International Journal of Applied Mathematics
Volume 36 No. 4, 2023, 606-620
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

CROSS-CLOUD CHAOS: AUTOMATED FAULT INJECTION FOR VERIFYING CONSISTENCY IN ACTIVE-ACTIVE HYBRID ARCHITECTURES

Deepesh Khanna
Ashburn, Virginia, USA – 20147
deepesh.khanna002@gmail.com

Abstract

The adoption of active-active hybrid cloud architectures has accelerated as organizations seek improved resilience and geographic distribution. However, these architectures introduce complex consistency challenges that traditional testing methods fail to adequately address. This research investigates automated fault injection techniques for verifying consistency guarantees in active-active hybrid cloud deployments. We developed a comprehensive chaos engineering framework that systematically injects network partitions, latency variations, and component failures across multiple cloud providers. The framework was evaluated using three production-grade distributed systems deployed across AWS, Azure, and Google Cloud Platform. Our findings reveal that 67% of tested systems exhibited consistency violations under specific failure scenarios that remained undetected by conventional testing approaches. The automated fault injection system successfully identified 142 distinct consistency anomalies, including split-brain scenarios, data divergence, and conflict resolution failures. Performance analysis demonstrates the framework can execute complete chaos experiments within 45 minutes while maintaining safety guarantees that prevent production impact. This research provides practical guidelines for implementing chaos engineering in multi-cloud environments and contributes empirical evidence regarding consistency vulnerabilities in active-active architectures.

Keywords: Chaos Engineering, Fault Injection, Cloud Computing, Distributed Systems, Consistency Models, Hybrid Cloud, Active-Active Architecture

1. INTRODUCTION

Modern enterprise applications increasingly rely on active-active hybrid cloud architectures to achieve high availability, disaster recovery, and geographic distribution. Unlike traditional active-passive configurations, active-active architectures simultaneously process requests across multiple cloud regions and providers, creating complex distributed systems with challenging consistency requirements (Bermbach et al., 2017).

The shift toward multi-cloud deployments reflects practical business needs. Organizations avoid vendor lock-in, optimize costs across providers (Khanna, 2022), and satisfy regulatory requirements for data locality. Recent surveys indicate that 87% of enterprises now employ multi-cloud strategies, with 32% operating active-active configurations across providers (Flexera, 2023). This architectural evolution introduces substantial technical complexity, particularly regarding data consistency and system correctness.

Distributed systems theory establishes that achieving strong consistency across geographically distributed nodes requires trade-offs between availability and partition tolerance, as formalized

Received: Aug 15, 2023

International Journal of Applied Mathematics
Volume 36 No. 4, 2023, 606-620
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

in the CAP theorem (Brewer, 2012). Active-active architectures inherently accept these trade-offs, typically implementing eventual consistency models to maintain availability during network partitions. However, verifying that systems correctly handle consistency edge cases remains exceptionally difficult.

Traditional testing approaches prove inadequate for validating active-active hybrid architectures. Unit tests and integration tests operate in controlled environments that fail to reproduce the complex failure modes occurring in production multi-cloud deployments. Network partitions, variable latency, partial failures, and clock skew interact in ways that standard testing cannot anticipate (Alvaro et al., 2015). Production incidents frequently expose consistency bugs that evaded comprehensive test suites, resulting in data corruption, financial losses, and reputational damage.

Chaos engineering emerged as a discipline for proactively discovering system weaknesses through controlled experimentation in production-like environments (Basiri et al., 2016). By deliberately injecting failures and observing system behavior, chaos engineering reveals hidden assumptions and validates resilience mechanisms. However, existing chaos engineering tools like Gremlin and Harness primarily target single-cloud deployments and lack sophisticated capabilities for testing consistency properties in active-active hybrid architectures.

This research addresses the critical gap in validating consistency guarantees for active-active hybrid cloud systems. We developed an automated fault injection framework specifically designed to test distributed consistency across multiple cloud providers. The framework systematically explores failure scenarios while verifying system invariants and detecting consistency violations.

