# ZL: Automating Failure Testing of Microservice Applications

Qingyi Lu
*qingyi_lu@brown.edu*

Xiling Zhang
*xiling_zhang@brown.edu*

## 1 Introduction

The modern distributed applications are moving toward a microservice architecture, consisting of fine-grained services that can be developed and managed independently, and the updates and new features are delivered continuously. In contrast to the monolithic systems, the microservice architectures could scale the systems with a growing number of concurrent requests and increasingly shorter development cycles. To provide an "always-on" experience to customers, the microservices must be designed to anticipate and work around a variety of error conditions. However, in the microservice-based systems, it is difficult to ensure that such fault-tolerant code is adequately tested, because there exists a huge number of recoverable failure modes within such architecture. In order to build confidence in such system that could generally be resilient to these failures, Chaos Engineering is introduced to perform the experiments with worst-case failure scenarios in the scaled-out production system and increase the overall resilience of a software system [7, 10, 11].

Chaos Engineers work on creating frameworks that automate failure injections by injecting faults in the production environment. More specifically, a fault in service might cause an error, and this error might result in a failure. Fault injection is the process of deliberately introducing faults in a system in order to evaluate its ability to prevent or mitigate failures [6]. Because of the complexity within the microservice-based system, the space of distinct failure scenarios that such an infrastructure can test is exponential in the number of potential faults. Hence, the fault injection infrastructure needs an efficient search strategy to explore such massive space of possible executions. There are some strategies on fault scenarios exploration. The developer-specified exploration strategy leverages the intuition of domain experts to guide the search through failure scenarios [2]. The random exploration strategy randomly choose a failure scenario for the injection [8, 14]. However, in practice, using either random or developer-specified fault injection technique could waste the time and resources on exploring redundant failure scenarios. It is also unlikely to uncover deep failures with combinations of different components and faults. To address these problems, lineage-driven fault injection (LDFI) technique is proposed to efficiently search through possible fault injection scenarios to test the fault-tolerance systems [3, 4]. LDFI leverages data lineage to explore the failure scenarios. It begins with a correct outcome, and reasons backward to determine whether injected faults in the execution could prevent the outcome. Netflix also adapted LDFI to their failure testing system.

Although LDFI could systematically cover the fault space and detect complex faults for testing the resilience of microservice-based applications, it still has room to be further improved. Firstly, LDFI focuses on the systematic exploration of the fault space for each request independently without specific order, and equally considers all faults. Secondly, although LDFI includes strategy to minimize fault injection points, in practice, some injection points are necessary to be tested based on the analysis of the previous testing results. It might take longer to cover the fault space and find the failures eventually. It might be feasible in the production environments, but remains inefficient for the exploration.

According to above observations, in this paper, we propose a new approach called ZL-LDFI, that improves the LDFI exploration of failure scenarios and automatically drives a fault injection, to efficiently detect fault-tolerance bugs in microservice-based applications. We enhance the LDFI system architecture with two strategies. Firstly, based on the historical fault injection results across different user requests, it dynamically computes the priority values of each service, and guides the selection of fault injection points. Secondly, we design a pruning strategy to further reduce some unnecessary fault injections from the minimal solutions solved by LDFI. This system architecture is implemented as a prototype to automate failure testing on a benchmark application. The experimental result shows that this improved approach of LDFI is efficient to detect the fault-tolerance bugs. Our contribution is summarized as follows:

- We improve the LDFI failure scenarios searching technique on automating failure testing of microservice-based applications. We enhance LDFI system architecture by dynamically computing and updating the priority values of each service to guide the selection on failure scenarios for the fault injections.

- We design a pruning strategy to avoid the unnecessary fault injections by leveraging and analyzing the historical fault injection results.
- We implement a prototype on a benchmark. The experimental result illustrates our approach could achieve a more efficient and quick way on automating failure testing of microservice applications.

