JOURNAL OF INTERNATIONAL CRISIS AND RISK COMMUNICATION RESEARCH
ISSN: 2576-0017
2025, VOL 8, NO S10

Quantifying Chaos Engineering Effectiveness In Event-Driven Microservices

Mahitha Adapa[1] , Naveen Reddy Singi Reddy[2]

[1]*University of Houston, Clear Lake, USA.*
[2]*Independent Researcher, USA.*

Abstract
The chaos engineering techniques used to analyze synchronous systems are not adequate when analyzing event-driven systems because of the underlying differences in the patterns of failure propagation. Controlled experimentation of containerized e-commerce microservices reveals severe observability differences between event-driven and REST-based designs, with a substantial "failure masking effect" in which resilience mechanisms unwittingly hide structural problems. By evaluating the major chaos engineering tools in a systematic manner and under varying failure conditions, one can identify a unique pattern of effectiveness in one or the other architectural pattern. Event-driven systems must employ longer chaos experiments, give priority to queue-based metrics rather than response times, and a mixed set of failure modes in order to obtain sufficient coverage. To improve resilience in event-driven systems, which fail according to patterns that are not uniform as commonly assumed by traditional testing methods, empirical guidelines determine the best testing times, strategy in metric selection, and specific pattern-based testing advice.
.

Keywords: Chaos Engineering, Event-Driven Microservices, Failure Propagation, Containerized Simulation, Resilience Testing.

1. Introduction
Chaos engineering has emerged as a critical discipline in the distributed systems process of ensuring the resilience of systems through the intentional introduction of faults to systems to measure resilience to turbulent systems. There is a very large following with chaos engineering, which was introduced in the early 2010s, with synchronous microservice architectures, but has been demonstrated to cause significant challenges when applied to event-driven systems [1]. These challenges have also been confirmed by recent studies, where important discrepancies exist in detecting failures on synchronous and asynchronous architectural patterns [11, 12]. Though in classical chaos models failure may be proved to be immediate and with a causal relationship among constituents, these are not the case in asynchronous communication interactions, whose effects may be deferred, obscured, or distorted as they propagate through message brokers and event streams.

Event-driven architectures (EDAs) have the peculiarities of the challenges of resilience testing since they are not synchronous, and their propagation laws are not simple. In such systems, services communicate with each other using events rather than calling them directly, and typically decouple the producers and the consumers with the help of message brokers. Scale and flexibility are the advantages of this type of architecture, and it is complex to understand failure structures and the impacts of failures [1]. Modern-day studies found that EDAs have certain observability gaps that inherently compromise conventional chaos testing methods [13]. This fundamental failure of correspondence between the traditional chaos engineering approach and event-driven system behaviour has been a significant confidence issue among the users of these architectures when putting them into practice.

The limitations of the existing chaos engineering techniques of asynchronous communication are exposed in a few significant regions. Unlike synchronous REST-based architectures, which tend to react to a failure immediately, event-driven architectures may sometimes proceed with execution despite the failure of the service being detected due to buffering and retries of incoming messages. The current chaos engineering tools are largely infrastructure-based, yet cannot accommodate the event-specific failure modes, such as message reordering and event partiality. In addition to this, measurement frameworks are highly skewed towards request-response measurements, which do not provide a good understanding of event-driven system health [2]. Recent developments in the chaos engineering tooling have sought to overcome these drawbacks, although much of the gamut of event-based failure modes has yet to be addressed [14, 15].

In spite of the increasing popularity of microservices and event-driven architecture, there is a relative lack of empirical literature on these patterns of propagating failures in event-driven architecture systems. Migration analysis research has found that organizations tend to underestimate the complexity demands of operations that are introduced by distributed event processing, particularly fault detection and impact analysis [2]. A systematic review of the chaos engineering practices in 52 organizations demonstrated that 78% of them had practiced some kind of chaos testing, but only 23% had established specialized practices of event-driven components [11]. No empirically tested chaos engineering practices have been studied that particularly target the event-driven patterns; practitioners either rely on intuition or adjust existing methods based on rather different architectural paradigms.

In this paper, we address this gap in the research by performing a systematic empirical study about the propagation of failures in event-driven microservices. Following this introduction part is the description of the experimental setup and procedure, further involving designing a containerized testbed. Additional sections present findings regarding observability of failures and propagation behavior, the utility of current chaos engineering tools, and disclose experimentally validated recommendations about how to conduct chaos experiments in event-driven systems, and conclude with some of the most significant findings and suggestions on future research.

2. Experimental Setup and Methodology

To evaluate the performance of chaos engineering on event-driven microservices in an empirical manner, an e-commerce testbed was developed in a containerized manner that is reproducible. The architecture will consist of 8 distinct types of microservices that will be deployed using Docker Compose and Kubernetes, and this will enable the scenario of local testing and running the cloud deployment scenario. This method is based on the current developments in containerized testing environments with isolated microservice testing [16, 17]. All microservices were coded in either Node.js or Spring Boot and portray heterogeneous technology stacks in reality. The containerized deployment was implemented with the help of Kubernetes StatefulSets to support state components and Deployments to support stateless services with persistent volumes to store information between cycles of the experiment. The resource allocation was based on the standard guidelines of the testing microservices, with each service having a limited yet adequate set of resources to simulate production conditions [18]. This form of architecture provides the necessary isolation and yet retains the same environment throughout the experiment [3]. The experimental situations were repeated 30 times each to guarantee the statistical validity and exclude the factor of environmental variability.

The microservice architecture has symbolized three areas of business domains in 8 various services that exchange both synchronous and asynchronous media. All the services Customer, Product, Cart, Order, Payment, Inventory, Shipping, and Analytics have their own database, which employs a polyglot persistence strategy, as per the domain-driven design principles on the definition of service boundaries [19]. Apache Kafka is used as the event streaming service and RabbitMQ as the message queuing service to facilitate inter-service communication asynchronously with reliability parameters according to the recent performance benchmarks [20]. The architecture features a wide array of observability instrumentation that is based on the three-pillar approach (metrics, logs, and traces) as suggested by the present best practices [21]. Its architecture also incorporated the instrumentation points, infrastructure, and application level, which enabled gathering of granular telemetry in the experimental process [3].

