This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2024.3363902

MicroFI: Non-Intrusive and Prioritized Request-Level Fault Injection for Microservice Applications

Hongyang Chen, Pengfei Chen*, Guangba Yu, Xiaoyun Li, Zilong He

Abstract—Microservice is a widely-adopted architecture for constructing cloud-native applications. To test application resiliency, chaos engineering is widely used to inject faults proactively in applications. However, the searching space formed by possible injection locations is huge due to the scale and complexity of the application. Although some methods are proposed to effectively explore injection space, they cannot prioritize high-impact injection solutions. Additionally, the blast radius of faults injected by existing methods is typically full of uncertainty, causing faults of multiple application functions. Although some tools are designed to conduct request-level injection, they require instrumentation on application code. To tackle these problems, this paper presents MicroFI, a non-intrusive fault injection framework, aiming to efficiently test different application functions with request-level injection. Request-level injection limits the blast radius to specified requests without any source code modification. Additionally, MicroFI leverages historical injection results and parallel technique to accelerate the searching. Moreover, An enhanced PageRank is used to measure the impact of faults and prioritize high-impact faults that fail more functions. Evaluations on three microservice applications show that MicroFI precisely injects faults and reduces up to 91% redundant faults on average. Additionally, by employing prioritization, MicroFI reduces an average of 47.3% injection budgets to cover all high-impact faults.

Index Terms—Microservice, Fault Injection, Chaos Engineering, Service Mesh, Tracing

1 Introduction

Microservice architecture has become a popular architecture to implement a large-scale, cloud-native application. With this architecture, the application is decomposed into dozens of small, loosely-coupled services that communicate through simple APIs [1], [2]. Each service comprises several to hundreds of instances. With the rapid evolution of user requirements, the quantity of application functions (e.g., Login, Search) as well as the kind of services increases, and the dependent relationships between services become complex. Such complexity hinders the development and orchestration of microservice applications. Therefore, service mesh [3], a communication infrastructure, is proposed to resolve this problem by providing proxies to manage network traffic between services. For example, in Fig.1, the proxies manage the traffic among service instances. As a result, the scale of applications increases, leading to a higher level of network complexity [4], [5]. However, the complex interactions, interference and dependencies among components render applications more fragile [6]–[9].

Postmortem reports published by Gremlin [10] that the absence or inadequacy of resiliency (i.e., fault-handling) mechanisms would probably lead to user-visible faults, and result in a huge economic loss and serious user experience violations. For example, a defective resiliency logic had led to failures of instance creation in Google Compute Engine in 2018 [11], impacting the services deployed on it. Therefore, it is a critical task to verify whether the resiliency mechanisms act as expected all the time.

Today, to assure applications are resilient to various faults, chaos engineering [12] is widely adopted in IT companies (e.g., Netflix [13], Google [14], Microsoft [15], Linkedin [16]). It is a discipline of randomly injecting faults into a production system to build confidence in the resiliency [17]. It helps IT companies to alleviate costly outages by preparing their teams for unknowns and protecting the customer experience [18].

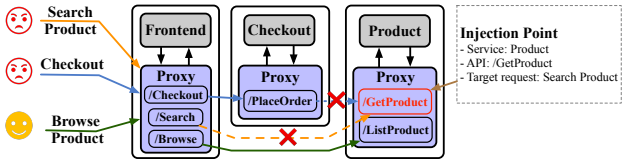


Fig. 1: Several call chains of three application functions. Service instances communicate with others through proxies which are equipped with forwarding rules for different API invocations. An API-level fault of the Product service affects requests issued to two other API functions, namely Search Product and Checkout.

Generally, chaos engineering is manually performed in production, and faults are typically injected according to domain knowledge [10], [13], [19]. For example, Netflix has exploited the knowledge from domain expertise within each team responsible for developing individual service to generate heuristics to guide the injection [13]. However, such a routine brings some problems. **i)** The chaos experiment may cause unintended impacts on production applications if the blast radius [12] is not controlled. The blast radius refers to the scope of the impact that fault injection may have. The injection of service-level faults, including faults at the service-instance-level and API-level, may affect requests that invoke different application functions which utilize the same malfunctioning service (service instance or API). For example, Table 1 shows the response time of various types of requests in an open-source application *Hipster Shop* [20].

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2024.3363902

TABLE 1: Average response time (ms) of different types of requests in different cases (without any faults (i.e. Normal) and with different service-instance-level faults injected)

URL	Method	Normal	CPU Exhaustion	Memory Exhaustion	I/O Exhaustion
http://hs-url/	Get	107	441	114	381
http://hs-url/cart	Get	102	556	143	2525
http://hs-url/cart	Post	194	759	280	3266
http://hs-url/checkout	Post	266	762	331	3171
http://hs-url/product/*	Get	95	671	167	2616

It can be seen that the blast radius of the injected service-instance-level fault is not controlled, leading to an increase in the response time of all types of requests that invoke the target instance. Such injected faults with uncontrolled blast radius typically cause unintended impacts to the application. For example, as shown in Fig.1, an injected API-level fault can abort the invocation of the API */GetProduct* for the propose of testing the SearchProduct function. However, if other functions also invoke the same API, the injected fault may also impact requests to those functions, such as the Checkout function. **ii)** The chaos experiment in production requires operators to concentrate on the applications in case some faults raise unnecessary customer pain. **iii)** The knowledge-based injection is inefficient and unscalable. Considering an application with 100 services, the potential solutions space of fault injection—combinations of fault locations—is $2^{100} - 1$. In such a huge space, experts need to invest significant time and effort to choose injection solutions and find a feasible fault.

Facing these problems, we argue that chaos engineering should be efficiently performed with a minimized blast radius, with a focus on restricting the impact to target requests. It is not a novel concept to conduct operations on the specified type of requests. For example, "Critical User Interaction attribution" has been introduced by Google [21] to provide aggregated observability for the specified type of requests using tracing technique. Similarly, it is important to consider the blast radius in chaos engineering to control unintended consequences and ensure the safety of the application under test. However, chaos engineering with a minimized blast radius requires a new injection tool since most of prior tools only support service-level [22], service-instance-level [23], [24] or API-level fault injection [25]. Faults injected by these tools affect all requests that pass through the target service, the target instance or the target API. Although some prior tools are designed to support request-level fault injection, they all require instrumentation on application source code. Additionally, efficient chaos engineering requires a new injection strategy since prior strategies explore injection space by exhaustive or knowledge-based searching [19], [22], [26]–[30]. These requirements pose challenges, including **i)** How to control the blast radius of the injected fault without any instrumentation to application codes. **ii)** How to utilize runtime information to test the application efficiently even if the quantity of application functions and the fault injection space are huge.

To address these challenges, we design MicroFI, a non-intrusive and fine-grained fault injection framework for calculating fault injection solutions in microservice applications. Each *injection solution* contains multiple *injection points*. Since MicroFI aims to break down invocations to service's API, the *injection point* consists of the target service and the target API. **i)** To address the first challenge, we design the non-intrusive request-level fault injection, which precisely restricts the blast radius to the specific type of requests without causing unexpected faults in applications. **ii)** To address the second challenge, we propose a new injection strategy by combining Lineage Driven Fault Injection (i.e., LDFI) [31] and PageRank [32]. With the extensions, our method supports testing multiple functions in parallel and prioritizing high-impact faults that simultaneously break down multiple functions. Experimental results on three representative open-source microservice applications show that MicroFI supports non-intrusive injection with a precisely controlled blast radius. MicroFI reduces execution time by up to 76% on average in calculating injection solutions, and brings an average reduction of 47.3% in the number of injection budget to expose all high-impact faults.

Contributions of this paper are: **i)** To control the blast radius of faults, we utilize service mesh and distributed tracing technique to implement the non-intrusive request-level fault injection. Only pre-configured requests are affected. This method can readily be applied to various applications without any knowledge of application source code. **ii)** We propose a lightweight and online injection strategy. It automatically calculates injection solutions without redundant failure exploration, and prioritizes high-impact faults within limited injections. **iii)** We design and implement MicroFI. Evaluations on three open-source microservice applications, namely *Hipster Shop* [20], *TrainTicket* [33] and *Hotel Reservation* [34] demonstrate the effectiveness of MicroFI.

This paper is organized as follows. Section 2 introduces the related work and Section 3 presents the preliminaries of main techniques in MicroFI and introduces the motivations. Section 4 and Section 5 outline and specify our approach. In Section 6, we evaluate our approach on three microservice applications. Section 7 discusses the limitations of MicroFI and the threats to validity. Section 8 concludes this paper.

2 Related Work

Table 2 presents a comprehensive comparison amongst existing methods on fault injection from different perspectives. Different columns represent different perspectives and their definitions are shown as follows. *Exploration* stands for injection space exploration methods. *Target System* presents the kind of systems where the tools target. *Fault Model* defines the kind of injected faults. *Pruning* indicates the optimization of the exploration that prunes the fault space. *Prioritization* indicates if injection solutions are prioritized. *Blast radius* presents the minimum impact scope of faults generated by these tools. *Instrumentation* indicates whether the tools need to modify source code. *Multiple Functions* indicates whether the tools support joint injection space exploration for all application functions.

Fault Injection Tools. Recently, several tools have been proposed to inject faults to the microservice application, general distributed systems or cloud infrastructure, shown in Table 2. Some of tools [27], [28] support injecting performance faults (CPU overload, memory exhaustion, network failure, disk failure, etc.) and crash failures (node crash, instance termination, etc.) into the infrastructure (nodes, containers, etc.) hosting the microservice application. They also manipulate network to inject performance faults (HTTP request delay, etc.) or crash failures (service unavailable,

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2024.3363902

TABLE 2: The comparisons between existing works from different perspectives

Exploration	Name	Target System	Fault Model	Pruning	Prioritization	Blast Radius	Instrumentation	Multiple Functions
Exhaustive search	ChaosMonkey [26]	Microservice, Infrastructure	Crash failures	No	No	Service Instance	No	No
	ChaosBlade [27], ChaosMesh [28]	Microservice, Infrastructure	Performance faults, Crash failures	No	No	Service Instance	No	No
Developer specified	Gremlin [10]	Microservice	Performance faults, Crash failures, Crash-recovery failures	No	No	Request	Yes	No
	Filibuster [22]	Microservice	Functional faults	Dynamic reduction	No	Service	Yes	No
	3MileBeach [35]	Microservice	Functional faults, Performance faults	No	No	Request	Yes	Yes
	Prefail [19]	Distributed Systems	Network failures, Disk failures, Crash failiures	Developer specified policies	No	System	No	No
	Setsudo [30]	Distributed Systems	Network failures, Disk failures, Crash failiures	No	No	System	No	No
	Fate and Destini [36]	Distributed Systems	Network failures, Disk failures, Crash failiures	Heuristics rules	No	System	Yes	No
LDFI	LDFI-Netflix [13]	Microservice	Functional faults, Performance faults	No	No	Request (FIT [37])	Yes	No
	IntelliFT [38]	Microservice	Functional faults, Performance faults	Heuristics rules	Heuristics rules	Service Instance	No	Yes
	MicroFI	**Microservice**	**Functional faults, Performance faults**	**Heuristics rules**	**Fault Impact**	**Request**	**No**	**Yes**

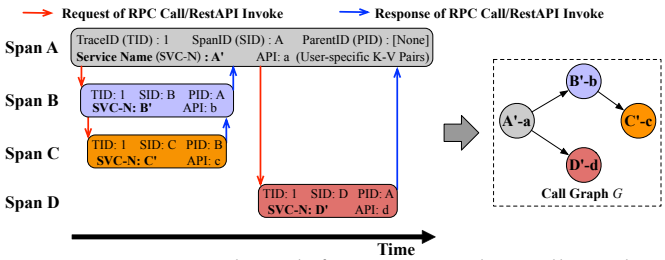


Fig. 2: A trace example with four spans and its call graph in one request

etc.) to the application. [19], [30], [36] supports injecting network failure, disk failure and crash failure to distributed systems by interposing failure logic into I/O related system calls. Other tools [10], [13], [22], [35], [38] manipulate network interactions between services to simulate the injection of performance and functional faults to microservice application. Among them, some tools [10], [35], [37] enable injection to affect specific requests, but their usages require instrumentation on source code. Since services are usually implemented in different programming languages, users could not afford to add additional code to the source code appropriately without any prior knowledge, hindering the widespread adoption of these tools. To address the issue, 3MileBeach [35] supports request-level fault injection by interposing at the message serialization library. While this approach avoids the need for application code instrumentation in some cases (e.g. Java, Python), applications implemented in certain languages (e.g., Go) may still require code modification to support request-level injection. Moreover, 3MileBeach requires modifications to the message serialization library for different languages, which can be costly and time-consuming. Since existing request-level injection tools conduct modifications to the application code, they require a restart of the application, leading to a period of service unavailability unavoidably. In production, such service unavailability is not acceptable as it would cause user experience violation. In contrast, MicroFI supports non-intrusive request-level injection which limits the blast radius to the target request without requiring any instrumentation and without restarting the application. The simplicity and effectiveness of non-intrusive request-level injection make MicroFI applicable to various applications.

