Automated Network-Level Fault Injection Testing of Microservice Architectures

Delano Flipse
Delft University of Technology
Delft, Netherlands
d.e.flipse@student.tudelft.nl

Hakan Simsek
ASML
Veldhoven, Netherlands
hakan.simsek@asml.com

Jeremie Decouchant
Delft University of Technology
Delft, Netherlands
j.decouchant@tudelft.nl

Burcu Kulahcioglu Ozkan
Delft University of Technology
Delft, Netherlands
b.ozkan@tudelft.nl

Abstract

Microservice architectures are inherently vulnerable to partial failures, and they rely on resilience patterns to tolerate them. However, it is challenging to design and implement the resilience logic. Unforeseen interactions with faulty services can lead to errors in dependent services, resulting in incorrect system behavior. Fault injection testing techniques aim to uncover these errors by examining the system's behavior in response to different fault combinations. However, existing automated techniques primarily focus on service-level fault injection, which limits their broader applicability.

We present an automated fault-injection testing method that operates at the network level, enabling broad applicability. The method models the system's resilience behaviors dynamically through the observed test executions and uses this information to reduce the set of fault combinations to explore. We implemented our method in a prototype tool called Reynard. Our evaluation demonstrates that Reynard efficiently explores system executions by significantly reducing the number of fault combinations to test. It incurs minimal overhead and can be easily integrated into existing benchmarks. Furthermore, we applied Reynard to test an industrial system, showcasing its applicability and effectiveness in a real-world context. Moreover, we uncovered a previously unknown resilience bug.

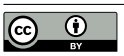
CCS Concepts

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

Keywords

microservice architectures, automated testing, fault injection

ACM Reference Format:
Delano Flipse, Hakan Simsek, Jeremie Decouchant, and Burcu Kulahcioglu Ozkan. 2026. Automated Network-Level Fault Injection Testing of Microservice Architectures. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3744916.3773193



ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3773193

1 Introduction

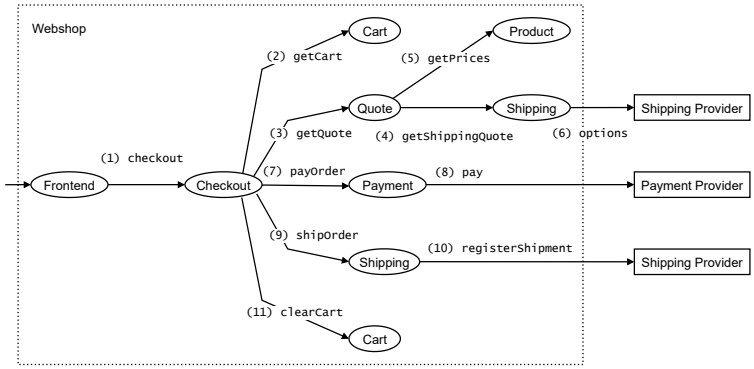
Modern software systems are increasingly adopting a *microservice architecture* to meet scalability, efficiency, and reliability demands. Microservice architectures decompose the system's functionality into distinct, decoupled domains, with each microservice handling the functionality specific to its domain. This approach allows microservices to be developed and deployed independently from each other, thereby enhancing the overall software development process. However, microservice applications are also notoriously challenging to design and implement correctly. One significant hurdle is the occurrence of partial failures [33], where some services fail, while others continue to operate. Such failures can lead to unforeseen and erroneous behavior in dependent microservices, threatening the reliability of the overall system [14, 18, 23, 34, 36]. It is essential to maintain resilience against partial failures to ensure that the system's correctness properties are preserved in their presence.

Fault injection testing is a well-known approach to test the resilience of distributed systems. For instance, with chaos engineering [3–5, 31], random faults are injected into the execution of a system to assess its resilience in production. Other techniques [2, 16, 26, 35] test the system by injecting faults in a controlled environment. Recent work in fault injection testing for distributed systems applies instrumentation within each service to inject specific partial failures [2, 16, 26, 35]. Systematically exposing the system to combinations of faults and checking its behavior in their presence increases confidence in the system's resilience. However, the number of fault combinations grows exponentially with the size of the system. This makes exhaustive testing intractable for most systems. Various exploration techniques address this challenge by reducing or prioritizing the fault combinations to explore [2, 20, 26]. While state-of-the-art automated fault injection testing provides some solutions to verify a system's behavior in a reasonable number of test executions, it does not offer practical solutions for broad applications. Applying existing techniques to a particular microservice system requires adding custom instrumentation to each individual microservice or providing system-specific information to tackle the fault combinations space. Moreover, existing work fails to address the polyglot nature of microservices and requires significant implementation effort.

In this work, we address this gap by proposing *a practical approach to automated fault injection testing of microservice applications* that leverages *network-level instrumentation*. Our approach

```python
@app.post("/checkout")
def performCheckout(uid, adress, payment):
  try:
    products = Cart.getCart(uid) # (2)
    price = Quote.getQuote(prdcts, adress) # (3)
    paymentId = Payment.payOrder(price, payment) # (7)
    shipmentId = Shipping.shipOrder(products, adress) # (9)
    Cart.clearCart(uid) # (11)
  except GetCartError e:
    throw TransientInternalServiceError("Please retry")
  except QuoteError e:
    throw TransientInternalServiceError("Please retry")
  except PaymentError e:
    Alerting.alert("Payment failed!") # (7.1)
    throw TransientInternalServiceError("Please retry")
  except ShipmentError e:
    Payments.RevertPayment(paymentId) # (9.1)
    Alerting.alert("Shipment failed!") # (9.2)
    throw InternalServiceError("Shipment failed")
  except ClearCartError e:
    throw InternalServiceError("Failed to clear cart")
```

(a) Pseudocode of the checkout service



(b) Call graph of the checkout endpoint

Figure 1: Checkout service from the example webshop system

is based on the observation that it takes significantly less effort to adjust the network configuration of a microservice application [17] than to adapt the source code of each individual microservice. However, as the network instrumentation operates externally to the services, it cannot access their runtime information. In contrast, the state-of-the-art test strategies employing *service-level* instrumentation utilize this information to reduce the number of fault combinations to explore. This information is not directly available at the network level.

We address these limitations by employing two techniques. First, we utilize the context propagation mechanism offered by distributed tracing solutions to obtain metadata related to causality. This causal relation provides essential information and helps reduce the blast radius of faults by allowing precise fault injection [6, 35]. State-of-the-art distributed tracing tools, such as OpenTelemetry [29], enable instrumentation without the need for source code modifications and are widely compatible with commonly used programming languages and frameworks. This makes it feasible to integrate distributed tracing into existing microservice applications. Second, we use observations from our instrumentation to dynamically infer the system's resilience patterns and how faults affect the system's behavior. Based on this model of the system's behaviour, we reduce the fault space by combining existing and novel reduction techniques. Our network-based instrumentation, in combination with a dynamic model of the system interactions, reduces the set of fault combinations to explore while being practical to apply.

Overall, we make the following contributions:

- We design a network-level fault injection method for automated resilience testing of microservice applications. It is practical to apply due to its network-level injection and is effective in reducing the search space of fault combinations.
- We define an abstract model of the expected system behavior in response to service faults, using dynamic observations.
- We design and implement Reynard, an automated network-level fault injection testing tool, and evaluate it by comparing it to state-of-the-art service-level fault injection. Reynard is publicly available [11].
- We apply Reynard to an industrial system and demonstrate its applicability to real-world microservice systems as well as uncover a previously unknown resilience bug in the system.

2 Motivation and Overview

We motivate and provide an overview of our fault injection testing method using an example microservice application that will serve as a running example throughout the paper.

