ART : Automated Resilience Testing for Microservices

Alex Yu
Brown University

Katie Scholl
Brown University

Abstract

ART is a framework for automated resilience testing on microservices deployed on top of Istio. Our purpose is to provide meaningful insights of application behavior during failure and improve ease-of-use of microservice fault-tolerance testing. ART accomplishes these goals by identifying critical application components, automatically conducting fault-injection experiments, and presenting the results to the user.

1 Background

Testing for fault tolerance is necessary for guaranteeing application availability under various scenarios [5, 11, 28]. At the application level, developers write unit tests to ensure correct behavior of code under invalid input. To test resiliency across services, developers often use language-specific RPC frameworks to test client behavior in the face of downstream service failure [19, 24].

However, two problems often emerge when performing resilience testing on applications whose requests traverse multiple services: injecting faults between polyglot services [21, 26] and choosing services to test [25].

For the former problem, traditional resilience testing often involves writing mock-ups for a client-side API [19, 22, 24]. In order to thoroughly test failure scenarios, developers are required to write or learn a language-specific framework that can simulate network delays, network failures, and error responses. This process must be repeated for each programming language used in a polyglot microservice environment to achieve extensive coverage.

For the latter problem, the amount of failure scenarios can grow at an exponential rate as services are added [17, 18]. Testing all of these scenarios quickly becomes unfeasible. As a result, developers need to choose a few critical components to test. Without in-depth application observability, developers must rely on their intuition when choosing these critical components.

Service meshes (e.g Istio and Linkerd) tackle both of these problems by abstracting the network communication between services via Layer-7 network proxies (e.g Envoy). Service meshes provide a high-level API that enable users to manipulate inter-service request routing, latency, and responses. These features can also be used to inject artificial faults between services. In terms of application observability, some service meshes (e.g Istio), provide out-of-the-box distributed tracing. Traces from a well-instrumented application can help identify critical application paths.

Our framework brings these service mesh components together to identify critical paths in the test application, automatically apply fault injections on these paths, and summarize how the test application fared during fault injection.

2 Related Work

There are many prior works that cover the topic of resilience testing [11, 23, 24, 27–29]. Most of them present novel frameworks to inject artificial faults and observe resulting application behavior. Recently, there has also been an increasing amount of research on identifying critical application components using logs or trace data [10, 25, 31]. Our work builds upon both of these topics as it uses techniques for identifying critical services from trace data, and it uses the API of a cutting-edge fault-injection framework to conduct experiments.

2.1 Fault Injection Frameworks

There is a rich set of fault injection frameworks that use either the hardware, network, or programming language level to inject faults into applications [11, 19, 23, 27–29].

Netflix's Chaos Monkey [11] simulates hardware failure by randomly triggering cloud infrastructure failures on AWS. At Netflix, engineers use this tool to observe application behavior during sudden failure of critical cloud resources such as block storage and virtual machines.

At the network level, Gremlin [23] is a system that intercepts all HTTP requests between services using network proxies. Based on user-supplied configuration, these proxies

are able to return failures to the calling services and record resulting network behavior to assert correct fault handling.

An example of resilience testing via language-specific libraries is how Netflix utilizes a common set of Java libraries deployed on each microservice to inject faults into any arbitrary service [19]. With language-based fault injection, Netflix is able to simulate faults of downstream services at the client level and can obtain detailed metrics on resulting request behavior, though their approach makes it more challenging to extend to a polyglot architecture.

Groupon employs a similar approach for their resiliency testing platform, Screwdriver, by modelling fault injections as Java objects [28]. However, they differ from Netflix in that they perform experiments on controlled environments rather than in production.

Jackpot is a system that helps DevOps engineers identify optimal versions of microservices to meet user-inputted KPIs [8]. Similar to our work, it uses Istio's Virtual Services and Jaeger to efficiently experiment with the application and measure results. However, Jackpot differs from ART in that it is mainly concerned with latency SLA requirements between microservices rather than application behavior during failure.

ART differs from the above works since it additionally suggests which application components to perform fault-injections on. While all of these works provide novel ways to experiment with an application, they do not analyze the test application for critical components to test.

