CLD Lab 04: App scaling on IaaS

Authors: Kevin Auberson and Léo Zmoos - L4GrR

Date: April 23, 2024

TASK 1: DEPLOYMENT OF A SIMPLE WEB APPLICATION

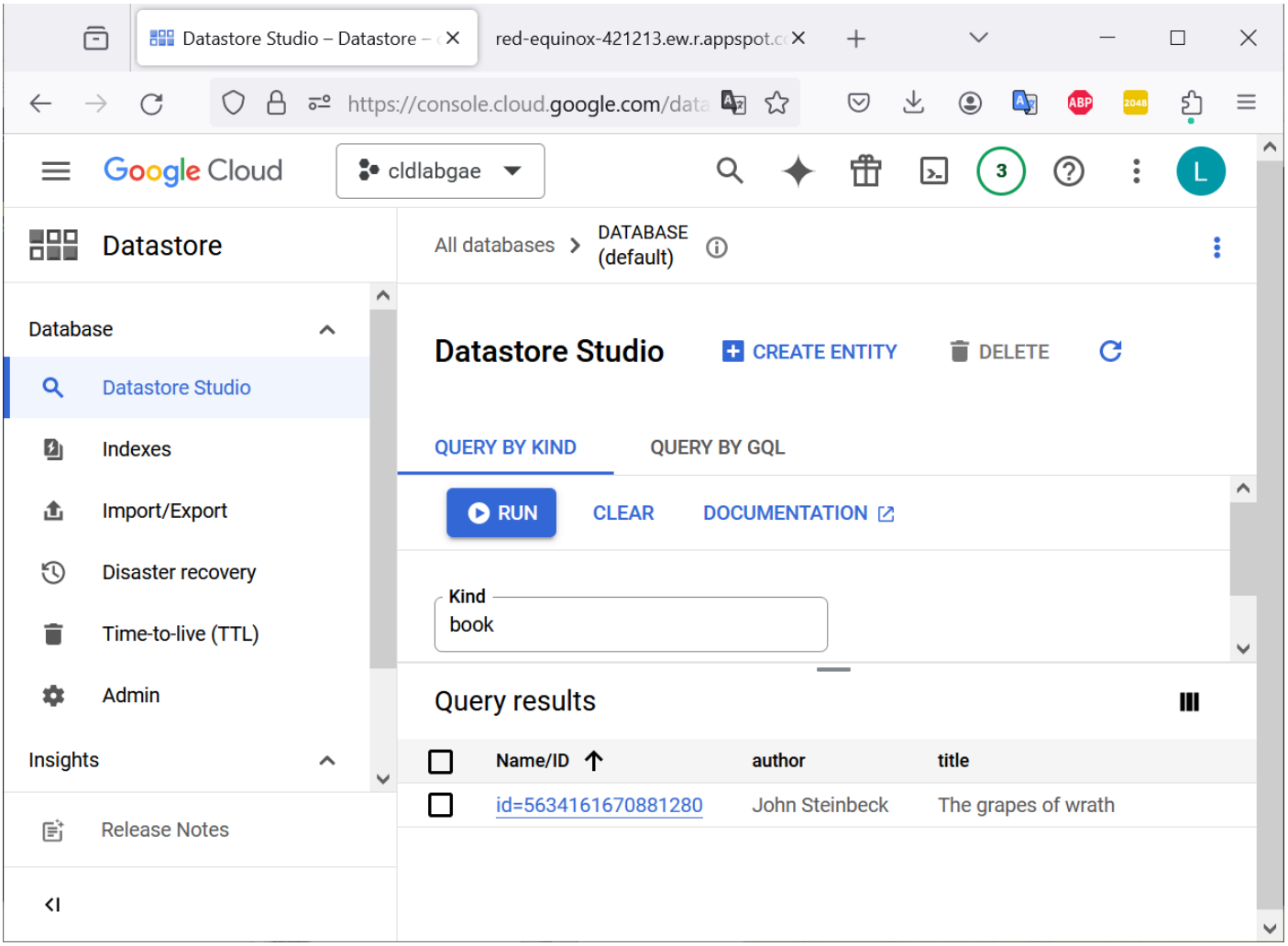
> Copy the Maven command to the report.

Maven command:

```
./mvnw clean package --batch-mode -DskipTests -Dhttp.keepAlive=false -f=pom.xml --quiet
```

TASK 2: ADD A CONTROLLER THAT WRITES TO THE DATASTORE

> Copy a screenshot of Datastore Studio with the written entity into the report.



TASK 3: DEVELOP A CONTROLLER TO WRITE ARBITRARY ENTITIES INTO THE DATASTORE

> Copy a code listing of your app into the report.

```
@GetMapping("/dswrite")
    public String writeEntityToDatastore(@RequestParam Map<String, String> queryParameters)
        StringBuilder message = new StringBuilder();
        // Extract _kind and _key from query parameters
        String kind = queryParameters.get("_kind");
        String keyName = queryParameters.get("_key");

        // Create Datastore service
        Datastore datastore = DatastoreOptions.getDefaultInstance().getService();

        // Create KeyFactory for the given kind
        KeyFactory keyFactory = datastore.newKeyFactory().setKind(kind);

        // If _key is present, create a Key with the provided key name
        // Otherwise, let Datastore generate the key
        Key key;
        if (keyName != null && !keyName.isEmpty()) {
            key = keyFactory.newKey(keyName);
        } else {
            key = datastore.allocateId(keyFactory.newKey());
        }

        // Create Entity with the extracted key and kind
        Entity.Builder entityBuilder = Entity.newBuilder(key);
        message.append("Entity : ");
        // Add properties from query parameters (excluding _kind and _key)
        for (Map.Entry<String, String> entry : queryParameters.entrySet()) {
            String paramName = entry.getKey();
            String paramValue = entry.getValue();
            if (!paramName.equals("_kind") && !paramName.equals("_key")) {
                entityBuilder.set(paramName, paramValue);
                message.append(paramName + ": " + paramValue + ", ");
            }
        }

        // Build the entity
        Entity entity = entityBuilder.build();

        // Write the entity to Datastore
        datastore.put(entity);
        message.append("written to Datastore\n");
        // Return success message
        return message.toString();
    }
```

TASK 4: TEST THE PERFORMANCE OF DATASTORE WRITES

> For each performance test include a graph of the load testing tool and copy three screenshots of the App Engine instances view (graph of requests by type, graph of number of instances, graph of latency) into the report.

The following tests were carried out using vegeta. For the burst load test on a simple URL without parameters, we sent 50 and 1000 requests to the App Engine server with a ramp-up period of 1 second for a duration of 1 minute. For the second test with writing to the datastore with parameters in the URL with a similar configuration, but the numbers of requests are 50 and 100. The objective of these two categories of tests was to observe if writing in the datastore impacts performance or not in two different scenarios.

HelloWorld Controller

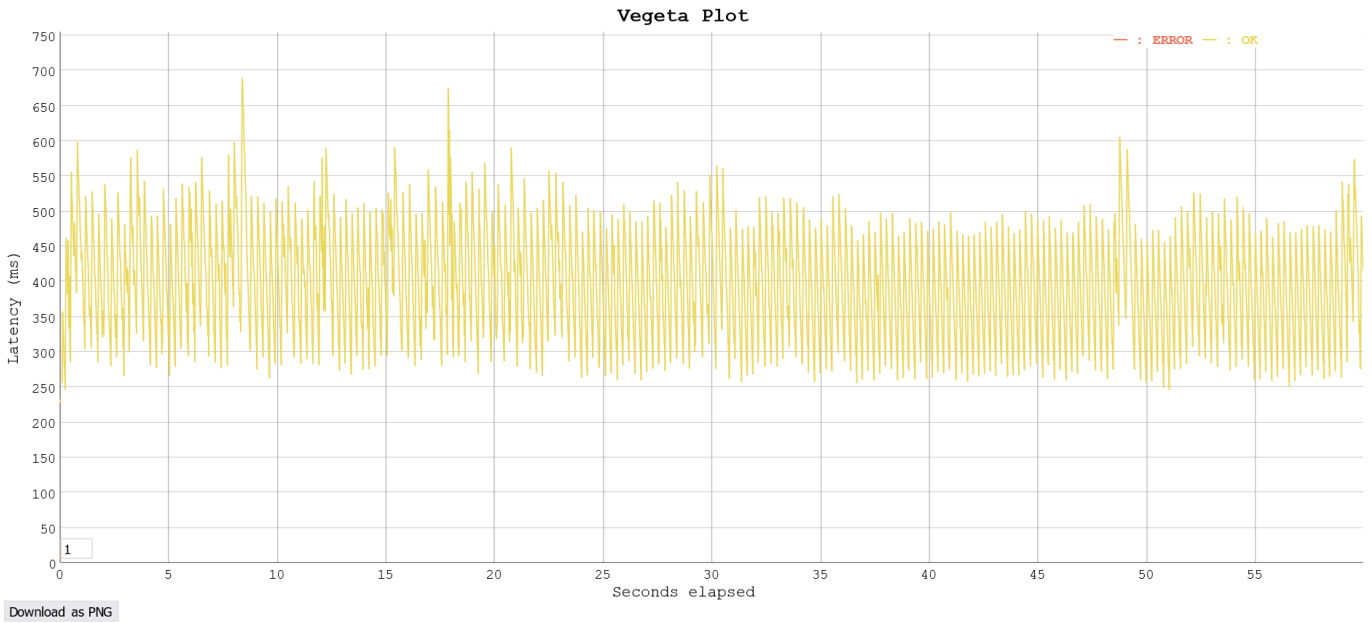
Default rate:

```
echo "GET https://ambient-stone-420513.uc.r.appspot.com/" | vegeta attack -duration=60s | t
```

```
dockerito@DESKTOP-6VLE4RG:~$ echo "GET https://ambient-stone-420513.uc.r.appspot.com/" | ./vegeta attack -duration=60s | tee results.bin |./vegeta report
Requests      [total, rate, throughput]         3000, 50.02, 49.59
Duration      [total, attack, wait]             1m0s, 59.979s, 415.759ms
Latencies     [min, mean, 50, 90, 95, 99, max]  188.1µs, 396.277ms, 394.184ms, 493.886ms, 515.814ms, 568.556ms, 686.895ms
Bytes In      [total, mean]                     38935, 12.98
Bytes Out     [total, mean]                     0, 0.00
Success       [ratio]                           99.83%
Status Codes  [code:count]                      0:5  200:2995
Error Set:
Get "https://ambient-stone-420513.uc.r.appspot.com/": dial tcp 0.0.0.0:0->[2a00:1450:400a:800::2014]:443: connect: network is unreachable
dockerito@DESKTOP-6VLE4RG:~$
```