Example microservice application. Consider a webshop system that includes a checkout service (Figure 1a). For the checkout, it first retrieves the list of products in the user's cart by sending a request to the *Cart* service (L4). Based on products in the cart, it determines the price by sending a request to *Quote* (L5). If either of these requests fails, the *Frontend* can retry in an attempt to resolve the fault (L10,12). If both succeed, it makes the payment on the user's credit card (L6). If this fails, the operation can be retried (L15). Additionally, an incident alert is created (L14). Once the payment is processed, the products are registered for shipment (L7). If this fails, the system attempts to revert the payment (L17) and creates an alert (L18). After all steps succeed, the cart is cleared (L8).

Figure 1b illustrates the (distributed) call graph of the checkout procedure. The interaction with the checkout service results in 8 indirect calls to other internal microservices and 2 external services. The semantics of the checkout service include resilience mechanisms (e.g, the retry logic, alerting, and payment restoration) that are not observable from its execution without failures, also known as *the happy path*. These are only observable if specific (combinations of) faults are present. For example, requests to the alerting service are only performed in the presence of faults. The resilience mechanisms aim to ensure correct functioning; however, they can

be implemented incorrectly. For example, the pseudocode contains a latent bug: What happens if the reversal of the payment fails? If the frontend naively performs a retry on a payment reversal failure, it may result in creating two payments without a shipment.

Overview of Reynard. Reynard uses network-level instrumentation and distributed tracing (e.g., OpenTelemetry [29]) to inject precise faults and capture the system's behavior. Based on the system's behavior, it dynamically generates new test cases with relevant combinations of faults, while avoiding redundant combinations. For the example system in Figure 1b, Reynard creates test cases for each relevant combination of faults. It detects the resilience mechanism where the payment is reversed due to a failed request to the *Shipping* service. It will then generate a test case with an additional fault in the request to reverse the payment. However, it will not generate a test case with an additional fault in the request to clear the cart, as these faults cannot be combined and are therefore redundant.

3 Reynard Design

Figure 2 shows the high-level design of our testing approach. We instrument the system under test (SUT) to collect information about the service interactions and to inject faults, and design an *automated test strategy* that generates test cases based on the collected information. The automated test strategy iteratively follows the steps shown in the figure: It (1) generates a *test scenario* by deciding on a set of faults to inject, starting with a fault-free execution (2) instructs the network-level instrumentation to inject the selected faults when observing the corresponding requests (3) executes the test scenario, which interacts with the SUT, and (4) analyses the test execution to check its correctness and collect information about possible fault injection points for the generation of new test cases. The captured service interactions in the test executions guide the automated test strategy's test generation. This iterative testing process continues until all relevant combinations of faults are exercised.

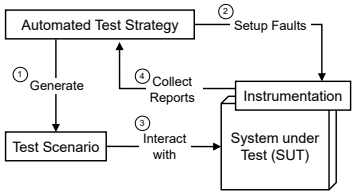


Figure 2: High-level design of Reynard, our testing approach

3.1 Network-Level Fault Injection

Figure 3 illustrates our network-level instrumentation (highlighted in green) added to an existing system. The instrumentation consists of a controller service and a proxy per service of interest. The controller service tracks the state of the service interactions and serves as an API for the test strategy. This API enables sending fault injection instructions to the proxies and receiving reports on the captured service requests and responses.

Figure 4 illustrates the fault injection process. The instrumentation intercepts requests to the target service (Figure 4a). We derive

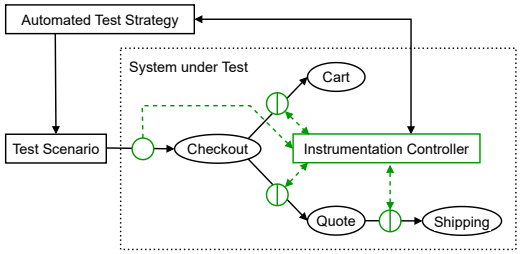
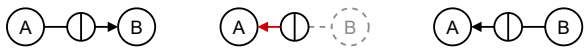


Figure 3: Network-level instrumentation can inspect and modify message exchanges between services

an identifier from the request's metadata. This identifier is used to decide whether to inject a fault in the response. A fault is injected by responding with an erroneous response, and we omit sending the real request (Figure 4b). The erroneous response is a message containing a status code (the failure mode), which represents a specific type of service failure, as specified in the test scenario. If no fault should be injected, the request and response of the related service are proxied (Figure 4c).



(a) A request **(b) Fault injection** **(c) Normal response**

Figure 4: Fault injection applied by the instrumentation.

3.2 Modeling of the Faults

Our instrumentation allows for injecting faults into service requests (i.e., a fault injection point). To distinguish them, we identify each request based on a distributed execution index defined in [27], adjusted for the information available to network-level instrumentation. A request identifier is a stack of tuples ⟨*Destination*, *Interface*, *Payload*, *Predecessors*, *Count*⟩, where *Destination* is the name of the receiving service, *Interface* is the requested service endpoint, *Payload* is (a hash of) the arguments given to the interface, *Predecessors* is an (optional) field representing the temporally preceding requests originating from the same service, and *Count* is the number of occurrences of the identifier in the interaction. Each tuple in the stack represents an interaction between two services, where the top tuple corresponds to the request that can be perturbed, while the rest of the stack defines its causal history of service calls. We keep the *Predecessors* field optional, as it makes the identification more precise when necessary, yet more costly. Moreover, we use wildcards for fields in the tuple to *query* fault injection points.

Example 3.1. In the call graph of Figure 1b, the request (5) from the Quote service to the Product service is identified as:

$$\begin{bmatrix} \langle \text{Checkout}, & /\text{checkout}, & \#, & \{\}, & 0\rangle, \\ \langle \text{Quote}, & /\text{getQuote}, & \#, & \{\text{Cart/getCart} : 0\}, & 0\rangle, \\ \langle \text{Product}, & /\text{getPrices}, & \#, & \{\}, & 0\rangle \end{bmatrix}$$

For deterministic identification of the fault injection points across executions, we assume that the executions of the service calls are deterministic, following the previous work [2, 26], Therefore, we

assume that the system reaches the same execution state, given that it receives and processes the same sequence of responses [26, 35]. Furthermore, we assume that the behavior of services is encapsulated [26]. That is, we assume that the response of a request to a service is solely dependent on its direct downstream responses.

Given the set of fault injection points P and failure modes M:

Definition 3.2. A **fault** is a pair of a fault injection point $p \in P$ (the when and where) and a failure mode $m \in M$ (the how), written (p, m) or f_p^m. The set of possible faults is $P \times M$. We denote a fault at point $p \in P$, regardless of its mode, as f_p.

Definition 3.3. A (local) **behavior** is a fault injection point and either a failure mode, or no failure mode ($\perp$), denoted as b_p^m.

Without a failure mode, the behavior of a fault injection point represents the happy path behavior, denoted as $b_p^\perp$. Whereas a fault defines an intention to inject a failure into a request, a behavior defines the corresponding observed response. A behavior with a failure mode can be caused by a fault or by the propagation of faults. We refer to a behavior with a failure mode as a *fault behavior*.

Definition 3.4. A **faultload** $F = \{f_{p_1}^{m_a}, f_{p_2}^{m_b}, \ldots, f_{p_n}^{m_z}\}$ is a unordered set of faults.

A fault injection point can only be represented once in a faultload, implying that a fault injection point can exhibit at most one failure mode at a time. We denote the set of fault injection points contained in the faultload as $P(F) = \{p_1, p_2, \ldots, p_n\}$.