The paper is organized in the following way. Section 2 presents the LDFI approach and its embodiment in the initial research prototype. Section 2 also presents the strategies that Netflix is used to adapt the LDFI technique into their failure testing system. It provides the motivation for our new approach. Section 3 describes the design and workflow of ZL-LDFI system architecture. Section 4 details our implementation on an open-source microservice-based application as the benchmark. We present some preliminary results and evaluation in Section 5. Finally, this paper closes with related work (Section 6), discussion and future work (Section 7), and conclusions (Section 8).

## 2 Background and Motivation

Lineage-driven fault injection (LDFI) is a technique to efficiently search through possible fault injection scenarios by leveraging data lineage and backwards reasoning [3]. Molly is the LDFI prototype system where both the system and the correctness specification that written in the Dedalus language (an executable specification language based on Datalog) [3]. It takes as input a distributed program that is a correctness specification. Then it inputs and bounds on execution length, and simulates the program's distributed executions under a variety of faults.

The LDFI approach is based on two key insights [3]. Firstly, fault-tolerance is redundancy. This indicates a fault-tolerant system or program can provide alternative way to obtain an expected outcome in the existed common faults. The second insight is to start with successful outcomes and reason backwards. Instead of exhaustive explorations of all possible executions from the initial state, it is efficient to start with successful outcomes and reason backwards to understand whether some combination of faults could prevent the outcome.

LDFI uses data lineage to simultaneously exploit both insights. It begins with a successful execution and reasons backward to find a sequence of steps that could prevent this execution. To analyze the reasons of this successful outcome, the lineage graph is extracted to represent the system executions. It uses conjunctive normal form (CNF) to represent the lineage graph as a formula, involving every path in the graph. The CNF formula is passed into a SAT solver to generate failure hypothesis. Then the solved combination of faults is transformed as inputs to simulate the successful executions in the next round of the loop. This process continues until either a fault tolerance bug is identified or the system exhausts its resources.

Netflix also adapt and implement the LDFI technique into their failure injection testing system [4]. In their study, they indicate the challenges to adapt the LDFI, and propose their solutions based on Netflix microservice architecture. We will explain the overview system architecture of Netflix's LDFI adaptation by using a simple example extracted from our case study (the details of case study and the evaluation present in Section 5). Figure 1a shows a failure-free outcome for a successful execution of a user request, and the invoked services are represented as a call graph. This call graph is then converted into CNF formula as $ui \vee station \vee route \vee food \vee food\_map$. This formal is taken as input into SAT solver to obtain the sets of injection points that should be tested. The minimal solutions for this example is $\{\{ui\},\{route\},\{food\},\{food\_map\}\}$. This set of injection points will combine with the different fault types (e.g. abort , HTTP delay) to create the failure scenarios represented as $\{\{\{route\}, \text{'}abort\text{'}\}, \{\{route\}, \text{'}delay\text{'}\}, \{\{food\}, \text{'}abort\text{'}\}, \{\{food\}, \text{'}delay\text{'}\}, \dots\}$. LDFI chooses a failure scenario from this set and inject it into system. In this example, after injecting the $\{\{food\_map\}, \text{'}abort\text{'}\}$ and replaying the user request, it still gets a successful outcome that shown in Figure 1b. It means that after injecting the '$abort$' into $food\_map$ service, the $food\_map\_rep$ service processes the executions when the $food\_map$ service fails. Then this new execution path is converted into CNF formula as $ui \vee station \vee route \vee food \vee food\_map\_rep$. The conjunction with the previous CNF formula is taken as input into SAT solver to obtain the minimal solutions that could invalid both execution paths. In this case, the solved solutions are $\{\{ui\},\{route\},\{food\},\{food\_map, food\_map\_rep\}\}$. Then, it recursively explores the failure scenarios to detect the bugs in the system.

