Resilience Testing of Spring Applications via Fault Injections

Camilo Velázquez-Rodríguez[1,*], Warren Gaure[2] and Coen De Roover[1]

[1]Vrije Universiteit Brussel, Brussels, Belgium
[2]CESI Graduate School of Engineering, Saint-Nazaire, France

Abstract

Nowadays, there exist multiple forms of testing such as unit tests, integration tests, etc. Another form of testing is *resilience testing*, which tests the endurance of a target system by injecting faults and observing the system's behaviour. Injected faults such as aborting messages or bringing down services aim to dramatically perturb the target system. This paper proposes a tool for resilience testing of microservices in Spring applications. Our tool takes as input a set of manually written scenarios to inject abort and delay faults between different microservices in a target application. After parsing the written scenarios, our tool identifies the subset of tests in the target application that use the Spring REST API. As a microservice application might consist of multiple modules or services, our tool also scans all modules to later execute the selected application tests properly. Finally, we inject faults into microservices and observe the system's behaviour. We demonstrate the effectiveness of our tool by applying it to an open-source Spring project.

Keywords

Microservices, Fault Injection, Resilience Testing, Spring apps

1. Introduction

Testing is the process of executing a program with the intent of finding errors [1]. A program can be tested with respect to multiple factors. Consequently, there are multiple forms of software testing such as unit, integration, resilience tests etc. Resilience testing focuses on verifying how resilient a software system is to chaotic or unexpected inputs. Resilience tests perturb the system under analysis and check its reaction to such perturbation [2, 3, 4].

In recent years, microservice applications have grown more popular. Major organizations such as Netflix, Amazon or Uber have adopted an architecture based on microservices. Advantages over other types of architectures (e.g., monolithic) include faster prototyping and ease of maintenance. However, not only has the architecture changed, but also the way of performing tests. In addition to traditional functionality tests, other elements such as message delivery/interruption are also accountable in microservices.

Resilience testing can be applied to microservices by interrupting the communication channels of a system and observing its response. This perturbation might include abruptly taking services offline, changes to the values of messages between services, sudden drop of messages, etc. For example, Chaos Monkey [4] is a Netflix tool that verifies the effects of terminating cloud resources on their services. FATE and DESTINI [5] tests resilience against disk and network failures, and Chaokka [3] automatically injects faults into Akka actors. The term *Chaos Engineering* [6] has been used to refer collectively to the techniques applied in these studies.

Chaos Engineering is a general approach that aims to identify failures in programs before system outages [6]. This identification is performed by perturbing the normal flow of a program through faults or unexpected inputs. Although the term *Chaos Engineering* has only recently been formalised, many popular companies have in the past adopted practices akin to it. For example, Amazon [7], Google

BENEVOL24: The 23rd Belgium-Netherlands Software Evolution Workshop, November 21–22, 2024, Namur, Belgium
*Corresponding author.
camilo.ernesto.velazquez.rodriguez@vub.be (C. Velázquez-Rodríguez); warren.gaure@viacesi.fr (W. Gaure); coen.de.roover@vub.be (C. De Roover)

[8] and Netflix [9, 4] have developed techniques to purposely inject failures into their systems. By forcefully pushing their products to the extreme, they are able to discover potential errors and resilient faults to could ultimately improve their application quality.

Gremlin [2] tests the capabilities of microservices by perturbing messages between services at the network layer. Gremlin testers specify failure scenarios and the corresponding recovery observations manually through *recipes*. These recipes are parsed by Gremlin and the specified services perturbed with the requested failure. Gremlin's approach defines three types of primitive faults: *abort*, *delay* and *modify*. More complex faults such as *overload* are the result of combining previous primitive faults.

Despite Chaos Engineering and resilience testing practices being active fields for some years now, there are insufficient implementations supporting microservice projects. For example, Spring [10] is one of the most used Java frameworks worldwide. However, we are not aware of a resilience testing tool targeting Spring projects. Resilience testing applied to Spring applications can uncover a lack of best coding practices in order to handle perturbation errors.

