Automated Testing and Resilience of Microservice's Network-link using *Istio* Service Mesh

Rupesh Raj Karn
Khalifa University, Abu Dhabi, UAE
rupesh.karn@ku.ac.ae

Rammi Das
Pokhra University, Nepal
rammidas1234@gmail.com

Dibakar Raj Pant
Tribhuvan University, Nepal
drpant@ioe.edu.np

Jukka Heikkonen
University of Turku, Turku Finland
jukhei@utu.fi

Rajeev Kanth
Savonia University of Applied Sciences, Kuopio Finland
rajeev.kanth@savonia.fi

***Abstract*—Microservices technology has gained considerable popularity in software design to deploy complex applications in the form of micro-modular microservice components. Each service is implemented as an autonomous system, and its internal constituent data can be accessed via a network interface. Such architecture increases the complexity of the network because each module is a separate entity for development and operations. A fault in any service affects the operation of another service and could completely break the application. It is, therefore, necessary to create a framework for the systematic testing and resilience of the network link in microservices, independent of the programming language and business logic. It helps the network administrator track the cause of the fault. In this paper, we have shown the use of the service mesh *Istio* to monitor communication between microservices and to develop automated testing and resilience. *Istio* provides various types of fault injectors for communication links between services. A *Locust* load testing tool is used to exert a microservice load. The faulty link is located via the Jaeger and Grafana dashboard within the *Istio* frame. For resilience or correction of the fault, a new connection is temporarily established between the affected microservice by deploying redundant services. In addition, microservices scaling and the implementation of the circuit breaker have been shown to remedy network congestion. The setup is demonstrated in the Kubernetes cluster with the *Hipster* shop *e-commerce* application.**

I. Introduction

Cloud applications are moving from monolithic architecture to microservice architecture. Microservice is a software development technology that organizes an application to loosely linked services[1] each serving a part of the application. The decomposition of an application into various smaller services has many advantages. It facilitates the application to understand, develop, test, deploy and monitor. In application development, it allows separated teams to independently develop their respective services. New features and updates are continuously added to a service without affecting other parts of an application, making it highly dynamic. It also eliminates the dependency on a single technology stack because individual services can run with different technology if it is loosely connected. Different programming languages can be selected for each service, and communication occurs via remote API calls. As a result, microservices must be resilient and trustworthy in order to facilitate communication. Despite these benefits, microservices have a few limitations. It is difficult to manage communications between services, i.e., service requests should be properly controlled. Any failure at one service can impact the operation of another, and this can extend all the way to the remote user level. As a result, it requires a framework that continuously monitors the communication link across several microservices and recognizes a victim link in the case of a problem. These tests must be automated and methodical, with feedback, so that the application administrator can orchestrate a specific failure, obtain a thorough explanation of the fault, and automatically correct the failure in the future. This may improve the application's resilience—ability to recover from failure. In addition, such feedback makes testing more valuable than a randomized injection of faults by enabling application developers to instantly locate and solve fault management logic. This paper demonstrates such a framework using a popular tool *Istio*.

The service mesh *Istio* [2] is an open-source tool for deploying an application and collecting monitoring signals to capture traces of communication between microservices. Service mesh refers to the microservice network that makes up applications and their interactions. The larger the size and complexity of the mesh, the more difficult it is to understand and manage. The state-of-art that uses *Istio* to manage microservice applications are [3], [4], [5], [6], [7]. A guide to deploy and run the *Istio* is available in [8], [9], [10]. A lecture was given at the SREcon'18 conference organized by USENIX [4] and SIGMOID'18 conferences organized by ACM [11] to demonstrate the use of *Istio* in Kubernetes. A survey paper of G. Schermann et al. [12] have described in their work that *Istio* is a leading tool for building a microservice mesh to run an application and meets complex operational requirements such as testing and proxy-based traffic routing. A survey on the history and future challenges of microservices [13] has also made a similar quote on *Istio*.

In the microservice application, each service has several dependencies between each other. Separate teams working on different microservices continue to add new features to their respective service. It is, therefore, necessary to set up an environment with all dependencies in which functional test

ISSN 2305-7254

cases can run from end to end. Various tools for testing (or loading) *e.g., JMeter, Tsung, Gatling, Grinder, etc.* are available for testing each application microservice, but no detailed network traces, monitoring, and logs are given. Although these tools can locate the fault, the reason for the fault remains unknown. Such details are necessary for resilience in the application. Such information has traditionally been extracted through rigorous manual testing and log collection for each test set. Under this scenario, the expertise of network administrators determines the type of knowledge gained and the amount of knowledge transmitted to their teammates.

In this work, these details are extracted directly from an open-source tool called *Istio*. A load test tool *Locust* has been integrated with *Istio* to build an automated microservice testing and resilience framework. It contains all the related libraries and packages for any microservice application. The resilience is strong if every part of the application handles reasonable errors and defects. Either unavailability, network congestion, latency, caching, or database problems, the use of distributed microservice fulfills these implicit requirements and corresponding handling. The designed framework can test these complications and automatically correct the fault. This work makes the following key contributions:

1) Steps to deploy applications microservice in *Istio* mesh with backend as Kubernetes.
2) Different rules of fault injection mechanisms in microservices.
3) Three types of fault-resiliency mechanisms with their implementation scripts.

The paper is organized as follows. The prior representative publications most related to this work are surveyed in Section II. Section III describes a methodology where the details of *Hipster* application, *Istio*, monitoring and load testing mechanisms are explained. The testing implementations are given in Section IV. The use of the *Locust* tool for fault injection and its detection over the dashboard is explained. The scripts and *YAML* code of each experiment are also presented. The resiliency methods are then explained in Section V. Three types of resiliency methodology, including scaling, failovers, and circuit-breaker, have been demonstrated. Finally, the paper concludes in Section VI.

This work is a part of the first author's Ph.D. thesis titled "Automated Management of Cloud Application Using Machine Learning Techniques" [14].