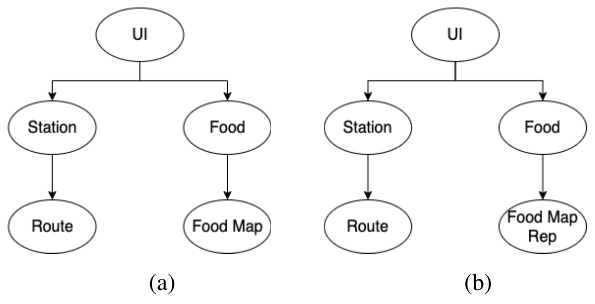

Figure 1: Call graph example for fault injection.

By comparing to random or developer-specified search strategies, LDFI could systematically cover the fault space and detect complex faults by leveraging data lineage and backwards reasoning. It reduces the exploration within the combinations of failure scenarios, and is applicable in practice. However, due to the complex architecture of microservice-based applications, there are lots of different types of users requests, and each of them invokes numbers of internal services. It indicates that the number of injections points could be in large space, which causes a lot of different failure scenarios should be explored. Therefore, it might waste the testing resource and time, and also not efficient to do the failure testing for the complex applications. Hence, we design an improved approach of LDFI system architecture to efficiently and quickly explore the failure testing and detect the error within the system, which is introduced in the following section (Section 3).

## 3 Design

In this section, we discuss the overall architecture of ZL-LDFI. We also mainly present the key insights of the improvements on LDFI technique, which enable the failure testing on the microservice-based applications be more effective.

### 3.1 Overview of System Architecture

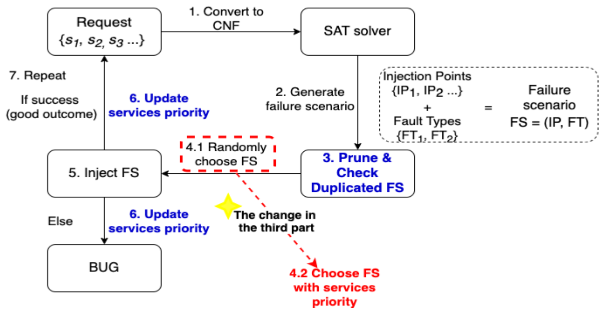
Request
$\{s_1, s_2, s_3 \ldots\}$
1. Convert to CNF
SAT solver
7. Repeat
If success (good outcome)
6. Update services priority
2. Generate failure scenario
Injection Points $\{IP_1, IP_2 \ldots\}$ + Fault Types $\{FT_1, FT_2\}$ = Failure scenario FS = (IP, FT)
4.1 Randomly choose FS
5. Inject FS
3. Prune & Check Duplicated FS
Else
6. Update services priority
The change in the third part
BUG
4.2 Choose FS with services priority

Figure 2: The overview of workflow in the second stage of ZL-LDFI system.

According to the observations of LDFI system architecture that discussed in Section 2, to optimize the fault space exploration of LDFI, we propose a new approach, called ZL-LDFI, to automatically drive a fault injection. This fault injection system extends the Netflix-LDFI technique on fault space exploration. We improve it by leveraging historical failure testing results of different user requests, and guiding the selection of failure scenarios that could have high impact on the reliable execution of the application. It dynamically computes and updates the priority values of each service based on the historical testing results. Additionally, we design a pruning strategy to avoid unnecessary fault injections. The overview of our system architecture is presented in three main stages as follows.

*First Stage* - Based on the implementation logic of application, it generates sequence of user requests, and traces the internal invocations to obtain a call graph. For each request, it extracts services information and represents as a set of internal services that invoked within this request. Then, we select a subset of these user requests. This subset involves the most important user requests according to the application business logic, and it also covers the services that are usually triggered.

*Second Stage* - As shown in Figure 2, given the pre-selected subset in the first stage, each user request is automatically converted into CNF formula (Step 1). It uses the SAT solver to obtain the sets of injection points, similar as the Netflix-LDFI that introduced in Section 2. Those injection points are combined with different fault types to create a set of failure scenarios (Step 2). It then randomly chooses a failure scenario (Step 3.1), and analyzes it with the pruning strategy to determine if it is necessary to be tested (Step 3). From the outcome of injection, the priority value of each service is computed and updated (Step 6). The details of the priority function and pruning strategy are presented in Section 3.3 and Section 3.4, respectively.

