

Chaos Engineering: A Multi-Vocal Literature Review

JOSHUA OWOTOGBE, INDIKA KUMARA, and WILLEM-JAN VAN DEN HEUVEL, Jheronimus Academy of Data Science, Netherlands and Tilburg University, Netherlands

DAMIAN ANDREW TAMBURRI, Jheronimus Academy of Data Science, Netherlands and University of Sannio, Italy

Organizations, particularly medium and large enterprises, typically rely heavily on complex, distributed systems to deliver critical services and products. However, the growing complexity of these systems poses challenges in ensuring service availability, performance, and reliability. Traditional resilience testing methods often fail to capture the intricate interactions and failure modes of modern systems. Chaos Engineering addresses these challenges by proactively testing how systems in production behave under turbulent conditions, allowing developers to uncover and resolve potential issues before they escalate into outages. Though chaos engineering has received growing attention from researchers and practitioners alike, we observed a lack of reviews that synthesize insights from both academic and grey literature. Hence, we conducted a Multivocal Literature Review (MLR) on chaos engineering to address this research gap by systematically analyzing 96 academic and grey literature sources published between January 2016 and April 2024. We first used the chosen sources to derive a unified definition of chaos engineering and to identify key functionalities, components, and adoption drivers. We also developed a taxonomy for chaos engineering platforms and compared the relevant tools using it. Finally, we analyzed the current state of chaos engineering research and identified several open research issues.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: Chaos Engineering, Fault Injection, Distributed System, Resilience, Multivocal Literature Review, Microservice

ACM Reference Format:

Joshua Owotogbe, Indika Kumara, Willem-Jan van den Heuvel, and Damian Andrew Tamburri. 2025. Chaos Engineering: A Multi-Vocal Literature Review. 1, 1 (June 2025), 35 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 Introduction

Cloud-based applications are being adopted across various domains, including healthcare, finance, construction, education, and defense [6, 20, 43]. While cloud computing promises to offer benefits such as cost savings, flexibility, and faster time to market [29], the growing complexity of cloud infrastructure poses a significant challenge to maintaining dependable systems, leading to an increased risk of technical failures and costly downtime [5, 106, 162, 183]. The consequences of these failures can be severe, as seen in significant financial losses incurred for companies, including the 2007-2013 outages of 38 cloud service providers, resulting in \$669 million in losses, and Facebook's six-hour outage in 2021, which resulted in an estimated \$160 million loss for the global economy [69]. The 2017 ITIC (Information Technology Intelligence Consulting) survey also reported that 98% of organizations experienced downtime with financial losses exceeding \$100,000 per hour [79].

An effective approach to overcoming the challenges of ensuring reliable and resilient cloud-based systems is chaos engineering, which proactively tests system resilience by introducing controlled failures into production environments [21, 53, 132]. This approach was pioneered by Netflix in 2008 to fortify its video streaming infrastructure. Since then, chaos engineering has grown and extended its influence, evolving from its original focus on cloud services and microservice architecture to broader applications across industries [15]. Today, this approach is widely employed in critical sectors, such as fintech and healthcare, with major technology companies like LinkedIn, Facebook, Google, Microsoft, Amazon, and Slack adopting it to enhance system resilience, improve customer experience, and drive business growth. [5, 15, 25, 53, 79, 136, 147].

We observed that many researchers and practitioners have explored chaos engineering in various contexts, discussing its implementations and challenges through research papers and grey literature such as online blogs, technical papers, and

Authors' Contact Information: Joshua Owotogbe, j.s.owotogbe@tilburguniversity.edu; Indika Kumara, i.p.k.weerasinghadewage@tilburguniversity.edu; Willem-Jan van den Heuvel, w.j.a.m.vdnHeuvel@tilburguniversity.edu, Jheronimus Academy of Data Science, 's-Hertogenbosch, North Brabant, Netherlands and Tilburg University, Tilburg, North Brabant, Netherlands; Damian Andrew Tamburri, datamburri@unisannio.it, Jheronimus Academy of Data Science, 's-Hertogenbosch, North Brabant, Netherlands and University of Sannio, Benevento, Benevento, Italy.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

Manuscript submitted to ACM

forums. However, despite its increasing adoption, much of the existing knowledge remains dispersed across the literature. To the authors' best understanding, no secondary research has been published to synthesize this scattered information, such as comprehensive reviews or survey papers. Hence, to address this research gap, in this paper, we present a Multivocal Literature Review (MLR) [63] on chaos engineering, synthesizing a broad spectrum of perspectives from academic and grey literature sources. Unlike traditional Systematic Literature Reviews (SLRs), which often overlook grey literature, an MLR includes insights from peer-reviewed academic studies alongside industry reports, technical blogs, and white papers [62, 63]. In particular, as chaos engineering emerged from the industry and most tools are from the industry, the grey literature is a highly relevant and valuable source of information. In this MLR, we selected and analyzed 96 sources, encompassing both academic and grey literature. We selected 2016 as the starting point, when chaos engineering principles were first formally articulated by Basiri et al [17]. Our findings provide a detailed overview of chaos engineering, including its core principles, quality requirements, and foundational components defined by researchers and industry experts. We also examine the motivations behind adopting chaos engineering, the specific resilience challenges it addresses, and the practices and evaluation methods that support its effective implementation. Additionally, we introduce a taxonomy of chaos engineering platforms and their capabilities, designed to support informed decision-making around tool adoption and deployment strategies. By analyzing the benefits and shortcomings of current approaches in the chaos engineering domain, this MLR contributes to a thorough understanding of chaos engineering and provides guidelines for future studies to advance this critical field.

The remainder of the paper is organized as follows. Section 2 outlines the background and motivation for this review, including a discussion of related surveys, the distinction between chaos engineering and traditional testing, and the need to extend the existing body of knowledge on chaos engineering. Section 3 presents the research methodology used in this study, while Section 4 provides a comprehensive definition of chaos engineering, along with its functionalities, quality requirements, and core components. Section 5 focuses on the motivations for chaos engineering and the specific challenges it addresses. Section 6 introduces a taxonomy of chaos engineering, covering tools, practices, and evaluation methods within the field. Section 7 examines the current state of chaos engineering in research and industry. Section 8 discusses the implications of our findings, highlighting key insights, identifying open issues, and addressing potential threats to validity. Finally, Section 9 concludes the paper.

2 Background and Motivation

This section outlines the motivations for conducting this review by examining the limitations of existing literature (Section 2.1), explaining how chaos engineering diverges from traditional testing methods (Section 2.2), and establishing the novelty and timeliness of this multivocal literature review (Section 2.3).

2.1 Related Work and Survey Gap

In this section, we consider key surveys on resilience, robustness, and fault injection techniques, exploring their relationship to the proactive focus of chaos engineering. For instance, Natella et al. [125] surveyed software fault injection techniques, categorizing faults by origin such as software, hardware, and environment, and by injection approach such as code changes or fault effect injection including state and interface errors. However, their focus is on pre-deployment environments (e.g., simulators, testbeds), whereas chaos engineering involves live, iterative experiments in production-like systems. Similarly, Colman-Meixner et al. [40] focus on cloud resilience through redundancy and failover, but explore reactive recovery rather than proactive experimentation, as in chaos engineering. Laranjeiro et al. [102] consider static robustness techniques such as fuzzing and API testing, which lack real-time, adaptive feedback loops. Ruholamini et al. [148] surveyed self-healing in cloud-native systems, emphasizing failure detection and recovery rather than proactive failure exploration.

More recently, Arsecularatne and Wickramarachchi [11] conducted a systematic review and qualitative study on the adoptability of chaos engineering in DevOps. Their work highlights organizational challenges and potential benefits but does not provide a structured analysis of tools, architectures, or evaluation metrics. Similarly, Mhatre et al. [112] examined chaos engineering within DevSecOps and SRE (Site Reliability Engineering) contexts, emphasizing conceptual alignment and use cases, but without offering a comprehensive taxonomy or resilience assessment frameworks. In summary, existing reviews examine fault injection or discuss chaos engineering within DevOps contexts. However, they offer little or no clear insight into its unified definition, functionalities, architectural components, quality attributes, comprehensive tool taxonomy, or the technical and organizational challenges of implementing it in production-like environments. This review addresses these gaps by presenting the first multivocal literature review (MLR) on chaos engineering, integrating both academic and industry sources to inform future research and practical adoption.

2.2 Chaos Engineering vs. Traditional Software Testing Techniques

Traditional software testing techniques, including unit, integration, stress, and robustness testing, primarily focus on validating correctness and performance under expected conditions in pre-deployment environments [18, 154]. These approaches typically operate in isolated, controlled settings and verify that system components behave correctly with known inputs [84, 154]. However, in modern distributed, cloud-native systems, such methods often fail to expose cascading or emergent failures caused by asynchronous interactions and dynamic runtime behaviors [55, 84, 114]. Chaos engineering addresses these limitations by introducing controlled faults into production or production-like systems to explore system resilience [55, 172, 194]. Rather than validating functionality using predefined test scripts, chaos experiments simulate failures, such as service crashes, latency spikes, or regional outages, to evaluate system behavior and recovery under adverse runtime conditions [114, 154, 194]. This methodology helps surface hidden vulnerabilities that traditional test environments often miss [84, 135]. Importantly, chaos engineering is not a replacement for conventional testing, but a complement. While traditional methods confirm system correctness under expected conditions, chaos engineering validates system behavior under known faults and real-world failure scenarios [18, 55]. This paradigm shift—from pre-deployment pass/fail validation to post-deployment resilience assessment—is what makes chaos engineering essential for today’s complex, distributed infrastructure [55, 84, 194]. To clarify these distinctions, Table 1 highlights key differences between traditional testing and chaos engineering. The comparison focuses on three main areas: the testing environment (pre-production vs. production-like systems), the primary objective (functional verification vs. uncovering hidden system weaknesses), and the types of failures addressed (expected vs complex real-world faults).

Table 1. Comparison Between Traditional Software Testing Techniques and Chaos Engineering

Category	Traditional Testing	Chaos Engineering	Articles Referenced
Testing Environment	Pre-production or simulated systems.	Production or production-like systems.	[15, 21, 22, 68, 98, 156, 188, 193, 194]
Testing Objective	Verifies specific functions or components against pre-defined requirements and standards.	Identifies obscure system weaknesses that may only become apparent during turbulent failure conditions.	[15, 79, 172]
Types of Failures	Focuses on expected and known usage scenarios and errors to verify how the system behaves under normal or predictable conditions.	Simulates real-world failures (e.g., network outages, high load, hardware issues) to see how the system reacts to these changes.	[15, 49, 84, 94, 115, 139, 172]

2.3 Research Contribution and Novelty

Despite the growing popularity of chaos engineering [118], there remains a lack of unified academic frameworks that synthesize its principles, classify its tool landscape, or integrate its practices across both research and industry domains. To the best of our knowledge, this paper presents the first Multivocal Literature Review (MLR) on chaos engineering, integrating both academic studies and grey literature. Our contributions are fourfold. First, we synthesize and unify definitions, core principles, functionalities, and architectural components of chaos engineering from academic and industry perspectives (**RQ1**). Second, we identify the key technical and socio-technical challenges driving its adoption (**RQ2**). Third, we propose a taxonomy of chaos engineering tools, classify evaluation methods, and highlight adoption practices (**RQ3**). Fourth, we analyze publication trends, venue types, and key contributors to assess the current state of research and practice (**RQ4**). Together, these contributions provide a foundational reference on chaos engineering that bridges research and practice, supports rigorous evaluation of chaos engineering tools, enables their systematic adoption in industry, and offers directions for future research.

3 Research Methodology

We employed the guidelines proposed by Garousi et al. [63] to systematically collect and analyze multivocal literature on chaos engineering. We also drew methodological inspiration from other multivocal literature reviews, such as those by Buck et al. [26] and Islam et al. [80]. Figure 1 illustrates our MLR methodology, comprising several steps divided into three phases: planning and designing the review protocol, executing the review, and documenting the findings. The remainder of this section provides a detailed discussion of each step.

3.1 Research Identification

We employed a search strategy to identify appropriate literature grounded in a series of research questions in Table 2. The first three questions cover the various aspects of chaos engineering, including its definitions, key features, benefits, challenges, use cases, tools and techniques, and best and bad practices. The final question provides an overview of the present landscape of chaos engineering literature.

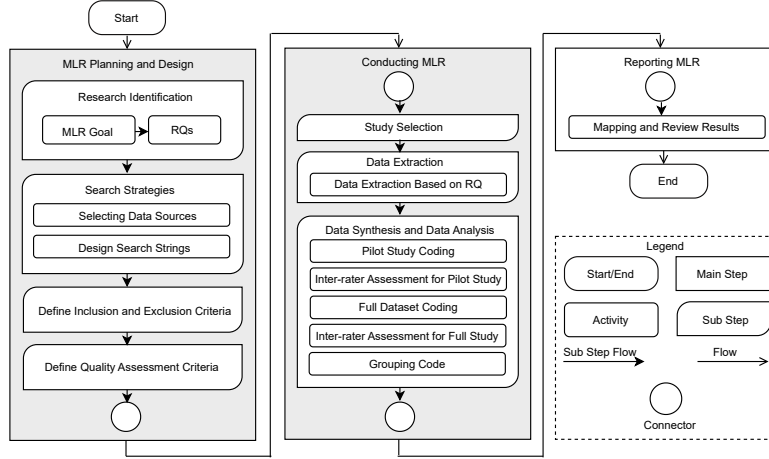


Fig. 1. Overview of the MLR Process [80].

Table 2. Research Questions and Motivation

Research Questions	Motivation
RQ1. How is chaos engineering defined in the literature? <i>RQ1.1.</i> Which core activities and primary functionalities are most commonly associated with chaos engineering? <i>RQ1.2.</i> What are the core components of a chaos engineering platform, and how do they support its core functions? <i>RQ1.3.</i> What quality requirements are essential for an effective chaos engineering platform, and how are they quantified?	This question clarifies the definition of chaos engineering, exploring its core activities, key components, and quality requirements. These elements build a foundation for understanding, implementing, and evaluating chaos engineering solutions.
RQ2. What challenges does chaos engineering intend to solve?	This question helps identify key challenges, guiding organizations to areas where chaos engineering can best enhance resilience and reliability.
RQ3. What kinds of solutions have been proposed for chaos engineering? <i>RQ3.1.</i> What tools and techniques are utilized, proposed, designed, and implemented by researchers and practitioners? <i>RQ3.2.</i> What best and bad practices are documented in adopting chaos engineering? <i>RQ3.3.</i> How are the chaos engineering approaches evaluated?	This question examines the tools and techniques in chaos engineering, highlighting key practices, common pitfalls, and methods for evaluating and enhancing solutions across various contexts.
RQ4. What is the state of chaos engineering in research and industry? <i>RQ4.1.</i> What is the extent and diversity of research conducted on chaos engineering, and what trends can be observed over time? <i>RQ4.2.</i> Who are the thought leaders and key contributors in the field of chaos engineering, and what intersections of thought do they represent?	Exploring this question allows us to assess the growth and direction of chaos engineering, highlighting current trends, influential contributors, and areas that need further exploration, ensuring continued advancement in the field.

3.2 Search Strategy

This section outlines the search strategy, detailing the data sources, queries, and the complete search process.

3.2.1 Sources of data. We conducted automated searches in four major digital libraries—Scopus, IEEE Xplore, ACM Digital Library, and Web of Science—using both peer-reviewed scholarly articles and grey literature as sources. These libraries serve as primary sources for software engineering studies [39]. Following the formal articulation of chaos engineering principles by Basiri et al. in 2016 [17], we limited our search to publications from January 2016 to April 2024. We performed multi-keyword searches over titles, abstracts, and keywords within this range. Additionally, Google Scholar was used to pinpoint any relevant

literature that was not included in the previous searches. We used Google Search for grey literature, similar to other multi-vocal literature reviews (MLRs) [61, 64, 80].

3.2.2 Search Terms. Following the SLR guidelines [92], we constructed search terms based on the research questions. The first two authors tested the search strings to assess their applicability for addressing the defined research questions. Multiple pilot searches were conducted to refine the search terms, ensuring the inclusion of all relevant papers—boolean operators (AND, OR) combined keywords, synonyms, and related concepts. Table 3 shows the final search strings used across the four digital libraries. We adapted and applied the Google Search search string “(Chaos AND Engineering)” for grey literature to capture relevant non-peer-reviewed sources.

Table 3. Search Strings and Results.

Search Strings	ACM DL	IEEE Xplore	Scopus	Web of Science	Total
"chaos engineering" OR "chaos test" OR "chaos experiment" OR "chaos toolkit" OR "chaos platform" OR "chaos mesh" OR "chaos integration" OR "chaos architecture" OR "chaos infrastructure" OR "chaos injection" OR "chaos tool"	108	81	142	57	388
"chaos engineering" AND ("trends" OR "developments" OR "progression")	30	0	20	0	50
"chaos engineering" AND ("leader" OR "pioneer" OR "expert" OR "contributor")	39	3	4	0	46
"chaos engineering" AND ("phases" OR "stages" OR "steps" OR "pipeline")	71	6	11	1	89
"chaos engineering" AND ("challenges" OR "issues" OR "problems")	91	19	37	4	151
"chaos engineering" AND ("applications" OR "benefits")	89	22	43	9	163
"chaos engineering" AND ("solutions" OR "approaches" OR "methods")	84	22	60	4	170
"chaos engineering" AND ("tools" OR "techniques")	92	30	42	10	174
"chaos engineering" AND ("best practices" OR "bad practices" OR "implementation" OR "Culture")	77	9	17	5	108
"chaos engineering" AND ("evaluation" OR "metrics")	85	25	28	4	142
Total	766	217	404	94	1481

3.3 Eligibility Criteria

Following some existing MLR studies [27, 63, 80], we applied a specific set of inclusion and exclusion criteria, as detailed in Table 4

Table 4. Inclusion and Exclusion Criteria.

Inclusion Criteria	Exclusion Criteria
IC1: Articles must be in English and be readily accessible in full text.	EC1: Short articles or brief papers (less than six pages).
IC2: Articles must include clear validation or evaluation methods, with supporting references.	EC2: Articles that lack the relevant focus on chaos engineering.
IC3: Articles that report practices and challenges in implementing chaos engineering, including case studies, best practices, or guidelines.	EC3: Articles that predominantly discuss physical infrastructure or purely hardware aspects without direct connection to chaos engineering.
IC4: Articles that provide insights into the trends and evolution of chaos engineering practice.	EC4: Studies enhancing specific algorithms or features of a single tool without a broader chaos engineering context unless they provide insights into the field's evolution or state of tools.
IC5: Articles that identify key contributors and diverse perspectives within chaos engineering.	EC5: Duplicate articles from all sources.
IC6: Articles must cover all identified phases of the chaos engineering pipeline, ensuring a holistic view of the field.	EC6: Articles lacking empirical evidence or practical applications to support theoretical propositions.

3.4 Study Selection

The libraries' advanced query modes were used to maximize search coverage. The search was conducted in April 2024, and the search results were exported from each database in BibTeX and CSV formats for reference management and data processing. Duplicates were removed before proceeding with further analysis. Figure 2 illustrates the selection process for academic and grey literature, showing the databases searched and the number of selected papers at each stage. Different approaches were followed to select grey and academic literature.

3.4.1 Academic Literature Selection. Searches in Scopus, IEEE Xplore, ACM DL, and Web of Science yielded 1,481 papers. After duplicate removal, 374 unique papers remained. Applying the inclusion and exclusion criteria, the selection was narrowed to 47 relevant papers. An additional search was conducted using Google Scholar, which identified two more articles, bringing the total to 49 papers.

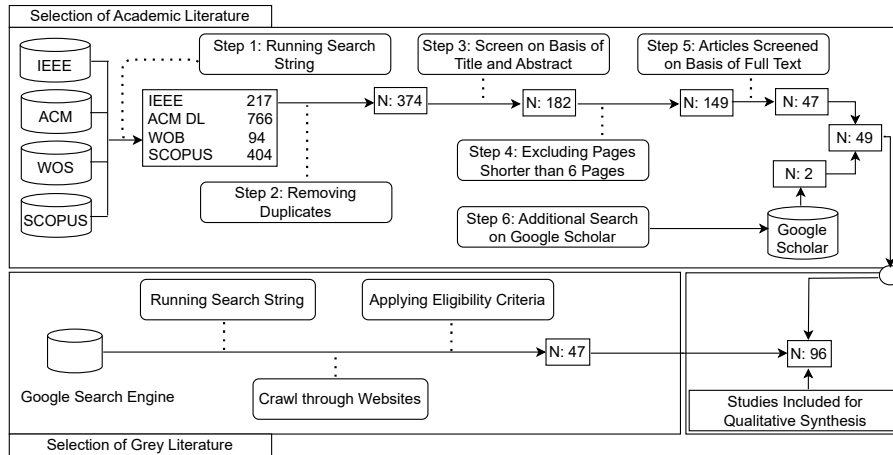


Fig. 2. The MLR study selection process. Steps are shown sequentially for both academic and grey literature. After final screening, inter-rater reliability was assessed using Cohen’s Kappa [101], yielding $\kappa = 0.826$ for academic and $\kappa = 0.857$ for grey literature, indicating strong reviewer agreement.

