# GEL benchmark

The benchmark is executed by using K6 and the Loki add-on. This combo gives enough flexibility to test the write and read path for various scenario’s

* <https://k6.io/>
* <https://github.com/grafana/xk6-loki>

# Prerequisites

To run the K6 scripts you need to build a Docker image with all the dependencies.

**Dockerfile**

|  |
| --- |
| FROM golang:1.17  RUN go install go.k6.io/xk6/cmd/xk6@latest  RUN git clone <https://github.com/grafana/xk6-loki>  WORKDIR "xk6-loki"  RUN xk6 build \  --replace "[github.com/mingrammer/flog=github.com/chaudum/flog@v0.4.4-0.20211115125504-92153be038e6](mailto:github.com/mingrammer/flog=github.com/chaudum/flog@v0.4.4-0.20211115125504-92153be038e6)" \  --with "github.com/grafana/xk6-loki=$(pwd)"  ENTRYPOINT ["./k6"] |

**Docker build/push**

***Note: Make sure you change the repo accordingly, also in the benchmark manifest script!***

|  |
| --- |
| docker build -t <repo>/k6-loki  docker push <repo>/k6-loki |

# K6 script

The script contains several important sections that needs to be tweaked to generate the desired load. Below we highlight each of these sections.

## Write scenario’s

Below is the write scenario that gives you control over the number of executions of the write function during the various stages of the benchmark. This will increase the number of batches send to the cluster

In this setup we use a ramping-arrival-rate ([docs](https://k6.io/docs/using-k6/scenarios/executors/ramping-arrival-rate/)), that allows for increasing or decreasing the load during the benchmark.

This config will start with 1 iteration (startRate) of the write function (exec) every 5 sec (timeUnit). Dring the first 3 minutes of the benchmark it will slowly increase up to 5 iterations per 5 sec (stages). The last 2 minutes it will ramp up to 10 iterations per 5 sec.

|  |
| --- |
| write: {  exec: 'write', // function to execute  executor: 'ramping-arrival-rate', // <https://k6.io/docs/using-k6/scenarios/executors/ramping-arrival-rate>  startRate: 1, // Number of iterations to execute each timeUnit period at test start.  timeUnit: '5s', // Period of time to apply the startRate to the stages' target value. Its value is constant for the whole duration of the scenario, it is not possible to change it for a specific stage.  preAllocatedVUs: 1, // Number of VUs to pre-allocate before test start in order to preserve runtime resources.  maxVUs: 5, // Maximum number of VUs to allow during the test run.  stages: [  { duration: '3m', target: 5 }, // first 3 min, ramp up to 5 iteration per sec (rate)  { duration: '2m', target: 10 }, // next 2 min, ramp up to 10 iteration per sec (rate)  ],  }, |

Another way to control the load on the system is tweaking the write function itself. Within this function we create the actual log lines, create a batch, and send it to the cluster. The batch size is determined by picking a random size between the lower bound (800 \* KB) and upper bound (1 \* MB).

***Note: Loki clients tent to keep the batch size below 1MB, hence the 800KB – 1MB range.***

The log generator returns log lines as longs as the batch size is below the randomly picked size. Every time the read scenario executes the write function, it will send a batch of log lines of at least 800KB.

streams does not control the number of log lines inside the batch, it only controls the number of unique label sets that are within the batch.

The write function itself controls the batch size per write and the variety of label sets within that batch.

|  |
| --- |
| export function write() {  let streams = randomInt(4, 8);  // Pick random size between min bound and max bound  // Generate logs till size is above the random picked size.  let res = write\_client.pushParameterized(streams, 800 \* KB, 1 \* MB);  check(res,  {  'successful write': (res) => {  let success = res.status === 204;  if (!success) console.log(res.status, res.body);  return success;  },  }  );  } |

## Current load

This example config is generating roughly:

* 210 lines per second
* 32 kB/s (83GB/month)

If you need to generate more load, the best way is to increase the number of executions by tweaking startRate and/or timeUnit. When increasing the load keep an eye on the number of VUs used during the benchmark. VU’s is the unit used within K6 to execute all the workload, see it as threads. If at one point the rate is too high for a single VU, you need to increase the number of VU’s (preAllocatedVUs and maxVUs).

***Warning: VU is also a label on the logs, so this will increase the cardinality of the logs (number of unique series).***

## Cardinality

The script creates series by using labels and random values for these labels. The number of labels and values can be controlled and will directly impact the number of series generated during the benchmark.

