Cast: Automated Resilience Testing for Production Cloud Service Systems

Zhuangbin Chen
School of Software Engineering
Sun Yat-sen University
Zhuhai, Guangdong, China
chenzhb36@mail.sysu.edu.cn

Zhiling Deng
School of Software Engineering
Sun Yat-sen University
Zhuhai, Guangdong, China
dengzhling3@mail2.sysu.edu.cn

Kaiming Zhang
School of Software Engineering
Sun Yat-sen University
Zhuhai, Guangdong, China
zhangkm7@mail2.sysu.edu.cn

Yang Liu
School of Software Engineering
Sun Yat-sen University
Zhuhai, Guangdong, China
liuy2355@mail2.sysu.edu.cn

Cheng Cui
Huawei Cloud
Shenzhen, Guangdong, China
cuicheng2@huawei.com

Jinfeng Zhong
Huawei Cloud
Shenzhen, Guangdong, China
zhongjinfeng@huawei.com

Zibin Zheng*
School of Software Engineering
Sun Yat-sen University
Zhuhai, Guangdong, China
zhzibin@mail.sysu.edu.cn

Abstract

The distributed nature of microservice architecture introduces significant resilience challenges. Traditional testing methods, limited by extensive manual effort and oversimplified test environments, fail to capture production system complexity. To address these limitations, we present Cast, an automated, end-to-end framework for microservice resilience testing in production. It achieves high test fidelity by replaying production traffic against a comprehensive library of application-level faults to exercise internal error-handling logic. To manage the combinatorial test space, Cast employs a complexity-driven strategy to systematically prune redundant tests and prioritize high-value tests targeting the most critical service execution paths. Cast automates the testing lifecycle through a three-phase pipeline (i.e., startup, fault injection, and recovery) and uses a multi-faceted oracle to automatically verify system resilience against nuanced criteria. Deployed in Huawei Cloud for over eight months, Cast has been adopted by many service teams to proactively address resilience vulnerabilities. Our analysis on four large-scale applications with millions of traces reveals 137 potential vulnerabilities, with 89 confirmed by developers. To further quantify its performance, Cast is evaluated on a benchmark set of 48 reproduced bugs, achieving a high coverage of 90%. The results show that Cast is a practical and effective solution for systematically improving the reliability of industrial microservice systems.

*Zibin Zheng is the corresponding author.

ACM ISBN 979-8-4007-2426-8/2026/04
https://doi.org/10.1145/3786583.3786867

CCS Concepts

• Software and its engineering → Software testing and debugging; **• Computer systems organization → Reliability**.

Keywords

Microservices, Software Reliability, Resilience Testing

ACM Reference Format:
Zhuangbin Chen, Zhiling Deng, Kaiming Zhang, Yang Liu, Cheng Cui, Jinfeng Zhong, and Zibin Zheng. 2026. Cast: Automated Resilience Testing for Production Cloud Service Systems. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-SEIP '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3786583.3786867

1 Introduction

The widespread adoption of microservice architecture has fundamentally transformed how modern cloud applications are designed and deployed. This architectural paradigm offers significant advantages in terms of agility, scalability, and maintainability by decomposing monolithic applications into loosely coupled, independently deployable services. However, the highly distributed nature of microservice systems introduces substantial challenges regarding *system resilience*, particularly in the face of complex and often non-deterministic failures in cloud environments [20, 51].

Resilience testing plays a critical role in ensuring the reliability of microservice applications [25, 38, 55]. Unlike traditional testing approaches that focus on functional correctness [7, 11, 30], resilience testing evaluates a software system's ability to tolerate, adapt to, and recover gracefully from failures once the underlying fault conditions are resolved. This is particularly crucial in production, where many failures are transient [18, 28, 29]. Issues like temporary network congestion or resource contention can cause service slowdowns or intermittent errors that may resolve on their own shortly. A resilient system must be able to withstand this temporary degradation without causing a complete outage. Moreover, the interconnected nature of microservice systems means that failures in individual components can propagate through the dependency chains, potentially affecting multiple services. In this case, when the faulty component is repaired, the affected services must be able to autonomously return to a healthy operational state.

Despite its importance, effective microservice resilience testing presents significant challenges. Traditional methods are limited in two key aspects. First, they often require extensive manual effort [27, 49, 54], rendering them impractical for large-scale systems. This includes manually designing test cases, configuring complex fault injection scenarios, and defining appropriate assertions beyond simple status checks. Such a manual process is slow, error-prone, and cannot scale with the complexity and rapid evolution of modern services. Second, their applicability to industrial systems is limited by the insufficient expressiveness of both fault models and verification mechanisms. Many studies [16, 23, 52] rely on small benchmarks (e.g., TrainTicket [22, 58], SockShop [50]) that lack the scale and intricate dependencies of production environments. The injected faults are often generic (e.g., basic HTTP errors) and validated with simple checks (e.g., status code 200), providing limited ability for testers to create customized checks. Such approaches fail to capture the rich spectrum of real-world failures like specific middleware exceptions or complex asynchronous issues [15, 37, 48].

To address these limitations, we present Cast, an end-to-end framework for scalable and effective microservice resilience testing in industrial environments. Our approach has three key design goals. First, for **production-level fidelity**, Cast combines online traffic record-and-replay with mocking technique for fine-grained, application-level fault injection. To ensure that recorded traffic can be successfully replayed [35], Cast dynamically identifies and updates state-dependent variables to reflect the current execution context. Moreover, Cast constructs a fault library of realistic failure modes (including software-level exceptions and communication errors) to rigorously exercise a service's internal error-handling logic. Second, for **scalability**, Cast tackles the combinatorial explosion of the test space through a complexity-driven strategy. It intelligently selects a small yet potent set of test cases that target the most intricate and failure-prone execution paths. This provides a systematic solution to proactively discover critical resilience vulnerabilities. Finally, for **automation**, Cast orchestrates the entire test execution lifecycle through a three-phase pipeline (i.e., startup, fault injection, and recovery), while automatically verifying the system's resilience against multi-faceted criteria.

We have deployed Cast in Huawei Cloud for over eight months, serving a large number of microservice applications for their automated resilience testing. These applications span various business domains including storage, networking, digital platform, and Internet of Things (IoT), etc. Through our long-term deployment, we have gained valuable insights into the practical challenges and trade-offs of resilience testing at scale. In this paper, we report results for four large-scale and representative applications, which comprise hundreds of microservices. They are characterized by high-volume traffic and complex architectures with intricate service dependencies and a mix of technologies. Overall, Cast uncovers 137 potential vulnerabilities, out of which 89 have been confirmed through developers' manual investigation at the time of writing. Furthermore, in a controlled experiment against a benchmark of 48 reproduced bugs, Cast achieves a 90% detection coverage. These results demonstrate the effectiveness of our framework in improving the reliability of microservice systems in industrial environments.

In summary, we make the following major contributions:

- We present Cast, an automated, end-to-end framework for microservice resilience testing that combines replay-based fault injection with a complexity-driven strategy to prioritize failure-prone execution paths and prune redundant tests. Cast orchestrates the entire testing lifecycle and employs a multi-faceted oracle to automatically assess resilience against both final outcomes and internal health states.
- We demonstrate the practical effectiveness of Cast through a large-scale deployment in Huawei Cloud over eight months, where it has been used by many development teams. We report on its long-term effectiveness and experimental results on reproduced vulnerabilities, validating its real-world impact.
- We share critical lessons learned from industrial deployment about the importance of deep verification oracles that go beyond API boundaries, the trade-offs between perfect test prioritization and automation efficiency, and practical considerations for integration into existing development workflows.

2 Background

2.1 Microservice Systems

The microservice architectural style has become the de facto standard for building large-scale cloud applications [33]. Unlike monolithic applications, a microservice system is composed of a collection of small, autonomous services, each responsible for a specific business capability. These services are independently developed, scaled, and maintained, offering organizations greater agility and flexibility. They typically communicate with each other over a network using lightweight protocols such as HTTP/REST or gRPC. The overall behavior of the system emerges from the intricate interactions between these services, making it difficult to reason about and predict. A fault in a single service can trigger a cascade of failures that propagates throughout the system, leading to widespread outages.

Microservices do not operate in isolation. They rely heavily on a shared backbone of platform components to manage state, communication, and data persistence. This includes *databases* (e.g., MySQL, MongoDB) for persistent storage, *message brokers* (e.g., Kafka, Pulsar) for asynchronous communication and event-driven workflows, and *in-memory caches* (e.g., Redis, Memcached) for performance and latency. While these components are engineered for high availability, their complex interactions with microservices introduce significant risks for error propagation. The application's ability to handle components unavailability, connection errors, or data serialization issues is paramount to its overall resilience. Thus, a modern microservice system forms an interdependent network of business services and infrastructure, where resilience depends on the robustness of every component and every interaction.