Table 5. Criteria for Qualitative Assessment of Grey Literature.

Category	Questions	Criteria Score
Authority	(Q1). Is the organization publishing the work credible?	2/3
	(Q2). Are there other publications by the author in this domain of study?	
	(Q3). How closely is the author connected to a reputable institution?	
Methodology	(Q4). Is the objective of the source evident and well-stated?	2/2
	(Q5). Can the author be considered an expert in this field?	
Date	(Q6). Is the publication date of the source within the past eight years?	1/1

3.4.2 Grey Literature Selection. We followed the approach of other multi-vocal and grey literature reviews [27, 61, 63, 64, 80] and used Google Search to gather grey literature. As described by these reviews, we examined the first ten pages of results, as this was deemed sufficient to identify the most pertinent literature since Google’s algorithm emphasizes the most relevant results on the first pages [80, 160, 178]. We continued scanning further only if necessary, stopping when no relevant new articles appeared [80]. To avoid bias, we searched in incognito mode, following the recommendations of Rahman et al. [142]. We assessed the quality and relevance of the primary grey literature sources we obtained, as it cannot be assumed that their quality is guaranteed. The quality assessment criteria suggested by Garousi et al[64] have been used for this purpose (see Table 5). Those criteria comprise three quality categories, ranging from the authority of the producer to the publication date (see the first column of Table 5). These categories encompass six criteria (see the second column) that were assessed individually for each grey literature item. The evaluation was conducted independently by the first two authors of this study, who indicated whether each item (a) met or (b) failed to meet a criterion. In cases where a criterion could not be assessed, such as when author or affiliation details were missing, it was treated as not satisfied. The final column of Table 5 specifies how many of the criteria needed to be satisfied for inclusion. For example, an item had to meet at least 2 out of 3 criteria related to the authority of the source to be considered valid. To further ensure internal agreement and consistency in applying these quality checks, all disagreements between reviewers were resolved through discussion. As a result of this procedure, 47 grey literature sources were selected. These, together with 49 academic papers, resulted in a total of 96 sources used for data extraction and synthesis (Figure 2). The distribution of selected sources across venues is summarized in Table 6.

3.5 Data Extraction, Synthesis, and Analysis

In alignment with the MLR process, we read and assessed the selected papers, extracting and summarizing data to address our research questions.

Table 6. Selected Study for Data Synthesis and Qualitative Evaluation.

Academic Literature	Grey Literature	
	Websites & Blogs	White Papers
[5, 7, 8, 17–19, 32, 33, 36, 37, 42, 46, 55–58, 78, 83, 84, 86, 89, 93, 94, 108, 109, 111, 124, 127, 128, 134, 139–141, 146, 156, 157, 159, 161, 163, 172–174, 180, 191, 192, 194, 195] = 49	[1, 2, 10, 14, 15, 21, 22, 25, 28, 41, 49, 50, 53, 67–69, 73, 75, 76, 79, 82, 85, 87, 96–98, 103, 104, 110, 115, 119, 120, 126, 129, 136, 147, 151, 152, 171, 179, 181, 185, 188, 190] = 44	[132, 164, 170] = 3

3.5.1 Data Extraction. To systematically collect essential details, such as author names, paper titles, publication venues, and years, we used a structured data extraction form based on guidelines from Kitchenham et al. [91] and Garousi and Felderer [60]. This information was organized in a spreadsheet.

3.5.2 Synthesis and Analysis. We applied thematic analysis to synthesize insights across our research questions, ensuring that coding remained objective and independent of the data extraction structure or biases [165]. The full papers were subsequently imported into ATLAS.ti¹, a qualitative data analysis tool that allowed for direct text coding. This process adhered to established thematic analysis steps [24], providing consistency and depth in our approach. We used a step-by-step approach to systematically identify codes, clusters, and classifications related to our research questions [100]:

- **Familiarization with the Data:** We began by reading the selected papers to understand the data comprehensively, enabling us to identify the key elements relevant to our research questions.
- **Pilot Study:** We randomly selected twenty sources for initial analysis, with the first author applying structural and descriptive coding [149] to conceptualize data relevant to the research questions [100]. The process began by extracting quotations—direct statements from the primary studies—and identifying keywords that captured their core meaning. For example (see Figure 3), from quotations referring to the need to introduce failure scenarios and monitor experiment duration, the keywords *Introduce failure scenarios* and *Experiment duration* were extracted. These were then grouped to form the code *Experiment Execution*. Similarly, the keywords *Monitoring sidecar* and *Observability for diagnosis* were extracted and abstracted into the code *Monitoring and Observability*. These codes were clustered into the broader theme *Functional Requirements*. This phase yielded 536 initial codes and an initial set of 28 candidate themes, which served as the foundation for subsequent refinement.
- **Inter-Rater Consensus Analysis for the Pilot Study:** After completing the pilot coding, the second author reviewed the full set of codes independently, assessing the naming, scope, and categorization of each code. The discrepancies between authors were discussed and resolved collaboratively. This consensus process refined the initial 536 codes to a validated set of 421 codes, improving the clarity and consistency of the thematic structure.
- **Complete Dataset Coding:** The remaining sources were subsequently coded using the validated codebook. The first author conducted the full-dataset coding, and the second author reviewed the outputs. Discrepancies identified during this phase were resolved collaboratively.
- **Inter-Rater Consensus Analysis for the Complete Study:** After the complete coding process, a final inter-rater assessment was conducted to ensure consistent application of the codes across the dataset. The discrepancies in interpretation were discussed and resolved. During this phase, the initial 28 themes were critically reviewed for internal coherence and conceptual distinctiveness. Overlapped or weakly supported themes were consolidated or removed, resulting in a final set of 15 distinct, non-overlapping themes. At this stage, each finalized theme was explicitly mapped to the specific research question(s) it addressed (e.g., RQ1.1 – Functional Requirements, RQ3.2 – Best Practices), ensuring that the thematic framework remained purposeful and traceable throughout the analysis.
- **Generating the report:** The results of this thematic analysis are presented in Sections 4, 5, 6, 7 and 8.

3.6 Replication Package

For the validation and reproducibility of this work, we have provided an extensive replication package online². This includes the complete compilation of sources, search queries, and thematic analysis performed using Atlas.ti (such as codes, groupings, quotations, and analyses), as well as the associated code tables.

¹<https://atlasti.com/>

²<https://doi.org/10.5281/zenodo.15696868>

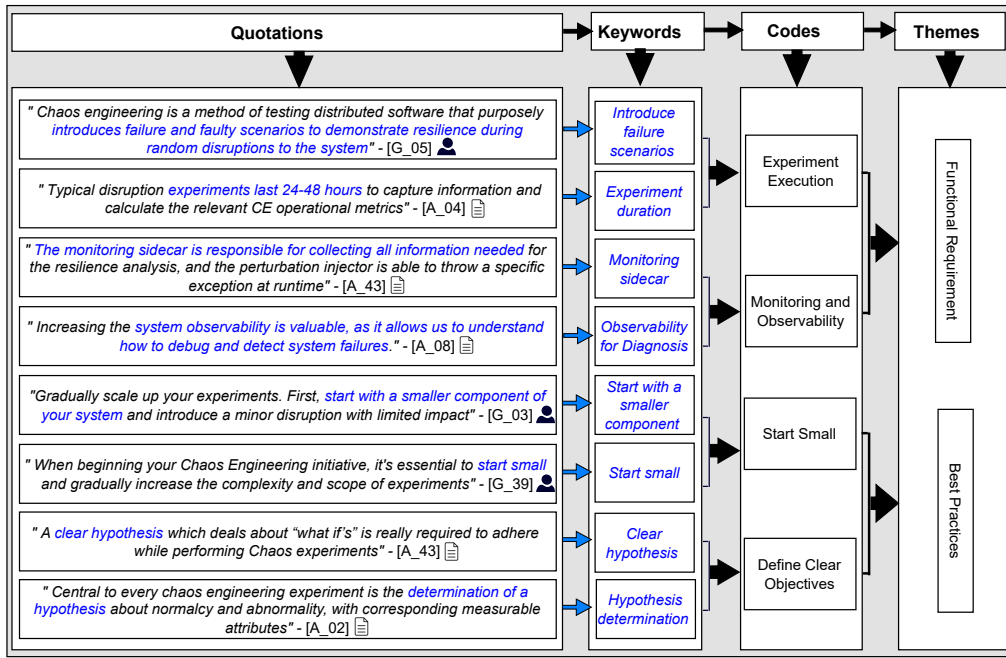


Fig. 3. A representative output from our thematic analysis process, showing how quotations from academic and grey literature were distilled into keywords, abstracted into codes, and grouped under broader themes. (👤: grey literature; 📄: academic literature)

4 Chaos Engineering: Definitions, Functionaries, and Elements (RQ1)

This section first reviews how chaos engineering is defined across academic and industry sources. Next, it outlines its core activities and commonly associated functionalities (RQ1.1). Finally, it introduces the essential components (RQ1.2) and quality requirements (RQ1.3) that form the foundation for implementing effective chaos engineering platforms.

4.1 Definitions

Chaos engineering has garnered significant attention [118] in both academia and industry, yet its definition remains fragmented and inconsistent. To address this, we analyzed a set of definitions identified during our thematic analysis process. These include representative definitions from each, summarized in Tables 7 and 8. Across industry and academic sources, several key elements consistently emerge: **system resilience**, **failure injection**, **controlled experiments**, and **testing in production or production-like environments**. These recurring elements highlight chaos engineering's focus on deliberately introducing failures in a controlled manner to assess how systems behave under stress and recover from disruptions.

Table 7. Practitioners' Definitions of Chaos Engineering

Definition	Articles Referenced
"The practice of deliberately breaking systems in controlled ways to identify potential failure points before they occur naturally."	[1, 73, 98, 133, 171, 179]
"The methodology of creating disruptive events and stressing applications to observe system responses and implement targeted improvements."	[15, 21, 68, 85]
"The technique of gauging system resiliency by intentionally causing failures under load to test self-healing mechanisms."	[15, 69, 85, 96, 136]

Table 8. Academic Definitions of Chaos Engineering

Definition	Articles Referenced
"A disciplined practice of experimenting on a software or distributed system to build confidence in its ability to withstand turbulent or unexpected conditions in production."	[5, 7, 17, 55, 83, 124, 139, 163, 172, 179, 180]
"An empirical and scientific approach for learning how software systems behave under stress by conducting controlled experiments to evaluate and improve their resilience and fault-handling mechanisms."	[78, 109, 161, 194]
"A resilience-testing approach involving the intentional injection of failures or disruptions into a system—especially in production or production-like environments—to uncover weaknesses and improve fault tolerance."	[84, 89, 94, 140]

We followed the synthesis method proposed by Gong and Ribiere [70], drawing on conceptual evaluation criteria from Wacker [182] and Suddaby [166] (see our online appendix Section 3.6 for details). A total of 48 definitions were analyzed, 27 from academic sources and 29 from grey literature. Based on this evaluation, 34 definitions were retained and 22 were excluded due to issues such as vagueness, conceptual overlap, or lack of definitional clarity. These inconsistencies highlight the difficulty of establishing a unified understanding of chaos engineering—a challenge observed in previous research, where fragmented definitions can hinder conceptual clarity and slow down the development of standardized practices [13, 47, 48, 70]. As demonstrated in those contexts, establishing a clear and shared definition can help eliminate conceptual ambiguity, enable the development of sound theory and valid constructs, and strengthen the connection between research and practice. Based on these insights, we propose the following unified definition:

Chaos engineering is a resilience testing practice that intentionally injects controlled faults into software systems in production-like or actual production environments to simulate adverse real-world conditions. It enables uncovering hidden vulnerabilities in systems that hinder their ability to withstand potential real-world disruptions. It also enables the assessment and enhancement of systems' capacity and operational readiness against such disruptions.

In line with the definition, chaos engineering relies on a set of core activities to guide the design and execution of experiments under controlled conditions [4, 17]. Table 9 summarizes these activities.

Table 9. Core Activities in the Chaos Engineering Lifecycle

Activity	Description	Articles Referenced
Establish Steady State	Understand the system's normal behavior and hypothesize how it will handle chaos (real-world disruptive incidents/failures).	[1, 15, 18, 19, 22, 25, 41, 42, 46, 53, 55–57, 68, 69, 73, 79, 82, 83, 87, 89, 94, 96, 98, 104, 109, 110, 124, 127, 132, 136, 139, 151, 156, 157, 172, 185, 194, 195]
Setup Monitoring Infrastructure	Set up tools to monitor the system for deviations from its normal behavior during controlled experiments.	[1, 15, 25, 68, 73, 79, 89, 109, 127, 128, 136, 139, 140, 151, 156, 157, 185, 192, 194]
Conduct Chaos Experiments	Design and execute experiments to simulate real-world failures.	[1, 15, 19, 25, 46, 53, 55, 68, 69, 79, 82, 83, 89, 109, 110, 124, 132, 136, 139, 147, 156, 172, 194]
Test and Refine	Review results, apply improvements, and repeat to boost resilience.	[15, 18, 25, 41, 55, 68, 73, 79, 83, 96, 136, 147, 151, 156, 171, 172, 185, 190, 194]
Grow Blast Radius	Gradually expand experiments to test increasingly complex systems and environments.	[15, 53, 57, 68, 73, 78, 79, 82, 83, 89, 96, 109, 110, 124, 127, 139, 140, 151, 156, 157, 171, 172, 185, 194, 195]
Expand to Production	Run experiments in production to build confidence in the system's ability to endure adverse real-world events.	[7, 17, 53, 94, 97, 119, 129, 134, 139, 156, 174, 187]

4.2 Functionalities of Chaos Engineering (RQ1.1)

Chaos engineering comprises a structured set of core functionalities designed to reveal system weaknesses under real-world failure conditions. Based on our thematic analysis of academic and grey literature (see Section 3.5), we identified five key functionalities that characterize how chaos engineering is typically conducted: *experiment design and planning*, *fault execution and control*, *monitoring and observability*, *post-experiment analysis*, and *automation and continuous integration* (see Figure 4). For full transparency, the detailed mapping of raw codes to these functionalities and supporting themes is provided in the online appendix (see Section 3.6). The remainder of this section describes each functionality, emphasizing its purpose and role in validating system resilience.

4.2.1 Experiment Design and Planning. This phase establishes the conceptual and procedural foundation for chaos experimentation. It ensures that testing is deliberate, measurable, and risk-aware [55, 136, 156, 180]. It involves three essential activities: identifying test targets and failure scenarios, defining steady-state behavior, and formulating testable hypotheses.

Identify Test Targets and Failure Scenarios. The first step is to identify components whose failure could significantly disrupt operations, such as microservices, APIs, databases, or message queues [25, 124, 136, 180]. Failure scenarios are then defined based on system architecture analysis (e.g., service dependencies, communication patterns), operational risks, or past incident reports [37, 55, 109, 128]. Typical disruptions include network latency, packet loss, service crashes, or CPU throttling [37, 156]. These scenarios are prioritized according to their likelihood and potential business impact [55, 124, 157, 172].

Define Steady-State Behavior. Steady-state refers to the normal, healthy performance of the system. Teams define this using measurable indicators such as latency, error rates, throughput, or availability [57, 109, 156]. These metrics act as baselines

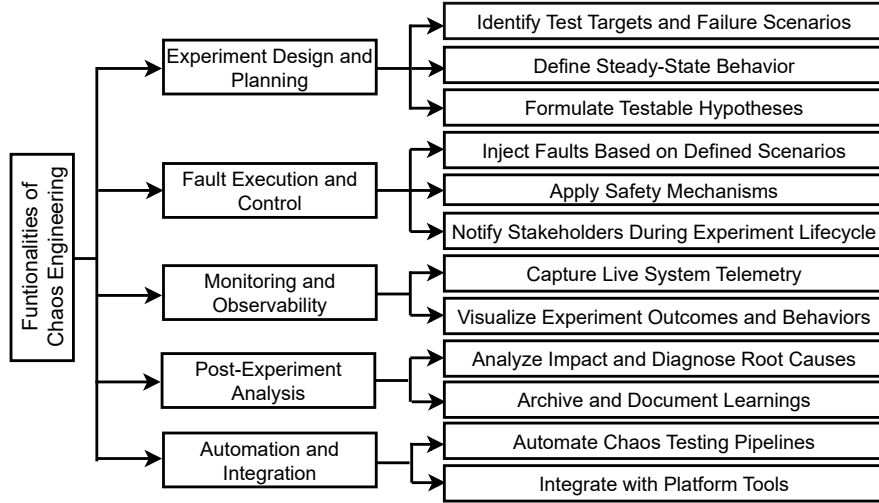


Fig. 4. Key Functionalities Provided by Chaos Engineering.

against which system behavior is compared during and after failure injection [68, 124, 157]. Establishing these baselines enables reliable detection of performance degradation and validation of resilience mechanisms [79, 124].

Formulate Testable Hypotheses. Hypotheses describe the expected system response to injected faults. For instance, a hypothesis might state: “If service A fails, traffic will reroute to service B within 200ms without user impact” [46, 156]. These statements guide the design of experiments and help interpret the results. To manage uncertainty and uncover hidden risks, teams can use structured frameworks such as the Rumsfeld Matrix, which categorizes system knowledge as follows [22, 46, 79, 109, 133, 139].

- **Known Knowns:** Aspects of the system that are fully understood and predictable.
- **Known Unknowns:** Recognized uncertainties that pose potential risks.
- **Unknown Knowns:** Understood risks that may not be immediately apparent or considered.
- **Unknown Unknowns:** Completely unforeseen issues that may emerge during experimentation.

. This framework supports a gradual testing strategy—starting with predictable behaviors and extending to more uncertain or unforeseen failure modes. Doing so helps reduce risk while expanding organizational understanding of system weaknesses [36, 79, 132].

4.2.2 Fault Execution and Control. This phase ensures that faults are introduced in a controlled, safe, and purposeful manner to validate the system’s resilience under adverse conditions. The goal is to simulate real-world failures while minimizing risk and maximizing learning [73, 136, 193].

Inject Faults Based on Defined Scenarios. Chaos experiments begin by executing predefined fault scenarios aligned with the experiment plan. These include disruptions such as service crashes, network partitions, and CPU saturation [42, 128, 157, 193]. Faults are introduced progressively—starting with a minimal scope and gradually increasing impact—to monitor system behavior and avoid widespread disruption [37, 55, 73, 156, 185]. Execution readiness checks ensure that failure types are verified, system health is stable, and rollback plans are in place [15, 36, 85, 180, 194].

Apply Safety Mechanisms. To protect against unintended system degradation, safety measures are implemented. These include automatic abort thresholds that halt experiments if system performance degrades beyond acceptable levels [32, 37, 57, 109]. Rollback protocols ensure the system can return to a known steady state, while blast radius limits prevent faults from affecting critical components [2, 73, 79, 85, 156].

Notify Stakeholders During Experiment Lifecycle. Transparent communication is maintained throughout the experiment. Key stakeholders are informed at defined checkpoints (start, progress, and end of experiment) to facilitate oversight, enable timely intervention, and foster cross-functional trust in the experimentation process [15, 73, 104, 136, 157, 172, 194].

4.2.3 Monitoring and Observability. This phase enables teams to understand how the system behaves under stress by continuously collecting and interpreting telemetry data during chaos experiments. Its purpose is to detect deviations, validate resilience assumptions, and guide response strategies [32, 79, 127, 194].

Capture Live System Telemetry. The process begins with configuring telemetry tools to collect real-time data, such as metrics, logs, and traces [127, 141, 156]. These data streams encompass both application-level indicators (e.g., service availability, request latency) and infrastructure metrics (e.g., CPU usage, memory consumption) [37, 68]. To ensure full system visibility, team members are assigned specific monitoring scopes—some focusing on business-critical services, others on backend components or network behavior [128, 180]. When unexpected behaviors occur, teams follow predefined diagnostic steps, such as analyzing error logs, correlating metrics, and reviewing recent deployment changes, to progressively isolate and identify the faulty components [36, 108, 136, 161].

Visualize Experiment Outcomes and Behaviors. Collected data is visualized through dashboards that track key metrics over time and flag deviations from expected behavior [15, 36, 68, 79, 127, 140]. These visual tools guide real-time decisions and help verify whether system behavior aligns with predefined hypotheses [25, 28, 128, 194]. Observations are documented with relevant metadata, such as timestamps, environment context, and test conditions—to support future analysis and continuous learning [46, 73, 124, 161].

