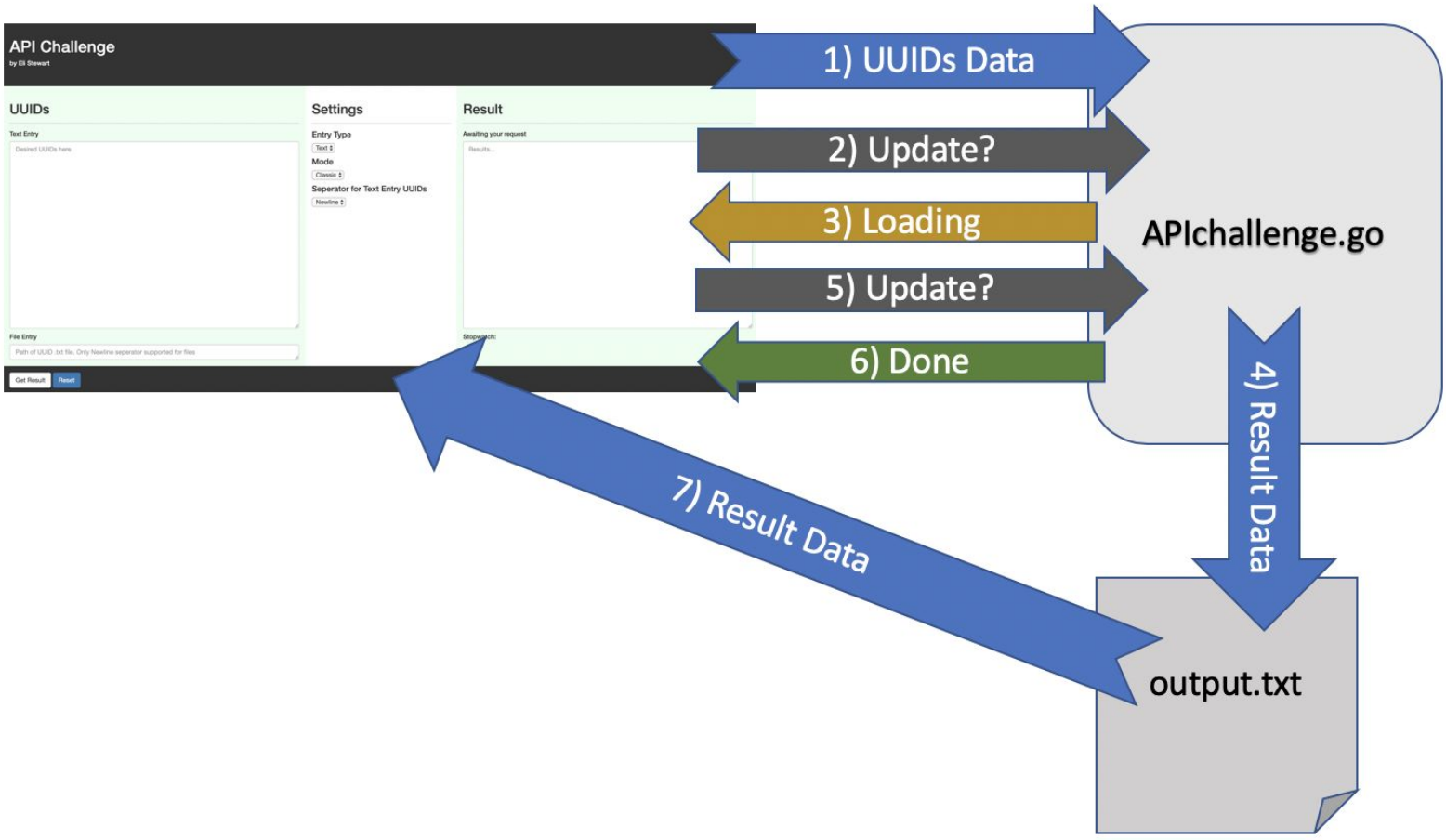
