Grafana Labs

k6

Supercharging Performance Testing: Bridging the Gap Between Backend and Frontend with k6

Ayush Goyal
Senior Software Engineer @ Grafana Labs

Agenda

How we'll structure our time

- What is Performance Testing?
- Introduction to Protocol-Based Load Testing
- Frontend and browser testing
- Bridging Backend and Frontend testing - Hybrid testing
- Exploring k6: Features and Capabilities
- Benefits of running hybrid tests in Grafana K6 cloud

Grafana Labs

What is Performance Testing?

It measures qualitative aspects of a user's experience of a system, such as its responsiveness and reliability.

Why should we do performance testing?

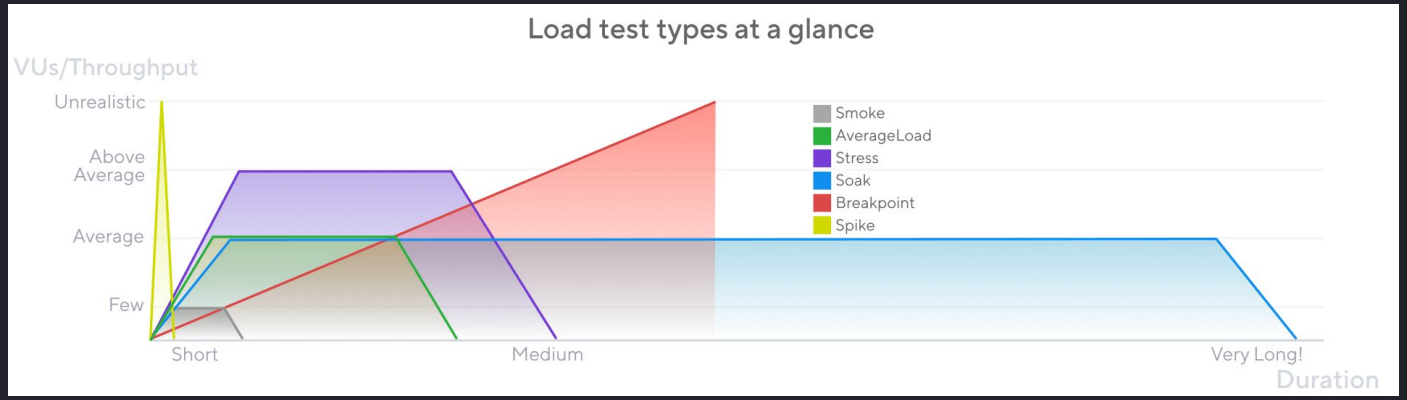
- Improve user experience.
- Prepare for unexpected demand.
- Increase confidence in the application.
- Assess and optimize infrastructure.

What is Load testing?

Load testing is the process of putting demand on a system and measuring its response

Different types of Load testing

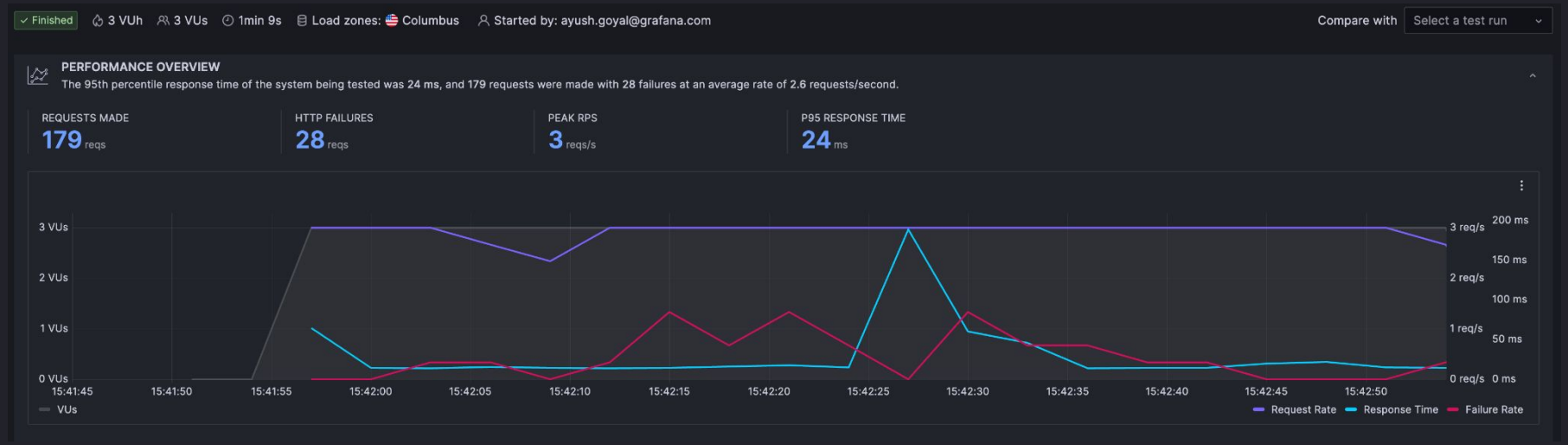
1. Smoke Testing
2. “Average” load test
3. Stress Testing
4. Soak Testing
5. Spike Testing
6. Breakpoint test



1. Smoke Testing

```
import http from 'k6/http';
import { check, sleep } from 'k6';

export const options = {
  vus: 3, // Key for Smoke test. Keep it at 2, 3, max 5 VUs
  duration: '1m', // This can be shorter or just a few iterations
};

export default () => {
  const urlRes = http.get('https://test-api.k6.io');
  sleep(1);
  // MORE STEPS
  // Here you can have more steps or complex script
  // Step1
  // Step2
  // etc.
};
```

- Minimal Load (around 5 VUs or less)
- Short duration (few seconds to couple of minutes)
- Verify test script does not have any errors
- Verify system-under-test is properly operational
- Run it first after any change to test script or application


Finished
3 VUh
3 VUs
1min 9s
Load zones: Columbus
Started by: ayush.goyal@grafana.com
Compare with
Select a test run
PERFORMANCE OVERVIEW
The 95th percentile response time of the system being tested was 24 ms, and 179 requests were made with 28 failures at an average rate of 2.6 requests/second.
REQUESTS MADE
179 reqs
HTTP FAILURES
28 reqs
PEAK RPS
3 reqs/s
P95 RESPONSE TIME
24 ms
3 VUs
2 VUs
1 VUs
0 VUs
3 req/s
2 req/s
1 req/s
0 req/s
200 ms
150 ms
100 ms
50 ms
0 ms
15:41:45
15:41:50
15:41:55
15:42:00
15:42:05
15:42:10
15:42:15
15:42:20
15:42:25
15:42:30
15:42:35
15:42:40
15:42:45
15:42:50
VUs
Request Rate
Response Time
Failure Rate

2. “Average” load test

```
import http from 'k6/http';
import { sleep } from 'k6';

export const options = {
  // Key configurations for avg load test in this section
  stages: [
    { duration: '5m', target: 100 }, // traffic ramp-up from 1 to 100 users over 5 minut
    { duration: '30m', target: 100 }, // stay at 100 users for 30 minutes
    { duration: '5m', target: 0 }, // ramp-down to 0 users
  ],
};

export default () => {
  const urlRes = http.get('https://test-api.k6.io');
  sleep(1);
  // MORE STEPS
  // Here you can have more steps or complex script
  // Step1
  // Step2
  // etc.
};
```

