1. Introduction
2. Guides and Design
3. System architecture

This tweets analysis system makes great use of the Melbourne Research Cloud to harvest tweets through Twitter API and shows stories in three different scenarios from the data. Because of the distributed architecture, the system has both flexibility, fault tolerance and efficiency features. It is distributedly deployed in three VMs (172.26.132.17, 172.26.133.152, 172.26.130.38) in cloud. Moreover, these three nodes can communicate with each other through appropriate security groups settings.

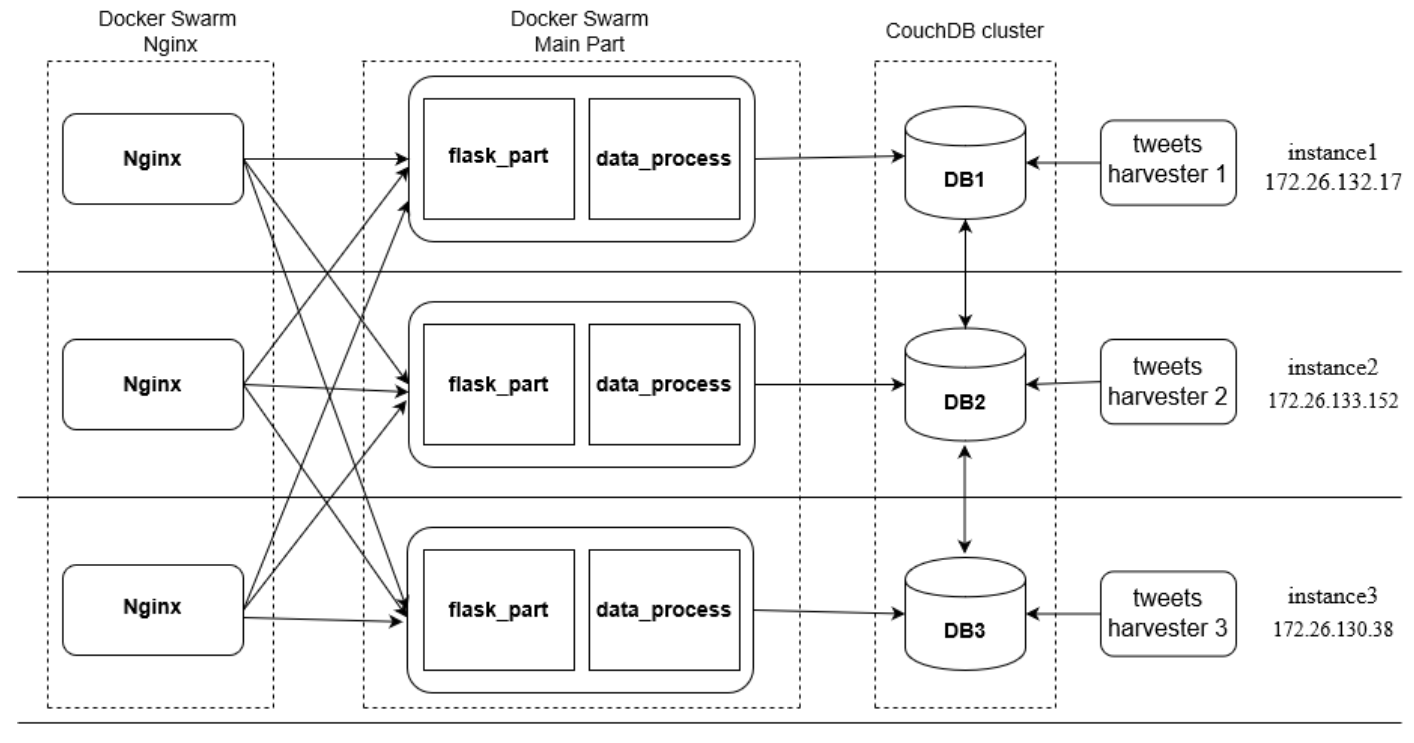


Fig 3.1 System Architectures

* 1. CouchDB Cluster

This system uses a CouchDB cluster to store both tweets data and Aurin data. As CouchDB has availability and partition tolerance features, our system can also cope with such kind of difficulties. The CouchDB services are deployed by docker which can help conquer the complexity of app deployment in different environments. There are complete data in each CouchDB container and these three containers can communicate with each other through port 4369. If the data are updated in one node, the other nodes will also update their data sequentially. Since the data have three copies in three nodes, as long as one node can work well, the breakdown of the databases in other nodes will not cause the breakdown of the whole system.

As the fig 3.1 shows, there are three different tweets harvesters in three instances respectively. They are also deployed by docker and each of them is correspond to one scenario and gain tweets through Twitter Stream API. The data in databases are updated in real-time. The more time the harvesters have worked, the more useful data will be stored in databases.

* 1. APP Main Body

Our system uses flask as the main technology to build the backend part. As a micro web framework written in Python, flask is easy to use without requirements for other tools and libraries.

In detail, the main body consists of two parts including flask\_part and data\_process part (Fig 3.1). Flask part comprises routes for different ways to get data from databases as well as routes for html pages. The flask\_part should have different responses when gaining different requests. If the requests are asking for data, the flask\_part should invoke corresponding data process methods to process and gain data from databases. The processed data should be responded as json format to front end which can be easily analyzed.

The data\_process part provides all the necessary methods to count and make calculations based on the data from databases. The information with great formats will be responded to flask\_part waiting for further integrations.