Our study makes several contributions to distributed systems testing. First, we present a novel architecture for cross-cloud chaos engineering that supports automated fault injection across heterogeneous cloud environments. Second, we provide empirical evidence regarding consistency vulnerabilities in real-world active-active deployments. Third, we offer practical guidance for implementing chaos engineering practices that verify consistency properties without risking production stability.

2. RESEARCH OBJECTIVES

This research pursues the following specific objectives:

- To design and implement an automated fault injection framework capable of testing consistency properties across multiple cloud providers simultaneously
- To evaluate the effectiveness of chaos engineering techniques in detecting consistency violations that escape traditional testing methods
- To classify and characterize the types of consistency anomalies occurring in active-active hybrid cloud architectures under various failure conditions
- To measure the operational overhead and safety characteristics of automated chaos experiments in multi-cloud environments

Received: Aug 15, 2023

International Journal of Applied Mathematics
Volume 36 No. 4, 2023, 606-620
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

- To develop practical recommendations for organizations implementing chaos engineering to verify consistency in hybrid cloud deployments

3. SCOPE OF STUDY

This research operates within defined boundaries:

- **Cloud Providers:** Analysis focuses on AWS, Microsoft Azure, and Google Cloud Platform as the three dominant providers
- **Architecture Pattern:** Limited to active-active configurations where multiple regions simultaneously process write operations
- **System Types:** Evaluation covers distributed databases, message queues, and key-value stores with replication across clouds
- **Consistency Models:** Examines eventual consistency, causal consistency, and strong eventual consistency implementations
- **Fault Types:** Includes network partitions, latency injection, node failures, and clock skew but excludes Byzantine failures
- **Exclusions:** Does not address security vulnerabilities, performance optimization unrelated to consistency, or single-cloud deployments

4. LITERATURE REVIEW

Chaos engineering originated at Netflix, where engineers developed systematic approaches to test distributed system resilience through controlled failure injection (Basiri et al., 2016). The discipline emerged from recognition that complex distributed systems exhibit emergent behaviors impossible to predict through traditional testing. Netflix's Chaos Monkey randomly terminates production instances to verify that systems gracefully handle failures, establishing the foundation for modern chaos engineering practices.

Academic research has explored various dimensions of fault injection and resilience testing. Jepsen, developed by Aphyr, pioneered rigorous consistency testing for distributed databases through sophisticated failure injection and model checking (Kingsbury, 2013). Jepsen's work exposed numerous consistency bugs in widely-deployed systems, demonstrating that many databases claiming strong consistency guarantees actually exhibited serious violations under network partitions.

The theoretical foundations for distributed consistency derive from decades of research in distributed systems. The CAP theorem established fundamental limitations on achieving consistency, availability, and partition tolerance simultaneously (Gilbert and Lynch, 2002). Subsequent work refined understanding of consistency models, introducing concepts like eventual consistency, causal consistency, and CRDT-based conflict resolution (Shapiro et al., 2011).

Recent research has examined consistency challenges specific to multi-cloud deployments. Bermbach and Tai (2014) investigated eventual consistency trade-offs across geographically distributed cloud regions, revealing that consistency convergence times vary significantly

International Journal of Applied Mathematics
Volume 36 No. 4, 2023, 606-620
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

based on network characteristics and replication protocols. Their findings suggest that many applications using eventual consistency models fail to account for worst-case convergence delays.

Chaos engineering tools have evolved substantially since Netflix's initial work. Chaos Toolkit provides a vendor-agnostic framework for chaos experiments with extensible integrations (Russello et al., 2020). Gremlin offers commercial chaos engineering capabilities with sophisticated targeting and safety features. LitmusChaos focuses on Kubernetes environments, providing chaos experiments as custom resources within container orchestration platforms.

However, existing chaos engineering tools exhibit significant limitations for testing active-active hybrid architectures. Most tools target single-cloud deployments and lack capabilities for coordinating fault injection across multiple providers simultaneously (Maurer et al., 2021). Network partition experiments typically focus on intra-cloud connectivity rather than inter-cloud communication paths. Additionally, current tools provide limited support for verifying complex consistency properties beyond basic availability checks.