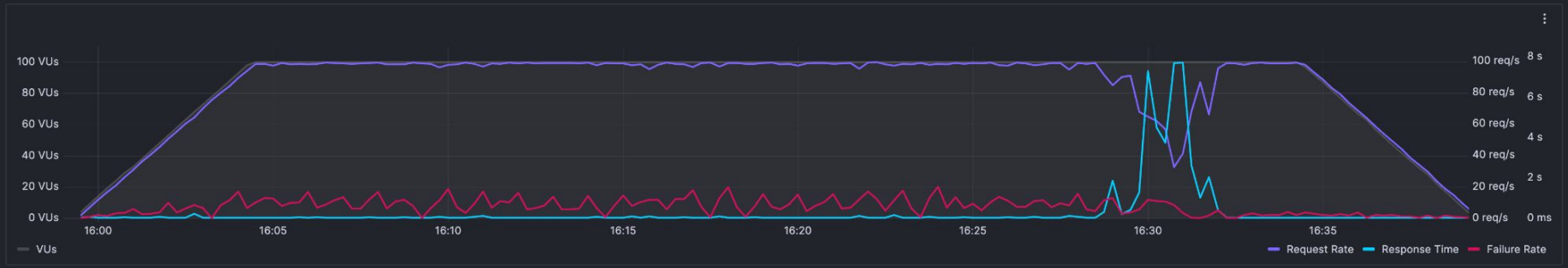
Collapse code

- Assess how the system behaves under expected normal conditions.
- Typically increases the throughput or VUs gradually and keeps that average load for some time.
- Also called day-in-life test or volume test

PERFORMANCE OVERVIEW

The 95th percentile response time of the system being tested was **57 ms**, and **200 287** requests were made with **17 442** failures at an average rate of **83** requests/second.

REQUESTS MADE	HTTP FAILURES	PEAK RPS	P95 RESPONSE TIME
200.3K reqs	**17.4K** reqs	**98.93** reqs/s	**57** ms



3. Stress testing

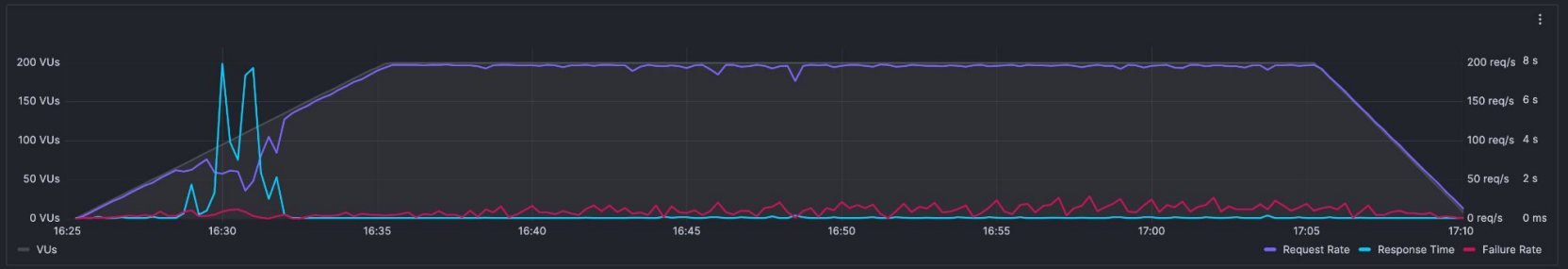
```
import http from 'k6/http';
import { sleep } from 'k6';

export const options = {
  // Key configurations for Stress in this section
  stages: [
    { duration: '10m', target: 200 }, // traffic ramp-up from 1 t
    { duration: '30m', target: 200 }, // stay at higher 200 users
    { duration: '5m', target: 0 }, // ramp-down to 0 users
  ],
};

export default () => {
  const urlRes = http.get('https://test-api.k6.io');
  sleep(1);
  // MORE STEPS
  // Here you can have more steps or complex script
  // Step1
  // Step2
  // etc.
};
```

- Assesses how the system performs when loads are heavier than usual.
- Main difference from the Average Load Test is the higher load.
- **Load should be higher than what the system experiences on average.**
- Ideal to run only after successful average load test using the same script

PERFORMANCE OVERVIEW

The 95th percentile response time of the system being tested was **42 ms**, and **435 196** requests were made with **25 659** failures at an average rate of **161** requests/second.

REQUESTS MADE	HTTP FAILURES	PEAK RPS	P95 RESPONSE TIME
435.2K reqs	**25.7K** reqs	**197.67** reqs/s	**42** ms



SCENARIOS (0) HIDE SHOW

No scenario was configured in this test. Check out how to create one

4. Soak Testing

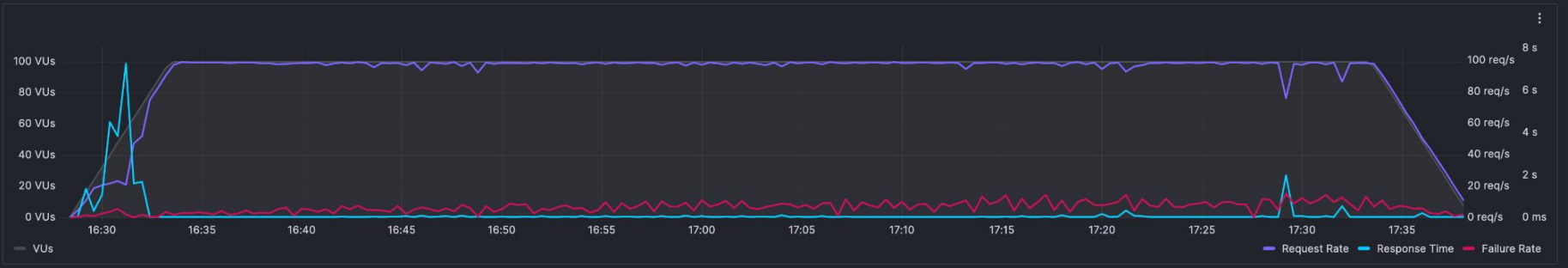
```
import http from 'k6/http';
import { sleep } from 'k6';

export const options = {
  // Key configurations for Soak test in this section
  stages: [
    { duration: '5m', target: 100 }, // traffic ramp-up from 1 to
    { duration: '8h', target: 100 }, // stay at 100 users for 8 ho
    { duration: '5m', target: 0 }, // ramp-down to 0 users
  ],
};

export default () => {
  const urlRes = http.get('https://test-api.k6.io');
  sleep(1);
  // MORE STEPS
  // Here you can have more steps or complex script
  // Step1
  // Step2
  // etc.
};
```

- It is another variation of the Average-Load test which focuses on extended periods.
- The peak load duration extends several hours or even days.
- Should be executed after successfully running smoke and average load test

PERFORMANCE OVERVIEW

The 95th percentile response time of the system being tested was **31 ms**, and **378 846** requests were made with **27 403** failures at an average rate of **90** requests/second.

REQUESTS MADE	HTTP FAILURES	PEAK RPS	P95 RESPONSE TIME
378.8K reqs	27.4K reqs	98.83 reqs/s	31 ms



SCENARIOS (0) HIDE SHOW

No scenario was configured in this test. Check out how to create one

4. Spike Testing

```
import http from 'k6/http';
import { sleep } from 'k6';

export const options = {
  // Key configurations for spike in this section
  stages: [
    { duration: '2m', target: 2000 }, // fast ramp-up to a high point
    // No plateau
    { duration: '1m', target: 0 }, // quick ramp-down to 0 users
  ],
};

export default () => {
  const urlRes = http.get('https://test-api.k6.io');
  sleep(1);
  // MORE STEPS
  // Add only the processes that will be on high demand
  // Step1
  // Step2
  // etc.
};
```

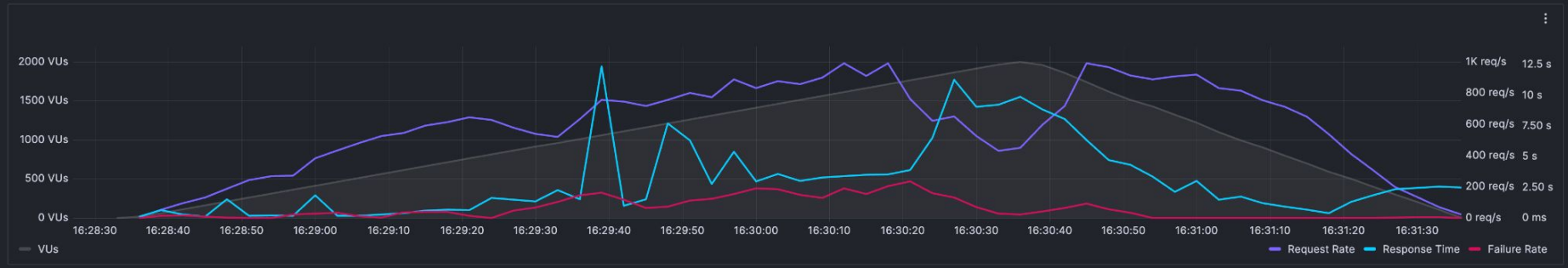
- It verifies system can survive sudden and massive traffic.
- Extremely high loads in a shot interval of time.
- Generally recommended to execute when the system expects to receive a sudden rush of activity.