Command result

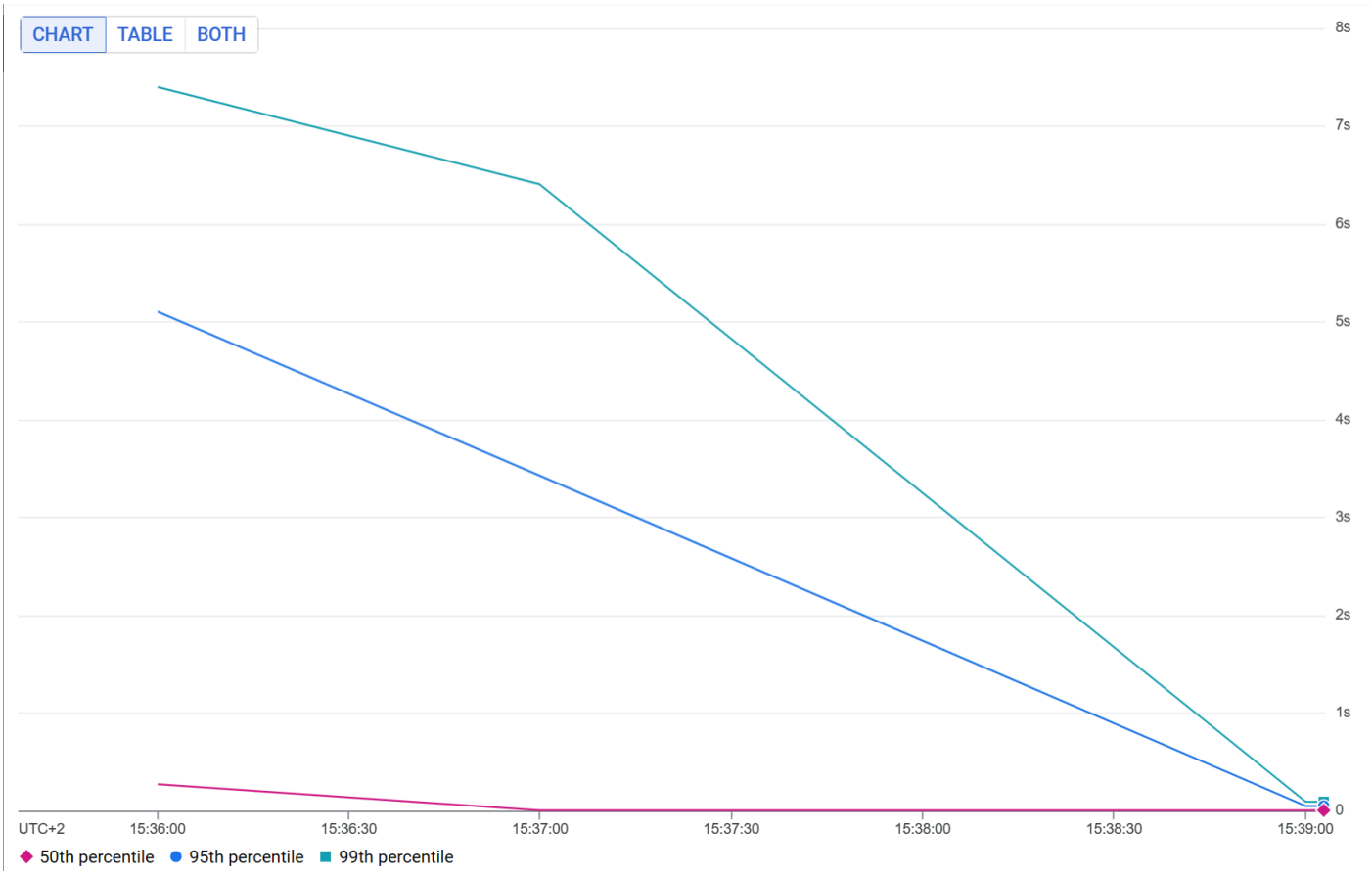
Vegeta plot (latency)



The Vegeta plot measures latency in milliseconds (ms), with fluctuations between approximately

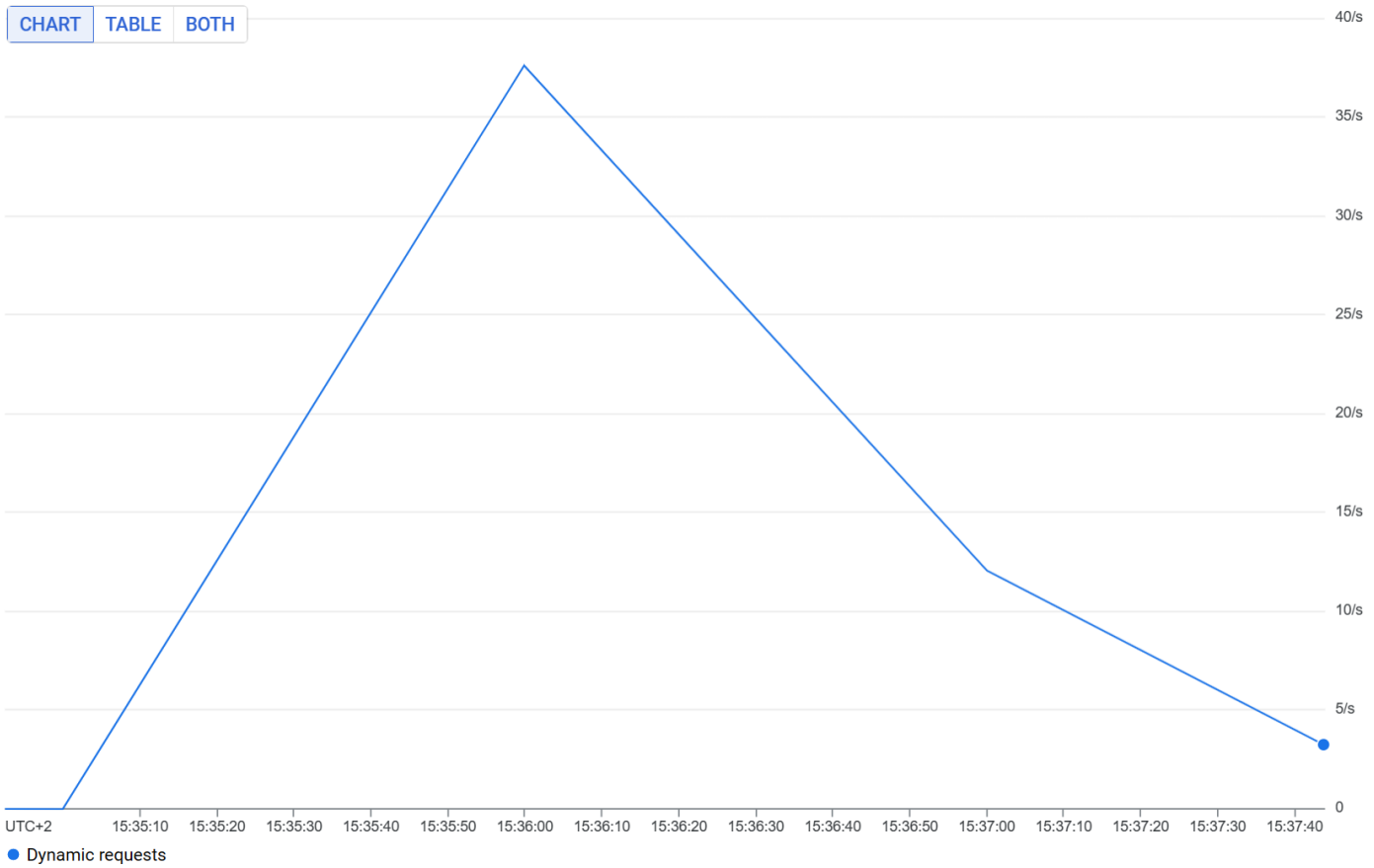
250ms and 750ms.