The main body services are deployed by Docker Swarm, which ensures the flexibility and tolerance of the system. The service named AppMainBody has three replicas. When one of them breaks down, the others will still work and the Swarm self-healing mechanism will fix the down container as soon as possible. In addition, the Docker Swarm also have load balance mechanism which cooperates with the Nginx to provide more efficient allocations.

* 1. Load Balance

For a complex distributed service, load balancers are necessary which make great use of the replicas of the service and markedly improve the efficiency of the system especially when a large number of users visit the app simultaneously. The load balancer will allocate the requests to different modules of the system. Some of them are in one node and some of them are distributed in different nodes.

For this system, we use a two-layer load balance mechanism which ensures the efficiency when facing high data throughput situations. The fist layer is Nginx (Fig 3.1), which are deployed by Swarm. Since the Nginx service has three replicas, users can visit the app through any one IP among the three VMs and the exposed port is 80. Round robin is the algorithm to determine which main body of the app should be visited. When the Nginx receives new requests, it will allocate the requests to these three nodes in order. Besides, when one node is down and cannot make valid response, the Nginx will choose another working node.

The second layer is Docker Swarm load balance mechanism. The same as Nginx, it also uses round robin algorithm to allocate requests to different replicas. The cooperation of these two load balance layers guarantees the availability and efficiency of our system.

1. Deployment
   1. **Environment Building**

For a distributed system, the deployment of each instance is difficult and time wasting. Some operations including the creation of instances, volumes, security groups and basic environment building should be repeated many times. If the environment of each instance has similar parts, Ansible is an excellent technology helping us build the basic environments efficiently and accurately. The basic environment building can be achieved by one ansible playbook. There are total eight roles each of them has different responsibilities (Fig 4.1).

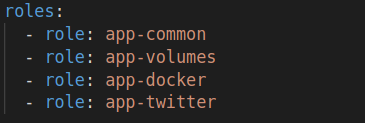
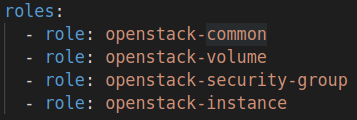


Fig 4.1 Roles in Ansible Playbook

*Openstack-common* role will do some basic preparatory works in localhost including updating necessary packages such as ‘pip’ and ‘openstack’.