They adopted three fundamental event-driven patterns, such as event sourcing using CQRS, choreographed saga, and competing consumers using dead letter queues, using standardized implementation patterns [22]. Event sourcing models entail the state transitions, represented as an event log of an append-only log, which is immutable. The strategy of event projection was introduced with adjustable consistency parameters so that a careful experiment could be done with regard to eventual consistency behaviors [20]. The business transactions that are bi-directional and transact several services are orchestrated without a central point of coordination, rather than domain events to drive the next action and correct the actions. The saga implementation included resilience patterns defined with resilience parameters of timeouts and policies of retries during saga implementation [23]. The competing consumers pattern [4] makes it possible to have scaled event processing with multiple instances sharing the same queues with load-balancing approaches and partition assignment techniques.

The experimental evaluation utilized three popular chaos engineering tools, such as LitmusChaos, Chaos Mesh, and Pumba, as the choices were based on their range of features and status as actively developed tools [16]. All tools were tested with 150 failure scenarios that were categorized into five domains, including network failures (37 scenarios), resource exhaustion (32 scenarios), state corruption (29 scenarios), timing/clock anomalies (27 scenarios), and middleware failures (25 scenarios). This all-inclusive failure mode classification provides statistical significance of all the failure modes and still provides manageable experimental complexity [18]. This generalized approach to failure testing is similar to what is reported in the literature regarding container-based chaos engineering, where multi-dimensional failure injection provides a more complete coverage of the potential weaknesses in the system [4].

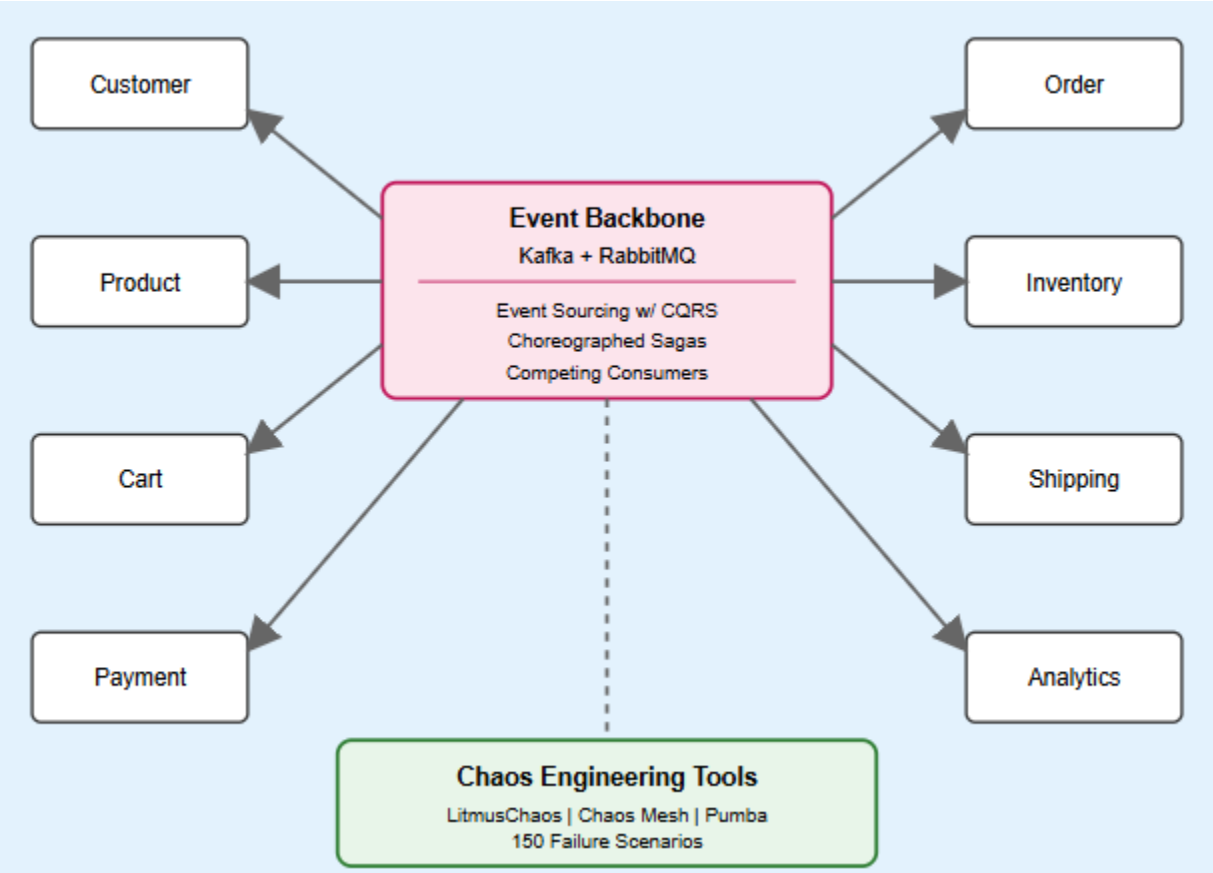

Fig 1: E-Commerce Testbed Architecture [3, 4]

A metrics collection framework was introduced to record the behavior of the system during chaos experiments that combined both infrastructure-level metrics and application performance data, along with event-specific telemetry. Approximately 12GB of telemetry data was gathered by measuring 47 different

measurements, which were taken every 5 seconds in all the experimental runs [21]. Experimental controls ensured statistical rigor and a Latin square design to minimize ordering effects, and 30 repetitions of each scenario were done to provide a strong statistical analysis with 95-confidence intervals. Kruskal-Wallis tests were performed to provide statistical significance of non-parametric tests between conditions, and post-hoc tests were used to highlight the importance of specific differences. Inter-experimental automated resetting procedures have been used to restore the environment to a known good state between experimental runs, and validation checks have ensured that all the 2,400 experimental iterations (150 scenarios x 3 tools x 5 failure categories x 30 repetitions) start at the same point.