This paper proposes a tool for checking the resilience of microservices in Spring applications. Our tool is inspired by the previous Gremlin work [2] as it is also based on a manual specification of which services have to be perturbed. However, we target Spring microservice applications, which often consist of multiple modules implemented as services. Additionally, we leverage aspect-oriented programming instead of network-oriented emulation, like the Gremlin tool does. Specifically we support our tool on the *AspectJ* framework to identify services at run time and test them.

Our tool aims to be zero-touch and not change the code of the application, or the tests. Instead, our tool injects faults in the runtime behaviour of the application using the tests as entry point for usages. We developed a fault injector prototype that can be included as external dependency of Spring projects. This prototype injects faults when the Spring REST API is tested and checks the system's behaviour afterwards.

This paper makes the following contributions:

- A tool to inject faults into Spring microservice applications. Based on manually written recipes our tool makes use of aspect-oriented programming and *AspectJ* to extract the information related to microservices at run time. Once services have been discovered, faults are injected in those services specified in the recipes.
- A case study in an open-source GitHub project that makes use of the Spring framework. We apply our prototype tool to the selected open-source project and perturb some messages in its microservices. We observe the results from the fault injection and how the systems reacts to such perturbation and report on them.

The remaining sections in the paper are structured as follows: Section 2 describes our tool for designing the fault injection prototype in order to test microservice Spring applications. Section 3 presents the case study of the application of our tool to an open-source GitHub project. Section 4 discusses limitations of our current tool prototype and proposes features to be incorporated in the future. Section 5 briefly visits previous works related to our described tool. Lastly, Section 6 concludes the paper.

2. Injecting Faults to Test Spring Microservices

This section describes our tool to inject faults and test the resilience of Spring microservices applications. Figure 1 shows the outline of our tool, where letters denote data nodes, and numbers denote action nodes.

The broad idea of our tool is to inject faults in the communication channels of Spring microservices and then observe how the system reacts. In general, the fault injection in our tool depends on the usages of the Spring REST API. These usages can be found for example, in the test suite of the target system (data node *A*). Therefore, our approach assumes that the target system has tests. These tests

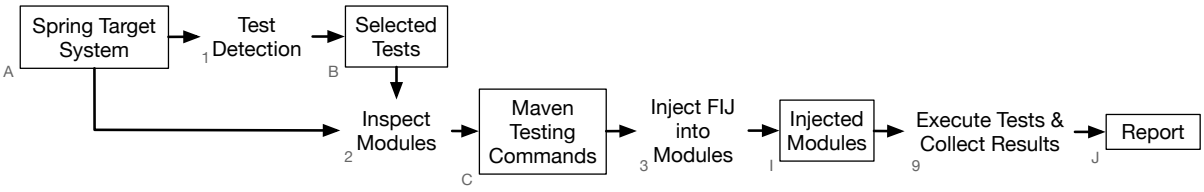

Figure 1: Fault-injection tool to test the resilience of Spring microservice applications.

could be of any type (e.g., unit tests, end-to-end, etc.), as our approach is test-agnostic. Our tool detects tests that make use of the Spring REST API and selects them for posterior execution.

The detection and selection of tests (action node *1* and data node *B*) has a major advantage over executing the full test suite. A reduced number of tests makes their execution faster in comparison to the full suite. Many projects are known for having a large test suite which can take hours or even days to complete.

Test detection (action node *1*) is performed using the *classgraph* [11] library for Java. This library allows to traverse a compiled Java project (e.g., *.class* files) looking for patterns such as classes inheriting a given class or implementing a given interface, etc. In our case, we aim to detect test classes using the *@SpringBootTest* annotation which is commonly used when Spring functionalities are tested. Listing 1 shows a code example of a test class with the *@SpringBootTest* annotation.

```
1 @SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
2 public class ActionControllerKitTests {
3     private RestTemplate restTemplate;
4     private int port;
5     ...
6 }
```

Listing 1: Example of a Java test class with the *@SpringBootTest* annotation.

The *@SpringBootTest* annotation provides the necessary environment to test the Spring REST API. We noticed the usage of this annotation across many projects on GitHub. Nonetheless, other structures (i.e., annotations, classes, methods, etc.) might also be used. We selected test classes where the *@SpringBootTest* annotation is used (data node *B*). Once tests are selected, the final goal is to execute them and check whether the system is resilient after the injection of the faults. All selected tests for our case study (cf. Section 3) successfully passed before fault injection.