II. Literature Review

In [15] an empirical method for automated testing has been presented. Software agents are used to capturing each component's behavior for testing specifications. Formal agent specifications and automated testing of microservices are integrated and fed into the automated test engine to run against the actual microservice code. The application for the development of the curriculum *Lasta* has been used as a testbench. Two frames are used, one for unit testing (RSpec) and the other for acceptance testing (Cucumber). A Gremlin framework has been produced in [16] to test microservice failure handling capabilities systematically. It is based on the behavior that microservices exchange messages over the network are loosely linked. It performs microservice testing by manipulating inter-service messages on the network layer. It has a python-based code describing a high-level breakdown scenario and a set of rules on how microservices should react during this breakdown. It has a data plane composed of network proxies to intercept, log, and manipulate messages exchanged between microservices and control planes to configure network proxies for injection of faults based on rules. Another representative work [17] provides an experimental assessment of performance interference between microservices in one or more containers. Experimental evaluations are carried out using HPC-based microservices to investigate the interference problems caused by the location of microservices in single or multiple containers. The result gives a detailed insight into microservice performance variation. The need for new performance engineering solutions for microservices is examined in [18]. Open problems with performance tests are identified, and possible research guidelines for testing, monitoring, and modeling microservices are outlined. In particular, the focus is on strategies for efficient performance regression tests, continuous software change performance monitoring, and appropriate performance modeling concepts for shifted use cases. In [19], an analysis of the existing literature on cloud applications testing has been carried out. A validation methodology for the microservice systems using the Mjolnir platform is proposed on the basis of the analysis. The functional unit, a load unit, security unit, integration, and system are included in the validation set. A learning-based test (LBT) is performed in [20] to assess the functional correctness and robustness of distributed systems to injected defects. It combines machine learning with model checking, integrated with the application in a feedback loop. A distributed microservice architecture for the credit risk analysis of counterparties called triCalculate is used. It claims to be successful in detecting application errors using models with low fidelity inferred, injection, and test case evaluation using LBT wrapper constructs and formal modeling of correctness and robustness. A reusable automated acceptance test architecture for microservices behavior-driven deployment (BDD) is discussed in[21]. The test architecture claims to improve the auditability of BDD steps, their re-usability, and the separation of concerns between developers, testers, and business analysts.

III. Framework Setup and Methodology

A. Hipster Shop applicaiton

An application named *'Hipster shop'* [22] is selected to conduct the experiments in the *Istio* service mesh. Nevertheless, our testing and resiliency mechanism has no dependency with the internal constituent of *Hipster shop* application. Our setup could be effectively used for any cloud applications, e.g. bookinfo [23], Sock Shop [24], E-shop [25], etc. *'Hipster shop'* is a $10 - tier$ web-based e-commerce application where users can browse, add, and purchase products. The microservices of this application are available in python, java,

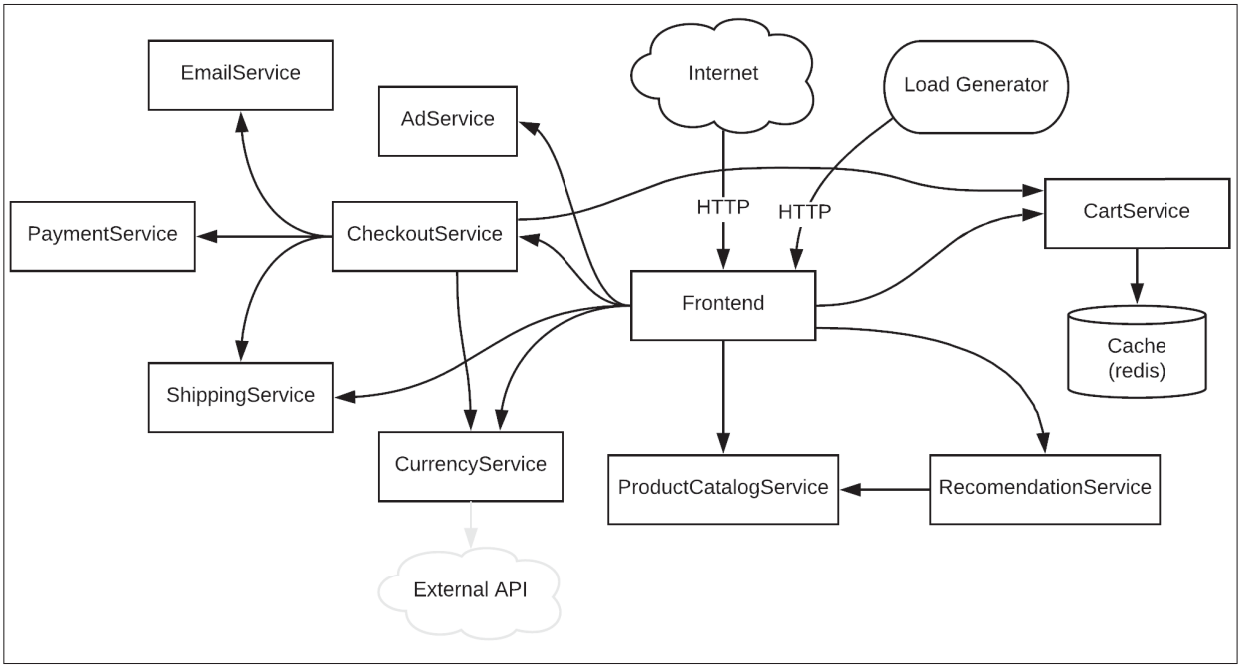

Fig. 1. Hipster shop application deployed in *Istio* service mesh

TABLE I. HIPSTER SHOP MICROSERVICE DETAILS

Service	Language	Description
Frontend	Go	Exposes an HTTP server to serve the website.
Cart service	C#	Stores the items in the user's shipping cart in redis.
Productcatalog	Go	Provides the list of products from a JSON file and search them.
Currency	Node.js	Converts one money amount to another currency. Uses real values fetched from European Central Bank.
Payment	Node.js	Charges the given credit card info with the given amount and returns a transaction ID.
Shipping	Go	Calculates shipping cost based on the selected products. Ships items to the given address
Email	Python	Sends users an order confirmation email.
Checkout	Go	Retrieves user cart, prepares order and orchestrates the payment, shipping and the email notification.
Recommendation	Python	Recommends other products based on products in the cart.
Ads	Java	Provides text ads based on given context words.
Loadgenerator	Python/ *Locust*	Continuously sends requests simulating the user shopping flows to the frontend.

go, c#, and javascript. In Fig. 1, the application setup and communication links between microservices are displayed by an arrow. Table I shows the details of each microservice. Microservices communicate with each other via the gRPC system [26] (grid remote procedure call).