Latency



Throughout the test, the median latency (50th percentile) fluctuated between 1 second and 4 seconds. The 95th percentile latency line is generally higher than the 50th percentile line. The values range from around 2 seconds to 7 seconds. The 99th percentile latency line is the highest, the values range from around 3 seconds to over 7 seconds.

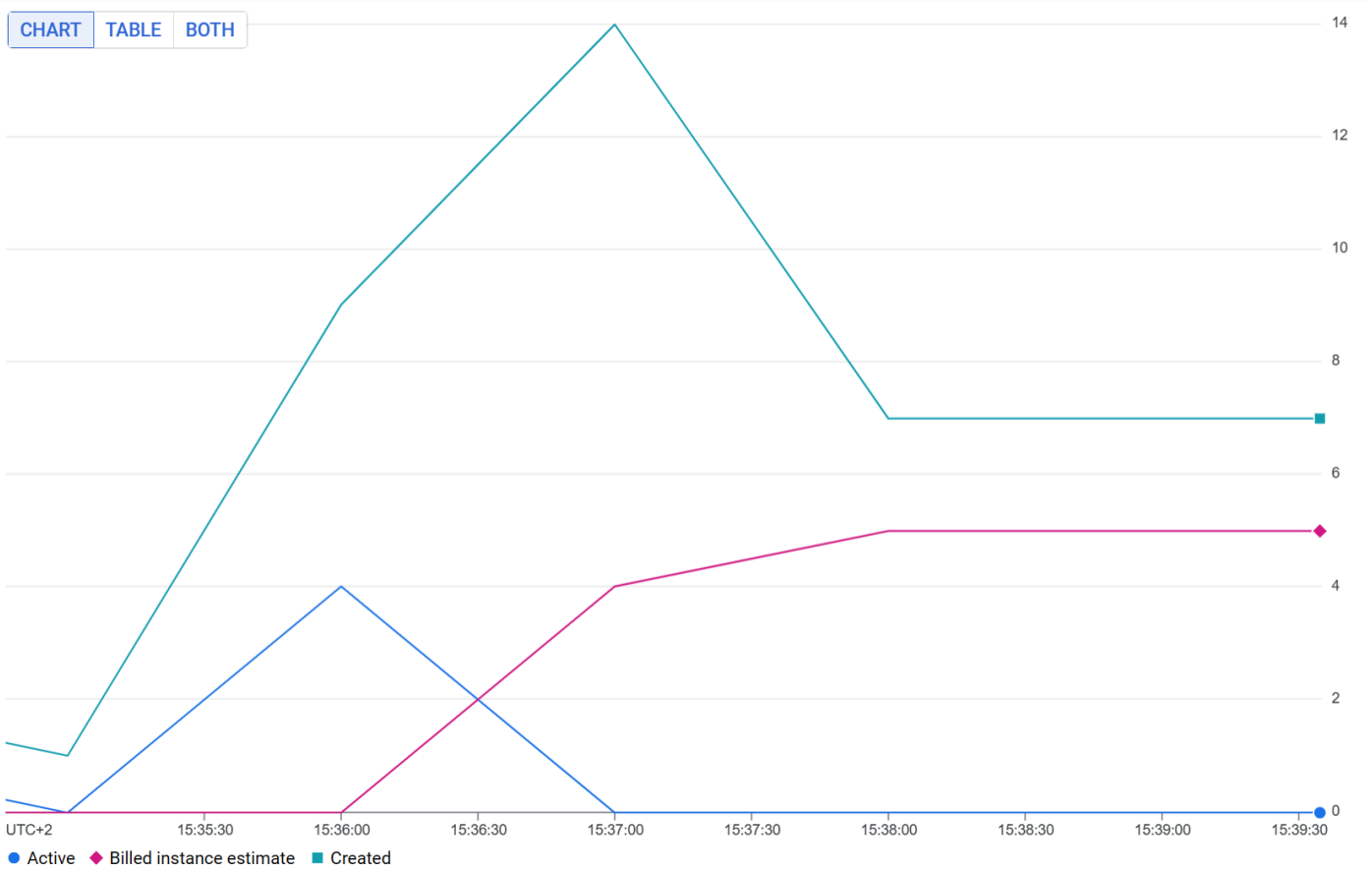
Requests by types



The data shows a very high volume of requests at the beginning of the time period, around 40 requests per second. The number of requests then steadily declines over the next two minutes to around 5 requests per second.

Using a default rate of 50 requests per second, we can see that the server has no problem processing all incoming requests.

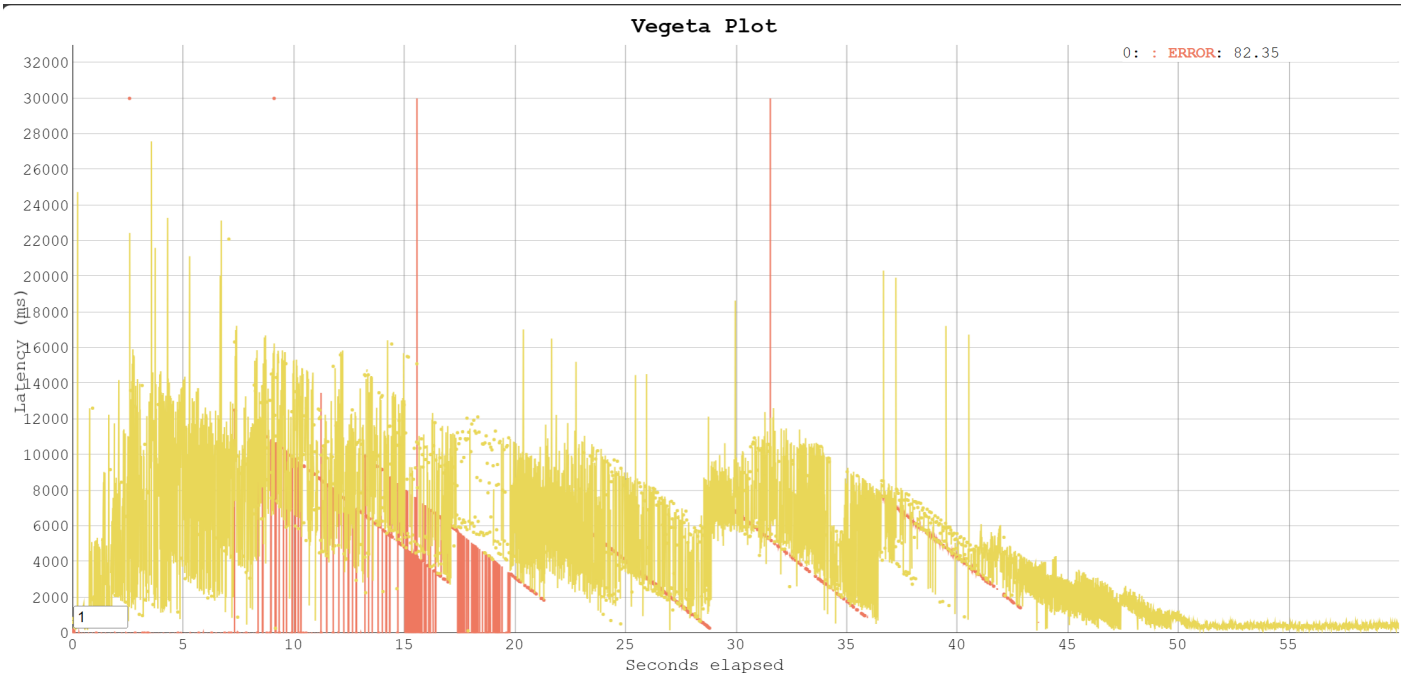
Instances



rate=1000:

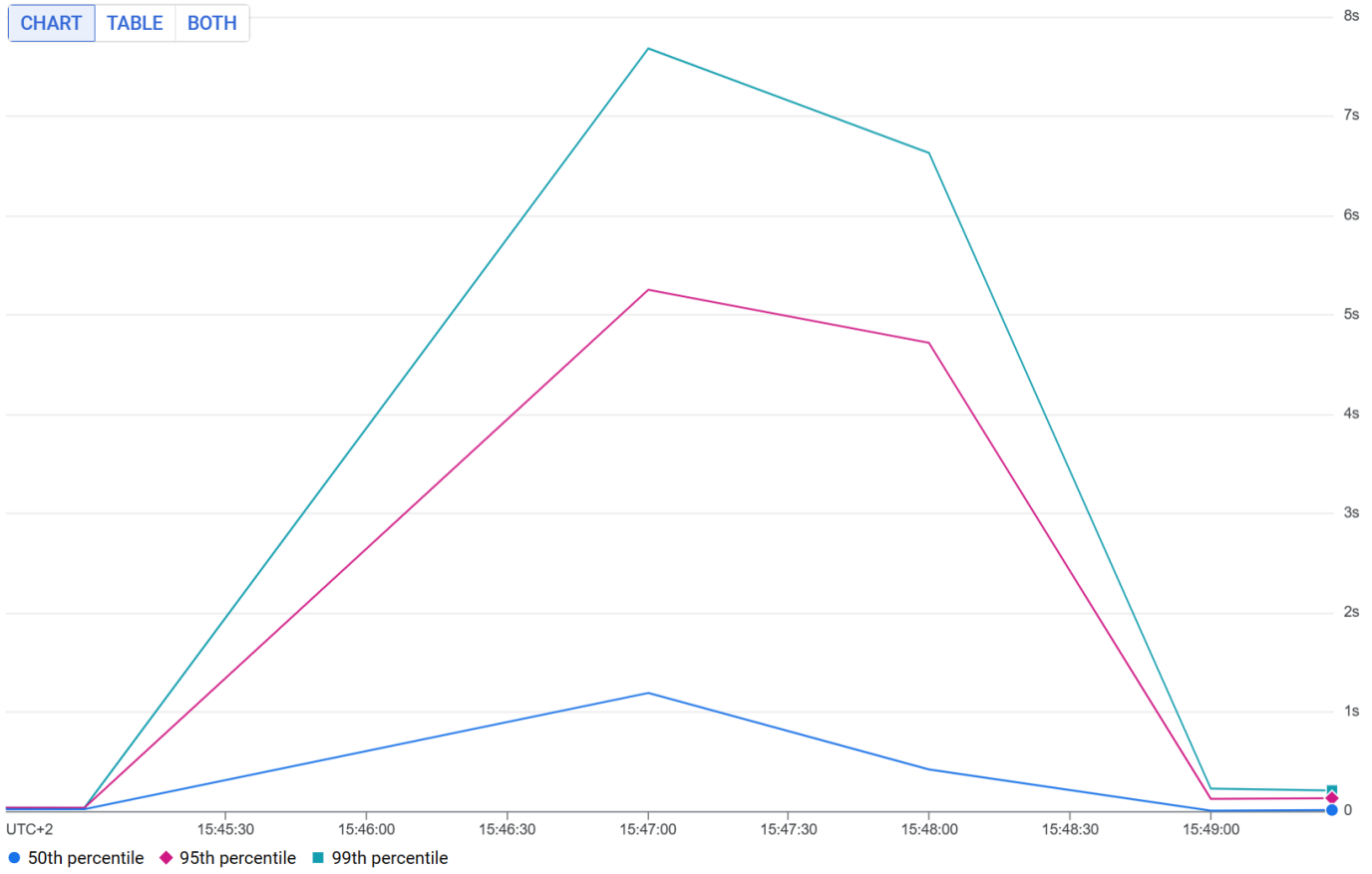
```
echo "GET https://ambient-stone-420513.uc.r.appspot.com/" | vegeta attack -duration=60s -ra
Requests      [total, rate, throughput]         60000, 1000.01, 885.52
Duration      [total, attack, wait]             1m2s, 1m0s, 1.573s
Latencies     [min, mean, 50, 90, 95, 99, max]  22.467µs, 4.538s, 4.285s, 9.438s, 11.012s,
Bytes In      [total, mean]                     1592540, 26.54
Bytes Out     [total, mean]                     0, 0.00
Success       [ratio]                           90.87%
Status Codes  [code:count]                      0:2740  200:54524  500:2736
Error Set:
Get "https://ambient-stone-420513.uc.r.appspot.com/": dial tcp 0.0.0.0:0->[2a00:1450:400a:8
500 Internal Server Error
Get "https://ambient-stone-420513.uc.r.appspot.com/": net/http: request canceled (Client.Ti
```

Vegeta plot (latency)



The graph shows a steady increase in concurrent users with no sharp drops, it suggests that App Engine was able to scale up to handle the increasing load from Vegeta's simulated users. In concurrent users followed by a sudden drop, it might indicate that App Engine reached a bottleneck and could not handle more users.

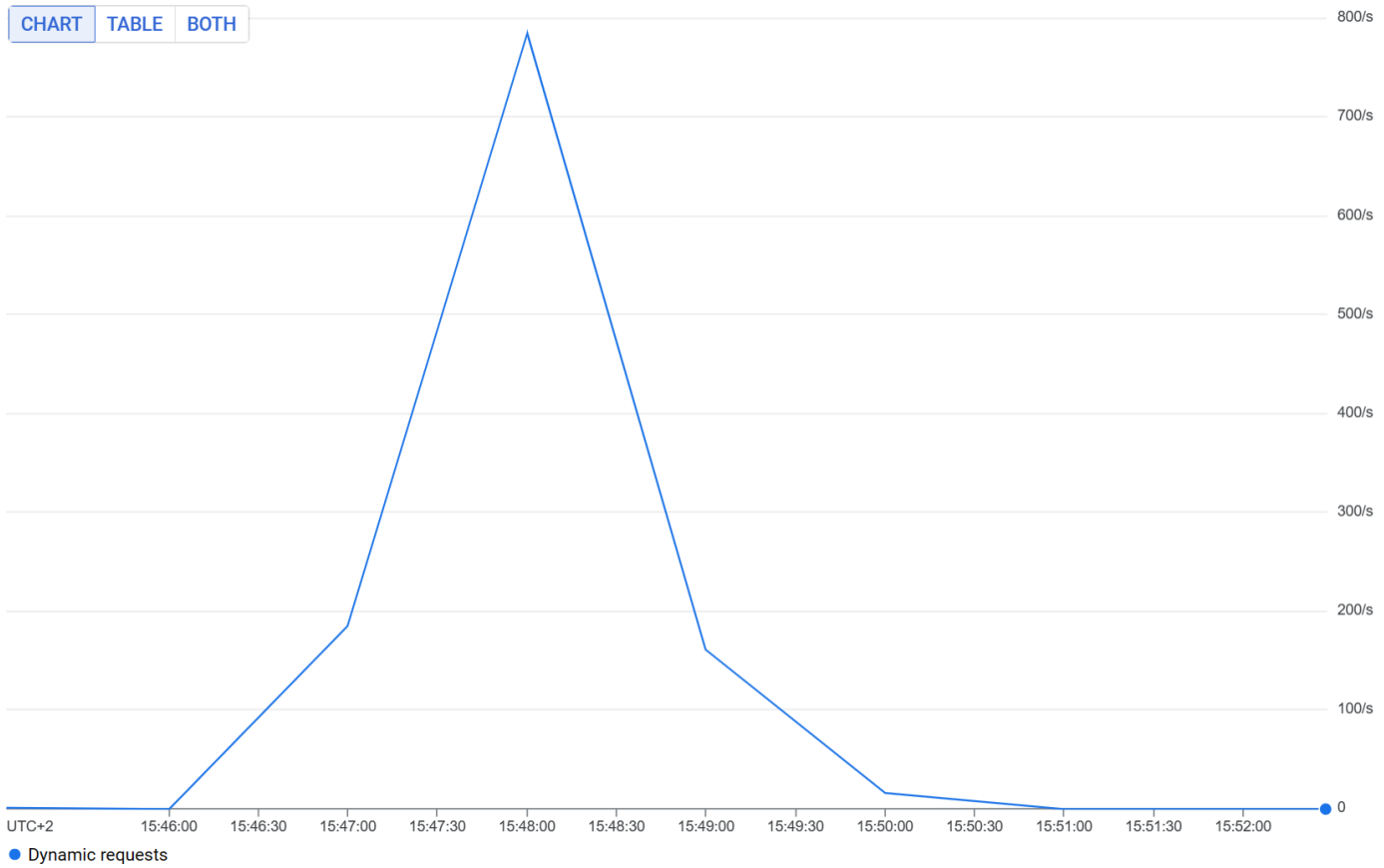
Latency



Throughout most of the test, the median latency (50th percentile) fluctuated between 2 seconds and 4 seconds. The 95th percentile latency line is generally higher than the 50th percentile line. The values range from around 3 seconds to over 7 seconds. The 99th percentile latency line is the