In a multimodule target system, i.e., a Java project consisting of multiple modules, test execution is not as simple as in a single-module system as each module has its own test suite. Therefore, our tool inspects each module in the target (action node *2*) to extract its full path from the root of the project (i.e., the root parent of all modules). Once the location of the modules containing the test classes has been found, our tool writes a testing command to an external file (data node *C*). The Maven build tool (i.e., *mvn*) includes an option (*-pl or --projects <arg>*) to specify the path of the module/project path to build. Whenever the path of a module is specified, all tests in this module can be executed. Listing 2 shows how a test inside an internal module that can be successfully executed from the root of the target system.

```
1 mvn -pl src/apps/geoserver/wcs test -Dtest="org.geoserver.wcs.WcsApplicationTest"
```

Listing 2: Command to execute a test located in an internal module of a multimodule Maven project.

Once all testing commands for the selected tests have been generated, faults are injected into the modules where the tests are located (action node *3*). For the first prototype of our tool, we aim to inject faults such as message aborts of message delays. Future versions of our tool will incorporate *modify* messages as Gremlin [2], and also more complex types of faults. In order to inject faults, and to execute the action node *3*, our tool includes a Fault Injector Jar (*FIJ*) as part of the dependencies of the selected modules.

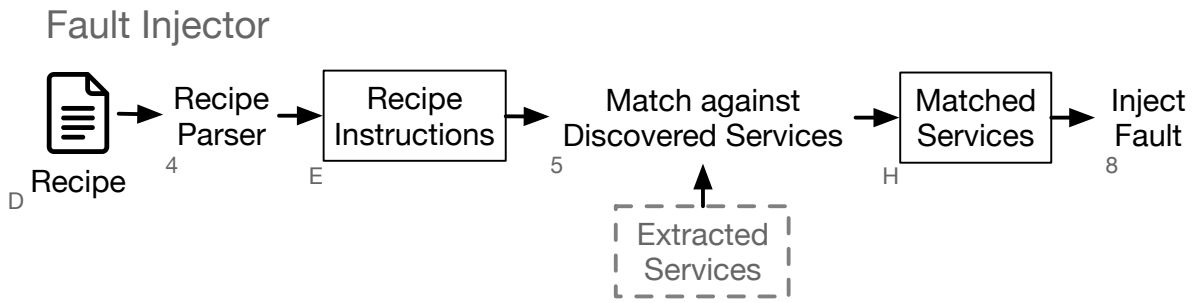

Figure 2: Fault-injection prototype that injects faults into microservices. Extracted Services are described in the next figure.

Figure 2 describes the Fault Injector tool that is included as a dependency of the modules with selected tests. This tool was compiled into a JAR (Java ARchive) file for posterior use (cf. Figure 1). Specifically, the Fault Injector in Figure 2 receives as input a recipe (data node *D*) indicating the type of fault to inject, the endpoint of the source service, the requesting method of the service (e.g., GET, POST), etc.

```
"recipes": {
    "Abort": {
        "Dst": "/ping",
        "Method": "POST"
    }
}
```

Listing 3: An example recipe to abort a message for a service with the specified method.

For example, the recipe in Listing 3 details that a message abort should be injected in the specified service that sends POST requests. All the fields in the recipe should be specified by the user, which includes the type of faults to be injected. Our current prototype processes a single recipe at a time. However, we envision future iterations to support multiple recipe processing in parallel. The sender of these microservices should have an endpoint equals to `/ping`. Likewise, a delay fault includes another field in the *JSON* recipe specifying the time to delay the exchange of messages as shown in Listing 4.

```
"recipes": {
    "Delay": {
        "Dst": "/pong",
        "Method": "GET"
        "Time": 5000
    }
}
```

Listing 4: An example recipe to delay a message for a service with the specified method.

We implemented a recipe parser (action node *4*) in the fault injector supporting the previous fields. After parsing the recipe (data node *E*), the injector proceeds to extracting the provided services by the

application (action node *5*). We use aspect-oriented programming (AOP) [12] to extract such system's services at run time whenever a test makes a REST API call. Specifically, we leverage the *AspectJ* library [13] for Java. AspectJ extends the latter with new constructs that provide means to identify crosscutting concerns. Examples of such constructs include *joint points*, *pointcuts* and *advices*[12].

