Load testing 101

Or: How I Learned To Stop Worrying And Love The Load

Dinko Vranjicic, Senior QA Engineer, Happening

Agenda

1. Introduction to performance tests
2. Introduction to k6
3. Your first load test
4. k6 in Happening
5. Things to consider
6. Q&A

Introduction to performance tests

What are performance tests?

- Type of **non-functional tests**
- Focused on the **general performance of the system under load**
- The desired load is usually generated by load testing **scripts**, executed with a certain load testing **tool**
- Usually executed **on demand**, but can also be made a **part of the CI/CD pipeline** or triggered with **scheduled runs**

Why performance testing?

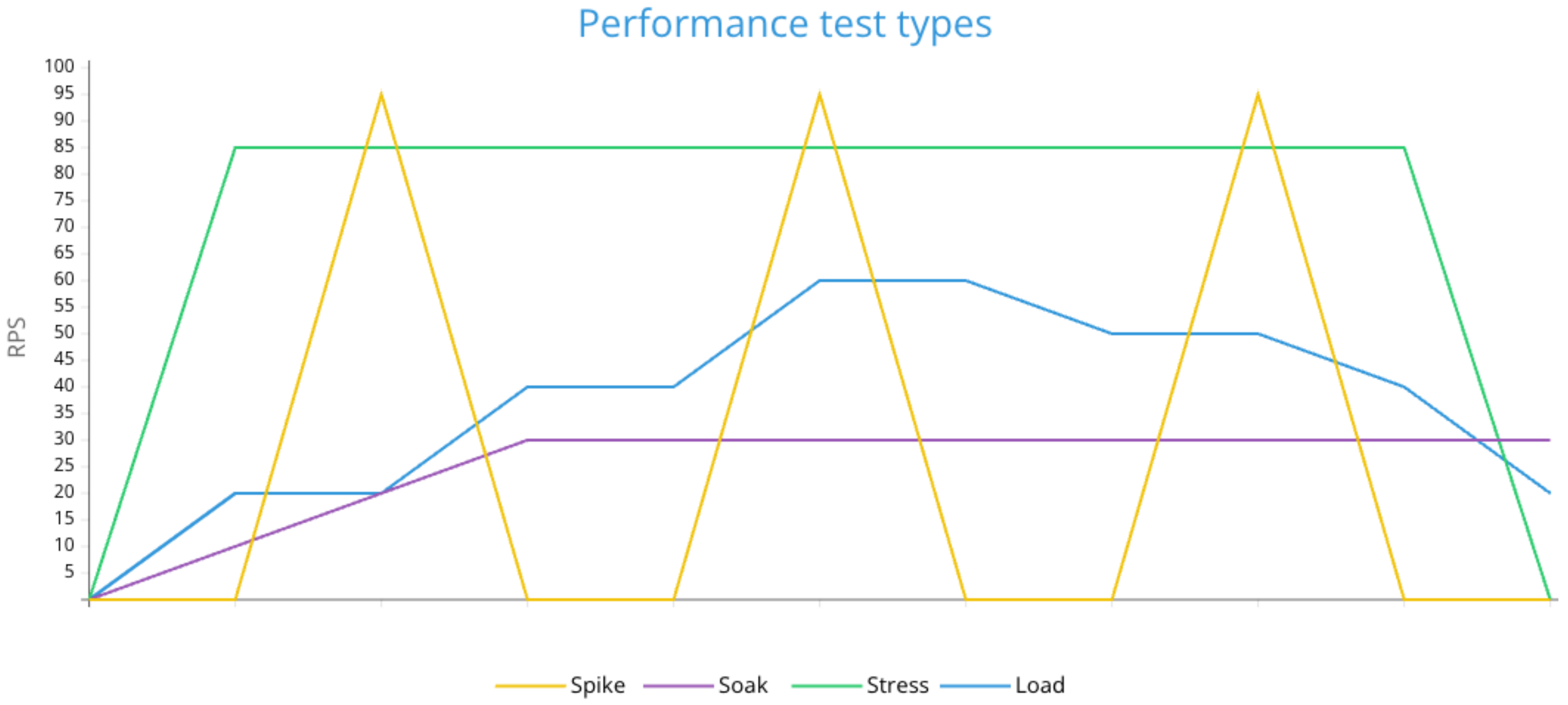
- **Builds confidence** that a service under test can withstand a certain amount of traffic without performance degradation
- Ensures that a new service is capable of performing well under **production-grade traffic**
- Become aware of the amount of load that introduces **performance degradations** in the system and **identify bottlenecks**
- Mitigate **impact on the business** by anticipating peak loads and scaling the system accordingly

