Overview

k6 is a reliability testing tool. It helps developers simulate realistic user behavior and test how their systems behave as a result. Writing tests in k6 allows you to identify potential issues, such as slow response times or system failures, before they occur in production.

The goal of your test can vary. You might want to check performance, reliability, or scalability. Depending on your goal, your script may need different configurations, such as simulating many users, or running tests for a long time.

k6 tests are written using the JavaScript or TypeScript programming language, making it accessible to developers, and easy to integrate in existing codebases and projects. By writing k6 test scripts, you control what the k6 does, the action it performs, and how it behaves.

Engineering teams, including Developers, QA Engineers, SDETs, and SREs, commonly use k6 for:

- Load and performance testing
  k6 is optimized for minimal resource consumption and designed for running high-load performance tests such as spike, stress, or soak tests.
- Browser performance testing
  Through the k6 browser API, you can run browser-based performance tests and collect browser metrics to identify performance issues related to browsers. Additionally, you can mix browser tests with other performance tests to get a comprehensive view of your website's performance.
- Performance and synthetic monitoring
  You can schedule tests to run with minimal load very frequently, continuously validating the performance and availability of your production environment. For this, you can also use Grafana Cloud Synthetic Monitoring, which supports running k6 scripts.

- Automation of performance tests
  k6 integrates seamlessly with CI/CD and automation tools, enabling engineering teams to automate performance testing as part of their development and release cycle.
- Chaos and resilience testing
  You can use k6 to simulate traffic as part of your chaos experiments, trigger them from your k6 tests or inject different types of faults in Kubernetes with xk6-disruptor.
- Infrastructure testing
  With k6 extensions, you can add support to k6 for new protocols or use a particular client to directly test individual systems within your infrastructure.

Get started

To write k6 scripts, you'll need:

- A basic knowledge of JavaScript or TypeScript.
  - If you're unfamiliar with these languages, check out k6 Studio, which helps users generate tests without writing code. Alternatively, explore our test authoring methods.
- Install k6 in your machine.
- A code editor to write your scripts, such as Visual Studio Code or JetBrains editors.

Basic structure of a k6 test

For k6 to be able to interpret and execute your test, every k6 script follows a common structure, revolving around a few core components:

Default function: This is where the test logic resides. It defines what your test will do and how it will behave during execution. It should be exported as the default function in your script.

Imports: You can import additional k6 modules or JavaScript libraries (jslibs) to extend your script's functionality, such as making HTTP requests or simulating browser interactions.

Options (optional): Enable you to configure the execution of the test, such as defining the number of virtual users, the test duration, or setting performance thresholds.

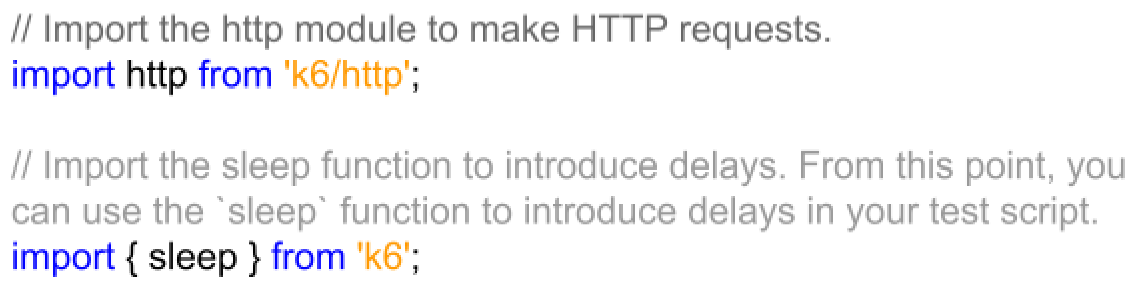
Lifecycle operations (optional): Because your test might need to run code before and/or after the execution of the test logic, such as parsing data from a file, or download an object from Amazon S3, lifecycle operations allow you to write code, either as predefined functions or within specific code scopes, that will be executed at different stages of the test execution.

Writing your first test script

Let's walk through creating a simple test which performs 10 `GET` HTTP requests to a URL and waits for 1 second between requests. This will help you understand the basic structure of a k6 test script.

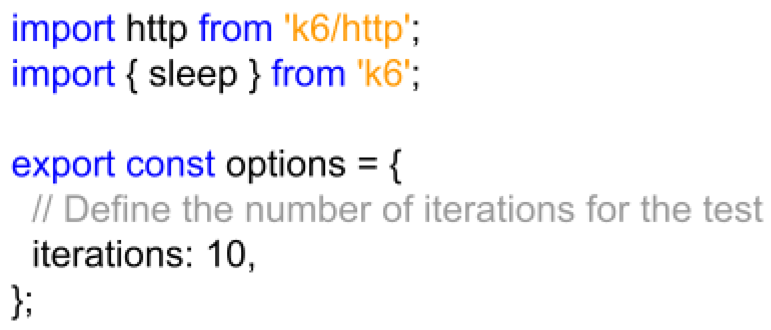
1. **Create a test file:** A test file can be named anything you like, and live wherever you see fit in your project, but it should have a `.js` or `.ts` extension.
2. **Import k6 modules**: As the end goal here is to perform HTTP requests, import the k6 `http` module at the top of the file. To help simulate a real-world scenario, import the `sleep` function from the `k6`

module as well.



```
// Import the http module to make HTTP requests.
import http from 'k6/http';

// Import the sleep function to introduce delays. From this point, you
can use the `sleep` function to introduce delays in your test script.
import { sleep } from 'k6';
```

3. **Define options**: To perform 10 HTTP requests, define an options block to configure the test execution. In this case, set the number of iterations to 10 to instruct k6 to execute the default function 10 times. Right beneath the imports, add the following code:



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

export const options = {
  // Define the number of iterations for the test
  iterations: 10,
};
```

4. **Define a default function**: The default exported function is the entry point for the test script. It will be executed repeatedly the number of times you define with the `iterations` option. In this function, make a `GET`

request to a URL and introduce a 1-second delay between requests.

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

export const options = {
  iterations: 10,
};

// The default exported function is gonna be picked up by k6 as the entry point for the test script.
// It will be executed repeatedly in "iterations" for the whole duration of the test.
export default function () {
  // Make a GET request to the target URL
  http.get('https://quickpizza.grafana.com');

  // Sleep for 1 second to simulate real-world usage
  sleep(1);
}
```

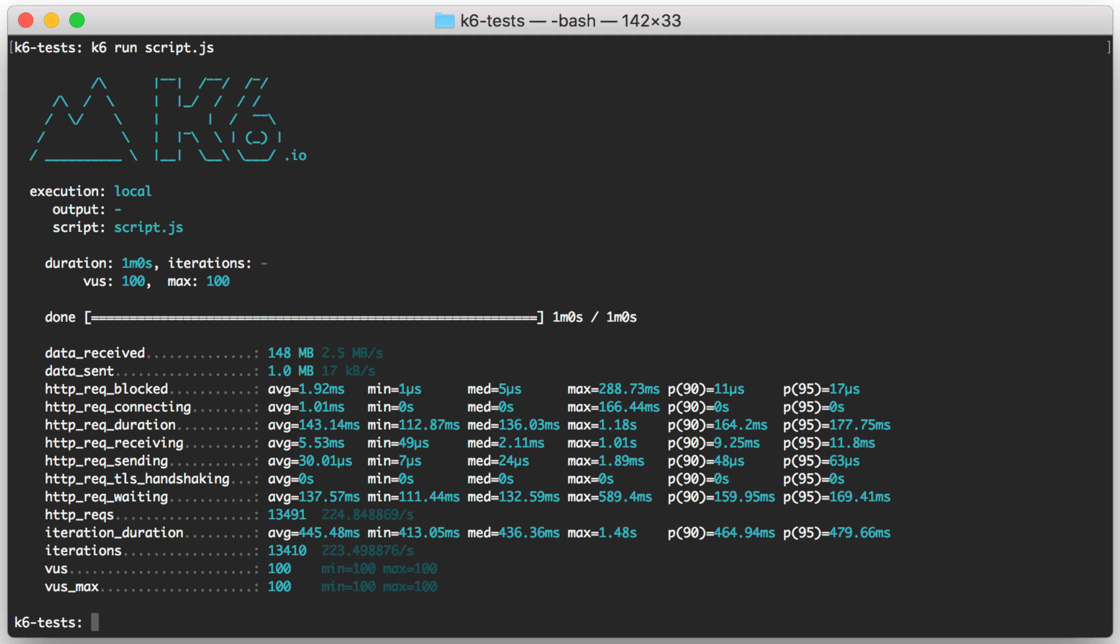
Running k6

If you run the script without flags, k6 uses the options defined in the script:

linux docker windows Copy

```
$ k6 run script.js
```

Results output



k6-tests — -bash — 142×33

```
k6-tests: k6 run script.js

          /\      |‾‾|  /‾‾/  /‾/
     /\  /  \     |  |_/  /  / /
    /  \/    \    |      |  /  ‾‾\
   /          \   |  |‾\  \ | (_) |
  / __________ \  |__|  \__\ \___/ .io

  execution: local
     output: -
     script: script.js

    duration: 1m0s, iterations: -
         vus: 100,  max: 100

    done [==========================================================] 1m0s / 1m0s

    data_received..............: 148 MB 2.5 MB/s
    data_sent..................: 1.0 MB 17 kB/s
    http_req_blocked...........: avg=1.92ms   min=1µs      med=5µs      max=288.73ms p(90)=11µs     p(95)=17µs
    http_req_connecting........: avg=1.01ms   min=0s       med=0s       max=166.44ms p(90)=0s       p(95)=0s
    http_req_duration..........: avg=143.14ms min=112.87ms med=136.03ms max=1.18s    p(90)=164.2ms  p(95)=177.75ms
    http_req_receiving.........: avg=5.53ms   min=49µs     med=2.11ms   max=1.01s    p(90)=9.25ms   p(95)=11.8ms
    http_req_sending...........: avg=30.01µs  min=7µs      med=24µs     max=1.89ms   p(90)=48µs     p(95)=63µs
    http_req_tls_handshaking...: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
    http_req_waiting...........: avg=137.57ms min=111.44ms med=132.59ms max=589.4ms  p(90)=159.95ms p(95)=169.41ms
    http_reqs..................: 13491  224.848869/s
    iteration_duration.........: avg=445.48ms min=413.05ms med=436.36ms max=1.48s    p(90)=464.94ms p(95)=479.66ms
    iterations.................: 13410  223.498876/s
    vus........................: 100    min=100 max=100
    vus_max....................: 100    min=100 max=100

k6-tests: 
```

Testing Guidelines

Load test types

Many things can go wrong when a system is under load. The system must run numerous operations simultaneously and respond to different requests from a variable number of users. To prepare for these performance risks, teams use load testing.

But a good load-testing strategy requires more than just executing a single script. Different patterns of traffic create different risk profiles for the application. For comprehensive preparation, teams must test the system against different *test types*.

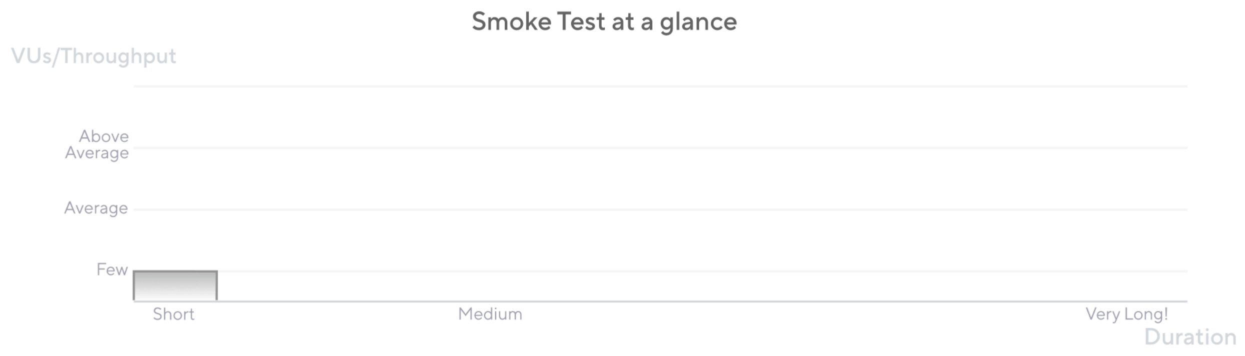
Types load testing

1. Smoke Test

Smoke tests have a minimal load. Run them to verify that the system works well under minimal load and to gather baseline performance values.

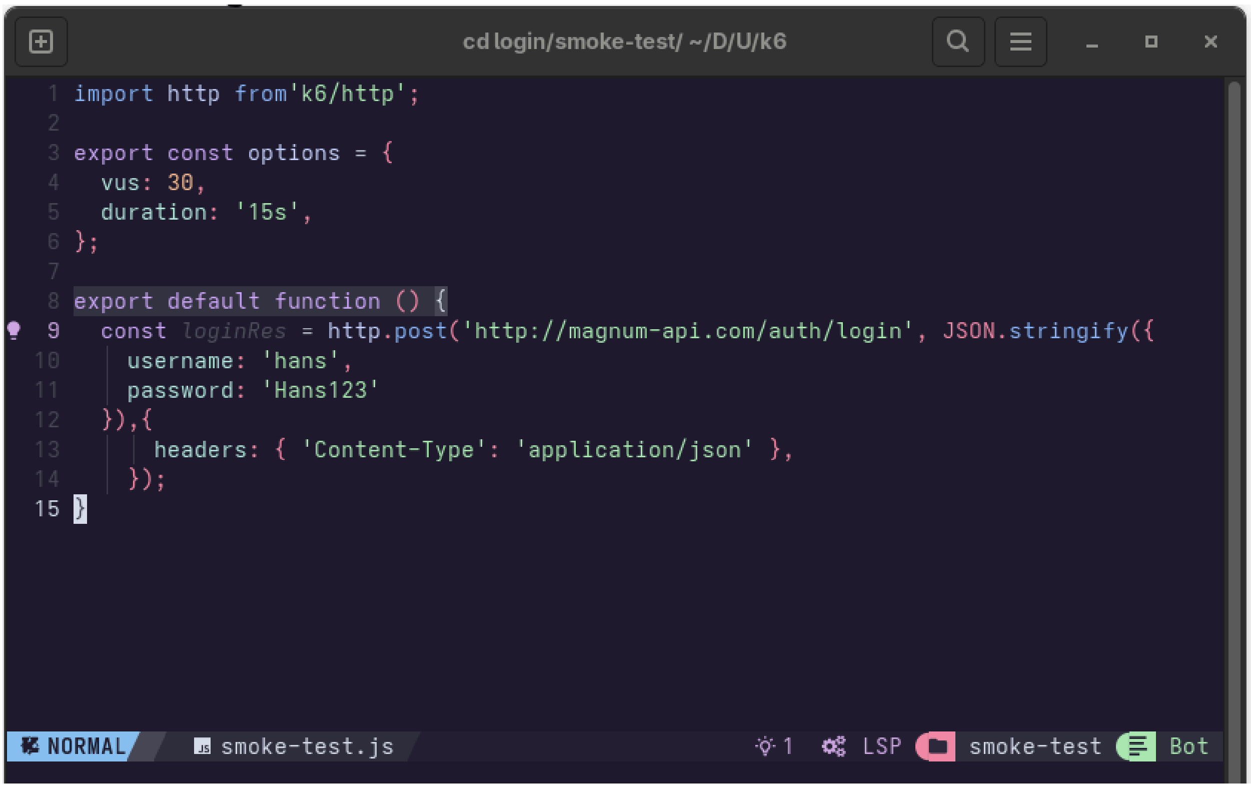
This test type consists of running tests with a few VUs — more than 5 VUs could be considered a mini-load test.

Similarly, the test should execute for a short period, either a low number of iterations or a duration from seconds to a few minutes maximum.