2.2 Identifying Fault Injections

Setting up a framework to perform fault-injections is only the first step of resilience testing. As application size grows, efficiently identifying important components to test is necessary for overall application resiliency and better developer productivity [17].

Linear Driven Fault Injection (LDFI) [18] is an algorithm that minimizes a distributed system's failure space by working backwards from correct system scenarios. Alvaro et al. then applied their work from LDFI to Netflix's microservice architecture to show how LDFI helps developers greatly cut down on the amount of services to test [17].

Other work has emerged that extrapolates problematic application components from failure logs. Zhou et al. propose a system that finds the simplest possible configuration of a microservice system to reproduce bugs [31]. Sifter is an add-on to tail-based tracing systems that uses machine learning sampling algorithms to identify abnormal traces [25]. Sifter accomplishes this by filtering out frequent types of traces, most likely successful requests, while preserving rare traces, most likely failed requests.

Lastly, recent work by Mania et al uses sub-graph mining to identify a distributed system's critical *workflow motifs* from trace data [10]. Since requests in distributed systems can be modeled as directed-acyclic graphs (DAGs), they define *workflow-motifs* as sub-graphs within these requests.

Our work primarily borrows from Mania et al. in that we use the same definition of *workflow-motifs* to describe *request workflows*. We also use sub-graph mining to identify these critical workflows. However, our work differs from that of Mania et al in that we additionally allow users to test and observe the resiliency of their application when these critical workflows fail.

3 Design

The purpose of our fault injection framework is to identify critical workflows in a microservice application and observe its behavior when those workflows fail.

3.1 Obtaining the application overview

To identify critical workflows, we first obtain a set of traces that cover all request paths that pass through the frontend service. We use these traces since application users directly interact with these requests. Also, we assume that the user provides their own method to generate load on the application.

Uber employs a similar strategy for their own resilience testing [12]. In order to find an accurate baseline for application behavior, Uber deploys business-specific load generators that make user requests on production servers. The resulting trace data is then aggregated to learn the typical system behavior. Subsequent production traces can then be compared to the aggregate, which allows Uber to detect anomalies. Connecting back to our work, Uber's approach shows that load generation must be the user's responsibility and that initial aggregation of trace data is crucial for differentiating between correct and incorrect system behavior.

3.2 Identifying critical workflows

ART automatically identifies critical workflows for the user so that they do not have to rely on their own intuition when choosing application components to test. After obtaining relevant traces from the previous step, we convert each of them into a directed-acyclic graph (DAG). From this graph data, we extract the most critical workflows which we define as the most commonly occurring sub-graphs.

To extract frequently occurring sub-graphs, we use the gSpan algorithm [30] which employs depth-first search (DFS) to recursively search across multiple graphs in a dataset. We chose gSpan since it was already implemented as a Python module and it supports directed graphs, which is crucial for accurately representing request workflows.

We also provide users the option to change the scope of these sub-graphs by setting the minimum and maximum amount of vertices for workflows. For example, to find the most common request that spans between 2 services, one would simply set the min and max vertices amount to 2. In