*Openstack-volume* role will create 6 volume with 30 GB storage for each and each instance will be attached two volumes later in openstack-instance phase (Fig 4.2).

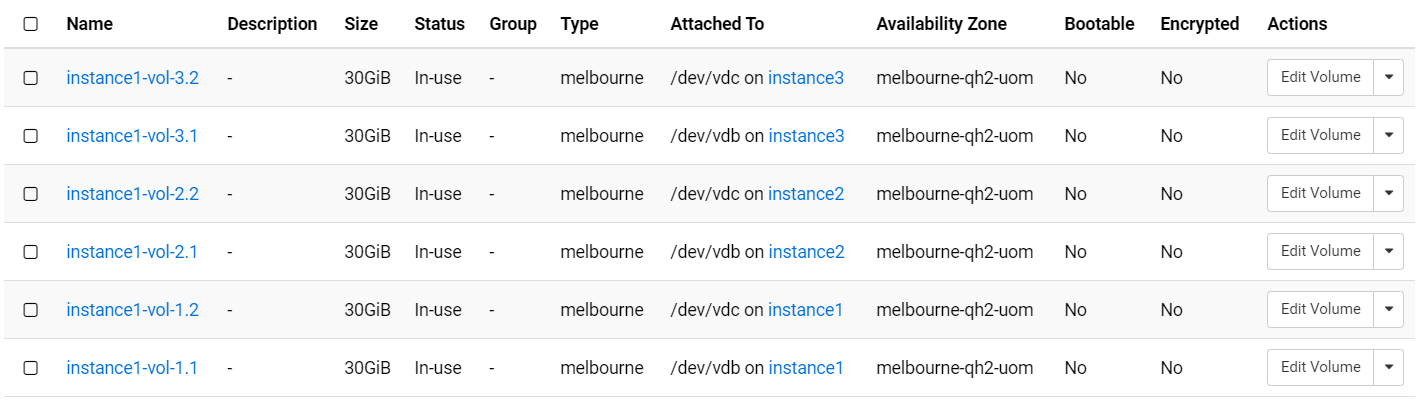


Fig 4.2 Volumes List

*Openstack-security-group* will

…….

Fig 4.3 Security Groups List

*Openstack-instance* role will create three instances in Melbourne-qh2-uom zone (Fig 4.4). All of them are based on Ubuntu1.8 system and the created volumes will be attached to these instances in order and the security groups will be added. After them are created, the IP addresses will be stored in localhost and be used to do further remote controls.

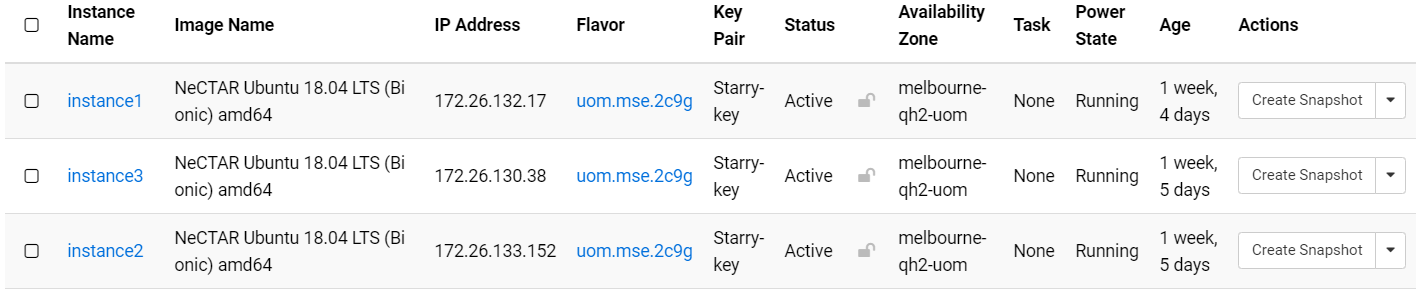


Fig 4.4 Instances List

*App-common* role will create some basic folder structures and install some basic tools in these three instances such as ‘vim’, ‘curl’ and python related packages. Furthermore, the instance proxy and docker proxy will be added to theses nodes in order to communicate with public addresses.