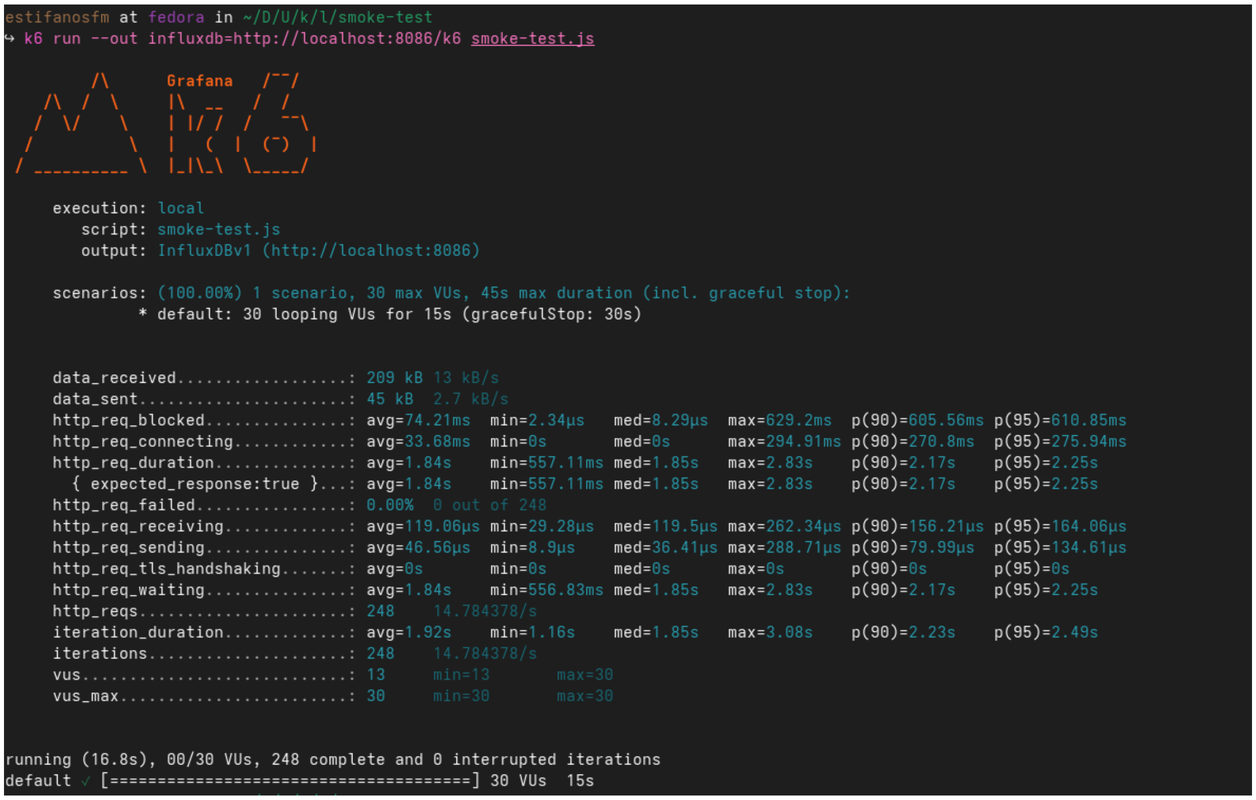
It’s a good practice to run a smoke test as a first step, with the following goals:

- Verify that your test script doesn’t have errors.
- Verify that your system doesn’t throw any errors (performance or system related) when under minimal load.
- Gather baseline performance metrics of your system’s response under minimal load.
- With simple logic, to serve as a synthetic test to monitor the performance and availability of production environments.

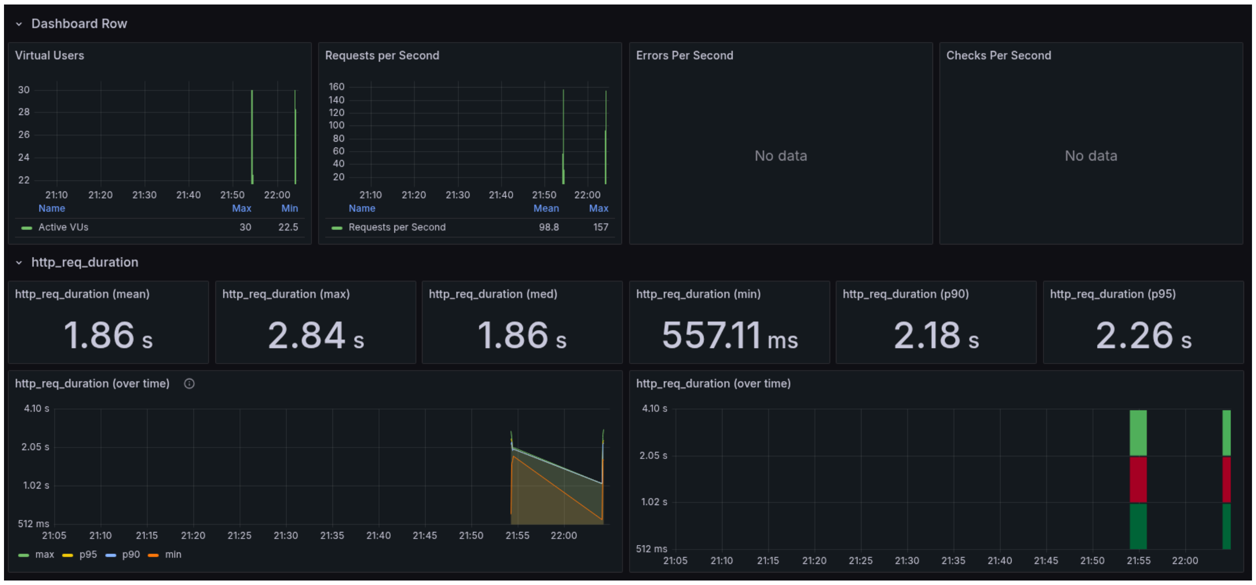


```
import http from'k6/http';

export const options = {
  vus: 30,
  duration: '15s',
};

export default function () {
  const loginRes = http.post('http://magnum-api.com/auth/login', JSON.stringify({
    username: 'hans',
    password: 'Hans123'
  }),{
      headers: { 'Content-Type': 'application/json' },
    });
}
```

Result



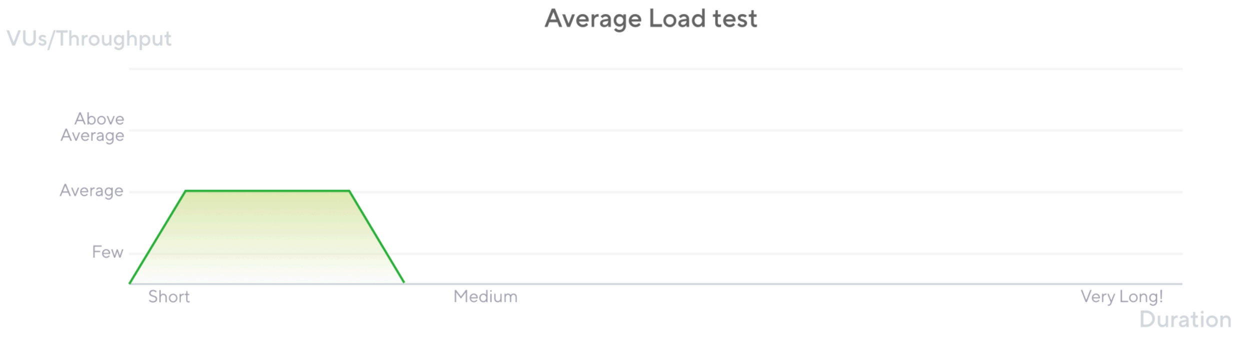
```
estifanosfm at fedora in ~/D/U/k/l/smoke-test
↪ k6 run --out influxdb=http://localhost:8086/k6 smoke-test.js

          /\      Grafana   /‾‾/
     /\  /  \     |\  __   /  /
    /  \/    \    | |/ /  /   ‾‾\
   /          \   |   (  |  (‾)  |
  / __________ \  |_|\_\  \_____/

     execution: local
        script: smoke-test.js
        output: InfluxDBv1 (http://localhost:8086)

     scenarios: (100.00%) 1 scenario, 30 max VUs, 45s max duration (incl. graceful stop):
              * default: 30 looping VUs for 15s (gracefulStop: 30s)


     data_received..................: 209 kB 13 kB/s
     data_sent......................: 45 kB  2.7 kB/s
     http_req_blocked...............: avg=74.21ms  min=2.34µs   med=8.29µs  max=629.2ms  p(90)=605.56ms p(95)=610.85ms
     http_req_connecting............: avg=33.68ms  min=0s       med=0s      max=294.91ms p(90)=270.8ms  p(95)=275.94ms
     http_req_duration..............: avg=1.84s    min=557.11ms med=1.85s   max=2.83s    p(90)=2.17s    p(95)=2.25s
       { expected_response:true }...: avg=1.84s    min=557.11ms med=1.85s   max=2.83s    p(90)=2.17s    p(95)=2.25s
     http_req_failed................: 0.00%  0 out of 248
     http_req_receiving.............: avg=119.06µs min=29.28µs  med=119.5µs max=262.34µs p(90)=156.21µs p(95)=164.06µs
     http_req_sending...............: avg=46.56µs  min=8.9µs    med=36.41µs max=288.71µs p(90)=79.99µs  p(95)=134.61µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s      max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=1.84s    min=556.83ms med=1.85s   max=2.83s    p(90)=2.17s    p(95)=2.25s
     http_reqs......................: 248    14.784378/s
     iteration_duration.............: avg=1.92s    min=1.16s    med=1.85s   max=3.08s    p(90)=2.23s    p(95)=2.49s
     iterations.....................: 248    14.784378/s
     vus............................: 13     min=13       max=30
     vus_max........................: 30     min=30       max=30


running (16.8s), 00/30 VUs, 248 complete and 0 interrupted iterations
default ✓ [======================================] 30 VUs  15s
```