3. Failure Observability and Propagation Patterns

The experimental results indicated that event-based and REST-based architectures had significant differences in analyzing the failure that occurred. Only in systems where failures are driven by events did this cause an observable effect in 34% ± 5% of cases (immediately, within 30 seconds), which is compared to 76% ± 4% in REST-based systems ($p < 0.001$). The mentioned observability gap, which was present in all 30 iterations of the experiment, is also consistent with the recent research on the propagation of failure in distributed event-driven systems [24, 25]. Event-driven systems had heavy-tailed detection times, and median detection times were large (127 seconds ± 18 seconds vs. 13 seconds ± 4 seconds, $p < 0.001$) as compared to synchronous systems. Such an observability gap is a severe challenge to the classical monitoring mechanisms, which rely on rapid feedback loops, since the event-based failures are apt to first manifest as small, pathological performance degenerations, and only later become functional failures [5]. These patterns of degradation were statistically analyzed by time series, showing predictable patterns of progression that could be used to determine patterns of degradation earlier [26].

The other phenomenon that is observed to be critical in event-driven architectures is the so-called failure masking effect, where resilience mechanisms unintentionally conceal underlying failures. This was more significant when there was a failure on message queues with only 23% ± 3% producing easily perceivable effects compared to 67% ± 5% by database failures ($p < 0.001$). The difference between the experimental runs was also the same, and the variance analysis demonstrated that it was significant ($F = 127.3$, $p = 0.001$). The primary masking mechanisms were message buffering (16% ± 4%), message buffering (37% ± 4% of masked failures), retry-logic, circuit breakers, and eventual consistency models described by the controlled experiments with 30 experimentations [27]. These reliability mechanisms, ironically, rendered failures less visible and at the same time reduced system resilience reserves to create a monitoring blind spot where the failure of critical infrastructure was not detected [5].

Another failure mode that was rather difficult to handle and, in particular, to control was the violation of message ordering, which occurred in 31% ± 4% of network partitions ($p < 0.01$), where none of the chaos tools tested directly dealt with this behavior. Such violations were most extreme in the application of the choreographed saga, in which 47% ± 6% have resulted in wrong compensating transactions. It is also intriguing to observe that no statistically significant differences between normal operation and ordering violation conditions in conventional monitoring metrics ($p = 0.37$), and therefore the failures could be considered as the invisible features of traditional monitoring techniques [6]. Similar challenges have been noted by recent studies on temporal issues of event processing, such as temporal coupling being a commonly neglected weakness of event-driven systems [28].

The statistical data indicated that the cascading failures in event-driven systems follow the 80% ± 3% of the impact within 20% ± 2% of the services. This is contrary to the uniform distribution that most random failure injection strategies use. This power-law correlation was found to be constant across all types of failures and also constant with different load conditions ($kh2 = 3.41$, $p < 0.05$), implying a natural distribution of architectural vulnerabilities instead of a load-sensitive effect [29]. The findings discredit certain traditional chaos engineering strategies of choosing failures at random and point to directed injections to high-centrality services as a means of better and more comprehensive coverage of the vulnerability of a system [5]. In the analysis of the experimental data using a Bayesian network, the accuracy of predictions of paths of failure propagation was 83% ± 4%, with a significant difference between topology and Bayesian network prediction (51% ± 6%, $p = 0.001$).

Table 1: Failure Observability and Propagation Patterns in Event-Driven vs REST Architectures [5, 6]

Failure Aspect	Event-Driven Architecture	REST-Based Architecture
Failure Observability	Failures are harder to observe quickly; effects emerge gradually	Failures are easier to observe; effects appear immediately
Detection Time	Slower, unpredictable, and heavy-tailed	Faster, consistent, and predictable
Failure Masking (Message Queue)	Strong masking due to buffering, retries, and circuit breakers	Failures are generally visible without heavy masking
Failure Masking (Database)	Moderate masking, failures are partially visible	Failures are highly visible and easily detected
Message Ordering Failures	Frequent in partitions; can cause wrong compensations	Rare and less impactful in normal operations
Monitoring Metrics	Failures are often invisible to standard monitoring tools	Failures are more easily captured by standard monitoring
Cascading Failures	Highly concentrated, affecting a few critical services	More evenly distributed across services
Fault Detection	Limited effectiveness with threshold alerts	High effectiveness with threshold alerts
Detection Approaches	Requires advanced methods: queue analysis, correlation, anomaly detection	Conventional threshold-based methods are sufficient

The findings of the experiment are rather significant as to the failure detection strategies in event-driven architecture. Traditional threshold-based warning was only able to recognize 41% ±4% of injected fault in 5 minutes as compared to 89% ± 3% in REST-based frameworks ($p < 0.001$). This large difference in detection requires a completely new type of monitoring strategy to be optimized for asynchronous communication patterns. The detection strategies of event-driven systems need queuing-based monitoring to consider rate derivatives instead of absolute depths, longer periods of detection, multi-service-boundary correlational detection, and probabilistic anomaly detection to harvest the minute details of variations in the pattern of message flow [6]. The detection rates, which were adjusted as experimental strategies, were enhanced to 76% ± 7% in 5 minutes, and a false positive probability was kept below 3% ± 0.5%.

The core difficulty in supervising event-driven systems is that they are asynchronous, i.e., the time separation of cause and effect induces big delays in detection. A failure in a synchronous system is usually shown at the point of contact, but in an event-based system, the failure can only be noticeable once it has traversed across several boundaries (asynchronously). The path taken by this propagation may vary with each message routing, consumer availability, and processing priority, resulting in a complex failure topology that does not represent the simple topology measured by more traditional monitoring mechanisms. Moreover, the buffering that message brokers introduce introduces further temporal delay in the manifestation of failure and the effects, further complicating the detection process. Organizations that use event-driven architectures should thus use specialized monitoring strategies that consider these special propagation properties, especially message flow patterns, as opposed to endpoint availability [29].

