Grafana Labs

k6

Shifting Left Chaos Testing

Pablo Chacin
Chaos Engineering Lead @ k6
Grafana Labs

Agenda

Why achieving reliability in modern applications is hard?

Chaos Engineering and the obstacles in adopting it

Introducing Chaos Testing

A tale of an incident

Chaos Testing with k6 disruptor extension

Why achieving reliability in is hard?

Complex architectures and infrastructures

Hard to predict failure modes

Inadequate testing tools and practices

How organizations can build confidence in their ability to withstand failures?

Chaos Experiments

Incident Enacting

Learn from Incidents

Production systems

Limitations of Chaos Engineering

Adoption bar is too high

Blast radius is hard to control

Results are hard to reproduce

Requires specialized tools

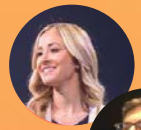
How **more** organizations can build confidence in their ability to withstand failures?

Fault Injection

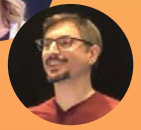
Is a software testing technique which introduces errors to a system to ensure it can withstand and recover from those conditions

From the distributed system perspective, almost all interesting availability experiments can be driven by affecting latency or response type.

Chaos Engineering, O'Reilly



Nora Jones



Casey Rosenthal

How effective is testing known errors?

According to a study of failures in real world distributed systems:

92% of the catastrophic system failures are the result of incorrect handling of non-fatal errors

In 58% of the cases the resulting faults could have been detected through simple testing of error handling code

Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems, Yuan et al. USENIX OSDI 2014

How hard it to improve?

In 35% of the cases, the error handling code fall into one of three patterns:

Overreacted, aborting the system under non-fatal errors

Was empty or only contained a log printing statement

Contained expressions like “FIXME” or “TODO” in the comments.