Exploration methods. Existing exploration methods can be divided into three categories, namely exhaustive search, developer specified and LDFI. Exhaustive search based methods [26]–[28] are inefficient because they explore the fault space exhaustively and check whether all the combinations of faults can lead to failures. Specifically, ChaosBlade [27] and ChaosMesh [28] enable users to inject multiple faults in parallel and orchestrate them by manually defining chaos experiment workflow. It allows users to prioritize which fault should be injected more. However, since the minimum blast radius of ChaosBlade and ChaosMesh is service instance, the impact of simultaneously injected faults is mixed, hindering the precise control of fault injections. Moreover, the fault orchestration mechanism requires manual configuration. Developer specified based methods [10], [19], [22], [30], [36] are not flexible enough because they require expert knowledge to guide the search. LDFI based methods [13], [38] are efficient and scalable by searching the potential faults with runtime information. LDFI-Netflix [13] explores the injection space using the basic LDFI while IntelliFT [38] optimizes LDFI by pruning exploration space with heuristics rules and prioritizing faults with integration test technique. The objective of IntelliFT is similar to MicroFI, but the execution of IntelliFT would be slow and hard to converge if the quantity of application functions is large. Further, its prioritization requires knowledge encoded heuristics rules while MicroFI automatically prioritizes faults with runtime information.

3 Background and Motivation

3.1 Background

3.1.1 Preliminaries

Service Mesh [3]. As the transition from a monolithic application to a microservice application can result in increased maintenance cost, service mesh has been widely adopted to facilitate the development and orchestration of microservice applications [39]. Specifically, the popular cloud providers (e.g., Alibaba [40], Tencent [41], Google [42], etc) have provided their service mesh platforms to their customers. Service mesh serves as a networking infrastructure that

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2024.3363902

enables inter-service communication, offering a transparent and language-agnostic means for dynamically configuring networking and security functions. Typically, service mesh is implemented as two layers, a control plane responsible for managing the configuration and behavior of the mesh, and a data plane implemented as a network proxy (e.g. Envoy [43]) deployed as a sidecar alongside each service instance. All the inbound and outbound traffic for each service instance must first pass through its proxy, which handles tasks such as routing, load balancing, health checking, fault injection. As shown in Fig.1, each service instance is deployed with a proxy that handles all requests within the system, making it simple to conduct fault injection to impact inter-service invocations.

In this research, MicroFI utilizes the service mesh framework, Istio [25], to inject faults to affect network interactions between services. Actually, MicroFI is not tightly dependent on Istio, it can also be applied to other frameworks.

Tracing [44]. End-to-end distributed tracing is a valuable tool for profiling and understanding the execution of requests in microservice systems. A trace depicts the execution process of a request through services, which can be represented as a call path of the request. As shown in Fig.2, a trace consists of a set of spans (colored blocks in Fig.2) organized in a tree structure and each span represents a remote service invocation. A trace denotes as $T = (s_1, s_2, \cdots, s_n)$, where s_i denotes the span. Each trace has a unique trace ID while each span has a unique span ID and records the context of each service invocation. The trace **context** includes trace ID, span ID, parent ID and any user-specific key-value pairs such as the service name and invoked APIs. The **propagation** mechanism supports correlating events across services. The context is bundled and propagated across services, often via HTTP protocol. Specifically, there are two popular tracing standards, namely *OpenTelemetry* [45] and *OpenTracing* [46]. The trace **context** of *OpenTelemetry* contains Traceparent (Tp) and Tracestate (Ts). Traceparent identifies the incoming request and consists of version (v), trace-id (t), parent-id (p), and trace-flag (f). Tracestate provides additional vendor-specific trace information. The trace **context** of *OpenTracing* contains Baggage which is a key-value pair that could record user-specific information. With the tracing logic, Traceparent and Tracestate of *OpenTelemetry* as well as Baggage of *OpenTracing* are injected into requests as request headers and implicitly propagated across services.

In this research, MicroFI utilizes the collected trace data to construct the API call graph for each request, which is then used to calculate the injection solutions. The API call graph contains a set of distinct call paths and each path corresponds to an alternative computation that can serve the request. Moreover, MicroFI utilizes the propagation mechanism of tracing to propagate the request token for implementing request-level fault injection.

CNF formula [47]. A *Conjunctive Normal Form* (CNF) formula is a Boolean formula that is a disjunction of clauses. A clause is a conjunction of literals and a literal is either a propositional variable or the negation of a propositional variable. A CNF formula represents a logical proposition that can be either *True* or *False*, depending on the assignment of truth values to its variables. All clauses in a CNF formula must be satisfied for the overall CNF formula to be *True*.

SAT Solving [48]. The Boolean Satisfiability Problem (SAT) is a decision problem that asks whether a given logical formula, represented in Conjunctive Normal Form, can be satisfied by a truth assignment of its variables. Given a CNF formula $\mathcal{F} = f_1 \wedge \cdots \wedge f_n$, the SAT problem answers whether there exists a solution such that the formula equals to *True*. A SAT solver is a tool that can be used to determine the satisfiability of a given CNF formula. If a satisfying truth assignment exists, the solver can find one or more such assignments.

Specifically, the failure of any node within the call path invalidates the whole path. Therefore, in the context of this research, the call path is encoded into a clause and solutions that satisfy the clause can effectively invalidate the path. If all paths in a call graph are destroyed (i.e., all clauses in $\mathcal{F}$ becomes *True*), the request can not be served and a *failure* occurs. Therefore, identifying a *failure* scenario corresponds to finding a solution to the CNF formula $\mathcal{F}$.

Davis-Putnam-Logemann-Loveland algorithm [49]. DPLL is a well-known method for determining the satisfiability and calculating the solutions of the CNF formula. The algorithm is the pillar of most modern SAT solvers (e.g. Z3 Solver [50], MiniSAT [51]). Due to its ability to quickly deduce the values of certain variables, it can significantly reduce the search space and improve the overall performance of the solver. Specifically, given a CNF formula $\mathcal{F} = f_1 \wedge \cdots \wedge f_n$, DPLL aims to find a truth assignment of the variables in F such that the formula evaluates to *True*, or to conclude that no such assignment exists.

DPLL runs iteratively by simplifying the formula and making decisions about the truth values of its variables through the adoption of three rules, namely unit propagation, pure literal elimination, and splitting. Unit propagation states that if a clause contains a single literal, the literal value can be directly deduced. Pure literal elimination states that if a literal appears in the formula but its negation does not, the value of that literal can be deduced and clauses that contain that literal can be eliminated. If there are no literals that can be deduced using the above rules, the splitting rule would choose a literal and split the formula into two branches. The first branch assumes that the literal is *True* and the other branch assumes that it is *False*. DPLL will then solve each branch separately and combine the results finally.

3.1.2 Problem Statement

After introducing several preliminaries, the problem statement is formulated as follows.

Different application functions have different types of requests. Given $AF = \{af_1, af_2, \cdots, af_n\}$ and $R = \{r_1, r_2, \cdots, r_n\}$, we use af_i to denote an application function and r_i to denote the corresponding type of request. By analyzing the trace of the request $r \in R$, we can obtain a complete execution process and the corresponding API call graph $G = (X, E)$, which contains one call path. $x \in X$ denotes the invoked API extracted from the corresponding span and $e \in E$ denotes the call relation. Given the call graph $G = (X, E)$ for one type of request r, the nodes $x \in X$ can be represented as different literals x'. We represent the call path in the graph using a clause, which can be constructed with a conjunction of all literals x'. The clause is denoted as $f = x'_1 \vee x'_2 \vee \cdots \vee x'_n$. If there exist

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2024.3363902

additional call paths for r such as g', the corresponding clause f' can be combined with f using the logical "and" relation. The final CNF formula for the call graph of r is denoted as $\mathcal{F} = f_1 \wedge f_2 \wedge \cdots \wedge f_n$, where $f_1, \cdots, f_n$ represent the clauses for the call paths $g_1, \cdots, g_n$.

After constructing the CNF formula F for each type of request, the objective of the problem is to efficiently find injection solutions P for each application function $bf \in BF$ by solving $\mathcal{F}$ and prioritize the solutions. The set of all solutions for a given formula $\mathcal{F}$ is denoted as $P = [p_1, \cdots, p_k]$, where each solution $p_i = [x'_i, \cdots, x'_j]$ represents a truth assignment in which the variables in p_i are assigned to *True* and the remaining variables are assigned to *False*.

3.2 Motivations

Motivation #1: Unexpected impacts caused by faults with an uncontrolled blast radius hinder the adoption of chaos engineering in production. A recent study [18] has reported that only 30% of chaos experiments were performed in production environment while others were performed in development or staging environments. However, testing in the last two environments without production workload is insufficient to find defects in resiliency mechanisms. One of the inhibitors to perform chaos experiments in production is that some unexpected faults might happen. Moreover, it is inevitable for the application to experience *partial failure* [52], where only some of its functions are broken. It could be precisely simulated by failing the specific type of requests corresponding to the target application function. Prior tools either demand code instrumentation or cannot precisely simulate *partial failure* because of the incapability of limiting the blast radius to the specific requests. **Therefore, we are motivated to design non-intrusive request-level fault injection to limit the blast radius to the specified requests.**

Motivation #2: Testing different functions encounters redundant injection calculations, causing the testing to be time-consuming. Microservice applications implement various application functions using different logic and internal service invocations. However, some functions multiplex same services and APIs to implement different operations. During one-by-one exploration for testing multiple functions, the existing exploration methods never utilize historical injection results to alleviate the repetitive injections and calculations for the subsequent application functions testing. **Therefore, we are motivated to use historical results to prune the fault space in testing different functions.**

Motivation #3: Complex and versatile applications hinder the fault prioritization. In practice, software reliability engineers (i.e., SREs) usually set an error budget [53] for each application, which defines the number of times the application is allowed to fail. Therefore, given the limited resources and time, it is essential to prioritize high-impact faults. Moreover, new application functions are continually deployed or updated to meet the business and user demands. Thus, the scale and complexity of the applications are changed dynamically. The complexity and dynamics impede knowledge-based prioritization because experts need to constantly encode their knowledge into rules whenever a function is deployed or updated. **Thus, we are motivated to automatically prioritize injection solutions during fault space exploration based on runtime information.**

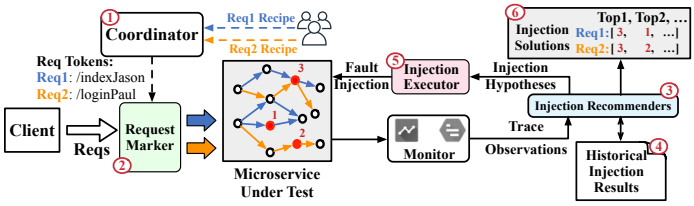


Fig. 3: The Overview of MicroFI

TABLE 3: Response time of requests to the ProductCatalog service in *Hipster Shop* under two conditions

Condition	Abnormal					Normal
Fault Type	*CPU*	*Memory*	*Read*	*Write*	*Crash*	
Response Time	0.44s	0.05s	2s	2s	Fail	0.03s

4 System Design

4.1 MicroFI Overview

Fig.3 provides an overview of MicroFI. For each target request type (e.g., Req1, Req2 in Fig.3), users need to write a recipe, which configures the fault type to inject and the type of request to affect. Initially, with the recipe, *Coordinator* (marked as ①) first translates it into the failure scenario—the combination of the type of injected fault and the target request to test. Then it sends attributes of the target request to *Request Marker* (marked as ②). Since there are two types of requests for testing, *Coordinator* bootstraps two corresponding *Injection Recommenders* (marked as ③) which automatically and parallelly calculate hypotheses for target requests. While requests are generated by the client, these requests will be intercepted by *Request Marker* before they invoke user functions, which utilizes the tracing technique to mark the target request. Subsequently, *Injection Recommender* retrieves tracing data of the target request to calculate and prioritize the injection hypotheses. These hypotheses are then utilized by *Injection Executor* (marked as ⑤) to construct service routing rules for request-level fault injection. After injection, *Injection Recommender* checks the observation data to determine whether the injected fault fails the target request. During parallel calculation, fault injection results are recorded into *Historical Injection Results* (marked as ④), and they are shared by all the *Injection Recommenders* to accelerate the subsequent calculation. Finally, failure solutions (marked as ⑥) for each target request are sorted and sent to users.

4.2 Fault Model

In a microservice application, services need to cooperate to process user requests. The cooperation is achieved through invocations between services, which is implemented with communication protocols like Restful API [54] and Remote Procedure Calls (i.e., RPCs) [55]. The response to an individual user request is a composition of responses from separate invocations to different services.