REQUESTS MADE
107.8K reqs

HTTP FAILURES
10.7K reqs

PEAK RPS
992 reqs/s

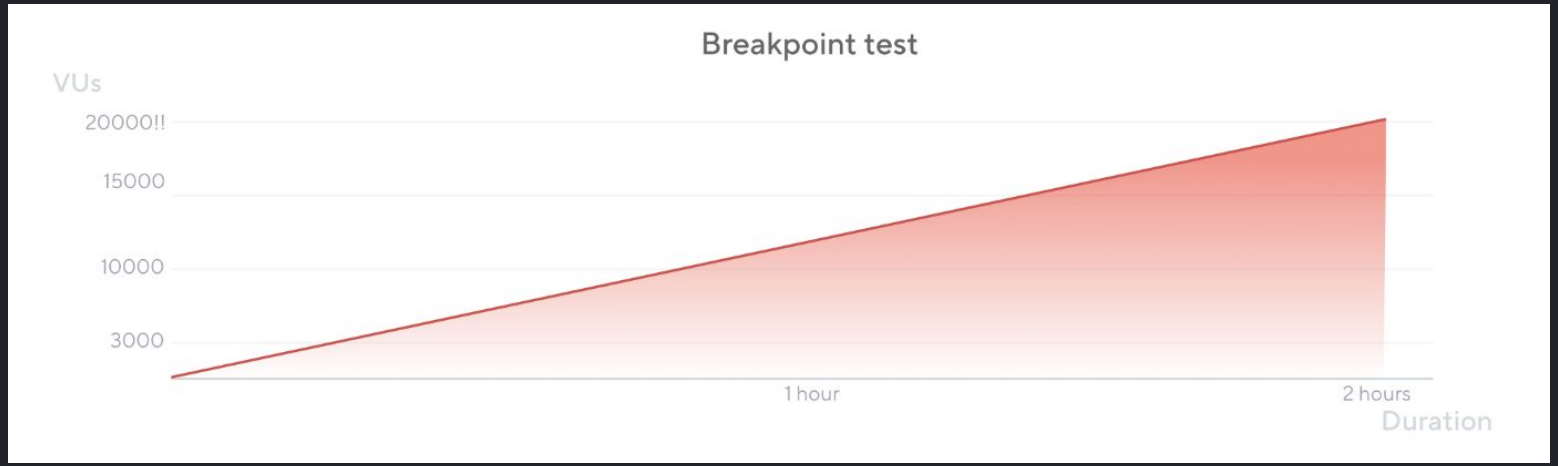
P95 RESPONSE TIME
4 047 ms



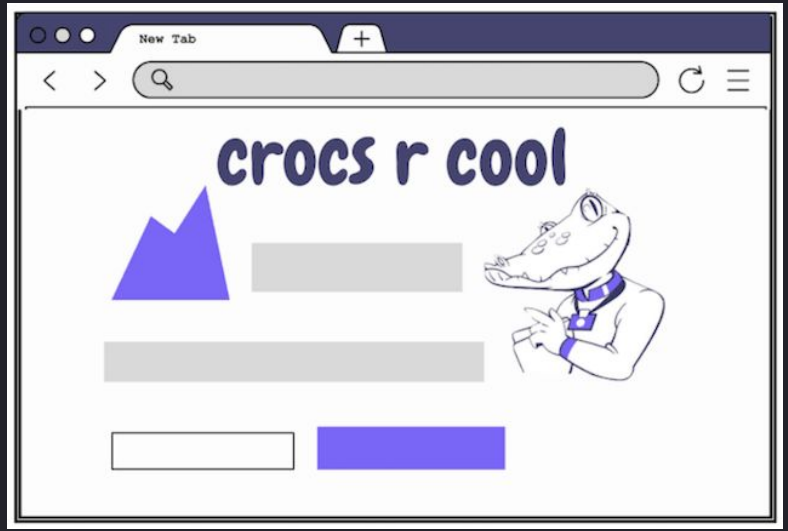
5. Breakpoint testing

```
import http from 'k6/http';
import { sleep } from 'k6';

export const options = {
  // Key configurations for breakpoint in this section
  executor: 'ramping-arrival-rate', //Assure load increase if t
  stages: [
    { duration: '2h', target: 20000 }, // just slowly ramp-up t
  ],
};

export default () => {
  const urlRes = http.get('https://test-api.k6.io');
  sleep(1);
  // MORE STEPS
  // Here you can have more steps or complex script
  // Step1
  // Step2
  // etc.
};
```

- The aim of this test is to find the system limits
- Gradually increase load to identify the capacity limits of the system.
- Recommended to run only after system is known to be functioning under all other load types.
- **Avoid breakpoint tests in elastic cloud environments.**


Breakpoint test
VUs
20000!!
15000
10000
3000
1 hour
2 hours
Duration

Frontend Testing



k6

What is Frontend testing?

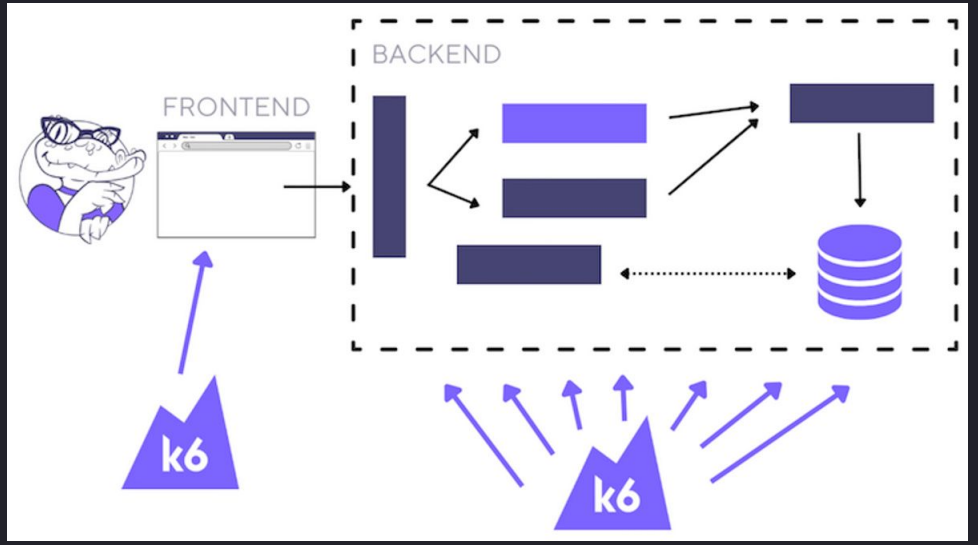
- Verifies application performance on the interface level.
- Concerned with the end-user experience of an application, usually involving a browser.
- Primarily measures a single user's experience of the system
- It has metrics that are distinct from backend performance testing like core web vitals - LCP (Largest Contentful Paint), CLS (Cumulative Layout Shift)

Browser testing with K6

Browser testing with K6

- Allows automating browser actions for end-to-end web testing
- Collects frontend performance metrics as part of your existing k6 tests
- Browser level API has rough compatibility with Playwright for easier migration.
- Easy to mix browser-level scripts with existing protocol-level scripts to implement a hybrid approach to performance testing.