B. Istio Service Mesh

In our experimentation, the *Istio* service mesh has been installed on Ubuntu 16.04 virtual machine. *Istio* platform has backend as Kubernetes, which is set up on a single node cluster using minikube [27]. *Istio* is a platform to deploy, connect, manage, and secure microservices. It provides an easy way to create a network of deployment, load balancing, and monitoring services. It also provides behavioral insights and operational control over the entire service mesh. *Istio* allows describing a network of load balancing services, service-to-service authentication, monitoring, and more in a *YAML* file without any change in service code. YAML (yet another markup language) is a data serialization language mostly used for writing configuration files for an application. Such *YAML* file could easily deploy the application configurations through Kubernetes API *(kubectl create -f <yaml-file-name>)*. A *Istio* support is provided through the deployment of a special sidecar proxy throughout the application. This proxy intercepts all network links between microservices and configures according to the rules defined in the *YAML* file. The list of features that are provided by *Istio* and we have used in our work includes:

- automatic load balancing for HTTP, gRPC, WebSocket, and TCP traffic.
- control of traffic behavior by routing rules, retries, failovers, and fault injection.
- policy layer and configuration API for access controls, rate limits and quotas.
- collect metrics, logs, and traces for all traffic.
- secure communication with strong identity-based authentication and authorization.

C. Monitoring

Tracing, monitoring, and logging provide a deep understanding of the service mesh and how one service impacts upstream

and downstream services. *Istio* provides the Jaeger dashboard [28] to collect each microservice's monitoring metrics and logs. The dashboard also gives visibility to the performance of all services. The monitoring metrics include IP address, port numbers, request type, completion time, the quantity of data sent and received, etc. In addition to the Jaegar dashboard, *Istio* also provides the Prometheus [29] and the Grafana [30] dashboard to view service traffic data. An open-source container monitoring tool *crawler* [31] is used to obtain further information on the processes and connection *URL* information of each microservice. It provides the details of a container (in this case, microservice), including CPU usage, memory usage, process status, number of connections to the active processes, etc.

D. Fault Injection

Purposefully creating events that cause services to malfunction is called fault injection. It is a way to simulate the effect of errors commonly occurring in cloud services, such as forced increases in resource pressure or a network interruption. There is no standard fault injection method—it is a simulation of pushing systems to the breakage point to anticipate the scenario and determine how to respond to such a situation. *Istio* provides the ability in each microservice to change the routing rules and inject faults. We have used two types of fault injection.

1) HTTP Delay: Delays are timing failures that simulate increased network latency or an upstream overloaded service. It injects delays in the communication between two microservices. The delay amount is specified in the *YAML* file. Suppose the request fulfillment time limit is smaller than such a delay, then the application crashes. A bug like this often occurs in enterprise applications where different teams independently manage different microservices. The delay test helps to identify these anomalies without affecting the end-users.
2) HTTP Abort: Aborts are crash failures mimicking upstream service failures. Aborts usually appear in the form of HTTP error codes or TCP connection failures. In contrast, to delay, it aborts the HTTP request immediately and displays user-specified error codes, e.g., Mail *HTTP status 500 internal server error*.

Istio allows the configuration of defects based on specific conditions to match and limits the portion of traffic to be subjected to a defect. For example, in the *YAML* file, the $2sec$ delay between checkout and payment service for user *tomcherry* at 30% traffic and *HTTP abort* at 20% traffic may be described. In this case, 50% of traffic requests from *tomcherry* is normally fulfilled. Further, all the traffic for users other than *tomcherry* is also normally fulfilled. These attributes enable to mimic various failure scenarios such as service failures, service overloads, high network latency, network partitions, inbound traffic limits, etc.

E. Load-testing

It is a kind of performance test that determines the performance of an application under real load. This test shows how the application works when many simultaneous users access the application. An open-source tool *Locust* [32] has been used in this work. During a test, *Locust* generates a large number of virtual users to swarm different application microservices. The behavior of each virtual user is described in a python script (commonly called *locustfile.py*), and the swarming process is monitored in real-time from a web interface. It also allows minimum and maximum waiting time per simulated user to be specified between the execution of tasks. Unlike other load testing tools, *Locust* does not use callbacks for each virtual user [32] and uses a separate lightweight process.

IV. Testing Implementation

A. Application Deployment

Each microservice of *Hipster* application, shown in Fig 1, is described in a *YAML* file. The microservice consists of a deployment and service object. The deployment contains pod details, including image, replica number, CPU, and memory resource allocation for the pod, labels, port number, entry point, etc. It is to be noted that there can be multiple pods in a microservice. Alpine Linux is used for all pods as a base image. Built with *Dockerfile*, this image is pushed into the docker registry. The *URL* for communication with other microservices is added in the *YAML* file as an environment variable. Such *URL* is generated by the label and the exposed microservice's port number. The service object is an abstraction of the pods which define the access policy. The *YAML* service file contains a component called *selector* and *targetPort* which specifies the name and port of the pods the service targets. The *YAML* service file also specifies a service port number to receive a request for access. As shown in Fig 1, the internet is only open to frontend service. Other services, therefore, be accessed externally from the cluster. Such attributes are described in *YAML* as *type*—frontend service with *type: LoadBalancer* while *type: ClusterIP* for other services. Each of the microservice is deployed using Kubernetes API: *kubectl create -f <YAML-file-name>* (in default namespace). After deployment, the *Istio-sidecar-injector* automatically injects *Istio's Envoy* containers into each of the application's pod through the command: *kubectl label namespace default istio-injection=enabled*. The application is accessed using *http://localhost:port-number*, where *port-number* is obtained from *istio-ingressgateway*. The command: *kubectl get services -n istio-system istio-ingressgateway* is run and port-number mapped with port 80 of *istio-ingressgateway* is used.