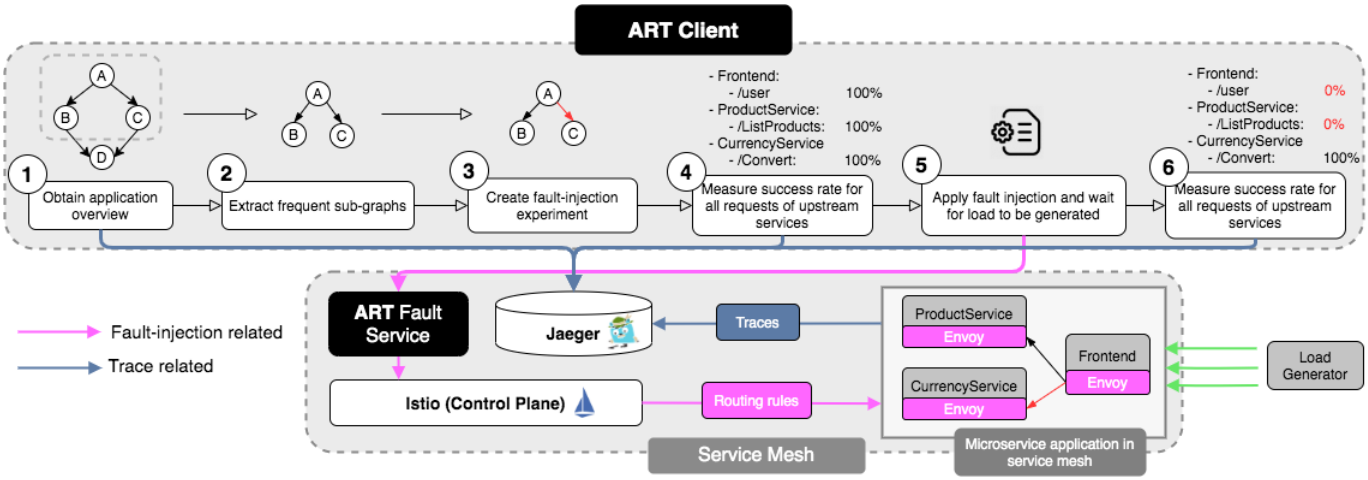

Figure 1: ART design overview

this sense, the user is allowed to tune the scope of their fault tolerance domain. They can test workflows that simply span between two services to workflows that traverse the entire application.

3.3 Generating and applying fault-injection code

Once we have a list of critical workflows, the user must select a vertex or edge within one of these workflows for fault injection. Vertices represent services and edges represent request paths [1].

Users may initially prefer service-based fault injections since they represent typical distributed system failure modes (e.g node failures, network partitions, and bugs that crash the entire pod). This is useful for measuring overall application resiliency. However, such experiments could lead to a copious amount of failed traces spread across the entire application, which can be hard for the user to digest. Thus, having the option to limit the failure scope can be useful for debugging as failed traces will be more concentrated in specific services or request paths. In such scenarios, Request-based fault injection is more suitable as it will leave all other request URLs for the injected service unaffected.

After the user has selected a service or request path to test, they must also specify a failure rate between 0-100%. Currently, ART only injects faults that abort immediately and return a 500 HTTP status back to the caller. While Istio's fault-injection API provides much more customization (e.g Delay type failures, custom HTTP status, multiple injections at once, etc) [6], we found that using Abort type failures was sufficient for evaluating ART.

We only provide a subset of Istio's fault-injection API because we implemented our own 'wrapper' API on top of Istio. Directly using the Istio API requires access to port 6443 on the master node and the user to have the cluster's signed root certificate [4] on their local machine. Thus, without a layer of abstraction above the Istio API, allowing multiple users to interact with ART would require either sharing the root certificate or configuring Kubernetes authorization roles [4].

This was too inconvenient for a prototype implementation of ART so we instead elected to write our own fault service that was reachable via insecure gRPC on an open port. This fault service resides inside the Kubernetes cluster and has special RBAC permissions for applying fault injections. Thus, when a user applies a fault injection using the ART CLI, the request is sent to the fault service, which then translates the query's parameters into an Istio configuration, and finally applies it to the cluster. In the future, we plan on expanding ART's fault service API to accommodate more complex failure scenarios.

3.4 Measuring application behavior

Before a selected fault injection is applied to the application, we first measure the application's behavior. This is done so that we can establish a baseline to compare with the measured results after the fault injection. We define an application's behavior as the percentage of requests that return a 200 HTTP status for each unique request URL. These measurements are compiled from recently queried trace data so that the baseline accurately represents the current application state.

One problem with this approach is the measurement of unnecessary data since an application may consist of hundreds, if not thousands of unique request URLs. We tackle this problem by only measuring the upstream services of the to-be-injected service. To give an example of upstream services, imagine a request that traverses 4 services, starting at service A and ending at service D: $A \rightarrow B \rightarrow C \rightarrow D$. If service C is selected for fault injection, ART will measure the

[1] We added request path fault injection after the final presentation

success percentage of all unique request URLs for services *A* and *B*. We measure *all* upstream services rather than just immediate ones so that the user can see the scale at which an injected fault propagates through the application.

After applying the fault injection, ART waits a predefined amount of time for load to be generated on the system. It then uses the exact same approach to measure application behavior after the fault injection. Given these two sets of measurements, ART presents the change in success percentage for each unique request URL that occurs in both measurements.