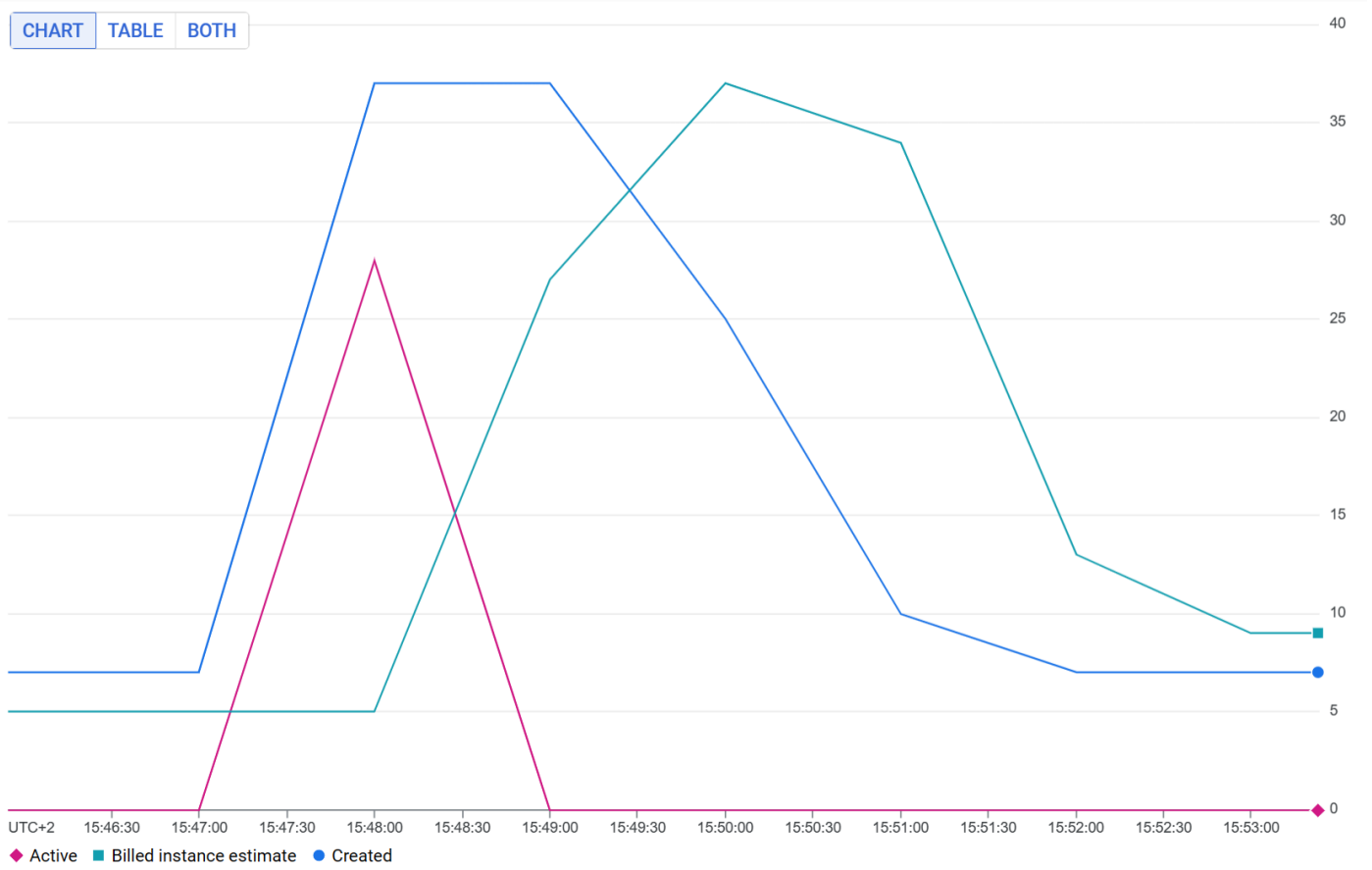
highest, the values range from around 4 seconds to over 8 seconds.

Requests by types



The data shows a very high volume of requests at the beginning of the time period, around 800 requests per second. The number of requests then steadily declines over the next one minutes to around 150 requests per second.

Instances



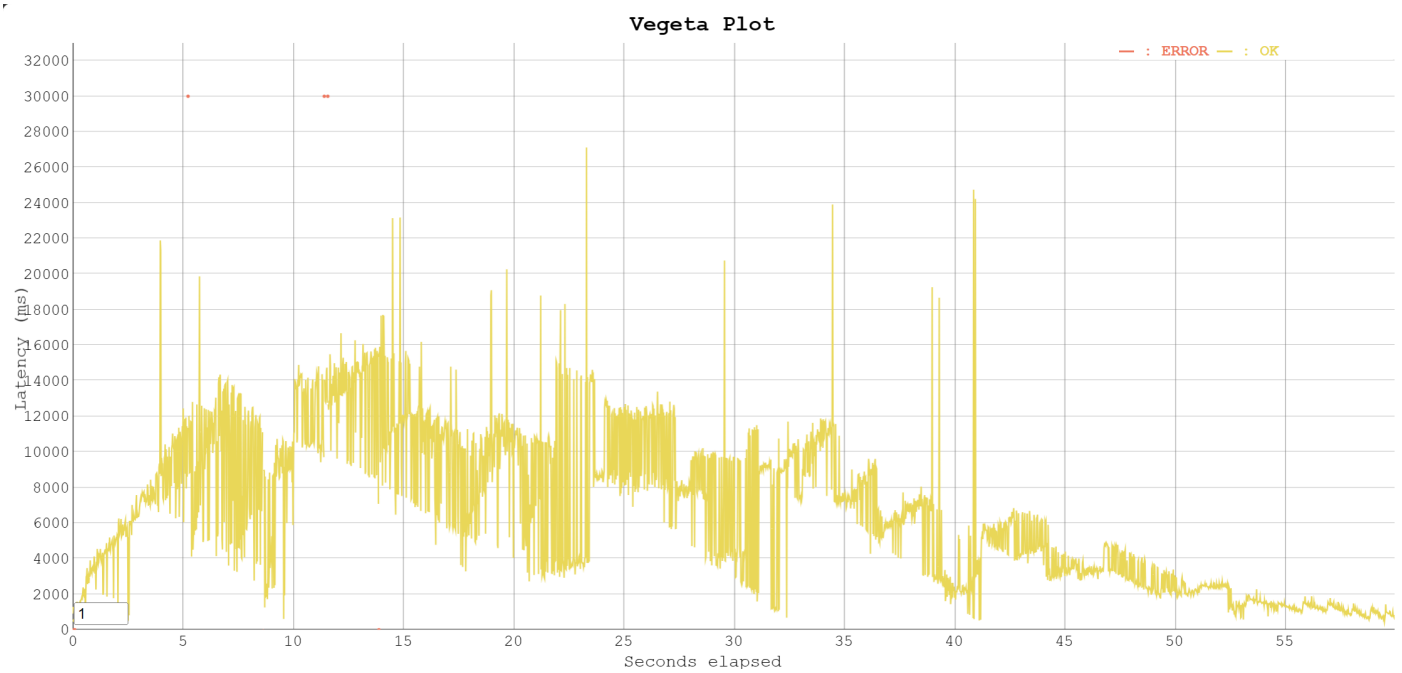
Here are the results after running vegeta with a rate of 1000 requests/s.

