### **Extended service**

Using existing setup, we can consider below points to enhance application high availability.

**1.Based on the CPU we can increase the worker count in Gunicorn.**

Each of the workers is a UNIX process that loads the Python application. There is no shared memory between the workers.

The suggested [number of workers](http://docs.gunicorn.org/en/latest/design.html#how-many-workers) is (2\*CPU)+1.

**Ex:** For a dual-core (2 CPU) machine, 5 is the suggested workers value.

gunicorn --workers=5

**2.We can use gunicorn threds to speed up the process**

Gunicorn also allows for each of the workers to have multiple threads. In this case, the Python application is loaded once per worker, and each of the threads spawned by the same worker shares the same memory space.To use threads with Gunicorn, we use the threads setting. Every time that we use threads, the worker class is set to gthread:

**Ex**: gunicorn --workers=5 --threads=2

**3.worker-connections**

There are some Python libraries such as [gevent](http://www.gevent.org/) and [Asyncio](https://docs.python.org/3/library/asyncio.html) that enable concurrency in Python by using “pseudo-threads” implemented with [coroutines](https://en.wikipedia.org/wiki/Coroutine).

Gunicorn allows for the usage of these asynchronous Python libraries by setting their corresponding worker class.Here the settings that would work for a single core machine that we want to run using gevent:

gunicorn --worker-class=gevent --worker-connections=1000 --workers=3

* When pipeline starts to ingest heavy data load into database, it will slowdown read api. Same time when it starts to read and write it will make a bottleneck.
* We can setup a Datadog Agent our gunicorn server to monitor the application health. Based on that we can improve performance. Using Datadog we can check Latency, throughput, HTTP status codes, resource utilization.
* Read and write operations need to be implemented in separate applications. So, the scaling can be done independently, also we can use batch processing service for update operation. Based on the nature of the batch process we are required to execute we can implement one of the following approaches.

**Additional questions**

1. Having EC2 instance template and configure autoscaling group with that. For batches to updated, create SQS and integrate that to the write application. Based on the queue items and the load using AWS cloud watch we can scale the group. And we can use AWS postgres as the backend for the database, we can have read replicas connected to the read application while write node connected to the batch application.

2. If the processing of the single batch does not take more than 15 minutes, we can create AWS Lambda function which will be consuming SQS messages with the batches need to be executed. Once a batch is completed it can remove the message from the queue.

3. Setting up AWS batch, by defining Compute/Job Definition/Job.

we can use AWS ECS orchestrate read and write application container and load balancing.