4.2.4 Post-Experiment Analysis. This phase focuses on deriving actionable insights from chaos experiments by assessing impact, diagnosing root causes, and preserving knowledge for future use [5, 25, 36, 57, 136]. The process involves three key areas of functionality.

Analyze Impact and Diagnose Root Causes. After fault injection, teams analyze telemetry, such as metrics, logs, and traces, to compare system performance against baseline expectations [46, 79, 156, 180]. Deviations are measured to understand the extent of impact on key indicators such as latency or availability [57, 83, 140]. This helps prioritize which failures matter most. Root cause analysis then investigates how the failure unfolded, using traces and service maps to locate the origin and propagation path of faults [15, 25, 37, 108, 192, 194].

Archive and Document Learnings. All findings are documented in structured postmortem reports, which capture the fault scenario, affected services, violated hypotheses, and recovery behavior [15, 36, 83, 85, 180]. These insights are stored in internal knowledge bases to support incident response and guide future experiments [73, 157, 194]. By archiving this information, teams enable iterative learning and systemic improvements in both technical and organizational resilience [46, 73].

4.2.5 Automation and Integration. This phase automates chaos experiments to minimize manual effort and then integrates them into delivery pipelines and platform workflows, ensuring that resilience testing becomes routine.

Automate Chaos Testing Pipelines. Automation reduces manual effort by enabling chaos experiments to run consistently across different stages of the software delivery process, such as development, staging, and production [19, 36, 139]. Teams can define test parameters and execution policies in advance, ensuring that faults are injected at predictable times or in response to deployment events [15, 110]. This supports repeatability, reduces errors, and allows for continuous resilience validation throughout the development lifecycle [68, 78].

Integrate with platform tools. To embed chaos engineering into day-to-day engineering workflows, tools are integrated with existing platforms such as CI/CD (Continuous Integration and Continuous Delivery/Deployment) pipelines (e.g., Jenkins, GitHub Actions), cloud environments (e.g., Kubernetes, AWS), and observability frameworks (e.g., Prometheus, Grafana) [32, 110, 134, 156]. This integration enables the automated execution of chaos experiments during the build, test, and deployment phases, while also ensuring that experiment telemetry is collected and visualized in real-time. As a result, teams can detect weaknesses early in the lifecycle and improve response times during failure scenarios [15, 68, 127, 157].

4.3 Key Components of Chaos Engineering (RQ1.2)

Before adopting a chaos engineering platform, organizations should carefully evaluate its key components to ensure that they align with the operational requirements and objectives of the organization. Based on our systematic thematic analysis, we identified five primary components: the *Experiment Design Unit*, *Fault Injection Unit*, *Observability Unit*, *Post-Experiment Analysis Unit*, and *Automation and Integration Unit* (see Figure 5). This classification is based on the functionalities discussed in Section 4.2, mapping each component to the corresponding phase of the experimentation lifecycle. Each unit encapsulates

modular services, such as hypothesis management, failure orchestration, observability pipelines, and automation engines, that together support scalable, safe, and repeatable experimentation. The remainder of this section details each component, its sub-modules, and their roles within the broader platform architecture. For details on coding, theme development, and mapping, see our online appendix (see Section 3.6).

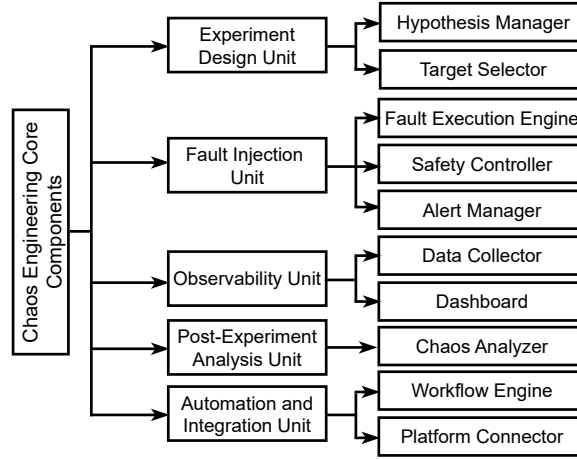


Fig. 5. Key Components of Chaos Engineering.

4.3.1 Experiment Design Unit. This architectural component transforms high-level test objectives into system-executable configurations [159, 193]. It offers formal interfaces to define hypotheses, select test targets, and validate readiness prior to chaos injection [14, 37, 46, 56, 151]. The experiment design unit comprises two integrated modules: the Hypothesis Manager and the Target Selector.

Hypothesis Manager. This enables teams to define, store, and manage test hypotheses through structured templates that capture expected system behavior using time-series metrics (e.g., latency, availability) and other indicators such as error rates and log patterns [21, 67, 73, 85, 104]. Each hypothesis is programmatically linked to safety controls, including abort thresholds and rollback triggers, via policy configurations [37, 140, 156]. This ensures automated enforcement of safety during fault injection and accurate post-experiment validation [46, 193].

Target Selector. This phase scopes where faults will be injected by allowing teams to define experiment boundaries using filters such as environment (e.g., staging, production), infrastructure layer (e.g., compute, network), and service tags or labels [14, 73, 159]. It ensures controlled impact by enforcing blast radius limits, so only approved targets are affected [151]. To maintain operational safety, it performs validation checks on all configurations, verifying correctness, authorization, and compatibility before execution. This prevents unsafe or unintended experiments from proceeding [15, 37, 42, 193].

4.3.2 Fault Injection Unit. This is the core architectural component responsible for executing chaos experiments by introducing faults into the system under test. It is composed of three core modules: *the Fault Execution Engine*, *the Safety Controller*, and *the Alert Manager*, each supporting a distinct aspect of the fault injection lifecycle.

Fault Execution Engine. This component orchestrates the actual injection of predefined faults and workloads. Its design follows the foundational model by Hsueh et al. [77] (See Figure 6), and has been extended to suit modern, distributed environments [56, 159]. It comprises several modules: (1) A **Controller** that sequences experiment steps and responds to telemetry, (2) A **Fault Injector** that applies disruptions from a configurable fault library, (3) A **Workload Generator** that simulates system activity using patterns from a workload library, (4) The **Target System**, which receives the injected faults and workloads and emits telemetry for analysis [185]. Other components from the original model (see Figure 6)—such as the **Monitor**—are represented in the subsequent architectural component through the *Observability Unit*, which is responsible for telemetry collection. Similarly, the **Data Collector** and **Chaos Analyzer** are addressed within the *Post-Experiment Analysis Unit*, which supports impact assessment and root cause diagnosis following fault injection.

Safety Controller. This enforces runtime protection by embedding safeguards directly into the fault injection lifecycle [21, 42, 105, 193]. It comprises three subsystems:

- **Blast Radius Control.** This limits the impact of injected faults by using traffic routing and isolation rules. For instance, in Kubernetes, these rules restrict disruptions to specific services, namespaces, or pods within the system [89, 159, 180, 190].
- **Abort Monitor.** This continuously checks key metrics (e.g., error rate, response time) and halts the experiment if thresholds are breached [73, 85, 104].
- **Rollback Handler.** This restores the system to a steady state by triggering recovery workflows or replaying system snapshots [21, 42, 85, 159].

Alert Manager. This component coordinates communication with stakeholders by triggering real-time notifications across channels (e.g., Slack, email) at critical points: start, escalation, and end [68, 73]. It also enforces escalation policies when thresholds are breached to ensure rapid response [14, 32, 136, 147].

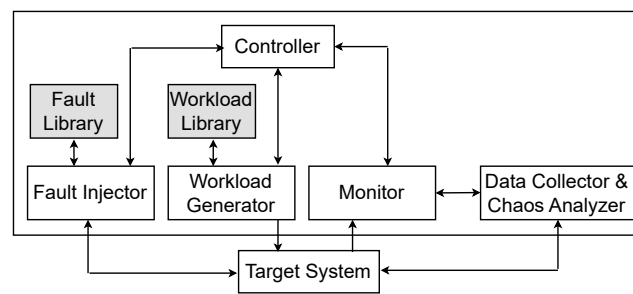


Fig. 6. Conceptual Schema of Fault Execution Engine[77].

4.3.3 Observability Unit. This component implements the infrastructure for capturing, processing, and presenting telemetry data during chaos experiments [73, 141]. It is composed of two core subsystems: the *Data Collector* and the *Dashboard*.

Data Collector. This subsystem captures diverse telemetry from systems under test [120, 124, 159, 193, 194]. It contains adapters/connectors for various data sources, buffering mechanisms for handling high-volume streams, and preprocessing capabilities for initial data normalization, such as timestamp alignment, unit standardization (e.g., converting memory metrics to a common format), and removal of malformed or incomplete entries [32, 73, 128, 141, 180].

Metrics Collector This component captures quantitative measurements, primarily in the form of time-series metrics, using instrumentation libraries and collection endpoints [14, 73, 128, 141]. It stores data in specialized time-series databases with retention policies and aggregation capabilities [127, 141]. It supports both system-level metrics (CPU, memory, network) and application-level indicators (throughput, error rates) with statistical analysis functions [21, 53, 141, 159, 193].

Log Aggregator This component collects logs from all participating systems during a chaos experiment [159, 180]. It uses log shippers such as Fluentd³ to forward entries to a central log system [14, 73]. Logs are parsed into structured formats (e.g., JSON) for consistency [36, 163]. The component typically indexes log data to enable fast retrieval and supports querying by time, service, or error type [46, 85].

Tracer This component tracks request flows across service boundaries in distributed systems [124, 128, 193]. It utilizes lightweight instrumentation, context propagation protocols, and sampling algorithms for data volume management [73, 124, 180]. It can also visualize service relationships, analyze performance bottlenecks, and correlate traces with metrics and logs [124, 127].

Dashboard. This component renders configurable monitoring interfaces from multiple data sources [127, 128, 141]. It can include sub-components such as query interfaces, graphical rendering engines, and templating systems [18, 67, 73]. The common features include real-time data refresh, threshold alerting, time-range selection, and export capabilities for sharing observations [21, 73, 141].

³<https://www.fluentd.org/>

4.3.4 Post-Experiment Analysis Unit. This unit implements the processes needed to evaluate chaos experiments, identify technical weaknesses, and retain organizational knowledge [25, 85, 156, 180, 194]. It includes two subsystems: the *Chaos Analyzer* and the *Knowledge Base Manager*.

Chaos Analyzer. This unit processes telemetry data to assess system behavior and diagnose failure mechanisms.

Impact Analyzer This module compares actual telemetry (e.g., metrics, logs, traces) against steady-state thresholds to quantify performance degradation [57, 156, 180]. It calculates impact scores based on key performance indicators (KPIs) such as latency and availability, and ranks vulnerabilities for follow-up [25].

Root Cause Analyzer This module Correlates data across logs, traces, and metrics to trace fault propagation and uncover causal links [85, 96]. It applies anomaly detection and pattern-matching to distinguish root issues from surface symptoms [156, 194].

Knowledge Base Manager. This module stores post-experiment findings, including violated hypotheses, impact severity, and root causes, in structured formats. These are indexed and archived in resilience knowledge bases, supporting institutional memory, cross-team learning, and onboarding of new engineers [85, 110, 180].

4.3.5 Automation and Integration Unit. This component operationalizes the process of continuous chaos validation. It allows teams to schedule, coordinate, and trigger fault injections across environments and platforms [151, 180, 193]. The unit includes two major subsystems: Workflow Engine and Platform Connector.

Workflow Engine. This module orchestrates the automatic execution of chaos experiments based on scheduled intervals or event-driven triggers, such as deployments, pipeline stages, or config changes [18, 32, 42, 55, 108, 127, 156, 180, 193]. It applies user-defined policies to determine timing, scope, and test conditions. The engine ensures fault injection is repeatable and synchronized with real system activity, minimizing manual intervention [15, 25, 36, 68, 69, 111, 128, 139, 151, 172]

Platform Connector. This facilitates seamless integration with external environments. It interfaces with CI/CD systems (e.g., Jenkins, GitHub Actions), cloud platforms (e.g., AWS, Azure, GCP), and observability tools (e.g., Prometheus, Grafana) [18, 56, 109, 111, 128, 180, 193]. Through these connections, it supports the triggering of chaos experiments based on pipeline events, deployment stages, or infrastructure changes. It also ensures synchronized data exchange across orchestration layers, telemetry systems, and alerting channels. [42, 46, 55, 156, 172, 194]

4.4 Quality Requirements for Chaos Engineering Platform (RQ1.3)

We have identified 11 key quality requirements for a chaos engineering platform from the selected literature: performance, recoverability, resilience, timeliness, observability, scalability, effectiveness, robustness, accuracy, fault tolerance, and adaptability. Table 10 provides these requirements and the corresponding metrics for measuring them.

5 Motivation Behind Chaos Engineering (RQ2)

This section directly addresses **RQ2** by delving into the challenges driving chaos engineering adoption in organizations. Figure 7 shows the technical and socio-technical challenges we identified from the reviewed literature. These challenges were identified through our coding and thematic analysis (see Section 3.5) and are documented in detail in our online appendix (see Section 3.6).

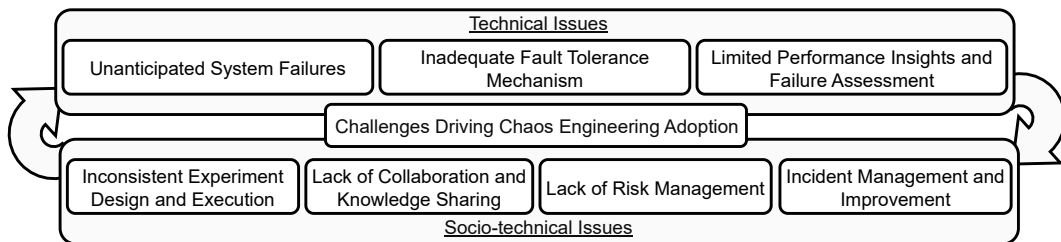


Fig. 7. Challenges that Drive Chaos Engineering Adoption.

Table 10. Quality Attributes, Measurement Metrics, and Referenced Articles.

Quality Attribute	Description	Measurement Metrics	Articles Referenced	Total
Performance	Evaluate resource and performance under attack; align chaos tests with production metrics.	Response time (ms), CPU/memory utilization (%), throughput (req/sec).	[42, 46, 55, 68, 93, 127, 136, 172, 180, 194]	10
Recoverability	Assess how quickly the system restores functionality after a failure.	Mean Time to Repair (MTTR), roll-back time, recovery time objective (RTO).	[32, 42, 46, 55, 56, 68, 85, 109, 127, 136, 139, 171, 172, 180, 192, 195]	16
Resilience	Evaluate the system's ability to operate under failure without service disruption.	Percentage of successful requests under fault, error rate, degradation threshold.	[5, 14, 22, 25, 46, 55, 57, 86, 94, 96–98, 110, 126, 128, 129, 172, 180, 192]	18
Timeliness	Measure how quickly faults are detected and addressed.	Time to detect (TTD), alert response time.	[18, 32, 42, 49, 50, 56, 68, 87, 109, 110, 132, 139, 152, 180, 190, 194]	16
Observability	Assess the visibility into system internals during chaos.	Log completeness (%), alerting coverage, trace propagation rate, telemetry accuracy.	[14, 25, 32, 46, 55, 57, 68, 75, 79, 85, 89, 94, 96, 109, 110, 136, 157, 172, 173, 180, 192]	20
Scalability	Gauge how chaos testing scales with system size and complexity.	Number of nodes/services covered per experiment, test execution time at scale.	[14, 33, 50, 55, 68, 86, 93, 96, 97, 109, 110, 128, 132, 136, 146, 161, 172, 179, 180, 190, 194]	20
Effectiveness	Determine whether chaos tests produce meaningful insights.	Number of unknown faults revealed, test success rate.	[36, 109, 161, 190]	4
Robustness	Evaluate stability at peak load and error tolerance.	System uptime under stress, error rate at max load.	[19, 36, 108, 179, 180, 190, 191]	7
Accuracy	Assess how precisely the system recovers and localizes issues.	Post-recovery state correctness (%), fault localization accuracy.	[36, 73, 156]	3
Fault Tolerance	Measure the success of self-healing and failover mechanisms.	Failover completion time, success rate of automatic recovery actions, redundancy coverage.	[14, 32, 46, 55, 68, 108, 110, 111, 127, 140, 161, 171, 172, 185, 192, 194]	15
Adaptability	Determine how easily the chaos system fits into varied environments.	Number of supported platforms, integration effort (e.g., setup time), configuration reuse rate.	[108, 180, 190, 191]	4

5.1 Technical Challenges

Chaos engineering aims to tackle various technical challenges that undermine system resilience. For example (see Section 3.6), the theme ‘Unanticipated System Failures’ includes codes such as ‘network-partition’ and ‘cascade-failure’, capturing failures that are often missed by traditional software testing. Similarly, the theme ‘Inadequate Fault Tolerance Mechanisms’ incorporates codes like ‘multi-region outage’ and ‘resource exhaustion’, highlighting gaps that chaos experiments are designed to expose. The following subsections discuss these and related technical challenges in more detail.

Unanticipated System Failures. In a distributed system, one of the most critical risks is the occurrence of unexpected failures in various parts of the system [3]. Given the highly interconnected nature of modern systems, a failure in one component can quickly cascade into widespread disruptions across multiple services or functions [185]. However, traditional testing methods often struggle to identify deep-rooted system vulnerabilities because they focus on known scenarios and isolated components, neglecting real-world systems’ dynamic and complex interactions [159]. In contrast, chaos engineering adopts a proactive approach by deliberately introducing failures, such as server crashes, network interruptions, and resource exhaustion, in environments that closely resemble production. As a result, engineers can observe how the system behaves under stress, thus revealing hidden weaknesses that traditional testing methods might overlook. By focusing on unpredictable interactions between distributed components, chaos engineering effectively uncovers vulnerabilities that arise from the complexity of the system [15]. For example, chaos experiments might simulate network partitioning between microservices, which can reveal issues related to data consistency and service availability [15, 55]. These controlled disruptions allow engineers to better understand how different system components interact under turbulent conditions [172].

Inadequate Fault Tolerance Mechanisms. Many distributed systems are built with fault tolerance mechanisms to handle routine issues, such as server failures or network delays [79]. However, when larger-scale failures occur, e.g., multi-region outages or prolonged network disruptions, these mechanisms often fail to maintain system stability [123, 132, 155]. This shortfall occurs because mechanisms designed to handle more minor issues may not be able to handle the broader impact of significant failures, leading to severe performance degradation or even total system outages [18, 68]. Chaos engineering addresses these shortcomings by targeting and testing fault tolerance mechanisms in scenarios replicating the most significant

risks to system stability [170]. For example, chaos engineering can simulate the unavailability of critical services in one region to determine whether traffic is adequately redirected to the other areas or if the load balancers of the system can effectively distribute incoming requests between redundant resources [2, 25]. Furthermore, by testing the ability of backup systems to maintain service levels during prolonged outages, chaos engineering reveals whether these fallback mechanisms are sufficient for extended failures [18, 79, 185]. These focused experiments help ensure systems can handle severe disruptions without experiencing widespread failure [15].

Limited Performance Insights and Failure Assessment. Another significant challenge in distributed systems is understanding how they perform during high-stress or failure scenarios [38, 54]. When systems operate under normal conditions, traffic bottlenecks, inefficient resource usage, or service communication failures often remain hidden [55]. This lack of visibility leaves teams unprepared for unexpected disruptions, which can lead to outages or performance degradation when systems are stressed [94]. Chaos engineering addresses this by introducing specific disruptions, such as intentionally increasing network latency or restricting memory resources on critical components [157]. These controlled failures are monitored through observability tools that provide real-time visibility into metrics such as latency, error rates, and resource consumption [181]. By analyzing these metrics, teams gain detailed insight into how services handle stress - uncovering issues such as delayed responses or inefficient resource utilization that would otherwise remain hidden during normal operations [25, 132].

5.2 Socio-Technical Challenges

Chaos engineering can address various socio-technical challenges, expanding its impact beyond technical issues to encompass the complex interplay between people, processes, and technology [116, 137, 168]. For instance, the theme ‘Inconsistent Experiment Design’ maps to codes such as ‘variable test timing’ and ‘lack of standardization’. Additional examples and detailed code-theme mappings can be found in the online appendix (see Section 3.6).