Datastore controller

Default rate

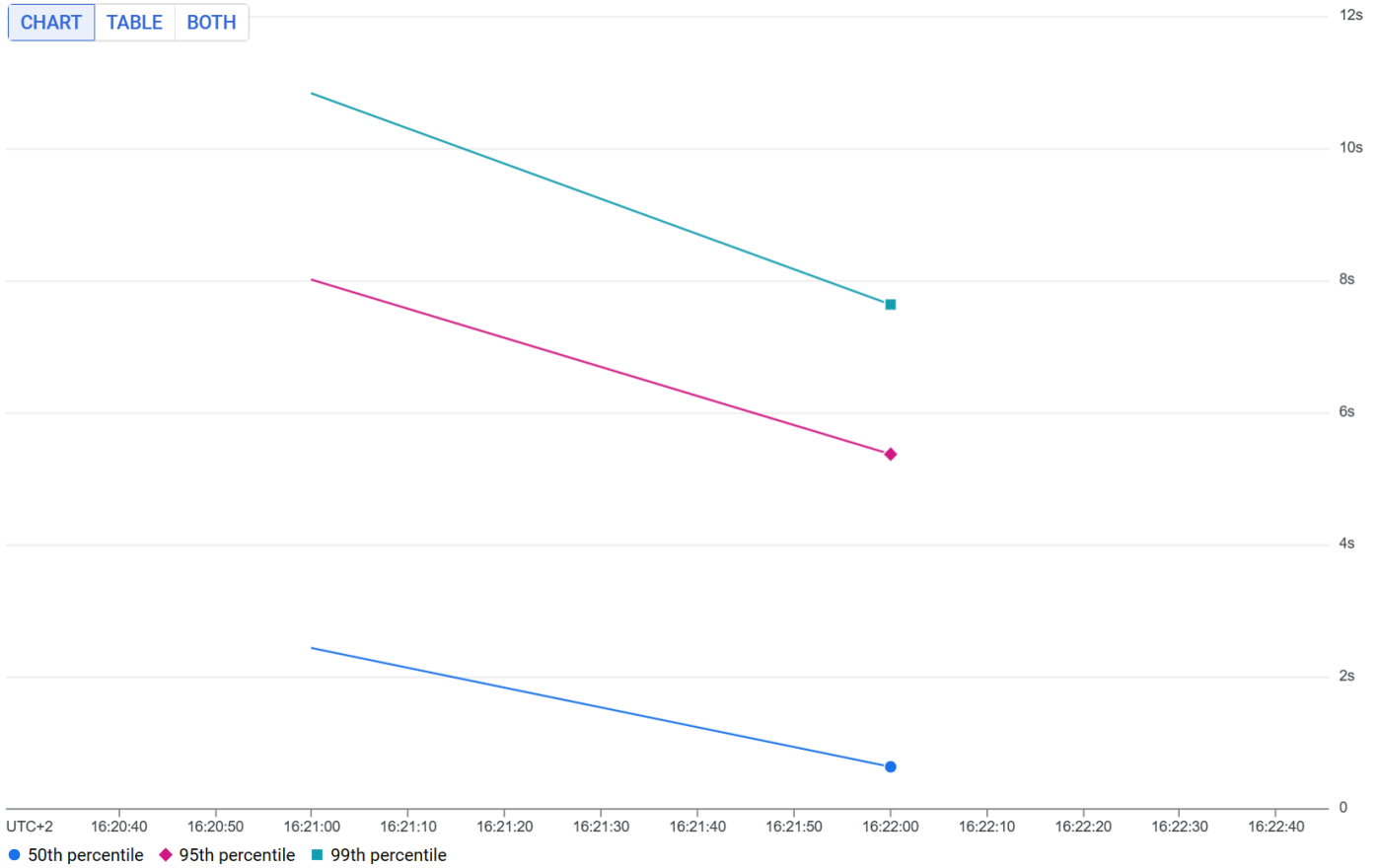
```
echo "GET https://ambient-stone-420513.uc.r
.appspot.com/dswrite?_kind=book&author=John%20Steinbeck&title=The%20Grapes%20of%20Wrath&type
a attack -duration=60s | tee results.bin | vegeta report
Requests      [total, rate, throughput]         3000, 50.02, 45.59
Duration      [total, attack, wait]             1m6s, 59.98s, 5.645s
Latencies     [min, mean, 50, 90, 95, 99, max]  372.759µs, 6.663s, 6.182s, 12.241s, 14.017s
Bytes In      [total, mean]                     281248, 93.75
Bytes Out     [total, mean]                     0, 0.00
Success       [ratio]                           99.73%
Status Codes  [code:count]                      0:8  200:2992
Error Set:
Get "https://ambient-stone-420513.uc.r.appspot.com/dswrite?_kind=book&author=John%20Steinbe
Get "https://ambient-stone-420513.uc.r.appspot.com/dswrite?_kind=book&author=John%20Steinbe
Get "https://ambient-stone-420513.uc.r.appspot.com/dswrite?_kind=book&author=John%20Steinbe
```

Vegeta plots (latency)



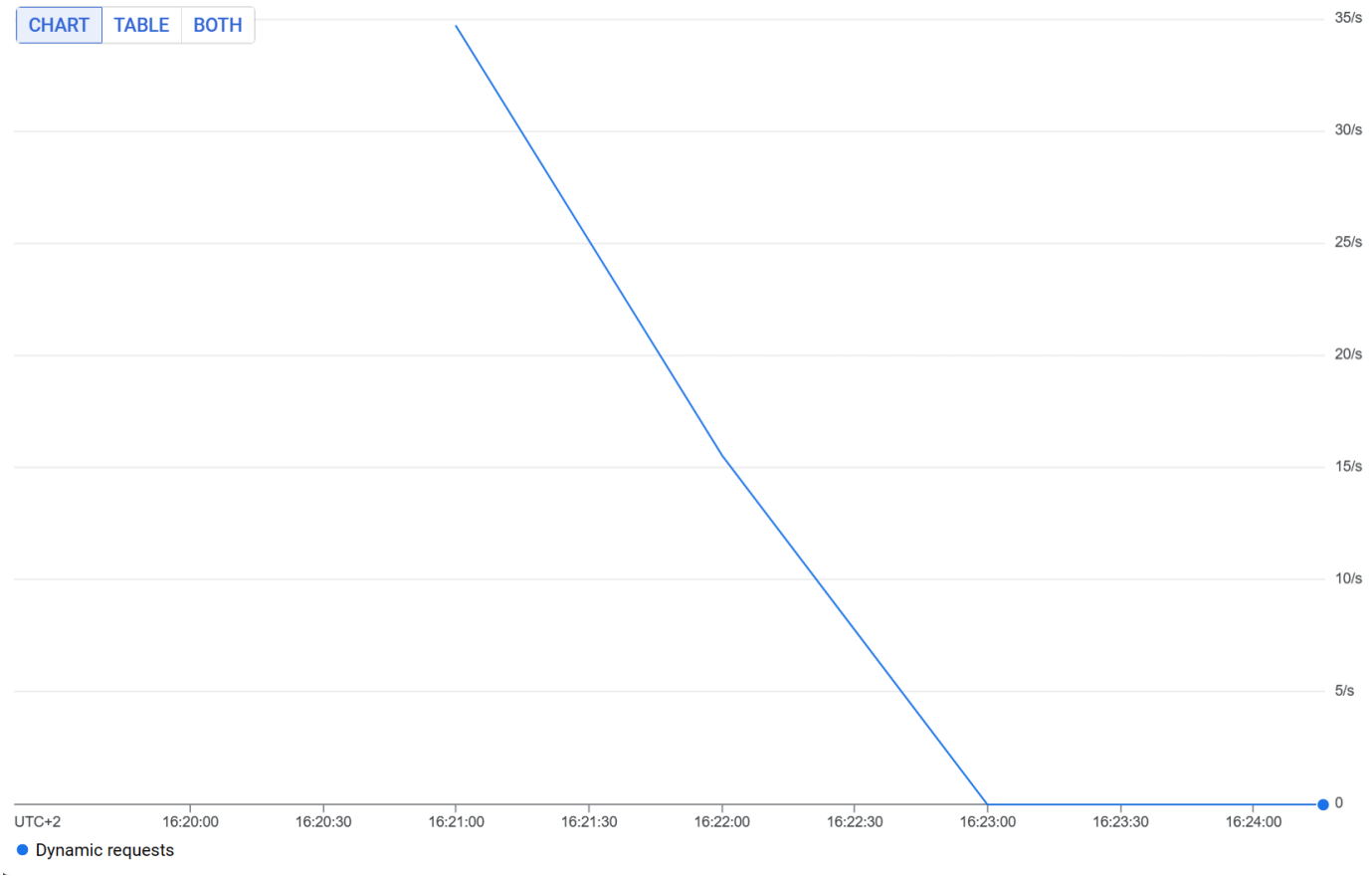
The error rate appears to be relatively low throughout the test, staying consistently below 1% for the duration displayed. The Datastore updates during the Vegeta load test were successful.

Latency



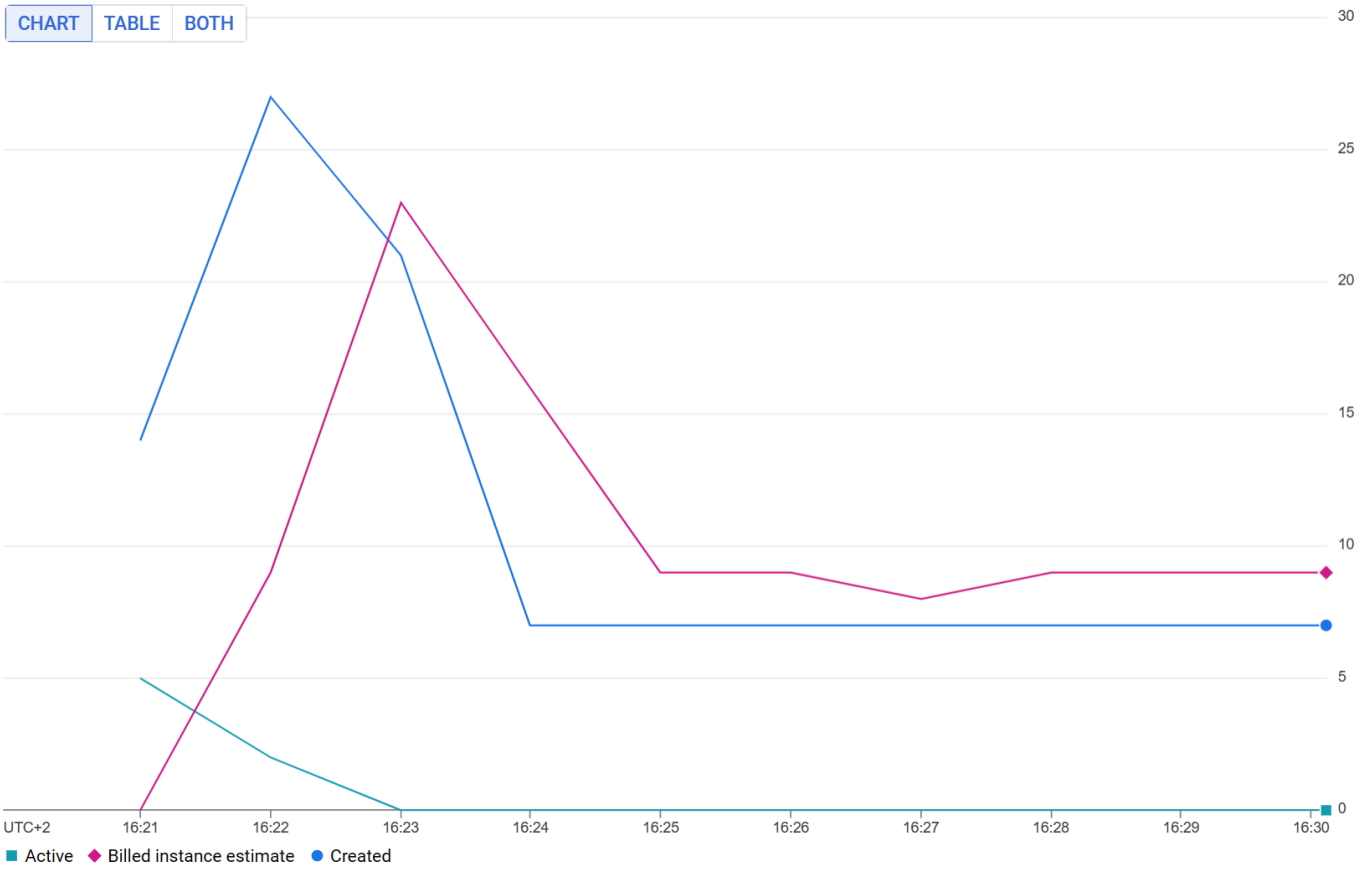
The 50th percentile line shows that for half of the requests, it took less than around 3 seconds to complete the update. The 95th percentile line shows that for 95% of the requests, it took less than around 8 seconds to complete the update. The 99th percentile line shows that for 99% of the requests, it took less than around 11 seconds to complete the update.