Currently, ART does not employ any methods to show whether the changes in success percentages are statistically significant. This sometimes leads to success percentages being skewed based on the amount of traces being measured. For instance, ART may show a 100% failure rate on a request URL, but it could be that it only measured a total of 1 request for that specific request path. We address this issue by displaying a list of failed trace IDs for requests that had a negative change in success rate. This serves two purposes: 1) Users can further examine these failed traces if need be 2) Users can see how many requests failed and perform a sanity check with the outputted failure rate.

```
frontend: {
        cartservice: {
                http://cartservice:7070/hipstershop.CartService/GetCart: -40.00%
                        Failed traceID: b8703c3cf8ac35ed2c92a5cec661d18e
                        Failed traceID: 87a0649c96de3de9580fc9a44f909f91
        }
        adservice: {
                http://adservice:9555/hipstershop.AdService/GetAds: 0.00%
        }
```

Figure 2: An example of fault-injection results. In this case, a 50% failure rate was applied to cartservice. We observe from the frontend service's perspective, the **difference** in success rate of its requests to cartservice and adservice before and after fault injection. The results shows failed trace IDs from cartservice and some of the frontend's other requests being unaffected by the fault injection.

3.5 Design summary

To summarize the entire process:

1. Query traces from each operation (request path) of the frontend service to capture an overview of the application. (Step 1 from figure 1)

2. Convert traces into DAGs while storing translation maps from vertex and edge labels to service and request paths respectively. (Step 2 from Figure 1)

3. Use the gSpan sub-graph mining algorithm on the extracted DAGs to find the most commonly occurring sub-graphs. These sub-graphs represent the application's critical workflows. (Step 2 from figure 1)

4. The user then selects a single workflow from the generated list of critical workflows. Within that workflow, the user selects a single service or request path for fault injection. Lastly, the user must input a value between 0-100 to represent the failure rate of the fault injection. (Step 3 from figure 1)

5. Before ART applies the fault injection, it queries recent traces from all upstream services of the to-be-injected service. This trace data is aggregated and used to measure the success rate of requests before fault injection. (Step 4 from figure 1)

6. ART sends the information gathered from Step 3 in figure 1 to the fault service, which creates an Istio fault-injection configuration and applies it to begin the experiment. (Step 5 from figure 1)

7. ART waits 30 seconds for load to be generated on the application and then queries traces from the same services from step 4 in figure 1. After measuring the success rate of requests from these traces, ART deletes the fault injection and presents the deltas between these results and the results gathered from step 4 in figure 1. For request paths that have a negative delta (i.e more failed requests as a result of the fault injection), ART will also include failed trace IDs. (Step 6 from figure 1)

4 Implementation

ART consists of a client application and dedicated fault service that both interact with Istio and its sub-components (e.g Jaeger and Kiali). The client application and dedicated fault service are both written in Go. One advantage with using Go is its extensive support for gRPC, which we used in the client application for querying traces from Jaeger. The biggest advantage with using Go, however, is that Istio only provides a native client library written in Go [1]. As a result, we had direct access to Istio's fault-injection structs, which made creating and applying fault-injections fairly straightforward. Lastly, much of the Istio and Kubernetes environment is written in Go as well [2, 3, 9, 13], so it would be easier for other developers in this field to make contributions to ART. You can find the code for ART at https://github.com/triplewy/art.

4.1 Querying traces

When starting a fault-injection experiment, ART first queries Jaeger for all of the frontend service's unique operations. Operations are equivalent to request URLs. Currently, the name of the frontend service is hard-coded, and thus so are the trace query parameters. However, we plan on making this configurable in the future. Once we have a list of unique operations, we filter out all non-"recv" operations. This is important so that we only measure request paths that users send

to the frontend rather than requests that the frontend sends to downstream services.

For each unique frontend request path, we query its traces from the last 60 seconds. In this manner, requests that occur more frequently than others will be proportionally represented. Each trace is then converted into a DAG where vertices represent services and edges represent request URLs. The exact format of these DAGs is determined by the gSpan sub-graph module [30] we use. Once each trace is converted into graph format and merged into a single file, we can perform sub-graph mining on them.