Hybrid Testing



k6

Issues with only doing:

Backend Testing

- Focuses solely on backend without any regard to user experience.
- Difficult to add and test complex user flows
- Difficult to maintain as the usage grows

vs

Frontend Testing

- Expensive and resource intensive to create high amount of load.
- Tests only for handful of users, hence does not tell how UI will behave under extreme stress.

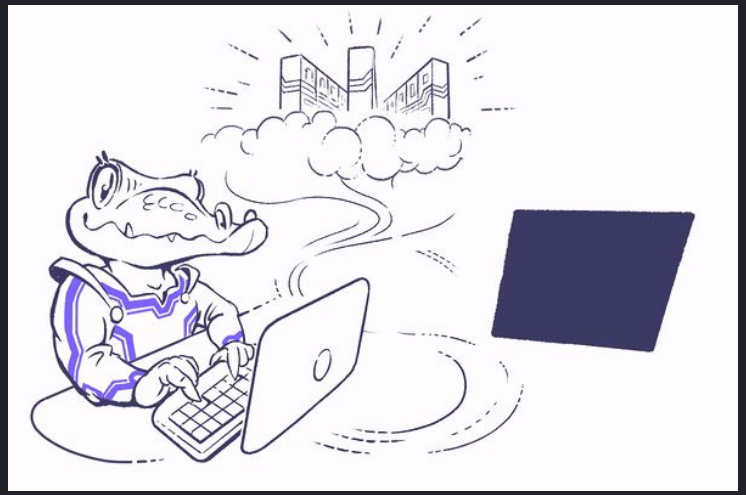
Benefits of Hybrid testing

Enhance user experience by monitoring browser performance metrics alongside existing protocol metrics.

Identify blind spots and errors with browser-based performance testing that can uncover browser-specific issues missed by protocol-level testing.

Facilitate cross-team collaboration by enabling developers, test automation engineers, and SDETs to utilize a shared tool for performance testing.

What is K6?



Tests as Code - Programmable

```
import { userFlowA, userFlowB } from './my-lib';

export default function () {
  userFlowA();
  userFlowB();
}
```

Portable open source load testing tool

Same k6 test script for [multiple execution modes]

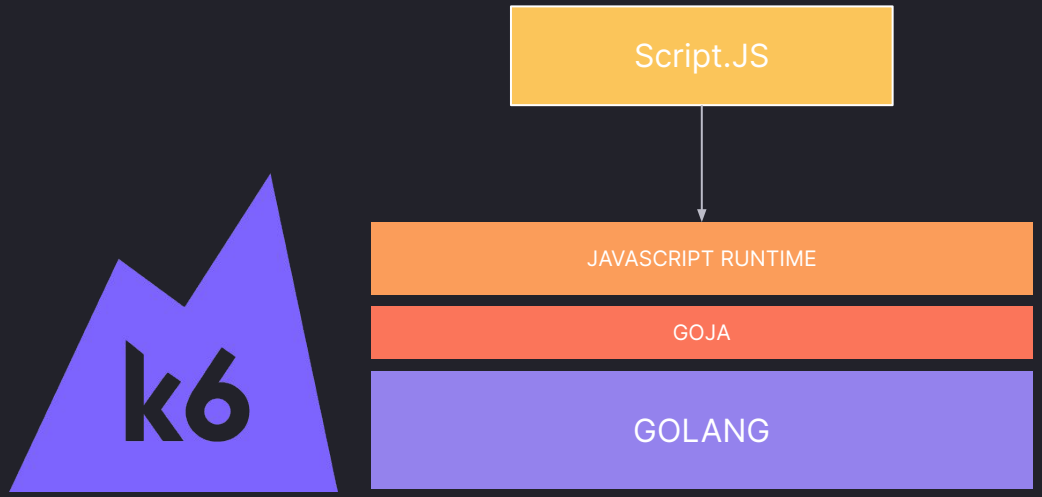
```
>_  k6 run script.js          LOCAL
```

```
>_  kubectl apply -f k6.yaml  K8S
```

```
>_  k6 cloud script.js        CLOUD
```