2.2 Resilience Testing of Microservice Systems

Resilience is the ability of a system to withstand failures and maintain an acceptable level of service. For microservice systems, this

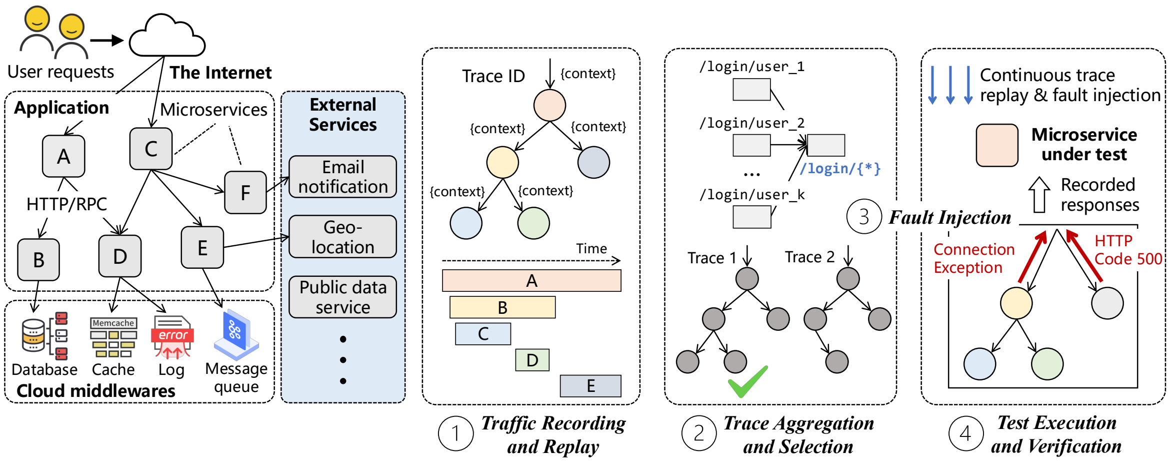


Figure 1: Overall Framework of Cast for Automated Microservice Resilience Testing

means gracefully handling the inevitable failures of services, network, and platform components. Resilience testing (popularized under the term *Chaos Engineering*) aims to proactively discover vulnerabilities by injecting faults to build confidence in a system's survival under volatile production conditions. Existing approaches can be broadly categorized into two streams based on the granularity of fault injection. *Infrastructure-level approaches* [52] focus on manipulating the environment, such as terminating VMs, introducing latency, or dropping packets. While these faults effectively exercise basic resilience tactics like retries and circuit breakers for connectivity issues, they are often too coarse-grained to trigger the application-specific error-handling logic tied to internal service states. In contrast, *application-level approaches* target business logic to verify service fallbacks or recovery procedures. For instance, simulating specific business-level exceptions (e.g., partial API failures or platform errors) requires this finer granularity.

To achieve high fidelity in application-level testing, researchers and practitioners often use two common and often complementary techniques, i.e., traffic record-and-replay and mocking. Record-and-replay generates realistic workloads by capturing and replaying production traffic, ensuring test scenarios reflect actual user interactions. Mocking provides fine-grained control by replacing downstream dependencies with simulated versions that can return programmed failures. However, approaches [9, 10, 36, 45, 46] leveraging these techniques often face the following critical challenges:

- **State Dependency**: Production traffic is stateful and time-sensitive. Directly replaying recorded requests may fail due to stale data, such as expired timestamps, single-use idempotency keys, or invalid tokens. An effective replay system must intelligently identify and refresh these dynamic variables to reflect the current testing context, ensuring the requests are valid.
- **Scalability**: The immense volume of production traffic makes replaying every request infeasible. Furthermore, the number of potential fault injection points (e.g., every service-to-service call, every database query) creates a combinatorial explosion of possible test cases. Naive or random selection of tests is inefficient and may miss critical, yet infrequent, execution paths.
- **Automation**: Fully automating the resilience testing lifecycle presents a significant operational challenge. It requires the complex orchestration of setting up test environments, initiating workloads, and synchronizing fault injections. Another core difficulty is the test oracle problem, i.e., how to automatically and reliably determine whether a test has passed or failed.

3 Methodology

3.1 Overview

To address these challenges, we present Cast, an automated framework for effective microservice resilience testing, targeting Java-based applications. The overall framework of Cast is shown in Fig. 1, which consists of four phases, i.e., *traffic recording and replay*, *trace aggregation and selection*, *fault injection*, and *test execution and verification*. In the first phase, Cast captures live production traffic and applies a lightweight heuristic to identify and parameterize state-dependent variables, transforming raw traces into replayable test templates. To address the immense volume of traffic, the second phase employs a complexity-driven strategy, aggregating similar user interactions and selecting a small set of test cases that target the most failure-prone execution paths. The third phase includes the design of a comprehensive library of realistic, application-level faults and a principled pruning mechanism to identify the most impactful injection targets. The final phase automates the entire test run through a three-stage pipeline and a multi-faceted verification oracle for assessing the system's resilience.

3.2 Traffic Recording and Replay

Cast begins by recording production traffic, which is reconstructed as distributed traces composed of causally-related spans. While industry standards like OpenTelemetry [43] provide capabilities for distributed tracing, they are primarily designed for passive monitoring. Cast requires not only execution recording but also active runtime intervention to support traffic replay and fault injection (e.g., intercepting a database call to return a mock response or throw an exception). Thus, we implement a non-intrusive instrumentation technique using a dynamic AOP framework based on

Java agents [2, 34, 35]. This allows us to intercept method calls at runtime without modifying source code, capturing comprehensive operations including inter-service communications (HTTP/RPC) and interactions with platform components, such as databases (via JDBC), message brokers (e.g., Kafka, Pulsar), and in-memory caches (e.g., Redis). These traces serve as the fundamental artifacts for our subsequent analysis, high-fidelity replay, and resilience testing.

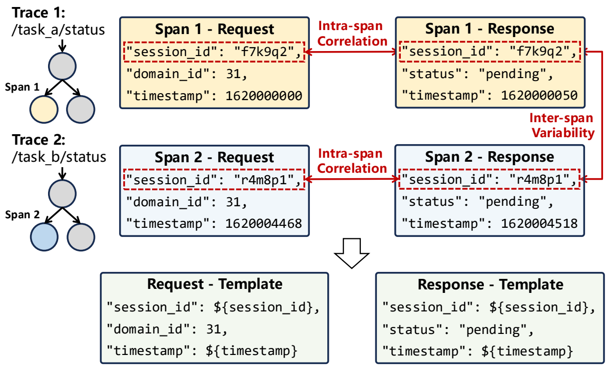
However, as discussed in Sec. 2.2, directly replaying traces often fails due to time-sensitive and state-dependent variables. For example, a request containing an outdated timestamp may get rejected. Also, many API calls include an idempotency key to prevent duplicate processing. Replaying a request with a stale key would be ignored. Liu et al. [35] studied this problem by categorizing these variables as system states, internal states, and external states, and proposed an approach to identify them based on static taint analysis. However, such an approach is not scalable for large service codebases and is tightly coupled to the specific frameworks adopted in [35], i.e., SOFA/SOFABoot. To address this, we introduce a lightweight dynamic variable identification technique that operates directly on the recorded request and response payloads. The core of our approach is a two-stage heuristic designed to distinguish dynamic variables from static data without analyzing source code:

- **Intra-span Correlation**: For each recorded span, we first analyze its request and response payloads to find tokens that appear verbatim in both. The intuition is that many state-dependent variables, such as session IDs or transaction tokens, are often received in a request and propagated directly into the response to maintain context.
- **Inter-span Variability**: To filter out static values (e.g., a hard-coded version string), we then compare these candidate tokens across multiple spans of the same operation. A token is confirmed as a dynamic variable only if its value differs across these independent instances.

Fig. 2 provides a visual walkthrough using two distinct spans recorded from the same logical operation. In the first stage, *Intra-span Correlation*, we identify candidate tokens like `session_id` that appear in both the request and response of a single span. In the second stage, *Inter-span Variability*, we compare these candidates across independent spans. We observe that `session_id` changes (e.g., `"f7k9q2"` vs. `"r4m8p1"`), confirming it is dynamic. Conversely, fields like `domain_id` and `status` remain constant across both spans and are therefore treated as static data. Tokens that satisfy both conditions are abstracted into parameterized placeholders (e.g., `${*}`), transforming recorded traffic into a reusable traffic template. During replay, these placeholders are automatically instantiated based on the test's execution context: unique identifiers like `session id` receive newly generated values, and timestamps are substituted with the current system time. This payload-centric approach treats services as black boxes, making it fundamentally more scalable and framework-agnostic than static taint analysis.

