# GEM benchmark

The benchmark is executed by using K6 and the Prometheus 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-client-prometheus-remote>

# 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 \  --with "github.com/grafana/xk6-client-prometheus-remote@latest"  RUN mkdir /modules  RUN curl -o /modules/httpx.js <https://jslib.k6.io/httpx/0.0.6/index.js>  RUN curl -o /modules/k6-utils.js <https://jslib.k6.io/k6-utils/1.1.0/index.js>  RUN curl -o /modules/k6chaijs.js <https://jslib.k6.io/k6chaijs/4.3.4.1/index.js>  ENTRYPOINT ["./k6"] |

**Docker build/push**

***Note: Make sure you change the repo accordingly***

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

# 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.

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

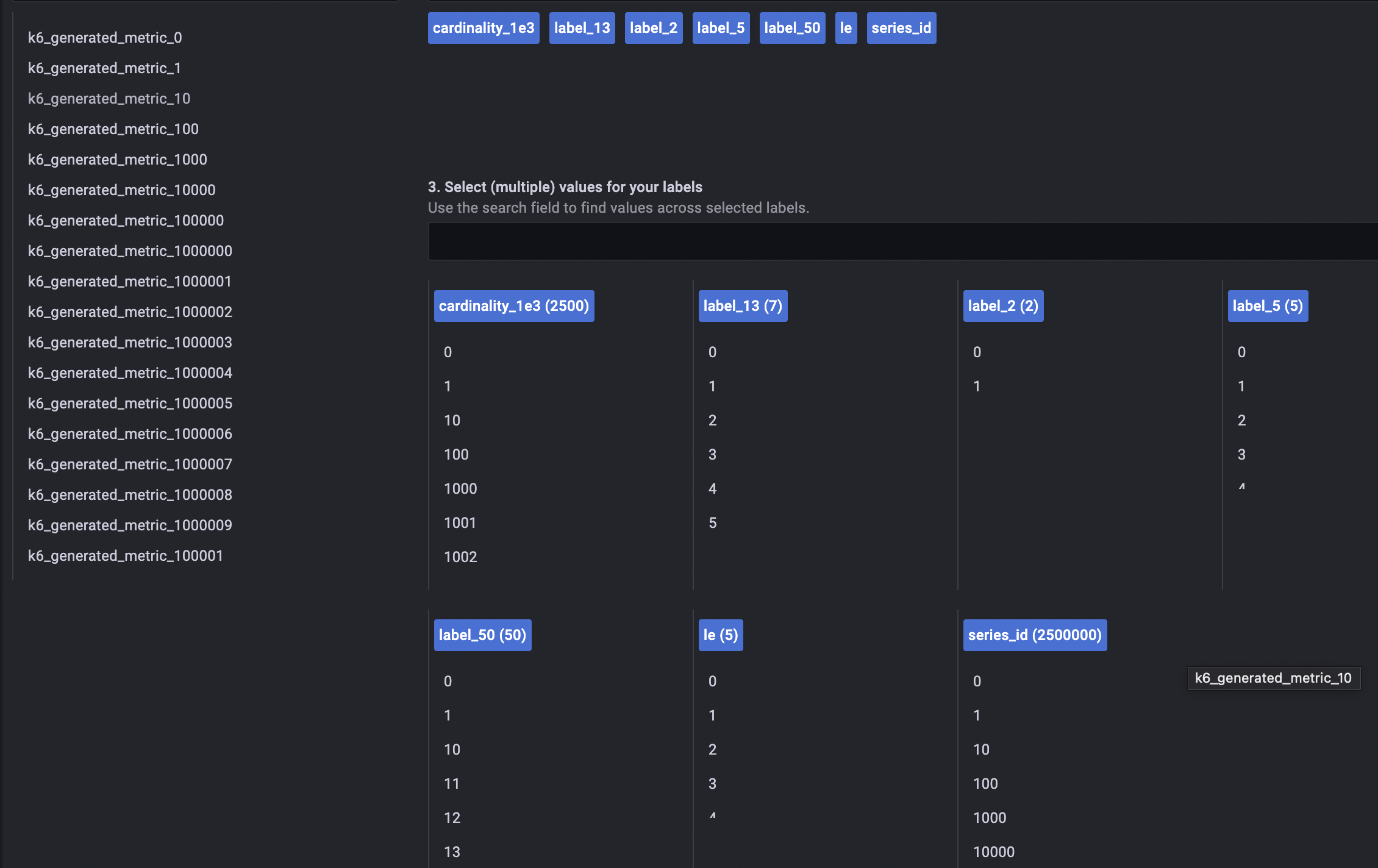
This config will start with 80 iterations (startRate) of the write function (exec) every 1 sec (timeUnit) for the first 5 minutes. This will slowly increase up to 100 iterations per 1 sec as configured (stages) during the next 25 minutes.