Definition 3.5. The **fault space** $\mathcal{F}$ represents all possible faultloads: $\forall F, F \in \mathcal{F}$.

As a faultload includes at most a single fault for a given fault injection point, the fault space is defined by how each set of points in the powerset of P can be combined with the failure modes M. Therefore, the size of the complete fault space is given by:

$$|\mathcal{F}| = \sum_{k=0}^{|P|} \binom{|P|}{k} |M|^k = (1 + |M|)^{|P|} \tag{1}$$

Given $|P| = n$ and $|M| = m$, exhaustively covering the fault space requires exercising $O(n, m) = (1+m)^n$ number of faultloads. Hence, the fault space is exponential in the number of fault injection points, $|P|$, and polynomial in the number of failure modes, $|M|$.

3.3 Representation of the Fault Space

Given the exponential size of the fault space, enumerating all possible faultloads is intractable. Instead, we represent the fault space as a search tree, which is built lazily during exploration. Moreover, it allows for the effective pruning of redundant faultloads.

Figure 5 illustrates the search tree of the exploration of the call graph in Figure 3, for 4 failure modes. The root node of the search tree represents the empty faultload, corresponding to the happy path. Each level of the tree adds one fault to the faultload. Hence, the nodes represent the set of points defined by the union of the node and its predecessors. In the figure, we merge faultloads with the same fault injection points into the same node. Labels on edges in the figure indicate the number of representative faultloads. The number of faultloads for n fault injection points and m failure modes is equal to m^n.

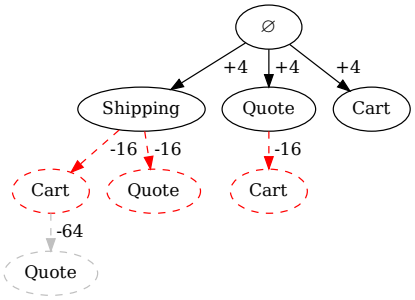


Figure 5: Search tree for the call graph in Figure 3.

The nodes in the search tree can represent redundant faultloads, as will be defined later. If we find a node to be redundant (the nodes in red color in the figure), we omit evaluating its subtree (the nodes in gray color), effectively pruning any superset of the corresponding faultload. Additionally, we can dynamically add new nodes to the search tree. By injecting faults, we can uncover resilience patterns and their related fault injection points, which are not known in advance and are added to the tree dynamically during the exploration.

3.4 Exploration of the Fault Space

Algorithm 1 explores the fault space given an executable test scenario, a correctness property, the failure modes to inject, and a list of pruning policies to reduce the search tree, as inputs. First, we define a queue of nodes to visit, starting with the root node, and an empty set of visited nodes (L2). While there are nodes in the tree to explore (L3), we take the first node from the queue (L4) and add it to the visited nodes (L5). We check, given the currently available information, if the node is redundant according to a pruning policy (L6). If it is redundant, we omit exploring it and do not consider its subtree (L7). Otherwise, we run the test scenario for the representative faultload and collect the reports from the instrumentation (L8). This information is fed back to the pruning policies to identify redundancies (L9). Then, we check the correctness property in the observed interaction (L10). If it does not hold, we return the counterexample (L17). Finally, we determine the faults that can be added to the faultload and add all non-redundant child nodes to the queue (L11-15). This process is repeated until all non-redundant faultloads in the search tree are explored.

The algorithm explores the search tree in a breadth-first manner, which is beneficial for our pruning policies. As we prune the fault space based on dynamic observations, the breadth-first exploration ensures we first observe the effect of individual faults before exploring their composition. The order in which we select fault injection points for exploration can influence efficiency, as it determines the order in which information becomes available. As faults can be propagated up the call graph, we prefer to explore deeper fault injection points first. Therefore, we order fault injection points by a depth-first traversal of the *call graph*, sorting sibling events in the temporal order in which the proxies recorded them.

Algorithm 1 Exploration algorithm.

Require: Test scenario T, Correctness property C, Failure modes M, Pruning policies P_s

```
1:  procedure FAULT-SPACE-EXPLORATION
2:      N_queue : List[F] ← [∅], N_visited : Set[F] ← {}
3:      while |N_queue| > 0 do
4:          F : F ← DEQUEUE(N_queue)
5:          N_visited ← N_visited ∪ {F}
6:          if SHOULD-PRUNE(P_s, F) then
7:              continue
8:          Reports ← EXECUTE-SCENARIO-WITH-FAULTS(T, F)
9:          ANALYZE(P_s, F, Reports)
10:         if HOLDS(C, Reports) then
11:             P_reachable ← P(TraceTree) \ P(F)
12:             for f_p^m ∈ P_reachable × M do
13:                 N_child ← F ∪ {f_p^m}
14:                 if N_child ∉ N_queue ∧ N_child ∉ N_visited then
15:                     N_queue ← N_queue || N_child
16:         else
17:             return F
18:     return Nil
```

3.5 Fault Space Pruning Policies

To reduce the fault space, we employ three pruning policies. The first policy determines the combinations of faults that cannot be combined using a model of the system. This model is dynamically constructed using observations of the test executions. The second policy prunes faultloads whose expected behavior is dominated by previously observed test executions, based on previous work [26]. Finally, the third policy reduces the fault space required to cover the behavior of retries, based on insights from previous work [8].

3.5.1 Pruning Infeasible Fault Combinations. Some combinations of faults are infeasible to combine, as the corresponding fault injection points are not guaranteed to occur in an execution. For example, some faults can cause the system to stop its normal execution and omit further service requests. Alternatively, it can attempt to resolve faults by introducing additional requests.

However, the semantics of the service behaviors are unavailable to the test faultload generation. To infer this information from the test executions, we observe the *effect* of faults in those executions and use that information to build a model of system behaviors.

We model a system's behavior using four main abstractions that capture the propagation of requests and responses through the service call graph. Specifically, requests may trigger downstream requests, responses can impact other requests made by the same originating service, and responses can generate upstream responses. In examples, we will refer to the fault injection points in Figure 1 by the number indicated on the corresponding edges or in the comments. E.g., we refer to the *getCart* request as p_2.

Downstream dependencies. An injected fault omits the request to a service, preventing it from calling its downstream dependencies. Therefore, if a fault is injected in a request, all causally dependent downstream requests do not appear in the execution. We obtain this information in the system's model from the causal dependencies between downstream requests in the tracing metadata and denote this relation as $p_a \xrightarrow{hb} p_b$, where p_b is a downstream dependency of p_a.

Example 3.6. In Figure 1b, p_4 is a downstream dependency of p_3, and so $F = \{f_{p_3}, f_{p_4}\}$ is infeasible.

Exclusions. For sibling requests in the call graph (requests originating from the same service), we can not trivially determine causal dependencies. Instead, we observe the effect of faults on other related fault injection points. If faults cause the exclusion of an expected fault injection point, we store the faults as an exclusion condition for the missing point. If an exclusion condition is met, we assume that the fault injection point will not occur.

Example 3.7. In Figure 1b, p_2 must succeed before the checkout service can continue, hence $\{f_{p_2}\}$ excludes all subsequent requests. Therefore, $F = \{f_{p_2}, f_{p_3}\}$ is infeasible.

Inclusions. The execution of the system in the presence of particular faults can create additional fault injection points that are not contained in the happy path. If we observe an unexpected fault injection point in the call graph of a test execution, we store the related faults as an inclusion condition for that new fault injection point.

Example 3.8. The request to revert the payment ($p_{9.1}$) only occurs if the shipment fails. Hence $\{f_{p_9}\}$ is a inclusion condition for $f_{p_{9.1}}$.