Requests by type



The rate of requests fluctuates throughout the test, generally between 10 and 35 requests per second. There appears to be a gradual increase in the update rate over time.

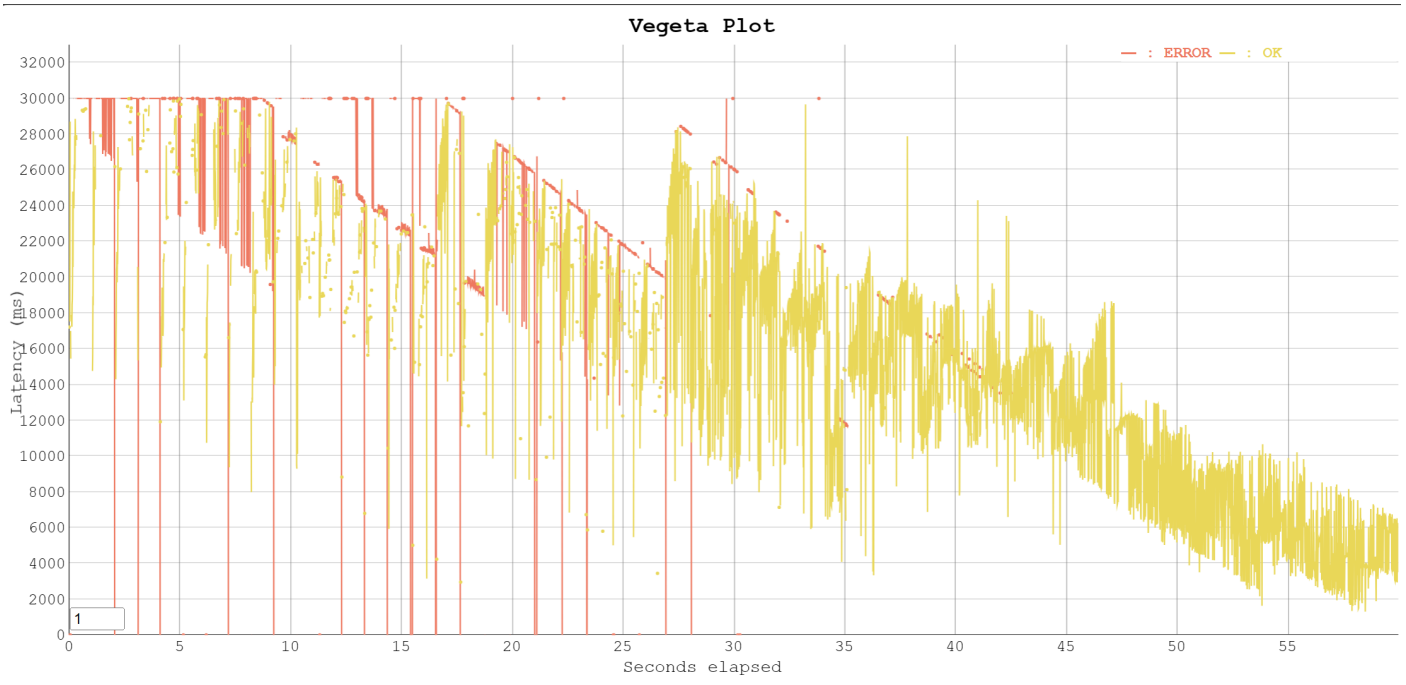
Instances



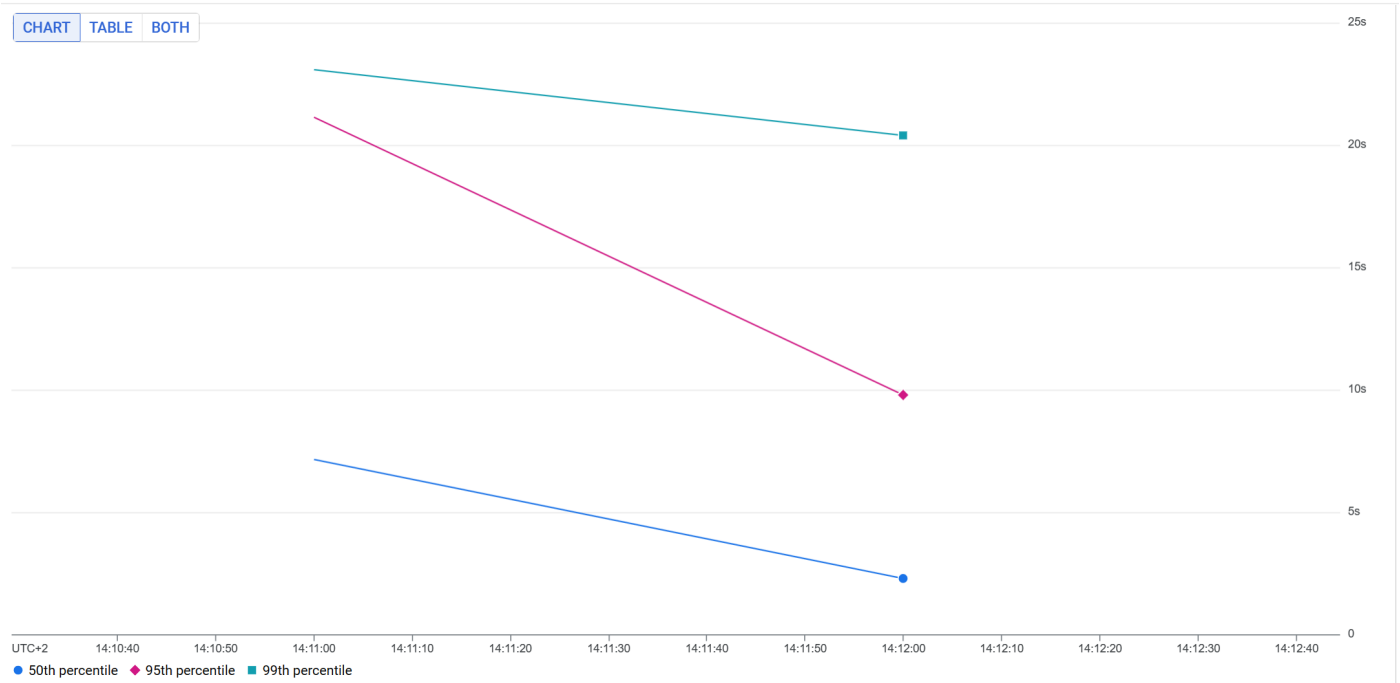
rate=100

```
echo "GET https://ambient-stone-420513.uc.r
.appspot.com/dswrite?_kind=book&author=John%20Steinbeck&title=The%20Grapes%20of%20Wrath&type
a attack -duration=60s -rate=100 | tee results.bin | vegeta report
```

Vegeta plot (latency)