*Third Stage* - For the rest of user requests, it basically follows the steps in the second stage. As marked in Figure 2, instead of randomly selecting the failure scenarios (Step 4.1), it explores the failure scenarios and tests it based on the priority of the services that computed in the second stage (Step 4.2). It leads the exploration of failure testing to select the most likely failure scenarios that potentially have high impact on the execution of the application.

In the following sections, we describe some details of the key insights and designs in our system architecture.

### 3.2 Pre-processing of User Requests

Based on the logic of integration tests for the application, the sequences of user requests to interact with the application are generated. A proxy is used to simulate the concurrent user requests. In the meantime, it uses the tracing technique to trace the internal service invocations and obtains the call graphs. For each call graph, it extracts the services information represented as a set, $req = \{serviceA, serviceB, \ldots\}$. To avoid repeated fault injections for the same type of user requests, similar as Netflix strategy [4], it groups different user requests into the same class if they trigger the same set of service invocations. We define these classes as different request types according to the application implementation logic. From those request types, according to the business logic of the application, we select a subset of them that contains the invocations on most important internal services. This subset of request types is used to compute the initial priority of each microservice in the second stage of system architecture, and then applied to guide the selection of failure scenarios in the third stage.

### 3.3 Priority Function on Microservices

To more effectively detect the fault-tolerance bugs, we optimize the LDFI approach of fault space exploration by leveraging the historical failure testing results across different user requests and analyzing whether they could potentially have high impact on the application's execution. We design a priority function to guide the selection of failure scenario to be tested. The priority function computes the priority of different microservices. Based on the outcome of fault injection, the priority of each service is dynamically computed and updated to guide the selection of next failure scenario.

The priority function of each service is defined as follow:

$$priority(s) = errored\_FS(s)/sum\_FS(s)$$

In this function, $errored\_FS$ denotes the number of failure scenarios are already tested for this service and a bug is detected. $sum\_FS$ denotes the total number of failure scenarios are tested for this service. The hypothesis behind this definition is that if this service can not handle the fault with higher probability, the failure scenario that contains this service should be tested at first. This could result a more efficient and quick way to detect the fault-tolerance bugs. It reduces the times on the recursive failure testing.

As mentioned in section 3.2, a subset of request types is chosen to compute the initial priority of each service. Because this subset might not cover all the microservices within the application, even though these request types are the most important ones based on the business logic. In this case, the priority of some services might not be initially computed. To make these services still have chance to be tested, we randomly add a small probability for each of them as the initial priority value.

### 3.4 Pruning Strategy

As we mentioned in Section 3.1, our new approach is designed based on the LDFI approach of fault space exploration. After the requests are generated, a set of internal services that are invoked is extracted from the call graph. Then it is encoded into CNF formula and uses the SAT solver to compute the initial sets of injection points. Each injection point is combined with different fault types to obtain the sets of failure scenario to be tested. Based on the analysis of fault injection results, we observe that there are some unnecessary fault injections within the solutions produced by the solver. To address that, we apply a pruning strategy to further reduce the fault injections to be tested.

To prune the unnecessary fault injections, we leverage the historical failure testing results. The rule is defined as if the injection of sample failure scenario already detected a bug within a system, then more complicated failure scenario is unnecessary to be tested. More specifically, if after injecting a sample failure scenario that contains a service or a set of services, a fault-tolerance bug is detected. This result indicates that it is unnecessary to inject a more complicated failure scenario which involves these services. For example, after injecting $FS = (\{ServiceA\}, `abort')$, there is a bug in the system, then $FS = (\{ServiceA, ServiceB\}, `abort')$ will be removed from the set of failure scenarios need to be tested. It is because the previous testing results of simple failure scenario injection already shows these services could not handle the faults. By applying this pruning strategy, it avoids to test the unnecessary fault injections.