3.3 Trace Aggregation and Selection

After establishing replayable traffic, the next step is to select a representative set of traces for resilience testing, which aims to address two scalability challenges. The first is the immense volume of traces



```
"session_id": "f7k9q2",
"domain_id": 31,
"timestamp": 1620000000
```

```
"session_id": "f7k9q2",
"status": "pending",
"timestamp": 1620000050
```

```
"session_id": "r4m8p1",
"domain_id": 31,
"timestamp": 1620004468
```

```
"session_id": "r4m8p1",
"status": "pending",
"timestamp": 1620004518
```

```
"session_id": ${session_id},
"domain_id": 31,
"timestamp": ${timestamp}
```

```
"session_id": ${session_id},
"status": "pending",
"timestamp": ${timestamp}
```

Figure 2: Trace Templating in Traffic Replay

from production-grade applications, e.g., millions of requests, rendering exhaustive testing impractical. This volume largely stems from redundancy, where repetitive calls to the same interface differ only in parameterized values. For instance, thousands of distinct traces might be generated when different users log in, all invoking the same `POST /api/login/{username}` API. Thus, to avoid redundant testing, we perform traffic aggregation to partition the entire corpus of traces into distinct categories based on their entry point interfaces. Specifically, we employ Drain [26], an algorithm commonly used in log parsing, to dynamically learn the static and variable parts of the root span's request line, which typically includes the HTTP method and URI path. This step effectively distills the vast number of raw calls into a set of distinct service interfaces.

Even after traffic aggregation, we are still facing another scalability problem, i.e., production applications often contain tens or even hundreds of microservices, each of which exposes multiple interfaces. Exhaustively replaying each unique trace to test all interfaces remains infeasible. Instead of random sampling, we develop a heuristic-based scoring mechanism to select a subset of traces that are more likely to reveal resilience issues. While sophisticated algorithms like lineage-driven fault injection (LDFI) [5] exist, they require detailed system models and static analysis that are impractical in our multi-team, polyglot environment with hundreds of rapidly-evolving services. Our approach starts by quantifying *trace complexity* using a weighted combination of three factors: trace length measured by the number of spans, the diversity of components (i.e., the unique services and infrastructure components involved), and the end-to-end duration of the trace execution. The weights are configurable based on specific testing priorities. Traces involving more diverse components and longer execution paths receive higher complexity scores, as they exercise more interactions and are inherently more failure-prone. This per-trace complexity score serves as a building block for our next level of selection by prioritizing microservice interfaces. An interface's complexity can vary significantly based on input parameters and business logic, leading to different execution paths. To capture a holistic measure of an interface's typical complexity, we calculate an aggregate score for it by averaging the complexity scores of all associated traces.

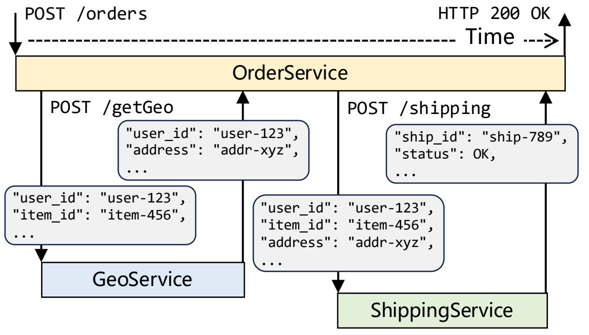
With each interface now assigned a complexity score, we employ a two-level selection strategy to identify the final set of test cases. First, we rank all interfaces by their aggregate score and select the top-K candidates. Second, for each candidate, we select its single most representative test case, i.e., the trace with the highest individual complexity score. This strategy concentrates testing efforts on the system's most complex interfaces and their most challenging execution paths, resulting in a small yet highly potent set of traces for the subsequent fault injection phase.

3.4 Endpoint Coverage-guided Fault Injection

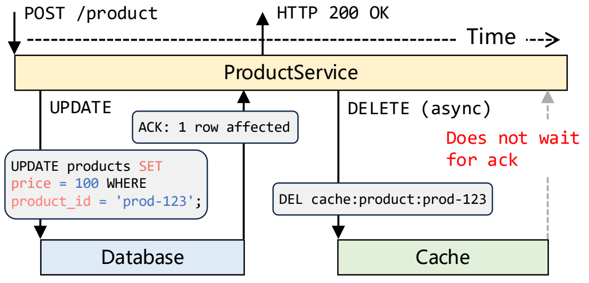
3.4.1 Fault Library Construction. The foundation of our fault injection is a comprehensive and extensible fault library that catalogs realistic failure modes. In particular, we focus on software-level faults to test an application's internal error-handling logic, instead of coarse-grained infrastructure failures. Using dynamic instrumentation, we create injectable fault modules that simulate the following two primary categories of failures:

- *Platform components exceptions*: This category simulates common exceptions thrown by client libraries of different platform components. To build a comprehensive list, we conduct a systematic analysis that combines: 1) a thorough review of official API documentation to identify documented failure modes; 2) an empirical analysis of historical incidents to prioritize high-impact errors; and 3) interviews with senior developers and SREs to capture expert domain knowledge. This has yielded a rich library of faults targeting various platform component categories, such as database (e.g., `SQLTimeoutException`), message brokers (e.g., `SerializationException`), and cache (e.g., `RedisConnectionException`). This fault library is not static but is continuously updated to reflect changes in our technology stack and to incorporate newly identified failure patterns.
- *Inter-service communication errors*: This category models synchronous communication (e.g., HTTP/RPC) failures by intercepting outgoing requests to simulate a wide range of disruptions. This includes introducing network delay (latency) to test for slow dependencies or throwing protocol-level exceptions like `java.net.SocketTimeoutException` to simulate a complete connection failure. Moreover, instead of making the actual network call, they can return a fully manipulated failure response. Our library provides extensive coverage for this, allowing for the alteration of the entire response, including standard HTTP status codes, such as Client Errors (e.g., `401 Unauthorized`) and Server Errors (e.g., `500 Internal Server Error`, `504 Gateway Timeout`), and the response body itself.

3.4.2 Fault Injection Target Identification. Based on the selected traces (Sec. 3.3) and fault library, the next step is to determine where to inject the faults, i.e., *fault injection targets*. To achieve comprehensive fault coverage, we introduce a fine-grained abstraction for injection targets called an *endpoint*. An endpoint is a tuple (`Component`, `Framework`, `Method`): `Component` defines the semantic context of the interaction (e.g., Database, Cache); `Framework` identifies the specific library or implementation (e.g., MyBatis, Jedis), and `Method` specifies the operation (e.g., update, get). This abstraction allows us to distinguish functionally identical operations implemented via different frameworks in different teams, each possessing unique failure-handling characteristics. For instance, we can differentiate critical endpoint pairs such as `RPC-gRPC-findUserById` vs. `RPC-Dubbo-findUserById` for user lookups, or `MQ-Kafka-send` vs.


POST /orders
HTTP 200 OK
Time
OrderService
POST /getGeo
"user_id": "user-123", "address": "addr-xyz", ...
"user_id": "user-123", "item_id": "item-456", ...
GeoService
POST /shipping
"ship_id": "ship-789", "status": OK, ...
"user_id": "user-123", "item_id": "item-456", "address": "addr-xyz", ...
ShippingService

(a) Producer-consumer Data Dependency Pattern



(b) Dual-write Data Dependency Pattern

Figure 3: Data-flow Dependency Patterns

`MQ-Pulsar-send` for messaging operations. This level of precision is essential for systematically discovering implementation-specific vulnerabilities. Each endpoint serves as a target for relevant faults from our library, allowing testing various failures for a single operation. For example, an `MQ-Kafka-send` endpoint can be injected with a `TimeoutException` to test for network issues or with a `SerializationException` to simulate a data format mismatch.