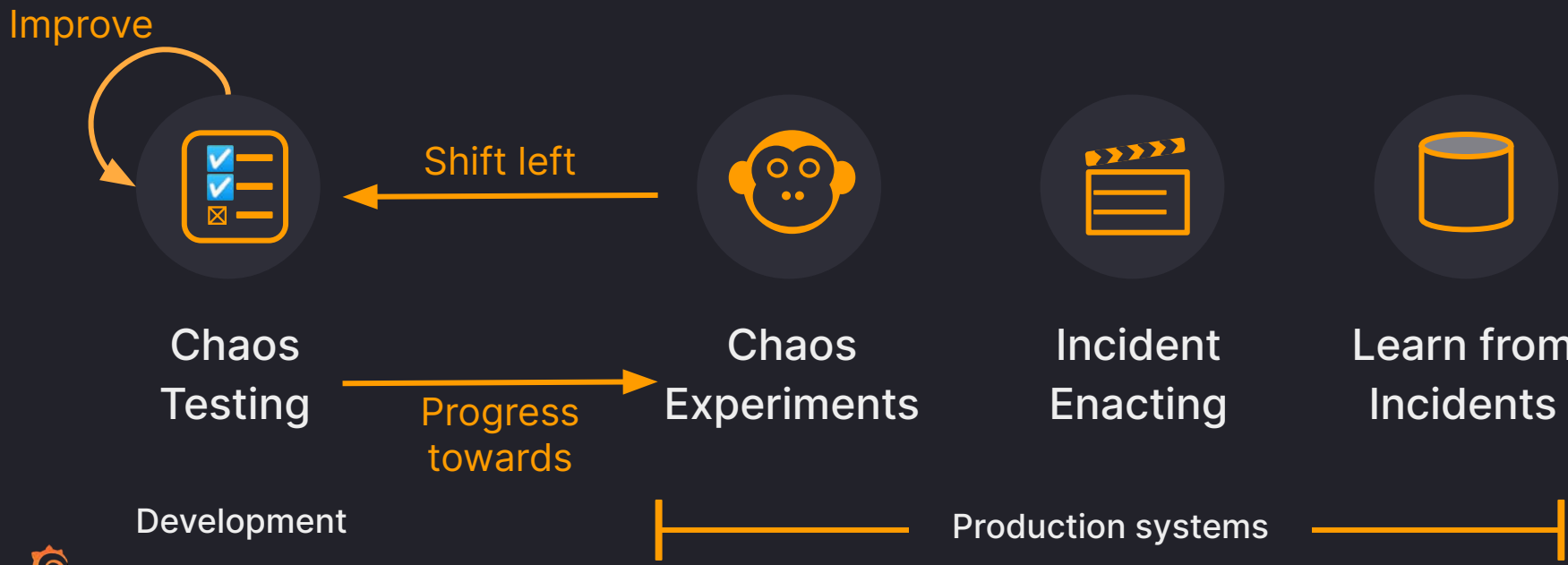
Chaos Testing

Incorporate the principles of chaos engineering early into the development process as an integral part of the testing practices

Shifting the emphasis from experimentation to verification

From uncovering unknowns faults to ensuring proper handling of known faults

Continuous Reliability Improvement



The four tenets of Chaos Testing

Incremental adoption

Application Centric

Chaos as Code

Controlled Chaos

Chaos Testing in action

OFFER OF THE DAY Buy 10 socks, get a pet human for free!

Login

weaveworks socks

HOME CATALOGUE

0 items in cart

WE LOVE SOCKS!