*App-volume* role will mount the attached two volumes to appropriate folders. One volume will be mounted to the folder of docker, the other one will be mounted to the folder of database. The total 60 GB storage will be enough for data storage and docker modules.

*App-docker* role will install the latest Docker and docker-compose in each VMs.

*App-twitter* role is the last role in playbook. It just creates folders for apps which can help with further app deployments.

**4.2 System Deployment**

All the system modules are deployed by docker. Docker is an open platform for developing, shipping, and running applications. It enables us to separate our applications from our infrastructure so that we can deliver software quickly. By taking advantage of Docker’s methodologies for shipping, testing, and deploying code quickly, we can significantly reduce the delay between writing codes and running it in production. The Nginx, AppMainBody and tweets\_harvesters are built as images by Dockerfiles and uploaded to DockerHub. When they are deployed in VMs, they are pulled from DockerHub.

4.21 Database

The containers of CouchDB in the CouchDB cluster are deployed in each instance respectively. Ports mapping includes 4369:4369, 5984:5984 and 9100-9200:9100-9200 ensure that they can communicate with each other well and cooperate with harvesters and app main bodies well (Fig 4.5). For security reasons, these ports are only opened between these three instances which means that the users cannot visit the databases directly from other IP addresses.

Because we have a large number of tweets harvested every day by the harvesters, the storage of the CouchDB should be considered. One of the mount point of 30 GB volume is the folder of CouchDB, which ensures that we can store a large number of tweets.

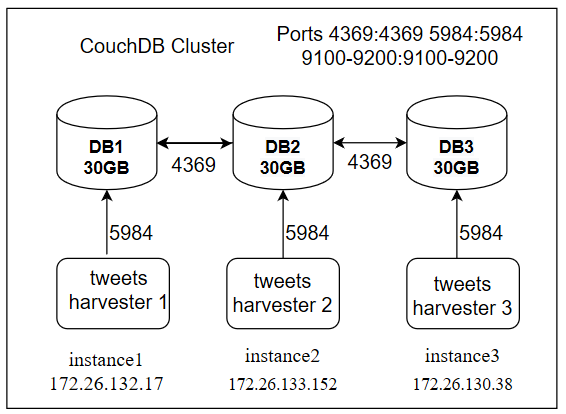


Fig 4.5 Nginx Architecture

4.22 APP Main Body

The main body of this system is also deployed by docker swarm (Fig 4.6). In order to communicate with database in their own VMs, the network of AppMainBody service is set as ‘host’. This means that the main bodies share the same ports with their own VMs and can communicate with database through 127.0.0.1:5984. Although the host network model may not be an excellent solution in complex systems because it limits the available numbers of ports. Instead, overlay network of Swarm may be a good choice. However, in this system, it can work well without considering the limitation of ports and make it easy for this service to communicate with database service. Besides, the Nginx can also communicate with this part through the IP of the instances since the containers of main body share the port 5000 with VMs.