Inconsistent Experiment Design and Execution. When experiment design and execution lack consistency, teams struggle to accurately assess the behavior of the system under stress [31, 52]. Factors such as varying test timing, failure scenarios, or changing system states lead to unreliable data. This makes it difficult to identify critical vulnerabilities or adequately assess the system’s ability to handle disruptions [93]. Without standardization, essential insights into system responses may be overlooked, creating false confidence in its resilience [44]. Chaos engineering addresses this by introducing a structured and repeatable approach, ensuring that predefined failure scenarios such as network issues or resource depletion are applied consistently across tests [96, 124, 180, 192]. This uniformity eliminates variability, producing reliable, comparable data that allows teams to pinpoint weaknesses and confidently evaluate recovery mechanisms [14, 85].

Lack of Collaboration and Knowledge Sharing. Limited collaboration and challenges in knowledge sharing between teams, such as development, operations, and security, can lead to communication gaps and the formation of silos, where vital information may not be effectively exchanged [74, 144]. This results in misaligned responses and repeated errors during system failures, which ultimately slows recovery efforts and misses opportunities for improvement [45]. Chaos engineering tackles this issue by fostering cross-functional collaboration, engaging all relevant teams in both the planning and execution of chaos experiments [14, 170, 171]. For example, when simulating a specific failure, such as a service outage or database failure, development teams define the expected behavior of the system, operations teams monitor the system’s performance under stress, and security teams evaluate potential vulnerabilities exposed during the failure [97]. This multi-team involvement ensures that the insights from the experiment, whether an unexpected failure response or a missed alert, are immediately shared and understood by everyone [49, 73, 87]. Furthermore, the results of each chaos experiment are documented and discussed in collaborative postmortem sessions, where all teams contribute to analyzing what went wrong, how to fix it, and how to prevent similar issues in the future [157].

Lack of Risk Management. Effective risk management is critical in both production and non-production environments, yet many systems are exposed to unexpected failures due to insufficient preparation [25, 124]. In non-production settings, such as development or staging, failure scenarios are often inadequately tested, leading to a false sense of security [83, 185]. These environments typically lack the complexity of live production systems, meaning that when software is deployed, it may not be fully equipped to handle adverse real-world events [73]. Consequently, when systems reach production, where they face real user traffic, external dependencies, and unpredictable conditions, even minor issues can escalate to significant outages, resulting in costly downtime and degraded user experiences [68, 84, 194]. Chaos engineering solves these challenges by enabling controlled experimentation that tests system behavior under simulated and real-world failure conditions [96, 172].

In production environments, controlled blast radiuses ensure that failures are confined to isolated parts of the system, thereby preventing widespread disruption [22, 67, 120, 132, 157]. Predefined abort conditions and rollback procedures allow teams to stop experiments and recover quickly if the stability of the system is threatened [1, 151, 157].

Incident Management and Improvement. In complex, distributed systems, incident management faces significant challenges due to the unpredictability of failures and the limitations of reactive strategies [67]. In these approaches, teams respond to incidents only after they occur, focusing on diagnosing and resolving issues after disruptions have already impacted operations [25]. This often leads to prolonged downtime and repeated failures, as reactive methods limit the opportunities to address these problems proactively [136]. Chaos engineering offers a proactive approach by simulating targeted failures that stress critical parts of the system, revealing weaknesses in the incident response process [10, 157]. For example, simulating a sudden network partition might show whether monitoring systems detect the issue promptly and if the system can reroute traffic as intended [68, 73, 179]. Similarly, inducing a Central Processing Unit (CPU) or memory exhaustion event could expose delays in triggering alerts or inefficiencies in the automated recovery process [32, 85, 132, 147]. By creating these controlled failure scenarios, chaos engineering helps teams identify where their incident response protocols fall short, whether in monitoring, alert escalation, or failover processes [190, 192]. This enables teams to proactively adjust their strategies, improving their ability to handle actual incidents faster, more accurately, and with minimal disruption [49, 146].

Table 11. Mapping of Core Chaos Engineering Functionalities with Organizational Benefits.

Core Functionality	Activity Performed	Benefits to Organization	Articles Referenced	Total
Identify Test Targets and Failure Scenarios	Select critical components (e.g., APIs, services) and likely fault types based on architecture and risk.	Focuses efforts on the most vulnerable areas, improving preparedness and reducing downtime	[1, 5, 10, 14, 25, 41, 42, 46, 49, 53, 58, 67–69, 73, 79, 96, 98, 110, 119, 128, 139, 140, 147, 156, 159, 172, 179, 180, 185, 192–194]	33
Define Steady-State Behavior	Establish baseline performance metrics (e.g., latency, throughput)	Provides a reference for detecting anomalies and validating system health under failure.	[1, 2, 15, 22, 25, 42, 46, 53, 55, 57, 68, 69, 79, 82, 83, 89, 94, 96, 98, 104, 119, 124, 127, 133, 134, 136, 139, 147, 156, 157, 172, 194, 195]	33
Formulate Testable Hypotheses	Describe expected system responses to faults in measurable terms.	Guides experiment design and enables structured interpretation of results	[1, 15, 22, 25, 42, 46, 53, 55, 57, 68, 69, 79, 82, 83, 89, 94, 124, 133, 134, 136, 156, 157, 172, 194, 195]	25
Inject Faults Based on Defined Scenarios	Execute fault (e.g., CPU overload, network delay) on selected systems.	Validates resilience strategies; exposes weak points under controlled stress	[18, 21, 36, 37, 42, 56, 67, 69, 78, 85, 108, 111, 127, 128, 139, 151, 159, 161, 180, 192–194]	22
Apply Safety Mechanisms	Enforce abort conditions, rollback procedures, and limit blast radius.	Minimizes impact, safeguards critical systems, ensures quick recovery.	[5, 15, 25, 36, 49, 56, 68, 73, 78, 82, 85, 87, 94, 124, 133, 136, 139, 151, 159, 164, 170, 179, 180, 185]	24
Notify Stakeholders During Experiment Lifecycle	Send alerts and updates before, during, and after chaos experiments.	Promotes transparency, readiness, and coordinated response among teams.	[1, 14, 49, 50, 68, 73, 94, 96, 97, 110, 127, 136, 139, 156, 157, 159, 170–173, 179, 185, 193, 194]	24
Capture Live System Telemetry	Monitor and record real-time metrics, logs, and traces during injection.	Enables detailed analysis of system and supports rapid anomaly detection.	[28, 32, 37, 46, 68, 79, 83, 104, 108, 124, 127, 128, 140, 141, 156, 157, 161, 180, 194, 195]	20
Visualize Experiment Outcomes and Behavior	Display telemetry on dashboards to highlight key deviations and trends.	Provides insights for decisions and facilitates shared understanding.	[25, 42, 46, 55, 73, 79, 127, 156, 172, 193, 194]	11
Analyze Impact & Root Causes	Compare metrics to baselines and trace root causes.	Identifies weaknesses, guides improvements.	[15, 25, 42, 46, 55, 73, 79, 127, 139, 140, 156, 159, 172, 192–194]	16
Archive and Document Learnings	Store findings, root causes, and lessons learned in accessible knowledge bases.	Preserves knowledge and supports future analysis, audits, and training.	[5, 18, 25, 33, 36, 46, 55, 57, 68, 73, 79, 85, 89, 94, 136, 156, 157, 172, 180, 190, 191, 194]	22
Automate Chaos Testing Pipelines	Schedule and execute experiments with scripts or workflows.	Enhances consistency, lowers overhead, supports continuous validation.	[19, 36, 68, 73, 78, 110, 127, 134, 139, 156, 157, 172, 194]	13
Integrate with Platform Tools	Link chaos tools to CI/CD, monitoring, and cloud systems for orchestration.	Embeds resilience testing into delivery pipelines and ensures synchronized operations.	[25, 32, 42, 46, 56, 127, 140, 159, 172]	9

5.3 Benefits

Table 11 outlines the organizational benefits derived from the functionalities described in Section 4.2, addressing the previously identified challenges. These functionalities were identified through our thematic analysis process. They include defining

chaos experiments, facilitating team communication, notifying stakeholders, maintaining documentation, and using real-time monitoring dashboards. The corresponding activities, such as establishing experimental parameters, analyzing system performance metrics, and fostering cross-functional collaboration, contribute to several organizational advantages. These include improved system understanding, enhanced error-handling capabilities, vital team coordination during stress conditions, and proactive system tuning. Furthermore, automating tasks and monitoring systems in real-time can minimize disruptions and optimize system performance and user experience.

6 Chaos Engineering Framework: Taxonomy, Tools, Practices, and Evaluation (RQ3)

This section addresses (RQ3) by outlining proposed solutions in chaos engineering. It presents a taxonomy of tools and techniques (RQ3.1), highlights best and bad adoption practices (RQ3.2), and describes the methods used to evaluate chaos engineering approaches (RQ3.3).

6.1 Taxonomy of Chaos Engineering (RQ3.1)

To address RQ3.1, we propose a taxonomy (Figure 8) that organizes chaos engineering platforms across several key dimensions: execution environment, automation strategy, automation mode, deployment type, and evaluation approach. The dimensions of this taxonomy were developed from the core themes identified through our coding and thematic analysis. Table 12 presents an excerpt of the intermediate coding structure that grounds this taxonomy, mapping each conceptual category to its subcodes and supporting sources. This taxonomy provides organizations with a structured approach to adopting chaos engineering practices by offering clear guidelines for selecting appropriate execution environments, choosing automation strategies that integrate essential tools such as monitoring, logging, and observability, and determining the optimal deployment types (e.g., production or pre-production) based on their risk tolerance. It also helps define task modes (manual, semi-automated, or fully automated). Additionally, it guides the selection of chaos engineering tools by aligning them with specific combinations of these dimensions—for example, tools differ in aspects such as the types of faults they inject and the level of automation they support, ensuring that organizations have the necessary evaluation methods to measure the impact of experiments and derive actionable insights for system improvement.

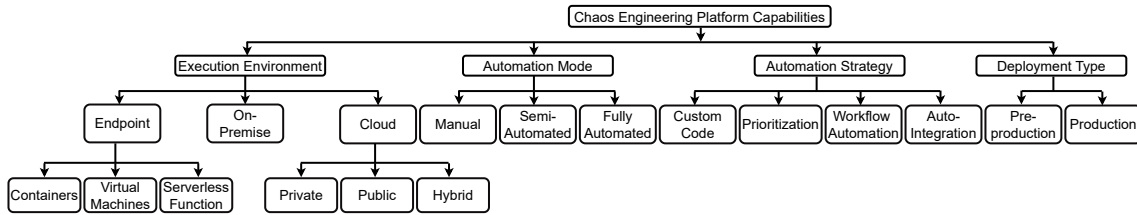


Fig. 8. A Taxonomy of Chaos Engineering Platforms and Their Capabilities.

Table 12. Representative Codes and Subcodes for the Taxonomy of Chaos Engineering Platforms and Their Capabilities.

Code	Sub-code	Sub-code
Execution Environment	Virtual Machines	Variability introduced by different guest OS configurations.
		Resource contention issues examined under CPU/memory saturation scenarios.
Automation Mode	Manual	Tends to be time and resource intensive, as each step must be performed and monitored manually.
		Experiments are custom designed for specific systems or hypotheses.
		Carries risk of human error, especially in timing, targeting, or recovery.
Automation Mode	Semi-Automated	Engineer-Defined Parameters.
		Human-in-the-Loop Oversight.
Automation Strategy	Custom Code	Used to simulate specific failure types (e.g., service crash, DB outage).
		Custom scripts extend existing chaos tools to support advanced scenarios.
		Allows precise control over failure timing, scope, and recovery validation.
Automation Strategy	Prioritization	Critical Component Identification.
		Resource Allocation Strategy.
Automation Strategy	Workflow Automation	Ensures repeatability and reduces manual error.
		Supports multi-step or multi-system coordination.
Deployment Type	Production	Live Environment Testing.
		Real-User Interaction Observation.
Deployment Type	Pre-production	Complement to Production Testing.
		Staging Environment Simulation.

6.1.1 Execution Environment. This refers to the underlying infrastructure and system components where chaos engineering experiments are performed. These components are categorized into endpoints, grouped by resource types and cloud environments, representing various deployment models. This section will discuss both of these in detail.

Endpoints. These are physical or virtual devices connected to a computer network to exchange data. These include virtual machines (VMs), containers, and serverless functions [113]. Each type of endpoint presents unique challenges that chaos engineering addresses by simulating failure conditions to improve the robustness of the system and ensure continuous operation.

Virtual machines (VMs) offer a layer of abstraction over physical hardware, allowing multiple independent operating systems to share resources efficiently [17, 153, 158]. However, VMs can experience performance issues when resources are overutilized, and they are vulnerable to hypervisor failures and migration difficulties [14, 55, 163]. Chaos engineering addresses these challenges by testing VMs under stress conditions such as CPU or memory exhaustion to assess if VMs can recover from failures or seamlessly migrate workloads [37, 73, 96].

Containers, which encapsulate applications and their dependencies in lightweight environments, provide greater efficiency and portability compared to VMs [19, 95]. However, containers face resource contention, network instability, and orchestration failures, especially in platforms such as Docker and Kubernetes [156]. Chaos engineering tests on containers simulate abrupt stoppages and network disruptions, helping to evaluate containers' ability to recover within container orchestration systems while maintaining application performance [19, 25, 32, 78, 96, 156].

Serverless functions allow applications to be triggered by specific events and run in isolated containers without requiring infrastructure management [2, 5, 34, 167]. While serverless architectures are scalable and cost-effective, they face challenges such as resource limitations and the reliance on third-party cloud providers [130, 143, 176]. Chaos engineering helps ensure that serverless functions can handle these challenges by simulating cold starts and execution failures, verifying that functions recover quickly and maintain performance under varying loads [19, 35, 83, 121, 164].

Cloud. Cloud environments are integral to modern Information Technology (IT) infrastructures, providing flexible and scalable resources to meet dynamic business demands [12, 65]. These environments are typically classified into three main models: public, private, and hybrid clouds [16, 71]. Each cloud model has unique characteristics, risks, and opportunities, where chaos engineering can address potential failures.

Public cloud platforms, such as AWS, Microsoft Azure, and Google Cloud, offer computing resources over the Internet that are shared between multiple organizations [150, 172]. This model enables businesses to scale their infrastructure on demand, eliminating the need for significant investments in physical hardware. Due to its cost efficiency and scalability, the public cloud has become a popular choice for many enterprises [41, 51, 82]. However, relying on shared infrastructure introduces specific vulnerabilities. For example, a regional outage or failure in an availability zone can have widespread impacts, simultaneously affecting many customers [131]. In addition, many organizations depend on third-party services, such as application programming interfaces (APIs) or managed databases, and failures in these services can trigger further disruptions in their systems [97, 196]. To address these risks, chaos engineering can simulate large-scale failures, such as regional outages or availability zone failures, to test how well systems can failover to other regions without significant service interruptions [55, 126]. Moreover, by injecting failures into critical third-party services, chaos engineering enables organizations to evaluate the effectiveness of their recovery strategies, ensuring that operations can continue even in the face of service disruptions [75, 179, 191].

Private cloud environments, which provide dedicated infrastructure exclusively for a single organization, can be hosted on-premises or managed by third-party providers, offering greater control over data privacy, security, and customization [66, 81]. This model suits organizations with strict regulatory requirements or managing sensitive information [138]. Despite their benefits, private clouds often face scalability and geographic redundancy challenges, as they typically lack the built-in redundancy of public clouds, making them more susceptible to service interruptions due to hardware failures or resource constraints [86, 99]. Without adequate failover mechanisms, private cloud environments may struggle to maintain operational continuity during hardware or network failures [145, 170]. Chaos engineering can be used in private clouds to simulate hardware failures, such as server crashes or network partitions, to evaluate the system's ability to recover from localized disruptions [136, 156]. By simulating resource exhaustion, such as CPU or memory overloads, organizations can test whether their infrastructure can scale to meet unexpected surges in demand, ensuring continuous service availability under pressure [68, 83].

Finally, hybrid cloud environments combine the strengths of public and private clouds, allowing organizations to manage sensitive workloads on private infrastructure while leveraging the scalability of public clouds for nonsensitive or high-demand tasks [72, 140]. This approach provides flexibility but introduces complexity in maintaining smooth integration between the two environments [107]. A key challenge of hybrid clouds is to ensure seamless communication between public and private components, as issues such as network latency, connectivity problems, and data synchronization delays can arise when workloads are distributed across both environments, potentially leading to service disruptions [9, 157]. Chaos engineering can address this issue by allowing organizations to simulate network interruptions or increased latency between public and private clouds [55]. These tests help to evaluate whether workloads can continue to operate despite connectivity issues [46, 172]. Furthermore, chaos experiments that simulate delays or failures in data replication help ensure that the system maintains data consistency and operational continuity, even when part of the infrastructure is compromised [32].

On-Premise. This refers to the computing infrastructure hosted and managed within an organization's physical location rather than relying on cloud providers [127, 175, 187]. These systems offer greater control over hardware, data, and network resources, essential for industries with strict regulatory requirements. However, they also face hardware failures and scaling limitations, affecting the reliability of the systems [30]. For example, an organization managing critical on-premise data might simulate a hardware failure, such as a server or disk crash, to test how quickly backup systems respond. If synchronization delays are found, the replication process can be optimized to ensure smooth transitions and compliance with data availability standards [46, 55]. Similarly, scaling limitations can be addressed by testing how an on-premise infrastructure responds to various demand surges, such as a sudden spike in user traffic during an e-commerce company's high-demand event [14, 67]. This helps identify bottlenecks, optimize resource allocation, and ensure that the system remains responsive under varying load conditions [32, 124, 127, 128, 185].

6.1.2 Experimentation Modes. Chaos engineering platforms offer three modes of experimentation, namely manual, semi-automated, and fully automated, each tailored to different levels of control, complexity, and system scale.

Manual experimentation is most appropriate when engineers require high precision and control, as they take full responsibility for designing, configuring, and executing chaos tests [96, 172]. This approach is suited for organizations with little or no experience in chaos engineering or when highly specific, customized experiments are necessary [50, 194]. For example, an engineer might manually trigger a database failure or introduce network latency to observe the system response in real time [25, 83]. While this mode allows for fine-tuned control, it can be time-consuming and prone to human error, making it ideal for small-scale, customized tests or critical systems where close monitoring is essential [37, 78].

Semi-automated experimentation provides a middle ground between manual control and complete automation, where engineers define the experiment parameters [56, 83, 139]. At the same time, the platform automates tasks such as triggering tests and collecting data, making this mode ideal for experiments that benefit from automated execution while still requiring moderate human involvement to reduce manual effort [68, 89, 127]. For example, engineers might set up an experiment to test resource exhaustion on specific microservices, with the platform automatically running the test at predefined intervals, collecting performance metrics, and allowing them to focus on adjusting parameters and interpreting results [46, 139].

Fully automated experimentation suits large-scale systems that require frequent and continuous testing [18, 136, 156]. After engineers configure the initial setup, the chaos engineering platform takes over, seamlessly integrating experiments into the system operations, typically through the CI/CD pipelines [41, 94, 110, 141]. For example, after a code deployment or infrastructure update, the platform can automatically simulate a service outage or introduce network disruptions, then autonomously gather performance metrics, identify system weaknesses, and generate detailed reports, requiring engineers to intervene only when necessary [15, 42, 46, 89, 127].

6.1.3 Chaos Automation Strategy. Chaos engineering platform employs an automation strategy to ensure consistent and repeatable experiments, reduce manual effort, and enable continuous testing [79, 96, 185].

Custom Code. This involves developing programs or scripts to extend the capabilities of existing tools and gain precise control over system configurations and experiments [42, 90, 122, 169]. It allows engineers to simulate unique failure conditions, ranging from simple component failures, such as restarting a single service, to complex multi-step sequences involving multiple systems, tailored to the specific challenges of their architecture that standard tools may not address [14, 68, 79, 128, 136, 172]. For example, custom code can terminate random containers in distributed environments such as Kubernetes, simulating unexpected service failures and testing the system's recovery capacity by automatically restarting services [25, 53, 156, 180]. In more complex scenarios, custom code can initiate a service outage followed by a database failure or a network partition to

observe how these failures impact system recovery and resilience [68, 156, 185]. Additionally, custom code can be integrated into CI/CD pipelines to simulate high resource consumption, such as CPU throttling or memory exhaustion, after each deployment, allowing early detection of performance bottlenecks before they affect production [14, 85, 94, 110, 132]. This custom code for chaos experiments can be written in various programming languages, such as Python, Bash, or Go, depending on the complexity and specific requirements of the experiment [109, 156, 157, 161, 193].