Fun fact: Socks were invented by woolly

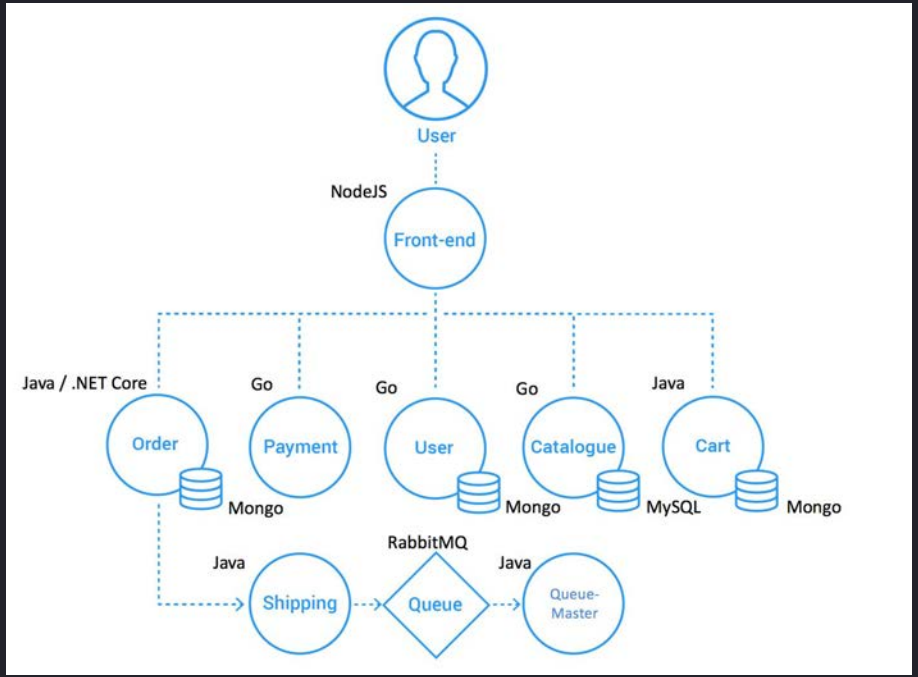
BEST PRICES

We price check our socks with trained monkeys

100% SATISFACTION GUARANTEED

Sock Shop application

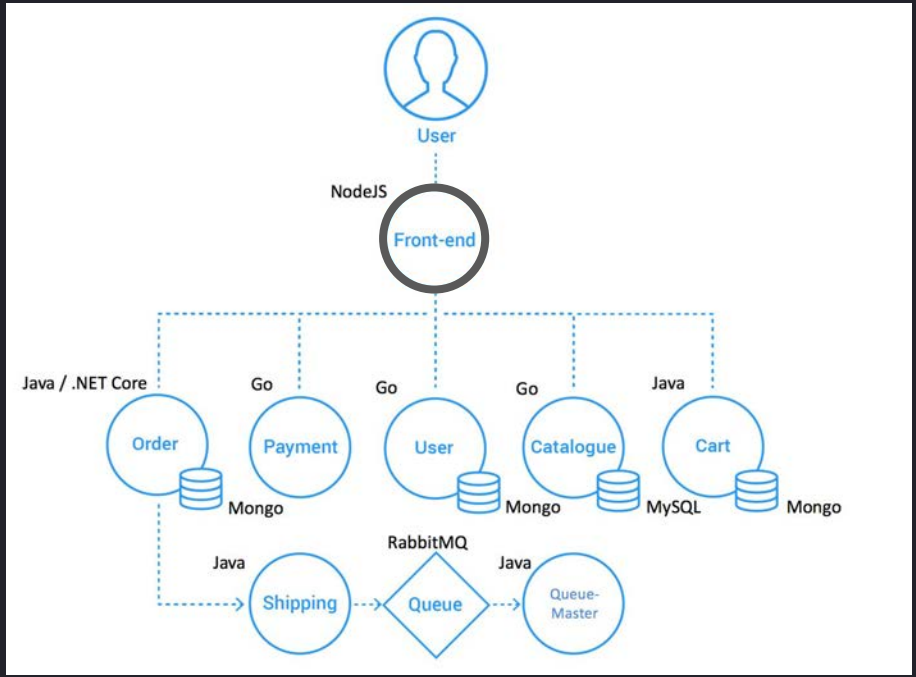
- Microservices architecture
- Http-based communication between services
- Polyglot (Go, Java, JS, ...)
- K8s-ready



https://github.com/microservices-demo/microservices-demo

Sock Shop application

- Microservices architecture
- Http-based communication between services
- Polyglot (Go, Java, JS, ...)
- K8s-ready



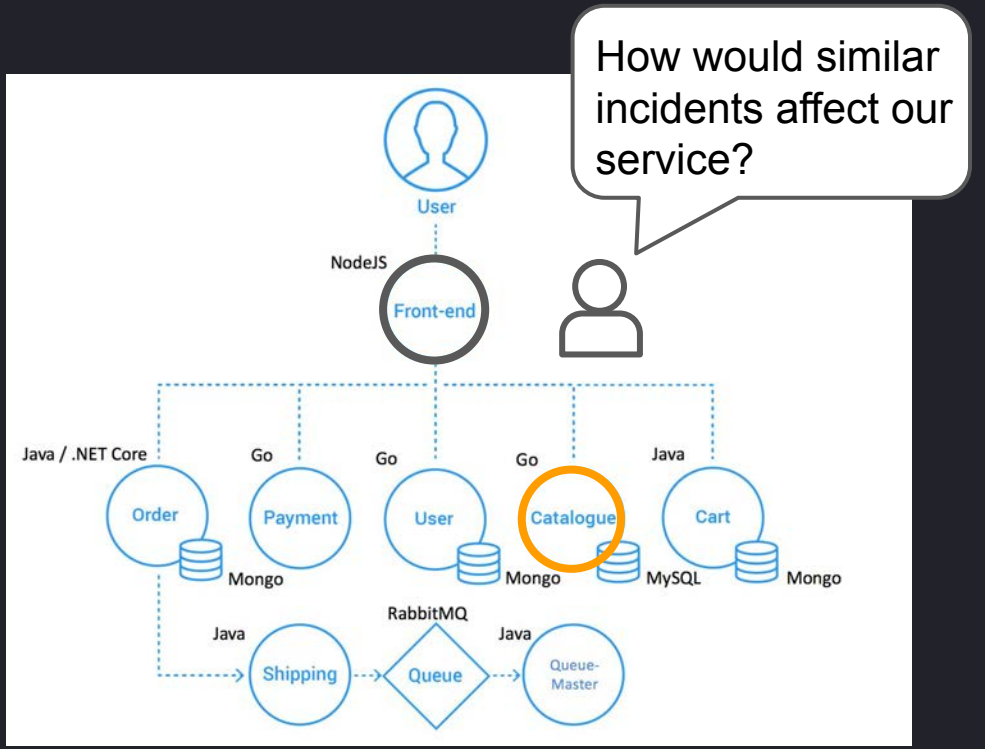
https://github.com/microservices-demo/microservices-demo