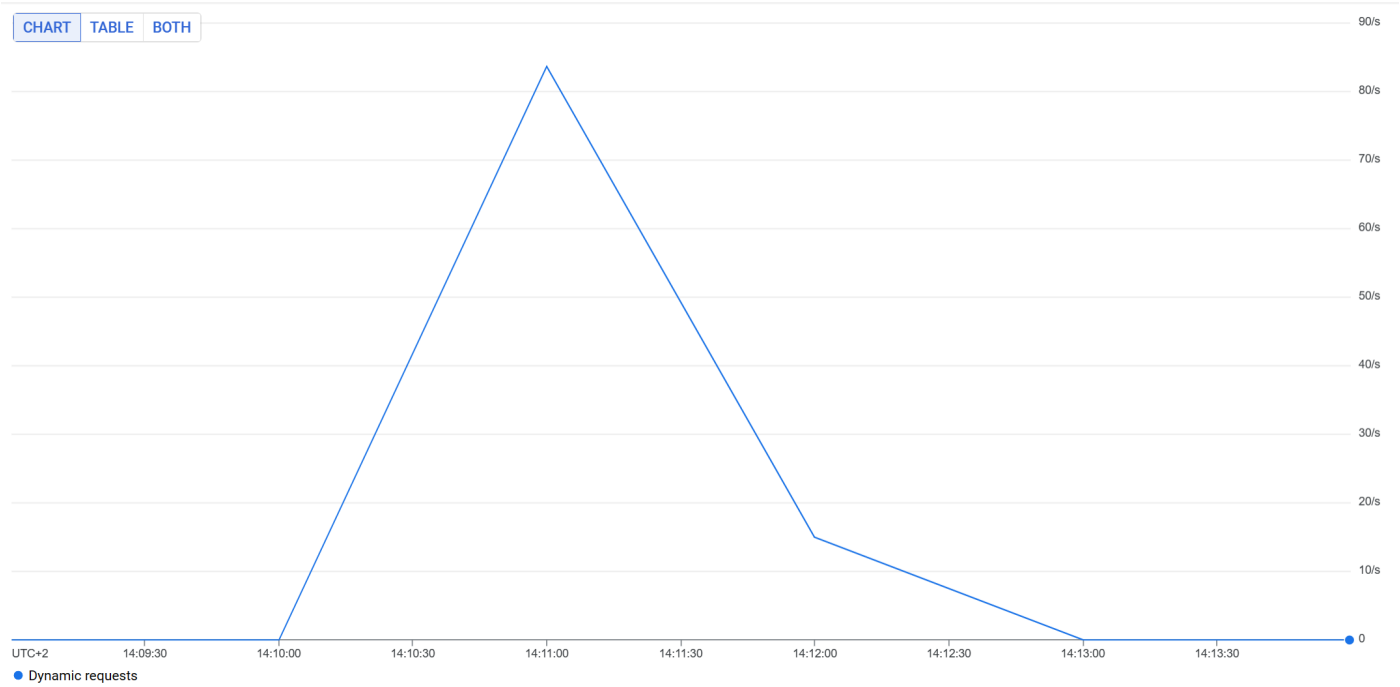


Latency



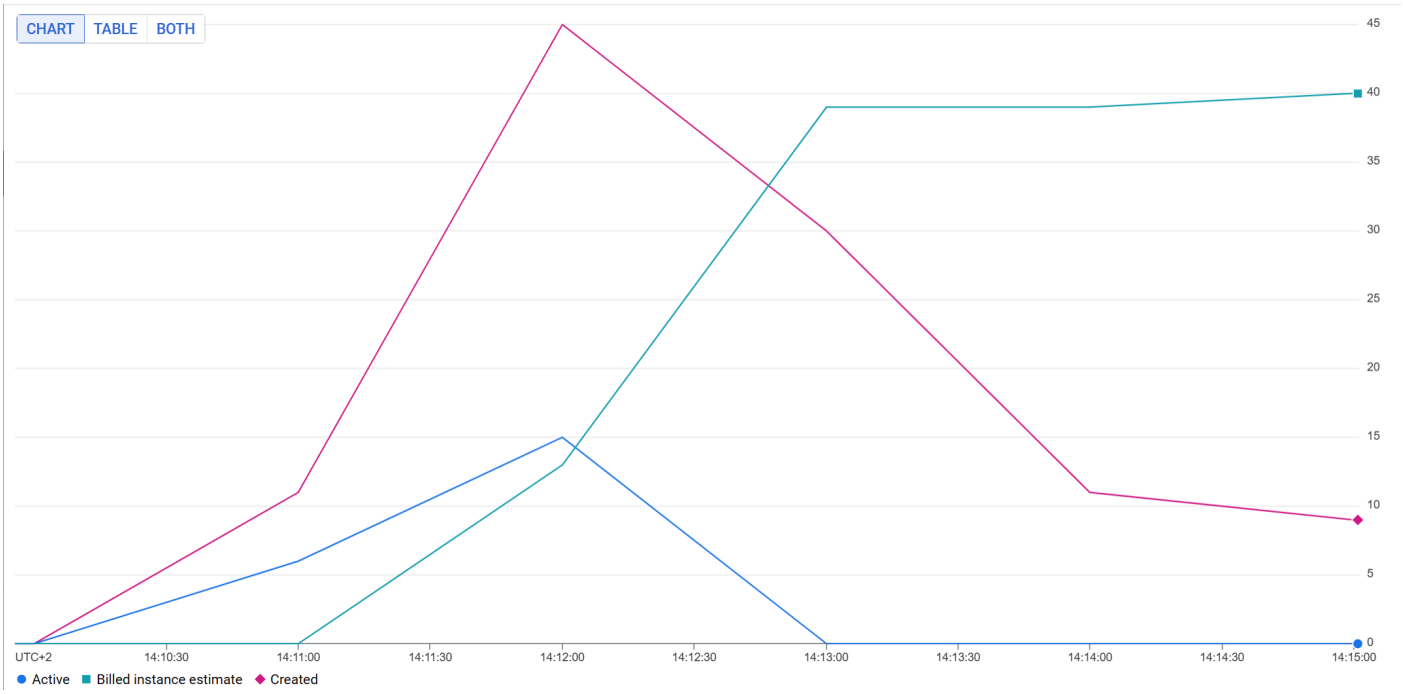
The 50th percentile line shows that for half of the requests, it took less than around 7 seconds to complete the update. The 95th percentile line shows that for 95% of the requests, it took less than around 22 seconds to complete the update. The 99th percentile line shows that for 99% of the requests, it took less than around 23 seconds to complete the update.

Requests by types



The rate of requests fluctuates throughout the test, generally between 15 and 85 requests per second. There appears to be a gradual increase in the update rate over time.

Instances



> What average response times do you observe in the test tool for each controller?

The average time is different in all cases we have done. For the attack with default rate, we have 400ms for the helloworld controller and 10000ms for the datastore controller.

> Compare the response times shown by the test tool and the App Engine console. Explain the difference.

Comparison and Explanation of Differences:

We carried out further experiments to confirm our findings, but these are not included in the report as we decided to keep the most relevant ones.

Cold Start Impact: Both graphs show the impact of cold starts. The Vegeta plot demonstrates high initial latency from the client's perspective, likely due to server resource initialization or starting new instances. The server graph reveals how quickly server performance improves after warming up.

Latency Perception: There's a difference in latency perception between the client and the server. The client experiences prolonged high latency, possibly due to cold starts and network-related delays. Server metrics may not fully capture these additional latencies but focus on processing times once requests are received.

Recovery Time: The server graph indicates faster recovery based on internal metrics, suggesting that server operations normalize quicker than client-side latency implies. Subsequent requests may be handled by already warmed-up instances.

Implications for Scaling and Performance: The Vegeta plot's initial high latency and errors emphasize the need to minimize cold start impacts. Strategies include keeping idle instances warm or optimizing instance startup times. The server graph shows robust performance once

operational, highlighting the effectiveness of the underlying infrastructure.

Metrics: Vegeta's plot shows the real response times for each request, while the App Engine plot breaks it down into percentiles, giving an idea of the thresholds that a certain percentage of requests meet. This helps understand how the system behaves under load, rather than just focusing on individual request times.

Data Capture and Representation: The spikiness of the Vegeta plot represent the sudden bursts of traffic. The smoothness of the App Engine plot is due to averaging over intervals to provide a clearer trendline, which is more useful for identifying improvements or degradations in performance over time.

The differences in the plots could largely be attributed to the different focuses and methodologies of the tools: one being more about real-time, raw response capture (Vegeta) and the other about trend analysis and performance thresholding over time (Google App Engine).

> Let’s suppose you become suspicious that the algorithm for the automatic scaling of instances is not working correctly. Imagine a way in which the algorithm could be broken. Which measures shown in the console would you use to detect this failure?

We would monitor **CPU utilization**, **Error rate**, **Latency**, **Instance count** and **incoming traffic**:

1. A sudden increase or decrease in the number of instances without any corresponding change in traffic or resource usage might indicate a problem.
2. If the latency of our application is consistently high or shows unexpected spikes could explains that the algorithm isn't provisioning enough instances to handle the traffic.
3. An high error rate, such as 5xx server errors, might suggest that instances are being terminated prematurely or not being allocated.
4. If the CPU utilization of our instances is consistently too high or too low indicate that the auto-scaling algorithm is not allocating resources based on the workload.