Fitness-guided Resilience Testing of Microservice-based Applications

Abstract—**Modern distributed applications are moving toward a microservice architecture, in which each service is developed and managed independently, and new features and updates are delivered continuously. A guiding principle of microservice architecture is that it must be built to anticipate and mitigate a variety of hardware and software failures. In order to test the fault handling capabilities of microservices automatically, this paper presents IntelliFT, a guided resilience testing technique for microservice based applications, which aims to expose the defects in the fault-handling logic effectively within a fixed time. The characteristic of IntelliFT is that it leverages existing integration tests of the applications under test to explore the fault space, and decides whether injected faults can lead to severe failures by designing fitness-guided search technique. Our experimental results on a medium-size microservice benchmark system show that the proposed technique is effective, improves the state-of-the-art, and can quickly expose bugs in the recovery logic.**

Index Terms—**microservice, fault injection, fault handling**

I. Introduction

Microservice architecture [17] is gaining more and more popularity in the industry when designing complex, cloud-based distributed systems. In this architecture, the application consists of small, loosely coupled, mono-functional services that communicate using REST-like interfaces over the network. Each microservice is developed, deployed and managed independently; new features and updates are delivered continuously, resulting in polyglot applications that are extremely dynamic and rapidly evolved. Many organizations, like Netflix, Amazon, eBay and Twitter, have already evolved their applications to microservice architecture.

In order to provide an "always on" experiences to customers, microservice should be designed to anticipate, detect and withstand various runtime failures and outages from its dependencies, and struggle to remain available when deployed. However, it is difficult to ensure that such fault-handling code is adequately tested. In the past, many popular highly available Internet services have experienced various failures and outages (e.g., cascading failures due to database overload), and according to the post-mortem reports, most of such outages were caused by missing or faulty recovery logic, with an acknowledgement that unit and integration tests are insufficient to catch bugs in the fault handling logic [14].

Chaos engineering [9], the practice of performing fault injection experiments on the production system, to increase the overall resilience of a software system, is emerging as a discipline to tackle the resilience of large-scale distributed system. It was pioneered by Netflix, which developed Chaos Monkey [2] to inject faults randomly in production system to test the resiliency of the AWS. Since then, Linkedin, Microsoft and Uber have developed fault injection framework to improve the resilience of their systems at scale.

However, random fault injection technique is not efficient, and lots of time and resources will be wasted to explore redundant failure scenarios. It is also unlikely to uncover deep failures involving combinations of different instances and kinds of faults. To address these issues, Alvaro et al. [7] proposed lineage-driven fault injection (LDFI) technique, to discover bugs in fault tolerant protocols/systems. It combines data lineage from database literature and satisfiability testing to infer backwards from correct system outcomes to determine whether injected faults in the execution could prevent the outcome. They also adapted LDFI and implemented a research prototype to automate failure testing of Netflix microservice platform [6].

Although LDFI technique [7] has achieved promising results for testing the resilience of microservice-based applications, there are still much room for improvement. Firstly, the computed injection points based on observed service call chains may contain some redundant or unnecessary fault injections, and can be optimized further to reduce fault space exploration (we will describe this in section III.C). Secondly, it focuses on exposing fault-handling bugs in the execution of individual user request in isolation. Considering the large number of state space of the application and complex dependencies between microservices, it is still not known how to quickly expose severe fault-handling bugs within limited testing time.

To address above limitations, in this paper, we present IntelliFT, a fitness-guided, automated resilience testing technique for microservice-based applications, which aims to maximize bugs located in the fault handling logic within a fixed amount of time. Our approach works as follows: 1) It firstly leverages the integration tests that are usually accompanied within the application under test (AUT) to compute initial injection points based on LDFI technique; 2) It then designs a fitness guided fault exploration technique to search for high impact fault tests based on the effect of previously injected faults, and chooses new fault scenarios (that have more chances to expose fault-handling bugs) in the subsequent tests. During the fault exploration, it further optimizes the injection points computed by LDFI technique to reduce unnecessary fault injection. IntelliFT leverages domain-specific knowledge contained in the integration tests to rank the impact level of injected faults for the end user. This process continues until a specified condition is reached, such as exposed severe bugs, or a time limit.

As far as we know, we presented the first search-based technique for the resilience testing of microservice-based applications based on the historical feedback during the fault space exploration. Generally, this paper makes the follow contributions:

- We design a novel fitness-guided, automated resilience testing technique for microservice based applications, which leverages the effect of previous fault injections to guide fault space exploration effectively;
- We propose two heuristic rules which can avoid unnecessary fault injections by observing the propagation of injected faults and testing results;
- Leveraging existing service mesh framework —Istio [1]), we implemented a prototype called IntelliFT, to explore the fault space of microservice-based applications automatically, and the experimental results show that proposed technique is effective.