4.2 Sub-graph mining

The gSpan sub-graph mining module we use is a Python module that produces a list of the most commonly occurring sub-graphs based on some input file. Three configuration options to consider are the minimum and maximum amount of vertices for sub-graphs and the minimum support level of sub-graphs to show. Choosing the size range of sub-graphs is dependent on the granularity of workflows that the user wants to test. Setting the minimum support level, which is the amount of times a sub-graph occurs, is important for performance. Too low of a support level leads to the mining library running for multiple minutes whereas too high a support level may not show any qualified sub-graphs. This is a future area of research to learn a sweet spot for the support level.

Once we have the resulting sub-graphs, we translate all of them back into request workflows by adding service labels to vertices and request URL labels to edges. Before presenting this list of critical workflows to the user, we must first filter out workflows that only cover the frontend service. Since the frontend has no upstream services, injecting faults into it would not produce any meaningful information about how the test application handles failure. We will further expand on this in section §4.3.

4.3 Istio's Virtual Services

ART uses Istio's Virtual Services [16] to perform fault-injections. Virtual Services can define rules for routing TCP and HTTP traffic between Envoy proxies. Each pod in the application cluster will have a sidecar Envoy proxy that captures all inbound and outbound traffic for the pod. HTTP fault injections are a feature provided by the Istio Virtual Service API, which allow users to inject abort or delay type faults for specific HTTP routes at a service.

Fault injections can be applied on an entire service or a specific HTTP Route using Istio's HTTPMatchRequest [7]. An HTTPMatchRequest is a set of criteria on a request's attributes (e.g path, headers, method, etc) and only applies routing rules if there is a match. Thus, we inject service-granularity faults using a path prefix '/' and inject request-granularity faults using a path prefix of the targeted request.

Fault injections are attributed to destination services, but are applied at the client side. For example, to fail a request that travels from Service $A \rightarrow B$, we apply an HTTP Fault Injection on Service B, but Istio will fail the request at Service A's envoy proxy. Thus, Service B will never receive this failed request. Referring back to the scenario presented in §4.2, applying a fault injection at the frontend would be meaningless since failed requests would never reach the frontend and thus would not be processed by the application.

Istio's Virtual Service architecture presents two major limitations for performing fault injections. First, since Istio only supports fault injections for the HTTP protocol, it cannot inject faults for TCP traffic. This is a major limitation for testing application resiliency during datastore failure since many datastores use their own TCP-based protocol (e.g MySQL, MongoDB, Redis). Second, one cannot inject faults into third-party services outside the Kubernetes cluster despite the client side implementing fault injection. This is because users can only specify Virtual Service rules for the destination service, not the sending service. As a result, one cannot use Istio to test how their application fares when a third-party API or cloud service becomes unreachable.

4.4 Calculating results

ART measures application behavior using two steps: Obtaining upstream services and querying traces. One important thing to note is our decision to measure upstream *services* rather than upstream *requests*. When we measure an upstream service, we query traces for all of its outgoing requests, regardless if they travel to the injected service. By doing this, we consider the possibility that failed requests of one path can affect other non-related request paths. For example, if one request path is generating many errors and taking up machine resources for error logging, this could impact the performance of other completely unrelated requests [20]. Thus, we sacrifice more compact analysis with the ability to detect these errors.

To obtain upstream services, ART builds a reverse call graph of the application and performs simple DFS to extract all children nodes of the to-be-injected service. We define a reverse call graph as downstream services being the parents of upstream services. Building the application's reverse call graph is done using the Kiali API. We chose Kiali for this step since it is deployed with Istio and provides out-of-the-box functionality for capturing real-time cluster topology [15].

Each fault-injection experiment consists of querying traces before and after the fault-injection. We measure the difference in success rate for each request URL in all upstream services. This way, the user can see the scale at which the injected fault propagated through the application.

One caveat with this delta approach is that we can only show requests URLs that occur in both before and after datasets. Thus, new request paths that arise due to down-

stream failure will be captured by ART's trace querying, but will not be shown in the final results. Potential future work to fix this issue could be adding a diff-like visualization between anomaly traces and an initial baseline. This approach is similar to how Uber helps engineers debug failed traces by identifying the differences between anomaly traces and a baseline representation they calculate via black-box testing [12].

5 Evaluation

Our main goals for ART are identifying critical workflows for a microservice application, applying fault injections on these workflows, and clearly presenting the results to the user. We evaluate ART based on these three criteria.