Auto-Integration of Third-Party Tools. This process automatically integrates external systems, such as monitoring, log, observability, alerting, incident management, and security platforms, into chaos engineering experiments [156, 172]. This automated integration ensures continuous data collection without manual intervention, enabling real-time analysis during system disruptions or failures. As a result, teams can monitor performance, identify issues quickly, and respond more efficiently to failures [96, 132]. This process is vital to integrating monitoring tools that track essential system metrics such as CPU usage, memory consumption, and network performance [67, 190, 194]. Monitoring tools can automatically detect performance deviations and provide immediate feedback. For example, if a service is overloaded due to an unexpected spike in user requests, monitoring systems measure the increase in CPU usage and network load, helping the team assess the impact and determine whether the system can handle the demand or if further intervention is needed [42, 55, 68, 109, 157, 185]. Logging tools record vital events such as service failures, error messages, and recovery attempts, documenting the system's response in real-time. For example, if network communication between two microservices fails, the logging systems capture the exact moment the failure occurred and show the error messages generated by failed retries [42, 108]. This data helps teams analyze how the failure affected communication and whether recovery mechanisms worked effectively [96, 109, 124, 132]. Building on this, observability tools provide a broader view by correlating logs and metrics to show how different components of the system interact during failure scenarios. These tools enable teams to identify how failures propagate across interconnected services [73, 96, 97, 110, 156]. For example, if latency is introduced between services in a chaos experiment, observability systems track the delays at each hop between services and how this latency affects downstream components such as database queries or API responses. This tracing allows the team to see if minor delays in one service cause issues elsewhere [37, 50, 57, 87, 96, 128]. Furthermore, alert notification systems can ensure that relevant teams are promptly notified when critical failures occur or performance thresholds are exceeded [25, 67, 132, 164, 170]. For example, if a database connection pool reaches its maximum capacity during a high-load chaos test, the alert system triggers a notification to the database operations team, allowing them to address the issue before it causes a system-wide failure. This early warning helps prevent more significant disturbances [73, 109]. Finally, incident management tools streamline the resolution process in conjunction with alerting by automatically generating tickets for failures detected during chaos tests. These tickets are assigned to the appropriate teams and tracked until the issue is resolved [73, 79, 104, 136]. For example, if a chaos experiment causes a loss of connection to external APIs, the incident management system creates a ticket assigned to the appropriate API support team, ensuring that the problem is tracked and resolved efficiently [146, 157].

Workflow. A chaos engineering workflow can automate the controlled execution of experiment steps, such as setting up the environment, executing failures, monitoring, and cleanup, ensuring that the experiments are consistent and repeatable [33]. This reduces the risk of errors and guarantees the reliability in each run. For example, automating a database outage simulation ensures that critical steps such as downgrading the database, monitoring the system response, and restoring functionality are executed reliably every time [185]. In addition, workflows simplify complex experiments that involve multiple systems or services. In such cases, they can coordinate the simultaneous failure of various microservices, allowing teams to test the system's ability to maintain service availability despite several disruptions [110, 127]. Furthermore, a workflow engine ensures that each step is executed correctly while capturing and analyzing the results, making the experimentation process both structured and efficient [15, 172].

Prioritization. This involves categorizing system components into critical and non-critical groups to focus testing efforts on the areas where failures would have the greatest impact [10, 37, 46]. Critical components are essential for maintaining core operations, and their failure would cause significant disruption [55, 89]. For example, API gateways and distributed databases are critical because these failures can disrupt communication between services, cause widespread outages, or lead to data loss and downtime, directly affecting users [157]. In contrast, non-critical components have minimal immediate impact on the core functionality of the system. For instance, log aggregation systems are non-critical; their failure may affect internal monitoring but will not disrupt user-facing services or essential system functions [46]. By prioritizing critical components, testing can

focus on areas where failure would result in the most severe consequences. In contrast, the resilience issues of non-critical components can be addressed less urgently, since their failure poses a lower risk to overall system stability [79, 185].

6.1.4 Deployment. The type of deployment selected for chaos experiments is a critical decision in chaos engineering, as it defines the testing environment and significantly affects the performance and stability of the system [1, 111, 194]. Choosing the appropriate deployment type ensures that the experiments yield meaningful insights while minimizing the risks to live operations [157, 185]. This section identifies two primary deployment types: production and pre-production.

Production. This refers to running tests or experiments in a live environment where real users actively interact with the System Under Test (SUT) [25, 42, 128, 180, 194]. For example, deploying a new feature on an e-commerce website during sales periods allows teams to observe user interactions and system performance under actual conditions [83]. A significant benefit of production deployment is the ability to uncover vulnerabilities that might remain hidden in more isolated testing environments [127, 185]. For example, testing a service during peak traffic can reveal weaknesses that would not otherwise emerge, making it crucial to gain insight into how the SUT handles real-world disruptions and help identify areas for further optimization [37, 55, 156]. Production deployment introduces risks, as failures during testing can directly affect users and disrupt business operations [83, 147]. Production experiments must be carefully planned to mitigate these risks, such as scheduling tests during low-traffic periods and implementing automated rollback mechanisms to restore normal operations if problems occur [25, 161].

Pre-production. This environment closely mirrors the production environment, allowing teams to validate system functionality and performance before going live [25, 124, 180]. An example of a pre-production environment is a staging environment, where the SUT can be deployed with the same configurations as production, enabling comprehensive checks on how it will perform under expected loads [83, 110]. Pre-production deployment allows for complex tests that might be too risky to execute in live environments [21]. For example, simulating a network partition can reveal how well the SUT responds to such failures, helping teams resolve potential problems before they reach production [73]. This added layer of testing ensures major releases and helps reduce the risk of failure once the system is live [96]. One limitation of pre-production testing is that it may not accurately reflect the complexities of real-world operations [21, 124]. Consequently, these tests should be viewed as a complement, rather than a substitute, to production testing [67, 172, 194].

6.2 Chaos Engineering Tools

To further address **RQ3.1**, we applied the taxonomy developed in this study to evaluate ten widely used chaos engineering tools (T1–T10 in Table 13).

We identified 71 tools mentioned across academic and grey sources during our thematic analysis. We filtered this list by inspecting their GitHub repositories. We excluded tools that had been archived (read-only), were no longer maintained, or lacked sufficient documentation. This screening process reduced the set to 41 active tools with publicly available repositories. From this refined pool of 41 tools, we selected the ten most starred repositories on GitHub as of November 2024. The ‘most stars’ metric was used as a proxy for community interest and real-world relevance [23]. Each selected tool was evaluated using our proposed taxonomy, which classifies tools based on four key dimensions: execution environments, automation modes, failure injection strategies, and deployment stages. This mapping is presented in Table 13 and is intended to help engineering teams align tool selection with the architecture of the systems under test (e.g., microservices, containerized environments) and operational goals (e.g., reducing latency, enhancing scalability) [88].

For example, T1 and T2 support Kubernetes environments. T1 enables manual testing in both pre-production and production stages, while T2 extends automation with custom code and integration, targeting pre-production use. T3 to T5 offer a broader platform support, including Kubernetes, Docker, and VMs, and provide full automation features such as workflow integration, prioritization, and custom code, applicable across all deployment stages. T6 and T7 focus on Kubernetes and containers. Both support manual, semi-automated, and fully automated modes; T6 emphasizes hybrid setups with prioritization, while T7 supports Docker and targets production-grade testing. T8 and T9 operate across Kubernetes and virtual machines. T8 supports complete automation and is usable in both staging and live systems, while T9 focuses on pre-production with advanced integration options. Finally, T10 is tailored for hybrid cloud environments, supporting Kubernetes and providing full automation across both pre-production and production environments.

Table 13. Top-10 Most Used Chaos Engineering Platforms.

Tool Code	Tool Name	Execution Environment							Automation Mode			Automation Strategy				Deployment Stages		
		Endpoint				On-Premise	Cloud			Manual	Semi-Automated	Fully Automated	Prioritization	Workflow Automation	Custom Code	Auto-Integration	Pre-production	Production
		Docker	Kubernetes	Virtual Machine	Serverless Function		Private	Public	Hybrid									
T1	Chaos Monkey		✓	✓			✓	✓		✓			✓	✓	✓		✓	✓
T2	Toxiproxy	✓	✓							✓	✓	✓		✓	✓	✓	✓	✓
T3	Chaos Mesh		✓			✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
T4	ChaosBlade	✓	✓	✓		✓		✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
T5	LitmusChaos		✓	✓			✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
T6	Kube-monkey		✓			✓	✓			✓	✓	✓	✓		✓	✓	✓	✓
T7	Pumba	✓	✓			✓		✓	✓	✓	✓	✓			✓	✓	✓	✓
T8	Chaos Toolkit		✓	✓		✓		✓	✓	✓	✓	✓	✓		✓	✓	✓	✓
T9	Powerfulseal		✓	✓		✓	✓	✓		✓	✓	✓	✓		✓	✓	✓	✓
T10	Chaosk8s		✓				✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

6.3 Adoption Practices (RQ3.2)

To address **RQ3.2**, we highlight best practices and common pitfalls in adopting chaos engineering derived from our literature sources. Successfully adopting chaos engineering requires a structured approach that balances best practices with avoiding common mistakes. To minimize risk during experimentation, organizations should follow well-defined guidelines for planning, executing, and evaluating chaos experiments [73, 157]. Table 14 summarizes essential best practices, such as starting with small, controlled experiments, setting clear objectives, automating tests, and closely monitoring system behavior. Additionally, organizations should limit the scope of experiments, encourage cross-team collaboration, and continuously refine processes based on experiment outcomes. While following best practices is crucial, organizations must also avoid common pitfalls, such as those outlined in Table 15, which include failing to tailor experiments to specific systems, neglecting proper planning and hypothesis formation, and lacking adequate safety mechanisms. By adhering to best practices and avoiding these common pitfalls, organizations can confidently integrate chaos engineering into their systems and achieve successful outcomes [157]. For full details, see the code-to-theme mapping in the online appendix (Section 3.6).

Table 14. Best Practices.

No	Best Practices	Description	Articles	Total
1	Start Small and Scale Up	Start with manageable tests, then expand gradually.	[10, 25, 36, 55, 68, 73, 83, 85, 132, 136, 157, 172]	12
2	Define Clear Objectives	Set precise goals to guide each experiment's purpose and outcomes.	[5, 46, 49, 73, 79, 82, 83, 85, 89, 136, 157, 159, 172, 192]	14
3	Automate Experiments	Use automation tools to run chaos experiments regularly, ensuring consistency and frequency.	[5, 14, 15, 21, 32, 53, 68, 69, 73, 78, 79, 83, 86, 109, 124, 147, 151, 157, 159, 172, 180, 185, 192, 194]	24
4	Track and Measure Results	Monitor the system's performance during experiments and collect data to see the impact.	[14, 25, 32, 46, 55, 68, 73, 75, 78, 79, 83, 85, 86, 94, 109, 110, 136, 157, 159, 172, 173, 180, 192]	23
5	Involve all Key Stakeholders	Include team members to ensure everyone is informed and aligned.	[25, 49, 56, 68, 69, 73, 79, 85, 96, 136, 157, 172]	12
6	Document and Improve	Record experiments to continually improve the system.	[25, 68, 73, 79, 85, 89, 132, 136, 157, 172, 194]	11

Table 15. Bad Practices.

No	Bad Practices	Description	Articles	Total
1	Misaligned Chaos	Chaos experiments that don't match the system yield misleading results.	[17, 46, 55, 76, 83, 94, 103, 104, 108, 128, 134, 136, 146, 161, 172, 180, 194, 195]	18
2	Inadequate Experimentation	Without planning, clear goals, and baseline tracking, assessing chaos test impacts becomes difficult.	[1, 2, 8, 10, 14, 15, 25, 49, 53, 56, 67–69, 73, 75, 79, 82–85, 94, 96, 98, 104, 120, 126, 132, 134, 159, 163, 164, 181, 185]	33
3	Safety Mechanism Oversight	Not testing rollback and safety features risks uncontrolled failures.	[5, 8, 15, 21, 25, 36, 49, 68, 73, 78, 79, 82, 85, 87, 94, 124, 132, 136, 139, 151, 159, 163, 164, 170, 179, 180, 185]	27
4	Insufficient Tooling	Lacking observability, automation, and integration tools makes monitoring and analyzing chaos experiments difficult.	[1, 5, 10, 14, 18, 19, 25, 33, 42, 49, 50, 67, 68, 73, 75, 78, 79, 82, 83, 85, 96, 98, 103, 104, 109, 120, 124, 126, 127, 136, 141, 147, 156, 159, 164, 171, 180, 185, 192]	39
5	Poor Stakeholder Collaboration	Excluding key team members causes misalignment and resistance.	[5, 25, 50, 56, 69, 73, 83, 87, 89, 97, 104, 140, 164, 170]	14

6.4 Evaluation Approaches (RQ3.3)

Evaluation refers to determining the effectiveness of chaos engineering experiments by measuring their impact on system stability, the system's ability to handle induced failures, and the growth and maturity of the organization's chaos engineering practices. To address RQ3.3, we categorized evaluation approaches into quantitative and qualitative methods based on themes derived from our systematic coding and thematic analysis (see Section 3.5 and detailed mappings in the online appendix 3.6).

Quantitative Evaluation. This involves systematically measuring numerical data to assess a system's performance under failure conditions. It focuses on metrics that can be measured, such as response time, error rates, and system availability [59]. By comparing the behavior of the system during chaos experiments with its regular operation, engineers can identify weaknesses and areas for improvement based on objective data. We identified eight quantitative evaluation approaches, which include the following:

- **Response Time Metrics** measure the total time it takes for a system to process a request, from when it is received to when the result is returned, encompassing both processing and response delivery [1, 5, 32, 42, 49, 56, 58, 73, 89, 96, 120, 126, 157, 161]. For example, a baseline response time of 18 milliseconds might increase to 27 milliseconds during chaos experiments such as 'KillNodes', where parts of the system are intentionally shut down to test disruptions, indicating slower request processing and completion [173].
- **Latency Metrics** measure the initial delay or waiting time between when a request is made and when the system begins processing that request [14, 25, 41, 42, 55, 96, 140, 185]. It does not include the total processing time, only the delay before the system starts responding [42, 50, 53, 58, 67, 157, 191]. For example, latency can increase from 44 to 70 milliseconds during chaos experiments, signaling that the system takes longer to react to requests [86].
- **Resource Utilization Metrics** track the consumption of system resources, such as CPU and memory [14, 32, 55, 67, 73, 85, 141, 147, 156, 172, 190, 192, 194]. For example, CPU usage may spike from 50% to 80% during chaos experiments, indicating an increased strain on the system due to failures [156].
- **Error Rate Metrics** measure the frequency of errors encountered during system operation [18, 41, 42, 50, 68, 96, 146, 147, 179, 192]. For instance, an increase in the error rate from 0.5% to 5% during chaos experiments suggests that the system is struggling to handle failures effectively [195].
- **Availability Metrics** refers to a system's ability to remain operational, measured by uptime and downtime [32, 68, 108, 127, 172]. Uptime indicates how long the system is fully functional, while downtime shows when the system is unavailable [1, 50, 67, 152].
- **Throughput Metrics** measure the volume of data or transactions the system can handle in a given time [5, 96, 172, 179]. For example, throughput might drop from 1,000 transactions per second to 600 during chaos experiments, leading to delays in processing user requests or data transfers [1, 86].
- **Mean Time to Recovery (MTTR)** measures the average time required to restore the system after a failure [1, 50, 67, 164]. A higher MTTR, indicating slower recovery, reveals areas that need faster recovery processes [21, 32, 56, 164]. For example, if a cloud-based e-commerce platform experiences a server crash during a flash sale and MTTR increases from 2 to 10 minutes, it signals the need for more efficient recovery mechanisms to prevent revenue loss and customer frustration [157].
- **Mean Time Between Failures (MTBF)** refers to the average time that a system operates without failure, reflecting the reliability of the system [21, 32]. A lower MTBF suggests frequent failures, highlighting the need for improved stability [85]. For example, if a healthcare system that manages patient records experiences failures every two hours, it points to the need for a more reliable system architecture to ensure uninterrupted access to patient data during critical procedures.

Qualitative Evaluation. This gathers non-numerical insights to assess system performance during failures, focusing on user feedback, team experiences, and business impacts. By understanding these aspects, engineers gain deeper insight into the broader effects of disruptions. This complements quantitative methods by adding meaning to numerical data. We identified four qualitative evaluation approaches, which include:

- **User Experience Evaluation** involves gathering qualitative feedback on how system incidents impact end users [68, 84]. This may include conducting surveys, interviews, or usability tests to assess how failures affect users' ability to interact with the system [32, 96]. For example, users might report frustration or difficulty in completing tasks during a chaos experiment that causes slow page load times or feature outages [53, 69, 151]. This evaluation is crucial because it reveals

how technical failures affect user satisfaction and retention, providing insight beyond response time or availability figures [42, 104, 190].

- **Business Impact Evaluation** assesses how chaos experiments affect key business metrics, including revenue, customer churn, and operational efficiency [79, 83]. This involves understanding specific financial losses, customer complaints, and service-level breaches resulting from experiments [126, 132]. For instance, a chaos experiment that causes downtime during a product launch could result in significant lost sales and a spike in customer service tickets [84, 164]. This evaluation is critical to linking technical failures with direct business consequences, enabling more targeted investment in stability measures where it matters most [110, 119].
- **System Behavior and Observability** focuses on how effectively system components behave and are monitored during chaos experiments [10, 111, 120, 191]. It involves reviewing logs/traces and monitoring gaps to identify areas where failures are not easily detected or diagnosed [73, 78, 159, 194]. For example, an experiment might expose missing alerts or inaccurate telemetry data for critical services, making it harder for engineers to respond [18, 41, 111]. This evaluation helps to ensure that the system's observability tools are robust enough to detect problems early and aid in faster diagnosis and recovery [46, 53].
- **Team Feedback** captures insights from engineering teams on their ability to respond to failures during chaos experiments [55, 96, 172]. It evaluates the effectiveness of incident response, communication, and decision-making processes under system stress [156, 194]. For example, feedback might highlight confusion in handling specific failure scenarios or delays in coordinating recovery efforts [36, 79, 97]. This evaluation is essential to identify operational weaknesses in real-time incident management and to improve team workflows and readiness for actual outages [53, 89, 190].

7 Current State of Chaos Engineering (RQ4)

This section addresses **RQ4** by exploring how chaos engineering has evolved in both academic and industry contexts. Specifically, it first presents the classification of research in chaos engineering, then examines the number of sources published per year and venue type (**RQ4.1**), and finally reviews the venue rankings and key contributors in the field (**RQ4.2**).

7.1 Number of Sources by Research Type (RQ4.1)

To better understand the current state of chaos engineering research (addressing **RQ4.1**), we classified the primary studies based on the research classification framework proposed by Wieringa et al. [186]. This framework includes the following categories:

- *Evaluation research*: Investigates real-world problems through empirical methods such as case studies and surveys.
- *Philosophical papers*: Propose new conceptual models, frameworks, or perspectives.
- *Solution proposals*: Present novel techniques or tools, typically without full empirical validation.
- *Validation research*: Rigorously assess proposed solutions in controlled environments (e.g., simulations, experiments).
- *Opinion papers*: Express the author's viewpoint on a topic, often arguing for or against certain practices.
- *Experience reports*: Describe applied practices and lessons from real-world projects.

The first author did the classification, which was then reviewed by the second author, with all discrepancies resolved through discussion. The results of this classification, shown in Figure 10, indicate that solution proposal papers were the most frequent type of research between 2016 and 2024. In the academic literature, solution proposal papers were the most prevalent, with nine papers in 2021 and 2022, and slightly fewer (eight) in 2023. On the other hand, the grey literature showed a slight upward trend in experience reports, rising from six in 2022 to eight in 2023. This classification highlights the evolving distribution and focus of research activities, emphasizing the dominance of solution proposals in academic literature and the increasing role of practical experience reports in the grey literature.

7.2 Number of Sources by Year and Venue Type

To further address **RQ4.1**, we analyzed the yearly distribution of academic and grey literature on chaos engineering from 2016 to early 2024 (see Figure 9). Academic output was limited from 2016 to 2018 but grew significantly from 2019, peaking in 2021 and 2022 with 13 and 12 studies, respectively, mainly in the form of conference papers. Grey literature remained minimal until 2019, rising steadily to a peak of 12 sources in 2023, dominated by blog posts and a few white papers. A slight drop in 2024 is likely due to the incomplete year [184]. Overall, these trends underscore the growing scholarly and industrial interest

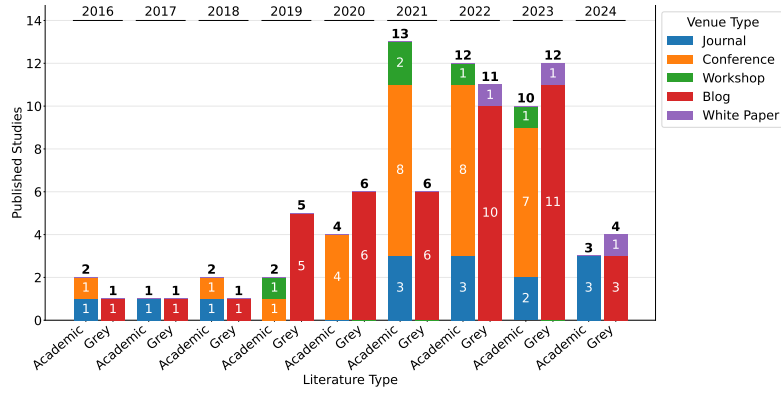


Fig. 9. Distribution of Academic and Grey Literature Studies over Years and Venue Type.

in chaos engineering. Further breakdowns of publication types, venues, and source classifications are available in our online appendix (Section 3.6).