II. Background and Definition

A. Fault Type

Due to microservice's ploygl ot nature, it is impractical to simulate and inject faults inside the implementation of each microservice. In microservice-based applications, as the response to a user request is the result of a series of interactions between microservices that communicate over the network, intelliFT confines the simulated faults to the failures that are observable from the network by other microserivces. Currently, based on two basic fault types: *abort* and *delay* (provided by Istio [1]), our approach also supports to simulate four commonly occurred failures: *overload*, *hang*, *disconnect*, *crash*. We will describe how to implement these faults in section IV.

B. Injection Point

Our approach leverages distributed tracing component (e.g., Jaeger [5]) that provides the ability to trace requests through their interactions with distributed services and find all of the injection points along the chain. An *injection point* specifies the positions (or the edges) in the service call chain where a fault is injected. It can be simple or complex. A simple injection point is represented as a triple: $\langle src, dst, api \rangle$, where src and dst represent the source and target microservice, respectively, and api represents the API of dst that is invoked by src. A complex injection point is a combination of simple injection points.

Based on user requests and corresponding service call chains, Alvaro et al. [6] adapts lineage-driven fault injection technique (which is proposed for discovering bugs in fault-tolerant data management systems) to compute the minimum set of injection points. Lineage-driven fault injection (LDFI) [7] is a technique which leverages data lineage in the database community and satisfiability testing to reason backwards from correct system outcomes to determine whether some combinations of faults could have prevented the outcome. LDFI is based on two key insights. The first is that fault-tolerance is redundancy, and a fault-tolerant system/program can provide alternative way to obtain an expected outcome in the presence of some common faults (e.g., component failure). The second insight is that instead of exhaustively exploring the space of all possible executions from initial state, a better strategy to quickly expose fault-tolerance bugs is to start with successful outcomes and reason backwards, to understand whether some combination of faults could prevent the outcome.

Generally, LDFI technique works as follows: firstly, the system under test is evaluated by performing a failure-free execution. Then by analyzing why successful outcome is achieved, lineage graph can be extracted, which is further converted into CNF formula that is passed to a SAT solver to generate failure hypothesis. The solved combination of faults will be transformed into inputs that try to falsify the successful execution in the next round of the loop. This process continues until either a fault tolerance bug is identified or the system exhausts its resources.

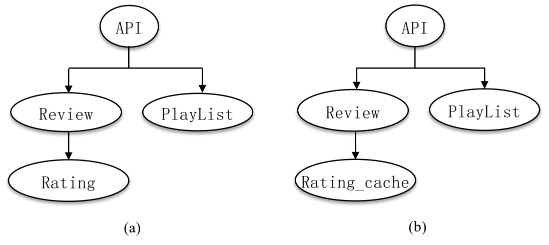

Fig. 1. service call graph without/with fault injection

Netflix also adapted LDFI technique to enable automated failure testing of microservice based applications [6]. We explain how it works using a concrete example. Figure 1(a) shows a failure-free service call chain for a user request. The execution path is first transformed into formula: $API \vee Review \vee Rating \vee PlayList$. The solution to this boolean represents the set of positions that we should test via fault injection. Using SAT, the minimal solutions based on current observed execution are: $\{API\}$, $\{Review\}$, $\{Rating\}$ and $\{PlayList\}$. Each solved set is referred to as an injection point in this paper (for simplicity, we use dst service to represent an injection point here), and combined with fault type, it can be used to create different *failure scenarios* — the sets of injection points into which different fault (e.g., abort, message delay) will be injected. In the next loop, LDFI chooses an injection point (e.g., $\{Ratings\}$) from the current solutions, and injects a fault. This time, after sending the request, the user still gets the successful response. Figure 1(b) shows the corresponding service call graphs after injecting a fault in *Rating* service. It can be seen that when *Rating* service fails, a backup path ($Review$ service calls $Rating_{cache}$) will appear to provide high reliability. The execution path is encoded as $API \vee Review \vee Rating_{cache} \vee PlayList$, which is conjuncted with the previous formula: $API \vee Review \vee Rating \vee PlayList$. Using SAT solver, the minimal solutions to invalid both execution paths are: $\{API\}$, $\{Review\}$, $\{Rating, Rating_{cache}\}$ and $\{PlayList\}$. Based on the solved solutions, LDFI will continue to explore different failure scenarios to find the defects in the fault handling logic.

C. Fault Test and Fault Space

IntelliFT leverages existing integration tests accompanied with the application to do fault injection. A *fault test* specifies the test case in which a fault is injected to try to expose the fault-handling bugs during the execution. It is defined as a vector consisting of four attributes: $\langle tc, req, ip, ft \rangle$, where tc represents the test case, req denotes the user request sent in tc, ip represents an injection point that can be computed based on the service call chain corresponding to req, and ft denotes the fault type that will be injected in ip. We introduce TC, REQ, IP and FT to represent the finite set of each axis, respectively. Therefore, the fault space of AUT can be defined as $\Phi = TC \times REQ \times IP \times FT$, denoting all the combinations of the values from each axis of the fault test vector.

Giving limited testing time, our approach is to find those fault tests that can expose high impact fault-handling bugs as sooner as possible. To achieve this, we propose a fitness-guided fault space exploration technique. In the following, we describe our approach in detail.