Fictitious Post Incident Review

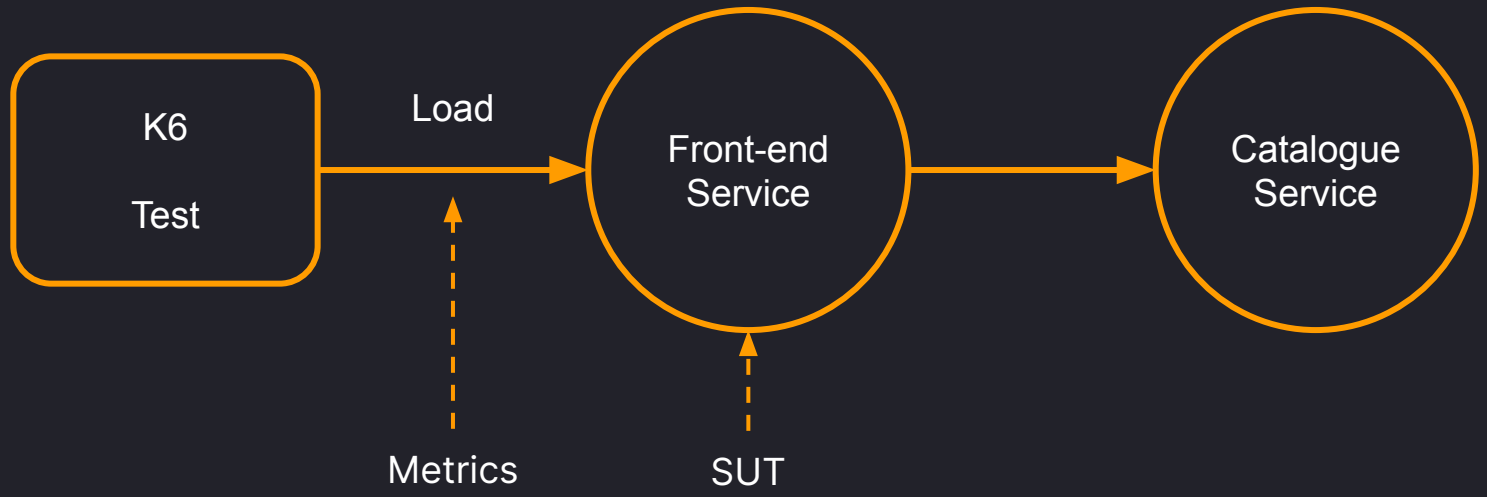
Long running queries in the **Catalogue service**'s DB caused **delays** in the requests the exhaustion of DB sessions that resulted in **errors** (HTTP 500)

Catalogue service team will investigate the incident to address the root cause

However, the front-end team wonders...



Let’s start with a load test for the Front-end service



k6.io

Open source reliability testing tool

Programmable tests using Javascript

Covers different testing needs: load, end-to-end, synthetic, chaos

Can send test results to common backends such as prometheus

Extensible using a growing catalog of extension (e.g. Kafka, Redis, K8s, Sql...)

Key concepts

Functions

Simulated user flow

Check

Response Validations

Scenario

Workload model

Thresholds

SLOs

Load test for the Front-end service

```
export function requestProduct() {
   const item = products[Math.floor(Math.random()
                * products.length)];
   const resp = http.get(`${url}/${item}`);
   const body = JSON.parse(resp.body);
   check(body, {
       'No errors': (body) => !('error' in body)
   });
}
```