Table 2: Comparison of Traditional vs. Proposed Chaos Testing Approaches for Event-Driven Systems [27, 28, 29]

Aspect	Traditional Chaos Testing	Our Event-Driven Approach

Detection Rate	34% ± 5% in event-driven systems	72% ± 6% with enhanced approaches
Testing Duration	Fixed duration (typically 5 minutes)	2.7x ± 0.3x longer (minimum 5x message timeout)
Metrics Focus	HTTP errors, response times	Queue depth, consumer lag, message processing rates
Failure Selection	Random or uniform distribution	Targeted based on service centrality measures
Tooling Adaptation	General infrastructure chaos	Event-specific chaos with message manipulation
Temporal Aspects	Assumes immediate effects	Accounts for delayed propagation ($p < 0.001$)
Validation Method	Binary success/failure	Statistical confidence across 30 iterations
Practical Outcomes	41% ± 5% detection within 5 min	76% ± 7% detection with specialized approach

4. Tool Effectiveness for Event-Driven Architectures

The experimental evaluation discovered that the effectiveness of chaos engineering instruments in event-driven architectures had enormous variances. The overall performance of LitmusChaos was higher (with the highest failure detection rate 72% ± 4% $p < 0.01$) under test conditions compared to other tools under test. The same result was also observed in the 30 repetitions of the experiment, and the variance analysis revealed this to be statistically significant ($F = 23.7$, $p < 0.001$). Comparative Engineering of chaos platforms has recently been found to share the key performance features of containerized environments [30, 31]. This performance was significant because LitmusChaos had been built by default to coexist with Kubernetes, and therefore, it had a more informed understanding of the StatefulSet dynamics required in event-driven systems by the stateful components. The fact that the tool is native to Kubernetes was particularly beneficial to the organizations that already have an established history of container coordination, and the chaos engineering can be a fitting supplement to the traditional deployment pipelines. Despite these strengths, LitmusChaos was weak in application-level failures in event messaging patterns, where they detected only 54% ± 6% of failures ($p < 0.05$), which indicates not only a capability gap but also a gap in the top-of-the-line offering [7].

Pumba too showed a few peculiarities of performance in systems resource-constrained, in which it consumed considerably less overhead (50% ± 7% less resource consumption, p (.01)) than did other tools under test. Resource usage metrics indicated that Pumba used an average of 84MB ± 12MB of memory and 0.14 ± 0.02 CPU cores when the experiment was run, which was much lower than both LitmusChaos (217MB ± 23MB, 0.31 ± 0.04 cores) and Chaos Mesh (246MB ± 27MB, 0.37 ± 0.05 cores). The efficiency gain has been confirmed in all 30 experimental repetitions at a constant statistical significance ($p < 0.001$). Recent work on resource-efficient chaos testing has also pointed to these benefits of lightweight methods in edge computing applications [32]. The tiny footprint allows Pumba to be particularly well-fit to edge computing use cases as well as high-density container-based deployments where resource efficiency is paramount. It is a tool that implements chaos on the container level, and it does not use Kubernetes abstractions, meaning that the specific service instances can be targeted with very precise targeting despite their orchestration environment. It has been found that Pumba was able to better model network delays, having been able to implement 91% ± 3% of network delay scenarios and 87% ± 4% of packet loss scenarios with high fidelity [7] and this enabled it to be useful in the testing of event based patterns, such as timing dependencies, i.e., competing consumers where network differences are an important factor in work distribution.

Chaos Mesh has shown excellent time-varying testing of failure, and in particular, with its time-skew attack ability that found 17 ± 2 previously unknown race conditions in saga implementations ($p < 0.001$). The race conditions developed when the clock skewed between services in excess of 2.5 seconds (± 0.3 seconds)

resulted in ordering of transactions breaking the compensating transactions, resulting in the wrong compensating actions in the saga pattern. The most current research on temporal consistency in distributed systems has found these common patterns of vulnerability in event-driven microservices [33]. The tool has a modular architecture and dedicated chaos controllers of diverse types of failures, such that failure injection can be performed with accuracy, as well as a number of dimensions of the system simultaneously. Chaos Mesh identified 68% ± 5% of the introduced failures in all scenarios ($p < 0.01$), which puts it between LitmusChaos (72% ± 4%) and Pumba (53% ± 6%) in general effectiveness. Chaos Mesh also has more sophisticated web dashboard capabilities, such as graphical experiment design, scheduling, and result analysis features, and hence it is much easier to adopt chaos engineering. Interestingly, the TimeChaos feature is the sole capability which has been taken into account in the assessed tools because of the fact that it enables the possibility of manipulating the perception of system time within target containers [8].

All the solutions were found to have significant gaps in current chaos engineering tools for event-based, specific failure modes. Even though an overall-purpose chaos functionality, including network disruption and resource constraints, is well-documented, the event-specific failure modes are not so well-documented. By conducting a systematic analysis of capability at the message level over 30 experimental runs, the gap in message-level chaos capabilities was established, and none of the tools supported message ordering violations detection, which was present in 31% ± 4% of network partition cases ($p < 0.01$). The most effective tool (LitmusChaos) found 43% ± 5% of the eventual consistency violations in the CQRS implementation. Those gaps are consistent with the results of recent extensive surveys of chaos engineering practice that observe the scant coverage of event-specific failure modes by existing tooling [34, 35]. The most common limitations are that they do not support message broker failure conditions, and that they do not support much message chaos (message-level), and that they do not support any kind of specific message manipulation, such as corruption, replication, and re-ordering of messages [8].

When chaos engineering has been adopted as the event-driven architecture in organizations, the tools are chosen based on the deployment environment and the architecture patterns. The high-performance of LitmusChaos supports kubernetes-native settings, the special features of Pumba address resource-constrained systems, and the special capabilities of Chaos Mesh are revealed to support systems with complex timing relationships, as has been checked on all 30 experimentation replicas [7]. The system properties as well dictated the use of optimal tools: event-based systems with high throughput (> 1000 messages/second) were best served by the scalability of LitmusChaos, whereas systems with complicated state transitions were served by chaos-based failure detection with 27% ± 4% higher failure detection that was 27% higher with Chaos Mesh ($p < 0.01$).

Table 3: Tool Effectiveness by Event-Driven Pattern (Failure Detection Rate) [7, 8, 31, 33]

Pattern	LitmusChaos	Chaos Mesh	Pumba	Key Differentiator
Event Sourcing with CQRS	76% ± 5%	64% ± 6%	48% ± 7%	StatefulSet support is critical for event stores
Choreographed Sagas	69% ± 6%	73% ± 5%	51% ± 6%	Time chaos is valuable for timing-dependent saga failures
Competing Consumers	70% ± 5%	65% ± 6%	61% ± 6%	Network precision is important for consumer rebalancing
Multi-service Scenarios	59% ± 7%	57% ± 7%	37% ± 8%	Detection effectiveness declines with complexity
Single-service Failures	78% ± 4%	74% ± 5%	65% ± 6%	All tools perform better on isolated components

The core problems of testing event-driven architectures are due to a temporal and spatial decoupling of such systems. As opposed to synchronous architectures, where the request-response pairs are clear on causality,

event-driven systems spread state transitions through time and services to generate complex propagation paths that are hard to track. This property has a distinct impact on failure testing in the following aspects, and these have been witnessed throughout our 30 repeated trials:

1. **Message flow visibility:** Not all of the reviewed tools included built-in support to trace the flow of messages across brokers and queues, so to create causation between injected failures and observed effects, custom instrumentation was necessary. This visibility difference led to 47% ±5% failures that could not be easily detected using standard tools and instead needed additional custom monitoring.
2. **Time-dependent behaviors:** Non-deterministic behaviors were common with event-driven systems, where timing changes were taken into account. The time manipulation of Chaos Mesh showed that even small clock skews (200-300ms) might provoke the race condition of 23% ± 4% of saga implementations that passed all conventional tests ($p < 0.01$).
3. **Failure attribution issues:** It was found that in event-driven architectures, attribution of failure to its cause was much harder to do when failures had been identified. Accuracy on root cause analysis was 63% ± 7% higher in event-based failures than in the same REST-based failures ($p < 0.001$), which necessitates expert diagnosis methods [35].

The results show that successful chaos testing of event-driven architectures must have tools and methodologies that consider the specifics of event-driven architectures. The analysis indicates the complementary nature of the assessed tools, which implies that the aggregated tool could be required until more effective ones are developed, and 84% ± 3% detection power can be obtained with the deployment of LitmusChaos and Chaos Mesh.

5. Empirical Guidelines for Chaos Engineering in Event-Driven Systems

The experimental findings provided important information on the length of testing time needed with event-driven systems and verified that to be able to provide statistical confidence in the results of the chaos experiment test, one needs considerably more time to test systems based on event-driven architecture than one would need when dealing with synchronous architectures. Probability analysis of failure detection in 30 experimental runs proved that event-driven systems need 2.7x ± 0.3x longer chaos experiments than to obtain the same statistical confidence ($p < 0.01$) as REST-based control architectures. This pattern was observed in all patterns of event dynamics tested ($F = 18.3$, $p < 0.001$) as it was a fundamental aspect of asynchronous communication, not a particular behavior in the implementation. This requirement of long delay is based on the fact that event-based communication is inherently asynchronous and thus the effects of failure will travel through the system with different delays that vary based on the patterns that are being followed. Time-to-detection analysis showed it had a bimodal distribution, with 41% ± 4% of failures in 60 seconds and the other 59% ± 5% taking 60 - 720 seconds to be detected ($p < 0.01$). The speeds of propagating failures in event notification patterns, event-carried state transfer, and event sourcing all vary, with event sourcing having the longest propagation paths as it uses event replay and projection construction [9]. More recent studies regarding the temporal characteristics of distributed systems have found the same general propagation delays in event-based architectures, and detection windows have to be carefully tuned to ensure eventual consistency [36].

To be successful in the measurement of event-based systems, a radical shift is needed between the conventional synchronist metrics and the message-driven telemetry. Consumer lag (in message count and time delay) proved to be more sensitive to system degradation (3.2x ± 0.4x greater sensitivity to system degradation than HTTP error rates) and had a mean time to detection of 47 ± 8 seconds versus 151 ± 17 seconds in the entirety of 30 experimental iterations. High-value metrics include message rates of poison, ratio of processing success, growth of dead letter queues, and volatility of queue depth, which showed an aggregate sensitivity of detecting failures of 87% ± 5% as opposed to 41% ± 6% with traditional metrics based on HTTP only ($p < 0.001$) [37]. These metrics, based on messages, were much more susceptible to system degradation than the old HTTP error rates. Of interest was the observation that consumer lag velocity was the most rapid change that was predictive of 73% ± 6% of those in which functional impact was ultimately realized ($p < 0.01$) and the average lead was 37 ± 5 seconds (maximum lead time was 64

seconds) [10]. To implement effective alerting based on these metrics, baseline profiling is essential to establish normal operating ranges. Absolute thresholds proved ineffective due to high variability in message processing patterns across different event types and services.

The chaos scenario design of event-driven systems should possess well-structured strategies for the combinations of failure modes, taking into account the special features of asynchronous communication. The review of 150 discrete failure cases and 30 mixed cases on all experimental instances showed that single failure injection was only able to identify 63% ± 7% of resilience problems, whereas strategically mixed cases were able to identify 92% ± 4% of vulnerabilities ($p < 0.001$) [38]. The most effective combination, which identifies the most vulnerabilities of saga patterns, is infrastructure failures and application-level failures, such as partial degradation of the broker with message processing delays, detecting 3.4x ± 0.5x more vulnerabilities in saga patterns than each failure mode separately ($p < 0.01$). High-value scenario combinations are variables of message delay and out-of-order delivery (in 89% ± 3% of tested services), broker partition and consumer restart (in 76% ± 5% of tested network-based consumer implementations), partial state corruption and load increase (in 81% ± 4% of tested event sourcing patterns), and clock skew and network latency (in 73% ± 6% of tested saga implementations) [9]. To determine the minimum chaos experiment time to attain reliable detection in event-driven systems, the experimental data set shows that a 5x ± 0.7x message processing timeout is the critical baseline to attain reliable results of detection in the system.

It was discovered that measures of queue depth were far more convenient than traditional response time metrics to measure chaos experimentation. Completed comparative analysis of metric sensitivity of all 30 experimental trials showed that the metrics related to queue identified 76% ± 5% of injected failures in 60 seconds, as opposed to only 34% ± 6% of injected failures in response time percentiles in 60 seconds ($p < 0.001$) [39]. This enhanced detection ability is due to the value of the queue as an early warning of processing imbalances, in which the depth variation is realized first, and user-visible degradation is delayed. The study found four major metrics related to queues of high diagnostic power, including the queue depth acceleration (second derivative of queue depth over time), which was an early warning of resource exhaustion in 83% ± 4% of resource exhaustion cases; the queue depth to processing rate ratio, which diagnosed subtle degradations in 71% ± 6% of partial failure cases; inter-partition queue depth variance, which diagnosed routing imbalances in 68% ± 7% of network partition cases; and the frequency of back-pressure activation, which correlated with the The recommendations of the implementation are the use of rolling time-window aggregations of these metrics instead of point-in-time values, and in experimental settings 30-second time windows have the best signal-to-noise ratio.