5.1 Hipster shop

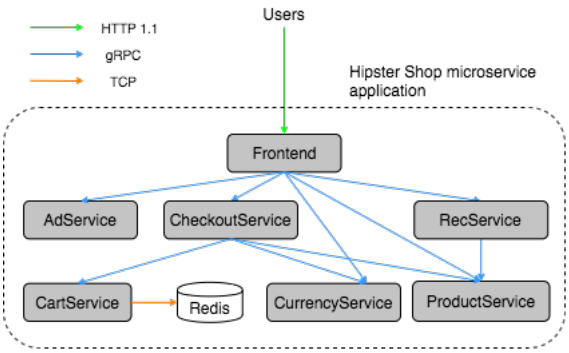

Figure 3: Simplified architecture of the Hipster Shop microservice application. Users interact with the frontend via HTTP 1.1, which in turn contacts downstream services using gRPC.

We tested our framework on our own fork of Google's microservice demo, Hipster Shop [14]. Hipster Shop consists of a frontend service that listens for HTTP 1.1 requests, multiple backend services that communicate via gRPC, and a Redis service that uses its own TCP protocol. Initially, we had some issues with using Istio's sidecar proxies to propagate trace headers through gRPC services so we ended up adding our own instrumentation to directly export traces to Jaeger. This was necessary since ART requires fully 'stitched' traces for extracting repeated workflows from the test application.

Once we had correctly formatted traces, we were able to extract the most common application workflows via sub-graph mining on the trace data. Specifically, we identified the RecService requesting a product list from the ProductService as the most commonly occurring workflow. This is corroborated by the fact that Hipster Shop will show product recommendations at the home page and any product page. In this case, identifying the specific RecService→ProductService workflow is useful since it provides the user context about which request path and upstream services will be primarily affected by a ProductService failure. Simply identifying ProductService as the most critical service makes testing individual request paths more difficult.

Given this specific workflow, we apply multiple different fault injections to verify that our fault service API is working properly. Specifically, we inject a 100% failure rate on the ProductService, a 50% failure rate on the request URL between the two services, and a 20% failure rate on the RecService. During each experiment, we examine the success rate of upstream requests to confirm that the fault injection is being applied with the correct scope and rate. From our experiments, we noticed that immediate upstream services correctly reflected the applied failure rate. However, the failure rate frequently *increased* for further upstream services (i.e frontend). This indicates how the frontend queries a workflow multiple times for a given request path.

Lastly, we take the failed trace IDs from the results of each experiment and use Jaeger to further examine them. We found that the injected request path or service correctly returns a Fault Filter Abort error with HTTP status 500. We also found that these errors are propagated all the way up to the frontend. This behavior is confirmed when we interact with the frontend during a fault injection and frequently encounter error pages.

6 Conclusion

Previous work on resilience testing either only tackle the mechanism of injecting faults or methods to identify critical application components. In this paper, we discuss how our framework, ART, makes use of state-of-the-art techniques in both areas to make resilience testing easier and more useful for microservice applications. First, we show how using sub-graph mining on trace data can expose critical request workflows that the test application heavily relies on. We then describe our approach for measuring the outcome of a fault-injection experiment. Notably, we analyze the request success rate of all upstream services of the to-be-injected service. This allows us to capture the scale at which faults can propagate through the application. Lastly, we provide failed trace IDs for request paths whose success rate decreased as a result of fault injection. This allows users to further examine failed requests and perform root cause analysis. While our framework is still a work in progress, we believe that it is the right direction in effectively testing microservice resiliency.

References

[1] Announcing istio client-go. https://istio.io/blog/2019/announcing-istio-client-go/. [Online].

[2] Cncf jaeger, a distributed tracing platform. https://github.com/jaegertracing/jaeger. [Online].

[3] Connect, secure, control, and observe services. https://github.com/istio/istio. [Online].

[4] Controlling access to the kubernetes api. https://kubernetes.io/docs/reference/access-authn-authz/controlling-access/. [Online].

[5] Gameday: Creating resiliency through destruction. https://www.youtube.com/watch?v=zoz0ZjfrQ9s&feature=youtu.be. [Online].