This function makes a request to the front-end service

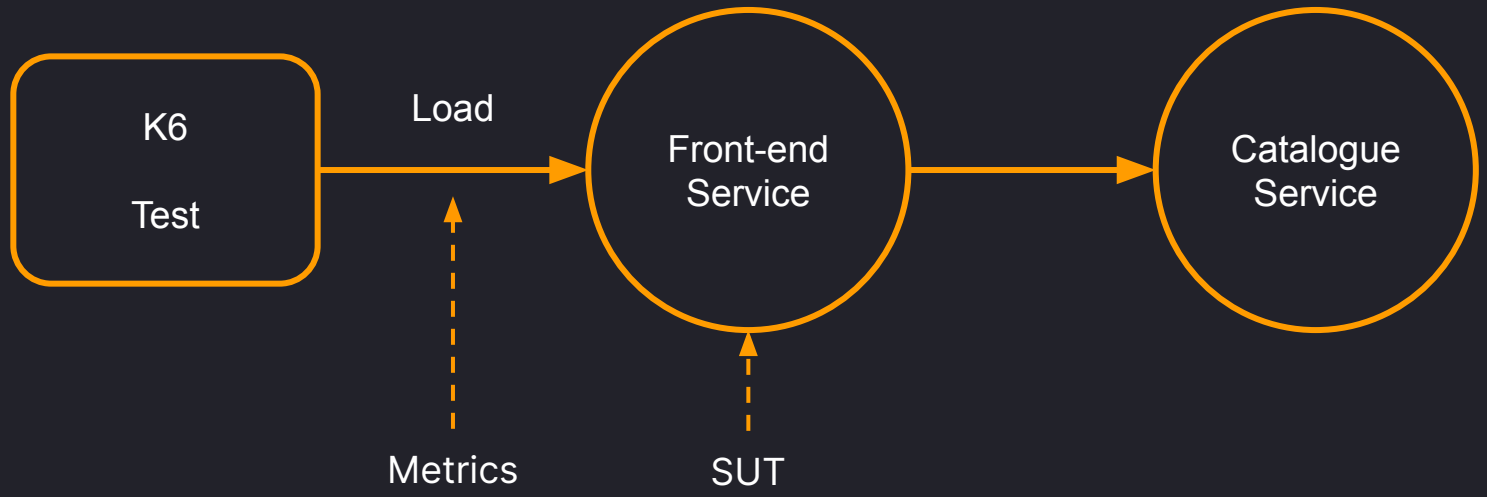
Also checks for errors in the response

```
export const options = {
    scenarios: {
        load: {
            executor: 'constant-arrival-rate',
            rate:   20,
            preAllocatedVUs: 5,
            maxVUs: 100,
            exec: 'requestProduct',
            startTime: '0s',
            duration: '60s',
        }
    }
}
```