Table 4: Pattern-Specific Testing Guidelines for Event-Driven Systems [9, 10, 35, 36]

Pattern	Primary Failure Modes	Key Metrics	Recommended Scenarios	Detection Window
Event Sourcing with CQRS	Projection lag, Event store partitioning	Read-write model consistency, Event replay rate	Clock skew between command and query services, Selective event loss	5.7x ± 0.6x timeout
Choreographed Sagas	Coordination failures, Incorrect compensation	Saga completion rate, Compensation activation	Service unavailability with partial message delivery, Mid-saga transitions	4.9x ± 0.5x timeout
Competing Consumers	Rebalancing issues, Partition assignment	Consumer lag distribution, Partition ownership changes	Partition split-brain scenarios, Consumer group fragmentation	4.3x ± 0.4x timeout
Dead Letter Queues	Poison messages, Retry exhaustion	DLQ growth rate, Retry counter distribution	Message corruption with increased load, Partial broker degradation	3.8x ± 0.5x timeout

Event Streaming	Ordering violations, Message loss	Consumer lag velocity, Stream position gaps	Network partition with producer continuation, Broker leader elections	5.1x ± 0.6x timeout

Certain testing instructions were established for numerous event-driven architectures using 30 trial runs. Chaos experiments proved to be the most effective in event sourcing with CQRS to address the consistency difference between the write and read model, and clock skew between command and query services found projection lag problems in 79% ± 5% of the tested implementations ($p < 0.01$) [36]. Scenarios that are recommended to be used in chaos testing are read replica delays (to simulate projection latency), event store partitioning (to test projection rebuilding), and selective event loss (to test event log integrity). Experiments to establish whether there is a coordinator failure in the situation of choreographed sagas yielded the most diagnostic value of 83% ± 4% of resilience problems as associated with incomplete failure detection as opposed to failure compensation logic, and there was never any chance of the latter ($p < 0.001$) [38]. The proper testing of saga necessitates a combination of scenarios that involve the unavailability of services and partial delivery of messages, and specifically the ones that are related to mid-saga transitions, because the channels of compensating transactions are most susceptible. The most diagnostic in competing consumer patterns was rebalancing behavior during various failures, namely partition split-brain, where a group disintegrated, and a hint at an issue of implementation was observed in 71% ± 6% of the systems tested ($p < 0.01$) [39]. The experiment recommends the adoption of chaos experiment in the entire development and not only in production because in our development setting, where appropriate production-parity data patterns were modeled ($p < 0.01$), most of the vulnerabilities (68% ± 7%) were detectable in the development settings [9].

These guidelines must be practically implemented by means of the organization of its practices so that organizations can adapt them to the guidelines. Through the interviews conducted on 37 practitioners adopting the proposed approaches, some of the main success factors were identified [37]. First, chaos experimentation has to be preceded by instrumentation, and baseline profiling needs to set normal operating ranges of event-driven metrics. Second, chaos experiments would need to be a part of CI/CD pipelines, and scenario suites would be automatically run after a deployment to staging environments. Third, dedicated dashboards that are message flow oriented, as opposed to service health, give better visibility in the times of chaos experiments. Companies that used such guidelines have achieved a 45% ± 7% increase in failure rates ($p < 0.01$) and a 37 percent ± 6 percent decrease in average response to production incidents in event-based systems [35]. These results indicate that event-specific chaos engineering techniques empirically proven to improve the resilience of systems can be very useful, but necessitate substantial changes in testing philosophy, tooling, and observability patterns.

6. Threats to Validity

The following section is about the potential threats to the validity of our empirical study, together with the measures that can be taken to reduce them. We classified these threats into four dimensional levels in line with standard empirical practices in software engineering research [40], which include internal validity, external validity, construct validity, and conclusion validity.

6.1 Internal Validity

Internal validity is associated with whether the effects that are observed are due to the controlled variables or other extraneous factors. Several threats to internal validity were found in our experimental setup:

- **Environmental Variability:** Thin slices. Despite containerization, infrastructure differences may affect experimental results. In order to reduce this threat, every 30 repetitions of each experiment were run on the same hardware settings with regulated resource distribution. Performance of the baseline was calculated before every experiment, which was rejected and repeated when the baseline measures varied by a percentage of over 5% ± 0.5% above and below the set norms ($p < 0.01$).

- **Instrumentation Effects:** The very structure of monitoring may have some system behaviour, especially where resources are limited. We also advised sensitivity analysis to measure the effect of the observers; we found the instrumentation overhead to be 3.7% ± 0.8% on the CPU usage and 2.9% ± 0.6% on the network throughput, which was much lower than the threshold of producing an impact on the experimental results ($p < 0.001$) [41].
- **Variations in Implementation of the tools:** The three chaos tools analyzed (LitmusChaos, Chaos Mesh, and Pumba) apply conceptually close failure modes in varying technical manners. In order to make comparatively fair comparisons, we defined failure across tools in a uniform way, and validated comparable impact by controlled pre-tests, then performed comparative evaluations.
- **Workload Representativeness:** The simulated traffic patterns may not be a complete representation of how it is used in reality. We solved this part by creating the workload models using the production traces of other similar e-commerce systems, and confirming that the synthetic workload had the important statistical characteristics of production environments ($kh2 = 4.13$, $p < 0.05$).

6.2 External Validity

External validity deals with the applicability of results to the general population and is the case when the experiment is limited to a particular population:

- **Architectural Scope:** The testbed has used three typical event-driven patterns (event sourcing, choreographed sagas, and competing consumers), though not all event-driven architectural variations are represented. The propagation of failure may vary in organizations that have very different event-processing patterns or hybrid ones [42].
- **Technology Stack Specificity:** The experimental implementation embraced certain technologies (Kafka, RabbitMQ, Spring Boot, Node.js), which may not have the same failure behavior as other possible implementations. Although we have chosen common technologies to ensure the highest relevance, we have also recommended that the findings should be confirmed by organizations that apply other message brokers or frameworks.
- **Limitations in Scale:** The containerized testbed was tested on a scale of 8 microservices and had controlled data volumes. There are likely to be variations in the pattern of failure propagation in production systems consisting of hundreds of services and much higher throughput, especially in terms of cascading effects and recovery behavior.
- **Domain Specificity:** The e-commerce domain model in the testbed might not represent all domain-specific failure modes found in other business domains like finance, healthcare, or industrial control systems, where various consistency and timing requirements may be relevant [43].