7.3 Publication trends and key Contributors in Academic and Grey Literature (RQ4.2)

We analyzed publication trends in chaos engineering by examining venues based on their type and ranking and identifying key contributors within academic and grey literature. This analysis provides insights into the maturity of the domain [177].

7.3.1 Academic Literature. First, we analyzed publication venues based on their names, types (journal or conference), and rankings. The venues were categorized using the Computing Research and Education Association of Australasia (CORE) ranking for conferences (CORE 2023)⁴ and the Scimago ranking for journals (Scimago 2023)⁵. As shown in Figure 11, the largest share of studies appeared in unranked venues (12), though many were published in top-tier sources—particularly CORE A/A* conferences and Scimago Q1 journals, often from IEEE and ACM. This suggests increasing scholarly validation of chaos engineering research. Lower-ranked venues (B, C) and workshops contributed modestly, with few studies published in Q2 venues. To highlight influential contributions, we identified the top 10 academic studies based on citation counts from Scopus and Google Scholar (first author only). The leading study, authored by Netflix [17], has become foundational to the field, followed by diverse works from institutions in China, Germany, Sweden, and the U.S. that span topics such as end-to-end latency localization (e.g., MicroRank) [191], chaos-based security testing [172], and exception handling in JVM environments [194]. These publications reflect a global effort toward improving system resiliency and reliability through chaos engineering. Full details are provided in our online appendix (Section 3.6).

7.3.2 Grey Literature. Unlike academic publications, grey literature lacks formal rankings and peer review. To evaluate its quality and impact, we utilized alternative metrics (e.g., likes, comments, shares) and, when social engagement data were limited, applied qualitative criteria such as venue reputation, author expertise, and content clarity (see Table 5). The most engaged grey literature source was “Chaos Monkey Upgraded” by Netflix [76], which received 443 likes and 2 comments. Other widely read posts include Lyft’s framework introduction (348 likes) [152], NAB’s work on observability (156 likes) [117], and technical contributions from Expedia [87] and the Chaos Toolkit community [115]. These practitioner-authored works offer valuable, experience-based perspectives that complement academic research. Further details, including top-ranked articles and alt-metric data, are available in the online appendix (Section 3.6).

8 Discussion

In this section, we summarize the key findings of our review and discuss their implications across **RQ1** to **RQ4**. We also outline open issues in chaos engineering research and discuss potential threats to the validity of our study.

8.1 Implications of Findings

The analysis highlights five key implications for researchers and practitioners in chaos engineering:

- **Extending Reliability Testing through Chaos Engineering:** Chaos engineering should be understood not as a replacement for established reliability or software testing techniques, but as an extension that focuses on system behavior under failure and uncertainty. While traditional methods, such as unit tests, integration tests, or stress tests,

⁴<https://portal.core.edu.au/conf-ranks/>

⁵<https://www.scimagojr.com/journalrank.php>

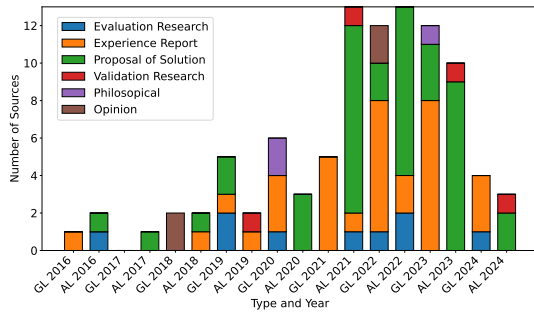


Fig. 10. Research Type of Academic (AL) and Grey (GL) Literature Studies.

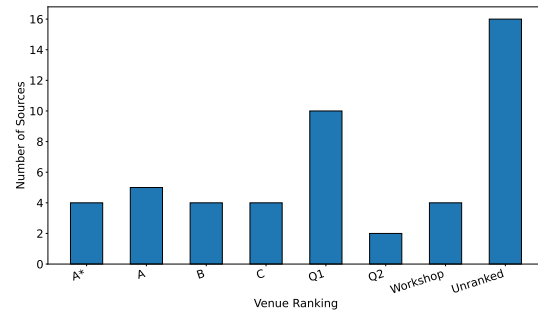


Fig. 11. Venue Ranking of Academic Literature Studies.

validate expected behaviors in controlled environments, chaos engineering introduces carefully scoped disruptions to examine how systems respond and recover from unexpected faults. This approach contributes a complementary dimension to reliability assessment, particularly in distributed or production-like environments where emergent failures often go undetected. Rather than displacing existing practices, chaos engineering enhances them by validating resilience under real-world operational stress.

- Targeted Tool Selection (RQ3.1):** This study introduces a taxonomy that helps practitioners select chaos engineering tools by considering factors such as execution environment, automation mode, and deployment stage (see Table 13). The taxonomy is intended to support informed decision-making by aligning tool capabilities with the specific infrastructure and operational needs of an organization. In practice, a practitioner would begin by assessing the architectural characteristics of their system, such as whether it runs on Kubernetes, serverless platforms, or hybrid cloud, and then identify tools from the taxonomy that match these conditions and the intended stage of testing, from early experimentation to full production deployment. Applying the taxonomy effectively also requires careful consideration of how well a tool integrates with existing workflows and team expertise. Practitioners must ensure that selected tools support their resilience objectives without introducing unnecessary complexity or risk. For example, using a highly automated tool in an immature testing environment could result in poorly scoped experiments or overlooked failure modes. Likewise, choosing tools that lack compatibility with the target infrastructure may lead to inconclusive or misleading outcomes. By mapping tools to their most appropriate use cases, the taxonomy can help mitigate such risks and promote more efficient and targeted adoption of chaos engineering practices across diverse technical landscapes.
- Standardized Practices (RQ3.2):** The findings highlight the importance of a standardized approach in chaos engineering to ensure consistent and practical application across diverse environments. Following best practices, which contribute to reliable and predictable resilience outcomes, such as faster recovery times and minimal service disruption [100], enables organizations to build resilience and systematically manage risks. These practices, such as starting with small, controlled experiments, setting clear objectives, and automating processes where possible, support organizations in achieving reliable outcomes crucial for managing complex systems (see Table 14). Conversely, neglecting these practices, such as misaligned experiment design or insufficient safety mechanisms (see Table 15), can lead to inconsistent testing and increased system vulnerability. These findings underscore the need to balance adherence to best practices with avoiding common pitfalls, ensuring that chaos engineering aligns with broader operational goals to foster stability and reliability.
- Insights through Quantitative and Qualitative Fusion (RQ3.3):** Another key implication of our findings is that combining quantitative and qualitative evaluation methods enhances the assessment and improvement of system stability in chaos engineering. Quantitative metrics provide objective data on performance under stress, including error rates, mean time to resolution, and availability. At the same time, qualitative insights, including user experience and team feedback, reveal the broader impacts of system failures. This combined approach enables organizations to identify technical vulnerabilities and understand their user and business impacts, allowing for targeted investments in both system stability and operational efficiency. This alignment of improvements with business goals and user expectations facilitates the development of solutions that effectively enhance system resilience, reduce failure impact, and improve user experience.

- **Growing Alignment of Research Theory and Practice (RQ4):** Our review empirically demonstrates the growing alignment between academic research and industry practice in chaos engineering. By classifying 96 primary sources, including both peer-reviewed papers and grey literature, we observed that solution proposal papers dominated academic publications between 2016 and 2023, while grey literature during the same period showed a slight increase in experience reports (see Figure 10). This dual trend reveals a convergence between theoretical contributions and practitioner-led experimentation, further reflected in the types of publication venues: top-ranked academic venues published novel frameworks and techniques, while widely cited blog posts detailed real-world implementation case studies. By capturing both perspectives, our study illustrates how academic research and industry practices are increasingly aligned, with researchers drawing insights from real-world applications and practitioners adopting concepts that resonate with emerging scholarly work to guide system resilience strategies. This evolving interplay signals an opportunity for deeper collaboration across academia and industry to accelerate innovation in chaos experiment design, such as advancements in automated fault orchestration, address implementation gaps, such as the lack of robust rollback or abort mechanisms in several research prototypes, and broaden the real-world adoption of chaos engineering.

8.2 Open Research Issues and Practical Challenges in Chaos Engineering

Our review highlights several ongoing challenges that limit the adoption and scalability of chaos engineering. These include cultural resistance, skill gaps, and resource overhead. The following subsections outline three key themes based on current literature and practice.

8.2.1 Organizational and Cultural Challenges. The findings highlight that organizational resistance, risk aversion, and a blame culture can pose significant barriers to the adoption of chaos engineering [22, 97]. Although chaos engineering promotes resilience, many teams perceive fault injection, especially in production, as disruptive and risky [46, 151, 194]. This perception discourages widespread adoption, as teams are often reluctant to participate in experiments that could disrupt production or damage service reliability [84, 97]. Adopting chaos engineering requires a mindset shift: from protecting systems at all costs to embracing failure as a tool for improvement [22, 25]. However, this shift is rarely supported by leadership and is often undermined by siloed teams, poor communication, and inconsistent reliability practices [1, 49, 87]. Without cultural alignment and collaboration, chaos engineering remains confined to isolated efforts with limited organizational impact [151, 170]. While these challenges are frequently noted in industry reports, academic research has yet to explore them systematically. There is a need for empirical studies that examine how factors like culture, team structure, and leadership influence chaos engineering adoption. Research could also contribute by developing and validating readiness frameworks or maturity models to guide organizations in scaling these practices effectively.

8.2.2 Skill and Expertise Gaps. Chaos Engineering requires specialized skills that many teams currently lack, creating a major barrier to adoption [96, 172]. First, effective implementation depends on understanding distributed systems, runtime behavior, and formal models like metric temporal logic, areas in which most practitioners have limited training [57, 127]. Next, managing experiment configurations, such as complex YAML files with interdependent settings, demands precision and familiarity with system internals [86]. Furthermore, many chaos engineering practices rely on intuition rather than standardized procedures, leading to inconsistent outcomes and reduced reliability [127, 179]. Adding to this challenge, some experiments require interaction with low-level components, such as kernel modules or orchestration layers, which increases the technical entry barrier [87, 120]. Although these challenges are recognized in both academic and practitioner sources, there is limited empirical research on how teams acquire and apply these skills in practice. Future work should focus on identifying specific knowledge gaps, evaluating effective training strategies, and designing practical tools and reusable templates to support broader and more consistent adoption [15, 42].

8.2.3 Resource and Operational Constraints. Chaos engineering demands significant time, computing resources, expertise, and cost, which often limits routine adoption [140, 180, 192]. Setting up environments, managing telemetry, and iterating experiments require sustained effort, especially for smaller or time-constrained teams [68, 192]. Deploying testbeds can take considerable time and need expert oversight [86]. These challenges scale with system size and are intensified by limited infrastructure, cloud expertise, and the financial overhead of tooling and operations [50, 87, 96]. As systems become increasingly complex, chaos engineering becomes more challenging to scale due to service heterogeneity and coordination overhead. Security and data concerns also arise. For example, experiments may expose vulnerabilities and generate sensitive telemetry. Managing this data poses storage, privacy, and analysis challenges. These factors—including time, complexity, and cost—often

clash with delivery timelines, leading to chaos engineering being deprioritized despite its long-term value [15, 96, 98]. Although frequently reported in practice, there is little empirical research on how teams manage these constraints. Future work could investigate how organizations reduce the setup overhead of test environments or prioritize experiments to align with available resources. Such studies would help tailor chaos engineering to practical operational limits.

8.3 Threats to Validity

We considered potential threats to the external, construct, internal, and conclusion validity [189] that may impact our study.

8.3.1 Threats to External Validity. External validity refers to the extent to which our findings can be generalized beyond the specific context of our study [189]. While this multivocal literature review (MLR) included both academic and grey sources, there is a possibility that some perspectives or practices in chaos engineering may not have been captured. A particular threat arises from the risk of excluding relevant literature that uses alternative terminology, such as *resilience testing* or *fault injection*, rather than *chaos engineering*. While we crafted inclusive search strings using multiple synonyms and related terms (see Table 3), some relevant works may still have been omitted. This limitation is common in MLRs and is explicitly acknowledged. Future studies may improve coverage by expanding search terms to encompass other reliability testing practices. Another limitation stems from the taxonomy itself, which has not yet undergone formal external validation. Although developed through thematic analysis and informed by structured internal review sessions, it remains untested in practical or industry settings. As such, its generalizability should be interpreted with caution until further validation studies are conducted. Finally, our tool analysis focused on ten widely adopted chaos engineering tools due to feasibility constraints. While these tools represent current practitioner usage, broader tool coverage, including lesser-known or homegrown solutions, could yield deeper insights. We consider this an avenue for future work.

8.3.2 Threats to Construct and Internal Validity. Construct validity concerns whether the study accurately captures the concepts it aims to investigate, while internal validity refers to the soundness of the methods used and the control over potential biases [189]. To address these concerns, we conducted at least four structured feedback sessions during our analysis. Discussions from these sessions were qualitatively examined and used to fine-tune both our methodological choices and the relevance of our findings. We have also compiled an online appendix (see Section 3.6) that contains all key artifacts, including the full list of sources, extracted codes, thematic groupings, and taxonomy classifications, to ensure transparency and reproducibility. We also adopted triangulation by synthesizing both academic and grey literature sources, thereby reducing our reliance on a single perspective. Inter-rater reliability was formally assessed during the study selection phase, using Cohen’s Kappa, which indicated strong agreement for both academic and grey literature. This ensured consistency in applying inclusion and exclusion criteria. The first author conducted manual coding, which was independently reviewed by the second author. Disagreements were resolved collaboratively through discussion. This collaborative process was applied throughout all coding phases to ensure consistency and reduce interpretation bias. Despite these precautions, some risk of observer bias remains inherent in qualitative research. However, we believe that our triangulation of sources, structured collaboration, and transparent artifact release significantly mitigate this threat.

8.3.3 Threats to Conclusion Validity. Threats to conclusion validity concern the degree to which the study’s conclusions are reasonably based on the data analyzed [189]. To mitigate this, we applied thematic coding using *ATLAS.ti* tool, and employed a structured analysis process to reduce observer and interpretation biases. Coding and classification were guided by a consensus-driven protocol between the first and second authors, ensuring consistency in the derivation of findings. Both authors independently formulated the conclusions presented in this study and later validated them through collaborative discussion. These conclusions were cross-checked against the original source material to ensure they remained grounded in the data. While our synthesis integrated both academic and grey literature, we acknowledge that our interpretations are limited to the scope of the selected articles. In particular, the absence of external expert validation for the taxonomy remains a limitation. Future research may benefit from follow-up evaluations, broader empirical testing, or practitioner feedback to confirm the applicability of our findings in real-world settings. We also acknowledge that our results are limited to the quality of the grey literature and peer-reviewed publications, which we mitigate by conducting a quality assessment analysis.

9 Conclusion and Future Work

As modern distributed systems become increasingly complex, chaos engineering offers a proactive approach to ensure stability by introducing controlled failures that reveal system weaknesses, prevent outages, and support high performance under stress.

This study bridges the gap between research insight and industry practices through a multivocal literature review (MLR) of 96 sources from academic and grey literature. To promote clarity across academia and industry, we proposed a unified definition of chaos engineering, reinforcing its role as a vital complement to traditional testing. We identified the main functionalities of a chaos engineering platform and compiled a set of activities that can guide systematic planning and conducting chaos experiments. In addition, we presented the key components of a chaos engineering platform and identified eleven quality requirements to guide the selection and evaluation of the platform. We also identified eight qualitative and four quantitative metrics essential for evaluating the impact of chaos experiments, ensuring that chaos engineering meaningfully enhances system robustness. Our review highlights three technical and four socio-technical challenges that chaos engineering seeks to resolve. To support organizations in structuring chaos engineering practices, we developed a taxonomy that organizes tools and techniques by environment, automation mode, automation strategy, and deployment stage while identifying the ten most commonly used tools within this framework. This categorization is valuable to practitioners and researchers when selecting tools that align with specific infrastructure needs and risk profiles. In addition, we identified six best practices and five common pitfalls in the implementation of chaos engineering. These practices and potential pitfalls guide organizations in optimizing their chaos engineering initiatives to foster safer and more robust systems. Our findings also reveal that solution-oriented research in academia and practical experience in industry converge, underscoring the essential role of chaos engineering in strengthening modern distributed systems. Finally, we outlined several research and practical challenges that must be addressed to realize the vision of chaos engineering fully. We believe that these identified open issues can guide future research in the domain of chaos engineering.

In future work, we plan to use GitHub repository mining to identify various open-source projects that implement chaos engineering practices and utilize associated tools. This exploration will complement the ten tools analyzed in this literature review, offering broader insights into the prevalence and practical application of chaos engineering in diverse projects. Additionally, we aim to apply chaos engineering to enhance the robustness of AI-enabled systems by evaluating how these systems respond to disruptions in accuracy, adaptability, and consistency, ensuring they continue to deliver reliable performance even under challenging conditions.

References

- [1] Kalam Abdul. 2024. Vault chaos engineering. Retrieved April 28, 2024 from <https://www.hashicorp.com/blog/vault-chaos-engineering>
- [2] Koby Aharon. 2024. Introduction to Chaos Engineering in Serverless Architectures. Retrieved April 28, 2024 from <https://www.ranthebuilder.cloud/post/introduction-to-chaos-engineering-serverless>
- [3] Waseem Ahmed and Yong Wei Wu. 2013. A survey on reliability in distributed systems. *J. Comput. System Sci.* 79, 8 (2013), 1243–1255.
- [4] Peace Aisosa. 2023. Principles of Chaos Engineering. Towards AI. <https://towardsai.net/p/l/principles-of-chaos-engineering> Last accessed: September 17, 2024.
- [5] Amro Al-Said Ahmad, Lamis F Al-Qora'n, and Ahmad Zayed. 2024. Exploring the impact of chaos engineering with various user loads on cloud native applications: an exploratory empirical study. *Computing* 106, 8 (2024), 2389–2425.
- [6] Mohammed M Alabbadi. 2011. Cloud computing for education and learning: Education and learning as a service (ELaaS). In *14th International Conference on Interactive Collaborative Learning (ICL2011)*. IEEE, Piestany, Slovakia, 589–594.
- [7] Peter Alvaro, Kolton Andrus, Chris Sanden, Casey Rosenthal, Ali Basiri, and Lorin Hochstein. 2016. Automating failure testing research at internet scale. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. 17–28.
- [8] Peter Alvaro and Severine Tymon. 2017. Abstracting the Geniuses Away from Failure Testing: Ordinary users need tools that automate the selection of custom-tailored faults to inject. *Queue* 15, 5 (2017), 29–53.
- [9] Koushik Annareddy. 2010. Security challenges in hybrid cloud infrastructures. *Aalto University* 7, 4 (2010), 1–6.
- [10] Shan Anwar and Balaji Arunachalam. 2019. Automating Resiliency: How To Remain Calm In The Midst Of Chaos. Retrieved April 28, 2024 from <https://medium.com/intuit-engineering/automating-resiliency-how-to-remain-calm-in-the-midst-of-chaos-d0d3929243ca>
- [11] Merishani Arsecularatne and Ruwan Wickramarachchi. 2023. Adoptability of Chaos Engineering with DevOps to Stimulate the Software Delivery Performance. In *2023 International Research Conference on Smart Computing and Systems Engineering (SCSE)*, Vol. 6. IEEE, 1–8.
- [12] Maricela-Georgiana Avram. 2014. Advantages and challenges of adopting cloud computing. *Procedia Tech.* 12 (2014), 529–534.
- [13] Kanchan Awasthi, Krupal Padwekar, and Subhas Chandra Misra. 2025. Digital Twin: A Unified Definition, Issues, Challenges, and Opportunities. *Encyclopedia of Information Science and Technology, Sixth Edition* (2025), 1–18.
- [14] Azure Readiness. 2023. Intro to Chaos Engineering and Azure Chaos Studio (Preview). Retrieved April 24, 2024 from <https://www.007fflearning.com/post/intro-to-chaos-engineering-and-azure-chaos-studio-preview>
- [15] Mekan Bairyev. 2023. Chaos Engineering: Principles and Best Practices. Retrieved April 28, 2024 from <https://maddevs.io/blog/chaos-engineering/>
- [16] R Balasubramanian and M Aramudhan. 2012. Security issues: public vs private vs hybrid cloud computing. *International Journal of Computer Applications* 55, 13 (2012), 35–41.
- [17] 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.
- [18] Ali Basiri, Lorin Hochstein, Nora Jones, and Haley Tucker. 2019. Automating chaos experiments in production. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, Netflix, Los Gatos, CA, 31–40.
- [19] Martin Bedoya, Sara Palacios, Daniel Diaz-López, Pantaleone Nespoli, Estefania Laverde, and Sebastián Suárez. 2023. Securing cloud-based military systems with Security Chaos Engineering and Artificial Intelligence. In *Proc. of the 18th Int. Conf. on Availability and Security*. IEEE, Belgota Germany, 1–9.