Technically, every span of a trace corresponds to one endpoint, as it records an operation performed within the microservice system (Sec. 3.2). However, as a single complex trace can contain hundreds or even thousands of spans, injecting faults into all its endpoints creates a combinatorial explosion of potential test cases. To address this, Cast employs a principled pruning mechanism to systematically reduce the test space by eliminating redundant tests and prioritizing high-value tests through the following three strategies:

- *Cross-service sampling*: The fault-handling logic for a given endpoint (e.g., a database call) is often implemented in a shared library or framework, making its behavior consistent across different services. It is inefficient to exhaustively test this same endpoint in every microservice that invokes it. Therefore, we randomly sample a small, configurable number of distinct microservices (e.g., three) for fault injection. This approach recognizes that resilience issues may still manifest differently across implementations despite using identical frameworks.
- *Intra-trace redundancy elimination*: A single trace may contain multiple spans of the same endpoint (e.g., repeated database queries). To avoid redundancy, we inject faults only at the last invocation. The rationale is that the final invocation operates

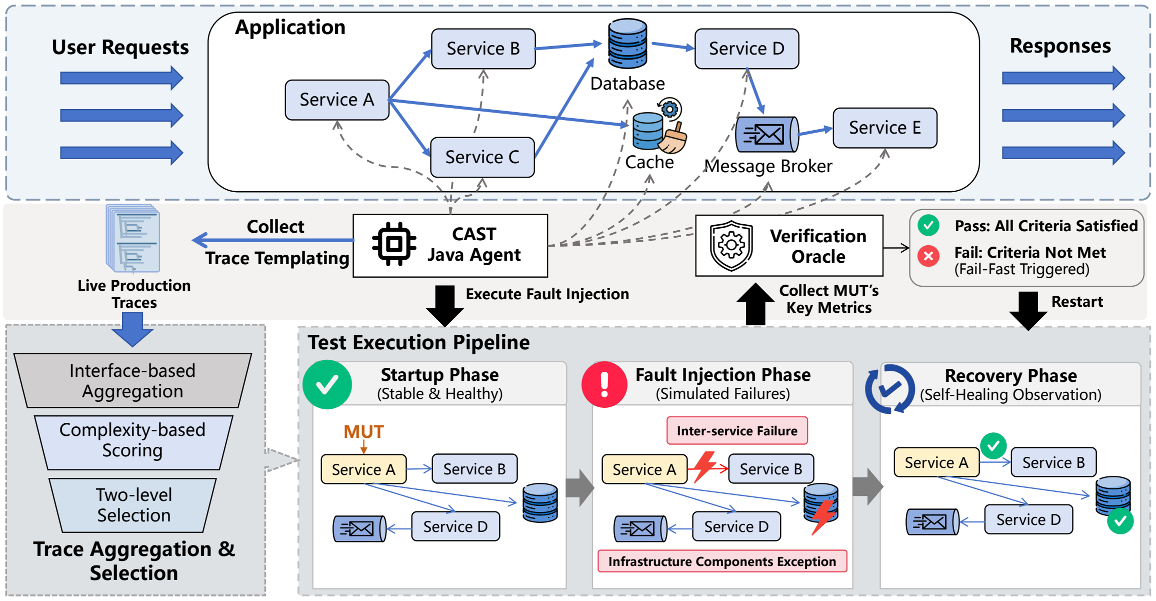


Figure 4: Test Execution Pipeline of the CAST System

on a more advanced system state, influenced by all preceding operations, making it more likely to expose complex, state-dependent vulnerabilities. Furthermore, for database transactions, faults at the last invocation test the system's ability to rollback a fully-staged transaction, which is typically more error-prone than rolling back earlier, partial operations.

- *Data-flow dependency prioritization*: This strategy prioritizes injection targets by analyzing the structure of data-flow dependencies within traces, as they are prone to causing cascading failures. Based on the data overlap in payloads and operation sequences, we identify two fundamental dependency structures, as shown in Fig. 3. The first pattern, *Producer-Consumer* (Fig. 3a), involves a service (e.g., OrderService) first calling a component to retrieve data (e.g., calling GeoService to validate an address) and then using that produced data in a subsequent call to another component (e.g., calling ShippingService with the validated address). The integrity of this entire workflow depends on the successful flow of data from the producer (GeoService) to the consumer (ShippingService). We prioritize the producer as a high-value injection target because its failure can propagate through the data chain, allowing us to skip testing downstream consumers in isolation for the same trace. The second pattern, *Dual-Write* (Fig. 3b), occurs when the same set of data in a service (e.g., the product ID and price in ProductService) is used to invoke two or more separate operations (e.g., `UPDATE` in the Database and `DELETE` in the Cache). This pattern is common during data synchronization tasks and is highly susceptible to silent but critical inconsistency bugs. Particularly, the second write to the cache is often performed asynchronously for performance reasons, where the service does not wait for an ack. In this case, we bypass the primary synchronous write and focus exclusively on the secondary/asynchronous operation. Since the primary write is typically covered by the Producer-Consumer or individual service tests, focusing on the secondary write targets the specific source of silent data inconsistency while significantly reducing the number of required test cases. As discussed in Sec. 3.5, this scenario necessitates a more sophisticated verification oracle.

This principled pruning mechanism yields a set of fault injection test cases with strategic diversity, covering a wide spectrum of operations and frameworks in microservices. It allows us to validate the common fault-handling patterns without the cost of exhaustive testing, striking a balance between coverage and efficiency.

3.5 Automated Test Execution and Verification

With the fault injection test cases, the final phase involves their systematic execution and the verification of the system's resilience.

3.5.1 Execution Pipeline and Test Scheduling. Each individual test case is executed within a well-defined, three-phase cycle. Fig. 4 provides a runtime perspective of this process, detailing how the Java agent interacts with the microservice under test (MUT) and platform components during each phase. Particularly, each phase is configured with a user-defined duration and injection frequency to ensure sufficient time for the system to stabilize and for effects to be observed. In this process, we continuously generate traffic against the MUT by replaying the selected trace.

- *Startup phase*: The MUT is deployed as a standalone, on-demand container. To ensure strict data isolation and avoid polluting shared environments, we initialize a fresh instance for each test. No faults are injected in this phase to ensure that the system is operating in a stable, healthy state.
- *Fault injection phase*: Selected faults are activated, targeting the designated endpoints. This allows us to monitor the system's behavior under specific failure conditions.

- *Recovery phase*: Fault injection is deactivated. This final phase is crucial for observing whether the system can autonomously recover and return to its original healthy operational state.

A practical challenge in this process is the overhead associated with the MUT's startup. This process can be time-consuming as it involves the initialization of the application and environments, as well as the time for the service to reach its stable operational state. Executing only one fault injection per startup would be prohibitively slow. To amortize this cost, CAST groups multiple fault injection tests into a single run. We employ a greedy scheduling algorithm that iteratively selects traces providing the maximum increase in new endpoint coverage. This minimizes the number of required startups to execute all fault injection tests. Within a single test run, these batched tests are executed sequentially.

However, such a design introduces a potential problem of state contamination, where a fault from a preceding test alters the system state and influence subsequent tests. To mitigate this, CAST adopts a "fail-fast" execution model. Specifically, as long as each test passes, the pipeline proceeds to the next one. When a test fails, indicating a potential vulnerability, the entire run is halted immediately. The system is then fully restarted for the next test, ensuring that the remaining tests start from a clean state. We acknowledge the possibility that even a previous passing test could subtly alter system state. However, we consider this a pragmatic trade-off for the significant efficiency gains. In practice, such scenarios are rare, and our cross-service sampling strategy provides inherent redundancy. Furthermore, to enhance long-term effectiveness across multiple testing cycles, CAST maintains a history of executed test cases. When scheduling new tests from subsequent batches of recorded traffic, this history is used to avoid re-executing any identical tests.

3.5.2 Verification Oracle. To automatically determine the outcome of a test, CAST employs a multi-faceted verification oracle with the following two key components:

- *Phase-based performance criteria*: Our oracle defines a set of performance criteria, which are quantifiable assumptions about the MUT's key metrics (e.g., success rate, latency, and throughput). Specific criteria values are derived from historical trace data, which can vary depending on the execution phase. For instance, the startup phase requires a 100% success rate to confirm correct system initialization. To verify that the fault is having an impact, the success rate during the fault injection phase is expected to drop significantly (e.g., $\leq$ 30%). During the recovery phase, the success rate must return to a healthy level (e.g., $\geq$ 80%) to demonstrate the system's ability to self-heal. This threshold is intentionally set below 100% to account for a realistic recovery time window as the system stabilizes. While this trade-off between automation and absolute precision may occasionally cause false positives or negatives (discussed in Sec. 4.2), it enables practical automated verification.
- *Granular assertion points*: A naive approach of checking only the final HTTP response code at the service entry point is often insufficient. We observed that for certain asynchronous operations, such as publishing a message to Pulsar, the call might return successfully to the application logic even if the Pulsar subsequently fails to deliver the message to the consumer. This "fire-and-forget" behavior, where the service considers the task complete once submitted without awaiting its actual delivery, would lead to an incorrect pass judgment. would lead to an incorrect pass judgment. To solve this, CAST establishes assertion points not only at the service entry point but also directly at the internal endpoint where the fault is injected. This dual-level verification ensures we accurately assess both the fault's direct impact and its effect on the service's overall behavior.

A test is considered passed only when the observed behavior meets the defined criteria for all three phases across all relevant assertion points. Failed tests are flagged for further investigation.