Dashboard Row
Virtual Users
Requests per Second
Errors Per Second
No data
Checks Per Second
No data
Name Max Min
Active VUs 30 22.5
Name Mean Max
Requests per Second 98.8 157
http_req_duration
http_req_duration (mean) 1.86 s
http_req_duration (max) 2.84 s
http_req_duration (med) 1.86 s
http_req_duration (min) 557.11 ms
http_req_duration (p90) 2.18 s
http_req_duration (p95) 2.26 s
http_req_duration (over time)
max p95 p90 min
http_req_duration (over time)

2. Average-load test

An average-load test assesses how the system performs under typical load. Typical load might be a regular day in production or an average moment.

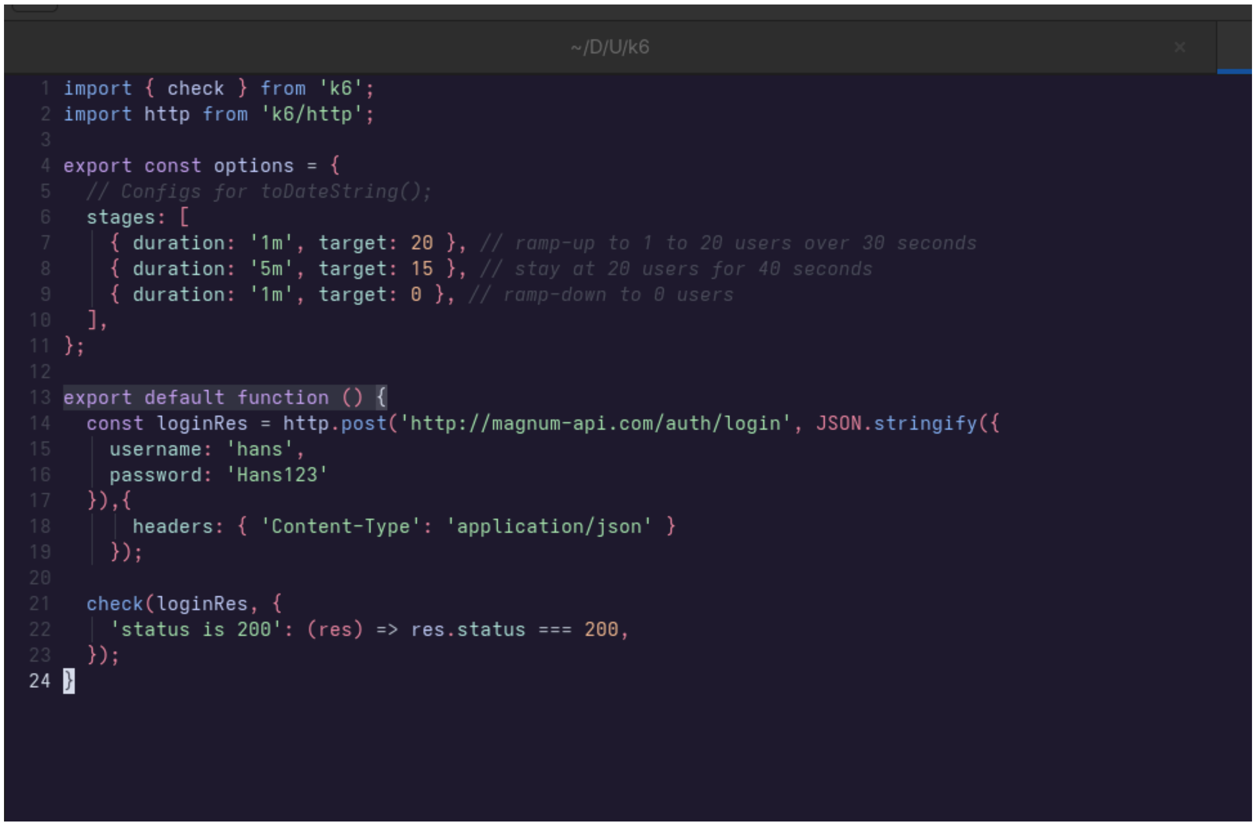
Average-load tests simulate the number of concurrent users and requests per second that reflect average behaviors in the production environment. This type of test typically increases the throughput or VUs gradually and keeps that average load for some time. Depending on the system's characteristics, the test may stop suddenly or have a short ramp-down period.



You should run an average-load test to:

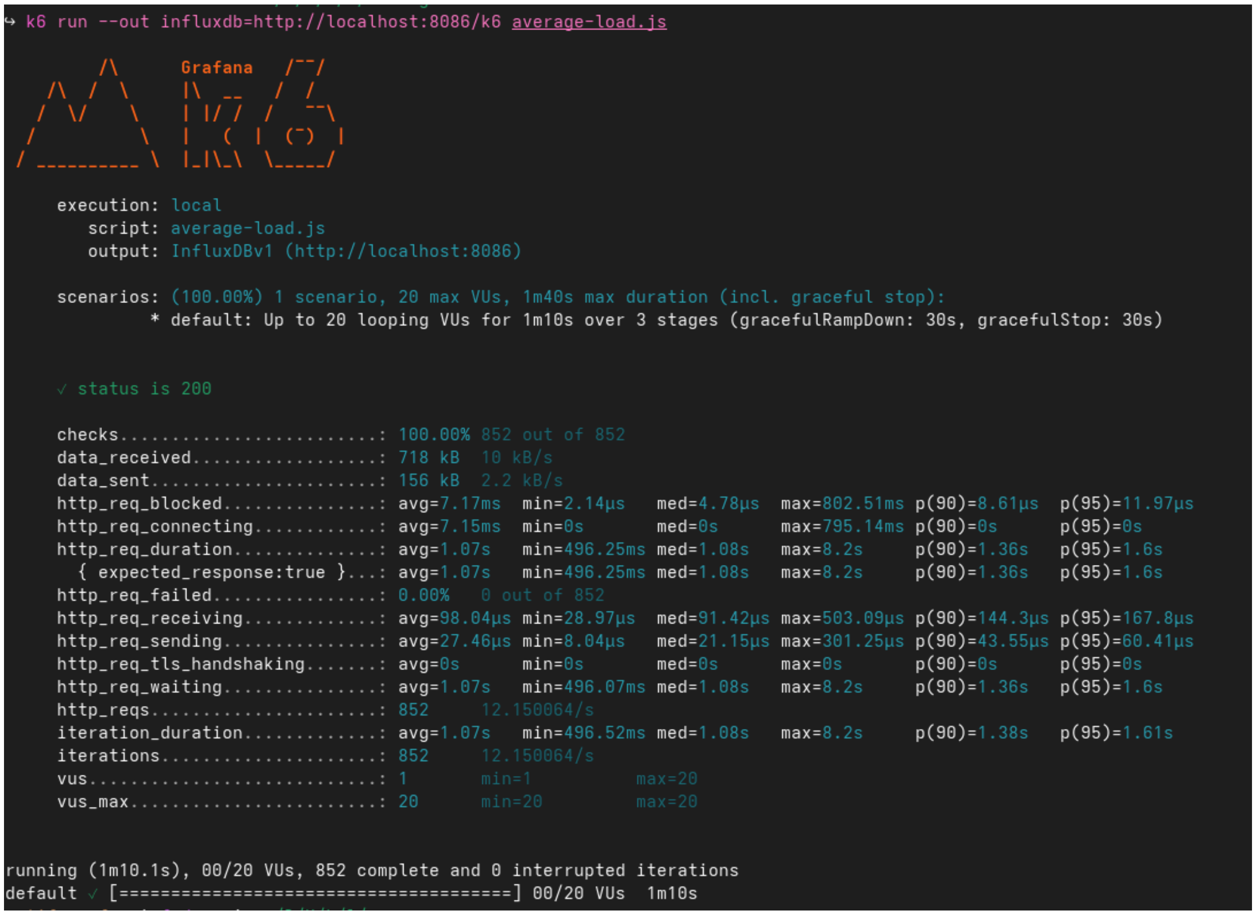
- Assess the performance of your system under a typical load.
- Identify early degradation signs during the ramp-up or full load periods.