[6] Istio: Virtual service httpfaultinjection. https://istio.io/docs/reference/config/networking/virtual-service/#HTTPFaultInjection. [Online].

[7] Istio: Virtual service httpmatchrequest. https://istio.io/docs/reference/config/networking/virtual-service/#HTTPMatchRequest. [Online].

[8] Jackpot: Online experimentation of cloud microservices.

[9] Kiali project, observability for the istio service mesh. https://github.com/kiali/kiali/. [Online].

[10] Leitmotif: An abstraction for debugging distributed applications - mania abdi, northeastern u. https://youtu.be/SHSJzxiuV-Y. [Online].

[11] Netflix - chaos monkey released into the wild. https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116. [Online].

[12] A picture is worth 1,000 traces - steve flanders, splunk yuri shkuro, uber technologies. https://www.youtube.com/watch?v=rMSqgSFuBi0. [Online].

[13] Production-grade container scheduling and management. https://github.com/kubernetes/kubernetes. [Online].

[14] Sample cloud-native application with 10 microservices showcasing kubernetes, istio, grpc and opencensus. https://github.com/triplewy/microservices-demo. [Online].

[15] Service mesh observability and configuration. https://kiali.io/. [Online].

[16] Virtual service. https://istio.io/docs/reference/config/networking/virtual-service/. [Online].

[17] Peter Alvaro, Kolton Andrus, Chris Sanden, Casey Rosenthal, Ali Basiri, and Lorin Hochstein. Automating failure testing research at internet scale. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 17–28, New York, NY, USA, 2016. Association for Computing Machinery.

[18] 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*, SIGMOD '15, page 331–346, New York, NY, USA, 2015. Association for Computing Machinery. https://doi.org/10.1145/2723372.2723711.

[19] Ali Basiri, Lorin Hochstein, Nora Jones, and Haley Tucker. Automating chaos experiments in production. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '19, page 31–40. IEEE Press, 2019.

[20] Nathan Bronson. Solving the mystery of link imbalance: A metastable failure state at scale. https://engineering.fb.com/production-engineering/solving-the-mystery-of-link-imbalance-a-metastable-fa [Online].

[21] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design Implementation*, NSDI'07, page 20, USA, 2007. USENIX Association.

[22] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. Fate and destini: A framework for cloud recovery testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 238–252, USA, 2011. USENIX Association.

[23] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar. Gremlin: Systematic resilience testing of microservices. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 57–66, 2016. http://www.cs.unc.edu/~victor/papers/gremlin-icdcs.pdf.

[24] Pallavi Joshi, Malay Ganai, Gogul Balakrishnan, Aarti Gupta, and Nadia Papakonstantinou. Setsudundefined: Perturbation-based testing framework for scalable distributed systems. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, TRIOS '13, New York, NY, USA, 2013. Association for Computing Machinery.

[25] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. Sifter: Scalable sampling for distributed traces, without feature engineering. In *Proceedings of the ACM Symposium on Cloud Computing*,

SoCC '19, page 312–324, New York, NY, USA, 2019. Association for Computing Machinery.

[26] Jonathan Mace and Rodrigo Fonseca. Universal context propagation for distributed system instrumentation. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[27] M. Munro N. Looker and J. Xu. Ws-fit: A tool for dependability analysis of web services. In *IEEE Computer Software and Applications Conference*, 2004. https://ieeexplore.ieee.org/document/1342690.

[28] A. Nagarajan and A. Vaddadi. Automated fault-tolerance testing. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 275–276, 2016. http://www2016.taicpart.org/slides/TAICPART2016-Vaddadi-AutomatedFaultToleranceTesting.pdf.

[29] K. G. Shin S. Han and H. A. Rosenberg. Doctor: An integrated software fault injection environment for distributed real-time systems. In *Proc. of International Computer Performance and Dependability Symposium*, 1995. https://www.eecs.umich.edu/techreports/cse/93/CSE-TR-192-93.pdf.

[30] Xifeng Yan and Jiawei Han. Gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02, page 721, USA, 2002. IEEE Computer Society.

[31] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Wenhai Li, Chao Ji, and Dan Ding. Delta debugging microservice systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 802–807, New York, NY, USA, 2018. Association for Computing Machinery.