## 4 Prototype

The prototype of ZL-LDFI system architecture is implemented on an open-source microservices based application as benchmark, called Train Ticket [17]. The Train Ticket system is a medium-size java based distributed system that is designed to consist of 22 fault types from an industry survey. We deployed Train Ticket in Google Cloud with Kubernetes and Istio. Istio provides the functionalities to perform fault injections for the failure testing, including HTTP abort and latency.

To simulate concurrent user requests, we used Apache Jmeter [16]. We leveraged some simple tests for UI behaviors provided by Train Ticket. Based on the user guide of Train Ticket system and our experimental results, to increase the coverage of microservices, we defined 9 different request types in Jmeter (the details of grouping the user requests are described in Section 3.2). Some examples of defined user request are shown in Table 1.

For each request type, Jaeger is deployed to participate with tracing information. The call graphs generated by Jaeger are represented in a JSON file. We then extracted these call graphs to get a set of services. For the SAT solver, we used the PicoSAT [15] to get the sets of injection points for the failure testing. To inject the failure scenarios into application system, we deployed Istio which supports HTTP abort and latency faults.

## 5 Evaluation

In this section, we show the experimental results that implemented on the Train Ticket across different fault space exploration strategies as configurations. We then describe our evaluation and justification for these results.

The experiments of ZL-LDFI, our improved version of native LDFI with priority and pruning strategies, is conducted on the Train Ticket application. According to the Section 3, a subset of request types is chosen to compute the initial priority of each service. We performed the experiments of ZL-LDFI among different pre-selected subsets. We evaluated these results and set up an optimal subset, which involves the most important request types according to the application business

| Request Type Name | Descriptions |
|---|---|
| type_cheapest_search | Simulate user behaviors to search all possible tickets from one city to another sorted by cheapest combination prices |
| type_food_service | Get all food services between two given stations |
| type_preserve | Preserve a ticket for the given contact and ticket choices such as food or consign |
| type_admin_get_route | Get all the routes in admin mode |

Table 1: Request types examples.

| Config | Average number of fault injections |
|---|---|
| Native LDFI | 22 |
| LDFI with Pruning | 10 |
| LDFI with Priority | 11 |
| LDFI with Priority & Pruning | 9 |

Table 2: Summary of the average number of fault injections on different fault space exploration strategies.

logic and also covers the microservices that are commonly invoked.

To evaluate the effectiveness of ZL-LDFI, we implemented an algorithm to randomly choose a failure scenario among the exponential space of failure scenarios. We counted the number of fault injections to detect all the errors for 9 request types. The result of random strategy is averaged to 31 times of fault injections, and ZL-LDFI is only 10 times. It demonstrates that the ZL-LDFI could efficiently detect the errors within the system than the random exploration strategy.

In order to comparing the performance of our improvements on LDFI approach, we also conducted the experiments on four different LDFI approaches of fault space exploration strategies. We represented them as different experiment configurations as the *Config* column in Table 2. We performed the experiments for each configuration in 10 times. The results involving the priority strategy are based on the optimal subset we pre-selected. As mentioned in Section 3, our algorithm recursively injects the failure scenarios for each request type. We set up a counter to calculate the times of recursive failure scenario injections among all the request types. The average number of fault injections for each configuration is shown in Table 2).

The Native LDFI configuration is implemented as a pure version of LDFI technique without priority or pruning [3, 4]. The average number of injections for this configuration is 22. The result of ZL-LDFI, the improved approach of LDFI with priority and pruning, shows that it takes 9 times in average to inject all the failure scenarios. Comparing to the result of Native LDFI, this suggests that our design of priority and pruning increases the efficiency of the failure testing.