Upstream fault propagation. For some downstream faults, a service may respond with a fault behavior. In that case, a fault is propagated upstream. If a fault behavior is observed, we store all direct downstream behaviors as a condition for the related upstream behavior.

Example 3.9. If the request to the payment provider (p_8) fails, this will result in a fault behavior in the response at p_7.

The observed information about the downstream dependencies, exclusions, inclusions, and upstream fault propagation, allow us to model the system behavior. Algorithm 2 presents the method for modeling the system based on observations from previous test executions. Starting from the initial (root) request (L1), we recursively reason about the expected downstream (↓) requests and responses to determine the root behavior and all downstream behaviors (L2). This results in the expected behaviors in the system (L3).

For each request (L4), we first check if a fault will be injected (L5). If so, we return the behavior of that fault (L6). Otherwise, we determine the response to the request. First, we look up the direct downstream requests in the happy path (L7). If there are no downstream requests (L8), we have reached a leaf node in the call graph and respond with the expected happy path behavior (L9). If there are downstream requests in the happy path, we determine which sibling downstream requests will and will not occur, and their behavior (L10-11). We determine both the direct downstream behaviors (L12) and *all* downstream behaviors (L13). To determine the upstream (↑) behavior of the fault injection point, we check if we have observed a propagated fault for the direct downstream behaviors (L15), or by default assume the happy path (L14). Ultimately, we return both the expected behavior of the fault injection point and all its downstream behaviors (L17). To determine the behavior of sibling downstream requests (L18), we must consider all known

Algorithm 2 Expected behavior modeling for a given set of faults.

Require: Downstream requests in the happy path,
stored per point $D_{req} : P \mapsto Set[P]$
Require: Exclusion conditions,
stored per point $C_{excl} : P \mapsto List[Set[\mathcal{B}]]$
Require: Inclusion conditions,
stored per point $C_{incl} : P \mapsto List[Set[\mathcal{B}]]$
Require: Upstream fault propagation,
stored by direct downstream responses $U_{res} : Set[\mathcal{B}] \mapsto \mathcal{B}$

1: **procedure** EXPECTED-BEHAVIOR-FOR(p_{root}, F)
2: $(b^m_{p_{root}}, B_\downarrow) \leftarrow$ UNFOLD-BEHAVIOR(p_{root}, F)
3: **return** $\{b^m_{p_{root}}\} \cup B_\downarrow$
4: **procedure** UNFOLD-BEHAVIOR(p, F)
5: **if** $f^m_p \in F$ **then**
6: **return** $(b^m_p, \emptyset)$
7: $P_\downarrow : Set[P] \leftarrow D_{req}[p]$
8: **if** $P_\downarrow = \emptyset$ **then**
9: **return** $(b^\perp_p, \emptyset)$
10: $B_\downarrow : (\mathcal{B} \mapsto Set[\mathcal{B}])$
11: $B_\downarrow \leftarrow$ GET-INCLUDED-BEHAVIOR($p, P_\downarrow, F$)
12: $B_{direct} : Set[\mathcal{B}] \leftarrow \bigcup_{u \in B_\downarrow} \{u\}$
13: $B_{transitive} : Set[\mathcal{B}] \leftarrow B_{direct} \cup \bigcup_{u \in B_\downarrow} B_\downarrow[u]$
14: $b_\uparrow \leftarrow b^\perp_p$
15: **if** $B_{direct} \in U_{res}$ **then**
16: $b_\uparrow \leftarrow U_{res}[B_{direct}]$
17: **return** $(b_\uparrow, B_{transitive})$
18: **procedure** GET-INCLUDED-BEHAVIOR($p_\uparrow, P_\downarrow, F$)
19: $B_\downarrow : (\mathcal{B} \mapsto Set[\mathcal{B}])$
20: $P_{rel} : P[\,] \leftarrow$ TOPOLOGICAL-SORT($p_\uparrow, P_\downarrow, C_{incl}, C_{excl}$))
21: **for** $p \in P_{rel}$ **do**
22: **if** SHOULD-INCLUDE($p, B_\downarrow, p \in P_\downarrow, C_{excl}, C_{incl}$) **then**
23: $(b^{m_x}_p, B_{downstream}) \leftarrow$ UNFOLD-BEHAVIOR(p, F)
24: $B_\downarrow[b^{m_x}_p] \mapsto B_{downstream}$
25: **return** $B_\downarrow$

downstream fault injection points. As fault injection points can be conditionally excluded and included, we start by topologically sorting all known downstream sibling points, based on their related inclusion and exclusion conditions (L20). For each possible fault injection point (L21), we determine whether it is expected to be included (L22). A fault injection point is expected to be included if the most complex valid inclusion condition is of equal or larger complexity than the most complex valid exclusion condition. The complexity is defined as the number of behaviors in its set of conditional behaviors. If a fault injection point is part of the happy path, it is considered an inclusion condition with complexity 0. If a point should be included, we recursively evaluate its downstream behavior (L23-24).

The model allows us to infer the expected behaviors for a faultload: $\tilde{B}(F) = \{b^{m_x}_{p_1}, ..., b^{m_z}_{p_n}\}$. Formally, a faultload F is redundant if $P(F) \supset P(\tilde{B}(F))$. In other words, a faultload is redundant if it attempts to inject faults at fault injection points that will not occur due to the effects of other faults.

3.5.2 Service Encapsulation Reduction Policy. The work by Meiklejohn et al. [26] proposes a pruning algorithm and exploration strategy based on the premise of *service encapsulation*. Using this reduction policy, we prune any combinations of faults that do not result in any new behavior, compared to what was already observed. The reduction algorithm is as follows: Given a faultload F, its expected behavior $\tilde{B}(F)$, and previously observed behaviors $H_B = [B_1, B_2, ..., B_n]$ from earlier test executions, F is redundant if: for every behavior $(b^{m_\uparrow}_{p_\uparrow} \in \tilde{B}(F))$ and its direct downstream dependencies $B_\downarrow = \{b^{m_\downarrow}_{p_\downarrow} \in \tilde{B}(F) \mid p_\uparrow \xrightarrow{hb} p_\downarrow\}$, there exists a previously observed execution $B_h \in H_B$ where all dependencies had the same observed behavior ($B_\downarrow \subseteq B_h$). Formally, a faultload F is redundant if $\forall b^{m_\uparrow}_{p_\uparrow} \in \tilde{B}(F),\ \exists B_h \in H_B,\ \{b^{m_\downarrow}_{p_\downarrow} \in \tilde{B}(F) \mid p_\uparrow \xrightarrow{hb} p_\downarrow\} \subseteq B_h$. In other words, a faultload is redundant if for each effect (the upstream behavior), we have already observed its expected causes (the downstream behaviors, if any). In that case, the faultload cannot provide any new information about the system, due to the service encapsulation property.

Example 3.10. Having observed the effect of faultload $\{f_{p_8}\}$, specifically the fault propagation of $b^m_{p_7}$, exploring faultload $\{f^m_{p_7}\}$ does not yield any new information about the system.