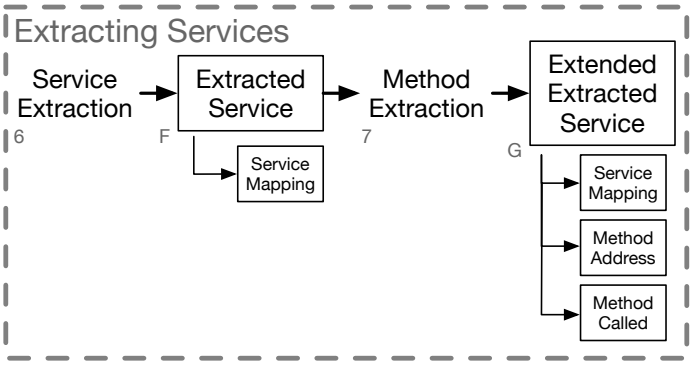

Figure 3: Tool for extracting services from a target system using AspectJ.

Figure 3 shows the tool for extracting services via the AspectJ library. Our tool first extracts all *service mappings* (action node *6* and data node *F*). A service mapping specifies which Java method should be invoked when a request is sent to an endpoint URL, such as `/ping`. In case a single service is specified by the recipe (e.g., Listings 3 and 4), the fault will be immediately injected in this service. In case two services are specified by the recipe, additional steps are required.

Our tool first inspects method calls to the Spring REST API from within the mapped methods. This is performed in the action node *7* and its goal is to establish possible communication channels between two microservices. Additionally, our tool also extracts internal requests from within the mapped method (data node *G*) and establishes the connection between services. Listing 5 shows the pointcut expressions used to capture the information in Figure 3. Pointcut expressions are manually coded to match the Spring REST API method calls. In case the pointcut expressions are matched, a fault is injected into the system.

```
1 @Pointcut("execution(@org.springframework.web.bind.annotation.RequestMapping * *(..))
  ↪  && @annotation(mapping)")
2
3 @Pointcut("call(public * org.springframework.web.client.RestTemplate.getFor*(..)) " +
4           "|| call(public * org.springframework.web.client.RestTemplate.postFor*(..))
            ↪  " +
5           "|| call(public * org.springframework.web.client.RestTemplate.put(..)) " +
6           "&& within(@org.springframework.web.bind.annotation.RequestMapping *)")
```

Listing 5: Pointcut expressions used by our tool to extract information related to microservices.

The last part of the fault injector tool (cf. Figure 2) consists of two steps. In the first step, extracted services (data node *H*) are compared with those specified in the *JSON* recipe. In case of a match, the fault injector proceeds to inject the fault (action node *8*). *Abort* faults will interrupt the message being transmitted. In other words, whenever there is a match our tool triggers an exception to interrupt the message flow. In case the system has adopted a resilience practice, it should retry the aborted message and continue normally.

Delay faults pause a message from being sent for a specified period. Delay faults could affect services that expect a certain message being received at a specific time. In this case, when our tool finds a match in the recipe it will trigger a `Thread.sleep` method call with the specified time in milliseconds. The current version of our tool injects faults only once to observe how the system handles them. However, we aim to inject multiple faults in future versions of our prototype.

The Fault Injector tool needs to be added as external dependency to the modules where the selected tests are located. Including the FIJ as an external dependency must be done in the modules' *pom.xml* in Figure 1. Once the dependency is included in a module (data node *I*), its selected tests are executed (action node *9*), which will immediately trigger the fault injector tool. AspectJ's aspects will process the information in the tests (cf. Figures 2 and 3) and will perturb them in case of a match with the specification in the recipe. Lastly, the test results are collected, and a final report is created that lists which tests failed and which tests passed in data node *J*. Test failures are indicative of a lack of resilience from the target system for a specific service perturbation.

3. Case Study

This section details how our tool was applied to an open-source project hosted on GitHub.

3.1. Project Description

The project, *CrowdDetector* [14], detects crowds to prevent the spread of the COVID-19 virus. *CrowdDetector* has two main components. A frontend, implemented in JavaScript, and a backend, implemented in Java. Figure 4 shows a screenshot of the application.