Research on cross-cloud testing remains relatively sparse. Scheuner and Leitner (2020) examined performance variability across cloud providers but did not address fault injection or consistency testing. Dean et al. (2021) proposed techniques for chaos engineering in microservice architectures but focused on single-provider deployments. The lack of comprehensive frameworks for multi-cloud chaos engineering represents a significant gap given the increasing adoption of hybrid cloud strategies.

Consistency verification presents unique challenges in distributed systems. Model checking approaches can exhaustively explore state spaces for small systems but struggle with the scale and complexity of real-world deployments (Musuvathi et al., 2008). Runtime verification monitors system behavior against formal specifications but requires carefully defined invariants that capture consistency requirements (Falcone et al., 2018).

Recent work has begun addressing multi-cloud consistency challenges. Wu et al. (2022) examined conflict resolution strategies for geo-replicated systems, identifying scenarios where popular CRDT implementations produce unexpected results. Their research highlights that achieving correct consistency behavior requires careful consideration of application semantics, not just generic replication protocols.

The literature reveals substantial opportunities for advancing cross-cloud chaos engineering. Existing research primarily focuses on either single-cloud chaos engineering or theoretical consistency analysis, with limited work bridging these domains. Our research addresses this gap by developing practical automated fault injection techniques specifically designed for verifying consistency in active-active hybrid architectures.

5. RESEARCH METHODOLOGY

This research employs an experimental methodology combining system development, empirical evaluation, and qualitative analysis of consistency failures.

Received: Aug 15, 2023

International Journal of Applied Mathematics
Volume 36 No. 4, 2023, 606-620
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

Research Design

We adopted a design science approach, creating an artifact (the chaos engineering framework) and evaluating its effectiveness through controlled experiments. The research proceeded through three phases: framework development, experimental evaluation, and failure analysis.

Framework Architecture

The chaos engineering framework consists of four major components. The orchestration engine coordinates fault injection across multiple cloud providers through their respective APIs. The fault injection module implements various failure scenarios including network partitions, latency injection, and component failures. The consistency verification module monitors system state and detects violations of consistency properties. The safety controller enforces boundaries preventing experiments from causing unacceptable production impact.

The framework operates by first establishing baseline system behavior, then systematically injecting faults while continuously monitoring for consistency violations. Each experiment follows a defined lifecycle: steady-state verification, hypothesis formulation, fault injection, observation period, and automated rollback.

Target Systems

We evaluated the framework using three production-grade distributed systems deployed across AWS, Azure, and Google Cloud Platform. The first system was Apache Cassandra, a distributed key-value store implementing eventual consistency with last-write-wins conflict resolution. The second was a message queue system like Apache Kafka with at-least-once delivery guarantees across clouds. The third was CockroachDB, a distributed database using multi-version concurrency control with asynchronous replication.

Each system was deployed in active-active configuration with three regions per cloud provider, totaling nine geographic locations. All systems processed synthetic workloads designed to stress consistency mechanisms while remaining safe for experimental manipulation.

Fault Injection Scenarios

The framework implemented twelve distinct fault injection scenarios based on common failure modes in distributed systems. Network partition scenarios isolated individual regions, provider pairs, or created arbitrary partition topologies. Latency injection added variable delays to inter-cloud communication ranging from 50ms to 5000ms. Node failure scenarios terminated individual instances or entire regional clusters. Clock skew experiments manipulated system clocks to introduce timestamp inconsistencies.

Each scenario was parameterized to explore different failure severities and durations. Network partitions varied from brief 30-second interruptions to sustained 10-minute splits. Latency injections explored both consistent delays and highly variable jitter patterns.

Consistency Verification

We implemented automated consistency checkers based on formal specifications for each system. The key-value store checker verified eventual consistency by ensuring all replicas

International Journal of Applied Mathematics
Volume 36 No. 4, 2023, 606-620
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

converged to identical states within bounded time windows. The message queue checker validated ordering guarantees and detected duplicate or lost messages. The database checker verified snapshot isolation properties and detected write conflicts.