4 Evaluation

In this section, we evaluate the performance of CAST in our production deployments. In particular, we aim to answer the following two research questions:

- **RQ1 (Replayability)**: How effective is CAST in making recorded traffic replayable?
- **RQ2 (Effectiveness)**: How effective is CAST in automatically discovering resilience vulnerabilities?

CAST is deployed as a centralized resilience testing platform within Huawei Cloud. For over eight months, CAST has been used by many development teams to proactively identify resilience issues in their applications. Given CAST's role as a shared platform where teams can independently run tests and analyze results, it is impractical to manually track and confirm the outcome of every potential vulnerability reported across all service teams. Therefore, for this paper, we focus our analysis on the testing results from four representative applications (anonymized as Service 1-4). They are specifically chosen because their characteristics align with the challenges CAST is designed to address, i.e., large scale (hundreds of microservices), high volume of business-critical traffic, and architectural complexity with intricate dependencies.

4.1 RQ1: The Replayability of Recorded Traffic

To answer RQ1, we evaluate the effectiveness of CAST's dynamic variable identification technique in making production traffic replayable. A successful replay is defined as a replayed request that is accepted by the target service and returns an expected success code (e.g., HTTP 2xx), establishing a baseline before fault injection.

We begin by collecting six hours of traffic (over three million traces) from the target applications. CAST's aggregation module distills this corpus into 291 unique service interface calls. For each interface, CAST analyzes all of its associated trace instances to apply the two-stage heuristic (Sec. 3.2), automatically identifying and parameterizing the time-sensitive and state-dependent variables. To validate the outcome, we randomly select one trace instance for each of the 291 interface calls and attempt to replay it against the live test system. Out of the 291 replay attempts, CAST successfully replays 286 traces, improving the success rate from below 80% (for naive replay) to 98.3% with our templating technique. This high success rate demonstrates that our automated approach is highly effective at handling the state-dependency problem for the vast majority of interfaces. Repeated experiments with different random samples yield consistent results, confirming the method's stability.

We manually investigate the 5 failed cases and find that they are primarily caused by highly business-specific or system-specific

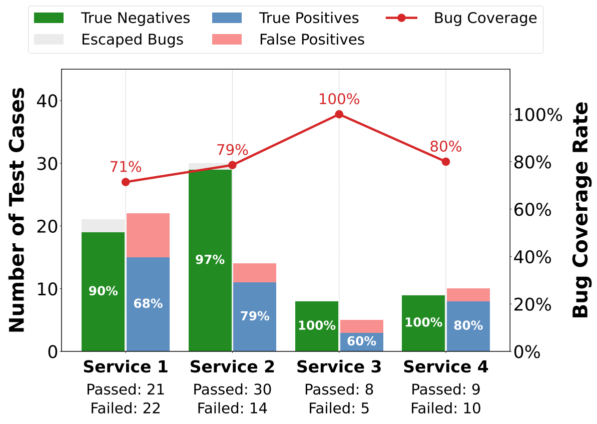


Figure 5: Experimental Results on Different Services

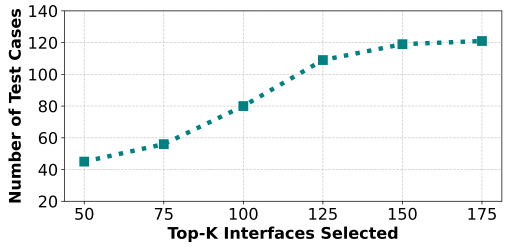
variables that escape our general-purpose identification method. For instance, some requests include dynamically computed security signatures or tokens derived from complex, proprietary business logic not reflected in the request-response payload structure. To address these edge cases in a practical industrial setting, CAST maintains a configurable list, allowing engineers to manually register or deregister variables for a given service. This hybrid approach ensures that comprehensive replay coverage can be achieved with minimal, one-time manual effort for these outliers.

4.2 RQ2: The Effectiveness of Resilience Bug Detection

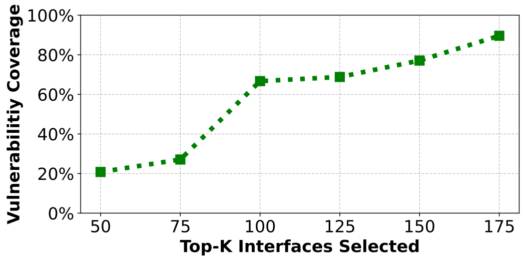
To evaluate CAST's effectiveness, we first consider its long-term impact in real-world production systems. Over an eight-month deployment in Huawei Cloud, CAST has been continuously adopted by different teams to improve service reliability. For this paper, we present the results of CAST's utilization across four large-scale and representative applications. Overall, CAST successfully uncovers 137 potential vulnerabilities, of which 89 have been confirmed through developers' manual investigation at the time of writing. This demonstrates CAST's practical value in production.

For a more controlled, quantitative analysis of its detection capability, we collect a ground truth dataset of 48 confirmed vulnerabilities. These cases consist of historical defects identified through incident reports and manual investigation. It provides a comprehensive benchmark for our experiment. We use the same six-hour corpus of recorded traffic from RQ1 to generate test cases against the systems containing these bugs. Following our complexity-driven selection strategy (Sec. 3.3), we select the top-150 interfaces with the highest complexity scores and their representative traces. We then apply our pruning logic (Sec. 3.4) to identify endpoints. For each type of endpoint, we randomly select at most three different microservices for testing. This process yields 119 distinct test cases that cover 477 fault injection targets.

The detailed results in Fig. 5 show that upon executing the 119 test cases, 68 passed and 51 failed. CAST demonstrates a high true negative rate, meaning nearly all passed cases are indeed free of bugs. Manual investigation of the 51 failed tests reveals that CAST successfully identifies 37 of the 48 known vulnerabilities, achieving



(a) The Number of Test Cases vs. Top-K Interfaces Selected



(b) Vulnerability Coverage vs. Top-K Interfaces Selected

Figure 6: Sensitivity Analysis with Varying Top-K Interfaces

a consistently high bug coverage across different services (77.1% in aggregate). The remaining 14 failed tests are identified as false positives. In these cases, faults are correctly injected and the system's behavior changes, but this does not represent a genuine resilience bug. Detailed analysis reveals that 9 of these false positives are caused by our default performance thresholds being overly sensitive for certain non-critical background operations (e.g., metrics collection, logging), while 5 are due to legitimate degraded-but-acceptable behavior under fault conditions. Similarly, for the 3 false negatives, a test case is executed but the oracle incorrectly reports a "pass." This typically occurs when the impact of a fault is too subtle to violate the default performance thresholds, e.g., success rates drop to 35% (just above the 30% threshold) but still represent a significant degradation. These results highlight the inherent trade-offs in automated oracle design between sensitivity and specificity. To address this, CAST allows developers to configure custom thresholds, thereby refining the oracle's precision over time.

We also analyze the 11 vulnerabilities that CAST fails to detect. Besides the 3 false negatives, the remaining 8 bugs are due to test generation limitations where no test is ever created for the vulnerable paths. Particularly, 5 of these bugs are missed because the recorded traffic corpus does not contain requests exercising the vulnerable paths, a fundamental limitation of any record-and-replay-based approach. The other 3 are associated with interfaces whose complexity scores fall outside our top-K selection. To mitigate this issue, CAST incorporates a historical execution-aware selection mechanism. This mechanism maintains a history of previously executed tests. When generating a new test suite, it prioritizes fault injections on interfaces that have not yet been tested or have previously failed, effectively avoiding the repetition of passed tests. This history is periodically reset (e.g., weekly) to ensure the system is re-validated against potential regressions, providing a balance between exploring new test space and continuous verification.

4.3 Parameter Sensitivity Analysis

To further investigate the trade-off between test suite size and bug coverage, we conduct a sensitivity analysis by varying the number of top-K interfaces from 50 to 175, as shown in Fig. 6. We first examine the execution cost, which is quantified by the number of generated test cases (Fig. 6a). The test suite grows substantially as the initial interfaces are added, since they introduce numerous unique endpoints. However, the curve begins to flatten as K increases, because the new traces are likely to contain endpoints that have already been covered by the previous traces. This demonstrates the effectiveness of our pruning logic in eliminating redundant test cases. Next, Fig. 6b shows the corresponding vulnerability coverage. The curve shows a critical inflection point between K=75 and K=100, where coverage dramatically increases to nearly 70%. This validates our endpoint coverage-guided strategy for its ability to locate critical fault injection targets with sufficient traces. Similar to Fig. 6b, this curve also shows diminishing returns, with each additional trace contributing progressively less to the overall coverage.