3.5.3 Retry Reduction Policy. Retries are a common resilience pattern in microservice applications [32]. However, each retry is a new fault injection point that grows the fault space. Chen et al. [8] found that most correctness property violations in retries can be found by testing for only transient or persistent faults. Retries are generally performed for specific failure modes and have specific error handlers for other failure modes. Hence, the system's behavior for retries is dependent on either reaching the retry limit or observing a specific non-retryable failure. We reduce the fault space by testing solely for a single transient fault or a persistent fault with the same failure mode. We omit testing for combinations of retries with different failure modes. This reduces the subspace for a retried event from $\sum_{i=0}^{n} m^i$, where n is the number of retries and $m = |M|$, to just 2. We detect retries by observing the inclusion of a fault injection point that is equal to a fault behavior, with a single increase in the count field. The persistent fault is represented as the same identifier with a wildcard in the count field. An additional benefit is that we can immediately uncover the retry limit. When we discover the retry mechanism, it is impossible to know how many retries will be issued. By instructing the proxies to inject faults on all subsequent retries, n faults will injected, instead of scheduling n faultloads. This pruning policy is not strictly sound, as it makes an assumption on the implementation of retries. Therefore, it is not enabled by default.

3.6 Automatic Anti-Pattern Detection

Aside from validating user-defined correctness properties, we check for the following two anti-patterns:

Faults without intentional cause. Due to fault propagation, we can expect to see fault behaviors that are caused by injected faults. However, if we observe fault behaviors without a causal relation to injected faults, this is likely an indication of a bug.

Semantically incorrect fault propagation. An erroneous pattern we observed propagates faults *including their status code*. This can lead to unintended upstream behavior due to the semantics

Table 1: Failure types and corresponding status codes per protocol included in Reynard

Failure type	HTTP	gRPC
None	2xx	1
Crash	500, 502	2, 13
Request-omission	503	14
Timing	504	4

of status codes. For example, a 503 HTTP status code ("Service unavailable") implies the downstream service has not received a request. Hence, it can be safe to attempt a retry even if the endpoint is non-idempotent. This behavior is part of some HTTP clients. Consequently, returning a 503 status code in response to a failure of a downstream dependency is semantically incorrect, as the service was available and processed the request.

4 Implementation

We implemented Reynard, our network-level fault injection testing method, and released it publicly [11]. The automated test strategy is implemented as a Java JUnit 5 plugin. Reynard applies network-level instrumentation by implementing layer-7 reverse proxies in Go, which must be placed before each service of interest. The instrumentation intercepts and inspects requests, using the body and headers to assign an identifier to each request. To infer causal relationships between service requests, Reynard requires a distributed tracing solution that implements the W3C trace context headers [21], specifically the *traceparent* and *tracestate* headers. We employ OpenTelemetry [29], which offers automated instrumentation for six commonly used programming languages and provides stable support for four additional languages, thereby enabling broad applicability. Table 1 details the failure types supported by Reynard, along with the corresponding status codes per protocol. The proxies support two protocols: HTTP and gRPC.

In our definition of fault injection points, we assume that the SUT is deterministic. However, realistic microservice applications can exhibit scheduling and data nondeterminism. To address this, we allow a configuration in which the *Payload* field is ignored in the identifier. Note that omitting this field can result in distinct requests having the same identifier. In cases where this occurs, including the *Predecessor* field can help distinguish these requests when there is no scheduling nondeterminism. Distinguishing requests and assigning each distinct call a unique, consistent identifier enables effective exploration of the fault space.

5 Evaluation

We evaluated Reynard on a set of open source benchmarks and with an industrial case study, measured its ability to *reduce the exploration space* and its *runtime efficiency*, and compared it to existing work. We answer the following research questions:

- **RQ1. Comparison:** How does Reynard compare to the state-of-the-art service-level fault injection testing?
- **RQ2. Effectiveness:** How effective is Reynard at reducing the exploration space of test executions?
- **RQ3. Efficiency:** What is the runtime overhead introduced by the network-level instrumentation of Reynard?
- **RQ4. Applicability:** Is Reynard applicable to real-world industrial systems?

We compare Reynard to the state-of-the-art service-level fault injection tool, Filibuster [26], using its suite of microbenchmarks (**RQ1**). We evaluate Reynard on several benchmark systems (**RQ2**) and by using a synthetic benchmark (**RQ3**) to measure the overhead of our instrumentation with a performance test. Finally, we apply Reynard to a case study on an industrial system (**RQ4**).

5.1 Experimental Setup

We evaluated Reynard using the following benchmark systems:

- The *Filibuster Corpus* of benchmarks [26] is a suite of Python-based (micro)benchmark microservice applications.
- The OpenTelemetry *Astronomy Shop* is a demonstration microservice application that highlights the use of OpenTelemetry. It has 13 services implemented in 9 programming languages, using both HTTP and gRPC as communication protocols.
- *DeathStarBench* is a suite of benchmark systems that represents large-scale microservice applications [12]. We include their *Hotel Reservation* system, replacing the use of OpenTracing [30] with its successor project, OpenTelemetry.
- The control plane of Reynard is itself a distributed system, comprising services that communicate using HTTP. We apply Reynard to test its own implementation. Furthermore, the communication pattern used by Reynard to register faultloads is a challenging case for fault space reduction, which is not included in the Filibuster Corpus.
- Additionally, we created a new set of microbenchmarks that include common resilience patterns.

To ensure that each evaluation is representative, repeatable, and fair, we took the following precautions. All benchmarks were run on the same machine, equipped with an AMD Ryzen 7 3700X 8-core CPU and 32 GB of RAM. For our industry system, we interacted with a production-like test environment using a virtual desktop with three virtual cores of an Intel Xeon Gold 6254 CPU and 32 GB of RAM. To ensure consistent results and an unbiased runtime, we reported the average results of 10 runs of the benchmark examples and 2 runs of the industrial case study due to its longer runtime. We excluded the start and stop times of the SUT from the runtime. A detailed instruction on how our results are produced can be found in our repository [11].

5.2 Comparison with Filibuster

We compare Reynard to Filibuster [26] by re-running their experiments with an adjusted configuration. We ensure that Filibuster injects the same failure modes, as it can inject library-dependent exceptions at the service level. These omitted failure modes are exceptions related to network-level faults, such as timeouts and connection errors. Additionally, we adjusted the corpus to work with Reynard. We included all benchmarks listed in their paper and ran them with our retry reduction policy when applicable.

Table 2 compares Reynard and Filibuster on the Filibuster Corpus. The fault space columns indicate the number of fault injection points, and how many are conditional (in brackets). Based on this, we indicate the size of the complete fault space, which consists of all combinations of fault injection points and 4 failure modes

Table 2: Comparison of the fault space exploration by Reynard and Filibuster and their respective runtimes.

		Fault space		Reynard (our tool)			Filibuster		
Benchmark	**Variant**	**|P|**	**|$\mathcal{F}$|**	**Explored**	**ΔSER**	**Time (s)**	**Explored**	**ΔSER**	**Time (s)**
cinema-1		2 (0)	25	9	0	0.8	9	0	1.0
cinema-2		2 (0)	25	8	1	0.8	8	1	1.0
cinema-3	normal	4 (2)	625	27	46	1.6	27	46	3.7
	w/ retry reduction	4 (2)	625	19	26	1.4	27	46	3.7
cinema-4		5 (0)	3 125	8	1	1.3	8	1	1.4
cinema-5		2 (0)	25	25	0	1.3	25	0	2.9
cinema-6		3 (1)	125	41	0	2.0	41	0	5.3
cinema-7		4 (1)	625	45	0	2.2	45	0	5.5
cinema-8	normal	2 (1)	25	21	0	1.2	21	0	2.2
	w/ retry reduction	2 (1)	25	9	0	0.8	21	0	2.2
expedia		2 (1)	25	21	0	1.2	21	0	2.2
audible		8 (0)	390 625	30	35	2.5	30	35	5.2
mailchimp		6 (2)	15 625	192	1	9.7	192	1	31.8
netflix*†	w/o bugs	9 (3)	1 953 125	2 440	1	388.2	2 440	1	644.8
	w/ bugs	10 (4)	9 765 625	2 444	9 729	479.0	2 444	9 729	887.9

(as indicated by Table 1). We report the number of concrete test scenario executions, the additional number of test cases when the Service Encapsulation Reduction (SER) policy is disabled, and the average total runtime of the test suite. To provide a more thorough analysis, we also include some variants for specific benchmarks. These variants enable the retry reduction pruning policy in benchmarks with retries, or run a benchmark with specific environment variables to reproduce a particular bug behaviour. Furthermore, we indicate if a benchmark was configured to disregard the *Payload* field (∗) or include the *Predecessors* field (†) in the identifiers.