Types of performance tests

- **Load testing**
  - Checks the system's ability to handle the expected user load
  - Identifies performance bottlenecks
- **Stress testing**
  - Checks the system performance under the extreme load
  - Identifies maximum system capacity
- **Soak (endurance) testing**
  - Tests the system over an extended period of increased load
  - Identifies resource leaks and degradations

Types of performance tests

- **Spike testing**
  - Testing the performance of the system under sudden load surges
  - Identifies system limits and performance
- **Capacity testing**
  - Similar approach to the stress test
  - Applying incremental load for a prolonged period of time
- **Scalability testing**
  - Useful for cloud-based systems
  - Checks upscaling/downscaling capability by introducing various load


Performance test types
RPS
100
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
Spike
Soak
Stress
Load


BlazeMeter®


NeoLoad


LOCUST


ARTILLERY.IO


APACHE
JMeter™


k6


A distributed Load Testing Tool
TSUNG


StormForge
ML-Powered K8s Optimization

BlazeMeter

NeoLoad

LOCUST

ARTILLERY.IO

APACHE
JMeter™


k6

A distributed Load Testing Tool
TSUNG


StormForge
ML-Powered K8s Optimization

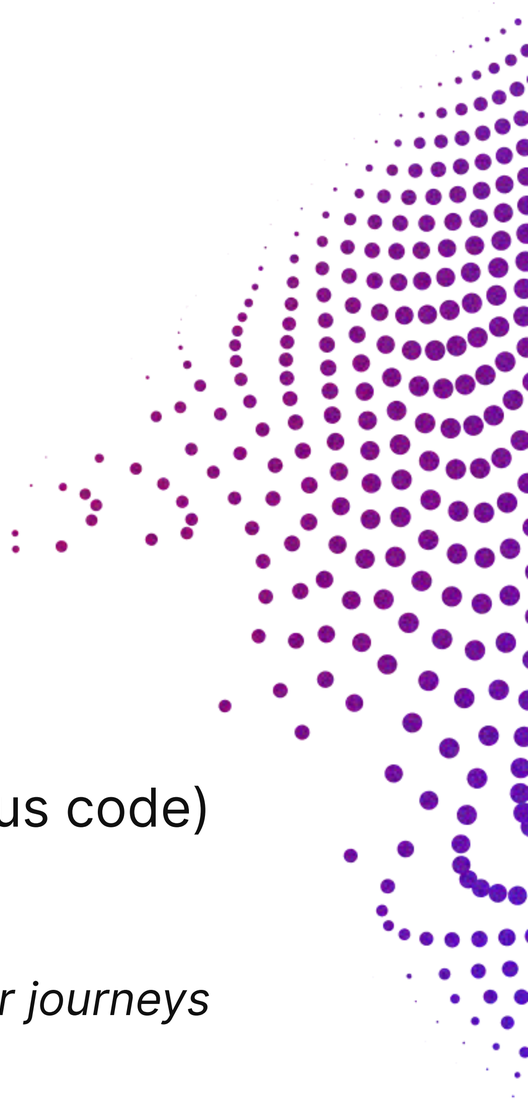
Introduction to k6

What is k6?