- [20] Sururah A. Bello, Lukumon O. Oyedele, Olugbenga O. Akinade, Muhammad Bilal, Juan Manuel Davila Delgado, Lukman A. Akanbi, Anuoluwapo O. Ajayi, and Hakeem A. Owolabi. 2021. Cloud computing in construction industry: Use cases, benefits and challenges. *Automation in Construction* 122 (2021), 103441. <https://doi.org/10.1016/j.autcon.2020.103441>
- [21] Sathyapriya Bhaskar. 2022. Chaos engineering: A step toward reliability. Retrieved April 28, 2024 from <https://www.virtusa.com/insights/perspectives/chaos-engineering> Published by Virtusa.
- [22] Sam Bocetta. 2019. How to Use Chaos Engineering to Break Things Productively. Retrieved April 28, 2024 from <https://www.infoq.com/articles/chaos-engineering-security-networking/>
- [23] Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Understanding the factors that impact the popularity of GitHub repositories. In *2016 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, Brazil, 334–344.
- [24] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.
- [25] Ralfs Bremmers. 2021. How Implementing Chaos Engineering Can Benefit Your Project. Retrieved April 28, 2024 from <https://www.testdevlab.com/blog/how-implementing-chaos-engineering-can-benefit-your-project>
- [26] Christoph Buck, Christian Olenberger, André Schweizer, Fabiane Völter, and Torsten Eymann. 2021. Never trust, always verify: A multivocal literature review on current knowledge and research gaps of zero-trust. *Computers & Security* 110 (2021), 102436.
- [27] Bert-Jan Butijn, Damian A Tamburri, and Willem-Jan van den Heuvel. 2020. Blockchains: a systematic multivocal literature review. *ACM Computing Surveys (CSUR)* 53, 3 (2020), 1–37.
- [28] Tammy Butow. 2018. Planning Your Own Chaos Day. Accessed: April 24, 2025. Available at: <https://www.gremlin.com/community/tutorials/planning-your-own-chaos-day>.
- [29] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. 2009. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems* 25, 6 (2009), 599–616. <https://doi.org/10.1016/j.future.2008.12.001>
- [30] Oliver Byström. 2022. A comparison between on-premise and cloud environments in terms of security: With an emphasis on Software-as-a-Service & Platform-as-a-Service.
- [31] Rocco Caferra, John D Hey, Andrea Morone, and Marco Santorsola. 2023. Dynamic inconsistency under ambiguity: An experiment. *Journal of Risk and Uncertainty* 67, 3 (2023), 215–238.
- [32] Carlos Camacho, Pablo C Cañizares, Luis Llana, and Alberto Núñez. 2022. Chaos as a Software Product Line—a platform for improving open hybrid-cloud systems resiliency. *Software: Practice and Experience* 52, 7 (2022), 1581–1614.
- [33] Matteo Camilli, Antonio Guerriero, Andrea Janes, Barbara Russo, and Stefano Russo. 2022. Microservices integrated performance and reliability testing. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*. ACM, Bolzano, Italy, 29–39.
- [34] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2017. Serverless programming (function as a service). In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, Watson Research Center, 2658–2659.
- [35] Pushpalika Chatterjee. 2023. Cloud-Native Architecture for High-Performance Payment System. (2023).
- [36] Guo Chen, Guotao Bai, Chun Zhang, Juan Wang, Kang Ni, and Zhi Chen. 2022. Big data system testing method based on chaos engineering. In *2022 IEEE 12th International Conference on Electronics Information and Emergency Communication (ICEIEC)*. IEEE, Beijing, China, 210–215.
- [37] Hongyang Chen, Pengfei Chen, Guangba Yu, Xiaoyun Li, Zilong He, and Huxing Zhang. 2024. MicroFI: Non-Intrusive and Prioritized Request-Level Fault Injection for Microservice Applications. *IEEE Transactions on Dependable and Secure Computing* 21, 1 (2024), 1–18.
- [38] Jessica Chen, Robert M Hierons, and Hasan Ural. 2006. Overcoming observability problems in distributed test architectures. *University of Windsor* 98, 5 (2006), 177–182.
- [39] Lianping Chen, Muhammad Ali Babar, and He Zhang. 2010. Towards an evidence-based understanding of electronic data sources. In *14th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. BCS Learning & Development, Limerick, Ireland, 1–4.
- [40] Carlos Colman-Meixner, Chris Develder, Massimo Tornatore, and Biswanath Mukherjee. 2016. A survey on resiliency techniques in cloud computing infrastructures and applications. *IEEE Communications Surveys & Tutorials* 18, 3 (2016), 2244–2281.
- [41] Adrian Colyer. 2019. Automating chaos experiments in production. Retrieved April 28, 2024 from <https://blog.acolyer.org/2019/07/05/automating-chaos-experiments-in-production/>
- [42] Domenico Cotroneo, Luigi De Simone, and Roberto Natella. 2022. Thorfi: a novel approach for network fault injection as a service. *Journal of Network and Computer Applications* 201 (2022), 103334.
- [43] L Minh Dang, Md Jalil Piran, Dongil Han, Kyungbok Min, and Hyeonjoon Moon. 2019. A survey on internet of things and cloud computing for healthcare. *Electronics* 8, 7 (2019), 768.
- [44] Czesław Danilowicz and Ngoc Thanh Nguyen. 2003. Consensus methods for solving inconsistency of replicated data in distributed systems. *Distributed and Parallel Databases* 14 (2003), 53–69.
- [45] Gert-Jan de Vreede, Pedro Antunes, Julita Vassileva, Marco Aurélio Gerosa, and Kewen Wu. 2016. Collaboration technology in teams and publishers: Introduction to the special issue. *Information Systems Frontiers* 18 (2016), 1–6.
- [46] Panagiotis Dedousis, George Stergiopoulos, George Arampatzis, and Dimitris Gritzalis. 2023. Enhancing Operational Resilience of Critical Infrastructure Processes Through Chaos Engineering. *IEEE Access* 11 (2023), 106172–106189.
- [47] Merkebu Zenebe Degefa, Iver Bakken Sperstad, and Hanne Sæle. 2021. Comprehensive classifications and characterizations of power system flexibility resources. *Electric Power Systems Research* 194 (2021), 107022.
- [48] Josu Diaz-De-Arcaya, Juan López-De-Armentia, Raúl Miñón, Iker Lasa Ojaguren, and Ana I Torre-Bastida. 2024. Large Language Model Operations (LLMOps): Definition, Challenges, and Lifecycle Management. In *2024 9th Int. Conf. on Smart and Sustainable Technologies*. IEEE, 1–4.
- [49] Ashwin Dua. 2024. What Is Chaos Engineering and What Are Its Benefits? Retrieved April 28, 2024 from <https://www.turing.com/blog/chaos-engineering-and-its-benefits>
- [50] Sindhuja Durai. 2022. Chaos Testing an Application on AWS. Retrieved April 28, 2024 from <https://developer.gs.com/blog/posts/chaos-testing-an-application-on-aws>
- [51] Pranay Dutta and Prashant Dutta. 2019. Comparative study of cloud services offered by Amazon, Microsoft & Google. *International Journal of Trend in Scientific Research and Development* 3, 3 (2019), 981–985.
- [52] Kleinner Farias, Alessandro Garcia, and Carlos Lucena. 2012. Evaluating the impact of aspects on inconsistency detection effort: a controlled experiment. In *Model Driven Engineering Languages and Systems: 15th Int. Conf., MODELS 2012. Proc. 15*. Springer, Austria, 219–234.
- [53] Amanda Fawcett. 2020. Chaos engineering 101: Principles, process, and examples. Retrieved April 28, 2024 from <https://www.educative.io/blog/chaos-engineering-process-principles>
- [54] Colin Fidge. 1996. Fundamentals of distributed system observation. *IEEE Software* 13, 6 (1996), 77–83.

- [55] Mattia Fogli, Carlo Giannelli, Filippo Poltronieri, Cesare Stefanelli, and Mauro Tortonesi. 2023. Chaos engineering for resilience assessment of digital twins. *IEEE Transactions on Industrial Informatics* 20, 2 (2023), 1134–1143.
- [56] Sebastian Frank, Alireza Hakamian, Lion Wagner, Dominik Kesim, Christoph Zorn, Jóakim von Kistowski, and André van Hoorn. 2021. Interactive elicitation of resilience scenarios based on hazard analysis techniques. In *European Conf. on Software Architecture*. Springer, Stuttgart Germany, 229–253.
- [57] Sebastian Frank, Alireza Hakamian, Denis Zahariev, and André van Hoorn. 2023. Verifying transient behavior specifications in chaos engineering using metric temporal logic and property specification patterns. In *2023 ACM/SPEC Int. Conf. on Performance Engineering*. ACM, Stuttgart Germany, 319–326.
- [58] Sebastian Frank, M Alireza Hakamian, Lion Wagner, Dominik Kesim, Jóakim von Kistowski, and André van Hoorn. 2021. Scenario-based Resilience Evaluation and Improvement of Microservice Architectures: An Experience Report. In *ECSA (Companion)*. Scopus, Stuttgart, Germany, 1–10.
- [59] Saurabh Kumar Garg, Steve Versteeg, and Rajkumar Buyya. 2013. A framework for ranking of cloud computing services. *Future Generation Computer Systems* 29, 4 (2013), 1012–1023.
- [60] Vahid Garousi and Michael Felderer. 2017. Experience-based guidelines for effective and efficient data extraction in systematic reviews in software engineering. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. ACM, Ankara, Turkey, 170–179.
- [61] Vahid Garousi, Michael Felderer, and Tuna Hacaloğlu. 2017. Software test maturity assessment and test process improvement: A multivocal literature review. *Information and Software Technology* 85 (2017), 16–42.
- [62] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. 2016. The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature. In *Proc. of the 20th int. conf. on evaluation and assessment in software engineering*. Elsevier, Ankara, Turkey, 1–6.
- [63] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. 2019. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and software technology* 106 (2019), 101–121.
- [64] Vahid Garousi and Mika V Mäntylä. 2016. When and what to automate in software testing? A multi-vocal literature review. *Information and Software Technology* 76 (2016), 92–117.
- [65] Justin Garrison and Kris Nova. 2017. *Cloud native infrastructure: Patterns for scalable infrastructure and applications in a dynamic environment*. " O'Reilly Media, Inc.", Tokyo.
- [66] Bernd Gastermann, Markus Stopper, Anja Kossik, and Branko Katalinic. 2015. Secure implementation of an on-premises cloud storage service for small and medium-sized enterprises. *Procedia Engineering* 100 (2015), 574–583.
- [67] Neharika Gianchandani, Dushyant Anoop Sahni, and Ramanpreet Singh. 2022. Is chaos engineering exclusive to Netflix? Well, no, it's for you too! Retrieved April 28, 2024 from <https://www.nagarro.com/en/blog/chaos-engineering-best-practices>
- [68] Navdeep Singh Gill. 2021. Chaos Engineering Principles, Tools and Best Practices. Retrieved April 28, 2024 from <https://www.xenonstack.com/insights/chaos-engineering>
- [69] Navdeep Singh Gill. 2022. Chaos Engineering For Cloud Native - A Definitive Guide. Retrieved April 28, 2024 from <https://www.xenonstack.com/blog/chaos-engineering-for-cloud-native> Accessed: 2024-07-28.
- [70] Cheng Gong and Vincent Ribiere. 2021. Developing a unified definition of digital transformation. *Technovation* 102 (2021), 102217.
- [71] Eugene Gorelik. 2013. *Cloud computing models*. Ph.D. Dissertation. Massachusetts Institute of Technology.
- [72] Sumit Goyal. 2014. Public vs private vs hybrid vs community-cloud computing: a critical review. *International Journal of Computer Network and Information Security* 6, 3 (2014), 20–29.
- [73] Simon Green. 2023. SRE's Guide to Chaos Engineering: Embrace the Chaos for Resilience. Retrieved April 28, 2024 from <https://www.linkedin.com/pulse/sres-guide-chaos-engineering-embrace-resilience-simon-green/>
- [74] Orabi Habeh, Firas Thekrallah, Said A Salloum, and Khaled Shaalan. 2021. Knowledge sharing challenges and solutions within software development team: a systematic review. *Recent Advances in Intelligent Systems and Smart Applications* 8, 9 (2021), 121–141.
- [75] Greg Hawkins. 2020. The Abyss of Ignorable: a Route into Chaos Testing from Starling Bank. Retrieved April 28, 2024 from <https://www.infoq.com/articles/chaos-testing-starling-bank/>
- [76] Lorin Hochstein and Casey Rosenthal. 2016. Netflix Chaos Monkey Upgraded. Accessed: April 24, 2025. Available at: <https://netflixtechblog.com/netflix-chaos-monkey-upgraded-1d679429be5d>.
- [77] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. 1997. Fault injection techniques and tools. *Computer* 30, 4 (1997), 75–82.
- [78] Hiroki Ikeuchi, Jiawen Ge, Yoichi Matsuo, and Keishiro Watanabe. 2020. A framework for automatic failure recovery in ict systems by deep reinforcement learning. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, Tokyo Japan, 1310–1315.
- [79] Gremlin Inc. 2023. Chaos Engineering: the history, principles, and practice. Retrieved April 28, 2024 from <https://www.gremlin.com/community/tutorials/chaos-engineering-the-history-principles-and-practice>
- [80] Chadni Islam, Muhammad Ali Babar, and Surya Nepal. 2019. A multi-vocal review of security orchestration. *Comput. Surveys* 52, 2 (2019), 1–45.
- [81] Yashpalsinh Jadeja and Kirit Modi. 2012. Cloud computing-concepts, architecture and challenges. In *2012 international conference on computing, electronics and electrical technologies (ICCEET)*. IEEE, Nagercoil, India, 877–880.
- [82] Madhuri Jakkaraju. 2020. 5 steps to getting your app chaos ready. Retrieved April 28, 2024 from <https://www.capitalone.com/tech/software-engineering/is-your-app-chaos-engineering-ready/>
- [83] Hugo Jernberg, Per Runeson, and Emelie Engström. 2020. Getting Started with Chaos Engineering-design of an implementation framework in practice. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, Lund Sweden, 1–10.
- [84] Zhenlan Ji, Pingchuan Ma, and Shuai Wang. 2023. Perfce: Performance debugging on databases with chaos engineering-enhanced causality analysis. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Bangalore India, 1454–1466.
- [85] Ruturaj Kadikar. 2023. Building Resilience with Chaos Engineering and Litmus. Retrieved April 28, 2024 from <https://www.infracloud.io/blogs/building-resilience-chaos-engineering-litmus/>
- [86] Ayham Kassab, Etienne Rivière, Guillaume Rosinosky, Ramin Sadre, and Viet Hoang Tran. 2022. C2B2: a Cloud-native Chaos Benchmarking suite for the Hyperledger Fabric Blockchain. In *2022 18th European Dependable Computing Conference (EDCC)*. IEEE, Icteam Belgium, 89–96.
- [87] Nikos Katirtzis. 2022. Chaos Engineering at Expedia Group. Retrieved April 28, 2024 from <https://medium.com/expedia-group-tech/chaos-engineering-at-expedia-group-e51a0288ee2>
- [88] David Kavalier, Asher Trockman, Bogdan Vasilescu, and Vladimir Filkov. 2019. Tool choice matters: JavaScript quality assurance tools and usage outcomes in GitHub projects. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, University of California, Davis, 476–487.
- [89] Dominik Kesim, André van Hoorn, Sebastian Frank, and Matthias Häussler. 2020. Identifying and prioritizing chaos experiments by using established risk analysis techniques. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Stuttgart Germany, 229–240.
- [90] Rohit Khankhoje. 2022. Beyond Coding: A Comprehensive Study of Low-Code, No-Code and Traditional Automation. *Journal of Artificial Intelligence & Cloud Computing*. SRC/JAICC-160. DOI: [doi.org/10.47363/JAICC/2022\(1\)148](https://doi.org/10.47363/JAICC/2022(1)148) (2022), 2–5.
- [91] Barbara Kitchenham. 2004. Procedures for performing systematic reviews. *Keele, UK, Keele University* 33, 2004 (2004), 1–26.