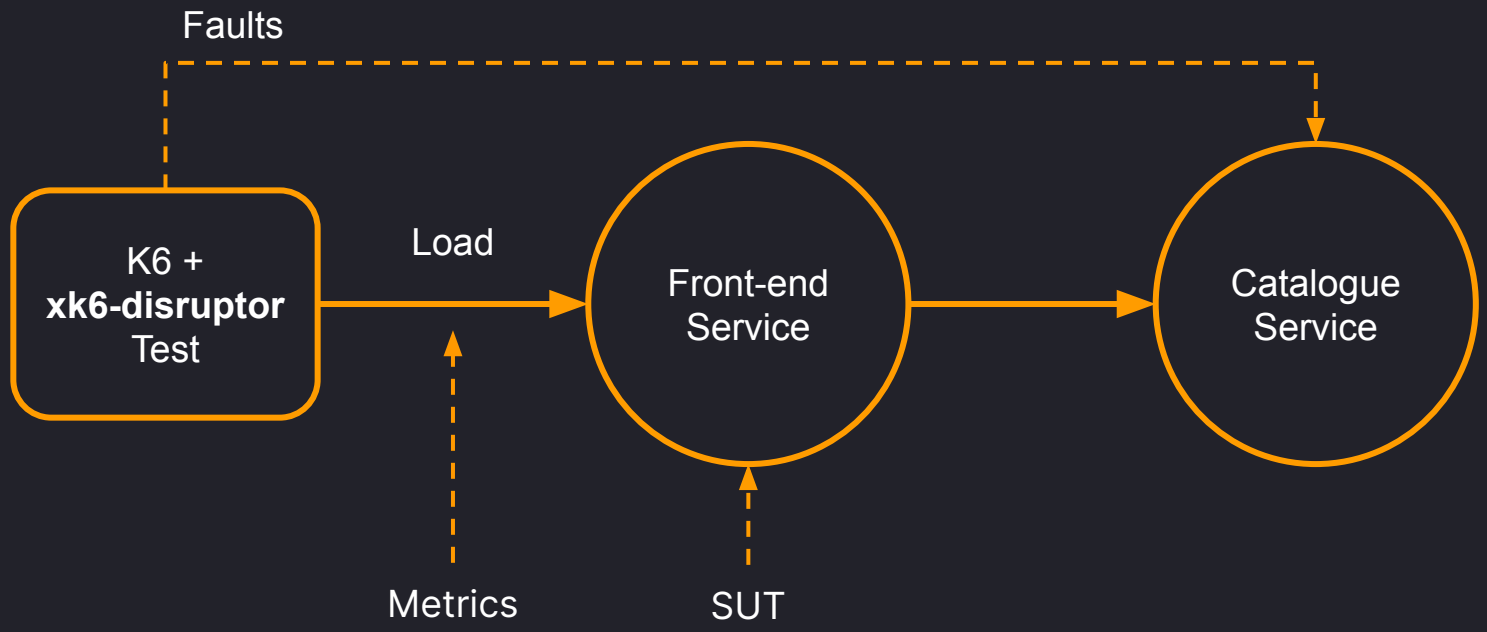
Here is where we apply load

Let’s run this test and check performance metrics ...

Now, let’s add some chaos to this test



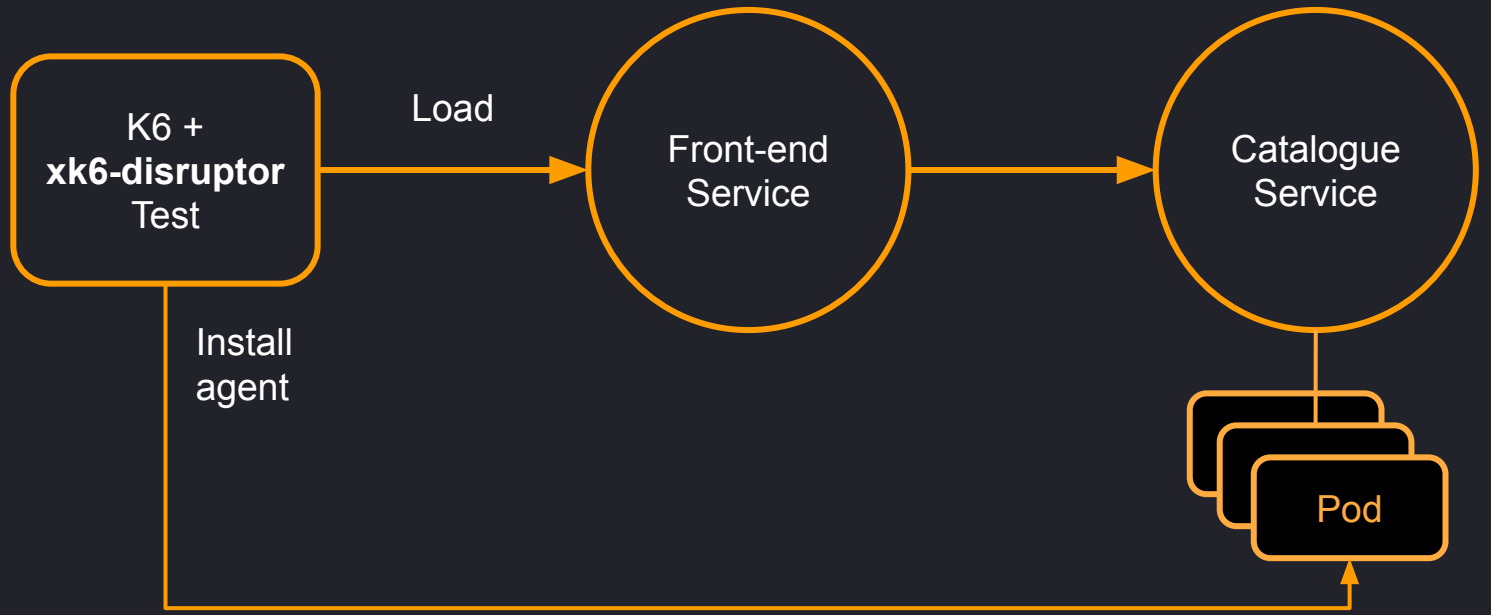
Let’s add some chaos to this test



xk6-disruptor

An extension that adds **fault injection** capabilities to Grafana **k6**

How the disruptor works



Chaos test for the Front-end service

```
export function requestProduct() {
   const item = products[Math.floor(Math.random()
                * products.length)];
   const resp = http.get(`${url}/${item}`);
   const body = JSON.parse(resp.body);
   check(body, {
       'No errors': (body) => !('error' in body)
   });
}

export function injectFaults(data) {
    const fault = {
       averageDelay: 100,
       errorRate: 0.1,
       errorCode: 500,
    };

    const disruptor = new ServiceDisruptor(
        'catalogue',
        'sock-shop'
    );
    disruptor.injectHTTPFaults(fault, 60);
}
```