- **Open-source** load testing tool
- Released as LoadImpact in 2017, rebranded into k6 in 2020, acquired by Grafana Labs in 2021
- Enables producing load via **HTTP client**, supporting various protocols (HTTP/1.1 and 2, WebSockets, gRPC)
- Underlying load-producing engine written in **Go**
- Test scripts written in **JavaScript**

Features

- Test scripts are fully **code-based** (no UI whatsoever*)
- Test execution is done via **CLI** commands
- Official **Docker image** for easily containerizing your tests
- **k6 operator** for distributed test execution in Kubernetes cluster
- **Real-time metrics streaming** to various destinations (JSON, CSV, Prometheus, Elastic, InfluxDB...)
- **Built-in metrics** for all supported protocols
- Support for **custom metrics** (e.g. counting the occurrences of 404 status code)



** There is a **k6 Studio** desktop app in public preview that can be used to record user journeys in the browser and transform them to k6 test scripts*

Features

- Support for validations via **checks** and **thresholds** for each iteration
- Advanced load modeling using **scenarios** (e.g., having a test with multiple requests but with different load for each request) and **executors**
- Support for k6-specific **environment variables** (e.g., running the same test on QA and STAGE envs)
- **xk6** - a rich collection of both official and community-based extensions with additional features
- **k6 browser** module for interacting with browsers via Playwright-like API and collecting frontend performance metrics
- Convenient grouping of tests with **groups** and **tags**

k6 pros

- Easy installation and initial setup
- Ease of test scripting
- Support for extensions and integrations
- Customizable metrics
- Customizable reports + built-in HTML reporter
- Open-sourced (*but there is a paid Cloud version*)
- Rich documentation
- Strong community

k6 cons

- Test scripts can be written only in JS
  - TS supported as well but without type safety checks
- Unable to use JS packages relying on Node.js and browser-specific APIs
  - CDNs can be used, such as k6's own *jslib* or *jsdelivr*
- No official support for some technologies (Kafka, SSE)

Your first load test

QuickPizza

Login/Profile Advanced

"Pizza makes anything possible..." - Ernest Hemingway

Looking to break out of your pizza routine?

QuickPizza has your back!

With just one click, you'll discover new and exciting pizza combinations that you never knew existed.

Pizza, Please!

Made with ❤️ by QuickPizza Labs.

WebSocket visitor ID: 775789

Looking for the admin page? Click here

Contribute to QuickPizza on GitHub

Steps

1. Initial test file
2. Constants
3. Helper functions
4. Load modeling
5. Custom metrics
6. Checks and thresholds
7. Data preparation
8. Execution
9. Reports

Initial test file

```
// These imports will be used in the test.
import http, { url } from 'k6/http';
import { check, sleep } from 'k6';
import { Counter, Trend } from 'k6/metrics';

// Everything that needs to be executed prior to test execution.
// Returns "data" object used by the test function.
// If something in setup() fails, teardown() will not be executed!
export function setup() {}

// Contains code that is treated as a single iteration of a test.
// Execution count depends on the load modeling and virtual users (VUs).
export default function(data) {}

// Everything that needs to be executed after the test execution.
// Console logs, report handling, Slack notifications, etc.
export function teardown() {}
```

Constants

```
// BASE_URL and INITIAL_LOAD constants will take either the
// environment variable value or the one specified in the code.

const BASE_URL = __ENV.BASE_URL || "https://quickpizza.grafana.com/api"
const USERNAME = "TestUser_" + Date.now();
const PASSWORD = "12345678";
const INITIAL_LOAD = __ENV.INITIAL_LOAD || 5;
```

Helper functions

```
// Helper functions are declared in the "init" context, outside
// of the test function itself. They can exist in different files
// as well, which need to be imported.

export function loginUser(url, username, password) {
    return http.post(url + '/users/token/login', JSON.stringify({
        username: username,
        password: password,
    }), {
        headers: {
            'Content-Type': 'application/json',
        },
    }).json().token
}
```

Load modeling

```
// The load is distributed into several stages;
// the first stage takes 10 seconds to reach the INITIAL_LOAD value
// the second stage takes 60 seconds to reach the INITIAL_LOAD * 3 value
// etc.

export const options = {
    stages: [
        { duration: '10s', target: INITIAL_LOAD },
        { duration: '60s', target: INITIAL_LOAD * 3 },
        { duration: '15s', target: INITIAL_LOAD * 2 },
        { duration: '10s', target: 0 },
    ],
}
```

Custom metrics

```
// Custom metrics are defined in the "init" context.
// Changing values of metrics is done in the test function,
// and they are present in the report summary.

const errorCounter = new Counter('error_count');
const ingredientsTrend = new Trend('ingredients', false);
```