- Assure that the system still meets the performance standards after system changes (code and infrastructure).

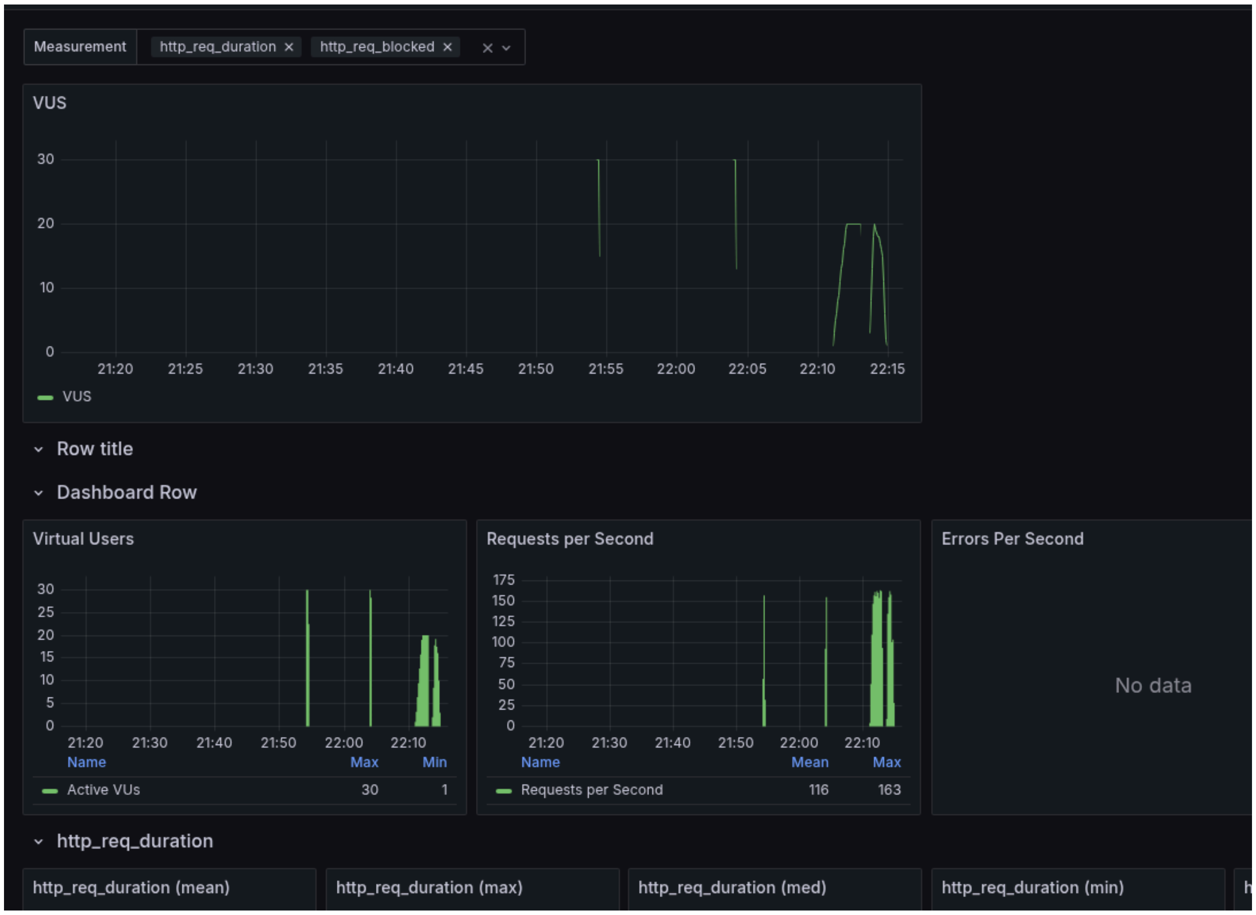

~/D/U/k6

```javascript
import { check } from 'k6';
import http from 'k6/http';

export const options = {
  // Configs for toDateString();
  stages: [
    { duration: '1m', target: 20 }, // ramp-up to 1 to 20 users over 30 seconds
    { duration: '5m', target: 15 }, // stay at 20 users for 40 seconds
    { duration: '1m', target: 0 }, // ramp-down to 0 users
  ],
};

export default function () {
  const loginRes = http.post('http://magnum-api.com/auth/login', JSON.stringify({
    username: 'hans',
    password: 'Hans123'
  }),{
      headers: { 'Content-Type': 'application/json' }
    });

  check(loginRes, {
    'status is 200': (res) => res.status === 200,
  });
}
```

Result

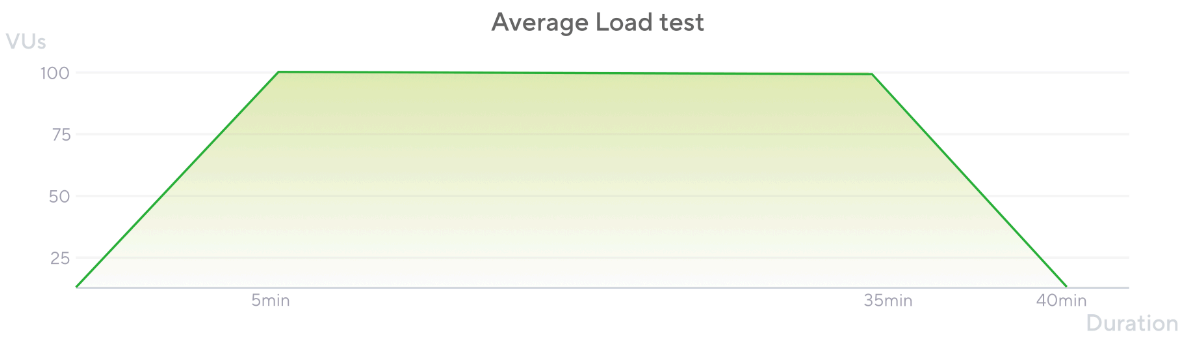


```
↪ k6 run --out influxdb=http://localhost:8086/k6 average-load.js

          /\      Grafana   /‾‾/
     /\  /  \     |\  __   /  /
    /  \/    \    | |/ /  /   ‾‾\
   /          \   |   (  |  (‾)  |
  / __________ \  |_|\_\  \_____/

     execution: local
        script: average-load.js
        output: InfluxDBv1 (http://localhost:8086)

     scenarios: (100.00%) 1 scenario, 20 max VUs, 1m40s max duration (incl. graceful stop):
              * default: Up to 20 looping VUs for 1m10s over 3 stages (gracefulRampDown: 30s, gracefulStop: 30s)


     ✓ status is 200

     checks.........................: 100.00% 852 out of 852
     data_received..................: 718 kB  10 kB/s
     data_sent......................: 156 kB  2.2 kB/s
     http_req_blocked...............: avg=7.17ms  min=2.14µs   med=4.78µs  max=802.51ms p(90)=8.61µs  p(95)=11.97µs
     http_req_connecting............: avg=7.15ms  min=0s       med=0s      max=795.14ms p(90)=0s      p(95)=0s
     http_req_duration..............: avg=1.07s   min=496.25ms med=1.08s   max=8.2s     p(90)=1.36s   p(95)=1.6s
       { expected_response:true }...: avg=1.07s   min=496.25ms med=1.08s   max=8.2s     p(90)=1.36s   p(95)=1.6s
     http_req_failed................: 0.00%   0 out of 852
     http_req_receiving.............: avg=98.04µs min=28.97µs  med=91.42µs max=503.09µs p(90)=144.3µs p(95)=167.8µs
     http_req_sending...............: avg=27.46µs min=8.04µs   med=21.15µs max=301.25µs p(90)=43.55µs p(95)=60.41µs
     http_req_tls_handshaking.......: avg=0s      min=0s       med=0s      max=0s       p(90)=0s      p(95)=0s
     http_req_waiting...............: avg=1.07s   min=496.07ms med=1.08s   max=8.2s     p(90)=1.36s   p(95)=1.6s
     http_reqs......................: 852     12.150064/s
     iteration_duration.............: avg=1.07s   min=496.52ms med=1.08s   max=8.2s     p(90)=1.38s   p(95)=1.61s
     iterations.....................: 852     12.150064/s
     vus............................: 1       min=1          max=20
     vus_max........................: 20      min=20         max=20


running (1m10.1s), 00/20 VUs, 852 complete and 0 interrupted iterations
default ✓ [======================================] 00/20 VUs  1m10s
```