Table 3 presents the response time of requests sent to the ProductCatalog service in the *Hipster Shop* under two conditions. All services in the application are deployed with one instance. The response time is calculated as the duration between the moment a client send a request to the ProductCatalog service and the moment it receives a response from it. So, in this experiment, the application was deployed with Istio and its observability for application performance was utilized to measure the response time. During

the experiment, a bunch of requests belonging to the same type (i.e., http://hs-url/product/*) and invoking the same API of the ProductCatalog were uniformly generated. It ensures that the requests invoke the same business function, thereby eliminating any interference caused by variations in function logic that may impact the measurement. Under a normal condition, we measured the response time of the request and obtain normal response time. Under an abnormal condition, we injected 4 types of faults into the instance of the ProductCatalog shown in Table 3, and measured their corresponding response time during injection. In details, we utilized ChaosBlade [27] to inject CPU exhaustion, memory exhaustion, and crash faults respectively. The CPU exhaustion fault occupied 50% CPU resources within the service, the memory exhaustion fault occupied 50% memory resources and the crash fault suspended the service. Additionally, Strace [56] was used to inject 2-second delay to system calls (i.e., sys_read and sys_write).

It has been observed that injecting faults into the service often leads to an increase in response time and the invocation may be aborted in some cases. This behavior can be attributed to the fact that these faults consume computational resources within the service or prolong the execution time of essential system calls (e.g., sys_read, sys_write), hindering the service to efficiently handle incoming requests. Moreover, the injection of these faults always has varying impact on the requests. Thus, as these faults ultimately impact inter-service invocations, we simulate them and emulate the desired failure effects by manipulating the network interactions between services. Therefore, in this study, we inject two different faults (i.e., `Delay` and `Abort`) to fail inter-service invocations by leveraging service mesh.

4.3 Non-Intrusive Request-level Injection

As **Motivation #1** highlights that the injected faults with an uncontrolled blast radius always cause unexpected impacts to the microservice application. Consequently, MicroFI aims to mitigate this issue by controlling the impact of injected fault on the target request without causing any impact to other business functions. The fault injection mechanism is designed with *Request Marker* (Section 4.3.2) and *Injection Executor* (Section 4.3.3). *Request Marker* marks requests through trace creation and token propagation, presented in Fig.4(a) and 4(b). *Injection Executor* injects faults by failing target invocations between services, presented in Fig.4(c).

4.3.1 Challenge of Non-intrusive Request-level Injection

To ensure the injected fault only affects the specified function, it is crucial to mark the request invoking the function and its subsequent API invocations serving the function.

MicroFI marks an individual request and its subsequent API invocations by adding a token to the request and propagating it along with the API invocations. However, ensuring the token propagation across services during the subsequent invocations is non-trivial. A straightforward solution is to manually add the token propagation codes in the application codes. However, the method necessitates appropriate application codes modification, which can be inefficient. Alibaba® proposes an non-intrusive solution by leveraging tracing. Specifically, Alibaba establishes a mapping table that correlates the trace ID of the marked request with the corresponding added token [57]. Then, with service mesh, the proxy of each service instance verifies whether the trace ID of each incoming request exists in the table. If it does, the proxy adds the token to the outcoming request header. Although this method supports non-intrusive token propagation, the table query operation for each request introduces additional computation overhead.

4.3.2 Process of Non-intrusive Request Marking

To address this challenge, we leverage the propagation mechanism of trace context, which has been introduced in Section 3.1.1. In this study, we use *OpenTelemetry* to introduce our method. Fig.4(a) and Fig.4(b) show the details of how *Request Marker* marks the request and propagates the token through the tracing mechanism.

Trace creation. The first step is to mark the target request by creating the trace for the request and add the token into the trace context. As shown in Fig.4(a), before the target request flows into the application and invokes an application function, a *Request Marker* agent will start to trace the request and add a token, a key-value pair ("T=`/indexJason`"), to the Tracestate. Then, the agent forwards the marked request to the application. The added key-value pair is passed across services as a request header.

Token propagation. Since the trace context is propagated across services automatically, MicroFI utilizes the tracing technique to implement the propagation of the generated token across services. Unlike the method used by Alibaba®, we embed the token in the trace context and use the propagation mechanism to propagate the token without creating the table. As shown in Fig.4(b), we embed the token into the field "Tracestate" and leverage it to convey the token. When the service instance receives the incoming request, the trace context is implicitly propagated to the outcoming request through trace context propagation mechanism.

4.3.3 Process of Request Failing

Combining the added header, the fault injection solution computed from *Injection Recommender*, and the type of injected fault, *Injection Executor* constructs a service routing rule (e.g., the rule in Fig.8(c)). Then, *Injection Executor* sends the rule to the proxy of the target service instance.

The rule of each service instance's proxy instructs the proxy to inspect invocation messages between services and perform injection actions if a message matches the specified criteria. As Fig.4(c) shows, the rule of A describes that A should normally forward all the requests that pass to A. Therefore, the marked request is forwarded normally by A. However, the rule of B describes that B should abort the request that has a header, "T=`/indexJason`". Therefore, B aborts the forwarding of the marked request.

4.4 Injection Solutions Recommendation

As an application may consist of hundreds of microservices, the potential solution space of fault injection (i.e., combinations of fault locations) is huge, making it challenging to determine which APIs MicroFI should target first in testing the application. To address this issue, MicroFI introduces the fault injection recommendation algorithm, depicted in Algorithm 1, which helps to overcome this problem.

For a target type of request $\mathcal{R}$, *Injection Recommender* first collects trace data δ for $\mathcal{R}$ and $\mathcal{T}$ for all requests (line 6). With

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2024.3363902

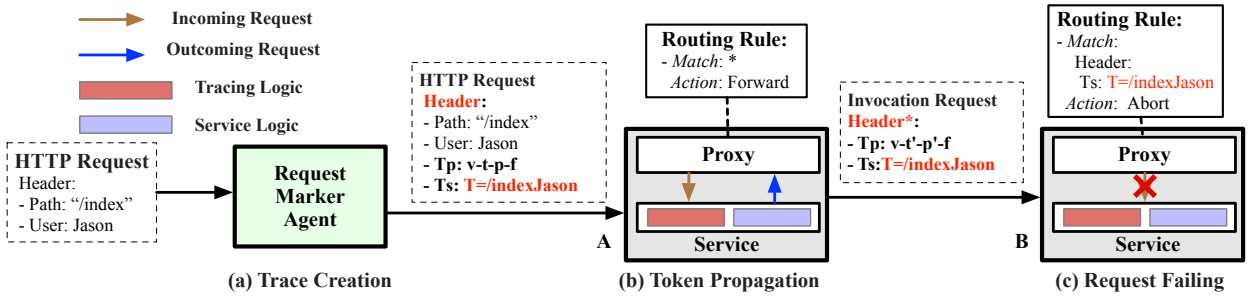


Fig. 4: The execution process of request marking (trace creation & token propagation) & request failing

Algorithm 1: Failure Solutions Recommendation

Input: The type of target request $\mathcal{R}$
Output: Final Injection Solutions Σ

1 $\tau \leftarrow$ Candidate Injection hypotheses;
2 $\mathcal{H} \leftarrow$ Historical Injection Results;
3 $\mathcal{S} \leftarrow$ scores of all APIs;
4 initialize $\mathcal{F}$;
5 **while** $|\tau| \neq 0$ **do**
6 Collect trace data δ and $\mathcal{T}$ for $\mathcal{R}$ and all requests;
7 Convert δ to clause f; $\mathcal{F} \leftarrow$ update as $\mathcal{F} = \mathcal{F} \cup f$;
8 $\Phi =$ Calculating minimal hypotheses($\mathcal{F}$);
9 $\Phi' =$ Redundant hypotheses elimination($\Phi, \mathcal{H}$);
10 $\tau = \tau \cup \Phi'$;
11 $\mathcal{S} \leftarrow$ update with trace data $\mathcal{T}$ using APIRank;
12 $\phi =$ Hypotheses Prioritization($\tau, \mathcal{S}$), $\tau = \tau - \phi$;
13 *fail* $\leftarrow$ Failure Effect Observe(execute(ϕ));
14 **if** *fail* **then**
15 $\Sigma = \Sigma \cup \phi$, $\mathcal{H} = \mathcal{H} \cup \phi$;
16 **end**
17 **end**

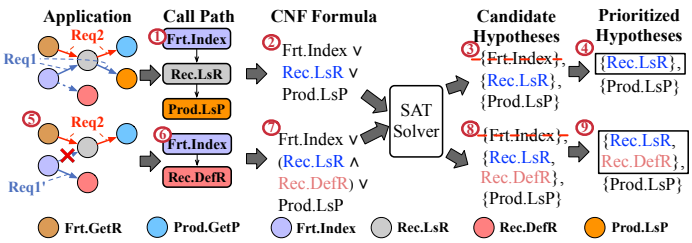


Fig. 5: An example of Algorithm 1

the trace data, the algorithm consists of four steps: **i)** By analyzing the call path recorded by δ, the clause f is obtained and the CNF $\mathcal{F}$ for $\mathcal{R}$ is updated (line 7). The details of trace collection are provided in Section 4.4.1. **ii)** By passing $\mathcal{F}$ into an SAT solver, a set of minimal candidate hypotheses can be generated (line 8), as described in Section 4.4.2. Specifically, the hypothesis is the minimal injection solution only when it has the ability to cause the target request to fail, while any subset of it is not able to fail the request. Then, any hypotheses that have already been recorded in Historical Injection Results $\mathcal{H}$ are removed (line 9-10). The process of eliminating redundant hypotheses is depicted in Section 4.4.3. **iii)** $\mathcal{T}$ is used to update the importance scores $\mathcal{S}$ of APIs using APIRank. With $\mathcal{S}$, candidate injection hypotheses τ are ranked and the highest-impact hypothesis ϕ is selected (line 11-12). The prioritization process is shown in Section 4.4.4. **iv)** A fault is injected based on ϕ, and the application is observed to decide whether the injected fault causes the request to fail, as introduced in Section 4.4.5. If the request fails, ϕ is written to $\mathcal{H}$ and recorded in Final Injection Solutions Σ (line 13-16). These processes (line 6-15) continues until all τ is empty. Specifically, it is necessary to update $\mathcal{H}$ if the implementation logic of a tested API (§) is updated. During updating, the solutions in $\mathcal{H}$ which are related to § should be removed from $\mathcal{H}$.

Before introducing the details, an example of Algorithm 1 is illustrated. Fig.5 shows two types of requests (Req1 and Req2) and depicts the process of calculating one of the solutions for failing Req1 (i.e., $\{Rec.LsR, Rec.DefR\}$). Note that, this process is followed by the process of obtaining the first solution $\{Frt.Index\}$, which is prioritized as the first hypothesis and evaluated as the first injection solution. ① shows the failure-free API-level service call path of Req1. It can be converted into a formula which only contains one clause: $Frt.Index \vee Rec.LsR \vee Prod.LsP$ (②). Then, the formula is fed into an SAT solver to generate a set of minimal hypotheses, each of which represents fault points that should be tested via fault injection. Based on the current formula, the minimal hypotheses (③) are: $\{Frt.Index\}$, $\{Rec.LsR\}$ and $\{Prod.LsP\}$. Note that, since we have validated that $Frt.Index$ is an injection solution, we eliminate the hypothesis $\{Frt.Index\}$ from the hypotheses set. Next, with the trace data generated by Req1 and Req2, the scores of APIs are calculated. The hypotheses are ranked (④) according to the scores. Then, the top-1 hypothesis ($\{Rec.LsR\}$) is selected and a fault is injected. After sending the same type of request (Req1'), the application still responds successfully and a new service call graph exists as ⑤. $Rec.DefR$ provides an alternative computation when $Rec.LsR$ fails. The new call path is transformed into the clause: $Frt.Index \vee Rec.DefR$, which is connected with ② to get ⑦. By solving the updated formula, the obtained latest minimal hypotheses are: $\{Frt.Index\}$, $\{Rec.LsR, Rec.DefR\}$ and $\{Prod.LsP\}$ shown in ⑧. The hypothesis $\{Frt.Index\}$ is also eliminated from the hypotheses set. With the updated scores, the hypotheses are ranked and the hypothesis $\{Rec.LsR, Rec.DefR\}$ is selected which finally causes Req1 to fail and regarded as a solution. The process in Fig.5 will continue to repeat until all hypotheses have been validated.

4.4.1 Call Graph Construction

Injection Recommender firstly collects trace data generated from all requests. Then, among the full trace data, trace data of $\mathcal{R}$ are filtered based on the propagated token added in the request header. With the aggregation of the filtered trace data, the corresponding call graph is extracted.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2024.3363902

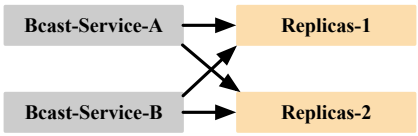


Fig. 6: Store operation on a storage system

4.4.2 The Minimal Candidate Hypotheses Calculation

As Section 3.1.1 presents, the request $\mathcal{R}$ can not be served and a *failure* occurs in the application if the corresponding extracted call graph is destroyed. However, it is non-trivial to find all minimal injection solutions that can destroy the call graph and fail the request due to application complexity. One straightforward method is to simply traverse each call path in the graph and inject faults at each edge to destroy the graph. However, it is not efficient to give all minimal solutions, especially in a complex application. For a call graph with N nodes, the injection space can be as large as:

$$\sum_{1 \leq k \leq N} \binom{N}{k} = 2^N - 1$$

For example, Fig.6 shows a storage system with replication, where data is stored on two replicas and the store is performed by two broadcasts. To fail the store operation, faults must be simultaneously injected into both replicas or into both broadcast services. It is important to note that the potential injection solution space of Fig.6 is $2^4 - 1$, which may seem feasible to explore. However, as the application scale expands, the exhaustive exploration becomes challenging. Moreover, the solutions that calculates by injecting fault at each edge are not minimal. To efficiently identify all minimal solutions, MicroFI employs the Boolean Satisfiability theorem. MicroFI first constructs the CNF formula for the generated call graphs. Then, the objective of identifying a *failure* scenario is converted into finding a solution to the generated CNF formula. MicroFI uses the SAT solver to calculate solutions to the CNF formula.