Overall, Reynard explores the fault space using the same number of test cases as Filibuster. Additionally, Reynard reduces the fault space using the service encapsulation reduction policy (introduced by Filibuster) by the same number of test executions. In cases where the retry reduction policy applies, Reynard achieves a lower number of test cases to exercise compared to Filibuster (highlighted in green). Moreover, the runtime of Reynard is in the same order of magnitude as, but consistently lower than, Filibuster.

Our evaluation on the Filibuster Corpus shows that *network-level fault injection of Reynard achieves the same level of fault space reduction as service-level instrumentation* (**RQ1**). Our fault model and fault space representation identify the same dependencies and redundancies that service-level instrumentation can detect. Moreover, Reynard allows for the composition of multiple reduction polices in our implementation, further reducing the fault space.

5.3 Application to Other Benchmarks

Table 3 lists the results obtained when applying Reynard to benchmark systems beyond the Filibuster Corpus. Similar to Table 2, it lists the fault space size, the injection points, and the number of conditional fault injection points. We again indicate if a benchmark was configured to disregard the *Payload* field (∗) or include the *Predecessors* field (†) in the identifiers. In addition to the concrete test executions and the average runtime for the complete test suite, we also list the average runtime per test execution.

Fault Space Reduction. Reynard reduces the fault space using its pruning policies both in the Filibuster Corpus (Table 2) and with the additional benchmarks (Table 3) (**RQ2**). In the *Astronomy Shop checkout* benchmark, the complete fault space is reduced by 8 orders of magnitude. This result is similar to the *audible* benchmark in the Filibuster Corpus, as the call graph is composed of sequentially dependent calls. For the *register* benchmark, the fault space is significantly reduced due to the retry reduction policy, but without retries, it *cannot* be reduced as each initial call is concurrent and independent.

We observe that including the optional predecessors field in the identification of fault injection points may *increase* the number of concrete test executions. In our microbenchmark, this is due to a retry in the call graph. The retry and its original request differ in their preceding requests, which influence the predecessors' fields of all subsequent requests, changing their identifiers and disallowing the reuse of already observed inclusion and exclusion conditions. However, in the *netflix* benchmark of the Filibuster Corpus, it is necessary to differentiate equal requests that happen for distinct reasons. [1]

When analyzing the effect of the call graph structure on the reduction of the fault space, we find that the search tree is dominated by the largest set of requests with the same direct causal dependency that can have faults injected independently.

Efficiency in Runtime Overhead. We evaluate the overhead introduced by our instrumentation through a benchmark setup that isolates the interaction between a single proxy, its target service, and the instrumentation controller (**RQ3**). For this benchmark setup, we developed a minimal web server in Go to function as the destination for the proxy. This web server features an endpoint that directly responds with a status code of 200 to prevent it from being a bottleneck during performance testing. Furthermore, we implemented two additional endpoints within this web server to simulate the controller, allowing us to measure the overhead of the proxy while minimizing the impact of interactions between the proxy and the controller.

Table 4 lists the results of stress testing the benchmark in several scenarios[2]. Each scenario runs for 60 seconds. We measure

[1] A detailed analysis of the benchmarks can be found in [10].
[2] The details of the test scenarios are available in our benchmark repository: https://github.com/reynard-testing/reynard/tree/main/util/experiments/overhead

Table 3: The fault space exploration and runtime of applying Reynard to benchmark systems.

			Fault space size		Reynard		
System	**Benchmark**	**Variant**	**|P|**	**|$\mathcal{F}$|**	**Explored**	**Time (s)**	**Time per test (ms)**
Astronomy Shop	Shipping*		4 (0)	625	15	1.1	76.6
	Recommendations*		7 (0)	78 125	635	28.8	45.4
	Checkout*		13 (0)	1 220 703 125	65	7.6	117.3
Hotel Reservation	Recommend*		2 (0)	25	9	0.8	90.2
	Reserve*		2 (0)	25	9	0.8	89.8
	Search Hotels*		5 (0)	3 125	17	1.6	96.4
Microbenchmark	Resilience patterns		7 (2)	78 125	548	23.4	42.8
		w/ predecessors†	7 (2)	78 125	555	25.8	46.4
		w/ retry red.	7 (2)	78 125	536	23.3	43.4
		w/ predecessors† and retry red.	7 (2)	78 125	543	25.4	46.7
Reynard	Register (3 proxies)	0 retries	3 (0)	125	125	3.4	27.2
		2 retries, w/ retry red.	9 (6)	1 953 125	729	22.4	30.8
		4 retries, w/ retry red.	15 (12)	30 517 578 125	729	27.3	37.5

Table 4: Measured overhead in stress testing of the scenarios

Scenario	Req/s	Avg. (ms)	P50 (ms)	P90 (ms)	P99 (ms)
Direct to service	10 305	0.38	0.34	0.49	0.56
Missing headers	5 838	0.59	0.66	0.86	1.21
Unknown trace	5 758	0.69	0.53	0.87	1.24
Known trace, no fault	841	4.93	4.17	8.93	15.28
Known trace, no fault, mocked controller	2 126	1.89	1.83	2.42	3.32
Matching fault	805	5.43	4.44	11.06	18.67
Matching fault in large faultloads	803	5.45	4.45	11.12	18.73

throughput in requests per second, as well as the average, 50th, 90th, and 99th percentiles of response time. We take the average of 10 runs. On average, Reynard introduces a small overhead per request, in the order of only a few milliseconds. Moreover, the latency introduced by the proxied benchmark service is minor, with requests and responses taking only 380 μs on average. We observe that the interaction between the proxy and the controller can result in outliers in response time, which are likely due to requests competing for a lock that is used to ensure atomic counts, resulting in delayed requests. In most cases, the induced latency does not affect the behavior of the underlying system.

5.4 Industrial Case Study

We apply Reynard to ASML's internal data analytics platform, a data-intensive system designed as a microservice architecture with approximately 50 to 100 microservices (**RQ4**).

We first selected a test scenario that contains key interactions within the system to explore its resilience against service failures. The selected scenario invokes an operation that involves 6 services and 8 endpoints. For 6 interactions, a single retry is attempted for a specific failure mode. Subsequent calls on these retries result in 9 conditional fault injection points. We ran the scenario with several variations.

Table 5 lists the number of fault injection points, the size of the fault space, and the number of explored test cases by Reynard, along with the test duration in seconds and the time per test in milliseconds. We report the results for each variant of the scenario, namely with no optimizations, with retry reductions, predecessors field, and with both predecessors field and retry reduction.

Our results show that the retry reduction policy reduces the fault space to explore, but less so than in the benchmarks. This is because the system performs retries only for specific status codes, while in the benchmarks, retries were often applied to *all* failure modes. Including the predecessor information in the identifiers leads to *a higher number of* test cases being explored. This is consistent with our findings in the benchmark results. The presence of retries affects the predecessor information, which prevents us from correlating information from identical fault injection points. As a result, we observe an increase in the number of test cases. Additionally, the predecessor information makes the analysis computationally more expensive, resulting in a longer runtime per test case. In this specific scenario, it is not necessary to include the predecessors information into the fault point identification, because there are no indistinguishable interactions with the same identifier, unlike in the *netflix* benchmark. However, this assessment can only be made by manually inspecting the semantics of the interactions. When testing a system with no information about its internal logic, it is safer to include it to identify the fault injection points precisely.