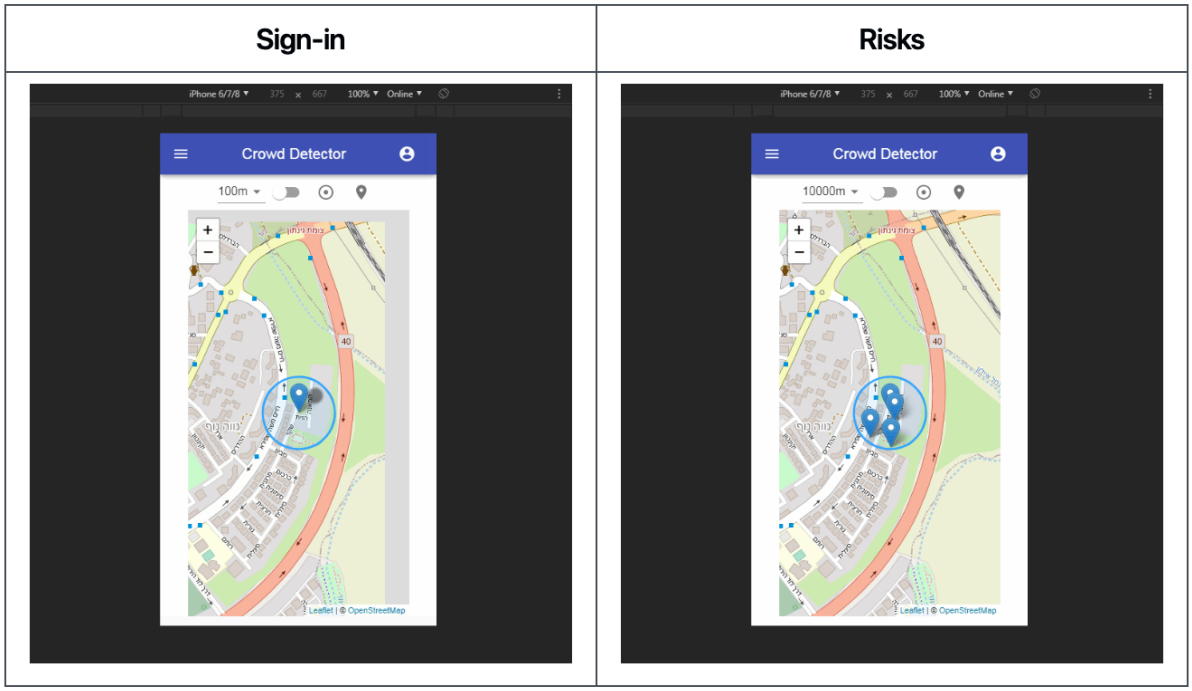

Figure 4: A screenshot of the CrowdDetector application.

The backend defines four services and controllers for `users`, `admin actions`, `elements`, and `user actions`. Each of these services has its own group of tests to verify their functionality. For example, the tests of the action services verify that created users are able to fetch information from the application. Such users have random locations defined by their latitude and longitude coordinates.

3.2. Configuring Target Project

We cloned the GitHub repository, manually compiled the code, and ran the tests using the Maven build tool. The test suite contains 23 tests, all of which were successfully executed. Therefore, we started with the setup of our tool, as previously explained in Section 2. First, we extended *CrowdDetector*'s

dependencies (i.e., its *pom.xml* file) to include not only our tool as an external library, but to include AspectJ's tools and runtime libraries [15, 16] as well.

An additional plugin is required [17] to modify the compilation and test processes. This is necessary because they must include the custom *AspectJ* aspects of our implementation. Consequently, once the tests have been executed, the aspects defined in our tool will also start matching pointcut expressions.

As mentioned in Section 2, our tool detects the tests featuring the *@SpringBootTest* annotation. These tests might indicate that the Spring REST API is being evaluated. All four test classes in the *CrowdDetector* application use this annotation. This open-source project is not multimodule, so it is not necessary to analyse multiple internal modules. Selected tests in *CrowdDetector* can be executed directly from the root of the project.