These two curves highlight the fundamental trade-off between coverage and cost. While a large K can achieve higher coverage in a single run, it demands the execution of a larger number of test cases, which may be impractical for frequent testing cycles. This is where Cast's historical execution-aware mechanism becomes critical. Instead of relying on a single, exhaustive run, teams can use a smaller K (e.g., 100) for daily or continuous testing. Over successive runs, Cast's history tracking ensures that it avoids re-testing previously passed scenarios and gradually explores other interfaces. This strategy allows Cast to achieve high cumulative coverage over time with smaller daily executions.

4.4 Case Studies

To further illustrate the types of bugs discovered by Cast and demonstrate the effectiveness of its design principles, we present two representative case studies from our deployments.

- **Case 1: Cascading Failure due to Unhandled HTTP Timeout.** A microservice, Service A, is responsible for processing uploaded data. Its workflow involves receiving a request, querying a third-party service via an HTTP REST API to validate and supplement the incoming information, and then publishing the final message to a Pulsar queue. When Cast injects a `java.net.SocketTimeoutException` into this HTTP call, the worker thread hangs indefinitely due to missing timeout policy. This causes the initial request to fail and prevents the thread from processing any subsequent requests, leading to a complete and persistent failure of that business function. The root cause is that the RestTemplate HTTP client used to call the third-party service is instantiated without a read timeout. This highlights the value of Cast's data-flow dependency prioritization. The workflow represents a classic Producer-Consumer pattern (User $\longrightarrow$ Service A $\longrightarrow$ 3rd Party Service $\longrightarrow$ Pulsar), and Cast's complexity-driven selection prioritizes such multi-hop traces as they are more likely to cause cascading failures.
- **Case 2: Silent Failure in Asynchronous Communication.** A service responsible for adding new data entries first writes the data to a database and then publishes a notification event to a Kafka topic. When Cast injects a `DisconnectException` into the Kafka publish process, the service fails to capture this error. Consequently, the original REST API call still returns an HTTP 200 OK success code to the caller, despite the internal failure. The data is correctly persisted, but downstream services are never notified, leading to system-wide data inconsistency. The root cause is a "fire-and-forget" pattern where the application triggers the asynchronous Kafka publish operation but fails to check its return value or handle exceptions. This vulnerability underscores the critical need for Cast's granular assertion points. A naive oracle checking only the final HTTP response would miss this bug. By placing an assertion point directly at the internal Kafka producer endpoint, Cast correctly identifies the failure, revealing the silent but critical issue.

4.5 Threats to Validity

External Validity. Our evaluation is conducted solely on Huawei Cloud, which may limit its generalizability. However, we believe our findings are broadly applicable for the following reasons. First, the architecture patterns, platform components (Kafka, Redis, MySQL), and communication protocols (HTTP, gRPC) we target are widely adopted in industry. Second, our design principles, i.e., black-box traffic analysis, application-level fault injection, and multi-faceted oracles, are platform-agnostic. Finally, the challenges we address (state dependencies, test space explosion, oracle design) are fundamental to resilience testing regardless of the specific cloud platform.

Internal Validity. Several factors may affect the accuracy of our results. First, reliance on threshold-based oracles may cause false positives or negatives. To mitigate this, Cast allows teams to customize thresholds based on their requirements and continuously refine them through operational feedback (Sec. 3.5). Second, our complexity-driven selection is heuristic and may miss bugs in simpler interfaces. We address this through our historical execution-aware mechanism that progressively explores untested interfaces over multiple runs, achieving cumulative coverage. Third, the record-and-replay approach is limited to execution paths present in captured traffic. While this is a fundamental constraint, we mitigate it by continuously recording new traffic patterns and maintaining a diverse trace corpus that grows over time, capturing an increasingly comprehensive set of execution scenarios.

5 Lessons Learned

The design, deployment, and evaluation of Cast in an industrial environment have yielded several critical insights into the nature of resilience testing for microservice systems. We distill our experience into the following key lessons.

Verification Oracles Must Go Beyond the API Boundary. Early in our work, we considered a simpler oracle model that only checked the final, user-facing API response (e.g., HTTP 200 OK), a common approach in existing research. This proved to be insufficient. We repeatedly encountered scenarios, particularly with asynchronous operations, where the primary API call would succeed while a critical background operation failed silently. As shown in our case study (Sec. 4.4), this can lead to severe data inconsistencies that go undetected by simple oracles. This experience underscores a fundamental lesson: effective resilience testing requires deep, internal visibility. A trustworthy verification oracle must establish assertion points not only at the service boundary but also at critical internal endpoints, such as platform component interactions, to catch failures that do not immediately propagate to the user.

Cumulative Coverage Through Automation Outweighs Perfect Prioritization. While many studies [16, 17, 37, 38] leverage sophisticated strategies to prioritize fault injection points, we learned that in a complex, multi-team industrial setting, it is challenging to predict bug locations. We found that the practical value of a testing system can be approximated by: *Value = (per-run Coverage × Frequency) / Effort*. A "perfect" prioritization strategy that requires substantial manual effort or complex system models may achieve high per-run coverage but low frequency due to operational overhead. In contrast, our "good enough" complexity-driven heuristic, combined with full automation, enables daily or weekly testing runs. The key is to design a system that is history-aware to intelligently avoid re-testing passed scenarios. This allows the framework to accumulate effectiveness over time, systematically exploring new and lower-priority interfaces in successive runs. This iterative approach achieves high cumulative coverage over time with smaller, more manageable daily executions, providing a more practical and robust path to improving system reliability.

Engineering Trade-offs Are Inevitable in Industrial Systems. While academic prototypes often show promise on simple applications, industrial systems present challenges of complexity and scale that demand different engineering trade-offs. Throughout Cast's development, we faced numerous design decisions where theoretical optimality conflicted with practical constraints. For instance, our payload-based variable identification heuristic is less precise than static taint analysis but scales to hundreds of services without source code access. Our fixed oracle thresholds occasionally cause false positives but enable full automation without per-test manual configuration. Similarly, our complexity metric is simpler than graph-based algorithms but requires no system models or dependency maintenance. These trade-offs reflect a fundamental reality of industrial software engineering, namely, a deployed system that finds 80% of bugs is more valuable than a perfect system that remains in the research lab. The key is to make these trade-offs consciously, provide configuration options for teams with specific needs, and continuously iterate based on operational feedback.

6 Related Work

6.1 Resilience Testing and Chaos Engineering

Resilience testing through deliberate fault injection has evolved from coarse-grained infrastructure failures to sophisticated testing at the application level. Early approaches like Netflix's Chaos Monkey [13, 14] pioneered the practice of randomly terminating virtual machines to test system recovery. While effective for infrastructure robustness, these methods fail to exercise application-specific error-handling logic [27, 40]. This limitation motivated finer-grained approaches. For example, Gremlin [27] intercepts inter-service messages, Filibuster [40] combines static analysis with concolic execution for HTTP services, while Doctor [1], Setsudo [32], and FATE/DESTINI [25] inject lower-level I/O failures.

The core challenge in resilience testing is managing the combinatorial explosion of fault space [37, 52]. Sophisticated prioritization strategies have emerged to address this. For example, LDFI [5] reasons backward from correct executions to identify minimal fault sets, IntelliFT [37] uses fitness-guided feedback to steer toward impactful scenarios, and MicroFI [16] employs PageRank algorithms for prioritization. While theoretically elegant, these approaches face significant adoption barriers in industrial settings. LDFI requires deterministic execution models impractical for multi-team environments, fitness-guided methods need multiple iterations unsuitable for large-scale systems, and graph-based approaches struggle with dynamic service topologies. Recent surveys [44] and quantum-ML optimizations [12] further highlight the complexity-efficiency trade-off. Cast addresses these practical constraints through a complexity-driven heuristic that requires no system models and converges in a single pass—trading theoretical optimality for operational feasibility, as our eight-month deployment validates [4, 42, 47, 56].

The evolution of testing frameworks reflects industrial needs. Specifically, CHESS [39] targets self-adaptive systems, adaptive chaos engineering uses reinforcement learning, Rainmaker [18] provides push-button testing for cloud applications and MicroRes [54] profiles degradation patterns. Recent deployments reveal practical insights. For example, Chen et al. [19] demonstrated 80% superior response stability in cloud-edge environments through extensive fault injection. The oracle problem remains critical. While LLM-based approaches like CANDOR [53] generate oracles through consensus, and SATORI [3] derives them from API specifications [24, 31], Cast employs pragmatic threshold-based oracles with configurable parameters, enabling full automation without per-test manual configuration.

6.2 Traffic Record-and-Replay and Mocking

Creating realistic test scenarios through traffic record-and-replay has become essential for high-fidelity testing [10, 35]. The fundamental challenge is state dependency, i.e., production traffic contains time-sensitive variables that cause replay failures. Web application tools like WaRR [6], Mugshot [41], and WebRR [36] address this through various browser-specific mechanisms. In microservices, the challenge is amplified due to distributed state across services. Liu et al. [35] used static taint analysis for variable identification but require SOFA/SOFABoot framework coupling, while Zhu et al. [59] leveraged AI for mocking point recommendations [8, 9].