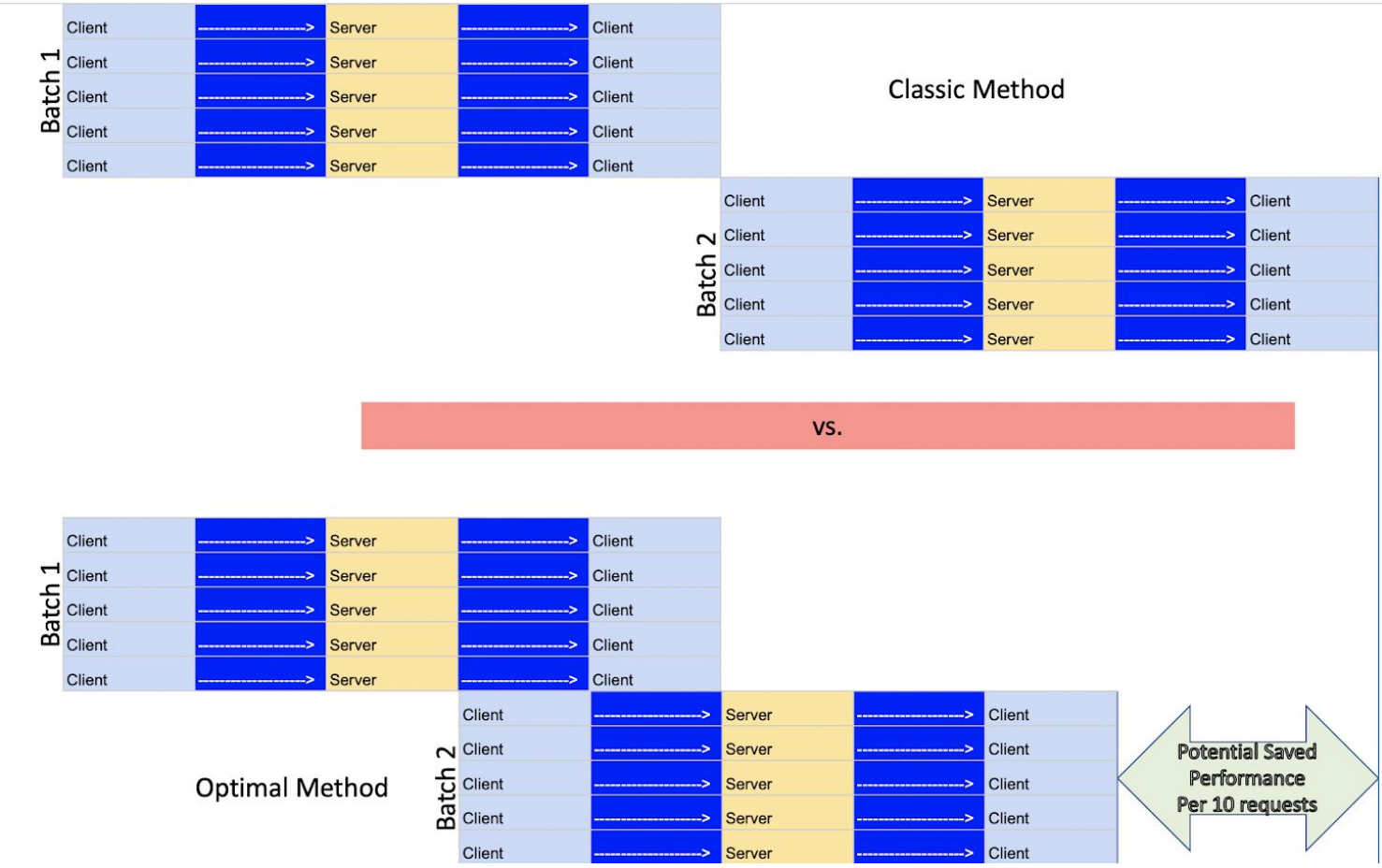
How it works:

The UI/browser sends the user's requested UUIDs to the APIchallenge.go server which then begins requesting the data for those UUIDs from the API. The UI makes periodic update requests to the APIchallenge.go server. Once the results for the UUIDs have all been retrieved, that data is put into static/output.txt. On the next update request from the UI, the server tells the UI that processing is done. Then the UI loads in the result data from static/output.txt and displays the result to the user.



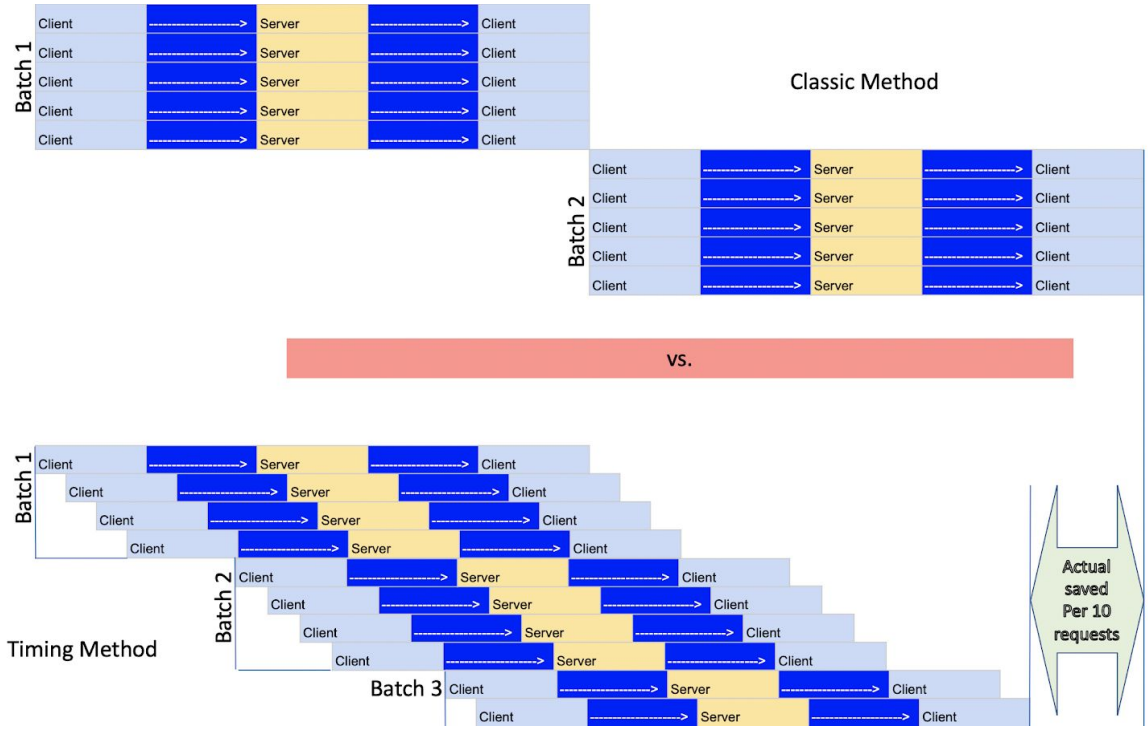
Idea for the timing method:

I initially wrote up the code for the *classic* function in APIchallenge.go. However, I realized that this method requires the client to wait for a response before requesting the next UUID from the API server. I imagined that the optimal method would send another batch of requests immediately after the API server finished processing the first batch and sent out its response. That way, the throughput of requests could be much higher. (Illustrated in the diagram below)



Implementation of the timing method:

I tried to write a program that would achieve an approximation of the optimal method. However, I found that if I tried to send out the requests 5 at a time with some waiting time in between them, no matter the length of the waiting time, I would end up triggering the API request limit way too much. I would then have to request data for the rejected requests again ultimately making these first implementations of the timing method slower. I tried putting some small wait time between each request within a batch and reducing the batch size to 4. This greatly reduced the amount of 429 Too Many Requests responses I received.



After some tweaking of the wait times, the timing method started performing slightly better than the *classic* method (on my computer and with my internet connection). Stopwatch results below:

# of Requests	20	100	300	1386
Classic (in sec)	5.316380238	24.54787675	69.44126692	311.6270118
Timing (in sec)	5.638810569	23.38961819	62.91119987	290.1241564
Classic/Timing	0.9428194427	1.049520199	1.103798164	1.074116046

However, the *timingMethod* function that I wrote does not offer as much improvement over *classic* as I had hoped. Furthermore, the values I used for waiting times between requests work well for my computer and connection but not necessarily for all. To improve *timingMethod*, I had plans to write algorithm that would test the performance of *timingMethod* using a wide range of different waiting times to determine the optimal timing strategy for my connection speed. To further improve, I would run that algorithm on a wide range of different connection speeds to compile a database of optimal timing strategies. Then, when a new user uses the utility, the timing method function would ping the API to determine the user's connection speed. Based on the user's connection speed, the program could determine the best timing strategy for that user.