B. Collecting network traces and metrics

Application is accessed using *http://localhost:port-number* from the browser multiple times to create network traces. Then Jaeger dashboard is opened by *http://localhost:16686*, where the port 16686 is forwarded from Jaeger pod to host using command: *kubectl port-forward -n istio-system (kubectl get pod -n istio-system -l app=jaeger -o jsonpath='.items[0].metadata.name') 16686:16686* . The screenshot of the dashboard in Fig. 2 shows one of the traces of the payment microservice. The trace consists of a collection

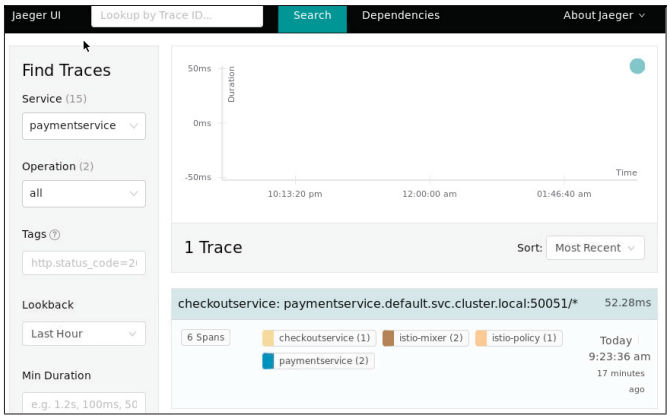

Fig. 2. Jaeger dashboard showing the traces for payment microservice

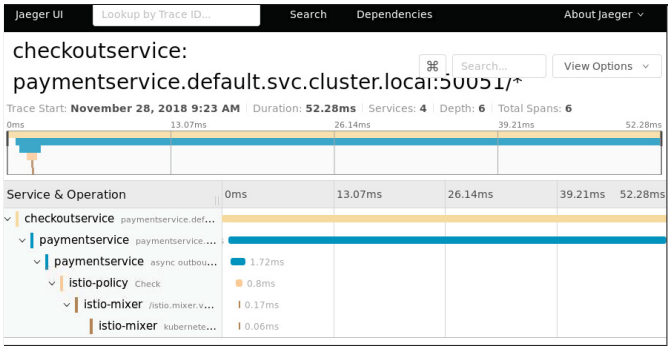

Fig. 3. Jaeger dashboard showing the details of traces for payment microservice

of spans in which each span corresponds to an application service *Hipster shop* and *Istio-envoys* invoked during payment service. As shown in Fig. 1, the payment service only talks to the checkout service, and the length of traces confirms such communication where no other microservices are in the trace. The details of each span are shown in Fig. 3. The trace represents two spans of payment service, one represents a $51.5ms$ client-side span, and the other is a first-span child and represents a $1.72ms$ server-side or outbound span of that tool. The total time to complete the request is $52.28ms$.

Istio captures traces for all requests sent and received at each microservice which could be observed on the Jaegar dashboard. The sampling rate of traces in case of high traffic mesh is changed by modifying the *PILOT_TRACE_SAMPLING* variable in *istio-pilot* deployment file using the command: *kubectl -n istio-system edit deploy istio-pilot*. After injecting $10sec$ delay fault in the payment microservice, the application is again accessed to create traces. The screenshot of the trace in Fig. 4 shows the total spanning time of approximately $10sec$. The start time at checkout microservice is $t = 0ms$, but the traffic reaches to payment service at $t = 10ms$. Time taken at payment service for the inbound and outbound request is $3.54ms$ and $1.38ms$ only. For the demonstration purpose, we have randomly picked the payment microservice and delay of $10ms$. Nevertheless, the testing has been performed similarly for each microservice with random values of delay.

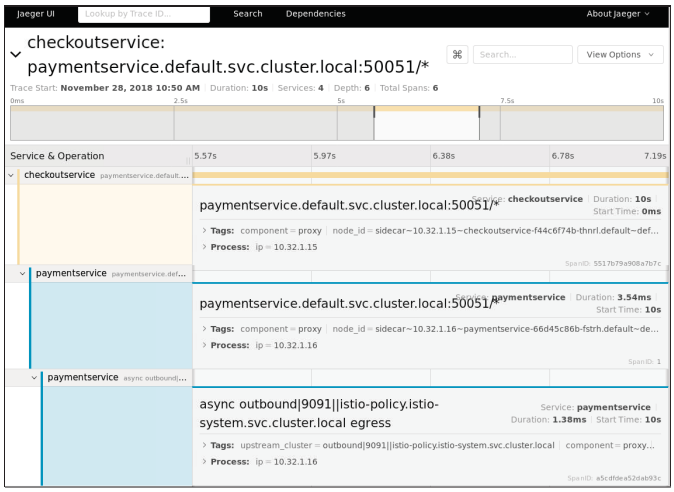

Fig. 4. Jaeger dashboard showing the traces for payment microservice under $10sec$ delay fault

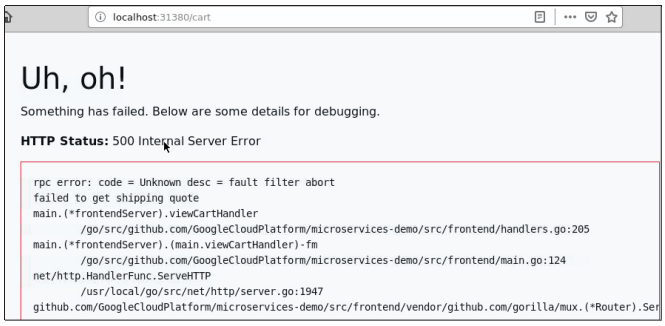

Fig. 5. HTTP abort fault in shipping service

C. Fault Injection

Istio empowers protocol-specific fault injection into the network link between services instead of traditional fault injection methods such as killing pods or delaying or corrupting TCP layer packets. It is based on the principle that the breakdown observed by the application layer is the same as it would experience when the physical network layer fails. The fault injection is written in the *YAML* file using traffic management configuration resources called *VirtualService*. The *YAML* file defines guidelines that control how traffic requests for service are routed within the *Istio* service mesh. The code 1 and 2 in Appendix show a sample of both types—HTTP dealy and HTTP aboard, of fault definition.