3.3. Fault Injection and Results Collection

As previously explained in Section 2, our tool is based on manually-written recipes, similar to Gremlin [2]. This requires at least some initial knowledge of the application's REST API endpoints. After inspecting the endpoints of *CrowdDetector* we selected one service to be injected with faults. The recipe for the target service is displayed in Listing 6.

```
"recipes": {
    "Abort": {
        "Dst": "/acs/users/login",
        "Method": "GET"
    }
}
```

Listing 6: Recipe to abort the messages from the specified service and method.

The recipe in Listing 6 specifies that at least one message to `/acs/users/login` requested with the `GET` method should be aborted. When the aspects in our fault injector match a REST API call with this pattern, an interruption of the message will be triggered. Our prototype implementation currently handles a single fault. Subsequent versions of our prototype will incorporate more faults in the recipes to be injected. If the system has correctly implemented some form of resilience mechanism, then injected faults will have no effect on the application as the affected services will recover from the faults. On the other hand, a lack of resilience practices makes the system vulnerable to injected faults. These faults might bring the system down, affecting its availability and as final result the access from their users.

Once previous steps have been set up, our tool executes the selected tests in the target project, to check the resilience of *CrowdDetector* to the injection of faults.

Figure 5 shows the results of initially running the selected tests before injecting faults. As observed in the figure, all tests from the selected classes are successfully executed. Once we inject the fault in Listing 6, we observe how the system reacts.

The results of the tests after the injection of the faults are shown in Figure 6. All test classes failed in their execution after injecting a fault to the service specified in Listing 6. Specifically, aborting a message from endpoints that start with `/acs/users/login` is deemed critical by the system. This service is responsible for login actions in the system. All selected tests access this service, which gets interrupted by our tool. This is reflected in the middle of Figure 6 where an aborted message is displayed.

4. Discussion

Previous results in Section 3 indicate that the open-source project *CrowdDetector* does not implement resilience practices. Adopting resilience practices could have prevented fault propagation in the system, making it *always* available [18]. Nowadays, there are many libraries that could assist the development

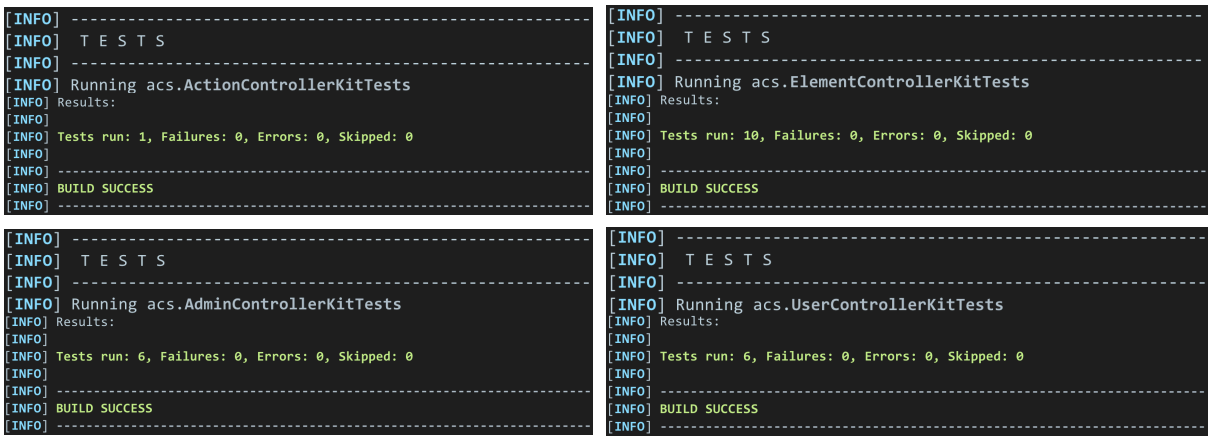

Figure 5: Results of the initial test execution of the selected tests before faults are injected.