Checks and thresholds

```
// Thresholds define some boundary metrics that
// must not be surpassed. In this case, 95% of
// requests should not be longer than 800ms, and
// percentage of failed requests should be lower than 5%

export const options = {
    ...
    thresholds: {
        http_req_duration: ['p(95)<800'],
        http_req_failed: ['rate<0.05'],
    }
}

// Checks are assertions between values, in this case, status
// code of the API request should be equal to 201.
const rating = addRating(
  data.url,
  data.token,
  pizza.json().pizza.id,
  getRandomArrayElement([1, 5]));
    check(rating, {
        'Rating API call status is 201': (r) => r.status === 201,
    }) || errorCounter.add(1);
```

Data preparation

```
// Prepared data is passed in to test function in the form
// of a "data" object. Properties are the accessed in the test
// function using JSONPath syntax, e.g. "data.url".

export function setup() {
    createUser(BASE_URL, USERNAME, PASSWORD);
    console.log(`User ${USERNAME} created!`)

    const token = loginUser(BASE_URL, USERNAME, PASSWORD);
    console.log(`User ${USERNAME} logged in!`)
    console.log(`Token: ${token}`)

    return {
        url: BASE_URL,
        token: token };
}
```

Test execution

```
# This will run the test file as-is.
k6 run path/to/test/file.js

# This will change values of constants in the file on runtime.
k6 run -e INITIAL_LOAD=10 -e BASE_URL="http://localhost:3000" k6 run path/to/test/file.js

# This will enable web dashboard with real-time data and HTML export
K6_WEB_DASHBOARD=true K6_WEB_DASHBOARD_EXPORT=report.html k6 run path/to/test/file.js
```

Reports

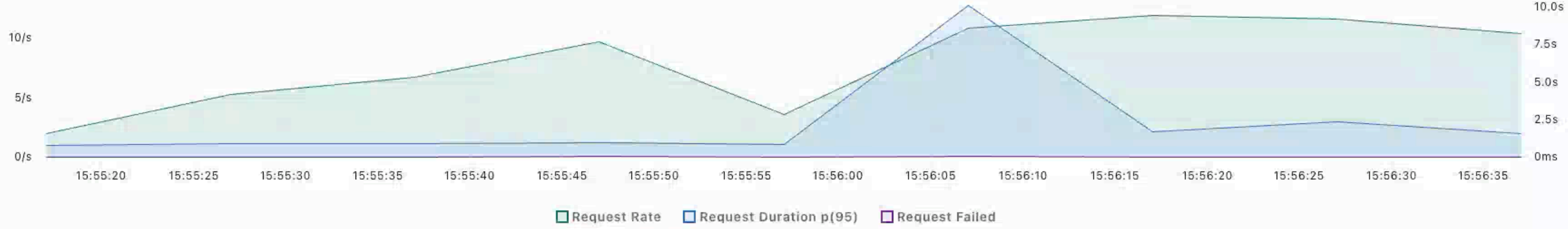
```
     ✓ Pizza API call status is 200
     ✗ Rating API call status is 201
      ↳  99% — ✓ 370 / ✗ 1

     checks.........................: 99.86% 741 out of 742
     data_received..................: 658 kB 6.7 kB/s
     data_sent......................: 214 kB 2.2 kB/s
     error_count....................: 1      0.010203/s
     http_req_blocked...............: avg=6.78ms   min=0s       med=1µs      max=708.92ms p(90)=3µs     p(95)=3µs
     http_req_connecting............: avg=2.92ms   min=0s       med=0s       max=236.33ms p(90)=0s      p(95)=0s
   ✗ http_req_duration..............: avg=641.56ms min=120.79ms med=529.45ms max=10.14s   p(90)=1.19s   p(95)=1.62s
       { expected_response:true }...: avg=547.93ms min=120.79ms med=526.02ms max=3.4s     p(90)=1.1s    p(95)=1.5s
   ✓ http_req_failed................: 1.84%  14 out of 757
     http_req_receiving.............: avg=951.38µs min=10µs     med=158µs    max=485.87ms p(90)=448.4µs p(95)=844.19µs
     http_req_sending...............: avg=430.34µs min=34µs     med=233µs    max=6.44ms   p(90)=915.2µs p(95)=1.34ms
     http_req_tls_handshaking.......: avg=3.31ms   min=0s       med=0s       max=188.16ms p(90)=0s      p(95)=0s
     http_req_waiting...............: avg=640.18ms min=120.41ms med=526.38ms max=10.13s   p(90)=1.19s   p(95)=1.62s
     http_reqs......................: 757    7.723657/s
     ingredients....................: avg=6.285714 min=4        med=6        max=8        p(90)=8       p(95)=8
     iteration_duration.............: avg=2.24s    min=128.45ms med=1.9s     max=10.14s   p(90)=2.88s   p(95)=3.81s
     iterations.....................: 384    3.917945/s
     vus............................: 1      min=0          max=15
     vus_max........................: 15     min=15         max=15


running (1m38.0s), 00/15 VUs, 384 complete and 0 interrupted iterations
default ✓ [======================================] 00/15 VUs  1m35s
ERRO[0098] thresholds on metrics 'http_req_duration' have been crossed
```