Code 1 sets the $7sec$ delay for 50% traffic accessing the microservice product catalog for all users. Code 2 defines abort with *HTTP Status 500* for 80% of traffic received from user $jason$ on shipping microservice while it works normally for other users. These numeric values are randomly chosen to demonstrate the fault injection. The screenshot of the shipping error *Http Abort* is shown in Fig. 5. The second line in the screenshot shows *'failed to receive a quote for delivery'*.

D. Performance-testing

Locust is used to generate load and to test performance. The microservice load generator, displayed in Fig 1, only interacts

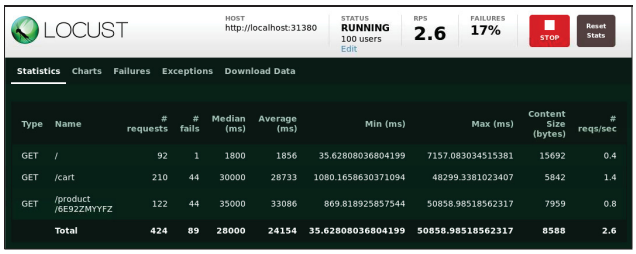

Fig. 6. *Locust* load testing statistics under normal case

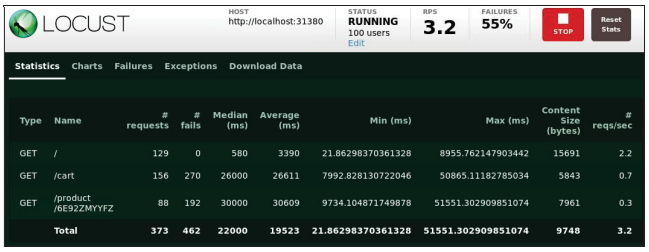

Fig. 7. *Locust* load testing statistics under delay fault

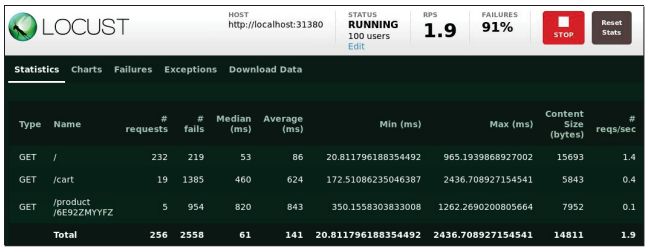

Fig. 8. *Locust* load testing statistics under abort fault

with frontend services. Load testing of other microservices is carried out via frontend service, as this is the application entry point for all external requests. A *locustfile.py* sample is displayed in Code. 3 available in Appendix. It accesses the *Hipster shop* web page, explores the product # *6E92ZMYYFZ* and adds it to the cart. Then it creates 100 virtual users with hatch rate *(users spawned per second)* of 10. The minimum and maximum wait times are specified in milliseconds—per simulated user—between the execution of tasks, and *Locust* choose the time randomly and uniformly between *min_wait* and *max_wait*.

The result of load testing for $2\,min$ is shown in Fig. 6. The top left corner shows the overall request per second and failure percentage. Next, a delay fault of $7sec$ is applied to productcatalogue microservice for 50% of traffic. The result of load testing again monitored for $2\,min$, is shown in Fig. 7. The number of failures for product microservice is 192 with $rps = 0.3$. Comparing that with normal case as in Fig. 6, the failure is only 44 with $rps = 0.8$. Next, the abort fault, *HTTP status 500*, is applied to productcatalogue microservice for 50% of traffic. The outcome is more serious than shown in Fig. 8. The abortion fault severely affected the cart and frontend network connections. The statistics also change when the percentage of traffic is changed for a fault.

The detailed *Locustfile* for complete order generation process

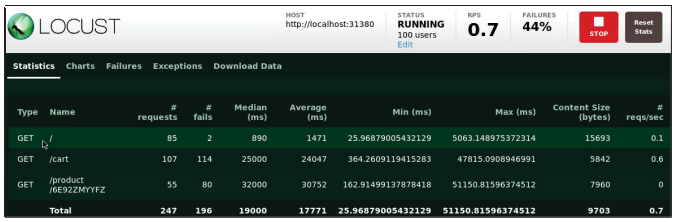

Fig. 9. *Locust* load testing statistics for one pod in productcatalogue and cart microservice

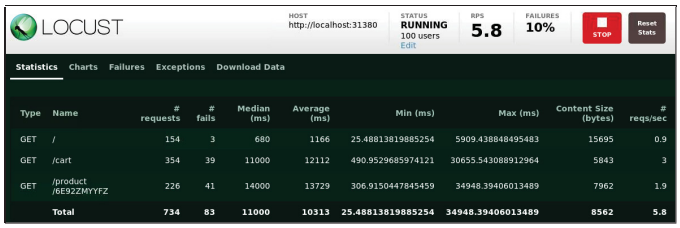

Fig. 10. *Locust* load testing statistics for two pods in productcatalogue and cart

is given in [33]. This is run for complete end-to-end testing of each microservice. The number of failures is viewed on *Locust GUI* as shown in Fig. 6 to 10. Once the failure rate of any step in order creation increases, then the cause is viewed on the Jaeger and Graffana dashboard as mentioned in step IV-B and Fig. 2 to 4. These tools automatically capture the cause of the fault and display it on the dashboard.

V. Resilience Implementation

The remedy of the faults that are generated in earlier sections are explained below:

A. Scaling

Accessing the application by several users creates congestion in the network. For each microservice, the extent of such congestion is different. The traditional remedy for congestion in the network is the scaling of the respective microservices. To demonstrate this remedy, an experiment has been conducted by scaling the number of a pod to analyze if there are any changes in *Locust* statistics. It turned out that the scaling does not affect the failure rate after delay and abort fault injection. It is because the defects are injected into the microservice network layer. An increase in the number of replicas does not add value. Under healthy conditions (without fault injection), the failures shown in the *Locust* GUI must be caused by the network congestion for the specific microservice. The number of pods associated with such microservice needs to be scaled up. To show the performance comparison after scaling, an experiment is executed with *locustfile* displayed in code 3, given in Appendix, with 100virtual users, $spawning\ rate = 10$, and *Locust* GUI monitored for $2\,min$. Every microservice runs an application with a single pod in the first case. The product catalog and cart microservice are scaled in the next case. Fig. 9 and 10 show the results of the two cases. Figs. 10 shows that the failure rate has decreased, and a large number of requests are processed in a $2\,min$ interval.