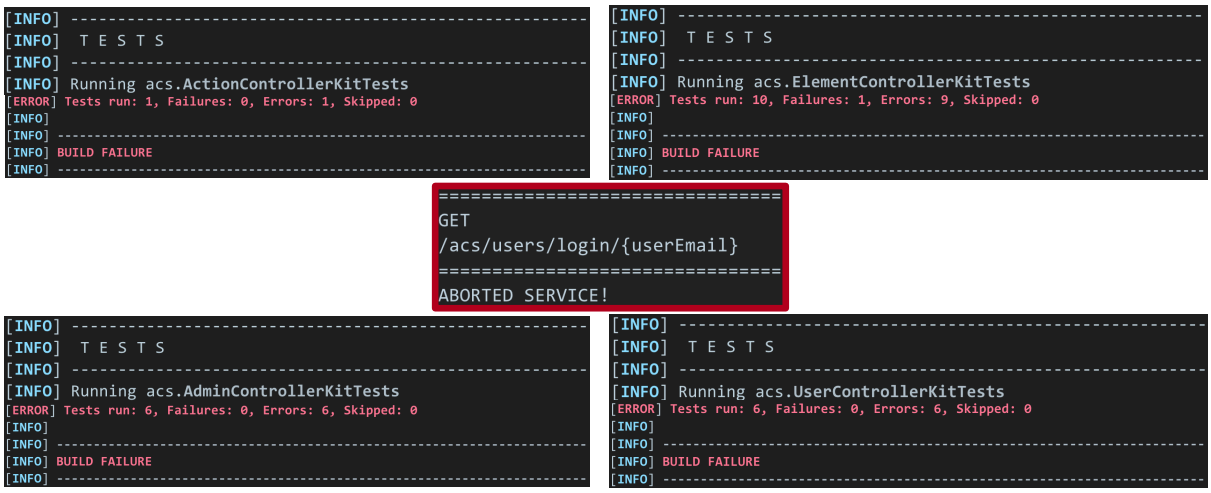

Figure 6: Results of the test execution of the selected tests after the faults are injected.

of more resilient applications. For example, *resilience4j* [19] is a library that provides easy access to fault tolerance patterns in the form of annotations. In particular, for the project *CrowdDetector*, the *@Retry* annotation could have been used to prevent system errors when faults are injected.

Our tool is currently a work in progress. Therefore, there are various limitations that still need to be addressed. For example, the implementation is still focused on a particular set of classes of the Spring REST API. Additionally, many test classes make use of mockups to quickly interact in isolation with some parts of the system (e.g., databases). Our tool still lacks support for mocking. Lastly, we target the Spring framework because of the large number of applications that use it for microservices. Our tool is therefore currently very specific to Spring.

These limitations suggest avenues of future research, such as including support for mockups and for domain objects in test classes. In general, our tool could cover other microservice frameworks (e.g., Akka [20]), by target the communication channels of these frameworks, rather than the specific details of their internal workings. This will render our tool more complicated, but it will make it more applicable to additional projects.

5. Related Work

The following section explores related works to our described tool in Section 2.

Gunawi et al. [5] propose FATE, a failure testing service and DESTINI, declarative testing specifications for resiliently test cloud services. FATE injects cloud systems failure scenarios while DESTINI is a Datalog extension that enables users to specify what behaviour is expected of the application after injecting a fault. Both are developed in a single framework for cloud recovery testing.

Chaos Monkey [4] is a Netflix Chaos Engineering tool that verifies whether a network is resilient to systematic failure injection. It consists of three major steps: defining failure scenarios, injecting failures and checking network invariants. The final step checks for changes to the network topology, etc.

Gremlin [2] perturbs inter-service messages at the network layer based on a manually written recipe system. Once the services to perturb are specified, faults are injected in their communication channel and Gremlin checks their responses. The expected behaviour of these responses is also manually specified in the recipe.

Chaokka [3] is an automated approach to test the resilience of actors in the Akka framework. The approach uses test amplification to improve existing test cases, and delta debugging to decide where to inject faults. The novelty of Chaokka lies precisely in the combination of both techniques as it leverages the hypothesis that failures might not be located in paths that are explored when exercising the system's most important features [21]. Their results indicate that the delta debugging technique is faster than random exploration of test paths for resilience testing.

Our tool is influenced by Gremlin [2] as we also adopted a recipe-based specification. However, our tool relies on aspect-oriented programming to discover services. This provides the advantage that we are able to develop a zero-touch approach where the code of the target application is not touched. For example, our tool only changes the behaviour of the target application at runtime when the tests are executed. Additionally, our target applications are Spring microservice projects which, at the moment, cannot be analysed by any of the previous works.