Performant

No NodeJS



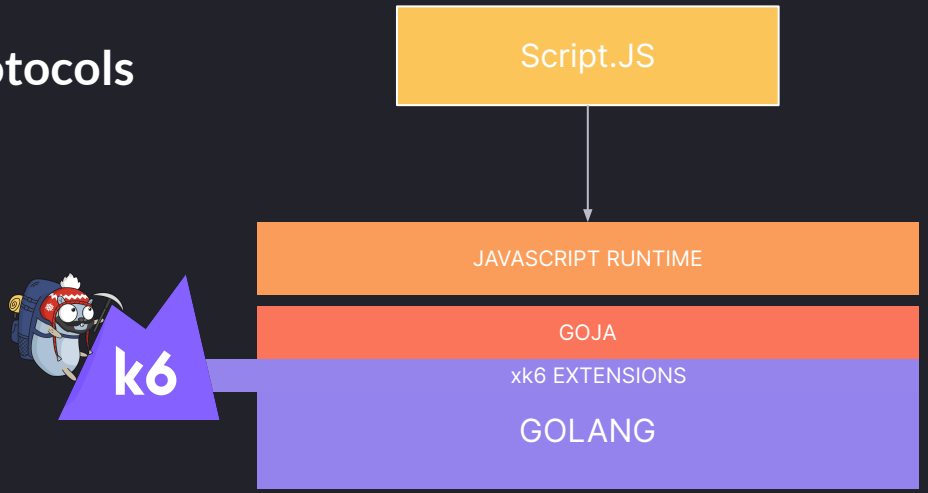
Extensible

k6.io/docs/extensions/getting-started/explore/

Other storage options

Other testing cases

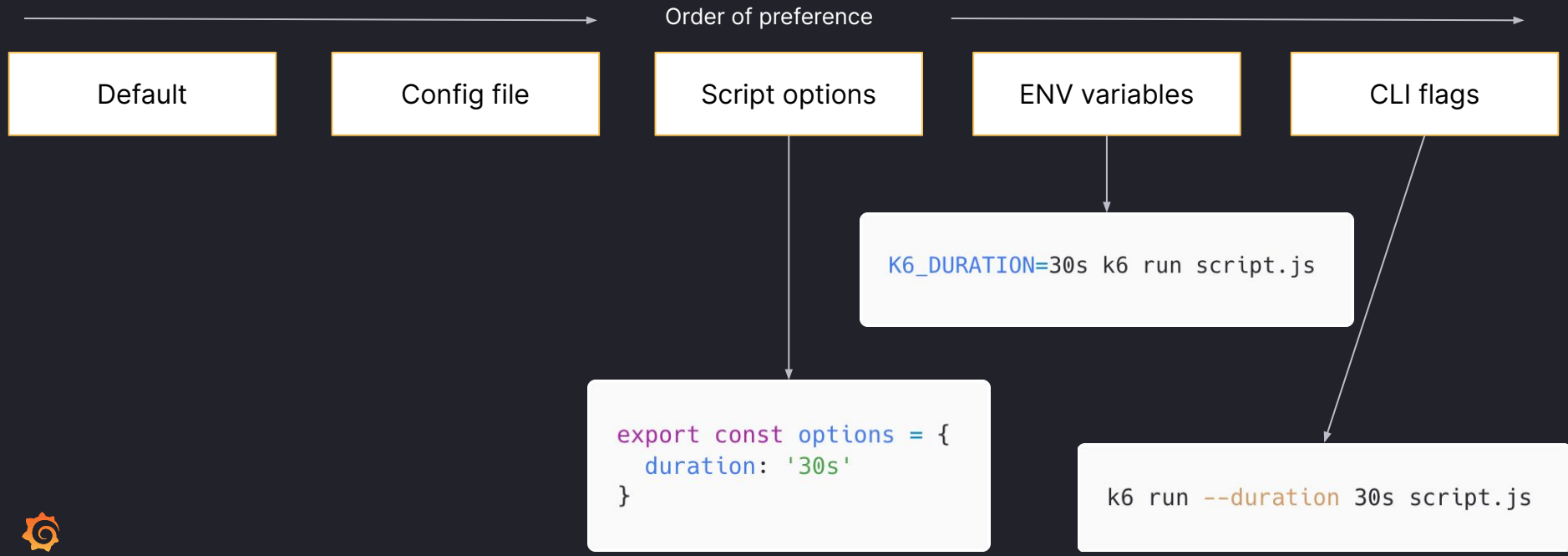
Other testing protocols



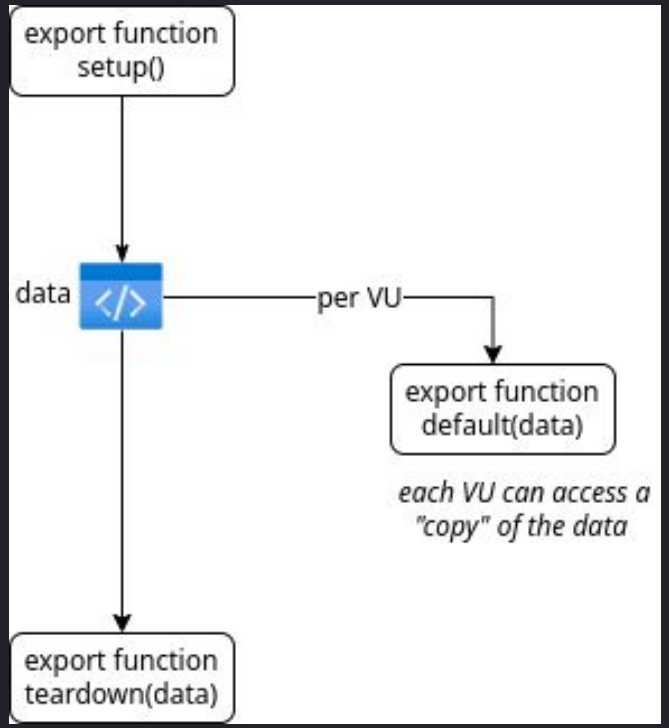
k6 Concepts

Options

Configurable options → https://k6.io/docs/using-k6/k6-options/reference/



Test life cycle / Test data -> https://k6.io/docs/using-k6/test-lifecycle/



Built-in Metrics

```
checks.........................: 100.00% ✓ 34       ✗ 0
data_received..................: 25 kB   2.6 kB/s
data_sent......................: 11 kB   1.1 kB/s
http_req_blocked...............: avg=275.57µs min=8µs    ...
http_req_connecting............: avg=137.6µs  min=0s     ...
http_req_duration..............: avg=154ms    min=7.47ms ...
  { expected_response:true }...: avg=154ms    min=7.47ms ...
http_req_failed................: 0.00%   ✓ 0        ✗ 35
http_req_receiving.............: avg=323.51µs min=81µs   ...
http_req_sending...............: avg=185.25µs min=46µs   ...
http_req_tls_handshaking.......: avg=0s       min=0s
http_req_waiting...............: avg=153.49ms min=4.64ms ...
http_reqs......................: 35      3.632738/s
iteration_duration.............: avg=1.09s    min=130µs  ...
iterations.....................: 34      3.528946/s
vus............................: 3       min=2      max=5
vus_max........................: 5       min=5      max=5
```

Custom Metrics

new Trend(‘metric_name’);

new Rate(‘metric_name’);

new Counter(‘metric_name’);

new Gauge(‘metric_name’);

```
import { Counter } from 'k6/metrics';

const myCounter = new Counter('my_counter');

export default function () {
  myCounter.add(1);
  myCounter.add(2);
}
```

Executors in K6

1. Shared iterations
2. Per VU iterations
3. Constant VUs
4. Ramping VUs
5. Constant Arrival Rate
6. Ramping Arrival Rate
7. Externally Controlled

```
export const options = {
  scenarios: {
    arbitrary_scenario_name: {
      //Name of executor
      executor: 'ramping-vus',
      // more configuration here
    },
  },
};
```

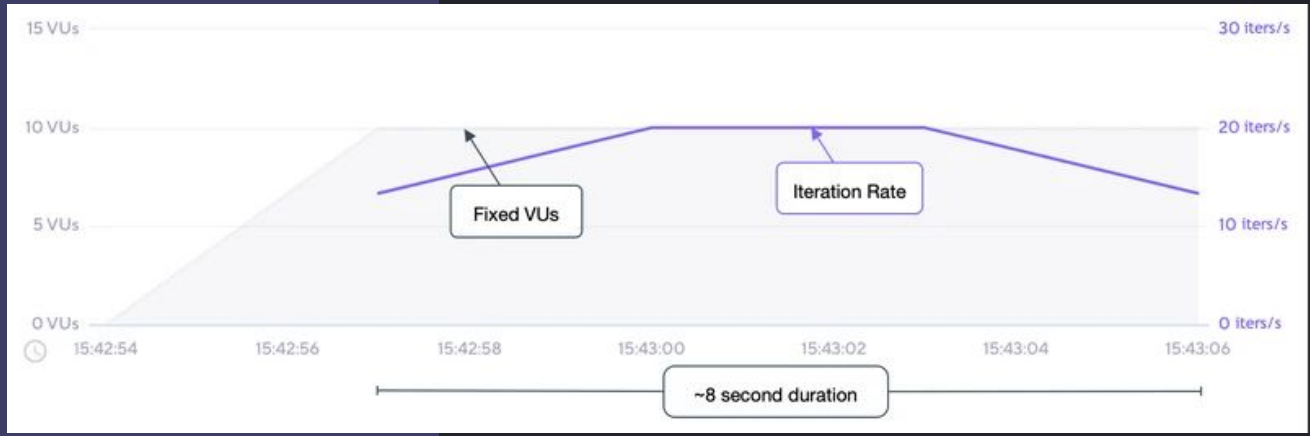
1. Shared iterations

- Shares iterations between the number of VUs.
- Test ends once k6 executes all iterations.

OPTION	TYPE	DESCRIPTION	DEFAULT
vus	integer	Number of VUs to run concurrently.	`1`
iterations	integer	Total number of script iterations to execute across all VUs.	`1`
maxDuration	string	Maximum scenario duration before it's forcibly stopped (excluding `gracefulStop`).	`"10m"`

shared-iters.js

```
1  import http from 'k6/http';
2  import { sleep } from 'k6';
3
4  export const options = {
5    discardResponseBodies: true,
6    scenarios: {
7      contacts: {
8        executor: 'shared-iterations',
9        vus: 10,
10       iterations: 200,
11       maxDuration: '30s',
12     },
13   },
14 };
15
16 export default function () {
17   http.get('https://test.k6.io/contacts.php');
18   // Injecting sleep
19   // Sleep time is 500ms. Total iteration time is sleep + time to fini
20   sleep(0.5);
21 }
```