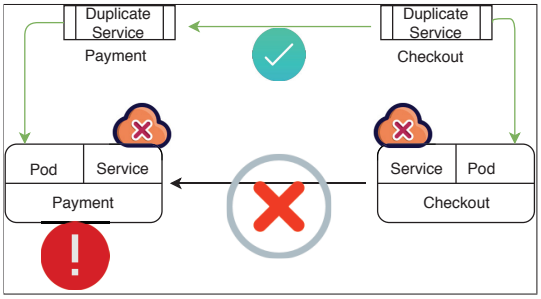

Fig. 11. Architectural change in *Hipster* application to perform connection fail-over for payment service. Only two microservices of the application*hipster* are displayed. The green arrow shows newly created connections and services.

B. Service Connection Failover

The term failover is a procedure for automatically transferring control to a duplicate system when a microservice detects a fault. As explained in the IV-A section, each microservice consists of a pod and a service. Microservices communicate via the *URL* connection where the *URL* contains the service name and port number. Consider that traffic originates from microservice A containing [*service-A*, *pod-A*] and directed to other microservice B containing [*service-B*, *pod-B*]. If the network link between A and B fails, then to remedy the link, only the service duplication is required. The duplicate service named *duplicate-service-A* and *duplicate-service-B* can still use *pod-A* and *pod-B*, respectively. Nonetheless, the pod can be duplicated as well. However, reusing the deployed pod protects the Kubernetes cluster's computational resources from being wasted. It should be noted that the pod portion of the microservice consumes the majority of the cluster resources in terms of CPU, memory, and storage.

One of the example is shown in Fig. 11, where *A:checkoutservice* and *B:paymentservice*. The service part of both microservices is duplicated with different labels once the fault in the network connection is detected, as shown in Fig. 14 in Appendix. Payment service is duplicated with *kubectl create -f duplicate-paymentservice.yaml* with a new name as marked by a white arrow, and similarly, the checkout service is duplicated. The new *URL* of the duplicated service is updated in the pod deployment file of checkout microservice as shown by the black arrow in Fig. 14. This modified *YAML* definition only updates the connection *URL* of payment service through command: *kubectl apply -f modified-checkoutservice.yaml* without disturbing other configurations of the deployment. Old service, as cross marked in Fig. 11, is throttled temporarily until the issue with the old network link sustains. After the fault is resolved, the throttled services are turned on using the command: *kubectl apply -f <old-YAML-file-name>* and remove the duplicated service through *kubectl delete -f duplicate-paymentservice.yaml*.

C. Circuit Breaker

A cascade failure is possible if one or more microservices fail due to high latency or any other issue in a multi-tier application. In such a case, the failure of one microservice affects the

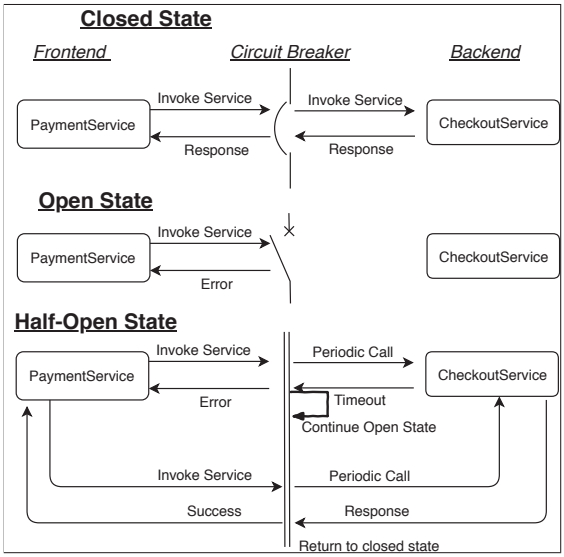

Fig. 12. Circuit breaker block diagram

performance of the next one in the tier, and eventually, it reaches the frontend service, where the end-user experiences application access blockage. Most of the time, retry logic is used to correct a cascade failure with the premise that the fault is transient, short-lived, and would resolve itself after repeated tries with a short time gap. In a microservice failure, retry logic might worsen the scenario even further and put the entire application to a halt. It is because too many retries increase traffic and pressure on the healthy microservice, making it prone to request queue overflow and network congestion. The circuit breaker helps to prevent such a malfunction across multiple services, and it allows the development of a resilient system that can survive when major services are not available. We have used three types of the circuit breaker as shown in Fig. 12:

1) Closed: The circuit breaker is in a closed state in a healthy case and is transparent for all calls between services. If the number of failures exceeds a defined limit, the breaker tips and enters an open state.
2) Open: It sends an error message to all requests addressed to the back-end service without executing the function call.
3) Half-Open: After the specified timeout period, the circuit switches to a half-open state to verify that the problem is corrected. If the problem still exists, the breaker returns once more to the open state; otherwise, it returns to the closed state.

In the *Hipster shop* application, a circuit breaker s placed between the checkout and payment microservice. As shown in Fig. 12, the checkout microservice is at the back of the circuit breaker. The breaker definition *YAML* is displayed in the code snippent 4, available in Appendix. In the *YAML* definition, the *maxConnections* variable threshold represents any excess connection above this in the pending state in queue. Similarly, the *http1MaxPendingRequests* variable is the maximum number