|  |
| --- |
| const MIN\_WRITE\_REQUEST\_RATE = 80;  const MAX\_WRITE\_REQUEST\_RATE = 100;  const WRITE\_SERIES\_PER\_REQUEST = 25000;  const WRITE\_VUS = 20;  const WRITE\_STAGES = [  {  target: (MIN\_WRITE\_REQUEST\_RATE), duration: `5m`,  }, {  target: (MAX\_WRITE\_REQUEST\_RATE), duration: `25m`,  }  ]  writing\_metrics: {  executor: 'ramping-arrival-rate', // <https://k6.io/docs/using-k6/scenarios/executors/ramping-arrival-rate>  startRate: MIN\_WRITE\_REQUEST\_RATE, // Number of iterations to execute each timeUnit period at test start.  timeUnit: `${SCRAPE\_INTERVAL\_SECONDS}s`, // 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: WRITE\_VUS, // Number of VUs to pre-allocate before test start in order to preserve runtime resources.  exec: 'write', // function to execute  stages: WRITE\_STAGES,  }, |

Another way to control the load on the system is by tweaking the WRITE\_SERIES\_PER\_REQUEST or WRITE\_TEMPLATE. The first one controls the number of series written per iteration and the toal number of unique series (25000 means 2.500.000 active series). The second will control the label and values for the metrics.

Make sure you assign enough VUs (preAllocatedVUs) to handle to configured load.

|  |
| --- |
| const WRITE\_TEMPLATE = remote.precompileLabelTemplates({  \_\_name\_\_: 'k6\_generated\_metric\_${series\_id}', // Name of the series.  series\_id: '${series\_id}', // Each value of this label will match 1 series.  cardinality\_1e3: '${series\_id/1000}', // Each value of this label will match 1000 series.  label\_50: '${series\_id%50}', // There are 50 possible values of this label  label\_13: '${series\_id%7}', // There are 7 possible values of this label  label\_5: '${series\_id%5}', // There are 5 possible values of this label  label\_2: '${series\_id%2}', // There are 2 possible values of this label  le: '${series\_id%5}', // The label 'le' is required for histogram\_quantile queries  }); |



*Fig. Example of the labels and possible values*

## Current load

This example config is generating:

* 2.500.000 active series (before replication)

If you need to generate more load, the best way is to increase the number of executions by tweaking startRate and/or timeUnit or the WRITE\_SERIES\_PER\_REQUEST. 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).

## Read scenario’s

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

In this setup we use a constant-arrival-rate ([docs](https://k6.io/docs/using-k6/scenarios/executors/constant-arrival-rate/)), that allows us to increase or decrease the load for the benchmark.

This config has a fixed (rate) number of 10 range queries and 30 instant queries per second for a fixed duration (duration).

|  |
| --- |
| let TOTAL\_DURATION\_MIN = 0;  for (let stage of WRITE\_STAGES) {  TOTAL\_DURATION\_MIN += parseInt(stage.duration);  }  TOTAL\_DURATION\_MIN;  const READ\_RANGE\_VU = 10;  const READ\_INSTANT\_VU = 10;  reads\_range\_queries: {  executor: 'constant-arrival-rate', // <https://k6.io/docs/using-k6/scenarios/executors/constant-arrival-rate/>  rate: 10, // Number of iterations to start during each timeUnit period.  timeUnit: '1s', // Period of time to apply the rate value.  duration: `${TOTAL\_DURATION\_MIN}m`, // Total scenario duration (excluding gracefulStop).  preAllocatedVUs: READ\_RANGE\_VU, // Number of VUs to pre-allocate before test start to preserve runtime resources.  exec: 'run\_range\_query' // function to execute  },  reads\_instant\_queries: {  executor: 'constant-arrival-rate',  rate: 30,  timeUnit: '1s',  duration: `${TOTAL\_DURATION\_MIN}m`,  preAllocatedVUs: READ\_INSTANT\_VU,  exec: 'run\_instant\_query\_cardinality'  }, |

To control the type of queries we need to tweak the ratios of the queries, ranges, and intervals within the script. Each iteration will select a random query, range and interval based on the ratio. The first number is the ratio, 50 meaning 50%. The total of the list must be exactly 100%.