CNF formula construction. Given the call graph G that comprises a set of call paths $g_1, \cdots, g_n$ for serving one type of request, the corresponding CNF formula is constructed as $\mathcal{F} = f_1 \wedge \cdots \wedge f_n$, where $f_1, \cdots, f_n$ represent the clauses correlated with the call paths $g_1, \cdots, g_n$.

Hypotheses calculation. Solutions that satisfy the CNF formula and make the formula as *True* indicate potential failures in serving the request. Therefore, with the constructed CNF formula, MicroFI calculates the set of minimal hypotheses using the SAT solver, whose key is the DPLL algorithm that has been introduced in Section 3.1.1. After calculation, the redundant hypotheses are removed (Section 4.4.3) and the remaining hypotheses are ranked (Section 4.4.4). Faults are injected based on the prioritized hypotheses. If a new call path is discovered, indicating that the injected faults do not fail the request, the corresponding clause is combined with $\mathcal{F}$.

4.4.3 Elimination of Redundant Hypotheses

As **Motivation #2** presents, there are some redundant solutions in multiple application functions testing. Therefore, a heuristic rule is introduced to optimize the process.

When multiple functions invoke the same API of the same service, the result of testing the API in the previous function testing is used to prune the searching space of the subsequent function testing. In subsequent function testing, retesting the same API is unnecessary and the previous result can be used as the solution. The principle behind this rule is that the fault handling logic of the same API is the same even in distinct invocations. Therefore, after obtaining candidate hypotheses, *Injection Recommender* checks if the same hypotheses exist in $\mathcal{H}$. If so, the corresponding solutions of existed hypotheses are obtained through $\mathcal{H}$ directly and will be validated by injection. For example, in Fig.1, both the SearchProduct and Checkout functions invoke the same API (*/GetProduct*). If the testing for the Checkout function has calculated the final solution corresponding to the API (*/GetProduct*), then another testing for the SearchProduct function should skip the calculation for the API and directly validate whether the solution fails the SearchProduct.

4.4.4 Prioritization of Fault Injection Hypotheses

As **Motivation #3** presents, due to the high demand for prioritizing high-impact injection solutions with a limited error budget as well as the difficulty in manual prioritization in complex applications, *Injection Recommender* uses an enhanced PageRank (i.e., APIRank) to prioritize hypotheses with higher impact based on runtime information.

In a microservice application, the impact scope is large once all invocations to an API fail. Thus, the problem of computing the impact value for each API is converted into computing the importance for each API. The evaluation for the importance of each API is based on two insights. **i)** If the API is invoked by more services, it is more important. Once a fault is injected into it, all services depending on it would be affected. **ii)** If the API is covered by more request traces, it is more important. Once a fault is injected into it, all requests depending on it would fail. Inspired by MicroRank [58], which measures the anomalous scores for each service with PageRank to localize faults, we propose APIRank which assigns a importance score for each API by combining the static application topology and dynamic request paths.

Personalized PageRank (PPR). APIRank is based on PPR [59], which analyzes the graph of heterogeneous nodes (i.e., APIs and traces in this study). Given an oriented graph $\mathcal{G} = (V, \mathcal{E})$, where $|V| = N$ represents the number of nodes and $|\mathcal{E}| = M$ represents the number of edges, PPR firstly constructs a transition matrix $\mathcal{A} \in R^{N \times N}$. The matrix element $\mathcal{A}_{st}$ denotes the probability that a random walk from s to t and its value is computed using the left equation in Eq(1), where $O(s)$ represents the out-neighbor of s. With the constructed transition matrix and a given preference vector u, the PPR equation can be solved as $v = c \cdot \mathcal{A}v + (1-c) \cdot u$, where c is the damping factor ranging from 0 to 1 and v is the Personalized PageRank vector (PPV) for u. The solution to this equation can be approximated through an iterative algorithm. The q-th iteration is shown as Eq (2). The final solution corresponds to scores of all nodes.

$$\mathcal{A}_{st} = \begin{cases} \frac{1}{|O(s)|}, & \exists t \in O(s) \\ 0, & otherwise \end{cases}; \quad \mathcal{A} = \begin{bmatrix} \mathcal{A}_{aa} & \mathcal{A}_{at} \\ \mathcal{A}_{ta} & 0 \end{bmatrix} \tag{1}$$

$$v^{(q)} = c \cdot \mathcal{A}v^{(q-1)} + (1-c) \cdot u \tag{2}$$

Construction of the transition matrix in APIRank. In this research, APIRank uses the transition matrix $\mathcal{A}$ to address both insights mentioned above. The matrix is

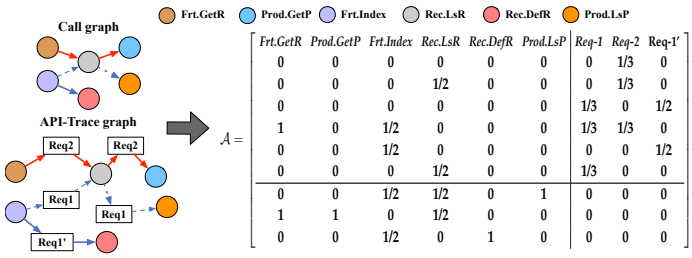
This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2024.3363902

partitioned as the right equation in Eq(1). To profile the invocation frequency of APIs called by other services (to meet insight **i)**), APIRank constructs $\mathcal{A}_{aa}$ based on the real time call graph, generated from the trace data $\mathcal{T}$. Since the call graph captures the invocation relationship among APIs from different service instance, $\mathcal{A}_{aa}$ profiles transitions between APIs. To profile the coverage of APIs invoked by different requests (insight **ii)**), APIRank constructs $\mathcal{A}_{at}$ and $\mathcal{A}_{ta}$ based on the API-trace graph, whose nodes represent APIs and traces respectively. Note that traces generating the same call path are identified as the same kind of trace. Since the API-trace graph describes the relationship between traces of requests and the corresponding invoked APIs, $\mathcal{A}_{at}$ and $\mathcal{A}_{ta}$ describe the transitions between APIs and traces.

Construction of preference vector. APIRank constructs the preference vector u as $u = [u_a^T, u_t^T]^T$, where u_a is for APIs and u_t is for traces. Since APIRank has no preference for APIs, u_a is set as $\vec{0}$. For the preference vector for traces, APIRank considers the number of traces that belong to the same kind. If a particular type of request is invoked frequently, the corresponding kind of traces will have a larger number of occurrences and deserves more attention. Thus, u_t is set as $[\theta_1, \theta_2, ..., \theta_m]^T$ and θ_i is set as $\frac{n_i^{-1}}{\sum n_j^{-1}}$, where m is the number of kinds of traces and n_i is the number of traces belonging to the kind of trace i.

Calculation of each API's score and the rank of hypotheses. After the calculation of $\mathcal{A}$ and u, the initial PPV $v^{(0)}$ is set as $[v_a^T, v_t^T]^T$. v_a is set as $[\frac{1}{N_a}, \frac{1}{N_a}, ..., \frac{1}{N_a}]$ where N_a denotes the number of APIs. v_t is set as $[\frac{1}{N_t}, \frac{1}{N_t}, ..., \frac{1}{N_t}]$ where N_t denotes the number of trace kinds. The estimated importance scores of all APIs are obtained by iteratively solving Eq(2) until the convergence is achieved. With the estimated scores, the score of each hypothesis is calculated. In the case of hypothesis with multiple injection points, the highest score among them is selected as the score for that hypothesis. With the scores of the candidate hypotheses, *Injection Recommender* ranks hypotheses and recommends the hypothesis with the highest score.

We utilize ⑤ in Fig.5 as an example to illustrate the prioritization. Candidate hypotheses are $\{Frt.Index\}$, $\{Rec.LsR, Rec.DefR\}$ and $\{Prod.LsP\}$. Fig.7 displays the call graph and the API-trace graph, which are the basis for constructing the transition matrix $\mathcal{A}$ in Fig.7. Since there are 3 distinct kinds of traces and each kind only has one trace, the preference vector for traces u_t is $[\frac{1}{3}, \frac{1}{3}, \frac{1}{3}]^T$. Combined with the preference vector for APIs u_a which is set to $\vec{0}$, the preference vector u is $[0, 0, 0, 0, 0, 0, \frac{1}{3}, \frac{1}{3}, \frac{1}{3}]$. Given that the number of APIs (i.e., N_a) is 6 and the number of kinds of traces (i.e., N_t) is 3, the initial PPV $v^{(0)}$ is $[\frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{3}, \frac{1}{3}, \frac{1}{3}]$. With a 20 iteration following Eq(2) where c is set to 0.85, the final scores of APIs are given as $S(Frt.GetR, Prod.GetP, Frt.Index, Rec.LsR, Rec.DefR, Prod.LsP) = [0.158, 0.161, 1, 0.517, 0.477, 0.354]$. Given that $\{Frt.Index\}$ has been previously identified as an injection solution, it will not be considered as further testing. Therefore, the prioritized hypothesis is $\{Rec.LsR, Rec.DefR\}$ which has the highest score except $\{Frt.Index\}$. Then, the selected hypothesis will be determined whether it fails the target request by injection. If so, the hypothesis is considered as a solution.



Frt.GetR	Prod.GetP	Frt.Index	Rec.LsR	Rec.DefR	Prod.LsP	Req-1	Req-2	Req-1'
0	0	0	0	0	0	0	1/3	0
0	0	0	1/2	0	0	0	1/3	0
0	0	0	0	0	0	1/3	0	1/2
1	0	1/2	0	0	0	1/3	1/3	0
0	0	1/2	0	0	0	0	0	1/2
0	0	0	1/2	0	0	1/3	0	0
0	0	1/2	1/2	0	1	0	0	0
1	1	0	1/2	0	0	0	0	0
0	0	1/2	0	1	0	0	0	0

Fig. 7: The transition matrix for ⑤ in Fig.5

4.4.5 Failure Effect Observation

To evaluate whether the injected fault invalidates target requests, three rules are designed based on the response. Firstly, we check the status code of the response. If the request is responded without a 200 status code, it fails. Secondly, we observe the payload of the response. If the payload contains keywords like error or timeout, the request fails. Thirdly, we measure the response time of the request. If the response time exceeds the threshold preset in the "timeout-retry" mechanism, the request fails.

4.5 Parallel Calculation

With request-level fault injection, the impact of each testing is isolated to a single type of requests, without affecting other requests. Therefore, if multiple types of requests need to be tested, a group of *Injection Recommender* instances can be launched simultaneously to calculate fault injection solutions for each type of request in parallel.

In details, when multiple types of requests need to be tested, *Coordinator* first initializes a group of *Injection Recommender* instances, which are responsible for calculating the injection solutions for each type of request. Then, *Coordinator* generates a global unique token for each type of target request, which will be used by *Request Marker* to differentiate requests and implement request-level fault injection. With global unique tokens generated for different types of requests, *Injection Executor* can construct the corresponding rules for failing each type of request accordingly. During the calculation of injection solutions for different types of request, the fault injection results validated by *Injection Recommender* are recorded into *Historical Injection results*. This historical records are then shared by all instances of *Injection Recommender* to support the elimination of redundant hypotheses, which is introduced in details in Section 4.4.3.

5 Implementation

We have implemented MicroFI based on Kubernetes [60] and Istio [25], two of the leading microservice infrastructures. MicroFI is implemented in Python language and chooses Z3 solver [50] as the SAT solver. MicroFI has been containerized to support directly deployed in a Kubernetes cluster. Moreover, MicroFI implements *Request Marker* in two versions to support injecting request-level fault in the application that utilizes either the *OpenTelemetry* or *OpenTracing* tracing standards.

Specifically, MicroFI implements *Coordinator* to translate a recipe into the failure scenario and activate a corresponding *Injection Recommender* for each scenario. The recipe that users need to configure contains two parts. The first part is the definition of the target request. The functions of an application are always triggered by different user requests.

