# Cluster and Cloud Computing Assignment 2
# City Analytic on the Cloud

Zhaofeng Qiu 1101584

University of Melbourne — May 25, 2020

## 1 System Design and Architecture

The architecture of our system is well designed. It is aimed to provide both a beautiful front-end web application and a RESTFul API server with high availability, high scalability, and fault tolerance. The overall architecture is shown in Figure 1.

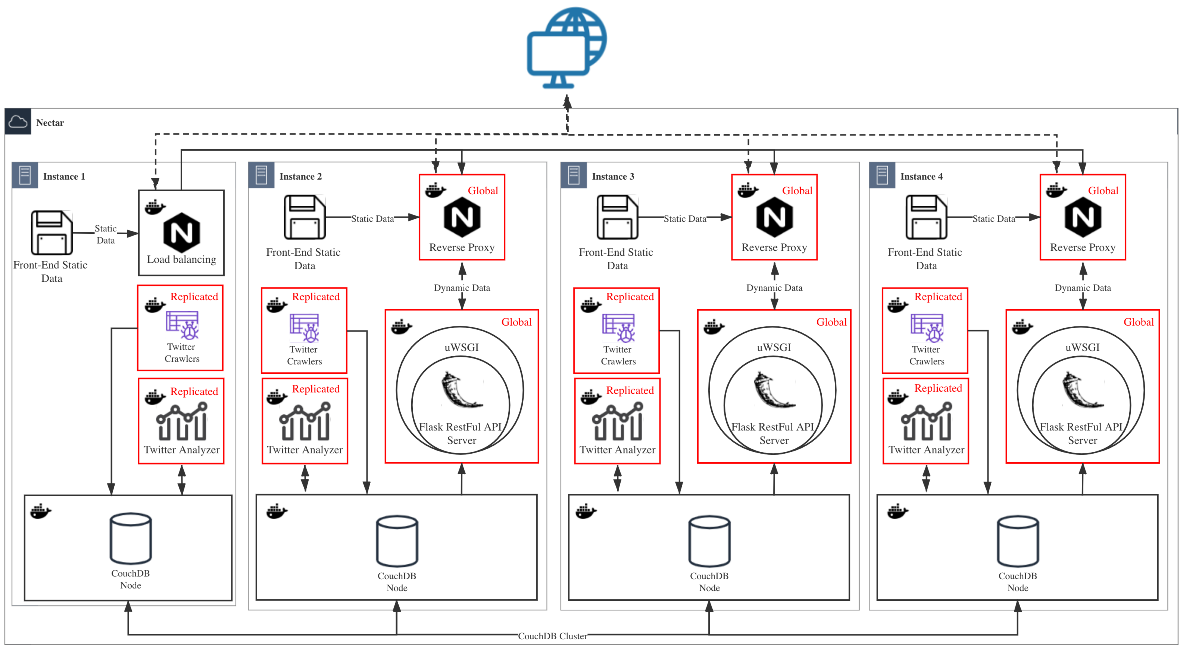

Figure 1: Velocity Re-estimation

### 1.1 High Availability

We continue to optimize our architecture through in-depth thinking and repeated practice to achieve high availability with limited resources. We mainly use the following method to improve the availability of our system.

Firstly, we run an *Nginx* server in our first instance to provide Load balancing. For requests that need to access the dynamic data in our database, it would pass the request to the other three servers averagely. In this way, we can efficiently distribute the incoming network traffic to our backend servers in the cluster. When it comes to the situation where the first instance may crash, users can still access our application through the IPs of the other three nodes. However, this solution is not good enough and is contradict to high availability. We had tried to use two *Nginx* server with a software called Keepalived to provide a more fault-tolerant system. However, we found out that we are not allowed to use floating IP in *Nectar*. Since we cannot set a virtual IP for our *Nginx* servers, the only way to handle this issue is to let the other three nodes to keep the full functionality of our application. Users can access our application through any IP of our instances, but only the IP of the first instance would provide load balancing control.

Secondly, we use *Docker Swarm* to manage part of our services. The containers which are in red boxes in Figure 1 are managed by *Docker Swarm*. The containers that with a replicated deploy mode setting would run in each of our machines except Instance 1. The containers with a replicated deploy mode setting would run in one of the four nodes. Actually, we have twelve *Twitter* Crawlers in our cluster, each for different purposes. They would be distributed to different nodes in our cluster by *Docker Swarm*. The *Docker Swarm* does not manage the *CouchDB* container and the Load Balancing *Nginx* server. But they can restart themselves if they crush since we set the "restart" configuration in docker-compose to "always". We didn't include the *CouchDB* into the *Swarm* cluster because using *Swarm* to manage the *CouchDB* database could be counter-productive if the *Swarm* manager deploys two *CouchDB* together to the same node. The Load balancing *Nginx* server is also not included because its port number is in conflict with the reverse proxy *Nginx* server in the *Docker Swarm* network. We set three *Swarm* manager nodes in our cluster. Any instance with an ID bigger than three would be set to be a worker node. A single leader is elected from the three *Swarm* manager nodes to conduct our orchestration tasks. Our manager would also run the services as worker nodes. According to [1] and *Swarm* Admin Guide[1], the number of our manager nodes is set based on the following facts:

- More managers would result in a longer election for a new leader when one goes down. The *Swarm* is in a read-only state which means that no new replica tasks can be launched and no service can be updates. No auto-recover can happen during the election.
- More managers would require tighter management of resources to prevent manager starvation.
- Fewer managers would increase the probability of all managers going down.
- To support manager node failures, the number of managers should be an odd number and should bigger than one. In our cluster, we can only provide four nodes. So, setting the number of managers to be three in our system is our best choice. In this way, our system can provide fault tolerance that one of our manager nodes can crash.

Thirdly, we use a *CouchDB* cluster with four nodes to improve our system's availability. We keep three replicas in our cluster, which means that any two nodes can be down in our system without crashing down our application. The Flask RESTFul API server, the *Twitter* Crawler, and the *Twitter* Analyzer of our system would only access the *CouchDB* database in their localhost. In this way, errors on a single node will not affect other nodes.

In conclusion, our system is a fault-tolerant system with high availability. Our application could run well even if two nodes in our cluster crashed. Except for the first node, any node in our system can provide all the services we need to run our front-end web application and RESTFul API server themselves. The system can be easily improved to a real high availability system by adding one more Load balancing *Nginx* server and using Keepalived to manage the floating IP system.

## 1.2 Dynamic and Static Separation

We use *Nginx* to separate static from dynamic traffic. For accessing the static data on our websites like HTML and image files, the *Nginx* server would handle them itself. For accessing the dynamic data from the database, the reverse proxy *Nginx* server passes the request to the uWSGI-Flask server through socket protocol. When using a socket, external browsers will not be able to directly access our uWSGI-Flask service, which can provide additional safety. It is faster to transfer through the socket. There are many advantages to separate static traffic. For one thing, it can help reduce the file I/O and storage consumption of our uWSGI-Flask server. *Nginx* is good at handling static traffic, which can provide a quick response to our front-end application. Further more, we can reuse the RESTFul API in our front-end application to reduce workload.

[1]https://docs.docker.com/engine/swarm/admin_guide/

## References

[1] B. Fisher, “Pros and cons of all managers as workers in swarm,” https://stackoverflow.com/questions/48853473/pros-and-cons-of-running-all-docker-swarm-nodes-as-managers, 2018.