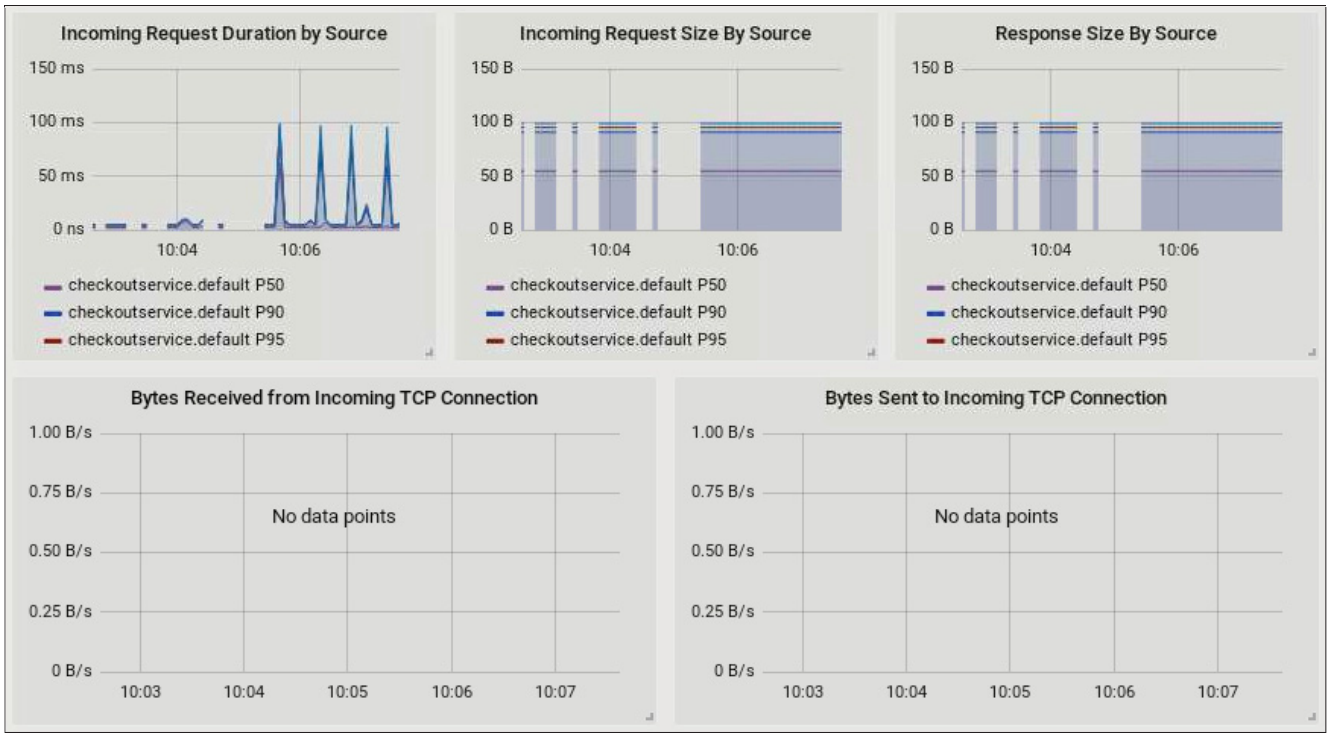

Fig. 13. Traffic details in Graffana dashboard. Circuit break between checkout and payment microservice

of requests pending in the queue. Any excess requests pending will be discarded. The number, as an example in the code snippet 4 shows that more than 2 requests are discarded at once for the payment service. Furthermore, the 2 consecutive errors in less than $1sec$ cause the circuit breaker to trip into the open state and eject the payment microservice pod from the load balancer for $3\,min$. To test this definition, the *Locust* fault testing code given at [33] is run at 10 users per second. This caused the network congestion error, and finally, the circuit tripped to open state. The result is viewed in the Graffana dashboard. Screenshot in Fig. 13 shows zero bytes received and sent to/from payment microservice even if requests are sent from the checkout service. We have only shown the use of a circuit breaker for checkout and payment processes with randomly selected users and a delay interval for demonstration purposes. Nonetheless, the same technique is used to test different microservices of *Hipster shop* application.

VI. Conclusion

In this paper, an automated testing and resiliency methodology for the distributed architecture of the application is shown. A service mesh *Istio* has been used to monitor traces between microservices to develop automated testing and resilience. Various types of fault injectors, including delay, traffic limiting, and abort, are used between services. A *Locust* load testing tool is used to create virtual users for load testing. The faulty microservice link is located through several dashboards of *Istio*. Three types of mechanisms are used for resilience or correction of the fault, including scaling, failover, and circuit breaker. The testing and resiliency setup can be effectively used for network troubleshooting and performance measurement for any cloud application during traffic congestion and fault.

Acknowledgment

The first authors would like to thank IBM Research for hosting him at the IBM T. J. Watson Research Center, Yorktown Heights, NY, to perform the research works given in this paper.

Appendix

A. Source code Snippets

Code Snippet 1. Delay fault.

```
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: productcatalogservice
spec:
  hosts:
  - productcatalogservice.default.cluster
  http:
  - fault:
      delay:
        fixedDelay: 7s
        percent: 50
    route:
    - destination:
        host: productcatalogservice
  - route:
    - destination:
```

Code Snippet 2. Abort fault.

```
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: shippingservice
spec:
  hosts:
  - shippingservice.default.cluster
  http:
  - fault:
      abort:
        httpStatus: 500
        percent: 80
      match:
      - headers:
          end-user:
            exact: jason
    route:
    - destination:
        host: shippingservice
  - route:
    - destination:
        host: shippingservice
```

Code Snippet 3. *Locustfile*.

```
from locust import HttpLocust
from locust import TaskSet,task
class UserBehavior(TaskSet):
    @task(1)
    def frontend(self):
        self.client.get("/")
    @task(2)
    def productcatalogue(self):
        self.client.get("/product
        .../6E92ZMYYFZ")
    @task(3)
    def cart(self):
        self.client.get("/cart")
class WebsiteUser(HttpLocust):
    task_set = UserBehavior
    min_wait = 3000
    max_wait = 6000
```