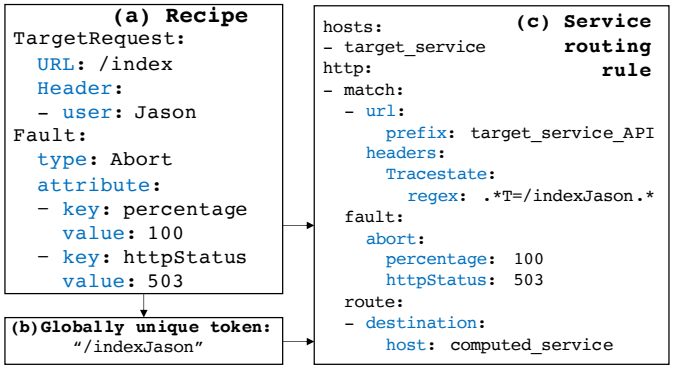


Fig. 8: An example of a recipe and the corresponding globally unique token and service routing rule

Requests are defined by different request URLs. Moreover, there are different users invoking requests to an application. The requests are distinguished by different user information in the header of requests. Therefore, the definition of the target request should be denoted with request attributes, such as its URL, the caller and query string parameters. With request attributes translated from the recipe, a globally unique token (T) for the target request is obtained by concatenating all request attributes together. For example, the globally unique token for the recipe in Fig.8(a) is `/indexJason`.

The second part of the recipe is the type of injected fault. As Section 4.2 presents, MicroFI supports two primitive faults (`Delay` and `Abort`) and they can be configured with different parameters. For both faults, users can specify the possibility of the target invocation being affected by the fault with *percentage*. For `Abort` fault, users can specify the error code that the target invocation returns with *httpStatus*. For the type of `Delay` fault, users can increase the request latency of the target invocation with *delayTime*.

Fig.8(a) shows an example of the recipe. The recipe instructs MicroFI to affect requests that Jason calls whose URL are "/index". During testing, MicroFI aborts all invocations to the `target_service_API` of `target_service` which are calculated by *Injection Recommender*.

6 Evaluation

Our evaluation seeks to address the questions. **RQ1:** Can MicroFI precisely control the blast radius of fault injection? (Section 6.2) **RQ2:** Is the optimization of MicroFI effective to accelerate injection solutions calculation? (Section 6.3) **RQ3:** Is MicroFI helpful to expose high-impact injection solutions through prioritization? (Section 6.4) **RQ4:** What is the overhead of MicroFI in controlling the blast radius of fault injection? (Section 6.5)

6.1 Evaluation Setup

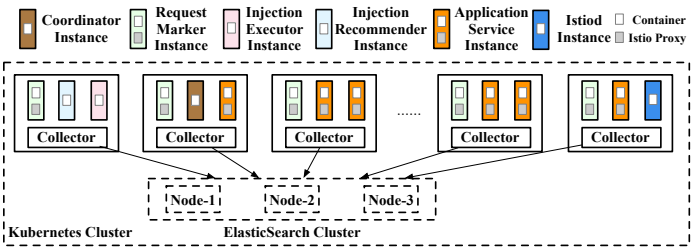


Fig. 9: The topology of experiment platform

TABLE 4: The description of three evaluation applications

Microservice Applications	Hipster Shop	Hotel Reservation	TrainTicket
# Services	11	8	41
# Business Functions	6	4	7
Communication Protocol	gRPC	gRPC	HTTP
Application Language	Java, Go, C#, NodeJS, Python	Go	Java
Tracing Standard	Open Telemetry	Open Tracing	Open Tracing

Our prototype is evaluated with 3 representative open-source microservice applications as presented in Table 4. Specifically, Hotel Reservation is a relatively small-scale benchmark designed to simulate an online system for booking hotels. Hipster Shop is a medium-scale application that serves as a web-based e-commerce that allows users to browse, order, and purchase products. TrainTicket is larger and more complex, providing various business functions for booking railway tickets. These applications reflect the characteristics of industrial microservice applications [33], [34], encompassing not only their scale and complexity but also in their programming languages, communication protocols (gRPC and HTTP), and tracing standards. Therefore, these application are representative and widely used as the experiment benchmarks [4], [38], [58], [61]–[64].

Since there are no timeout handling logic in these applications, we use Istio to set timeout for these services. To generate external requests and invoke corresponding business functions in applications, we utilize the workload generator Locust [65]. As shown in Fig.9, we have built a distributed testbed that comprises 12 virtual machines (VM) and a 3-node ElasticSearch [66] cluster. Each VM is equipped with a 4-core CPU, 16 GB memory, and runs with the Ubuntu 18.04 operating system. All VMs are located within the same local area network to elimintate network jitter. With these VMs, we have created a Kubernetes cluster (v1.20.1) with Istio (v1.12.0) installed. We utilize an open-source tracing system Jaeger [67] to collect tracing data. The microservice benchmark and Jaeger collectors are deployed within the cluster. Service instances first send trace data to the Jaeger collector on each node, which then forwards the aggregated trace data to the ElasticSearch cluster for persistent storage. Components of MicroFI are also deployed in the Kubernetes cluster. The number of *Request Marker* instances is scaled out according to the benchmark application workload, whereas other components of MicroFI are deployed as a single instance. The instance of *Injection Recommender* receives trace data from the ElasticSearch and computes injection solutions that will be sent to the instance of *Injection Executor* to construct routing rules. The routing rules are then distributed to the target service via Istio.

For **RQ2** and **RQ3**, since most services in these three applications have no fault handling logic, timeout error message would propagate backwards to upstream services along the service call chain. As a result, injected faults cause failures in application functions easily. Therefore, we use version-based request routing that Istio provides to handle faults, which is the same way as in [38]. In details, when the error message is propagated from the injected service to the upstream service, the upstream service will invoke another backup service that has the same function as the injected service. In this study, a small-scale application *hr-3* is constructed by adding 2 replicas (i.e., backup service)

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2024.3363902

for services in *Hotel Reservation*. A medium-scale application *hipster-2* is constructed by adding 1 replica for services in *Hipster Shop*. A larger scale application *hipster-3* is constructed by adding 2 replicas in *Hipster Shop*. Another larger scale application *tt-2* is constructed by adding 1 replica for services in *TrainTicket*.

6.2 Control of the Blast Radius

In this evaluation, we compare the blast radius of three levels of faults to show the capability of MicroFI to limit fault impact on specified requests. We respectively select the injection tool ChaosBlade (CB) to inject Service-Instance-level (SI-level) faults, utilize Istio (IST) to inject API-level faults [25] and use MicroFI (MFI) to inject request-level faults. Gremlin (GM) and 3MileBeach (3MB) are also used to inject request-level faults to compare with MicroFI.

During evaluation, we deploy 1 to 3 instances for each service in three benchmark applications and leverage the load balancer in Istio to distribute requests equally to each instance. In total, there are 9 groups of experiments in this evaluation. The experiment indexes in three different colors are shown in Table 5 and Table 6 presents experiments in three applications (①-③ in Hipster Shop, ①-③ in TrainTicket, ①-③ in Hotel Reservation). Each experiment includes three different levels of injection experiments. For each experiment, Locust uniformly generates requests to invoke different business functions evenly for 36 minutes, with a concurrency level set as 100. In each experiment, we conduct 7 injection experiments sequentially and each injected fault lasts for 3 minutes. Moreover, a 2-minutes-recovery period is set between two consecutive injection experiments, ensuring that the application can recover from the previous fault before injecting next fault.

Fault configurations are shown in Table 5, where each row corresponds to settings of three levels of injections experiment (i.e., SI-level, API-level, request-level) in the same group of experiments. ChaosBlade is used to intermittently drop 50% of the target service instance's network packets to affect requests passing through the target instance. Istio is used to abort 100% of target API invocations in target services. MicroFI and Gremlin are used to either abort or delay 100% of the target API invocation requests when the target business function is called, with a delay time set to $1.5s$. Since 3MileBeach injects faults by manipulating serialization libraries to modify status code of requests, it lacks the capability to delay requests. Thus, it is only used to inject *Abort* faults. For example, experiment ① includes 7 injection experiments on Hipster Shop. The SI-level fault is to drop 50% packet of Cart service instance and the API-level fault is to abort 100% requests that invoke */GetCart* API in Cart. The request-level `Abort` fault is to abort 100% requests that invoke the */GetCart* API when the function (http://hs-url/cart) is invoked while the request-level `Delay` fault is to delay these invocation requests for 1.5s.

Moreover, each application service may provide multiple callable APIs, with each API being invoked by different requests. For example, the Cart service in Hipster Shop offers multiple APIs (*/GetCart*, */AddCart*, etc.) , where the */GetCart* API is invoked by different business function requests (e.g., http://hs-url/, http://hs-url/cart/checkout, etc.). Due to the possibility of multiple business function requests calling the same service or the same API, the blast radius of injected faults varies depending on the level of fault injection.

To measure the blast radius of different injection level, we assess the number of business functions that are affected during injections. The impact on each function is characterized by measuring changes in its success rate, response time of requests, and number of failed requests. Note that, in each experiment, different faults are injected respectively and the blast radius of each fault is measured separately.

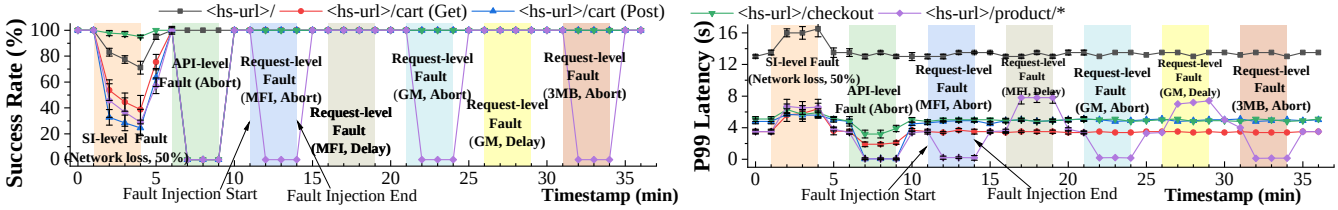
Fig.10 presents changes in success rate and P99 latency of different application functions requests during the experiment ②, where three levels of faults (7 faults) are injected into the ProductCatalog service. The blocks with different colors present the time periods during which different faults are injected. Specifically, lines with different colors in Fig.10(a) present changes in success rate of different functions requests during the experiment while lines with different colors in Fig.10(b) present changes of p99 latency of different function requests. **i)** As shown in Fig.10(a), requests of almost all functions are broken down by the SI-level fault, and requests of four different business functions are broken down completely by the API-level fault. This is because ProductCatalog service is usually invoked to serve all application functions and the target API (*/GetProduct*) is invoked to serve 4 functions. **ii)** In contrast, request-level faults injected by MFI, GM and 3MB only affect one target type of requests due to their capability of blast radius control. Specifically, the request-level `Abort` fault only fails all requests that invoke one business function (http://hs-url/product/*) in Fig.10(a) and the request-level `Delay` fault only delays requests that invoke the same function in Fig.10(b). Since the request-level `Delay` fault does not trigger the timeout logic, the target requests respond successfully with longer responses and the success rates of all requests remain unchanged. **iii)** Since these request-level injection tools only affect specific requests, the impacts of faults injected by them are similar. **iv)** Particularly, in Fig.10(a), during the injection of an SI-level fault, since the service has two instances, the success rates of the affected requests do not drop to 0. Further, as shown in Fig.10(b), because the API-level and request-level `Abort` faults abort all target requests, we can find that the response time of those failed requests decreases. Due to the space limitation, one application is selected to present changes of success rates and response time. Complete results for all applications are presented in Table 6.

Table 6 demonstrates the failure ratios of requests for each application function during fault injection experiments with experiment index ①-③, ①-③, ①-③. Since request-level `Delay` fault does not trigger the timeout logic in services, there are no failed requests. Thus, we present results of `Abort` fault. Table 6 reveals the same conclusions as in Fig.10. **i)** As *CB* columns show, SI-level faults cause failures across almost all different types of external requests (i.e, different business functions). However, due to the presence of another replica instance for the injected service instance, failure rates of the affected functions do not reach 100%. ii) As *IST* column shows, API-level faults fail multiple types of external requests completely because all invocations that pass through the target API are broken down. iii) As *MFI/ 3MB/ GM* column shows, request-level faults only fail API

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2024.3363902

TABLE 5: Details of Different fault injection experiments to answer **RQ1**.