III. The Overall Approach

Figure 2 shows the overall framework of the proposed fault space exploration technique, and it generally consists of the following steps:

1) Execute fault-free integration tests of AUT, and compute the initial set of injection points for each request in the tests.
2) Randomly generate a batch of fault tests, execute them and evaluate their impacts;
3) Choose a previously fault test ϕ which has high impact value;
4) Mutate one of ϕ's attribute to obtain a new test ϕ', execute it and evaluate its impact;
5) Repeat step 3;

The proposed fitness-guided exploration technique is similar to the common strategy adopted when playing minesweeping game, which starts by clicking randomly some cells in the grid. Then according to the exposed hints, the player continues to click in the neighborhood of those exposed cells to guess the orientation or the position of the mine until all the mines are marked successfully. Next, we elaborate the key components in the approach.

A. Correlate request with service call chain

In order to correlate user request with the corresponding service call chain, during the execution of test cases, IntelliFT deploys a proxy at the client side to capture user request/response and their corresponding timestamps. At the server side, IntelliFT will trace the internal service invocations leveraging distributed tracing technique (e.g., Zipkin, Jaeger). Based on the logged timestamps of the requests/responses at the client and the service call chains at the server, IntelliFT will correlate the frontend's request/response with the corresponding service call chain in the backend server. Note that, user requests for the static resource files (e.g., image, js, css) will be filtered firstly as most of such responses will be returned directly from the server without triggering the backend's service call chain.

B. Compute Impact Value

In order to find high-impact fault tests automatically, IntelliFT computes the impact value of a fault test execution in terms of the execution result of tc, the response of user request req, and new discovered execution path (which can be considered as the backup path to handle injected fault). Generally, we define the impact of a fault test considering the following fives cases:

- **Case 1: Both req and tc succeed without backup path exposed**. It means that the service call chain of req is robust to handle ft. However, no new execution path is exposed in the service call chain. The impact of such ϕ is minimum, and is set to 10;
- **Case 2: Both req and tc succeed with backup path exposed**. It means that the service call chain of req is robust to handle ft with backup recovery logic, which is worthy to explore further. The impact of ϕ is set to 40 – a high score;
- **Case 3: req fail but tc succeed**. It means that the functionality of req is not critical in tc, and even though it fails, it does not have severe impact on the application functionality. The impact value of such fault test is set to 20;
- **Case 4: req succeed but tc fail**. It means that the corresponding service call chain of req has recovery logic, but the recovery code has defect, leading to the failure of tc in the end. The impact value of such fault test is set to 40;
- **Case 5: both req and tc fail**. It means functionality of req is critical in tc, and its failure will lead to the failure of tc. The impact value of such fault test is set to 50;

Note that, our technique does not consider the backup path for case 3-5, and the reason is that in case either req or tc fails, it is unnecessary to explore the backup path further (even if it appears in the service chain).

To evaluate whether a user request is handled successfully, IntelliFT currently relies on the status code ("200 OK") of the response as an indication of the success. To evaluate the execution result of a fault test ϕ, our approach checks whether test case tc can run successfully after injecting a fault. For integration test tc, it mainly consists of two kinds of operations: test action and test assertion. Test action denotes the user action (e.g., click a button, input some texts), and test assertion specifies the expectation to the content in the page.

When executing each test operation, it is usually required that the target element (involved in the operation) should be located firstly (using various DOM-related locators, such as id, XPath and name). However, an injected fault in req may incur different DOM changes to the page compared with fault-free processing of req, leading to the failure of identifying target element.

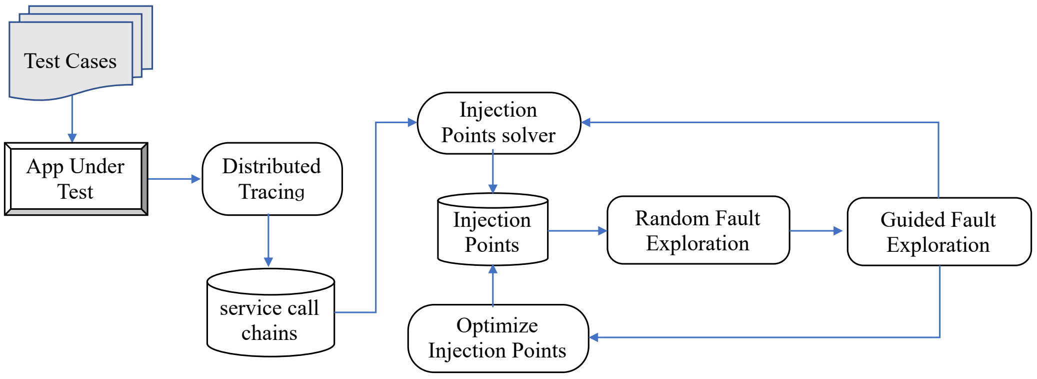

Fig. 2. The overall framework of IntelliFT

In order to robustly locating web element involved in each operation (even though a fault is injected in the backend), when executing fault-free tc in the first step, our approach also extracts image of the target element in each test operation. When executing fault test, IntelliFT first locates target element using original DOM locator, and in the case of failure, it adopts *visual locator* proposed in the work [19] to identify target based on the extracted image. The advantage of using *visual locator* is that it does not depend on the internal DOM tree structure when locating element, and can identify target element as long as it appears on the page.