Mocking, traditionally used in unit testing [45, 46], is increasingly integrated into system-level resilience testing. EvoMaster [57] supports test seeding and service mocking, while MicroFuzz [21] explicitly targets microservice environmental complexities. Cast distinguishes itself through a black-box, payload-centric approach that treats services as opaque entities. Our two-stage heuristic, i.e., identifying tokens appearing in both request and response, then filtering by cross-span variability, achieves 98.3% replay success without source code access or framework dependencies. This design choice, while potentially less precise than white-box analysis [34], proves crucial for polyglot industrial environments where teams use diverse languages and frameworks. Unlike approaches requiring extensive system knowledge or training data, Cast provides immediate value through lightweight dynamic analysis, making it practical for the continuous evolution of production systems.

7 Conclusion

In this paper, we present Cast, a framework that addresses the critical challenges of resilience testing in production microservice systems. The core contribution of Cast is its holistic methodology for creating realistic, scalable, and fully automated resilience tests. It bridges the gap between testing and production reality by combining high-fidelity workloads, derived from live traffic, with a library of fine-grained, application-level faults. To navigate the combinatorial complexity, Cast provides a complexity-driven approach that prunes the vast test space, enabling an efficient yet comprehensive evaluation of the most critical execution paths. Deployed on Huawei Cloud for over eight months, Cast successfully uncovers 137 potential resilience vulnerabilities, with 89 confirmed by developers. Furthermore, we also perform controlled experiments on a benchmark set of 48 reproduced bugs, where Cast achieves a high bug coverage of 90%. These results validate that our approach not only achieves high replayability of production traffic but also efficiently discovers critical, real-world vulnerabilities.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 62402536).

References

[1] 1995. DOCTOR: an integrated software fault injection environment for distributed real-time systems. In *Proceedings of the International Computer Performance and Dependability Symposium on Computer Performance and Dependability Symposium (IPDS '95)*. IEEE Computer Society, USA, 204.

[2] Alibaba. 2025. JVM Sandbox. Retrieved December, 2025 from https://github.com/alibaba/JVM-Sandbox

[3] Juan C. Alonso, Alberto Martin-Lopez, Sergio Segura, Gabriele Bavota, and Antonio Ruiz-Cortés. 2025. SATORI: Static Test Oracle Generation for REST APIs. *CoRR* abs/2508.16318 (2025). arXiv:2508.16318 doi:10.48550/ARXIV.2508.16318

[4] Peter Alvaro, Kolton Andrus, Chris Sanden, Casey Rosenthal, Ali Basiri, and Lorin Hochstein. 2016. Automating Failure Testing Research at Internet Scale. In *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*, Marcos K. Aguilera, Brian Cooper, and Yanlei Diao (Eds.). ACM, 17–28. doi:10.1145/2987550.2987555

[5] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. 2015. Lineage-driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 331–346. doi:10.1145/2723372.2723711

[6] Silviu Andrica and George Candea. 2011. WaRR: A tool for high-fidelity web application record and replay. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*. IEEE Compute Society, 403–410. doi:10.1109/DSN.2011.5958253

[7] Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM Trans. Softw. Eng. Methodol.* 28, 1 (2019), 3:1–3:37. doi:10.1145/3293455

[8] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2014. Automated unit test generation for classes with environment dependencies. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, 79–90. doi:10.1145/2642937.2642986

[9] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2015. Generating TCP/UDP network data for automated unit test generation. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 155–165. doi:10.1145/2786805.2786828

[10] Nipun Arora, Jonathan Bell, Franjo Ivancic, Gail E. Kaiser, and Baishakhi Ray. 2018. Replay without recording of production bugs for service oriented applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 452–463. doi:10.1145/3238147.3238186

[11] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: stateful REST API fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 748–758. doi:10.1109/ICSE.2019.00083

[12] Gopichand Bandarupalli. 2025. The Impact of Software Testing with Quantum Optimization Meets Machine Learning. *CoRR* abs/2506.02090 (2025). arXiv:2506.02090 doi:10.48550/ARXIV.2506.02090

[13] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. 2016. Chaos Engineering. *IEEE Softw.* 33, 3 (2016), 35–41. doi:10.1109/MS.2016.60

[14] Cory Bennett and Ariel Tseitlin. 2012. Chaos monkey released into the wild. *Netflix Tech Blog* 30, 1 (2012).

[15] Matteo Camilli, Antonio Guerriero, Andrea Janes, Barbara Russo, and Stefano Russo. 2022. Microservices Integrated Performance and Reliability Testing. In *IEEE/ACM International Conference on Automation of Software Test, AST@ICSE 2022, Pittsburgh, PA, USA, May 21-22, 2022*. ACM/IEEE, 29–39. doi:10.1145/3524481.3527233

[16] Hongyang Chen, Pengfei Chen, Guangba Yu, Xiaoyun Li, and Zilong He. 2024. MicroFI: Non-Intrusive and Prioritized Request-Level Fault Injection for Microservice Applications. *IEEE Trans. Dependable Secur. Comput.* 21, 5 (2024), 4921–4938. doi:10.1109/TDSC.2024.3363902

[17] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. 2020. CoFI: Consistency-Guided Fault Injection for Cloud Systems. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 536–547. doi:10.1145/3324884.3416548

[18] Yinfang Chen, Xudong Sun, Suman Nath, Ze Yang, and Tianyin Xu. 2023. Push-Button Reliability Testing for Cloud-Backed Applications with Rainmaker. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, Mahesh Balakrishnan and Manya Ghobadi (Eds.). USENIX Association, 1701–1716. https://www.usenix.org/conference/nsdi23/presentation/chen-yinfang

[19] Zihao Chen, Mohammad Goudarzi, and Adel Nadjaran Toosi. 2025. Resilience Evaluation of Kubernetes in Cloud-Edge Environments via Failure Injection. *CoRR* abs/2507.16109 (2025). arXiv:2507.16109 doi:10.48550/ARXIV.2507.16109

[20] Zhuangbin Chen, Yu Kang, Liqun Li, Xu Zhang, Hongyu Zhang, Hui Xu, Yangfan Zhou, Li Yang, Jeffrey Sun, Zhangwei Xu, Yingnong Dang, Feng Gao, Pu Zhao, Bo Qiao, Qingwei Lin, Dongmei Zhang, and Michael R. Lyu. 2020. Towards intelligent incident management: why we need it and how we make it. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1487–1497. doi:10.1145/3368089.3417055

[21] Peng Di, Bingchang Liu, and Yiyi Gao. 2024. MicroFuzz: An Efficient Fuzzing Framework for Microservices. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 216–227. doi:10.1145/3639477.3639723

[22] FudanSELab. 2021. Train Ticket - A Benchmark Microservice System. Retrieved July, 2025 from https://github.com/FudanSELab/train-ticket

[23] Luca Giamattei, Antonio Guerriero, Roberto Pietrantuono, and Stefano Russo. 2022. Automated Grey-Box Testing of Microservice Architectures. In *22nd IEEE International Conference on Software Quality, Reliability and Security, QRS 2022, Guangzhou, China, December 5-9, 2022*. IEEE, 640–650. doi:10.1109/QRS57517.2022.00070

[24] Suavis Giramata, Madhusudan Srinivasan, Venkat Naidu Gudivada, and Upulee Kanewala. 2025. Efficient Fairness Testing in Large Language Models: Prioritizing Metamorphic Relations for Bias Detection. *CoRR* abs/2505.07870 (2025). arXiv:2505.07870 doi:10.48550/ARXIV.2505.07870

[25] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. 2011. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*, David G. Andersen and Sylvia Ratnasamy (Eds.). USENIX Association. https://www.usenix.org/conference/nsdi11/fate-and-destini-framework-cloud-recovery-testing

[26] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. 2017. Drain: An Online Log Parsing Approach with Fixed Depth Tree. In *2017 IEEE International Conference on Web Services, ICWS 2017, Honolulu, HI, USA, June 25-30, 2017*, Ilkay Altintas and Shiping Chen (Eds.). IEEE, 33–40. doi:10.1109/ICWS.2017.13

[27] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K. Reiter, and Vyas Sekar. 2016. Gremlin: Systematic Resilience Testing of Microservices. In *36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016, Nara, Japan, June 27-30, 2016*. IEEE Computer Society, 57–66. doi:10.1109/ICDCS.2016.11

[28] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. 2018. Capturing and Enhancing In Situ System Observability for Failure Detection. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 1–16. https://www.usenix.org/conference/osdi18/presentation/huang