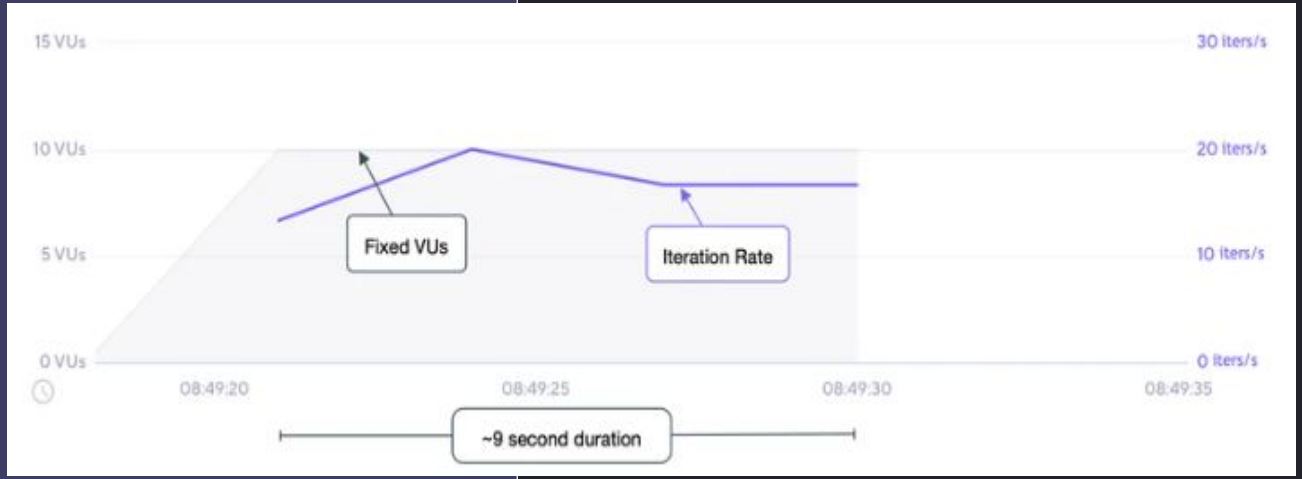
2. Per VU iterations

- Each VU executes an exact number of iterations.
- The total number of completed iterations equals vus * iterations.
- Can be useful when you have fixed sets of test data that you want to partition between VUs.

OPTION	TYPE	DESCRIPTION	DEFAULT
vus	integer	Number of VUs to run concurrently.	`1`
iterations	integer	Number of `exec` function iterations to be executed by each VU.	`1`
maxDuration	string	Maximum scenario duration before it's forcibly stopped (excluding `gracefulStop`).	`"10m"`

per-vu-iters.js

```
1  import http from 'k6/http';
2  import { sleep } from 'k6';
3
4  export const options = {
5    discardResponseBodies: true,
6    scenarios: {
7      contacts: {
8        executor: 'per-vu-iterations',
9        vus: 10,
10       iterations: 20,
11       maxDuration: '30s',
12     },
13   },
14 };
15
16 export default function () {
17   http.get('https://test.k6.io/contacts.php');
18   // Injecting sleep
19   // Sleep time is 500ms. Total iteration time is sleep + time to finish requ
20   sleep(0.5);
21 }
```



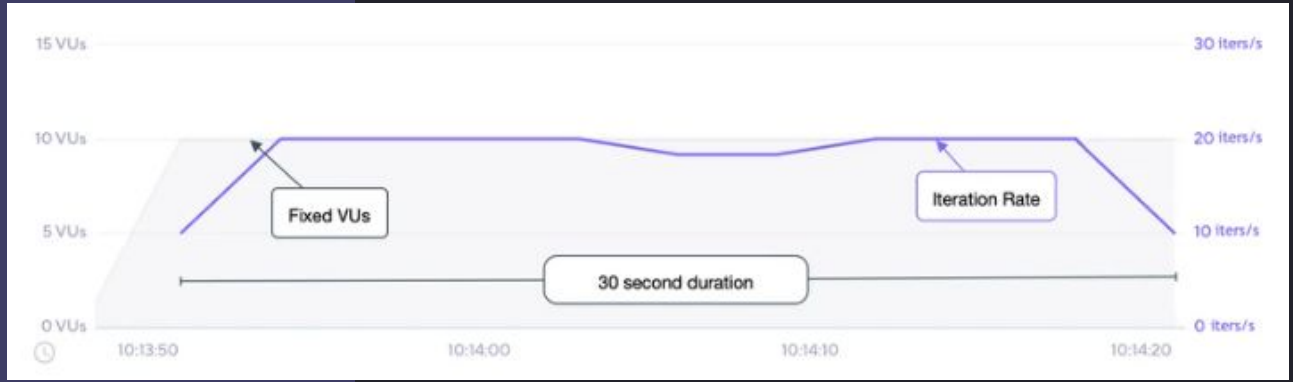
3. Constant VUs

- A fixed number of VUs execute as many iterations as possible for a specified amount of time.
- Useful if you need a specific number of VUs to run for a certain amount of time

OPTION	TYPE	DESCRIPTION	DEFAULT
duration(required)	string	Total scenario duration (excluding `gracefulStop`).	-
vus	integer	Number of VUs to run concurrently.	1

constant-vus.js

```
1  import http from 'k6/http';
2  import { sleep } from 'k6';
3
4  export const options = {
5    discardResponseBodies: true,
6    scenarios: {
7      contacts: {
8        executor: 'constant-vus',
9        vus: 10,
10       duration: '30s',
11     },
12   },
13 };
14
15 export default function () {
16   http.get('https://test.k6.io/contacts.php');
17   // Injecting sleep
18   // Total iteration time is sleep + time to finish request.
19   sleep(0.5);
20 }
```



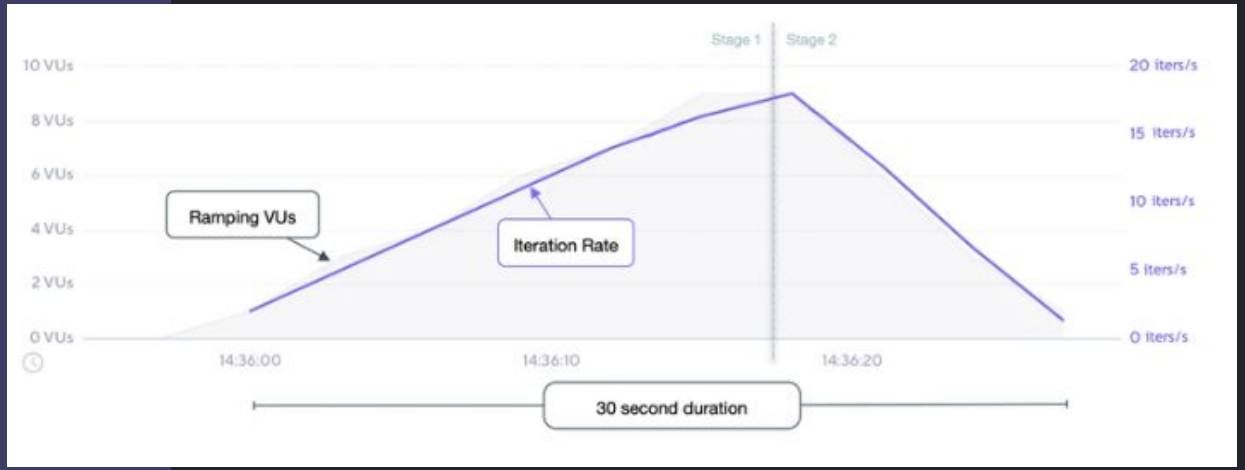
4. Ramping VUs

- A variable number of VUs executes as many iterations as possible for a specified amount of time.
- A shortcut to this executor, use the `stages` option i.e. a list of “{ target: ..., duration: ... }” objects.
- A good fit if you need VUs to ramp up or down during specific periods of time

OPTION	TYPE	DESCRIPTION	DEFAULT
stages(required)	array	Array of objects that specify the target number of VUs to ramp up or down to.	`[]`
startVUs	integer	Number of VUs to run at test start.	`1`
gracefulRampDown	string	Time to wait for an already started iteration to finish before stopping it during a ramp down.	`"30s"`

ramping-vus.js

```
1  import http from 'k6/http';
2  import { sleep } from 'k6';
3
4  export const options = {
5    discardResponseBodies: true,
6    scenarios: {
7      contacts: {
8        executor: 'ramping-vus',
9        startVUs: 0,
10       stages: [
11         { duration: '20s', target: 10 },
12         { duration: '10s', target: 0 },
13       ],
14       gracefulRampDown: '0s',
15     },
16   },
17 };
18
19 export default function () {
20   http.get('https://test.k6.io/contacts.php');
21   // Injecting sleep
22   // Sleep time is 500ms. Total iteration time is sleep +
23   sleep(0.5);
24 }
```



5. Constant Arrival Rate

- Continues to start iterations at the given rate as long as VUs are available.
- Iterations start independently of system response.