Code Snippet 4. Circuit breakers.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: paymentservice-circuit
  namespace: default
spec:
  host: paymentservice
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 1
      http:
        http1MaxPendingRequests: 1
        maxRequestsPerConnection: 1
    outlierDetection:
      consecutiveErrors: 2
      interval: 1s
      baseEjectionTime: 3m
      maxEjectionPercent: 100
```

B. Future Work - Causal Relationship Extraction

It becomes increasingly difficult to understand how information flows across different services in applications with many microservices. More information regarding the location of possible checkpoints is needed. Also, in a commercial application, the cause of delay experienced by a user can be either due to an artifact of the network or a microservice in the call chain. It is needed to establish the caller and callee relationship between microservices. Such types of knowledge can be retrieved from causal relationship [34] between request-response pairs across microservices. The concept of causal analysis has been retrieved from Six-Sigma projects for root cause analysis. The Pareto diagram is the most known tool to analyze categorical information on root causes to identify the major contributors. Also, many statistical tests such as the T-test, F-test, analysis of variance (ANOVA), and chi-square test have been used effectively for cause-effect relation. For cloud applications, the cause-effect relationship is mostly built by the system administrator from domain knowledge. The properties of each service are extracted, and they are mapped using graphical symbols to show the causal relationship. In the faulty condition, the administrator's intuition and his knowledge base determine the root cause. This approach has a few disadvantages as below:

Fig. 14. Connection fail-over for payment service. Change in *YAML* file is marked by an arrow, and a payment backup service is created as shown in the top right corner.

1) Over dependency on network administrator to locate the fault.
2) Difficult to maintain the knowledge base for the behavior of each microservice if they change continuously over time.
3) A graphical model is relatively more complex to build and difficult to interpret than a quantitative model, e.g., machine learning, regression, etc.
4) Creating a script for monitoring several signals could be cumbersome.

A quantitative model using machine learning analytics is

the superior option to develop a causal relationship among microservices of a distributed application to remove such issues. A call chain relationship across services could be derived by analyzing a large set of messages or by transaction tracing. Using statistical measures, the cause-effect relationship between each possible combination of the microservices of the application can be built. A python causality library at https://pypi.org/project/causality/ could be a viable option to try out. The first task would be to generate a set of relevant metrics or features corresponding to microservice operation from several dashboards of *Istio* and crawler [31] to run baseline experimentation. We will cover this part in our follow-up publications.

REFERENCES

[1] W. H. C. Almeida, L. de Aguiar Monteiro, R. R. Hazin, A. C. de Lima, and F. S. Ferraz, "Survey on microservice architecture-security, privacy and standardization on cloud computing environment," *ICSEA 2017*, p. 210, 2017.

[2] https://istio.io/docs/concepts/what-is-istio/, accessed: 2018-08-16.

[3] O. Sheikh, S. Dikaleh, D. Mistry, D. Pape, and C. Felix, "Modernize digital applications with microservices management using the istio service mesh," in *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, 2018, pp. 359–360.

[4] F. Moyer, "Comprehensive container-based service monitoring with kubernetes and istio," 2018.

[5] R. Sharma and A. Singh, "Istio gateway," in *Getting Started with Istio Service Mesh*. Springer, 2020, pp. 169–192.

[6] R. Harabi, "Managing microservices with istio service mesh," 2019.

[7] M. Song, Q. Liu, and E. Haihong, "A mirco-service tracing system based on istio and kubernetes," in *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*. IEEE, 2019, pp. 613–616.

[8] L. Calcote and Z. Butcher, *Istio: Up and Running: Using a Service Mesh to Connect, Secure, Control, and Observe*. O'Reilly Media, 2019.

[9] A. Khatri and V. Khatri, *Mastering Service Mesh: Enhance, secure, and observe cloud-native applications with Istio, Linkerd, and Consul*. Packt Publishing Ltd, 2020.

[10] R. Sharma and A. Singh, "Installing istio," in *Getting Started with Istio Service Mesh*. Springer, 2020, pp. 99–136.

[11] E. Brewer, "Kubernetes and the new cloud," in *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018, pp. 1–1.

[12] G. Schermann, J. Cito, and P. Leitner, "Continuous experimentation: Challenges, implementation techniques, and current research," *IEEE Software*, vol. 35, no. 2, pp. 26–31, 2018.

[13] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.

[14] R. R. Karn, "Aautomated management of cloud application using machine learning techniques," Ph.D. dissertation, Khalifa University of Science and Technology, Abu Dhabi, UAE, 2019.

[15] J. G. Quenum and S. Aknine, "Towards executable specifications for microservices," in *2018 IEEE International Conference on Services Computing (SCC)*. IEEE, 2018, pp. 41–48.

[16] 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)*. IEEE, 2016, pp. 57–66.

[17] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li, and R. Ranjan, "A holistic evaluation of docker containers for interfering microservices," in *2018 IEEE International Conference on Services Computing (SCC)*. IEEE, 2018, pp. 33–40.

[18] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. Wettinger, "Performance engineering for microservices: research challenges and directions," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ACM, 2017, pp. 223–226.

[19] D. I. Savchenko, G. I. Radchenko, and O. Taipale, "Microservices validation: Mjolnirr platform case study," in *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on*. IEEE, 2015, pp. 235–240.

[20] K. Meinke and P. Nycander, "Learning-based testing of distributed microservice architectures: Correctness and fault injection," in *International Conference on Software Engineering and Formal Methods*. Springer, 2015, pp. 3–10.

[21] M. Rahman and J. Gao, "A reusable automated acceptance testing architecture for microservices in behavior-driven development," in *2015 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. IEEE, 2015, pp. 321–325.

[22] https://github.com/GoogleCloudPlatform/microservices-demo, accessed: 2018-08-16.

[23] https://github.com/istio/istio/tree/master/samples/bookinfo, accessed: 2021-08-16.

[24] https://github.com/microservices-demo/microservices-demo, accessed: 2021-08-16.

[25] https://github.com/dotnet-architecture/eShopOnContainers, accessed: 2021-08-16.

[26] G. Sheerin and G. Krabbe, "Sre your grpc—building reliable distributed systems (illustrated with grpc)," 2017.

[27] https://github.com/kubernetes/minikube, accessed: 2018-08-16.

[28] https://istio.io/docs/tasks/telemetry/distributed-tracing/, accessed: 2018-08-16.

[29] https://prometheus.io/, accessed: 2018-08-16.

[30] https://grafana.com/, accessed: 2018-08-16.

[31] https://github.com/cloudviz/agentless-system-crawler, accessed: 2018-08-16.

[32] https://locust.io/, accessed: 2018-08-16.

[33] https://github.com/rkarn/automated-testing-resciliency/blob/master/locustfile-complete.py.txt, accessed: 2018-08-16.

[34] S. Hassan and R. Bahsoon, "Microservices and Their Design Trade-offs: A Self-adaptive Roadmap," in *2016 IEEE International Conference on Services Computing (SCC)*. IEEE, 2016, pp. 813–818.