The evaluation of Reynard on the case study provided several insights and uncovered a previously unknown resilience bug.

Higher Runtime per Test. The runtimes per test in the industrial system are significantly higher than in the idealized benchmark systems. This is partly due to its more complex functionality, intricate computations, and database accesses. Moreover, the retry mechanisms in real-world systems often incur a timeout-based retry interval. Therefore, executions that trigger multiple retries produce longer-running tests. Finally, executing a test scenario requires invoking a sequence of other requests to bring the system into a specific state for the test case, and this incurs additional time. Nevertheless, Reynard could reduce the fault space from trillions to less than 500 test cases, which run in ~ 30 minutes in total.

Data Nondeterminism. Unlike idealized benchmark systems, request handling in the industrial case involved data nondeterminism in the form of generated UUIDs. The nondeterminism resulted in varying payloads when invoking the same service call, preventing us from deterministically identifying the fault injection point across executions. Therefore, in our application to the industrial

Table 5: The fault space exploration and runtime of applying Reynard to an industry system.

Scenario Variant	|P|	$\|\mathcal{F}\|$	Explored	Time (s)	Time per test (ms)
no optimizations	18 (9)	3 814 697 265 625	517	1 664.2	3 218.9
w/ retry reduction	18 (9)	3 814 697 265 625	493	1 620.8	3 287.6
w/ predecessors†	18 (9)	3 814 697 265 625	601	2 068.4	3 441.6
w/ predecessors† and retry reduction	18 (9)	3 814 697 265 625	594	2 083.3	3 507.2

case study, we configured Reynard to disregard the *Payload* field in the identification of fault injection points. While this helped identify identical requests, it additionally resulted in non-identical requests having identical identifiers. However, as these requests were not subject to scheduling nondeterminism, they could be uniquely identified by their *Count* field.

Uncovered State Divergence Bug. While testing the industrial system with Reynard, our automated anti-pattern detection found an unexpected system behavior. Although Reynard only injected 5XX HTTP failures, the system responded in some cases with a 404 status code. After inspection, we uncovered a state divergence bug [8] in the interaction between three services. That is, there was a mismatch between the expected and the real state of a service.

Figure 6 illustrates the high-level interaction between these services. Service *A* communicates with service *B*, which relies on service *C*. When faults occur in the call to *C*, then service *B* performs an internal state update and propagates the fault. The issue occurred because service *A* utilized an HTTP client that automatically performs a retry on a 503 status code, in an attempt to resolve a transient fault. Under normal circumstances, this would be safe for non-idempotent endpoints: a 503 status code indicates that the services never received the request. However, service *B* propagated the status code from service *C* in its response, which led to an unexpected retry from service *A*. The endpoint of service *B* is not idempotent, and returns a 404 response due to the state update it performed on the first attempt. This unintended behavior was not discovered during the component testing of the system, but was detected by our tests using Reynard.

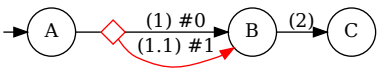


Figure 6: The high-level interaction that caused a resilience bug. Service *A* sends a request to service *B*. In turn, *B* sends a request to *C*. If *B* responds with a status code 503 ("Service Unavailable"), *A* performs a retry.

6 Related Work

Fault-injection testing has been the focus of extensive research to ensure the resilience of distributed systems.

FATE and DESTINI [16] is a pioneering framework for exploring executions with partial failures, comprising FATE, a fault injection service, and the DESTINI specification language for defining correctness properties on executions. FATE uses service-level fault injection by intercepting I/O events, uses unique identifiers for faults as an abstraction, and explores the entire fault space. To combat the exponential explosion of the execution space, FATE incorporates test prioritization using combinatorial testing [9]. PreFail [20] utilizes FATE [16] as its underlying fault injection tool. It offers a *programmable* fault injection framework enabling users to program pruning policies, utilising expert knowledge.

Jepsen [22] targets consistency violations in distributed databases and injects network partitions, process crashes, and clock skews to uncover them. More recent techniques extend the randomized fault generation by guiding the injection using some metadata or coverage information [7, 13, 15, 19, 25, 28]. RainMaker [8] focuses on partial failures of cloud-backed services. It proposes a taxonomy of error-handling bugs in such services and introduces automated test oracles for checking resilience properties. Reynard draws inspiration from RainMaker in its test oracles and fault injection into retry mechanisms. 3MileBeach [35] offers a distributed tracing solution with service-level fault injection capabilities by replacing the protocol (de)serialization libraries. In our work, we avoid adapting the serialisation libraries and instead rely on OpenTelemetry [29].

Lineage-Driven Fault Injection (LDFI) [2] uses the system's lineage information encoded as logical statements and finds new test cases that could violate a property with a SAT Solver. LDFI has been applied to a real-world scenario at Netflix [1] to demonstrate its effectiveness in tackling a large fault space. IntelliFT [24] finds relevant fault injection points using LDFI and then applies an evolutionary search algorithm to guide fault injection. Similar to LDFI, Reynard is also applicable to large-scale industrial systems. However, unlike LDFI, it does not require domain-specific knowledge or encoding of the lineage information of the system components.

Gremlin [17] is a network-level fault injection tool that inspired the fault-injection design of Reynard. However, the exploration in Gremlin is not automated. It primarily supports systematic assertions on the correct functioning of resilience patterns, but does not systematically explore the fault space. Reynard systematically explores the fault space employing effective pruning techniques.

Filibuster [26] is a service-level automated fault injection tester that differs from our work primarily in its fault injection and fault loads exploration methods. Filibuster simulates service- and network-level failures by modifying the OpenTelemetry instrumentation, and has access to the runtime information of services, which enables accurate identification of causal dependencies and fault injection points pinpointing. Reynard does not alter OpenTelemetry instrumentation to provide correlation and implements network-level fault injection testing, and is therefore directly applicable to various runtime environments, while Filibuster necessitates tailored instrumentation for each programming language and framework.

7 Limitations and Future Work

Nondeterminism. Following previous work [2, 26], we assumed that requests, responses, and their effects are deterministic. However, many microservice applications exhibit scheduling and data nondeterminism. For example, microservices can have asynchronous or concurrent runtimes, resulting in scheduling nondeterminism, while UUIDs and timestamps are causes of data nondeterminism. Both forms of nondeterminism influence fields in the injection point identifiers, causing equivalent points to have different identifiers across different executions. Consequently, the information stored to model the system behavior is incorrect. This results in an ineffective pruning of the search space. To tackle this, we allow for related fields to be ignored by configuration. This provides a sound exploration as long as each distinct call has a unique identifier. However, when this is not the case, it will result in an unsound exploration of the search space. Future work can investigate how realistic cases with data or scheduling nondeterminism can be handled. With extended instrumentation, it is possible to ensure a deterministic ordering of requests. Alternatively, we can define equivalence classes of identifiers, similar to those in PreFail [20].

Asynchronous service communication. Our analysis focuses on request-response communication patterns between microservices. However, microservice application often include *asynchronous* communication such as publisher-subscriber patterns using message queues. This poses two challenges for Reynard. First, message queues use their own binary protocols for producing and consuming events, which the proxies do not understand. Second, Reynard assumes the system interaction to be complete once the initial request receives a response, which is not trivial for asynchronous systems. Future work can investigate how to incorporate asynchronous communication patterns into Reynard.