More info on these labels: <https://github.com/grafana/xk6-loki#labels>

By default, it will use 3 labels and several values:

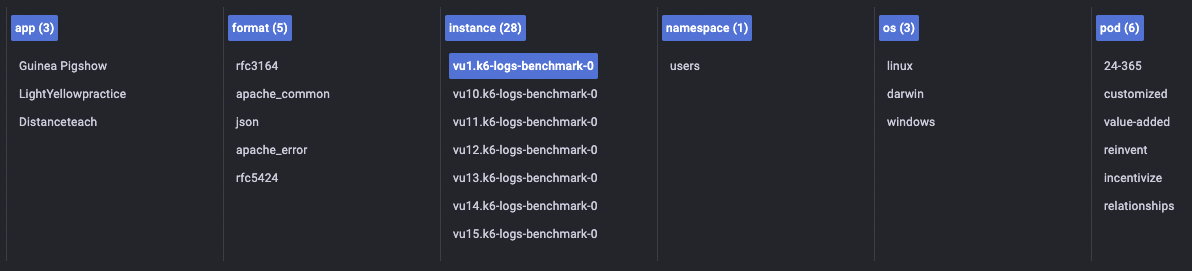
|  |  |
| --- | --- |
| **Label** | **Value(s)** |
| instance | 1 per k6 worker. VU id + hostname |
| format | apache\_common, apache\_combined, apache\_error, rfc3164, rfc5424, json |
| os | darwin, linux, windows |

In the test we have add 3 more labels, with a total of 10 values.

|  |
| --- |
| const labelCardinality = {  "app": 3,  "namespace": 1,  "pod": 6,  }; |

If the number of VU’s (maxVUs) is 5, the cardinality (number of streams) will be the following:

**5** Instance **\* 6** format **\* 3** os **\* 3** app **\* 1** instance **\* 6** pod **= 1620**

*Fig. Example of the labels and possible values*

## Read scenario’s

Below is the read scenario that gives you control over the number of executions of the read function during the various stages of the benchmark.

In this setup we use a ramping-arrival-rate ([docs](https://k6.io/docs/using-k6/scenarios/executors/ramping-arrival-rate/)), that can increase or decrease the load during the benchmark.

This config will start with 5 iterations (startRate) of the read function (exec) every 10 sec (timeUnit). This will slowly increase up to 15 iterations per 10 sec as during the first 5 minutes (stages). You can add as many stages you want to control the ramp up and ramp down.

|  |
| --- |
| read: {  exec: 'read', // function to execute  executor: 'ramping-arrival-rate', // <https://k6.io/docs/using-k6/scenarios/executors/ramping-arrival-rate>  startRate: 5, // Number of iterations to execute each timeUnit period at test start.  timeUnit: '10s', // Period of time to apply the startRate to the stages' target value. Its value is constant for the whole duration of the scenario, it is not possible to change it for a specific stage.  preAllocatedVUs: 50, // Number of VUs to pre-allocate before test start in order to preserve runtime resources.  maxVUs: 100, // Maximum number of VUs to allow during the test run.  stages: [  { duration: '5m', target: 15 }, // first 5 min, ramp up to 15 iteration per sec (rate)  ],  }, |

Another way to control the load on the system is tweaking the read function itself. Within this function we create the actual query to execute on the cluster.

The type of query and time range within the query are both selected randomly based on a list and ratio.

***Note: the sum of the ratios within a single list should be 1***

|  |  |
| --- | --- |
| const queryTypeRatioConfig = [  {  ratio: 0.1,  item: readLabels  },  {  ratio: 0.1,  item: readLabelValues  },  {  ratio: 0.1,  item: readSeries  },  {  ratio: 0.5,  item: readRange  },  {  ratio: 0.2,  item: readInstant  },  ]; | const rangesRatioConfig = [  {  ratio: 0.2,  item: '15m'  },  {  ratio: 0.2,  item: '30m'  },  {  ratio: 0.3,  item: '1h'  },  {  ratio: 0.2,  item: '3h'  },  {  ratio: 0.1,  item: '12h'  },  ]; |

The actual queries are listed in the following functions of the script and can be changed accordingly:

* readLabels()
* readLabelValues()
* ReadSeries()
* rangeQuerySuppliers()
* instantQuerySuppliers()

## Current load

This example config is generating roughly:

* 15 queries per sec

If you need to generate more load, the best way is to increase the number of executions by tweaking startRate and/or timeUnit. When increasing the load keep an eye on the number of VUs used during the benchmark. VU’s is the unit used within K6 to execute all the workload, see it as threads. If at one point the rate is too high for a single VU, you need to increase the number of VU’s (preAllocatedVUs and maxVUs).

Queries can take some time to execute (seconds +). Therefor a much higher number of VU’s is needed for the read scenario. In case queries execute on average around 3 seconds, you need at least 6 VU’s to reach 2 queries per second. From a certain number of VUs it might be beneficial to create an additional pod replica.

# Run the benchmark

The benchmark will run as a Statefulset. The reason for this is that the instance label contains the host name of the pod. To prevent 1620 new series every time we restart the benchmark, we choose for a Statefulset, which will always give us the same pod name (k6-logs-benchmark-0) and there for the same value for the instance label (i.e. vu1.k6-logs-benchmark-0).

**Commands**

|  |
| --- |
| kubectl -n fo-monitoring-gel apply -f k6-logs-bench-k8s.yaml  kubectl -n fo-monitoring-gel delete -f k6-logs-bench-k8s.yaml |

**Output**