testing:
Postman application could be used to test the flask api that is running in the server (as a server).
It takes the input as either the DNS name of the ELB or the ip of the ELB. Once the ELB is hit with the json object, it redirects the path to the private ip of one of instances in the Auto Scaling Group and returns the output.

8. Further Improvements:
i. There exists a possibility of making this a serverless application using lambda function which significantly cut down the costs of the server usage in idle times.
Ii. Implementing exception catch in the main python file.

Detailed Implementation in AWS:

EC2 - Launch and config:

- AMI for Linux2 is used to launch the EC2 instance and choose the private subnet.
- In advanced config, the following bash script is given to automate the process of install the required environment for the flask api to run as a server.
  ```
  #!/bin/bash
  sudo yum update -y
  sudo yum install httpd -y
  sudo service httpd start
  sudo chkconfig httpd on
  sudo yum install mod_wsgi.x86_64 -y
  sudo yum install python3 -y
  sudo yum install python-pip -y
  sudo pip install flask -y
  sudo mkdir /var/www/html/flaskapp
  ```
- In the directory /var/www/html/flaskapp create a file name app.wsgi that is invoked by mod_wsgi extension.
  ```
  <<app.wsgi>>
  #!/usr/bin/python
  import sys
  sys.path.insert(0,"/var/www/html/flaskapp/")
  from app import app as application
  ```

- Since *httpd* only serves html, the *wsgi* extension is used to manipulate *httpd* to serve the .py files through wsgi configuration
- While configuring the security group of the EC2 instance, add an inbound rule to only receive traffic from the security group of ELB through port 80.
- After launching and connecting to the EC2, copy/clone the python files into the flaskapp directory that is created in /var/www/html/.
- Now, to enable *mod_wsgi,* the following block must be added just after the DocumentRoot /var/www/html in the httpd configuration file location in /etc/httpd/conf/httpd.conf

```
WSGIDaemonProcess app processes=5 python-path=/lib/python2.7/site-packages threads=1
WSGIScriptAlias / /var/www/html/flaskapp/app.wsgi
<Directory flaskapp>
   WSGIProcessGroup app
   WSGIApplicationGroup %{GLOBAL}
```

```
    Order deny,allow
    Allow from all
</Directory>
```

- Restart the httpd service for changes to take effect(*service httpd restart*)

ELB:

- Deploy an Application Load Balancer in the public subnet and add an inbound rule to accept request from any ip address (0.0.0.0/0,0::0/0).
- Select the *Target Group* of the load balancer based on the ip and either provide the private ip of the instance launched in private subnet or its DNS name.
- The above step allows the request that hit the load balancer to be redirected to the server in the private subnet.

Implementation in Microsoft Azure

The infrastructure of the deployment stays the same as in AWS, i.e/. Load Balancer on the public subnet accessed by the internet, on top of the Scale Set of the Virtual Machines that act as the Flask API server which is placed in private subnet for security purposes.

Step-wise Implementation:

- In the networking module of Azure's portal, select and create Load Balancer and put it in a newly created Resource Group "ServerLB" (Azure insists on usage of Resource Groups, because all the external configurations like routing, Network security group can be assigned directly to the resource group, instead of individual entities.). While creating the Load Balancer we get to choose if it should have a static ip or dynamic ip allocation. Static IP would be beneficial for deployment purposes.
- To create the VM, browse to Compute module, select the image of the OS and deploy it. Assign it to a Resource Group of its own, "ServerVM", also assign inbound rules on port 80
- In the Load Balancer that is created, it is to be specified which Backend Pool the Load Balancer points to. Add the "ServerVM" Resource Group to the Backen Pool of the LB.
- Create a Network Security Group for "ServerVM" and add inbound rules mentioning the source and destination to be from Load Balancer, on port 80(HTTP)
- Create a Network Security Group for "ServerLB" and add inbound rules mentioning the source and destination to be from Internet, on port 80(HTTP)

- In the Load Balancer mention that where the requests should be forwarded to in the inbound NAT rules, add a rule specifying the the communication between the Scale Set of VMs and the Load Balancer.
- Access the Linux VM that is created and configure the following, to deploy it as a Flask Server

```
apt-get update -y                                  //updates the existing packages in the system
apt-get install -y apache2                         //installing the apache2 web server
service apache2 start                              //starting the apache2 web server
apt-get install -y libapache2-mod-proxy-uwsgi      //installing the mod_wsgi extension
apt-get install python3 -y
apt-get install python-pip -y
Pip install flask -y
Apt-get install python3-pip -y
Pip3 install flask -y                              //installing Python, Python3, pip,pip3 and flask
mkdir /var/www/html/flaskapp                       //make directory flaskapp in /var/www/html
nano /var/www/html/flaskapp/app.wsgi               //create a file app.wsgi in flaskapp dir
```

add the following contents to the app.wsgi:

```
#!/usr/bin/python
import sys
sys.path.insert(0,"/var/www/html/flaskapp/")
from app import app as application
```

- Now, to enable *mod_wsgi,* the following block must be added just after the DocumentRoot /var/www/html in the apache2 configuration file location in /etc/apache2/sites-available/000-default.conf

```
WSGIDaemonProcess app processes=5 python-path=/usr/local/lib/python2.7/site-packages
threads=1
WSGIScriptAlias / /var/www/html/flaskapp/app.wsgi
<Directory flaskapp>
   WSGIProcessGroup app
   WSGIApplicationGroup %{GLOBAL}
   Order deny,allow
   Allow from all
</Directory>
```