Name	Max	Min
Active VUs	30	1

Name	Mean	Max
Requests per Second	116	163

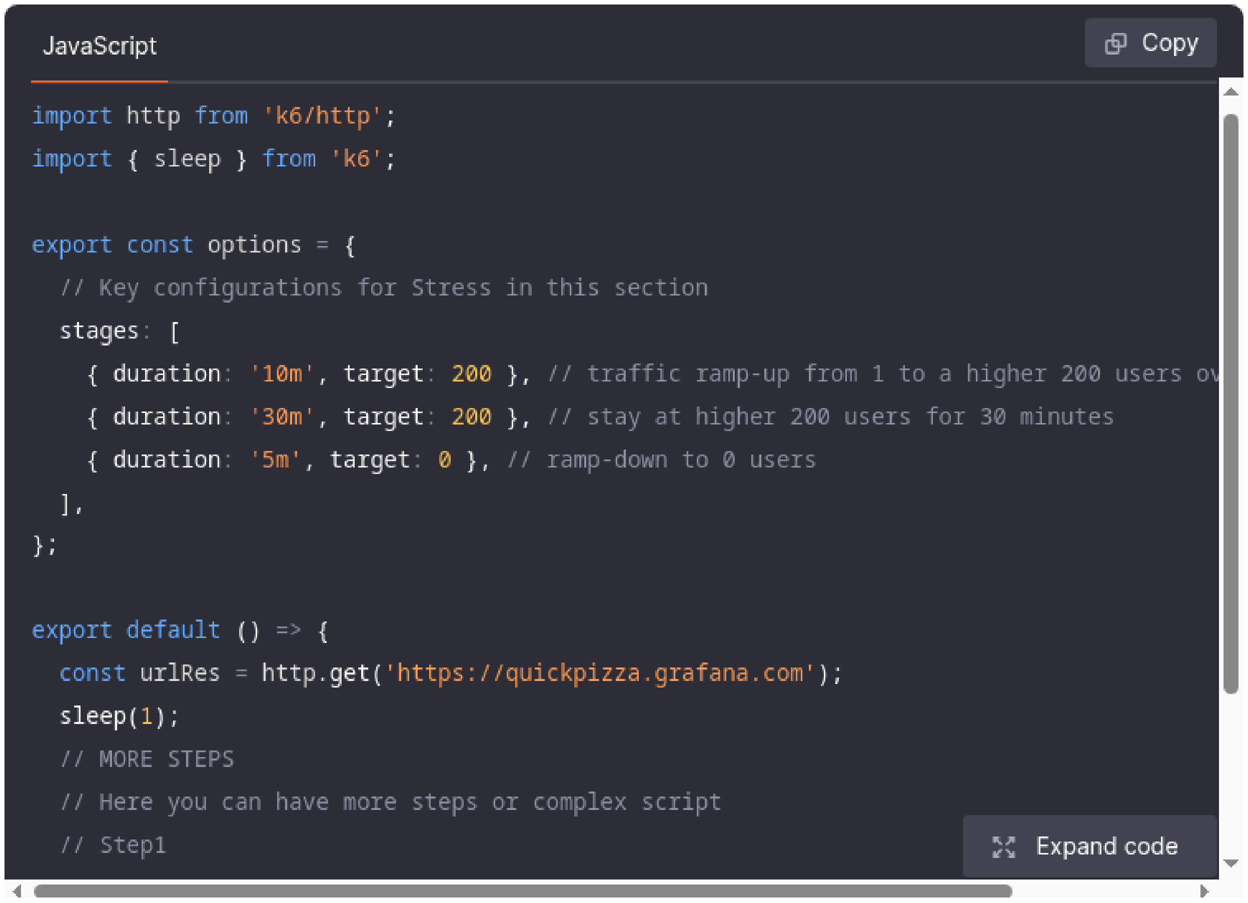


3. Stress Test

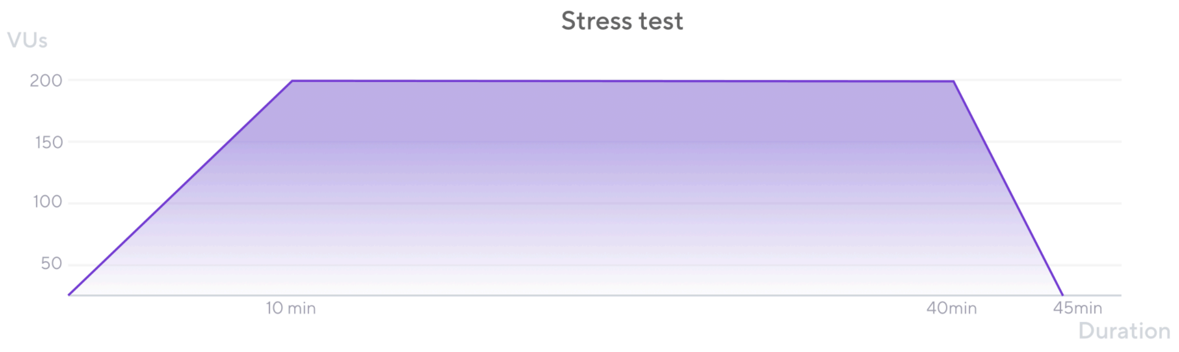
Stress testing assesses how the system performs when loads are heavier than usual.

The load pattern of a stress test resembles that of an average-load test. The main difference is higher load. To account for higher load, the ramp-up period takes longer in proportion to the load increase. Similarly, after the test reaches the desired load, it might last for slightly longer than it would in the average-load test.

Stress tests verify the stability and reliability of the system under conditions of heavy use. Systems may receive higher than usual workloads on unusual moments such as process deadlines, paydays, rush hours, ends of the workweek, and many other behaviors that might cause frequent higher-than-average traffic.



```javascript
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 to a higher 200 users ov
    { duration: '30m', target: 200 }, // stay at higher 200 users for 30 minutes
    { duration: '5m', target: 0 }, // ramp-down to 0 users
  ],
};

export default () => {
  const urlRes = http.get('https://quickpizza.grafana.com');
  sleep(1);
  // MORE STEPS
  // Here you can have more steps or complex script
  // Step1
```