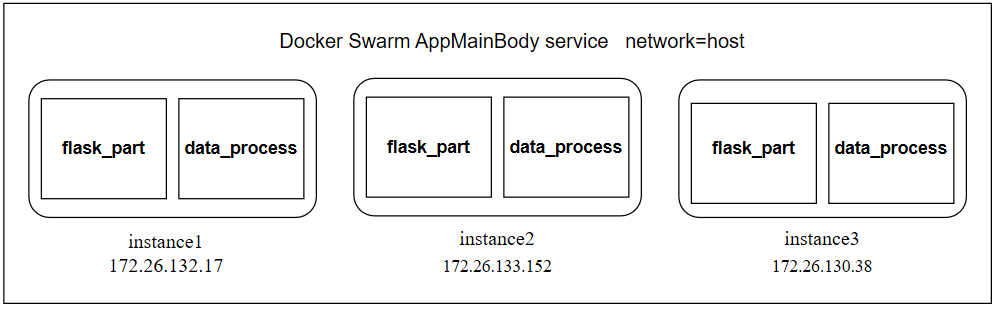


Fig 4.6 App main body Architecture

4.23 Nginx

Similar to the deployment of app main body, Nginx containers are deployed by Swarm. The difference is that they use port mapping instead of host network model. As the AppMainBody service uses host network model, Nginx can easily transfer requests to each of main body container through their remote IP addresses. Some parts of the Nginx configuration are shown in fig 4.8. The exposed port of Nginx is 80, and it will deliver the received requests to the three instances in order.

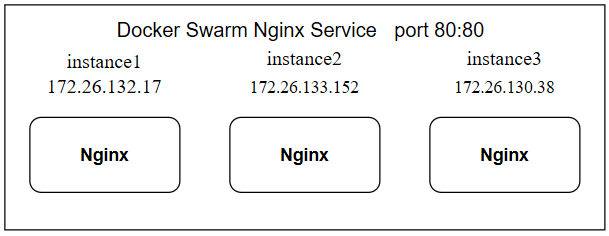


Fig 4.7 Nginx Architecture

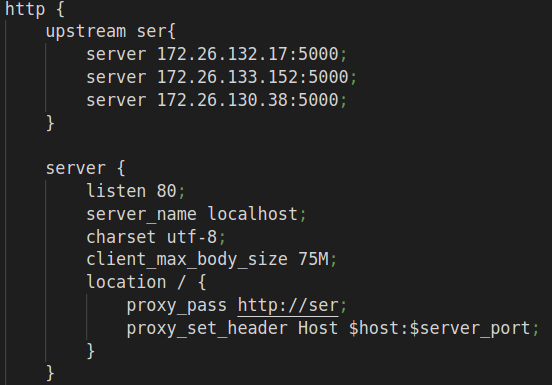


Fig 4.8 Nginx Configuration

1. Scenarios
   1. Scenario1
2. Challenges

In order to gain useful tweets data, choosing the proper Twitter API is important. At the beginning, we chose to use normal search API which seems to search tweets with specific keywords easily. However, as we have tried it in many different ways, we found many problems with this API. Firstly, it can only gain up to 100 tweets once, which are not enough for our analysis. Secondly, although it can limit the search zones, the search zone must be a circle which means that it is difficult to define the zones without overlaps between two adjacent search zones. The last problem is fatal and it lets us to choose stream API. If we use the same commands of search API, such as searching the same keywords tweets in same zones, it may return the same data. It means that there are bunch of repeated data and we have no way to determine whether the returned data are repeated.

Compared with normal search API, stream API has many advantages. Firstly, it can gain tweets in real-time. Hence, if we deploy the system in cloud, it will gain data continuously. It is easy to gain enough tweets for analysis in several days. Secondly, the tweets search zone can be a rectangle, which is better than circle since it can help avoid overlapping zones. Thirdly, because the data are gained in real-time, there will not be repeated tweets and the proportion of the number of tweets will be corresponded to the populations in different areas. Because of these advantages, we choose stream API to gain data which can make great advantage of the clouds.

The security of the system should also be considered. Since the database is the key of this system, it is necessary to prohibit others visiting the databases directly. Hence, appropriate security groups should be chosen. For the sake of communication between these nodes, the ports for CouchDB cluster are opened to specific IP addresses so that only these three nodes can communicate with others in port 4369 and 5984. Besides, the port 80 is exposed to all the IP addresses for users to visit this app.

Since the Melbourne Research Cloud has limitation to VMs, the proxy should be added to make sure the VMs can visit public IPs. Except the instance proxy, the docker containers also need a proxy or the tweets\_harvesters will not be able to gain tweets.