Application	Experiment Index	Target Service	Service-instance-level (CB)	API-level (IST)		Request-level (MircoFI (MFI), 3MileBeach (3MB), Gremlin (GM))		
			Fault Type	*Fault Type*	*Target API*	*Fault Type*	*Target API*	*Target Request*
Hipster Shop	①	Cart	Network loss for 50% packets	Abort 100% requests	/GetCart	Abort 100% requests (MFI, 3MB, GM) / Delay 100% requests for 1.5s (MFI, GM)	/GetCart	http://hs-url/cart
	②	ProductCatalog			/GetProduct		/GetProduct	http://hs-url/product/*
	③	Recommendation			/ListRecommendation		/ListRecommendation	http://hs-url/cart/checkout
TrainTicket	①	Order			/order/tickets		/order/tickets	http://tt-url/searchTicketReserve
	②	Route			/route/*		/routes/*	http://tt-url/ticketPreserve
	③	Route-Plan			/cheapestRoute		/cheapestRoute	http://tt-url/advancedSearch
Hotel Reservation	①	Search			/Nearby		/Nearby	http://hr-url/hotels
	②	Profile			/GetProfiles		/GetProfiles	http://hr-url/recommendations
	③	Reservation			/MakeReservation		/MakeReservation	http://hr-url/reservations



(a) Success rate (b) P99 Latency

Fig. 10: The success rate and P99 latency of different application function requests in *Hipster Shop* during experiment ②

TABLE 6: Comparison of the failure ratios of application functions requests under different levels of fault injection

Application	External Requests		Failure Ratio of Application Functions Requests								
	URL	*Method*	*CB*	*IST*	*MFI/ 3MB/ GM*	*CB*	*IST*	*MFI/ 3MB/ GM*	*CB*	*IST*	*MFI/ 3MB/ GM*
			① Cart			② ProductCatalog			③ Recommendation		
Hipster Shop	http://hs-url/	GET	✓(6.8%)	✓(100%)	✗	✓(30.4%)	✗	✗	0	✗	✗
	http://hs-url/cart	GET	✓(4.1%)	✓(100%)	✓(100%)	✓(48.7%)	✓(100%)	✗	✓(47.8%)	✓(100%)	✗
	http://hs-url/cart	POST	✓(5.9%)	✗	✗	✓(39.8%)	✓(100%)	✗	✗	✗	✗
	http://hs-url/cart/checkout	POST	✓(2%)	✓(100%)	✗	✓(2.8%)	✓(100%)	✗	✓(50%)	✓(100%)	✓(100%)
	http://hs-url/product/*	GET	✓(9.5%)	✓(100%)	✗	✓(69.8%)	✓(100%)	✓(100%)	✓(45.8%)	✓(100%)	✗
Train Ticket			① Order			② Route			③ Route-Plan		
	http://tt-url/searchTicketReserve	GET	✓(40.8%)	✓(100%)	✓(100%)	✓(48.9%)	✓(100%)	✗	✗	✗	✗
	http://tt-url/ticketPreserve	POST	✓(35.7%)	✓(100%)	✗	✓(40.1%)	✓(100%)	✓(100%)	✗	✗	✗
	http://tt-url/advancedSearch	GET	✓(52.7%)	✓(100%)	✗	✓(44.9%)	✓(100%)	✗	✓(46.7%)	✓(100%)	✓(100%)
	http://tt-url/clientPayment	POST	✓(49.7%)	✗	✗	✗	✗	✗	✗	✗	✗
Hotel Reservation			① Search			② Profile			③ Reservation		
	http://hr-url/recommendations	GET	✗	✗	✗	✓(50.1%)	✓(100%)	✓(100%)	✗	✗	✗
	http://hr-url/hotels	GET	✓(51.1%)	✓(100%)	✓(100%)	✓(43.7%)	✓(100%)	✗	✓(45.7%)	✓(100%)	✗
	http://hr-url/reservations	POST	✓(50.2%)	✓(100%)	✗	✗	✗	✗	✓(50.8%)	✓(100%)	✓(100%)

• ① - ③, ① - ③, ① - ③ denote the Experiment Index in Table 5. ✓ presents that the injected fault fails function requests while ✗ presents that the fault does not.

invocations that are triggered by specified requests, leading the target type of requests to fail completely.

Summary. Unlike SI-level and API-level faults which cause a less controlled impact on microservice systems with a higher degree of randomness (i.e., potentially fail multiple business functions), MicroFI achieves a fine-grained and accurate request-level injection by precisely controlling the blast radius of a fault into the target request. The benefit of the fine-grained control is similar to that of other request-level injection tools, while MicroFI requires no instrumentation to application source code. This capability facilitates chaos engineering in production and the simulation of *partial failures*, removing concerns proposed in Motivation#1.

6.3 Effectiveness of Calculating Injection Solutions

In **RQ2**, MicroFI is compared with other methods to prove its efficiency in calculating injection solutions. Moreover, in this evaluation, MicroFI calculates injection solutions without prioritization. In addition to choosing the basic LDFI and the LDFI optimized by IntelliFT as baselines for comparison, we have also implemented a random strategy called RandomFI to prove the effectiveness of LDFI. RandomFI follows a similar process to LDFI but gives injection hypotheses randomly. In each iteration, it selects a random value v ranging from 1 to the number of explored APIs. Then it selects v APIs randomly to construct injection hypothesis and inject faults into them. If the target request fails, the hypothesis is considered as one injection solution. Otherwise, RandomFI updates the list of explored APIs and repeats injections. The process continues until all the minimal solutions have been obtained or the given number of injections is reached.

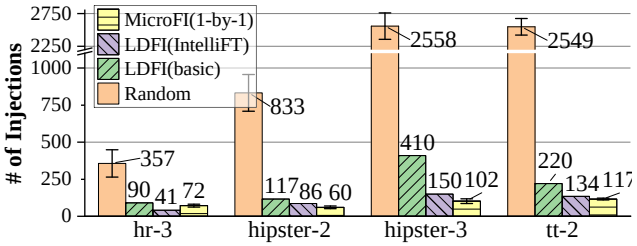
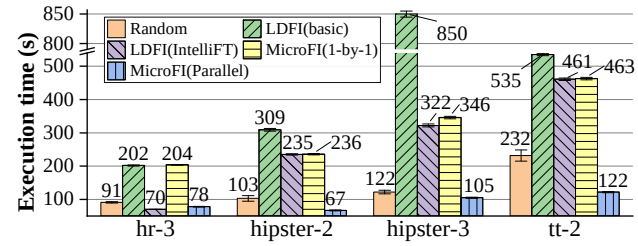


Fig. 11: Quantity of injections for calculating all the minimal solutions

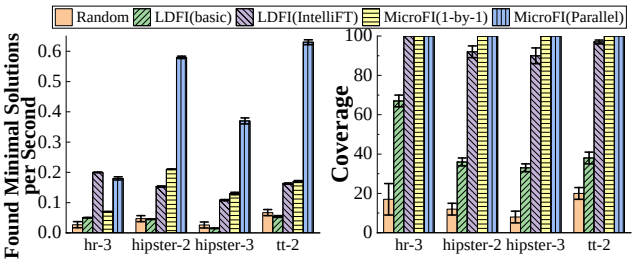
To prove the effectiveness of MicroFI in giving all the minimal solutions for multiple application functions, we compare the number of injections used by MicroFI with those of baselines when calculating all the minimal solutions for each function. Additionally, we conduct a comparison with baselines using the same quantity of injections which equals to the injection quantity required by MicroFI. The evaluation is based on three aspects, namely execution time, the number of found minimal solutions per second and solutions coverage. The solutions coverage is calculated as the ratio of solutions obtained by each method to the total number of solutions. The execution time comprises the time spent on calculation, trace collection and injection execution. The measurements are repeated for five times.

Fig.11 presents the number of injections used to calculate all injection solutions by different methods. Fig.12(a) shows the execution time used by different methods with the same injection quantity. **i)** Compared with LDFI (basic), the injec-

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2024.3363902



(a) Execution time



(b) Found minimal solutions/s and Coverage

Fig. 12: Comparisons of different methods given the same quantity of injections

tion quantity and the execution time are reduced through the pruning and the parallel technique. About 48% injections on average are reduced with the pruning technique and about 76% execution time on average are reduced with the parallel technique. This is because MicroFI skips the calculation and injections used to get the known solutions. By using the results from previous testings, MicroFI is able to directly validate whether solutions for a previously tested API are also solutions for the current testing, without the need for additional injection executions. Furthermore, since MicroFI provides the request-level injection, the testings for multiple functions are performed in parallel. Therefore, the time used in parallel testing only depends on the time used in the longest testing. **ii)** The execution efficiency improvement achieved by MicroFI compared to LDFI (basic) varies across different applications. In Fig.12(b), for applications with services deployed with one replica service (i.e., *hipster-2* and *tt-2*), the improvement in found solutions per second is 10.6x in *hipster-2* and 11.6x in *tt-2*. **iii)** The execution efficiency improvement achieved by MicroFI, relative to LDFI(basic), becomes more significant as the scale of tested applications increases. Testing functions one by one with MicroFI reduces 49% injections in *hipster-2* while it reduces 75% in a larger application (*hipster-3*). Parallel testing by MicroFI reduces 78% execution time in *hipster-2* while it reduces 87% in *hipster-3*. This improvement is closely related to the complexity of the application. When dealing with complex application, LDFI (basic) requires more injections to validate the calculated hypotheses. This increased complexity leads to longer time spent on SAT solving due to the construction of tedious CNF formulas corresponding to the application functions. In contrast, MicorFI accelerates testing for multiple functions by sharing historical injection results, which further simplifies the constructed CNF formula. The simplification reduces injection quantity and execution time needed to obtain all solutions.

Fig.11 and Fig.12 also prove the effectiveness of SAT solving in calculating the minimal solutions. **i)** In Fig.11, it is proved that RandomFI requires more injections to get

TABLE 7: Number of failed functions with 3 prioritized methods in 3 applications

	Method	T1	T2	T3	T4	T5	T6	T7	T8
Hotel Reservation (hr-3)	*Static*	1	2	2	3	4	4	4	4
	Dynamic	2	2	2	4	4	4	4	4
	APIRank	2	4	4	4	4	4	4	4
Hipster Shop (hipster-3)	*Static*	1	3	4	4	4	5	5	6
	Dynamic	4	5	5	5	5	5	5	6
	APIRank	4	5	5	5	5	6	6	6
TrainTicket (tt-2)	*Static*	3	3	7	7	7	7	7	7
	Dynamic	4	4	4	4	4	4	4	4
	APIRank	4	4	4	7	7	7	7	7

• T1-T9 denote Top1-Top9 prioritized hypotheses.

all solutions due to its adoption of a random strategy to guide injection. It introduces plenty of redundant injections. **ii)** Given same injections as MicroFI, RandomFI needs less execution time without the need of SAT solving as shown in Fig.12(a). However, it exhibits significantly lower solution coverage and finds fewer solutions per second in Fig.12(b). These findings indicate that while RandomFI may be faster in terms of execution time, it lacks the effectiveness and efficiency in identifying solutions.

Fig.12 also demonstrates that IntelliFT significantly reduces the injection quantity and the execution time. Since most services in the call chain lack recovery logic, the same error message is propagated throughout the call chain. Therefore, the heuristic rules employed by IntelliFT effectively prune the injection space. However, since IntelliFT requires more injections than MicroFI to get all minimal solutions, the solution coverage of IntelliFT is 5.25% on average lower than MicroFI.

To mitigate the interference caused by the randomness in the evaluation (i.e., time measurement, random selection of RandomFI), we conduct a Mean Difference T-test with a significance level 5% to investigate potential significant differences in the measurement results of each method. The analysis indicates that in the case of relative small-scale application (*hr-3*), MicroFI exhibits similar performance to IntelliFT in terms of obtained solutions/s and coverage (with $p - value \geq 0.05$). However, for other measurements on different applications, MicroFI significantly outperforms other methods (with $p - value \leq 0.05$). Moreover, the measurement results in other evaluations have been subjected to a statistical test with a significance level set to 5%.

Summary. Compared with LDFI (basic) and RandomFI, MicroFI removes about 48% and 91% redundant injections on average, while decreasing execution time by 76% and 28% on average given the same injection quantity. Compared with IntelliFT, MicroFI decreases time by 51% on average and obtains a higher coverage improved by 5.25% on average. Moreover, MicroFI always obtains more solutions during execution. It proves that MicroFI addresses the time-consuming problem in testing multiple functions presented in Motivation#2.

6.4 Effectiveness of Prioritizing Injection Hypotheses

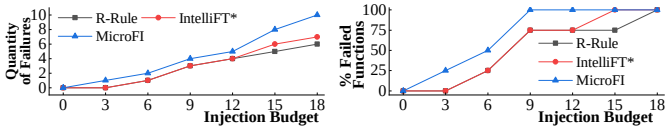
In **RQ3**, we demonstrate MicroFI's effectiveness in prioritizing hypotheses that can be high-impact solutions, considering two perspectives. **i)** To prove the effectiveness of prioritization through the integration of both static application topology and dynamic request traces, we perform a comparison between APIRank and two methods. These methods solely rely on either the static topology (*Static*) or

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2024.3363902

the dynamic traces (*Dynamic*). **ii)** We also compare MicroFI with other methods to evaluate their effectiveness regarding the prioritization of injection hypotheses.

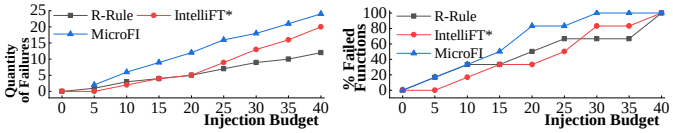
In the first evaluation, APIRank is compared with *Static* and *Dynamic*. *Static* ranks candidate injection hypotheses using the application topology. It relies on the topology to rank each service API based on the number of other APIs that depend on it. Then *Static* prioritizes hypothesis whose injection points are ranked on the top. *Dynamic* relies on dynamic traces. It first calculates the number of invocations for each API and then ranks each API based on the number of invocations. *Dynamic* prioritizes injection hypothesis whose injection points are invoked by most requests. For comparisons, we measure the number of business functions failed by Top1-Top8 hypotheses ranked by each method.