8 Conclusion

We presented Reynard, which utilizes network-level instrumentation to enable automated fault injection testing, facilitating practical application to existing microservice applications. Reynard allows for precise fault injection by using a flexible fault identifier. Furthermore, it enables efficient and effective exploration of the fault space by using pruning policies based on a model of the system's behavior. Our evaluation demonstrated that Reynard can reduce the fault space to the same degree as a state-of-the-art service-level fault injection, while being more efficient in its runtime and having minimal overhead. Moreover, our industrial case study highlights its applicability to real systems and its ability to find new bugs.

Acknowledgments

We thank Laurens Versluis for his support and feedback on the industrial application. We also thank Christopher S. Meiklejohn for his support with the Filibuster benchmarks.

References

[1] 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 ACM Symposium on Cloud Computing*. 17–28. doi:10.1145/2987550.2987555

[2] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. 2015. Lineage-Driven Fault Injection. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 331–346. doi:10.1145/2723372.2723711

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

[4] Ali Basiri, Lorin Hochstein, Nora Jones, and Haley Tucker. 2019. Automating chaos experiments in production. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, Helen Sharp and Mike Whalen (Eds.). IEEE / ACM, 31–40. doi:10.1109/ICSE-SEIP.2019.00012

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

[6] 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 Transactions on Dependable and Secure Computing* 21, 5 (Sept. 2024), 4921–4938. doi:10.1109/TDSC.2024.3363902

[7] 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

[8] Yinfang Chen, Xudong Sun, Suman Nath, Ze Yang, and Tianyin Xu. 2023. Push-Button reliability testing for Cloud-Backed applications with Rainmaker. In *USENIX symposium on networked systems design and implementation*. 1701–1716.

[9] David M. Cohen, Siddhartha R. Dalal, Jesse Parelius, and Gardner C. Patton. 1996. The Combinatorial Design Approach to Automatic Test Generation. *IEEE Software* 13, 5 (Sept. 1996), 83–88. doi:10.1109/52.536462

[10] Delano Flipse. 2025. *Exploring Beyond the Happy Path: Practical Automated Network-Level Fault Injection Testing of Service-Oriented Distributed Systems*. Master's thesis. Delft University of Technology. https://resolver.tudelft.nl/uuid:1b478188-f61d-4f3f-93ab-443cfdccc78f

[11] Delano Flipse. 2025. *Reynard: Practical Network-Level Fault Injection Testing of Service-Oriented Distributed Systems*. https://github.com/reynard-testing/reynard

[12] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18. doi:10.1145/3297858.3304013

[13] Yu Gao, Wensheng Dou, Dong Wang, Wenhan Feng, Jun Wei, Hua Zhong, and Tao Huang. 2023. Coverage Guided Fault Injection for Cloud Systems. In *International Conference on Software Engineering*. IEEE / ACM, 2211–2223. doi:10.1109/ICSE48619.2023.00186

[14] Supriyo Ghosh, Manish Shetty, Chetan Bansal, and Suman Nath. 2022. How to fight production incidents?: an empirical study on a large-scale cloud service. In *Proceedings of the Symposium on Cloud Computing Pages*. ACM, 126–141. doi:10.1145/3542929.3563482

[15] Ege Berkay Gulcan, Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Srinidhi Nagendra. 2025. Model-Guided Fuzzing of Distributed Systems. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 282 (Oct. 2025), 28 pages. doi:10.1145/3763060

[16] 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 *USENIX Symposium on Networked Systems Design and Implementation*.

[17] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K. Reiter, and Vyas Sekar. 2016. Gremlin: Systematic Resilience Testing of Microservices. In *IEEE International Conference on Distributed Computing Systems*. 57–66. doi:10.1109/ICDCS.2016.11

[18] Mike Isaac and Sheera Frenkel. 2021. Gone in Minutes, Out for Hours: Outage Shakes Facebook. *The New York Times* (Oct. 2021).

[19] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2020. Fuzzing Error Handling Code using Context-Sensitive Software Fault Injection. In *USENIX Security Symposium*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2595–2612. https://www.usenix.org/conference/usenixsecurity20/presentation/jiang

[20] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. 2011. PREFAIL: A Programmable Tool for Multiple-Failure Injection. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 171–188. doi:10.1145/2048066.2048082

[21] Sergey Kanzhelev, Morgan McLean, Alois Reitbauer, Yuri Shkuro, Bogdan Drutu, and Nik Molnar. 2021. *Trace Context*. W3C Recommendation. W3C. https://www.w3.org/TR/2021/REC-trace-context-1-20211123/

[22] Kyle Kingsbury. 2024. Distributed Systems Safety Research. https://jepsen.io

[23] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. 2019. What bugs cause production cloud incidents?. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. ACM, 155–162. doi:10.1145/3317550.3321438

[24] Zhenyue Long, Guoquan Wu, Xiaojiang Chen, Chengxu Cui, Wei Chen, and Jun Wei. 2020. Fitness-Guided Resilience Testing of Microservice-based Applications.

In *IEEE International Conference on Web Services*. 151–158. doi:10.1109/ICWS49710.2020.00027

[25] 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 ACM Symposium on Operating Systems Principles*, Tim Brecht and Carey Williamson (Eds.). ACM, 114–130. doi:10.1145/3341301.3359645

[26] Christopher S. Meiklejohn, Andrea Estrada, Yiwen Song, Heather Miller, and Rohan Padhye. 2021. Service-Level Fault Injection Testing. In *ACM Symposium on Cloud Computing*, Vol. 40. 388–402. doi:10.1145/3472883.3487005

[27] Christopher S. Meiklejohn, Rohan Padhye, and Heather Miller. 2022. Distributed Execution Indexing. (Sept. 2022). doi:10.48550/ARXIV.2209.08740

[28] Ruijie Meng, George Pîrlea, Abhik Roychoudhury, and Ilya Sergey. 2023. Greybox Fuzzing of Distributed Systems. In *Proceedings of the SIGSAC Conference on Computer and Communications Security*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM, 1615–1629. doi:10.1145/3576915.3623097

[29] OpenTelemetry. 2025. *OpenTelemetry*. https://opentelemetry.io/

[30] OpenTracing. 2025. *OpenTracing*. https://opentracing.io/

[31] Casey Rosenthal. 2017. Principles of Chaos Engineering. USENIX Association, San Francisco, CA.

[32] Bogdan Alexandru Stoica, Utsav Sethi, Yiming Su, Cyrus Zhou, Shan Lu, Jonathan Mace, Madanlal Musuvathi, and Suman Nath. 2024. If At First You Don't Succeed, Try, Try, Again...? Insights and LLM-informed Tooling for Detecting Retry Bugs in Software Systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*. ACM, 63–78. doi:10.1145/3694715.3695971

[33] Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. 1994. *A Note on Distributed Computing*. Technical Report. Sun Microsystems, Inc.

[34] Yingying Wang, Harshavardhan Kadiyala, and Julia Rubin. 2021. Promises and challenges of microservices: an exploratory study. *Empirical Software Engineering* 26, 4 (2021), 63. doi:10.1007/s10664-020-09910-y

[35] Jun Zhang, Robert Ferydouni, Aldrin Montana, Daniel Bittman, and Peter Alvaro. 2021. 3MileBeach: A Tracer with Teeth. In *ACM Symposium on Cloud Computing*. 458–472. doi:10.1145/3472883.3486986

[36] 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 Transactions on Software Engineering* 47, 2 (2021), 243–260. doi:10.1109/TSE.2018.2887384

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009