Report: 2025-03-10 15:55:07

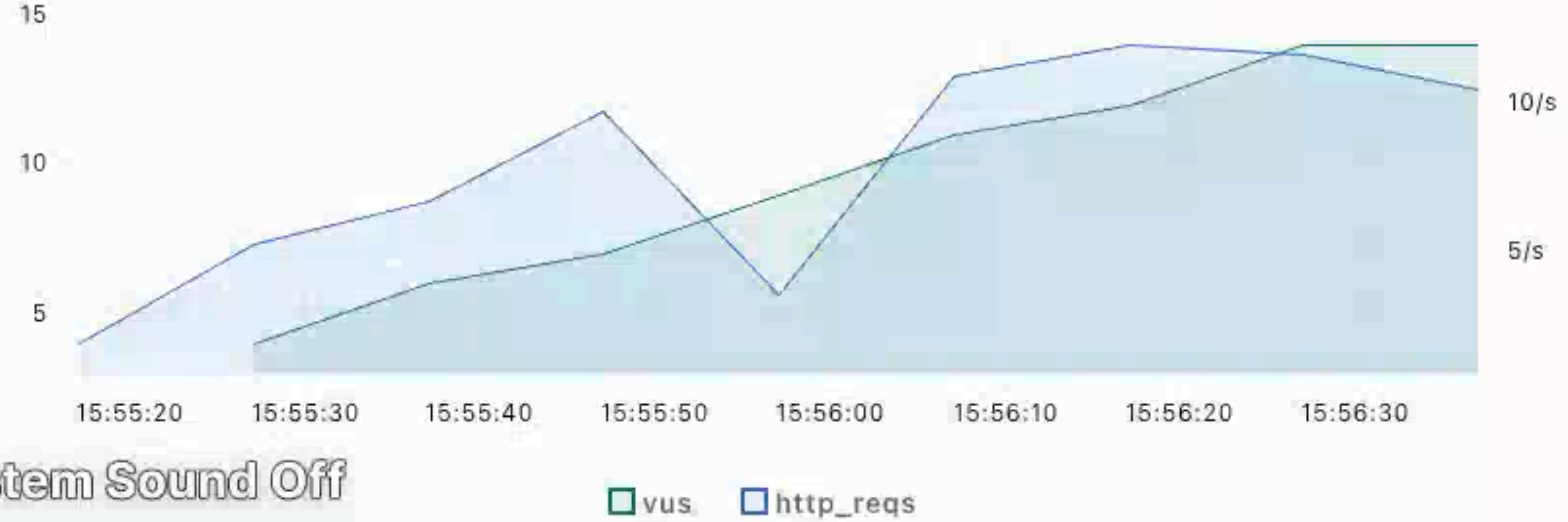
Overview

This chapter provides an overview of the most important metrics of the test run. Graphs plot the value of metrics over time.