Leveraging computer vision enhanced test execution, IntelliFT confirms the execution failure of ϕ from the following perspectives: 1) if target element involved in the test operation cannot be located after applying *visual locator*, ϕ fails; 2) if an exception is thrown when executing test action in tc, ϕ fails; 3) If an assertion is violated in tc, ϕ fails.

C. Fitness-guided Fault Space Exploration

As described in the overall approach of our technique, the proposed fault space exploration algorithm first randomly generates a fault test for each test case. Then based on the impact of the result, it enters into the second stage: fitness-guided fault space exploration. Algorithm 1 shows the detailed steps.

In order to choose which fault test to mutate, the algorithm uses three queues: a priority queue $Q_{priority}$ which saves the executed high-impact tests, a queue $Q_{pending}$ which saves the fault tests that have been generated but not yet executed, and a queue $Q_{History}$ which contains all previously executed tests. Once a test in $Q_{pending}$ is executed, its impact is evaluated and gets moved to $Q_{priority}$ if needed.

$Q_{priority}$ has limited size. Whenever its size is reached, a fault test will be dropped from the queue, sampled with a probability inversely proportional to its fitness. The fitness of a fault test is initially equal to its impact value, but then decreases over time. The test which has low fitness will have a higher probability to be dropped. We also set a threshold for $Q_{priority}$, and once the fitness of the test drops below the threshold, it will be removed from the $Q_{priority}$ directly. The "aging" mechanism enables IntelliFT to cover more fault tests accompanied with the improvements in the total impact value. Without aging, the exploration algorithm may get stuck in a high-impact vicinity, while failing to find any new high-impact fault tests.

On line 1-4, IntelliFT picks a parent test ϕ from $Q_{priority}$, to mutate into the offspring ϕ'. Instead of always selecting the highest fitness test, the algorithm samples $Q_{priority}$ with a probability proportional to fitness value – high fitness test is favored, but other fault tests still have chances to to be picked.

In order to select an attribute of ϕ to generate mutation, on lines 5-6, we choose attribute α_i with a probability proportional to axis A_i's normalized gain. We use *Gain* to capture the history of fitness gain: the gain of each axis A_i reflects the history benefit of modifying attribute α_i when generating a new fault test. The gain of A_i is computed by summarizing the fitness value of the previous *n* fault tests in which attribute α_i was mutated. This sum helps to detect "*impact orientation*" in the currently sampled vicinity. If mutating A_i can find more fault-handling bugs than other axises, then we expect this sum of previous fitness values – the gain to the mutations along A_i – to be high as well, otherwise not. Our use of *Gain* is similar to the fitness-gain computation in Fitnex [20], since it is essentially corresponds to betting on choices that have proven to be good in the past.

Gain guides the choice of which attribute α_i to mutate. After that, on line 7, the algorithm invokes corresponding mutator function according to selected attribute. Currently, IntelliFT defines four types of mutators for each attribute in ϕ, that is, test case mutator, request mutator, injection point mutator and fault type mutator.

1) **Test Case Mutator**. It mutates the tc attribute of ϕ to select a test case tc' which also contains the req. To facilitate mutation, in the first step of our approach, IntelliFT classifies requests according to whether two requests have the same service calls based on collected service call chain (Note that, we ignore the structure when comparing two service call chains). Each request class also has a list, which saves the test case that contains the same request. Test case mutator queries this list to select a new test case tc'. If no other test cases contains such request, IntelliFT will apply request mutator.

2) **Request Mutator**. It mutates the req in ϕ to select another request req' in the same test, in which injection point ip of req also exists in req'. To facilitate this mutation, for each test case, IntelliFT maintains a mapping from the injection point to the request, where the key is ip, and the value is a list of requests. It first finds req' in tc containing this ip and then in other test cases. For the later case, the impact value of ϕ' will be divided equally between the gain value of TC and REQ axises. If no such request exists, IntelliFT will randomly generate a fault test.
3) **Injection Point Mutator**. It mutates the ip in ϕ to select a new injection point.
4) **Fault Type Mutator**. It mutates the ft in ϕ to select a new fault type.

Next, we describe how each mutator mutates the chosen attribute in ϕ in order to obtain a new fault test. One intuitive way is to randomly select a new attribute value when applying corresponding mutator. For example, for *test case mutator*, it randomly selects a different test case which has the same req based on the request class it belongs. However, random mutation does not utilize the effects of fault injections in the past, and may not maximize the number of fault-handling bugs exposed in a fixed amount of time. Next, we describe the optimized mutators to generate impactful offspring $\phi\prime$ based on fault injection history.