[29] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. 2017. Gray Failure: The Achilles' Heel

of Cloud-Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017*, Alexandra Fedorova, Andrew Warfield, Ivan Beschastnikh, and Rachit Agarwal (Eds.). ACM, 150–155. doi:10.1145/3102980.3103005

[30] Mingxuan Hui, Lu Wang, Hao Li, Ren Yang, Yuxin Song, Huiying Zhuang, Di Cui, and Qingshan Li. 2025. Unveiling the microservices testing methods, challenges, solutions, and solutions gaps: A systematic mapping study. *J. Syst. Softw.* 220 (2025), 112232. doi:10.1016/J.JSS.2024.112232

[31] Shan Jiang, Chenguang Zhu, and Sarfraz Khurshid. 2024. Generating executable oracles to check conformance of client code to requirements of JDK Javadocs using LLMs. *CoRR* abs/2411.01789 (2024). arXiv:2411.01789 doi:10.48550/ARXIV.2411.01789

[32] Pallavi Joshi, Malay K. Ganai, Gogul Balakrishnan, Aarti Gupta, and Nadia Papakonstantinou. 2013. SETSUDŌ: perturbation-based testing framework for scalable distributed systems. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems, TRIOS@SOSP 2013, Farmington, PA, USA, November 3, 2013.* ACM, 7:1–7:14. doi:10.1145/2524211.2524217

[33] James Lewis and Martin Fowler. 2014. Microservices: a definition of this new architectural term. Retrieved December, 2025 from https://martinfowler.com/articles/microservices.html

[34] Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. 2017. DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, Yunji Chen, Olivier Temam, and John Carter (Eds.). ACM, 677–691. doi:10.1145/3037697.3037735

[35] Jiangchao Liu, Jierui Liu, Peng Di, Alex X. Liu, and Zexin Zhong. 2022. Record and Replay of Online Traffic for Microservices with Automatic Mocking Point Identification. In *44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022, Pittsburgh, PA, USA, May 22-24, 2022.* IEEE, 221–230. doi:10.1109/ICSE-SEIP55303.2022.9793867

[36] Zhenyue Long, Guoquan Wu, Xiaojiang Chen, Wei Chen, and Jun Wei. 2020. WebRR: self-replay enhanced robust record/replay for web application testing. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1498–1508. doi:10.1145/3368089.3417069

[37] Zhenyue Long, Guoquan Wu, Xiaojiang Chen, Chengxu Cui, Wei Chen, and Jun Wei. 2020. Fitness-guided Resilience Testing of Microservice-based Applications. In *2020 IEEE International Conference on Web Services, ICWS 2020, Beijing, China, October 19-23, 2020.* IEEE, 151–158. doi:10.1109/ICWS49710.2020.00027

[38] Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. 2019. CrashTuner: detecting crash-recovery bugs in cloud systems via meta-info analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 114–130. doi:10.1145/3341301.3359645

[39] Sehrish Malik, Moeen Ali Naqvi, and Leon Moonen. 2023. CHESS: A Framework for Evaluation of Self-Adaptive Systems Based on Chaos Engineering. In *18th IEEE/ACM Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2023, Melbourne, Australia, May 15-16, 2023.* IEEE, 195–201. doi:10.1109/SEAMS59076.2023.00033

[40] Christopher S. Meiklejohn, Andrea Estrada, Yiwen Song, Heather Miller, and Rohan Padhye. 2021. Service-Level Fault Injection Testing. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, Carlo Curino, Georgia Koutrika, and Ravi Netravali (Eds.). ACM, 388–402. doi:10.1145/3472883.3487005

[41] James W. Mickens, Jeremy Elson, and Jon Howell. 2010. Mugshot: Deterministic Capture and Replay for JavaScript Applications. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, CA, USA.* USENIX Association, 159–174. http://www.usenix.org/events/nsdi10/tech/full_papers/mickens-mugshot.pdf

[42] Sam Newman. 2015. *Building microservices - designing fine-grained systems, 1st Edition.* O'Reilly. https://www.worldcat.org/oclc/904463848

[43] OpenTelemetry. 2025. *High-quality, ubiquitous, and portable telemetry to enable effective observability.* Retrieved December, 2025 from https://opentelemetry.io/

[44] Harendra Singh, Laxman Singh, and Shailesh Tiwari. 2022. A Systematic Literature Review on Test Case Prioritization Techniques. *Int. J. Softw. Innov.* 10, 1 (2022), 1–36. doi:10.4018/IJSI.312263

[45] Davide Spadini, Mauricio Finavaro Aniche, Magiel Bruntink, and Alberto Bacchelli. 2017. To mock or not to mock?: an empirical study on mocking practices. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, Jesús M. González-Barahona, Abram Hindle, and Lin Tan (Eds.). IEEE Computer Society, 402–412. doi:10.1109/MSR.2017.61

[46] Davide Spadini, Maurício Finavaro Aniche, Magiel Bruntink, and Alberto Bacchelli. 2019. Mock objects for testing java systems - Why and how developers use them, and how they evolve. *Empir. Softw. Eng.* 24, 3 (2019), 1461–1498. doi:10.1007/S10664-018-9663-0

[47] Haley Tucker, Lorin Hochstein, Nora Jones, Ali Basiri, and Casey Rosenthal. 2018. The Business Case for Chaos Engineering. *IEEE Cloud Comput.* 5, 3 (2018), 45–54. doi:10.1109/MCC.2018.032591616

[48] Zhigang Wang, Lixin Gao, Yu Gu, Yubin Bao, and Ge Yu. 2018. A Fault-Tolerant Framework for Asynchronous Iterative Computations in Cloud Environments. *IEEE Trans. Parallel Distributed Syst.* 29, 8 (2018), 1678–1692. doi:10.1109/TPDS.2018.2808519

[49] Muhammad Waseem, Peng Liang, Mojtaba Shahin, Amleto Di Salle, and Gastón Márquez. 2021. Design, monitoring, and testing of microservices systems: The practitioners' perspective. *J. Syst. Softw.* 182 (2021), 111061. doi:10.1016/J.JSS.2021.111061

[50] Weaveworks. 2016. SockShop: A microservice demo application. Retrieved July, 2025 from https://github.com/microservices-demo/microservices-demo

[51] Thomas Welsh and Elhadj Benkhelifa. 2021. On Resilience in Cloud Computing: A Survey of Techniques across the Cloud Domain. *ACM Comput. Surv.* 53, 3 (2021), 59:1–59:36. doi:10.1145/3388922

[52] Huayao Wu, Senyao Yu, Xintao Niu, Changhai Nie, Yu Pei, Qiang He, and Yun Yang. 2023. Enhancing Fault Injection Testing of Service Systems via Fault-Tolerance Bottleneck. *IEEE Trans. Software Eng.* 49, 8 (2023), 4097–4114. doi:10.1109/TSE.2023.3285357

[53] Qinghua Xu, Guancheng Wang, Lionel C. Briand, and Kui Liu. 2025. Hallucination to Consensus: Multi-Agent LLMs for End-to-End Test Generation with Accurate Oracles. *CoRR* abs/2506.02943 (2025). arXiv:2506.02943 doi:10.48550/ARXIV.2506.02943

[54] Tianyi Yang, Cheryl Lee, Jiacheng Shen, Yuxin Su, Cong Feng, Yongqiang Yang, and Michael R. Lyu. 2024. MicroRes: Versatile Resilience Profiling in Microservices via Degradation Dissemination Indexing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 325–337. doi:10.1145/3650212.3652131

[55] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay Jain, and Michael Stumm. 2015. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. *login Usenix Mag.* 40, 1 (2015). https://www.usenix.org/publications/login/feb15/yuan

[56] Long Zhang, Brice Morin, Philipp Haller, Benoit Baudry, and Martin Monperrus. 2021. A Chaos Engineering System for Live Analysis and Falsification of Exception-Handling in the JVM. *IEEE Trans. Software Eng.* 47, 11 (2021), 2534–2548. doi:10.1109/TSE.2019.2954871

[57] Man Zhang, Andrea Arcuri, Yonggang Li, Yang Liu, and Kaiming Xue. 2023. White-Box Fuzzing RPC-Based APIs with EvoMaster: An Industrial Case Study. *ACM Trans. Softw. Eng. Methodol.* 32, 5 (2023), 122:1–122:38. doi:10.1145/3585009

[58] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2021. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Trans. Software Eng.* 47, 2 (2021), 243–260. doi:10.1109/TSE.2018.2887384

[59] Hengcheng Zhu, Lili Wei, Ming Wen, Yepang Liu, Shing-Chi Cheung, Qin Sheng, and Cui Zhou. 2020. MockSniffer: Characterizing and Recommending Mocking Decisions for Unit Tests. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020.* IEEE, 436–447. doi:10.1145/3324884.3416539