Enable this web application to return two requests concurrently. Ex: Open two tabs and load /csv_test at the same time. Both requests with delay = 10s return in 10s (not 20s).

1. Please explain more than one approaches to tackle this problem – and the justification behind your choice.

Solution:

Flask's built-in server is **NOT** suitable for production as it doesn't scale well and by default serves only one request at a time. There are a variety of approaches to pull this off (handling concurrently requests).

Gunicorn is a solid, easy-to-use **WSGI** server that will spawn multiple workers (separate processes), and even comes with asynchronous workers that will speed up the app and make it more secure.

Gunicorn works by internally handing the calling of flask code. This is done by having workers ready to handle the requests instead of the sequential one-at-a-time model that the default flask server provides. The end result is the app can handle more requests per second.

One simple approach is to create 2 workers (using --*workers* environment variable) that will handle both requests concurrently. I chose this approach as it achieves true parallel processing. Workers are the number of process that gets spawned. The operating system will then schedule these processes to run on different cores. In this way we can have several (in our case 2 workers) workers running at the same time and finish the computation independently on different cores. Synchronization problems are less common than with threads.

Another approach is to keep one worker only and increase the number of threads in that worker. Since threads are more lightweight (less memory consumption) than processes, I will have only one worker and two threads. *Gunicorn* will ensure that the master can then send more than one requests to the worker. Since the worker is multithreaded, it is able to handle TWO requests.

The third approach is to change the worker type to Async. This is extremely lightweight. Even threads consume memory. There are coroutines implemented by *gevent* library that allow you to get threads without having to create threads. We can configure Gunicorn with *gevent* async worker.

An important point worth mentioning here is Flask doesn't spawn or manage threads or processes. That's the responsibility of *Gunicorn*.

2. Explain possible sources of latency for the application above. What type of possible conditions may make latency better/worse?

Possible source of latency:

(a) The obvious latency in the application is the delay time that the user submits to the application.
(b) The static data (csv file) is stored in S3. It takes time to fetch the data (csv file) from the geographic location it is stored in.
(c) Network Latency (bandwidth) is another source of latency in the application.

Steps to make the latency better:

(a) The application is idle after the user submits the delay time. During this time, it would be efficient to fetch the data (csv file) stored in S3 and return it right after the delay expires. This is obviously faster than fetching the data after delay time. This way we can reduce the idle time. (Parallelism)
(b) Since the **same** data (csv file) is fetched for all requests, it will smarter to introduce a caching layer between the application and S3. (Caching)
(c) To reduce the network latency we can leveraging edge nodes in Content Delivery Network (CDN).

3. What is the first thing you would do in order to "scale" this app up to 1MM possible users?

(a) Spawn the application through multiple ***EC2*** instances and a load balancer to distribute load and serve requests.
(b) Scale web servers (***Gunicorn***) by using multiple threads or multiple workers.
(c) Use *Asynchronous* flask requests.
(d) Use a task queue like Celery. It is desirable to have a fixed pool of workers dedicated to running asynchronous tasks.
(e) Use a cache since the same data is always retrieved. Serve static content from S3 via CloudFront.