1) Mutate Fault Type and Injection Point: When mutating a fault type, in order to generate offspring ϕ' that have high impact value, for each microservice, we also define a metric vector vul, which represents the capability that a service srv can handle each kind of fault type ft, and is defined as follows:

$$srv.vul[ft] = \frac{error\_scenario(Q_{history}, srv, ft)}{sum\_scenario(Q_{history}, srv, ft)}$$

Here, $Q_{history}$ saves a list of injected fault tests, $sum\_scenario$ function summarizes the number of executed tests where fault type is *ft*, and the calling service *src* is *srv* in the injection point, and $error\_scenario$ function summarizes the number of "*valid*" injected faults where error message is observed in the upstreaming service of srv.

The hypothesis behind this metric definition is that each microservice is developed by individual team, and if one API cannot handle the injected fault, it can be inferred that other APIs of this service may also fail to handle this fault with high probability. Therefore, when mutating a fault type, according to dst service in ip, new designed fault type mutator will pick a new fault type, sampled with a probability proportional to the corresponding metric value maintained in $dst.vul$.

Based on vul metric, we further define a metric $frail$ for each service, which represents the overall fragility of a service srv for the injected faults, and is computed by the summarization of each value in $srv.vul$. The improved injection point mutator selects a new one from the injection point set of req, sampled with a probability proportional to the fragility of src service. The injection point whose source service have high $frail$ value will have a high probability of being selected.

2) Mutate Request and Test Case: To guide request mutation, for each req, we define a fc metric, which represents the ratio of tested fault scenarios to the total fault scenarios based on current observed service call chain of req. IntelliFT picks the request with a probability inversely proportional to fc value of req.

To guide test case mutation, for each test case, we define a metric $unique\_error$, which represents the discovered unique failures based on previously injected faults. IntelliFT selects a new test case with a probability proportional to $unique\_error$ value of tc.

Note that, same failures can be caught repeatedly during fault space exploration. In order to remove duplicate test failures, for the observed failure of a fault test, IntelliFT also captures the user action sequence (seq) before the failure ($error$), and then checks whether $\langle seq, error\rangle$ already appears in the previous fault tests. A new observed failure will be saved and added into $crash$ list (line 10-12).

Algorithm 1: fitness-guided Fault Space Exploration

```
Input: $Q_{priority}$: priority queue of high fitness fault tests;
       $Q_{pending}$: queue of tests waiting for execution ;
       $Q_{history}$: set of all previously executed tests;
       $Gain$: vector of benefit values, one for each axis in the fault space
Output: $crash$: saves the observed unique failures
1  begin
2      foreach fault test $\phi_x \in Q_{priority}$ do
3          $testProbs[\phi_x] \leftarrow computeProbs(\phi_x.fitness)$;
4      $\phi \leftarrow sample(testProbs, Q_{priority})$;
5      $attributeProbs \leftarrow normalize(Gain)$;
6      $\alpha_i \leftarrow sample(\phi[\alpha_1, ..., \alpha_4], attributeProbs)$;
7      $\phi' \leftarrow mutate(\phi, \alpha_i)$;
8      if $\phi' \notin History \wedge \phi' \notin Q_{priority}$ then
9          $Q_{pending} \leftarrow Q_{pending} \cup \phi'$;
10     $\langle impact, error, seq\rangle \leftarrow execute(\phi')$;
11     if $\langle seq, error\rangle$ is unique then
12         $crash.add(error, \phi')$;
```

D. Optimize Fault Injection

IntelliFT computes initial injection points leveraging LDFI technique proposed in the work [7]. However, the computed injection point set still contains some redundancy which need not to be explored based on the result of previous fault injections. In the following, we introduce two heuristic rules to optimize fault scenarios generation further.

(Rule 1). When new fault handling logic appears in the call chain, new computed injection point may contain previously computed one stored in req's ip. In this case, such new injection point will not be added into the injection point set of req. The principle behind this rule is that if simple fault scenario cannot be handled successfully by the application, more complex failure scenarios do not need to be tested as they cannot be handled by the application definitely. If simple fault scenario can be successfully handled, new service call chain will be combined with previous ones to generate more complex injection points.

For example, consider the following service call chain in Fig.3(a), without fault injection, the injection point set contains: $\{API\}$, $\{Query\}$ and $\{DB\}$. After injecting a fault in the call to Query service, new call chain will be shown as Fig.3(b), in which $Query$ service will invoke $Cache$ service to return a previous result to the client. Based on LDFI technique, the new injection points are $\{API\}$, $\{DB\}$, $\{Cache, Query\}$ and $\{Cache, DB\}$. As $\{DB\}$ is a sub-set of $\{Cache, DB\}$, the later will be removed from the injection point set. In fact, in this example, even $\{Cache, DB\}$ is selected (as is done by the work [6]), the injected fault for $Cache$ service will not work as there is no call to $Cache$ service if $\{Query\}$ is not injected into fault.