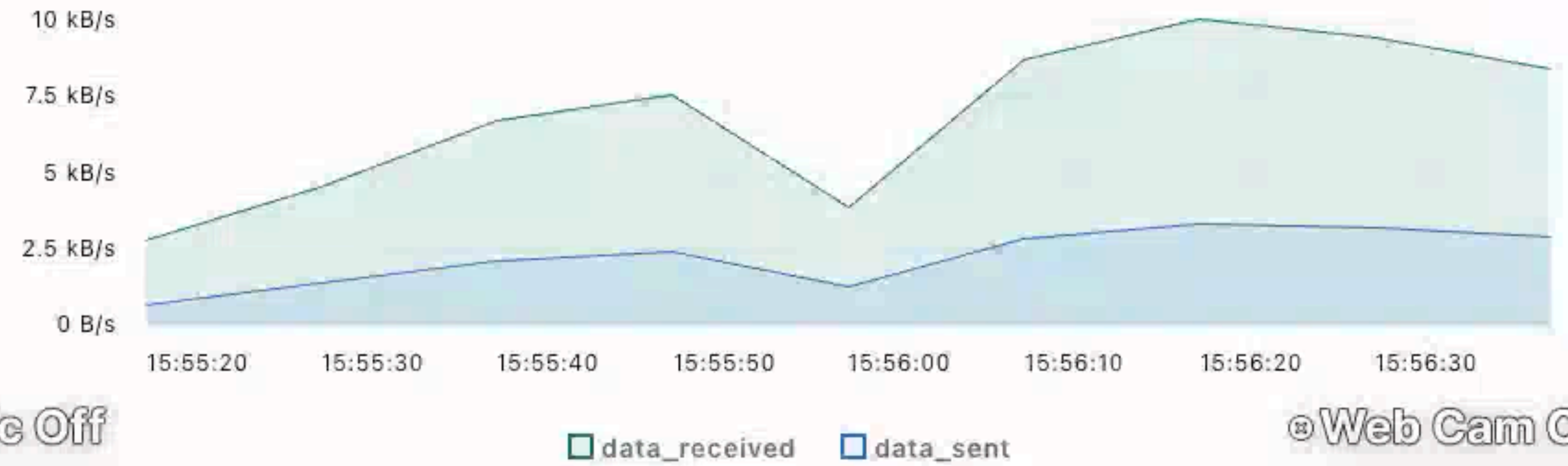
HTTP Performance overview



VUs



Transfer Rate



k6 in Happening

The story so far

- The first traces of k6 usage in Happening date back to **early 2023**
  - Performance testing of the new k8s cluster (REST, WS)
- Today, multiple teams are utilizing k6 for their performance testing needs:
  - Various **user journeys** covering login and payments
  - **Casino games** launch
  - Performance limits of the **cashout** system
  - Optimizations for the **Bet Builder** feature
  - Releasing a new version of our **client-facing API**
  - Hitting the new in-house **analytics collection tool** with production-grade traffic

Our experience

- The number of teams adopting k6 is **rising steadily**
- Overall, the experience is very **positive**
- Out-of-the-box features have proven more than enough for our use cases
  - A community-maintained extension has been utilized for testing **SSE** endpoints, for which k6 doesn't have official support (yet)
- **Easy onboarding** has been praised
- Various test execution approaches
  - Execution from the **local** machine
  - Execution via k8s cluster using **k6 operator**
  - Execution via k8s cluster using **cronjob** deployment

Things to consider

Good practices...

- Define **performance testing goals** for the system under test (SUT) upfront
  - Different goals = different approaches
- Make sure you **understand the SUT** and **how it is being used**
- Set up your **load test environment** properly
  - Ideally a copy of the production setup in terms of versions/resources
- Use **historical data** and define representative timeframes for load modeling
- Be mindful of the financial aspect - **load testing costs money...**
  - ...but probably less than having a performance issue in the production!

...and food for thought

- Properly define your **baseline load**
  - Can you extract it from the **production** load?
  - What if the service has **never** been deployed in production?
  - What about **traffic surges**?
- Make sure the **SUT is observable**
  - Can you easily check **resource usage** (CPU/RAM/disk)?
  - Can you easily check **logs** and spot **errors** and **crashes**?
  - How quickly can you pinpoint performance **bottlenecks**?

Q&A

Thank you!