Table 7 shows the comparison results in different applications. These results show that APIRank requires fewer high-ranking hypotheses to fail all functions. For example, APIRank fails all functions in *hr-3* by injecting faults based on Top1-Top2 hypotheses while *Static* needs Top1-Top4 hypotheses and *Dynamic* needs Top1-Top5 hypotheses. This discrepancy arises from the limitation of relying solely on either static topology or dynamic traces, as they provide incomplete information about the running application. Solely relying on static topology for prioritization falls short in capturing the service invocation relationships in serving different application functions. Prioritizing with dynamic traces prefers hypotheses whose injection points are covered by a large number of requests. However, if the volume of requests is significantly skewed towards an application function, *Dynamic* might neglect other functions with fewer requests. Specifically, *Static* needs fewer high-ranking solutions to fail all functions in *tt-2*. This is because the performance of *Static* is related to the application topology. In *TrainTicket*, all application functions invoke the same two services that are depended by the largest number of other services. The preference for them contributes to the improvement in prioritization capabilities of *Static*. In contrast, APIRank prioritizes solutions based on runtime information, making it less sensitive to the application implementation and topology.



(a) Quantity of failures (b) Percentage of failed functions

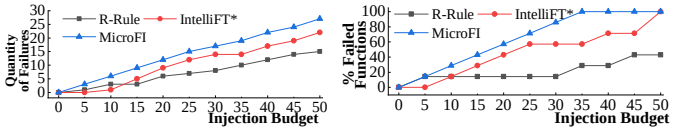
Fig. 13: Evaluation of prioritization methods in hr-3



(a) Quantity of failures (b) Percentage of failed functions

Fig. 14: Evaluation of prioritization methods in hipster-3

Then, we compare MicroFI with other methods in testing multiple functions within limited injections. Motivation #3 claims that SREs always set an error budget for each application to define the number of failures, making it important for prioritization method to prioritize high-impact faults within a limited injection budget. Thus, we evaluate the



(a) Quantity of failures (b) Percentage of failed functions

Fig. 15: Evaluation of prioritization methods in tt-2

performance of different methods in prioritizing injection solutions within limited injection budgets (quantity of injections). Given a sufficient number of injections, hypotheses prioritized by these methods can always fail all functions, hindering the valid evaluation for these methods. Consequently, the injection budgets in the evaluation have been set within a range from about 0% to 40% of the minimum injection quantity required for hr-3 (41), hipster-3 (102), tt-2 (117), shown in Fig. 11. We select IntelliFT and a rules-based method **R-rules** as baseline methods. IntelliFT is tailored to **IntelliFT*** in this study in two aspects to ensure fair and focused comparisons. **i)** The initial set of injection hypotheses for each request is directly provided by LDFI (the same as IntelliFT) at the initialization stage. The modification is made to ensure the injection budgets only include the number of injections used in prioritization. So, the focus of the evaluation lies in the performance of prioritization strategies. **ii)** The second aspect of the modification is about the mutation process in constructing each test. We have restricted the mutation of **IntelliFT*** to changing the request type and the injection hypothesis. Unlike IntelliFT, which also mutates fault types and test cases, **IntelliFT*** does not change the fault type. Only *Abort* fault is selected in this evaluation to directly fail the target invocation. Additionally, since each application function is only invoked by one type of request, the mutation of the request type is equivalent to changing the test case. Thus, the reduction of mutation attributes does not increase the number of injections required during prioritization.

Moreover, we also implement another method, **R-rules**, which prioritizes faults with two heuristic rules. Each heuristic rule corresponds to one intuition respectively. **i) R-rules** prefers to inject faults to APIs that have more dependencies. A failed API only affects services that invoke it and invocations cause dependencies between services. Therefore, the dependencies of service APIs can be used to approximate their failure impact [68], [69]. **ii) R-rules** prefers to inject faults to injection hypotheses that contains APIs that belong to the same service or locate on the same node. This rule is derived based on the likelihood of injection solutions [13]. As the resiliency logic remains similar among APIs within the same service [13], the probability of simultaneous failures in those APIs is higher. Additionally, considering that single-node crashes are more likely than the simultaneous failure of multiple nodes [70], APIs on the same node are more prone to fail. The likelihood of each solution is used to determine the order.

Fig. 13-15 show the evaluations on different prioritization methods. Quantity of failures denotes the count of triggered failures within the system, reflecting the times when business functions are observed as failed when faults are injected based on prioritized hypotheses. Furthermore, to capture the diversity of distinct failed functions, we use the percentage of failed functions as an additional metric.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2024.3363902

It indicates the percentage of business functions externally observed as failed, which is calculated as the proportion of failed functions in all functions. **i)** Given limited injection budgets, the hypotheses prioritized by MicroFI always cause more failures and fail more application functions than hypotheses prioritized by **IntelliFT*** and **R-rules**. **ii)** Although hypotheses prioritized by **IntelliFT*** cause fewer failures and fail fewer functions when the given injection budgets are few, its performance improves as injection budgets increase. The explanations for **i)** and **ii)** are shown as follows. **IntelliFT*** needs accumulation of test history, which could guide the constructions of subsequent testings. Specifically, **IntelliFT*** randomly constructs fault tests and executes them initially. Then it relies on the test history to guide the fault space exploration, including selecting which attributes (Request or Injection point) to mutate and how to mutate. If the test history is not enough, the exploration and the mutation will be performed randomly. As a result, **IntelliFT*** causes fewer failures and fails fewer functions at the beginning, hindering the achievement of good performance. In contrast, MicroFI relies on trace data rather than test history to prioritize high-impact injection hypotheses, rendering it to always cause failure at the beginning. Since **R-rules** does not consider historical injection results, it conducts many duplicate injections and thus performs worse.

Summary. Motivation#3 claims that the application complexity and dynamic hinder the manual prioritization. The evaluation proves the effectiveness of the prioritization strategy of MicroFI. Compared with other methods, MicroFI reduces an average of 47.3% injection budgets to prioritize hypotheses for failing all business functions. These prioritized hypotheses can be recommended as high-impact solutions.

6.5 Overhead of Request-level Fault Injection

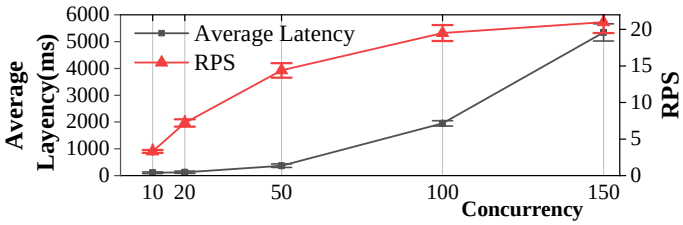


Fig. 16: Performance in *hipster-2* under different workloads

In **RQ4**, we evaluate the request-level injection overhead on *hipster-2*. We first use Locust to uniformly generate requests without any request-level injection capacity in different concurrency, and then compare the results on average latency and RPS (i.e., request per second) of requests as shown in Fig.16. The generation at each concurrency lasts for five minutes. As we can see, the curve of average latency becomes steeper and the curve of RPS becomes flatter when the concurrency reaches 100. Meanwhile, when the concurrency reaches 150, *hipster-2* serves abnormally and fails to handle some generated requests. Thus, the maximum concurrency is 100 and concurrency levels are set to 10, 20, 50, 100. Under these different concurrency levels, we uniformly generate requests to *hipster-2* for five minutes and measure the average latency in four cases. The first case is for *hipster-2* without any deployment of injection tools and other three cases are for *hipster-2* with the deployment of MicroFI, Gremlin and 3MileBeach. As injected faults affect the response time of requests, we measure the response time of requests which experience all necessary steps for injection but still respond normally. Each measurement lasts for five minutes and is repeated for five times.

Fig.17 shows average response time at different concurrency levels. It is evident that the response time increases in the range of 43%-75% with the use of request-level fault injection, regardless of the tool employed (e.g., MicroFI, Gremlin, or 3MileBeach). There are several reasons. Firstly, all these tools necessitate the use of *Request Marker* to mark the target requests. This introduces additional overhead as the target request must traverse the *Request Marker* before reaching the application. However, this overhead can be minimized in production by integrating the *Request Marker* logic into the existing gateway. Secondly, both MicroFI and Gremlin rely on the proxy provided by the service mesh to inject faults, and proxies of services need to sequentially check routing rules to determine if incoming requests should be affected by the fault. Therefore, additional latency is introduced, which can be mitigated through service mesh optimization techniques, such as Cilium [71]. As Gremlin does not require tracing to propagate the request token and 3MileBeach avoids interaction overheads caused by the service mesh, Gremlin (48%-52%) and 3MileBeach (43%-51%) increase lower latency compared to MicroFI (71%-75%). However, to enable the capability of request-level fault injection, they require the target application to be restarted, resulting in a period of service unavailability. In contrast, MicroFI does not introduce this overhead. Additionally, Gremlin and 3MileBeach require application-level instrumentation, whereas MicroFI does not.

To evaluate the resource overhead introduced by the request-level injection of MicroFI, we conduct experiments on *hipster-2* and the concurrency is set as 100. Then, we record the CPU and memory usage of each node every second during experiments. Linux commands *sar* and *free* are used to measure CPU and memory respectively. Once we obtain the CPU usage and memory usage of each node, we calculate the average usage per node. The resource usages are captured in four cases, namely MicroFI deployed with workload, MicroFI deployed without workload, MicroFI not deployed with workload, and MicroFI not deployed without workload. Each measurement lasts for 3 minutes and is repeated for five times. Fig.18 shows the results.

As we can see in Fig.18(a), the deployment of request-level injection of MicroFI always consumes more memory. When no workload is generated, the deployment of MicroFI consumes an additional 256MB of memory per node compared to the case where MicroFI is not deployed. When the workload is generated, the case where MicroFI is deployed consumes 282MB more memory per node than the case where MicroFi is not deployed. This increase in memory consumption is attributed to the deployment of service mesh and *Request Marker*, which requires more memory on nodes. On the contrary, the deployment of MicroFI does not increase the CPU usage. As shown in Fig.18(b), the average CPU usage per node in the case where nodes are deployed with MicroFI is almost equal to the case where nodes are deployed without MicroFI. The generated workload does not increase the consumption of CPU.

Summary. To support request-level injection, the re-

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2024.3363902

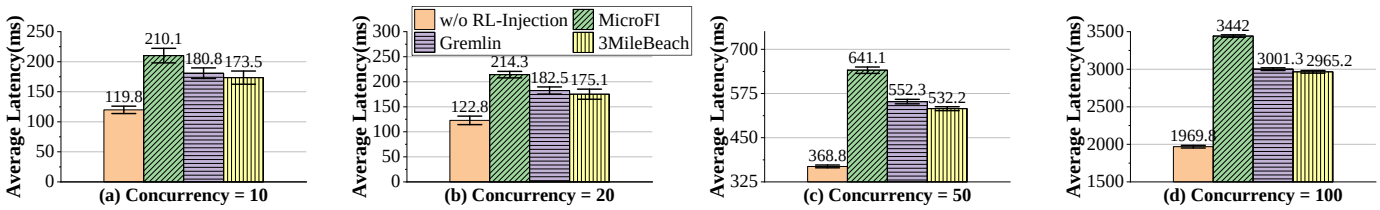


Fig. 17: Average response time of *hipster-2* in different concurrency under different cases (without request-level injection (i.e. RL-Injection), with RL-Injection using MicroFI, Gremlin and 3MileBeach)

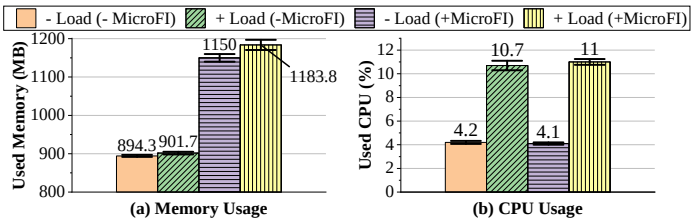


Fig. 18: Comparison results on CPU and memory usage of MicroFI with and without workload

sponse time of the request is prolonged by MicroFI in the range of 71%-75% due to the introduction of *Request Marker* and service mesh. Compared to other request-level injection tools, MicroFI increases latency by 18%-23% to support non-intrusive injection. Moreover, the deployment of MicroFI consumes approximately 250MB of memory without incurring any significant overhead in terms of CPU usage.