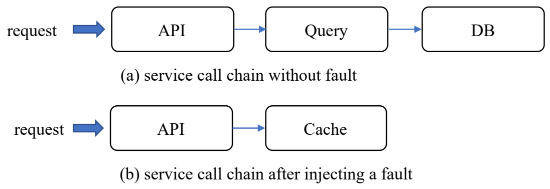
request
API
Query
DB
(a) service call chain without fault
request
API
Cache
(b) service call chain after injecting a fault

Fig. 3. An example of applying rule 1

(Rule 2). If one injected fault cannot be handled by its upstreaming services, that is, the same error message is observed to propagate backwards along the service call chain, it can be inferred that all the upstreaming calls (where error message is observed to propagate) cannot handle this injected fault. IntelliFT can safely avoid to generate the same fault scenario for these upstreaming calls.

For instance, consider service call chain in Figure 1(a), after returning 503 error for the call to $Ratings$ service, if the same error message is observed in the response of $Review$ and API calls, it means both $Review$ and API services cannot handle such fault, and IntelliFT will avoid to inject the same fault for these two injection points.

IV. Implementation

We implemented the main functionality of IntelliFT using Java languages (*https://github.com/ccx1024cc/IntelliFT*). It uses Jaeger [5] to collect service call chains, and Z3 solver [12] to compute the set of injection points. It leverages the fault injection capability that Isitio [1] provides to simulate typical faults in microservice based applications. Istio currently supports two basic fault types, where *Abort* returns a response with specified error code for the message *msg* from *src* to *dst* specified in *ip*, and *Delay* delays forwarding of message *msg* from *src* to *dst* with specified time interval.

Based on *Abort* and *Delay* fault, we implemented *Overload* fault as the combination of 80% of *Abort* with error code 503 and 20% *Delay* with interval 5s for all the calls to *dst* service in the *ip* of fault test ϕ, *Hang* fault as the 100% *Delay* with interval 5s for all the calls to *dst* service in *ip*, *Disconnect* fault as 100% *Abort* with 404 error code for all the calls to *dst* service in *ip*, and *Crash* fault as 100% *Abort* with null error code for all the calls to *dst* service in *ip*.

To support vision enhanced test case execution, we leverage open-source implementation of Vista [19], which intercepts Selenium WebDriver method calls (e.g., click()) to create visual locator for each web element in the test operations when running fault-free test cases, and uses the algorithms available in OpenCV [4] library, including SIFT [16], FAST [18] and fast NCC [11] to locate target elements visually.

V. Evaluation

To evaluate the effectiveness of our technique, we conducted a thorough empirical evaluation on an available benchmark of microservice based application. In our evaluation, we mainly investigated the following research questions:

- RQ1: Are the proposed fault injection optimization rules effective to reduce the generation of fault scenarios?
- RQ2: Is the optimized mutation technique effective to expose fault-handling bugs during the fault space exploration?
- RQ3: How does the IntelliFT's ability to expose fault-handling bugs compared to the state-of-the-art technique?

In the rest of this section, we present the subject application, the experimental protocol and the results.

A. Experiment Setup

TrainTicket application is a medium-sized open source microservice benchmark [25], which consists of 41 microservices. This application also provides 24 test cases, which exercise different functionalities that the application has (e.g., login, book ticket, cancel ticket). In our experiment, it was deployed on an in-house cluster consisting of 3 nodes, and each node is equipped with Intel(R) Core(TM) i7-8550U CPU@ 1.80GHz, 32GB DDR3 memory, running with Ubuntu 18.04 OS. The nodes in the cluster are connected via 10 Gigabit Ethernet.

One problem of selected benchmark is that most of microserices in the application have no fault handling logic. Therefore, the injected faults can lead to response failure and test case execution failure easily. To remedy this, we adopted two ways to add the recovery logic for some selected microserices. The first way is to modify the source code of the microservices directly to handle the injected faults using Hystrix [3], which is a resilient library which provides common fault handling patterns for microservices, such as timeout, retry and circuit break. The second way is to use version-based request routing that Istio provides to make *src* service in the injection point to invoke another service that has same function but with different version when an error response is received. Figure 4 shows such an example. Fig.4(a) shows the original service call chain for user login request, and Fig.4(b) shows the modified request handling logic, in which when *ts-login-service* fails, *ui-dashboard* will invoke *ts-login-service-back1*, which has the same functionality with *ts-login-service*. When *ts-login-service-back1* also fails, *ui-dashboard* will invoke *ts-login-service-back2* to handle user request. Both *ts-login-service-back1* and *ts-login-service-back2* can be seen as the backup paths for the original service call, which simulates the recovery logic when a fault is injected into the call to *ts-login-service*.