The experimental results also show that the fault space exploration with the combination of priority and pruning are closed to the results of only priority or pruning. It potentially indicates that the implementations with only priority or pruning already achieve the effectiveness on fault space exploration by comparing to native LDFI, and the combination of them does not provide much advantages. One possible reason is the Train Ticket system can not handle failure very well, and our request types are built with some business logic we learned in their fault replicate steps. Therefore, it is easier to trigger the failures within the system. As we observed in our experiments' log, it could normally detect the bugs within the system by only injecting some simple failure scenarios. By applying our pruning strategy, the complicated failure scenarios are pruned based on the historical results. Hence, this is one reason that our experimental results show the pruning on the failure scenarios could provide significant benefits. Recalling that LDFI relies on successful outcomes and then reasons backwards. If triggering failure too fast, it leads to quickly end the recursive injections. So there are less successful outcomes could be used to solve new sets of injection points. In this case, the fault injections could not take the advantages on the priority of each services, and there does not exist many failure scenarios to be pruned. Thus, the experimental results on the Train Ticket does not show the significance on the fault space exploration with the combination of priority and pruning. In our future work, we will work on adding some fault-tolerant code into Train Ticket to better evaluate the ZL-LDFI algorithm. We also expect to test our algorithm in larger distributed systems and dig more into this problem (we discuss it in Section 7).

## 6 Related Work

Exhaustive (random) exploration of fault scenarios is the original and most popular Chaos Monkey was introduced by Netflix in 2010 to test system stability by injecting failures using pseudo-random termination of instances services [2, 8, 14]. Netflix migrated their services to cloud reliant upon Amazon Web Services(AWS). Trying to make sure their system can still manage to work when critical components are taken down, they present the concept of Chaos Engineering which intentionally causes failures that exposes the weakness to guide them for solution design. By that time, Chaos Monkey focused on handling termination of random instances. Later on, due to the level of intricacy and interconnectedness in their systems, Netflix also implemented a suite of failure-inducing tools, Simian Army [9], to extend the capabilities of injections failures. Janitor Monkey checks any given resource by disposing them against configurable rules to determine if it's an eligible candidate for cleanup. Chaos Gorilla simulates the outage of the entire AWS available zone and Chaos Kong simulates specific region outages. Latency Monkey causes artificial delays in RESTful client-server communications. There are many tools focused on different areas of the system and have the ability to exhaustively perform failure injections.

There are some tools allow developer-specified exploration of fault scenarios to perform failure testing. For example, Gremlin is a framework for systematically testing the failure-handling capabilities of microservices [2, 10, 11]. It performs resiliency testing that tests the application's ability to recover from failures. The resiliency testing has to be systematic and feedback-driven to help developers to locate failures when they failed to recover. They suggest this can give more valuable information than random fault injection by doing automatic validation. Therefore, Gremlin relies on operators to provide a recipe, python based code describing a high-level outage scenario, along with a set of assertions on how microservices should react during such an outage. Gremlin targets faults from network communication, including fail-stop or crash failures, performance or omission failures, and crash-recovery failures.

## 7 Discussion and Future Work

The request types we defined according to the fault replicate package of Train Ticket and the business workflow we observed in our experiments. Thus, these pre-defined 9 request types are limited based on our understanding and could possibly miss some crucial request types for this system. Currently, we relied on Jmeter to manually generate request types step by step using Train Ticket api, where writing Jmeter scripts is time-consuming. This step requires prior knowledge and the understanding of Train Ticket system and might not cover all possible request types. Our future direction is to write an automate request type generator. Similar to Gremlin recipe, we will allow engineers to write high level request and produce corresponding request types.

Another challenge in this project is to locate traces for each request type. Every request type can produce different traces information, and the traces in Jaeger can be retrieved by specifying microservice and operations. However, when running concurrently, it is hard to build connection between each trace under that microservice to different request type. We now connected the traces by specifying the entry microservices that can uniquely identify the traces in our experiment with request types. To further extend this, we will try to have each request to include certain unique identifier in HTTP request's body. When we retrieve traces, those identifiers help classify the request types. Another possible direction is to work with industrial requests and traces. Then it is possible to classify requests to requests types based on traces other than collecting traces from generated requests.