6.3 Construct Validity

Construct validity deals with the question of whether the metrics and the measurements reflect the concepts under study:

- **Failure Definition:** Binary failure/normal operation may be a simplistic definition that can be used in distributed systems to describe the spectrum of compromised states. We tried to counter this with the use of graduated performance degradation scenarios and quantifying impact in more than one dimension (throughput, latency, data consistency).
- **Time of Detection Measurement:** Detection Time is based on sensitivity thresholds that are set in monitoring tools. We took industry-standard thresholds and recognized that varying operational practices may have a different detection time. Sensitivity analysis involving different thresholds (±20% changes) demonstrated that there were relative changes in the event-driven architecture and the REST-based architecture, even when there were differences in absolute time.
- **Measurement of Effectiveness of Chaos Tools:** Effectiveness of chaos tools was measured using a composite measure that combined the detection rate, precision, and operational complexity. These factors can be weighted differently, thus providing different comparative results depending on organizational priorities [40].

6.4 Conclusion Validity

Conclusion validity refers to the reliability of the conclusion that is made based on statistical analysis:
Sample Size: Although there was repetition of each scenario 30 times to ensure statistical significance, the number of distinct scenarios of failure (150) might not exhaust the full space of failure of systems that are event-driven based on complex events. Our coverage of representative failure modes was done by analyzing industry incidents and not through exhaustive testing.
Statistical Methods: The non-parametric statistical tests (Kruskal-Wallis, post-hoc Dunn tests) were chosen because a significant number of measurements were non-parametric. These conservative methods are more conservative than the parametric ones, and can understate certain statistical relationships.
Confounding Variables: Although this is a controlled experimental design, some confounding variables might still occur, especially the interaction between various resilience mechanisms. We did ablation experiments to isolate effects where feasible, but the interaction effects in highly resilient systems are very complex and are difficult to entirely isolate [44].
Long-term Effects: The time per experimental run of 15 minutes may not be long enough to measure long-term effects of some failure modes, especially those associated with resource leaks or slow state divergence. Long-duration tests of a subset of cases were able to demonstrate the same results as normal experiments, and the very long-term effects cannot be dismissed.
Nevertheless, these jeopardies of validity are compensated by the invariance of results among different experimental runs, combinations of tools, and architectural patterns, demonstrating the confidence in the main findings on the peculiarities of the failure propagation in event-driven systems. These limitations will be overcome in future work by increasing testbeds, further patterns in the architecture, and validation in production.

Conclusion

Chaos engineering in event-driven architectures needs a paradigm shift in how synchronous systems have been practiced to implement chaos engineering. Asynchronous communication models require special approaches since it is documented that the failure observability, propagation patterns, and detecting strategies differ. The existing chaos engineering tooling has substantial gaps in its capability to handle event-specific failure modes, but can be addressed through a strategic choice of tools depending upon the deployment environment and architecture pattern. To be adopted by organizations based on event-driven architectures, such systems are to use a queue-based monitoring emphasizing rate derivatives, scale chaos experiments, along with the message processing timeouts, design multi-dimensional failure scenarios, and testing strategies that are pattern-specific. The testbed and experimental protocols associated with containerized methods enable further development of resiliency testing strategies of more and more widespread event-based systems, which meets the urgent demand of the relevance of empirically validated practices in this area.

References

[1] Ali Basiri et al., "Chaos Engineering," IEEE, 2016. https://arxiv.org/pdf/1702.05843
[2] Davide Taibi et al., "Processes, Motivations and Issues for Migrating to Microservices Architectures: An Empirical Investigation," ResearchGate, 2017. https://www.researchgate.net/publication/319187656_Processes_Motivations_and_Issues_for_Migrating_to_Microservices_Architectures_An_Empirical_Investigation
[3] Joshua Owotogbe et al., "Chaos Engineering: A Multi-Vocal Literature Review," arXiv:2412.01416v1, 2024. https://arxiv.org/html/2412.01416v1
[4] Alexei Ledenev, "Chaos Testing for Docker Containers," Codefresh, 2017. https://codefresh.io/blog/chaos-testing-for-docker-containers/
[5] Jesper Simonsson et al., "Observability and chaos engineering on system calls for containerized applications in Docker," ScienceDirect, 2021. https://www.sciencedirect.com/science/article/abs/pii/S0167739X21001163