7 Discussions

Limitations. (i) One potential problem is that MicroFI relies on distributed tracing. Powered by the observablility infrastructures [72]–[74], tracing has been widely adopted in industrial microservice applications [75], rendering MicroFI to be easily utilized in these systems. However, MicroFI cannot work in applications without tracing. Firstly, the request-level fault injection relies on the trace propagation mechanism to mark specified requests and their subsequent invocations. Some instrumentation to application code is required to propagate the request token for applications without tracing. Secondly, the fault injection strategy relies on trace data to calculate injection solutions. Trace data is crucial as it records the execution process of each business function. Though application topology can be considered as an alternative to calculate solutions, it is unable to accurately calculate solutions and the obtained solutions would fail the entire application rather than a specific business function. (ii) It is beyond the scope of this study to discuss how to conduct SAT solving more efficiently. Since the time used in calculating solutions is occupied by SAT solving, the execution time would increase as the application grows larger. However, MicroFI is still more efficient than other works. (iii) MicroFI targets at injecting faults to directly fail or delay the requests to application functions. A full discussion of how to inject an appropriate resource exhaustion fault (e.g., which type of faults to inject and how much intensity of the injected resource exhaustion fault) to precisely fail or delay the requests are not discussed in this study.

Threats to Validity. The *internal threat to validity* mainly lies in the robustness to distributed tracing standards. MicroFI supports two popular tracing standards (i.e., OpenTracing and OpenTelemetry) among them. To alleviate this threat, MicroFI provides a flexible extension method to support other tracing standards as necessary. The *external threat to validity* mainly lies in the applications selected in this study. They may not represent the existing microservice applications. We try to cover the implementation differences of microservice applications, such as communication protocol, application scale and programming languages. However, MicroFI still requires a little customization to adopt the cases that we have not taken into account.

8 Conclusion

We propose MicroFI—a fine-grained fault injection framework for microservice applications, which tests the resiliency for multiple application business functions based on the request-level fault injection and prioritizes high-impact faults. We design the non-intrusive request-level fault injection that limits the blast radius into the target requests with the tracing and service mesh technique. Then we extend LDFI algorithm with injection space pruning technique and parallel technique to accelerate multiple business functions testing without redundant injections. An enhanced PageRank algorithm is also employed to support high-impact faults searching. The evaluations on three representative open-source microservice applications confirm the effectiveness of MicroFI. In the future, we plan to improve MicroFI in considering more information to construct fault tests.

References

[1] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *Service-Oriented Computing - 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings*, ser. Lecture Notes in Computer Science, C. Pahl, M. Vukovic, J. Yin, and Q. Yu, Eds., vol. 11236. Springer, 2018, pp. 3–20.

[2] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang, "Overload control for scaling wechat microservices," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: ACM, 2018, p. 149–161.

[3] O. Sheikh, S. Dikaleh *et al.*, "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*, ser. CASCON '18. USA: IBM Corp., 2018, p. 359–360.

[4] Y. Gan, Y. Zhang, , C. Delimitrou *et al.*, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, p. 19–33.

[5] X. Zhou, X. Peng *et al.*, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: ACM, 2019, p. 683–694.

[6] P. Huang, C. Guo *et al.*, "Gray failure: The achilles' heel of cloud-scale systems," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17. New York, NY, USA: ACM, 2017, p. 150–155.

[7] H. S. Gunawi, M. Hao *et al.*, "Why does the cloud stop computing? lessons from hundreds of service outages," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16. New York, NY, USA: ACM, 2016, p. 1–16.

[8] D. Cotroneo *et al.*, "How bad can a bug get? an empirical analysis of software failures in the openstack cloud computing platform," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: ACM, 2019, p. 200–211.

[9] X. Li, G. Yu, P. Chen, H. Chen, and Z. Chen, "Going through the life cycle of faults in clouds: Guidelines on fault handling," in *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, 2022, pp. 121–132.

[10] 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)*, 2016, pp. 57–66.

[11] "Google compute engine incident #18012," https://status.cloud.google.com/incident/compute/18012, 2022.

[12] H. Tucker, L. Hochstein, N. Jones, A. Basiri, and C. Rosenthal, "The business case for chaos engineering," *IEEE Cloud Computing*, vol. 5, no. 3, pp. 45–54, 2018.

[13] P. Alvaro, K. Andrus *et al.*, "Automating failure testing research at internet scale," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16. New York, NY, USA: ACM, 2016, p. 17–28.

[14] "Google dirt: Disaster recovery testing." https://www.oreilly.com/library/view/chaos-engineering/9781492043850/ch05.html.

[15] "Azure chaos studio." https://azure.microsoft.com/en-us/services/chaos-studio/\#overview, 2021.

[16] "Chaos engineering at linkedin: The "linkedout" failure injection testing framework." https://www.infoq.com/news/2018/06/linkedout-failure-injection/", 2021.

[17] A. Basiri, N. Behnam *et al.*, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[18] Gremlin, "state-of-chaos-engineering," https://www.gremlin.com/state-of-chaos-engineering/2021/?ref=blog, 2022.

[19] P. Joshi, H. S. Gunawi, and K. Sen, "Prefail: A programmable tool for multiple-failure injection," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, p. 171–188.

[20] "Hipstershop," https://github.com/GoogleCloudPlatform/microservices-demo, 2021.

[21] K. Lee, "Beyond distributed tracing," in *SRECon 2022*. San Francisco, CA: USENIX Association, Mar. 2022.

[22] C. S. Meiklejohn, A. Estrada, Y. Song, H. Miller, and R. Padhye, "Service-level fault injection testing," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 388–402.

[23] L. Zhang, B. Morin, B. Baudry, and M. Monperrus, "Maximizing error injection realism for chaos engineering with system calls," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2695–2708, 2022.

[24] J. Simonsson, L. Zhang *et al.*, "Observability and chaos engineering on system calls for containerized applications in docker," *Future Gener. Comput. Syst.*, vol. 122, pp. 117–129, 2021.

[25] "Istio," https://istio.io/latest/docs.

[26] "Chaosmonkey," https://github.com/Netflix/chaosmonkey.

[27] "Chaosblade," https://github.com/chaosblade-io/chaosblade.

[28] "Chaosmesh," https://chaos-mesh.org/".

[29] "Litmus," https://github.com/litmuschaos/litmus.

[30] P. Joshi, M. Ganai *et al.*, "Setsudo: Perturbation-based testing framework for scalable distributed systems," in *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, ser. TRIOS '13. New York, NY, USA: ACM, 2013.

[31] 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*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 331–346.

[32] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120.

[33] X. Zhou, X. Peng, T. Xie *et al.*, "Benchmarking microservice systems for software engineering research," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: ACM, 2018, p. 323–324.

[34] Y. Gan, Y. Zhang, C. Delimitrou *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud& edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, p. 3–18.

[35] J. Zhang, R. Ferydouni, A. Montana, D. Bittman, and P. Alvaro, "3milebeach: A tracer with teeth," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 458–472.

[36] H. S. Gunawi, T. Do *et al.*, "Fate and destini: A framework for cloud recovery testing," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. USA: USENIX Association, 2011, p. 238–252.

[37] A. Basiri, L. Hochstein, N. Jones, and H. Tucker, "Automating chaos experiments in production," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '19. IEEE Press, 2019, p. 31–40.

[38] Z. Long, G. Wu *et al.*, "Fitness-guided resilience testing of microservice-based applications," in *2020 IEEE International Conference on Web Services (ICWS)*, 2020, pp. 151–158.

[39] A. O. Duque *et al.*, "A qualitative evaluation of service mesh-based traffic management for mobile edge cloud," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing*, 2022, pp. 210–219.

[40] "Alibabacloud service mesh," https://www.alibabacloud.com/es/product/servicemesh, 2023.

[41] "Tencent cloud mesh," https://www.tencentcloud.com/products/tcm, 2023.

[42] "Anthos service mesh," https://cloud.google.com/anthos/service-mesh?hl=es, 2023.

[43] "Envoy," https://www.envoyproxy.io/, 2022.

[44] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc., Tech. Rep., 2010.

[45] "Opentelemetry," https://opentelemetry.io/, 2022.

[46] "Opentracing," https://opentraicng.io/, 2022.

[47] P. Jackson and D. Sheridan, "Clause form conversions for boolean circuits," in *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*, ser. SAT'04. Berlin, Heidelberg: Springer-Verlag, 2004, p. 183–198.

[48] T. Schoning, "A probabilistic algorithm for k-sat and constraint satisfaction problems," in *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, 1999, pp. 410–414.

[49] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, p. 394–397, jul 1962.

[50] "The z3 theorem prover," https://github.com/Z3Prover/z3.

[51] "The minisat page," http://minisat.se, 2022.

[52] C. Lou, P. Huang, and S. Smith, "Understanding, detecting and localizing partial failures in large system software," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, 2020, pp. 559–574.

[53] "Little known ways to better use your error budgets," https://www.blameless.com/blog/4-surprising-error-budget-use-cases.

[54] R. T. Fielding and R. N. Taylor, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000, aAI9980887.

[55] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, p. 39–59, feb 1984.

[56] "Strace," https://man7.org/linux/man-pages/man1/strace.1.html, 2023.

[57] "Realizing the new value of service mesh: accurately controlling the blast radius (chinese)," https://developer.aliyun.com/article/878287, 2023.

[58] G. Yu, P. Chen *et al.*, "Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments," in *Proceedings of the Web Conference 2021*, 2021, pp. 3087–3098.

[59] G. Jeh and J. Widom, *Scaling Personalized Web Search*. New York, NY, USA: Association for Computing Machinery, 2003, p. 271–279.

[60] "Kubernetes," https://kubernetes.io/, 2021.

[61] X. Zhou, X. Peng, T. Xie *et al.*, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2021.

[62] Z. Huang, P. Chen *et al.*, "Sieve: Attention-based sampling of end-to-end trace data in distributed microservice systems," in *2021 IEEE International Conference on Web Services*, 2021, pp. 436–446.

[63] Z. He, P. Chen *et al.*, "Graph based incident extraction and diagnosis in large-scale online systems," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: ACM, 2023.

[64] Y. Gan, C. Delimitrou *et al.*, "Sage: Practical and Scalable ML-Driven Performance Debugging in Microservices," in *Proceedings of the Twenty Sixth International Conference on Architectural Support for Programming Languages and Operating Systems)*, April 2021.

[65] "Locust: An open source load testing tool." https://locust.io/.

[66] elasticsearch. (2015) elasticsearch/elasticsearch. [Online]. Available: https://github.com/elasticsearch/elasticsearch

[67] "Jaeger," https://www.jaegertracing.io/, 2022.

[68] T. Yang, M. R. Lyu *et al.*, "Aid: Efficient prediction of aggregated intensity of dependency in large-scale cloud systems," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 653–665.

[69] J. Yin, X. Zhao *et al.*, "Cloudscout: A non-intrusive approach to service dependency discovery," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1271–1284, 2017.

[70] A. Raina and R. Ellupuru, "Madaari: Ordering for the monkeys." Brooklyn, NY: USENIX Association, Mar. 2019.

[71] "Cilium," https://cilium.io/, 2022.

[72] Datadog, "Cloud monitoring as a service," https://www.datadoghq.com/, 2023.

[73] Apache, "Skywalking," https://skywalking.apache.org/, 2023.

[74] Alibaba, "Application real-time monitoring service," https://www.alibabacloud.com/es/product/arms, 2023.

[75] C. Zhang, X. Peng, D. Zhang *et al.*, "Deeptralog: Trace-log combined microservice anomaly detection through graph-based deep learning," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 623–634.

Hongyang Chen received his BE degree from Sun Yat-sen University, China, in 2020 and is currently pursuing the Ph.D. degree in the School of Computer Science and Engineering, Sun Yat-sen University. His current research areas focus on distributed system, cloud computing, chaos engineering and software defined networking.

Pengfei Chen is currently an associated professor in the School of Computer Science and Engineering of Sun Yat-sen University. Meanwhile, he is a Ph.D. advisor. Dr. Chen graduated from the department of computer science of Xi'an Jiaotong University with a Ph.D. degree in 2016. Now, he is interested in distributed systems, AIOps, cloud computing, Microservice and network systems. Especially, he has strong skills in cloud computing. So far, Dr. Chen has published more than 80 papers in some international conferences including ACM ASE/ICSE/FSE, IEEE INFOCOM, WWW, ACM/IEEE CCGRID,IEEE ISSRE, IEEE ICWS, IEEE DSN, ICPP and journals including IEEE TDSC, IEEE TNNLS, IEEE TR, IEEE TSC, IEEE TETC, IEEE TCC. He serves as of program committee member of multiple conferences and reviewers of some internal journals such as IEEE TDSC, IEEE TC, and Information Science.

Guangba Yu received his master degree from Sun Yat-Sen University, China, in 2020. He is now a phd student at School of Computer Science and Engineering with Sun Yat-Sen University, China. His current research areas include distributed system, cloud computing, and AI driven operations.

Xiaoyun Li is currently pursuing the Ph.D. degree in the School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China. She received her B.Eng. from Sun Yat-sen University, in 2019. Her current research areas focus on ensuring cloud system reliability by applying AI techniques, especially in log analysis and mutli-modal fault diagnosis.

Zilong He received his BE degree and MS degree from Sun Yat-sen University, China, in 2019 and 2021. He is now a phd student at School of Computer Science and Engineering of Sun Yat-sen University, China. His current research areas include anomaly detection algorithms, AI driven operations.