Additionally, the Train Ticket system is a project that builds based on industrial report to reproduce some common fault types. It does not have industrial system sizes with complicated structures. Our future work will apply the current algorithm in larger system to see if the effectiveness of failure testing holds when services dependency grows larger. We will also extend our algorithm to include more fault types. For example, using Chaos Mesh [12] or Chaosblade [13] can inject Kubernetes related faults such as pod killing or container killing.

For using the priority to guide the failure scenarios selection, once more fault types are involved, it is possible to design a priority function for each fault type. Other than adding priority for fault types, there are a few more ways to extend priority strategy. One direction is to assign priority to microservices based on code complexity that can be evaluated from cyclomatic complexity or logic condition complexity [1]. Probabilistic model that powered by logistic regression can help improve priority and initial subset in the first stage. Code commits and popularity are also potential parameters for priority calculation [5].

## 8 Conclusion

This paper presents the ZL-LDFI, an improved version of LDFI technique, to automate failure testing for the microservice-based applications. It leverages the historical fault injection results to optimize the fault space exploration across different user requests. After applying the priority function and pruning strategy, it guides the selection of failure scenarios for fault injection testing to the ones that potentially have high impact on the application. The experimental results on a medium-size microservice benchmark system show that this approach could efficiently and quickly detect the fault-tolerance bugs within the application. To comprehensively evaluate the effectiveness of this approach, in the future, we will conduct the experiment on larger or real industry microservice-based applications.

## References

[1] Jaime -. *Code Complexity: An In-Depth Explanation Complexity Metrics*. Oct. 2020. URL: https://blog.codacy.com/an-in-depth-explanation-of-code-complexity/.

[2] *Chaos Monkey at Netflix: the Origin of Chaos Engineering*. URL: https://www.gremlin.com/chaos-monkey/the-origin-of-chaos-monkey/.

[3] Peter Alvaro, Joshua Rosen, and Joseph M Hellerstein. "Lineage-driven fault injection". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 2015, pp. 331–346.

[4] Peter Alvaro et al. "Automating failure testing research at internet scale". In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. 2016, pp. 17–28.

[5] Sundaram Ananthanarayanan et al. "Keeping Master Green at Scale". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19. Dresden, Germany: Association for Computing Machinery, 2019. ISBN: 9781450362818. DOI: 10.1145/3302424.3303970. URL: https://doi.org/10.1145/3302424.3303970.

[6] D. Avresky et al. "Fault injection for formal testing of fault tolerance". In: *IEEE Transactions on Reliability* 45.3 (1996), pp. 443–455. DOI: 10.1109/24.537015.

[7] A. Basiri et al. "Chaos Engineering". In: *IEEE Software* 33.3 (2016), pp. 35–41. DOI: 10.1109/MS.2016.60.

[8] Netflix Technology Blog. *Netflix Chaos Monkey Upgraded*. Mar. 2018. URL: https://netflixtechblog.com/netflix-chaos-monkey-upgraded-1d679429be5d.

[9] Netflix Technology Blog. *The Netflix Simian Army*. Sept. 2018. URL: https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116.

[10] *Chaos Engineering*. URL: https://www.gremlin.com/chaos-engineering/.

[11] *Chaos Engineering: the history, principles, and practice*. URL: https://www.gremlin.com/community/tutorials/chaos-engineering-the-history-principles-and-practice/.

[12] *Chaos Mesh®*. URL: https://chaos-mesh.org/.

[13] Chaosblade-Io. *chaosblade-io/chaosblade*. URL: https://github.com/chaosblade-io/chaosblade.

[14] Netflix. *Netflix/chaosmonkey*. URL: https://github.com/Netflix/chaosmonkey.

[15] *pycosat*. URL: https://pypi.org/project/pycosat/.

[16] Peter Wainwright. "Getting Started with Apache". In: *Pro Apache* (2004), pp. 37–100. DOI: 10.1007/978-1-4302-0658-3_2.

[17] X. Zhou et al. "Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study". In: *IEEE Transactions on Software Engineering* (2018), pp. 1–1. DOI: 10.1109/TSE.2018.2887384.