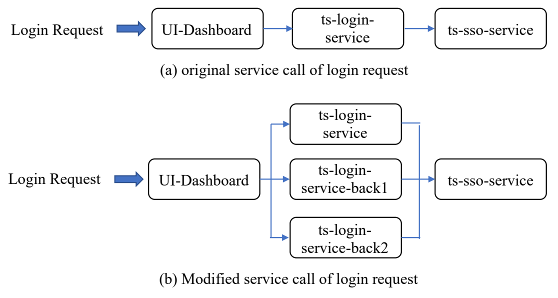

Fig. 4. Retrofit request processing logic

For RQ1, we selected 10 user requests from TrainTicket application randomly. Similar to work [7], in this experiment, we just adopted LDFI technique to do failure testing for each request individually without fault space exploration across different requests. The process of fault injections are done with two configurations: without and with optimization rules. Two basic fault types *Abort* and *Delay* were selected during the test. The experiment was ran 5 times and we compared the total average number of generated fault scenarios and exposed errors in the response with two configurations. We also apply Rule 1 and Rule 2 respectively, to compare the effectiveness of two rules in reducing the numbers of injected fault scenarios.

To investigate RQ2, IntelliFT leverages existing test cases accompanied with the TrainTicket application to start fault space exploration. We ran the proposed fault space exploration algorithm 30 minutes with two configurations: optimized mutators and random mutators. Every five minutes, discovered test failures will be checked. For each configuration, the exploration algorithm was run 5 times, and the average number of exposed failures are computed.

For RQ3, as there is no similar work to do guided fault space exploration for microservice based applications, we implemented two random fault exploration techniques. The first one works in black-box mode completely (called $randFT_1$). It has no knowledge about the internal service call chain, but has the APIs description of each microservice. The second technique (called $randFT_2$) has the knowledge about service dependency graph of TrainTicket application, which is built based on the collected service call chains when executing test cases. We compared IntelliFT with two random fault exploration techniques, and for each technique, the testing time is set as 1 hour. We ran the experiment 5 times, and computed the average number of exposed unique test failures and triggered backup paths.

B. Experimental Result

To answer *RQ1*, Figure 5 compares the simulated fault scenarios for the selected user requests with different configurations. It can be seen that, after applying reduction rules, less fault scenarios are generated obviously compared with the original LDFI technique. Furthermore, for the selected requests, rule 2 is more effective than rule 1 in reducing the injected fault scenarios. The reason is that most services in the call chain (for the selected requests) have no recovery logic, and the same error message is observed to propagate along the call chain. Therefore, rule 2 can reduce the generation of redundant fault scenarios quickly. It also should be noted that, although the proposed rules reduce the number of injected faults, it exposes the same number of response errors as the original LDFI technique. For space limitation, we did not show the detailed results here.

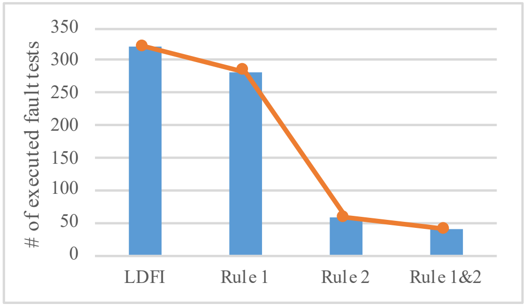

Fig. 5. Lineage driven fault injection with/without optimization

To answer *RQ2*, Figure 6 compares the number of exposed unique test failures of IntelliFT configured with optimized and random mutators. It can be seen that, the optimized mutation technique outperforms random one, and at the end of the testing time, IntelliFT with optimized mutators identified 31 unique failures, while only 20 failures with random mutators. Initially, the same number of failures are discovered for two configurations. However, with the increase of testing time, IntelliFT with optimized mutators discovered more test failures than random mutators as more feedback is accumulated during the exploration, which can guide optimized mutators to generate more high impact fault tests.

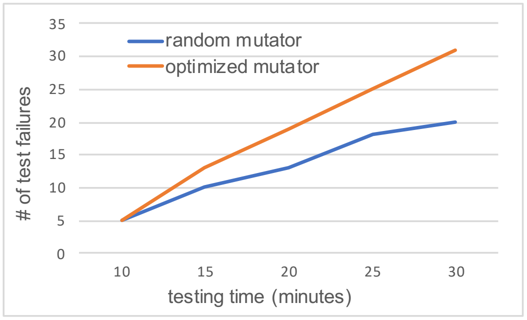

Fig. 6. IntelliFT configured with random/optimized mutators

To answer *RQ3*, Table 1 compares the number of explored fault tests, exposed (unique) test failures and backup paths of three techniques. It can be seen that IntelliFT is the most effective compared with $randFT_1$ and $randFT_2$. Although it explores less fault tests in an hour (as it needs to collect service call chains and compute injection points), 46 unique failures were exposed. $RandFT_2$ is more effective compared with $randFT_1$, as the later is black-box completely, and lots of generated fault tests are not valid. Although $RandFT_2$ generates less invalid fault injections (compared with $randFT_1$), it cannot avoid this completely as it lacks the relation between request and service call chain. Furthermore, both random techniques generate many duplicate fault scenarios in the experiment without exposing new failures, and lots of testing time is wasted. For the backup paths, IntelliFT exposed all 20 fault handling logic that we added in the application (especially for the cases shown in figure 4), while both $RandFT_1$ and $RandFT_2$ can only expose several superficial backup paths.