6. Conclusion

This paper presents a tool for injecting faults into microservice Spring applications. Our tool processes manually-written recipes that specify which services to inject and with which specific type of faults. The prototype implementation of our tool supports two types of faults: abort and delay messages in services. Aspect-oriented programming and its concepts (e.g., aspects, pointcuts, etc.) are fundamental for the injection of faults in our implementation.

We applied our tool to an open-source GitHub project which makes use of the Spring REST API. After including our prototype as external dependency, the results show that our prototype is capable of discovering resilience errors in the application. Despite being an initial prototype, our tool is able to successfully inject faults into Spring microservice projects. Our tool makes it possible to identify errors due to the lack of implementation of resilience practices. Finally, we described the major limitations and future plans for our tool.

Acknowledgments

The authors are partially funded by the FWO SBO BaseCamp Zero project (Code: S000323N).

References

[1] G. J. Myers, C. Sandler, T. Badgett, The Art of Software Testing, John Wiley & Sons, 2011.

[2] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, V. Sekar, Gremlin: Systematic Resilience Testing of Microservices, in: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), IEEE, 2016, pp. 57–66.

[3] J. De Bleser, D. Di Nucci, C. De Roover, A Delta-Debugging Approach to Assessing the Resilience of Actor Programs through Run-time Test Perturbations, in: Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test, 2020, pp. 21–30.

[4] M. A. Chang, B. Tschaen, T. Benson, L. Vanbever, Chaos Monkey: Increasing SDN Reliability through Systematic Network Destruction, in: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, 2015, pp. 371–372.
[5] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, D. Borthakur, FATE and DESTINI: A framework for cloud recovery testing, in: 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11), 2011.
[6] C. Rosenthal, N. Jones, Chaos Engineering: System Resiliency in Practice, O'Reilly Media, 2020.
[7] Amazon Game Day, https://wa.aws.amazon.com/wat.concept.gameday.en.html, 2015. [Online; accessed 13-11-2024].
[8] K. Krishnan, 10 Years of Crashing Google, USENIX Association, Washington, D.C., 2015.
[9] The Netflix Simian Army, https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116o, 2011. [Online; accessed 13-11-2024].
[10] The Spring framework website, https://spring.io, 2024. [Online; accessed 18-09-2024].
[11] A Java classpath scanner and module scanner, https://github.com/classgraph/classgraph, 2024. [Online; accessed 18-09-2024].
[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-Oriented Programming, in: ECOOP'97—Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings 11, Springer, 1997, pp. 220–242.
[13] The AspectJ website, https://eclipse.dev/aspectj/doc/latest/index.html, 2024. [Online; accessed 18-09-2024].
[14] COVID-19 Crowd-Detection tool, https://github.com/AvivShabtay/CrowdDetector, 2020. [Online; accessed 18-09-2024].
[15] Maven Central Repository - AspectJ Tools, https://mvnrepository.com/artifact/org.aspectj/aspectjtools, 2024. [Online; accessed 18-09-2024].
[16] Maven Central Repository - AspectJ Runtime, https://mvnrepository.com/artifact/org.aspectj/aspectjrt, 2024. [Online; accessed 18-09-2024].
[17] Maven Central Repository - AspectJ Maven Plugin, https://mvnrepository.com/artifact/org.codehaus.mojo/aspectj-maven-plugin, 2024. [Online; accessed 18-09-2024].
[18] K. Merker, What is Five 9s Availability? Do you really need 99.999% Server Uptime?, https://www.nobl9.com/resources/do-you-really-need-five-nines, 2020. [Online; accessed 18-09-2024].
[19] GitHub repository resilience4j, https://github.com/resilience4j/resilience4j, 2024. [Online; accessed 18-09-2024].
[20] The Akka framework website, https://akka.io, 2024. [Online; accessed 18-09-2024].
[21] D. Bowes, T. Hall, J. Petric, T. Shippey, B. Turhan, How Good Are My Tests?, in: 2017 IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics (WETSoM), IEEE, 2017, pp. 9–14.