OPTION	TYPE	DESCRIPTION	DEFAULT
duration(required)	string	Total scenario duration (excluding `gracefulStop`).	-
rate(required)	integer	Number of iterations to start during each `timeUnit` period.	-
preAllocatedVUs(required)	integer	Number of VUs to pre-allocate before test start to preserve runtime resources.	-
timeUnit	string	Period of time to apply the `rate` value.	`"1s"`
maxVUs	integer	Maximum number of VUs to allow during the test run.	If unset, same as `preAllocatedVUs`

constant-arr-rate.js

```
import http from 'k6/http';

export const options = {
  discardResponseBodies: true,
  scenarios: {
    contacts: {
      executor: 'constant-arrival-rate',

      // How long the test lasts
      duration: '30s',

      // How many iterations per timeUnit
      rate: 30,

      // Start `rate` iterations per second
      timeUnit: '1s',

      // Pre-allocate VUs
      preAllocatedVUs: 50,
    },
  },
};

export default function () {
  http.get('https://test.k6.io/contacts.php');
}
```

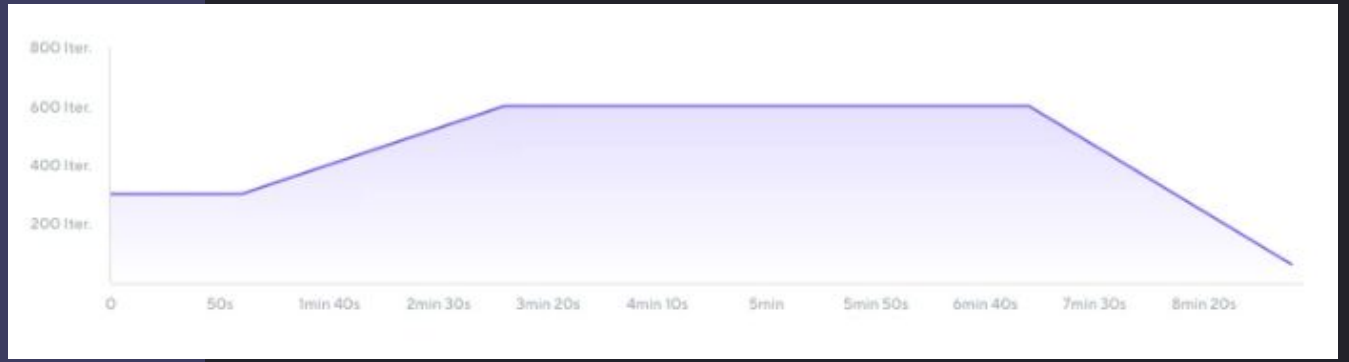
6. Ramping arrival rate

- Starts iterations at a variable rate.
- Dynamically changes the number of iterations to start according to the stages.

OPTION	TYPE	DESCRIPTION	DEFAULT
stages(required)	array	Array of objects that specify the target number of iterations to ramp up or down to.	`[]`
preAllocatedVUs(required)	integer	Number of VUs to pre-allocate before test start to preserve runtime resources.	-
startRate	integer	Number of iterations to execute each `timeUnit` period at test start.	`0`
timeUnit	string	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.	`"1s"`
maxVUs	integer	Maximum number of VUs to allow during the test run.	If unset, same as `preAllocatedVUs`

ramping-arr-rate.js

```
import http from 'k6/http';

export const options = {
  discardResponseBodies: true,

  scenarios: {
    contacts: {
      executor: 'ramping-arrival-rate',

      // Start iterations per `timeUnit`
      startRate: 300,

      // Start `startRate` iterations per minute
      timeUnit: '1m',

      // Pre-allocate necessary VUs.
      preAllocatedVUs: 50,

      stages: [
        // Start 300 iterations per `timeUnit` for the first minute.
        { target: 300, duration: '1m' },

        // Linearly ramp-up to starting 600 iterations per `timeUnit` over the fol
        { target: 600, duration: '2m' },

        // Continue starting 600 iterations per `timeUnit` for the following four
        { target: 600, duration: '4m' },

        // Linearly ramp-down to starting 60 iterations per `timeUnit` over the la
        { target: 60, duration: '2m' },
      ],
    },
  },
};

export default function () {
  http.get('https://test.k6.io/contacts.php');
}
```



7. Externally controlled

- Control execution at runtime via K6's rest API or the CLI
- Change aspects like number of VUs, Max VUs, pause or resume the test, list groups, set and get the setup data
- Helpful during the exploratory phase of deciding how many VUs your system can handle
- Only available when using k6 locally and not available in cloud

Using scenarios for hybrid testing

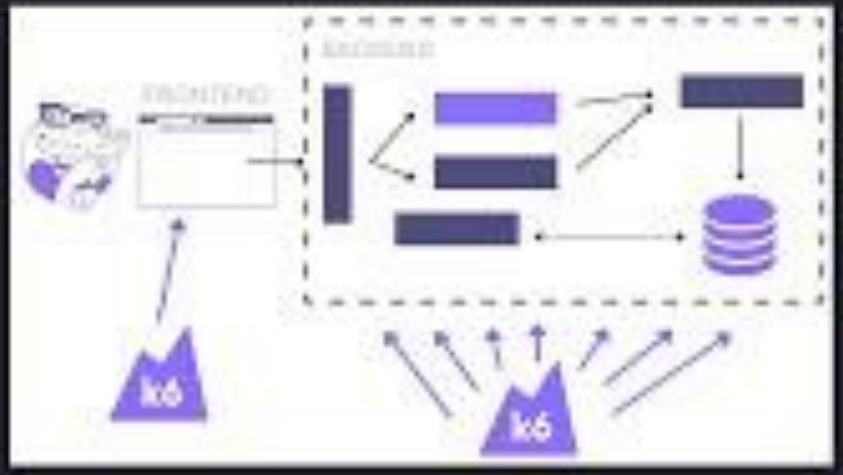
- Scenarios are independent from each other and run in parallel.

  Use `startTime` property to get sequential behaviour.
- Generate load on service using `stress` scenario and test actual user flow using `userFlow` scenario.

```
export const options = {
  scenarios: {
    stress: {
      exec: "generateLoad",
      executor: "ramping-vus",
      stages: [
        { duration: '5s', target: 5 },
        { duration: '10s', target: 5 },
        { duration: '5s', target: 0 },
      ],
    },
    userFlow: {
      exec: 'addproductToCart',
      executor: "constant-vus",
      vus: 1,
      duration: "30s",
      options: {
        browser: {
          type: "chromium",
        },
      },
    },
  },
};
```