TABLE I
EVALUATION OF INTELLIFT VS. RANDOM EXPLORATION

	IntelliFT	$randFT_1$	$randFT_2$
# fault tests	534	873	752
# test failures	46	22	25
# backup paths	20	6	7

VI. Related Work

There are lots of work on testing of the resiliency of the applications. Zhang et al. presents ChaosMachine [21] to assess the exception handling capabilities of the applications in Java. It considers the error handling capabilities at the fine-grain level of programming language exceptions, and can reveal the resilience strengths and weaknesses for every try-catch block executed. AFEX [8] is a black-box approach which designs metric driven search algorithm to test the recovery code of the systems automatically.

Gunawi et al. [13] designs an exhaustive exploration approach to systematically injecting sequences of faults for cloud systems. It presents Failure IDs as a means of identifying injection points, and injects errors only at appropriate points to avoid duplicated injections. Heorhiadi et al. proposed Gremlin [14], a failure testing framework which supports to expose the fault tolerance bugs in microservice based applications. It allows developer or tester to write test scripts, which consists of the outage scenario to be created and assertions to be checked, and then executes them by manipulating the interaction messages between microservices. Similar to Gremlin, WS-FIT [15] is a tool for dependability testing of SOAP RPC-based web services, which injects faults by manipulating messages at the application layer between web services. LDFI technique [7] can generate the minimal injection point sets based on observed successful execution. Leveraging lineage graph [7] constructed by tracing the service call chain and existing fault injection infrastructure, Alvaro et al. [6] proposes an approach to automating failure testing of microservice-based applications at Netflix.

There are also some work which aims to debug microservice based applications [23]. For example, Peng et al. [24] apply delta debugging to minimize failure inducing of circumstances (e.g., environmental configurations). In the work [22], they further leverages ShiViz [10], a debugging visualization tool for distributed system to identify faulty microservice invocation.

VII. Conclusion

This paper proposes IntelliFT – a fitness guided, automated failure testing technique for microservice based applications, which can explore the fault space of AUT effectively to expose severe fault-handling bugs. In the future work, we will conduct more comprehensive studies to evaluate our approach by considering more different types of microservice benchmarks.

References

[1] Istio. retrieved sep. 2018 from https://istio.io/.

[2] Netflix - chaos monkey released into the wild. http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html.

[3] Netflix hystrix. retrieved apr. 2019 from https://github.com/netflix/hystrix.

[4] Opencv 2018. open source computer vision library. https://opencv.org.(2018).

[5] Uber jaeger. retrieved apr. 2019 from https://www.jaegertracing.io/.

[6] P. Alvaro, K. Andrus, C. Sanden, C. Rosenthal, A. Basiri, and L. Hochstein. Automating failure testing research at internet scale. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 17–28. ACM, 2016.

[7] P. Alvaro, J. Rosen, and J. M. Hellerstein. Lineage-driven fault injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 331–346. ACM, 2015.

[8] R. Banabic and G. Candea. Fast black-box testing of system recovery code. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 281–294, 2012.

[9] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016.

[10] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst. Debugging distributed systems. *Communications of the ACM*, 59(8):32–37, 2016.

[11] K. Briechle and U. D. Hanebeck. Template matching using fast normalized cross correlation. In *Optical Pattern Recognition XII*, volume 4387, pages 95–102. International Society for Optics and Photonics, 2001.

[12] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[13] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur. Fate and destini: A framework for cloud recovery testing. In *NSDI*, 2011.

[14] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar. Gremlin: Systematic resilience testing of microservices. In *ICDCS*, pages 57–66. IEEE, 2016.

[15] N. Looker, M. Munro, and J. Xu. Ws-fit: A tool for dependability analysis of web services. In *COMPASC, 2004.*, volume 2, pages 120–123. IEEE, 2004.

[16] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[17] S. Newman. *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.", 2015.

[18] E. Rosten, R. Porter, and T. Drummond. Faster and better: A machine learning approach to corner detection. *IEEE transactions on pattern analysis and machine intelligence*, 32(1):105–119, 2008.

[19] A. Stocco, R. Yandrapally, and A. Mesbah. Visual web test repair. In *FSE, pages=503–514, year=2018.*

[20] T. Xie, N. Tillmann, J. De Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 359–368. IEEE, 2009.

[21] L. Zhang, B. Morin, P. Haller, B. Baudry, and M. Monperrus. A chaos engineering system for live analysis and falsification of exception-handling in the jvm. *IEEE Transactions on Software Engineering*, 2019.

[22] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 2018.

[23] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *FSE*, 2019.

[24] X. Zhou, X. Peng, T. Xie, J. Sun, W. Li, C. Ji, and D. Ding. Delta debugging microservice systems. In *ASE*, pages 802–807. ACM, 2018.

[25] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao. Benchmarking microservice systems for software engineering research. In *IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 323–324, 2018.