This function injects faults

Fault definition

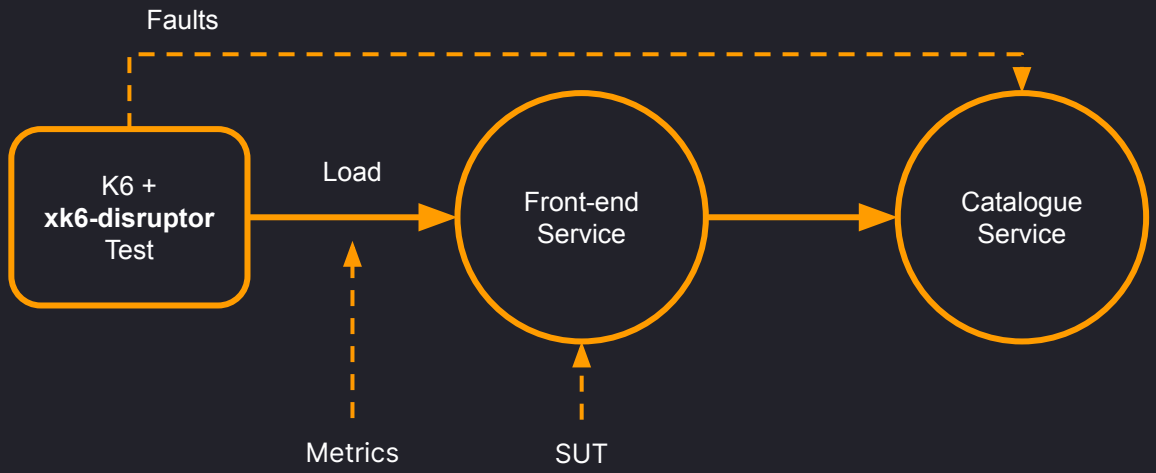
Select target service

Inject fault

```
export const options = {
    scenarios: {
        load: {
          executor: 'constant-arrival-rate',
          rate:   20,
          preAllocatedVUs: 5,
          maxVUs: 100,
          exec: 'requestProduct',
          startTime: '0s',
          duration: '60s',
        },
        fault: {
           executor: 'shared-iterations',
           iterations: 1,
           vus: 1,
           exec: 'injectFaults',
           startTime: '0s',
         }
    }
}
```

Here we add the fault injection to the test

Let’s run this chaos test...



How this test helps the front-end team?

Uncover improper error handling logic

Validate different solutions until obtaining an acceptable error rate

Fine-tune the solution and avoid issues such as retry storms

Chaos testing principles in action

- A load or functional test can be reused to test the system under turbulent conditions
- These conditions are defined in terms that are familiar to developers: latency and error rate
- The test has a controlled effect on the target service
- The test is repeatable and the results are predictable
- The fault injection is coordinated from the test code
- The fault injection does not add any operational complexity

Final remarks

The ability to operate reliably should not be a privilege of the technology elite

Chaos Engineering can be democratized by promoting the adoption of Chaos Testing

To be effective, Chaos Testing must be compatible with the existing testing practices used by development teams

Our Goal

Make Chaos Engineering practices accessible to a broad spectrum of organizations by building a solid foundation from which they can progress towards more reliable applications.

Thank you for attending!

Additional resources

- xk6-disruptor project

  https://github.com/grafana/xk6-disruptor

- xk6-disruptor documentation

  https://k6.io/docs/javascript-api/xk6-disruptor

- Chaos testing microservices with xk6-disruptor

  https://k6.io/blog/chaos-testing-microservices-with-xk6-disruptor

Grafana Labs

k6

Shifting Left Chaos Testing

Pablo Chacin
Chaos Engineering Lead @ k6
Grafana Labs