- [92] Barbara Kitchenham, Stuart Charters, et al. 2007. Guidelines for performing systematic literature reviews in software engineering.
- [93] Floriment Klinaku, Martina Rapp, Jörg Henss, and Stephan Rhode. 2022. Beauty and the beast: A case study on performance prototyping of data-intensive cloud applications. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*. ACM, Stuttgart, Germany, 53–60.
- [94] Charalambos Konstantinou, George Stergiopoulos, Masood Parvania, and Paulo Esteves-Verissimo. 2021. Chaos engineering for enhanced resilience of cyber-physical systems. In *2021 Resilience Week (RWS)*. IEEE, Saudi Arabia, 1–10.
- [95] Mikael Koskinen, Tommi Mikkonen, and Pekka Abrahamsson. 2019. Containers in software development: A systematic mapping study. In *International conference on product-focused software process improvement*. Springer, University of Helsinki, Finland, 176–191.
- [96] Nikola Kostic. 2024. Chaos Engineering: Definition, Principles, Best Practices. Retrieved April 28, 2024 from <https://phoenixnap.com/blog/chaos-engineering>
- [97] Narayanan Krishnamurthy. 2021. Chaos Is Good! — In Tech. Retrieved April 28, 2024 from <https://eng.lifion.com/chaos-is-good-in-tech-2c487fce102f>
- [98] Andreas Krivas and Rafael Portela. 2020. Comparing Chaos Engineering Tools for Kubernetes Workloads. Retrieved April 28, 2024 from <https://blog.container-solutions.com/comparing-chaos-engineering-tools>
- [99] Santosh Kumar and RH Goudar. 2012. Cloud computing-research issues, challenges, architecture, platforms and applications: a survey. *International Journal of Future Computer and Communication* 1, 4 (2012), 356.
- [100] Indika Kumara, Martín Garriga, Angel Urbano Romeu, Dario Di Nucci, Fabio Palomba, Damian Andrew Tamburri, and Willem-Jan van den Heuvel. 2021. The do's and don'ts of infrastructure code: A systematic gray literature review. *Information and Software Technology* 137 (2021), 106593.
- [101] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics* (1977), 159–174.
- [102] Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. 2021. A systematic review on software robustness assessment. *ACM CSUR* 54, 4 (2021), 1–65.
- [103] Doug Lardo. 2019. Controlled Chaos with Fault Injection Testing. Retrieved April 28, 2024 from <https://technology.riotgames.com/news/controlled-chaos-fault-injection-testing>
- [104] Andy Le. 2022. Chaos Engineering in Accounting team. Retrieved April 28, 2024 from <https://engineering.zalopay.vn/how-we-apply-chaos-engineering/>
- [105] Rakesh Kumar Lenka, Sarthak Padhi, and Kabita Manjari Nayak. 2018. Fault injection techniques-a brief review. In *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*. IEEE, Greater Noida, India, 832–837.
- [106] Chunxiao Li, Anand Raghunathan, and Niraj K Jha. 2011. A trusted virtual machine in an untrusted management environment. *IEEE Transactions on services computing* 5, 4 (2011), 472–483.
- [107] Yuqian Lu, Xun Xu, and Jenny Xu. 2014. Development of a hybrid manufacturing cloud. *Journal of manufacturing systems* 33, 4 (2014), 551–566.
- [108] Fuchen Ma, Yuanliang Chen, Yuanhang Zhou, Jingxuan Sun, Zhuo Su, Jiaguang Jiang, and Huizhong Li. 2023. Phoenix: Detect and locate resilience issues in blockchain via context-sensitive chaos. In *Proc. of the 2023 ACM SIGSAC Conf. on Computer and Communications Security*. ACM, Beijing, China, 1182–1196.
- [109] Sehrish Malik, Moeen Ali Naqvi, and Leon Moonen. 2023. CHESS: A Framework for Evaluation of Self-adaptive Systems based on Chaos Engineering. In *2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, Oslo Norway, 195–201.
- [110] Neelanjan Manna. 2021. Part-2: A Beginner's Practical Guide to Containerisation and Chaos Engineering with LitmusChaos 2.0. Retrieved April 28, 2024 from <https://medium.com/litmus-chaos/a-beginners-practical-guide-to-containerisation-and-chaos-engineering-with-litmuschaos-2-0-5f4f3cf2a55d>
- [111] Christopher S Meiklejohn, Andrea Estrada, Yiwen Song, Heather Miller, and Rohan Padhye. 2021. Service-level fault injection testing. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, PA, United States, 388–402.
- [112] Nihar Ajay Mhatre, Mugdha Shailendra Kulkarni, and Fatima Ali. 2024. The Role of Chaos Engineering in DevOps for Software Robustness. In *Applied Intelligence and Computing*. Mukesh Saraswat and Rajani Kumari (Eds.). Symbiosis Centre for Information Technology, Symbiosis International (Deemed University), Computing & Intelligent Systems, SCRS, India, Pune, India, 9–17. <https://doi.org/10.56155/978-81-955020-9-7-2>
- [113] Microsoft. 2024. What is an endpoint? Microsoft. <https://www.microsoft.com/en-us/security/business/security-101/what-is-an-endpoint?msocid=399306b6d8e86cfc287812c0d9446d18> Last accessed: October 18, 2024.
- [114] Samuel Migirditch, John Asplund, and William Curran. 2022. Chaos engineering: stress-testing algorithms to facilitate resilient strategic military planning. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 2160–2167.
- [115] Russ Miles. 2019. Chaos Engineering with Humans in the Loop. Retrieved April 28, 2024 from <https://medium.com/chaos-toolkit/chaos-engineering-with-humans-in-the-loop-f4854900b1eb>
- [116] Russ Miles. 2019. *Learning Chaos engineering: discovering and overcoming system weaknesses through experimentation*. O'Reilly Media, USA.
- [117] Subhra Mondal and Prateek Sachan. 2020. Observability in the realm of Chaos Engineering. Retrieved April 28, 2024 from <https://medium.com/@nabtechblog/observability-in-the-realm-of-chaos-engineering-99089226ca51>
- [118] Sophia Monroe. 2019. Investigate Methodologies for Intentionally Introducing Failures to Improve System Resilience and Fault Tolerance. *International Journal of Artificial Intelligence and Machine Learning in Engineering* 405 (2019), 405–418.
- [119] Matthew Moon. 2022. Causing Chaos. Retrieved April 28, 2024 from <https://medium.com/justeattakeaway-tech/causing-chaos-3ab9bb5a7235>
- [120] Mallory Mooney. 2023. Security-focused chaos engineering experiments for the cloud. Retrieved April 28, 2024 from <https://www.datadoghq.com/blog/chaos-engineering-for-security/>
- [121] Andrea Morichetta, Nikolaus Spring, Philipp Raith, and Schahram Dustdar. 2023. Intent-based management for the distributed computing continuum. In *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 239–249.
- [122] Brad Myers, Scott E Hudson, and Randy Pausch. 2000. Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)* 7, 1 (2000), 3–28.
- [123] Lavan Nallainathan. 2023. Mitigating Downtime and Increasing Reliability: Strategies for Managing Complexity in the Cloud. <https://techcommunity.microsoft.com/t5/azure-architecture-blog/mitigating-downtime-and-increasing-reliability-strategies-for/ba-p/3810399> Last accessed: April 28, 2024.
- [124] Moeen Ali Naqvi, Sehrish Malik, Merve Astekin, and Leon Moonen. 2022. On evaluating self-adaptive and self-healing systems using chaos engineering. In *2022 IEEE international conference on autonomic computing and self-organizing systems (ACSOS)*. IEEE, CA, USA, 1–10.
- [125] Roberto Natella, Domenico Cotroneo, and Henrique S Madeira. 2016. Assessing dependability with software fault injection: A survey. *ACM Computing Surveys (CSUR)* 48, 3 (2016), 1–55.
- [126] National Australia Bank. 2020. Observability in the realm of Chaos Engineering. Retrieved April 23, 2024 from <https://medium.com/@nabtechblog/observability-in-the-realm-of-chaos-engineering-99089226ca51> Medium.
- [127] Fotis Nikolaidis, Antony Chazapis, Manolis Marazakis, and Angelos Bilas. 2021. Frisbee: automated testing of Cloud-native applications in Kubernetes. *arXiv preprint arXiv:2109.10727* abs/2109.10727, 6 (2021), 1–14.
- [128] Fotis Nikolaidis, Antony Chazapis, Manolis Marazakis, and Angelos Bilas. 2023. Event-Driven Chaos Testing for Containerized Applications. In *International Conference on High Performance Computing*. Springer, Rethimno Greece, 144–157.
- [129] Jesús Gil Nombela. 2023. Chaos Engineering: The Art of introduce Intentional Failures. Retrieved April 28, 2024 from <https://www.santanderconsumergs.com/news/https-impulsate-between-tech-chaos-engineering>

- [130] Guruprasad Nookala. 2023. Serverless Data Architecture: Advantages, Drawbacks, and Best Practices. *Journal of Computing and Information Technology* 3, 1 (2023).
- [131] Santeri Paavolainen. 2016. *Observed Availability of Cloud Services*. Master's thesis. University of Helsinki.
- [132] R. Palani and J. Gupta. 2023. Adopting Chaos Engineering. LTIMindtree. <https://www.ltimindtree.com/wp-content/uploads/2023/09/Adopting-Chaos-Engineering-WP.pdf> Retrieved April 28, 2024.
- [133] Ragupathi Palani and Joydeep Gupta. 2023. Adopting Chaos Engineering. Retrieved April 28, 2024 from <https://www.ltimindtree.com/wp-content/uploads/2023/09/Adopting-Chaos-Engineering-WP.pdf> © LTIMindtree | Privileged and Confidential.
- [134] Sumin Park, Zelalem Mihret Belay, and Doo-Hwan Bae. 2019. A simulation-based behavior analysis for mci response system of systems. In *Proc. of the 2019 IEEE/ACM 7th Intl. Workshop on SESoS and 13th WDES*. IEEE, South Korea, 2–9.
- [135] Brian Parsons. 2021. Using Chaos Engineering to Improve the Resiliency of Transportation Cyber Physical Systems. In *INCOSE Americas Sector 14th Annual North-Central and Great Lakes Regional Conference*. International Council on Systems Engineering, North-Central and Great Lakes Region.
- [136] Viral Patel. 2022. What Is Chaos Engineering and Why You Should Break More Things On Purpose. Retrieved April 28, 2024 from <https://www.contino.io/insights/chaos-engineering>
- [137] Riccardo Patriarca, Andrea Falegnami, Francesco Costantino, and Federico Bilotta. 2018. Resilience engineering for socio-technical risk analysis: Application in neuro-surgery. *Reliability Engineering & System Safety* 180 (2018), 321–335.
- [138] Siani Pearson. 2013. *Privacy, security and trust in cloud computing*. Springer, Bristol, UK.
- [139] Tony Pierce, Jason Schanck, Alex Groeger, Raed Salih, and Michael R Clark. 2021. Chaos engineering experiments in middleware systems using targeted network degradation and automatic fault injection. In *Open Architecture/Open Business Model Net-Centric Systems and Defense Transformation 2021*, Vol. 11753. SPIE, United States, 24–36.
- [140] Filippo Poltronieri, Mauro Tortonesi, and Cesare Stefanelli. 2022. A chaos engineering approach for improving the resiliency of it services configurations. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*. IEEE, Ferrara Italy, 1–6.
- [141] Lucas Eduardo Gulka Pulcinelli, Diego Frazatto Pedroso, and Sarita Mazzini Bruschi. 2023. Conceptual and comparative analysis of application metrics in microservices. In *2023 International Symposium on Computer Architecture and High Performance Computing Workshops*. IEEE, Carlos Brazil, 123–130.
- [142] Akond Rahman, Dibyendu Brinto Bose, Farhat Lamia Barsha, and Rahul Pandita. 2023. Defect Categorization in Compilers: A Multi-vocal Literature Review. *Comput. Surveys* 56, 4 (2023), 1–42.
- [143] Arokia Paul Rajan. 2020. A review on serverless architectures-function as a service (FaaS) in cloud computing. *TELKOMNIKA (Telecommunication Computing Electronics and Control)* 18, 1 (2020), 530–537.
- [144] Andreas Riege. 2005. Three-dozen knowledge-sharing barriers managers must consider. *Journal of knowledge management* 9, 3 (2005), 18–35.
- [145] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. 2009. A taxonomy and survey of cloud computing systems. In *2009 fifth international joint conference on INC, IMS and IDC*. IEEE, Kookmin University in Seoul, South Korea, 44–51.
- [146] Luis F Rivera, Norha M Villegas, Gabriel Tamura, Hausi A Muller, Ian Watts, Eric Erpenbach, and Xiaotong Shwartz. 2023. Using Digital Twins for Software Change Risk Assessment. In *Proc. of the 33rd Annual CASCOn: Intl. Conf. on Computer Science and Software Engineering*. ACM, BC Canada, 211–216.
- [147] Yury Niño Roa. 2022. Chaos Engineering and Observability with Visual Metaphors. Retrieved April 28, 2024 from <https://www.infoq.com/articles/chaos-engineering-observability-visual-metaphors/> Reviewed by Ben Linders.
- [148] Seyed Reza Rouholamini, Meghdad Mirabi, Razieh Farazkish, and Amir Sahafi. 2020. Proactive self-healing techniques for cloud computing: A systematic review. *Concurrency and Computation: Practice and Experience* 23, 3 (2020), e8246.
- [149] Johnny Saldaña. 2021. The coding manual for qualitative researchers. *an international journal* 12, 2 (2021), 169–170.
- [150] Manish Saraswat and RC Tripathi. 2020. Cloud computing: Comparison and analysis of cloud service providers-AWs, Microsoft and Google. In *2020 9th international conference system modeling and advancement in research trends (SMART)*. IEEE, UP, India, 281–285.
- [151] Takieddine Shiai, Gorik Van Steenberge, Adrian Hornsby, and Milosz Kosmider. 2023. Lessons from Amazon Search's Chaos Engineering Journey. Retrieved April 28, 2024 from <https://community.aws/content/2gBgHy9s00swu4qCxCkUf1b8fDiP/amazon-search-chaos-engineering-journey?lang=en>
- [152] Kathan Shah. 2021. Chaos Experimentation, an open-source framework built on top of Envoy Proxy. Retrieved April 28, 2024 from <https://eng.lyft.com/chaos-experimentation-an-open-source-framework-built-on-top-of-envoy-proxy-df87519ed681>
- [153] Prateek Sharma, Lucas Chaufourrier, Prashant Shenoy, and YC Tay. 2016. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th international middleware conference*. Springer, Amherst, USA, 1–13.
- [154] Pareek Chandra Shekhar. 2024. Chaos Testing: A Proactive Framework for System Resilience in Distributed Architectures. *International Journal of Science and Research (IJSR)* 13, 11 (2024), 851–855. <https://doi.org/10.21275/SR241110081650> Fully Refereed, Open Access, Double Blind Peer Reviewed.
- [155] Scheila Farias Silveira. 2023. Fault Tolerance in Microservices: Ensuring Service Resilience and High Availability. <https://ubiminds.com/en-us/fault-tolerance/> Last accessed: April 28, 2024.
- [156] Jesper Simonsson, Long Zhang, Brice Morin, Benoit Baudry, and Martin Monperrus. 2021. Observability and chaos engineering on system calls for containerized applications in docker. *Future Generation Computer Systems* 122 (2021), 117–129.
- [157] Gautam Siwach, Adinarayana Haridas, and Nagaraj Chinni. 2022. Evaluating operational readiness using chaos engineering simulations on kubernetes architecture in big data. In *2022 International Conference on Smart Applications, Communications and Networking (SmartNets)*. IEEE, Hyderabad India, 1–7.
- [158] James E Smith and Ravi Nair. 2005. The architecture of virtual machines. *Computer* 38, 5 (2005), 32–38.
- [159] Jacopo Soldani and Antonio Brogi. 2021. Automated generation of configurable cloud-native chaos testbeds. In *Dependable Computing-EDCC 2021 Workshops: DREAMS, DSOGRI, SERENE 2021, Munich, Germany, September 13, 2021, Proceedings 17*. Springer, Pisa, Italy, 101–108.
- [160] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. 2018. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software* 146 (2018), 215–232.
- [161] Shiv Sondhi, Sherif Saad, Kevin Shi, Mohammad Mamun, and Issa Traore. 2021. Chaos engineering for understanding consensus algorithms performance in permissioned blockchains. In *Proc. of the 2021 IEEE Intl. Conf. on DASC, PiCom, CBDCom, and CyberSciTech*. IEEE, Canada, 51–59.
- [162] Chi-hoon Song and Young-woo Sohn. 2022. The influence of dependability in cloud computing adoption. *The Journal of Supercomputing* 78, 10 (2022), 12159–12201.
- [163] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. 2018. Engineering software for the cloud: External monitoring and failure injection. In *Proceedings of the 23rd European conference on pattern languages of programs*. 1–8.
- [164] Richard Starr, Animesh Kundu, and Haresh Nandwani. 2022. Automating and Scaling Chaos Engineering using AWS Fault Injection Simulator. Retrieved April 28, 2024 from <https://aws.amazon.com/blogs/industries/automating-and-scaling-chaos-engineering-using-aws-fault-injection-simulator/>
- [165] Lauren Stewart. 2024. Sampling Bias in Research: How to Avoid it. <https://atlati.com/research-hub/sampling-bias> Last accessed: April 28, 2024.
- [166] Roy Suddaby. 2010. Editor's comments: Construct clarity in theories of management and organization. , 346–357 pages.

- [167] Davide Taibi, Nabil El Ioini, Claus Pahl, and Jan Raphael Schmid Niederkofler. 2020. Patterns for serverless functions (function-as-a-service): A multivocal literature review. In *Proceedings of the 10th International Conference on Cloud Computing and Services Science* 6, 4 (2020), 181–192.
- [168] John E Thomas, Daniel A Eisenberg, Thomas P Seager, and Erik Fisher. 2019. A resilience engineering approach to integrating human and socio-technical system capacities and processes for national infrastructure resilience. *Journal of Homeland Security and Emergency Management* 16, 2 (2019), 20170019.
- [169] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70–85.
- [170] Ian Tivey, Brett Aukburg, Paul Jones, Jim Oulton, and Ming Zheng. 2019. Implementing Chaos Engineering for Financial Services. Retrieved April 28, 2024 from <https://www.synchrotron.com/sites/default/files/2022-04/Implementing-chaos-engineering-continuous-compliance-for-financial-services.pdf>
- [171] Martin Tomka. 2024. Resilience and Chaos Engineering. Retrieved April 28, 2024 from <https://devblogs.microsoft.com/dotnet/resilience-and-chaos-engineering/>
- [172] Kennedy A Torkura, Muhammad IH Sukmana, Feng Cheng, and Christoph Meinel. 2020. Cloudstrike: Chaos engineering for security and resiliency in cloud infrastructure. *IEEE Access* 8 (2020), 123044–123060.
- [173] Kennedy A Torkura, Muhammad IH Sukmana, Feng Cheng, and Christoph Meinel. 2021. Continuous auditing and threat detection in multi-cloud infrastructure. *Computers & Security* 102 (2021), 102124.
- [174] Haley Tucker, Lorin Hochstein, Nora Jones, Ali Basiri, and Casey Rosenthal. 2018. The business case for chaos engineering. *IEEE Cloud Computing* 5, 3 (2018), 45–54.
- [175] Tuomas Väisänen. 2023. *Security review of Cloud Application architectures*. Master's thesis. Aalto University, Espoo, Finland. <https://aaltodoc.aalto.fi/items/9ff107f8-4ca4-46c3-8f86-ba0723e973c0> Thesis submitted for examination for the degree of Master of Science in Technology.
- [176] Erwin van Eyk and Alexandru Iosup. 2018. Addressing performance challenges in serverless computing. *Proc. ICT. Open* 4, 6 (2018), 1–2.
- [177] Ángel Jesús Varela-Vaca and Antonia M Reina Quintero. 2021. Smart contract languages: A multivocal mapping study. *ACM Computing Surveys (CSUR)* 54, 1 (2021), 1–38.
- [178] Roberto Verdecchia, Ivana Malavolta, and Patricia Lago. 2019. Guidelines for architecting android apps: A mixed-method empirical study. In *2019 IEEE International Conference on Software Architecture (ICSA)*. IEEE, Amsterdam The Netherlands, 141–150.
- [179] Bryant Vinisky. 2024. Improving Database Resilience with Observability and Chaos Testing. Retrieved April 28, 2024 from <https://newrelic.com/blog/how-to-relic/improving-database-resilience-with-observability-and-chaos-testing>
- [180] Bich Vu, J Darby Mitchell, Katherine Stowell, Mark Rabe, Orton Huang, Robert Lychev, and Martine Kalke. 2022. Mission resilience experimentation and evaluation testbed. In *MILCOM 2022-2022 IEEE Military Communications Conference (MILCOM)*. IEEE, United States, 173–179.
- [181] Jessica Wachtel. 2023. EBay Explores Chaos Fault Testing at the Application Level. Retrieved April 28, 2024 from <https://thenewstack.io/ebay-explores-chaos-fault-testing-at-the-application-level/>
- [182] John G Wacker. 2004. A theory of formal conceptual definitions: developing theory-building measurement instruments. *Journal of Operations Management* 22, 6 (2004), 629–650.
- [183] Guiping Wang, Shuyu Chen, and Jun Liu. 2015. An environment-aware anomaly detection framework of cloud platform for improving its dependability. In *Proceedings of the Int. Conference on Parallel Processing Techniques and App*. Committee of The Congress in Computer Science, Chongqing, CHINA, 431.
- [184] Sylwia Werbińska-Wojciechowska and Klaudia Winiarska. 2023. Maintenance performance in the age of Industry 4.0: A bibliometric performance analysis and a systematic literature review. *Sensors* 23, 3 (2023), 1409.
- [185] Shanika Wickramasinghe. 2023. Chaos Engineering: Benefits, Best Practices, and Challenges. Retrieved April 28, 2024 from https://www.splunk.com/en_us/blog/learn/chaos-engineering.html
- [186] Roel Wieringa, Neil Maiden, Nancy Mead, and Colette Rolland. 2006. Requirements engineering paper classification and evaluation criteria: a proposal and a discussion. *Requirements engineering* 11 (2006), 102–107.
- [187] Jill Willard and James Hutson. 2024. Fail Fast, Fail Small: Designing Resilient Systems for the Future of Software Engineering. *SSRG International Journal of Recent Engineering Science* 11 (2024), 51–58.
- [188] Benjamin Wilms. 2018. Chaos Engineering – withstanding turbulent conditions in production. Accessed: April 24, 2025. Available at: <https://www.codecentric.de/en/knowledge-hub/blog/chaos-engineering>.
- [189] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. 2012. *Experimentation in software engineering*. Vol. 236. Springer, Karlskrona, Sweden.
- [190] Zhaojun Wu. 2021. Securing Online Gaming: Combine Chaos Engineering with DevOps Practices. Retrieved April 28, 2024 from <https://www.pingcap.com/blog/securing-online-gaming-combine-chaos-engineering-with-devops-practices/>
- [191] Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Tianjun Jing, Xinmeng Sun, and Xiaoyun Li. 2021. Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments. In *Proceedings of the Web Conference 2021*. ACM, China, 3087–3098.
- [192] Jun Zhang, Robert Ferydouni, Aldrin Montana, Daniel Bittman, and Peter Alvaro. 2021. 3milebeach: A tracer with teeth. In *Proceedings of the ACM Symposium on Cloud Computing*. Scopus, Stockholm Sweden, 458–472.
- [193] Long Zhang, Brice Morin, Benoit Baudry, and Martin Monperrus. 2021. Maximizing error injection realism for chaos engineering with system calls. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2695–2708.
- [194] Long Zhang, Brice Morin, Philipp Haller, Benoit Baudry, and Martin Monperrus. 2019. A chaos engineering system for live analysis and falsification of exception-handling in the JVM. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2534–2548.
- [195] Long Zhang, Javier Ron, Benoit Baudry, and Martin Monperrus. 2023. Chaos engineering of ethereum blockchain clients. *Distributed Ledger Technologies: Research and Practice* 2, 3 (2023), 1–18.
- [196] Minqi Zhou, Rong Zhang, Dadan Zeng, and Weining Qian. 2010. Services in the cloud computing era: A survey. In *2010 4th international universal communication Symposium*. IEEE, Beijing, China, 40–46.