Running a hybrid test with K6 in Cloud

Hybrid Testing



Benefits of running hybrid tests in cloud

1. Test for users from different geographic locations

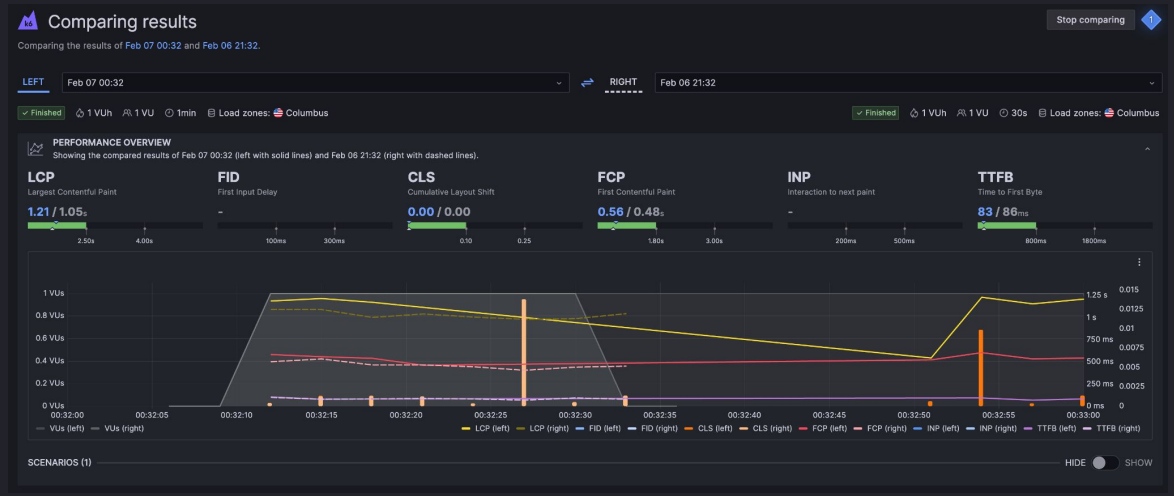
- Easily distribute your virtual users across the world with a simple configuration.
- Simulate real world traffic generated from different parts of the world.

```
export const options = {
  ext: {
    loadimpact: {
      distribution: {
        distributionLabel1: { loadZone: 'amazon:us:ashburn', percent: 50 },
        distributionLabel2: { loadZone: 'amazon:ie:dublin', percent: 50 },
      },
    },
  },
};
```

COPY

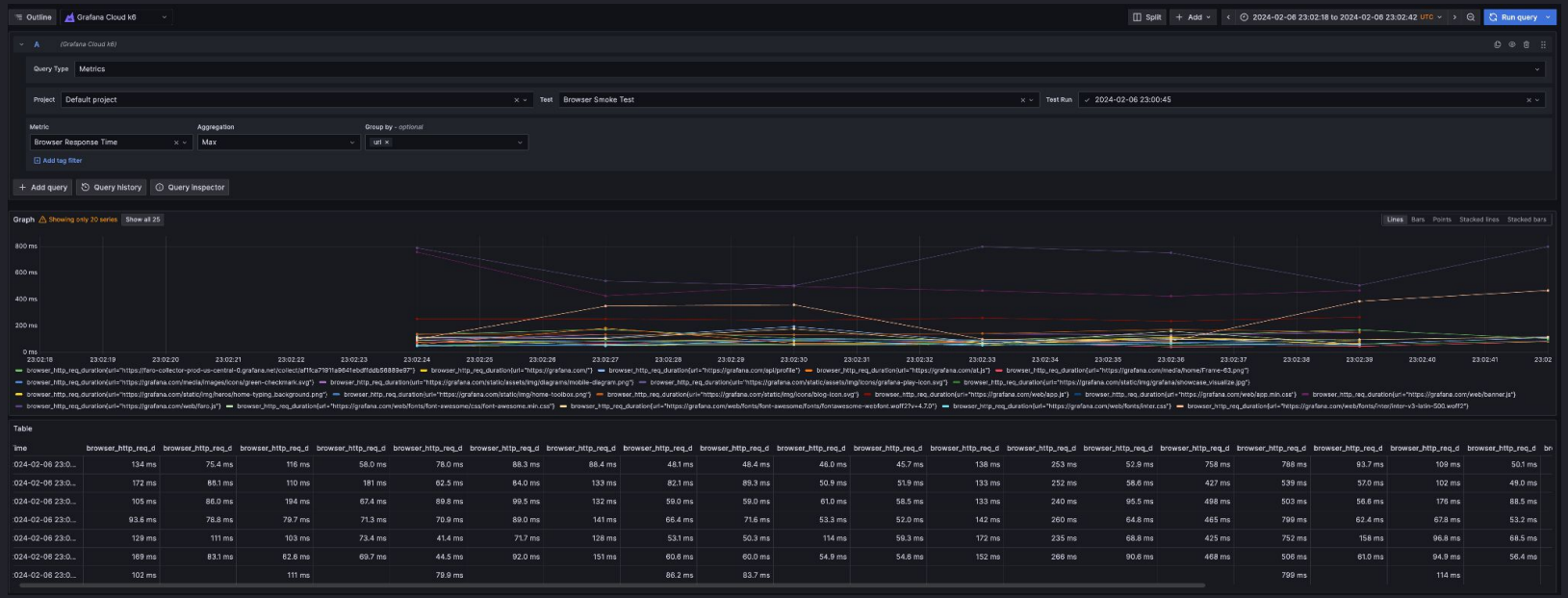
2. Easily compare results with previous test runs

- Compare test run results to find regressions and fix performance bottlenecks faster
- Learn More on test comparison.



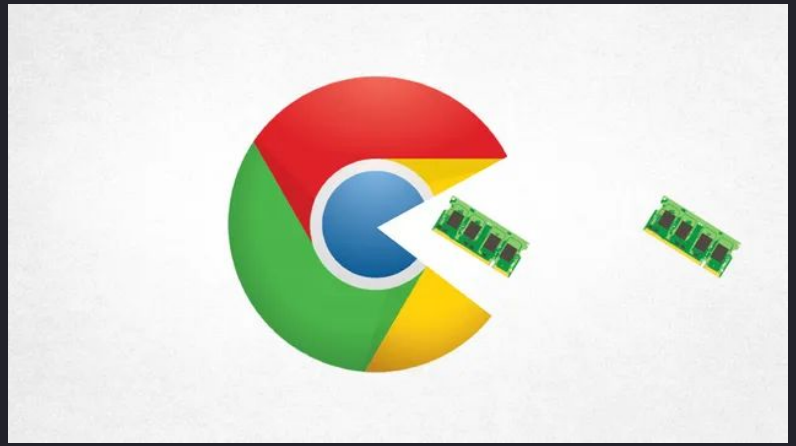
3. Custom dashboard from test run results

- Using Grafana cloud k6 adds a “K6” datasource to your stack which can be queried to get specific test run metrics to create custom dashboards for better tracking and visualizations on your test runs.



4. No infrastructure maintenance for browsers

- Running browser tests requires you to have, either
  - Beefy machines to support multiple browsers in the same instance, or
  - Use costly cloud browsers and maintain custom connection logic
- With Grafana K6 browser in Cloud, run browser tests in cloud without worrying about the infrastructure changes.

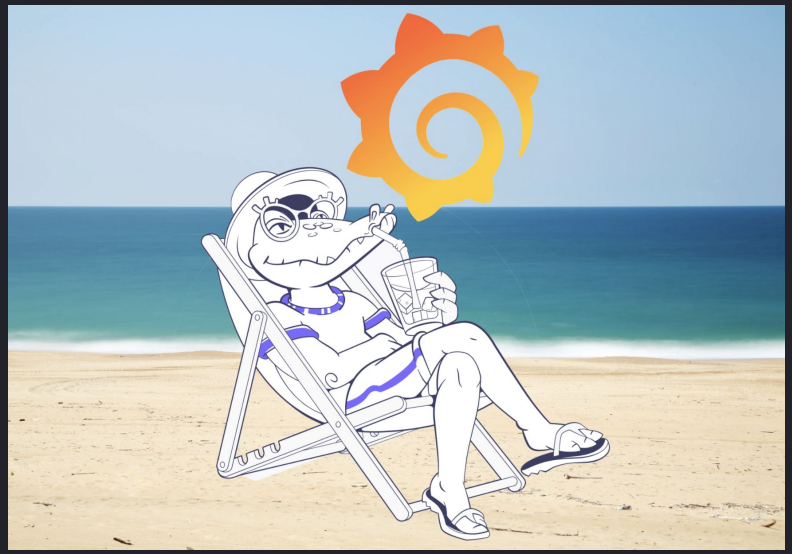


Recap

- Performance testing
- Load testing
  - Types of Load testing
- Frontend and Browser testing
- Hybrid testing
- K6 & Running a hybrid test using K6
- Benefits of running on hybrid test on cloud

Grafana Labs

Thank you



k6