4. Soak Test

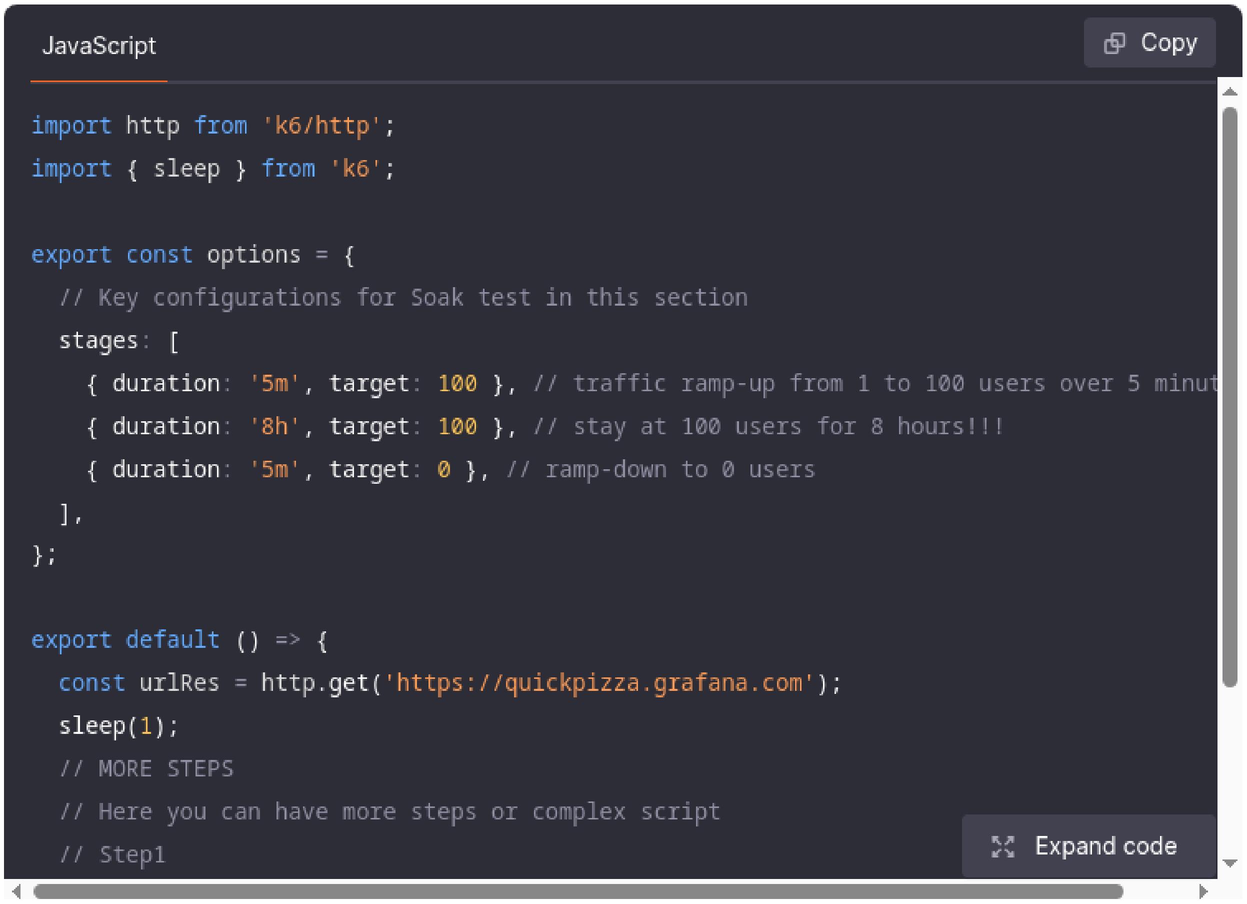
Soak testing is another variation of the Average-Load test. It focuses on extended periods, analyzing the following:

- The system’s degradation of performance and resource consumption over extended periods.
- The system’s availability and stability during extended periods.

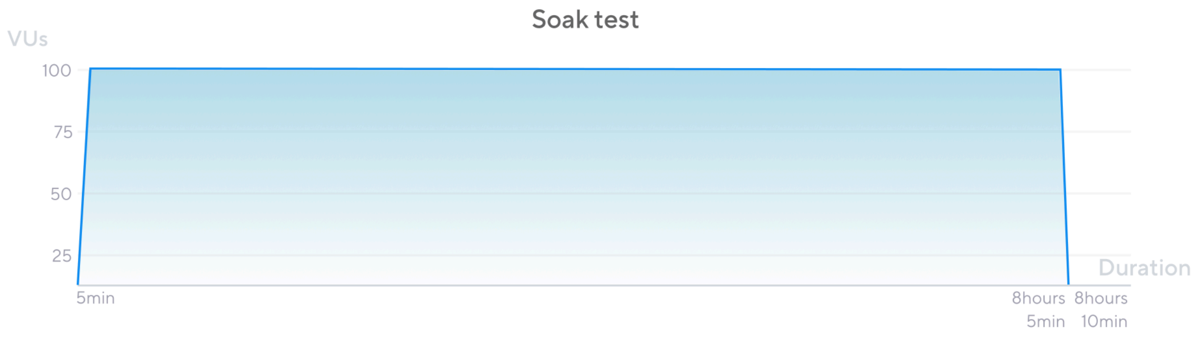
The soak test differs from an average-load test in test duration. In a soak test, the peak load duration (usually an average amount) extends several hours and even days. Though the duration is considerably longer, the ramp-up and ramp-down periods of a soak test are the same as an average-load test.

When you prepare to run a soak test, consider the following:

- Configure the duration to be considerably longer than any other test.
  Some typical values are 3, 4, 8, 12, 24, and 48 to 72 hours.
- If possible, re-use the average-load test script
  Changing only the peak durations for the aforementioned values.
- Don’t run soak tests before running smoke and average-load tests.
  Each test uncovers different problems. Running this first may cause confusion and resource waste.
- Monitor the backend resources and code efficiency. Since we are checking for system degradation, monitoring the backend resources and code efficiency is highly recommended. Of all test types, backend monitoring is especially important for soak tests.



```javascript
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 100 users over 5 minut
    { duration: '8h', target: 100 }, // stay at 100 users for 8 hours!!!
    { duration: '5m', target: 0 }, // ramp-down to 0 users
  ],
};

export default () => {
  const urlRes = http.get('https://quickpizza.grafana.com');
  sleep(1);
  // MORE STEPS
  // Here you can have more steps or complex script
  // Step1
```



5. Spike Test

A spike test verifies whether the system survives and performs under sudden and massive rushes of utilization.

Spike tests are useful when the system may experience events of sudden and massive traffic. Examples of such events include ticket sales (Taylor Swift), product launches (PS5), broadcast ads (Super Bowl), process deadlines (tax declaration), and seasonal sales (Black Friday). Also, spikes in traffic could be caused by more frequent events such as rush hours, a particular task, or a use case.

Spike testing increases to extremely high loads in a very short or non-existent ramp-up time. Usually, it has no plateau period or is very brief, as real users generally do not stick around doing extra steps in these situations. In the same way, the ramp-down is very fast or non-existent, letting the process iterate only once.

This test might include different processes than the previous test types, as spikes often aren’t part of an average day in production. It may also require adding, removing, or modifying processes on the script that were not in the average-load tests.

When preparing for a spike test, consider the following:

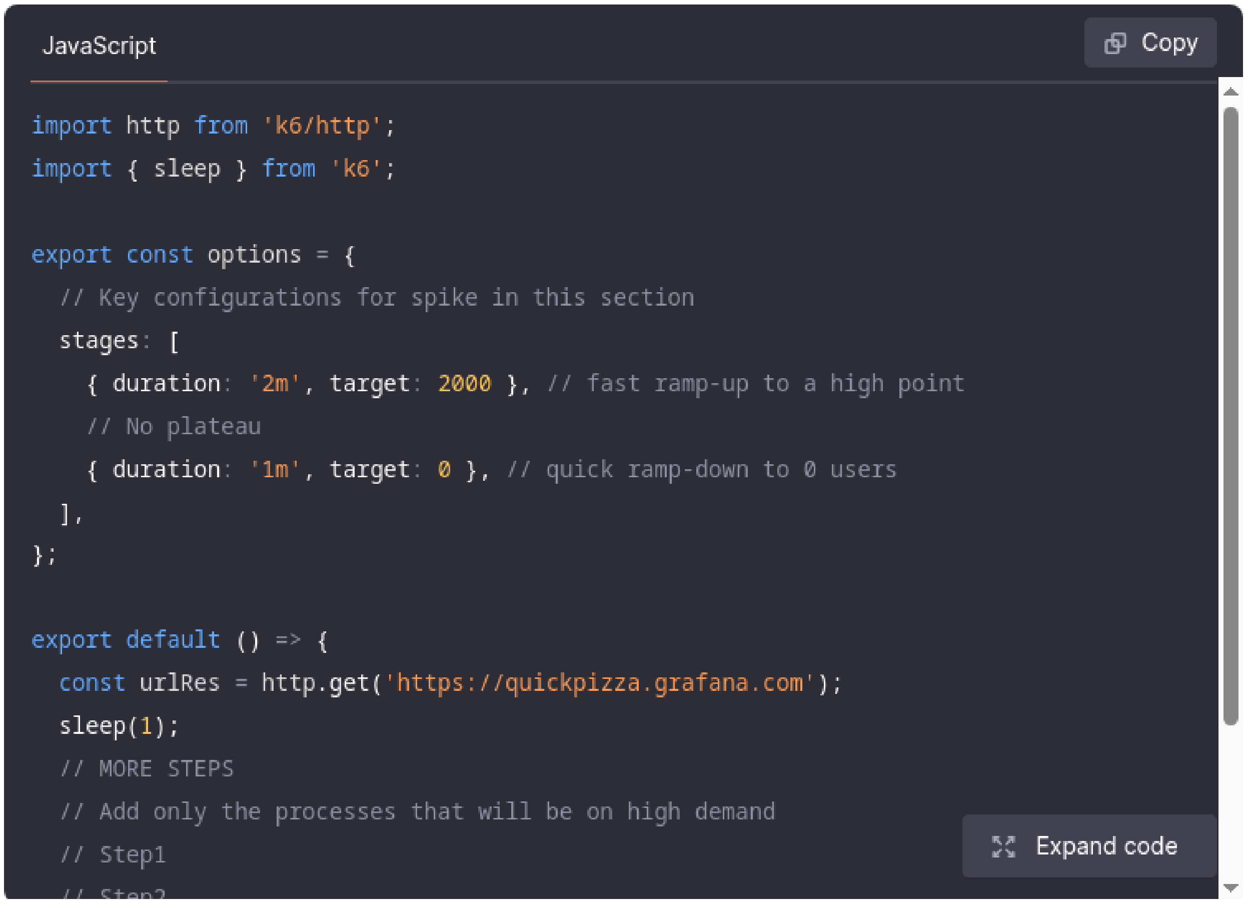
- Focus on key processes in this test type.

  Assess whether the spike in traffic triggers the same or different processes from the other test types. Create test logic accordingly.
- The test often won’t finish.

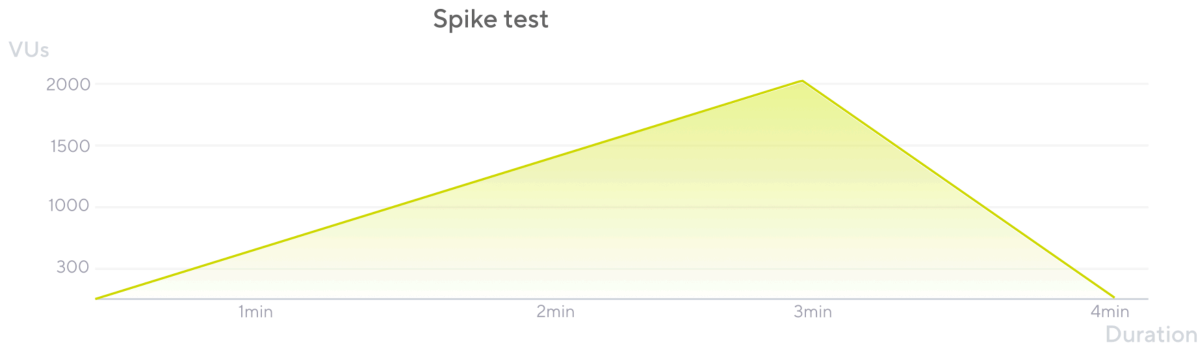
  Errors are common under these scenarios.
- Run, tune, repeat.

  When your system is at risk of spike events, the team must run a spikes test and tune the system several times.
- Monitor.

  Backend monitoring is a must for successful outcomes of this test.



```javascript
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://quickpizza.grafana.com');
  sleep(1);
  // MORE STEPS
  // Add only the processes that will be on high demand
  // Step1
  // Step2
```



6. Breakpoint Test

Breakpoint testing aims to find system limits. Reasons you might want to know the limits include:

- To tune or care for the system’s weak spots to relocate those higher limits at higher levels.
- To help plan remediation steps in those cases and prepare for when the system nears those limits.

In other words, knowing where and how a system starts to fail helps prepare for such limits.

A breakpoint ramps to unrealistically high numbers. This test commonly has to be stopped manually or automatically as thresholds start to fail. When these problems appear, the system has reached its limits.

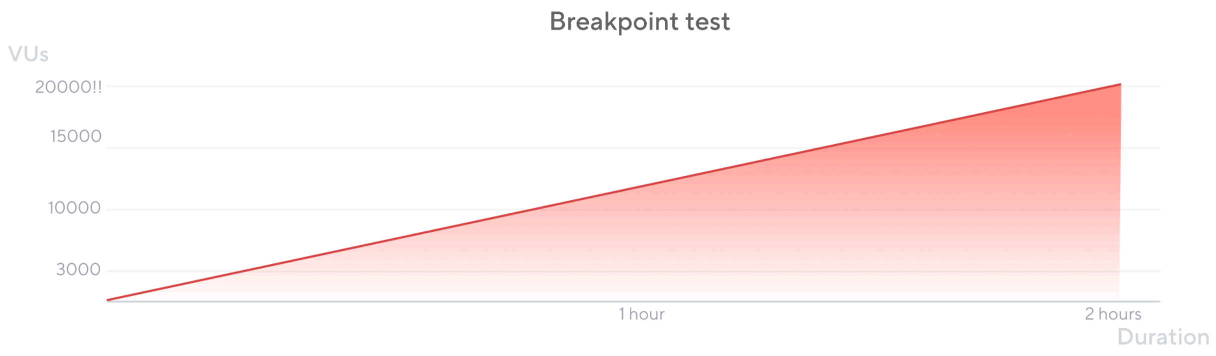
Teams execute a breakpoint test whenever they must know their system’s diverse limits. Some conditions that may warrant a breakpoint test include the following:

- The need to know if the system’s load expects to grow continuously
- If current resource consumption is considered high
- After significant changes to the code-base or infrastructure.

How often to run this test type depends on the risk of reaching the system limits and the number of changes to provision infrastructure components.

JavaScript

```javascript
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 the system slows
  stages: [
    { duration: '2h', target: 20000 }, // just slowly ramp-up to a HUGE load
  ],
};

export default () => {
  const urlRes = http.get('https://quickpizza.grafana.com');
  sleep(1);
  // MORE STEPS
  // Here you can have more steps or complex script
  // Step1
  // Step2
  // etc.
```



Different tests for different goals

Start with smoke tests, then progress to higher loads and longer durations.

The main types are as follows. Each type has its own article outlining its essential concepts.

- Smoke tests validate that your script works and that the system performs adequately under minimal load.
- Average-load test assess how your system performs under expected normal conditions.
- Stress tests assess how a system performs at its limits when load exceeds the expected average.
- Soak tests assess the reliability and performance of your system over extended periods.
- Spike tests validate the behavior and survival of your system in cases of sudden, short, and massive increases in activity.
- Breakpoint tests gradually increase load to identify the capacity limits of the system.

Start with a smoke test

Start with a smoke test. Before beginning larger tests, validate that your scripts work as expected and that your system performs well with a few users.

After you know that the script works and the system responds correctly to minimal load, you can move on to average-load tests. From there, you can progress to more complex load patterns.

Aim for simple designs and reproducible results

While the specifics are greatly context-dependent, what's constant is that you want to make results that you can compare and interpret.

Stick to simple load patterns. For all test types, directions is enough: ramp-up, plateau, ramp-down.

Avoid “rollercoaster” series where load increases and decreases multiple times. These will waste resources and make it hard to isolate issues.

Resources

Documentations

- K6 Official documentation

Communities

- Grafana Labs k6 Community

Articles

- APIs load testing using K6
- Performance Testing with K6: The New Top Dog for Load Testing and CI Pipelines

Videos

- How to do Performance Testing with k6 - Alex Hyett
- Is Your API ACTUALLY Ready for User Traffic? - Tech Vision