|  |
| --- |
| const query\_distribution = {  25: 'sum by (label\_50) (irate($metric[$rate\_interval]))',  22: 'sum($metric)',  23: 'histogram\_quantile(0.9, sum by (le, label\_50) (rate($metric[$rate\_interval])))',  20: 'sum(rate($metric[$rate\_interval]))',  10: 'avg($metric)'  }  time\_range: {  50: 1 \* HOUR\_SECONDS,  30: 0.5 \* HOUR\_SECONDS,  20: 2 \* HOUR\_SECONDS,  },  rate\_interval: {  1: '1h',  9: '10m',  20: '5m',  70: '1m',  }, |

## Current load

This example config is generating roughly:

* 10 range queries per sec
* 30 isntant 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

**Commands**

|  |
| --- |
| kubectl -n fo-monitoring-gem apply -f k6-metrics-bench-k8s.yaml  kubectl -n fo-monitoring-gem delete -f k6-metrics-bench-k8s.yaml |

**Thresholds**

The benchmark is configured with 4 thresholds. If during the test these thresholds are exceeded the test will stop and marked as failed. These are the current checks and thresholds, the measured value for each of the thresholds is shown in the output after the benchmark.

|  |
| --- |
| thresholds: {  // SLA: 99.9% of writes succeed.  'checks{type:write}': ['rate > 0.999'],  // 99.9% of writes take less than 10s (SLA has no guarantee on write latency).  [`http\_req\_duration{url:${remote\_write\_url}}`]: ['p(99.9) < 10000'],  // SLA: 99.9% of queries succeed.  'checks{type:read}': ['rate > 0.999'],  // SLA: average query time for any 3 hours of data is less than 2s (not including Internet latency).  'http\_req\_duration{type:read}': ['avg < 2000'],  }, |

**Output**

The output provides useful information about the success and failures rates and the response times.

|  |
| --- |
| **✓ write worked**    **█ range query**    **✓ range read success**  **✓ has valid json body**  **✓ expected status field is 'success' to equal 'success'**  **✓ expected resultType is 'matrix' to equal 'matrix'**    **█ instant query**    **✓ instant read success**  **✓ has valid json body**  **✓ expected status field to equal 'success'**  **✓ expected data.resultType field to equal 'vector'**    **checks.........................: 100.00% ✓ 19915 ✗ 0**  **✓ { type:read }................: 100.00% ✓ 4799 ✗ 0**  **✓ { type:write }...............: 100.00% ✓ 719 ✗ 0**  **data\_received..................: 29 MB 238 kB/s**  **data\_sent......................: 400 MB 3.3 MB/s**  **dropped\_iterations.............: 1 0.008323/s**  **group\_duration.................: avg=29.4ms min=15.97ms med=23.59ms max=355.78ms p(90)=48.23ms p(95)=56.6ms**  **http\_req\_blocked...............: avg=22.09µs min=2.4µs med=3.9µs max=13.77ms p(90)=6.1µs p(95)=9.7µs**  **http\_req\_connecting............: avg=10.76µs min=0s med=0s max=2.59ms p(90)=0s p(95)=0s**  **http\_req\_duration..............: avg=52.98ms min=11.77ms med=19.95ms max=610.6ms p(90)=232.02ms p(95)=253.25ms**  **{ expected\_response:true }...: avg=52.98ms min=11.77ms med=19.95ms max=610.6ms p(90)=232.02ms p(95)=253.25ms**  **✓ { type:read }................: avg=23.6ms min=11.77ms med=18.56ms max=348ms p(90)=38.1ms p(95)=44.8ms**  **http\_req\_failed................: 0.00% ✓ 0 ✗ 5518**  **http\_req\_receiving.............: avg=282.51µs min=20.29µs med=76.3µs max=16.11ms p(90)=233.15µs p(95)=1.88ms**  **http\_req\_sending...............: avg=1.25ms min=13.4µs med=38.8µs max=110.01ms p(90)=4.5ms p(95)=10.48ms**  **http\_req\_tls\_handshaking.......: avg=0s min=0s med=0s max=0s p(90)=0s p(95)=0s**  **http\_req\_waiting...............: avg=51.45ms min=11.65ms med=19.82ms max=608.79ms p(90)=223.03ms p(95)=243.54ms**  **http\_reqs......................: 5518 45.925492/s**  **iteration\_duration.............: avg=62.01ms min=16.17ms med=25.36ms max=644.19ms p(90)=260.15ms p(95)=281.46ms**  **iterations.....................: 5518 45.925492/s**  **vus............................: 40 min=40 max=40**  **vus\_max........................: 40 min=40 max=40** |