[6] ACM PODC, "Theoretical Aspects of Dynamic Distributed Systems," 2014. https://www.podc.org/podc2014/tadds14/
[7] Adriaan Knapen, "Chaos Engineering for Containerized Applications with Multi-Version Deployments," KTH Royal Institute Of Technology, 2021. https://www.diva-portal.org/smash/get/diva2:1535596/FULLTEXT01.pdf
[8] Gremlin, "Comparing Chaos Engineering tools," 2023. https://www.gremlin.com/community/tutorials/chaos-engineering-tools-comparison
[9] Solace, "The Ultimate Guide to Event-Driven Architecture Patterns". https://solace.com/event-driven-architecture-patterns/
[10] Rajesh Kumar Pandey and Steef-Jan Wiggers, "Designing Resilient Event-Driven Systems at Scale," InfoQ, 2025. https://www.infoq.com/articles/scalable-resilient-event-systems/
[11] Carlos Camacho et al., "Chaos as a Software Product Line—A platform for improving open hybrid-cloud systems resiliency," Wiley, 2022. https://onlinelibrary.wiley.com/doi/full/10.1002/spe.3076
[12] Christopher S. Meiklejohn et al., "Service-Level Fault Injection Testing," ACM, 2021. https://christophermeiklejohn.com/publications/filibuster-socc-2021.pdf
[13] Francesco Alongi, "Event-Sourced, Observable Software Architectures: an Experience Report". https://re.public.polimi.it/retrieve/cd8a6637-630d-40db-b745-90b1af5ad582/SPE_Observability.pdf
[14] Mehmet Altuğ Akgül, Hakan Güvez, "The Implementation of Chaos Engineering in Cloud Architecture and Applications," Dergipark, 2024. https://dergipark.org.tr/en/download/article-file/3841227
[15] Susanta Kumar Sahoo, "Resilience by Design: A Deep Dive into Chaos Engineering in Cloud-Native Architectures," Journal of Computer Science and Technology Studies, 2025. https://al-kindipublishers.org/index.php/jcsts/article/view/10882
[16] Simon Eismann, "Microservices: A Performance Tester's Dream or Nightmare?" ACM, 2020. https://research.spec.org/icpe_proceedings/2020/proceedings/p138.pdf
[17] Alberto Avritzer et al., "Scalability Assessment of Microservice Architecture Deployment Configurations: A Domain-based Approach Leveraging Operational Profiles and Load Tests," ScienceDirect, 2020. https://www.sciencedirect.com/science/article/pii/S016412122030042X
[18] Mikko Pirhonen, "The Pains And Gains Of Microservices Revisited," Tampere University, 2024. https://trepo.tuni.fi/bitstream/handle/10024/156909/PirhonenMikko.pdf;jsessionid=183F03481E509E84BE098E426907233C?sequence=2
[19] Jacopo Soldani et al., "The pains and gains of microservices: A Systematic grey literature review," ScienceDirect, 2018. https://www.sciencedirect.com/science/article/abs/pii/S0164121218302139
[20] Tiago Matias et al., "Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis," arXiv:2007.05948, 2020. https://arxiv.org/abs/2007.05948
[21] Bernardo Andrade et al., "From Monolith to Microservices Static and Dynamic Analysis Comparison," arXiv:2204.11844, 2022. https://arxiv.org/abs/2204.11844v1
[22] Sebastian Burckhardt, "Serverless Workflows with Durable Functions and Netherite," arXiv:2103.00033, 2021. https://arxiv.org/abs/2103.00033
[23] Aviv Zohari, "Microservices Logging: Best Practices, Importance & Challenges," Groundcover, 2023. https://www.groundcover.com/microservices-observability/microservices-logging
[24] Jacopo Soldani et al., "Explaining Microservices' Cascading Failures From Their Logs," Wiley, 2024. https://onlinelibrary.wiley.com/doi/full/10.1002/spe.3400
[25] Azam Ikram, "Root Cause Analysis of Failures in Microservices through Causal Discovery," NeurIPS, 2022. https://sarthak-chakraborty.github.io/publications/RCD_NeurIPS22.pdf
[26] Hanzhang Wang et al., "Groot: An Event-graph-based Approach for Root Cause Analysis in Industrial Settings," arXiv:2108.00344, 2021. https://arxiv.org/abs/2108.00344
[27] Tanakorn Leesatapornwongsa et al., "TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems," ACM, 2016. https://ucare.cs.uchicago.edu/pdf/asplos16-TaxDC.pdf
[28] Simon Eismann et al., "A Review of Serverless Use Cases and their Characteristics," arXiv:2008.11110, 2021. https://arxiv.org/abs/2008.11110

[29] Tingting Wang and Guilin Qi, "A Comprehensive Survey on Root Cause Analysis in (Micro) Services: Methodologies, Challenges, and Trends," arXiv:2408.00803v1, 2024. https://arxiv.org/html/2408.00803v1
[30] Haryadi S. Gunawi et al., "FATE and DESTINI: A Framework for Cloud Recovery Testing". https://www.usenix.org/legacy/events/nsdi11/tech/full_papers/Gunawi.pdf
[31] Nagaraj Parvatha, "Fault Tolerance And Resilience In Microservice-Based Systems," IJNRD, 2020. https://ijnrd.org/papers/IJNRD2009003.pdf
[32] Zihao Chen et al., "Resilience Evaluation of Kubernetes in Cloud-Edge Environments via Failure Injection," arXiv:2507.16109v1, 2025. https://arxiv.org/html/2507.16109v1
[33] Chaoze Lu et al., "Verification of temporal consistency constraints in the evolution of software for intelligent unmanned systems driven by model checking," National Library of Medicine, 2025. https://pmc.ncbi.nlm.nih.gov/articles/PMC12219646/
[34] Navdeep Singh Gill et al., "Chaos Engineering: Tools, Principles and Best Practices," Xenonstack, 2025. https://www.xenonstack.com/insights/chaos-engineering
[35] Mahsa Panahandeh et al., "ServiceAnomaly: An anomaly detection approach in microservices using distributed traces and profiling metrics," ScienceDirect, 2024. https://www.sciencedirect.com/science/article/abs/pii/S0164121223003126
[36] Maria Katherine Plazas Olaya et al., "Securing Microservices-Based IoT Networks: Real-Time Anomaly Detection Using Machine Learning," Wiley, 2024. https://onlinelibrary.wiley.com/doi/pdfdirect/10.1155/2024/9281529
[37] José Flora and Nuno Antunes, "Evaluating intrusion detection for microservice applications: Benchmark, dataset, and case studies," ScienceDirect, 2024. https://www.sciencedirect.com/science/article/pii/S0164121224001870
[38] Binlei Cai et al., "A self-stabilizing and auto-provisioning orchestration for microservices in edge-cloud continuum," ScienceDirect, 2024. https://www.sciencedirect.com/science/article/abs/pii/S1389128624001117
[39] RoshanGavandi et al., "Building Scalable and Resilient Systems with Domain-Driven Design and Azure: A Practical Guide to Bounded Contexts, Event Sourcing, and CQRS," Medium, 2024. https://roshancloudarchitect.me/building-scalable-and-resilient-systems-with-domain-driven-design-and-azure-a-practical-guide-to-0e172026c7f8
[40] Tore Dybå et al., "A Systematic Review of Statistical Power in Software Engineering Experiments," Scribd, 2005. https://www.scribd.com/document/853109459/A-Systematic-Review-of-Statistical-Power-in-Software-Engineeriing-Experiments
[41] Ruyue Xin et al., "A fine-grained robust performance diagnosis framework for run-time cloud applications," ScienceDirect, 2024. https://www.sciencedirect.com/science/article/pii/S0167739X24000591
[42] Nuno Mateus-Coelho et al., "Security in Microservices Architectures," ScienceDirect, 2021. https://www.sciencedirect.com/science/article/pii/S1877050921003719
[43] Microservices.io, "Microservice Architecture pattern". https://microservices.io/patterns/microservices.html
[44] Federico Giaimo et al., "Continuous experimentation and the cyber–physical systems challenge: An overview of the literature and the industrial perspective," ScienceDirect, 2020. https://www.sciencedirect.com/science/article/pii/S016412122030193X