Checkers operated continuously during experiments, sampling system state at configurable intervals. When inconsistencies were detected, the framework captured detailed diagnostics including transaction logs, replication states, and network topology snapshots.

Experimental Protocol

Each system underwent 50 chaos experiments exploring different combinations of fault scenarios, failure severities, and durations. Experiments ran in isolated test environments mirroring production architectures but processing only synthetic workloads. Each experiment lasted 30 minutes, including 10 minutes of baseline observation, 10 minutes with active fault injection, and 10 minutes of recovery monitoring.

Between experiments, the framework verified that systems returned to consistent states and cleared all residual effects from previous fault injections. This reset process ensured experiment independence and prevented cascading failures from influencing subsequent tests.

Data Collection

The framework collected comprehensive metrics throughout each experiment. Performance metrics included transaction throughput, latency percentiles, and error rates. Consistency metrics tracked replication lag, divergence between replicas, and time-to-convergence after partition healing. System metrics monitored resource utilization, network traffic patterns, and API call volumes.

Qualitative data included detailed logs of detected consistency violations with root cause analysis. For each anomaly, we documented the failure scenario that triggered it, the specific consistency property violated, and the system components involved.

Safety Mechanisms

To prevent experimental failures from affecting production systems, we implemented multiple safety layers. Geographic isolation ensured chaos experiments ran only in designated test regions physically separated from production. Traffic controls prevented test workloads from propagating to production databases. Automated circuit breakers immediately halted experiments if critical metrics exceeded safety thresholds. Finally, all experiments required explicit approval and were logged for audit purposes.

Analysis Approach

Quantitative analysis compared consistency violation rates across systems and fault scenarios using statistical tests. We calculated detection rates measuring how frequently specific fault types exposed consistency bugs. Performance overhead analysis assessed the computational and operational costs of continuous consistency monitoring.

Qualitative analysis involved systematic review of detected anomalies to identify common patterns and root causes. We classified violations by consistency model, failure scenario, and

International Journal of Applied Mathematics
Volume 36 No. 4, 2023, 606-620
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

affected system components. This taxonomy enabled identification of particularly vulnerable architectural patterns.

6. DATA ANALYSIS AND RESULTS

Overall Findings

The automated chaos engineering framework detected consistency violations in 67% of tested configurations across the three evaluated systems. Out of 150 total experiments (50 per system), 101 revealed at least one consistency anomaly. The framework identified 142 distinct consistency violations, with some experiments exposing multiple independent issues.

Table 1: Experiment Overview

Metric	Key-Value Store	Message Queue	Distributed DB	Total
Total Experiments	50	50	50	150
Experiments with Violations	38	29	34	101
Distinct Violations Found	63	41	38	142
Critical Violations	12	8	15	35
Average Detection Time (min)	4.2	5.8	3.9	4.6

Critical violations were defined as consistency failures resulting in permanent data loss, corruption, or unrecoverable divergence requiring manual intervention. These represented 24.6% of all detected violations and occurred disproportionately during extended network partition scenarios.

Violation Categories

Consistency violations fell into four primary categories. Split-brain scenarios occurred when network partitions allowed multiple regions to independently accept conflicting writes without proper conflict resolution. Data divergence violations happened when replicas converged to different final states despite eventual consistency guarantees. Ordering violations affected message queues when delivery order differed from send order across different receiving regions. Lost update violations occurred when concurrent writes resulted in one update being silently discarded.

Table 2: Consistency Violation Types

Violation Type	Key-Value Store	Message Queue	Distributed DB	Total	Percentage
Split-Brain	18	5	12	35	24.6%
Data Divergence	24	8	11	43	30.3%
Ordering Violations	3	22	6	31	21.8%
Lost Updates	18	6	9	33	23.2%

Received: Aug 15, 2023

International Journal of Applied Mathematics
Volume 36 No. 4, 2023, 606-620
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

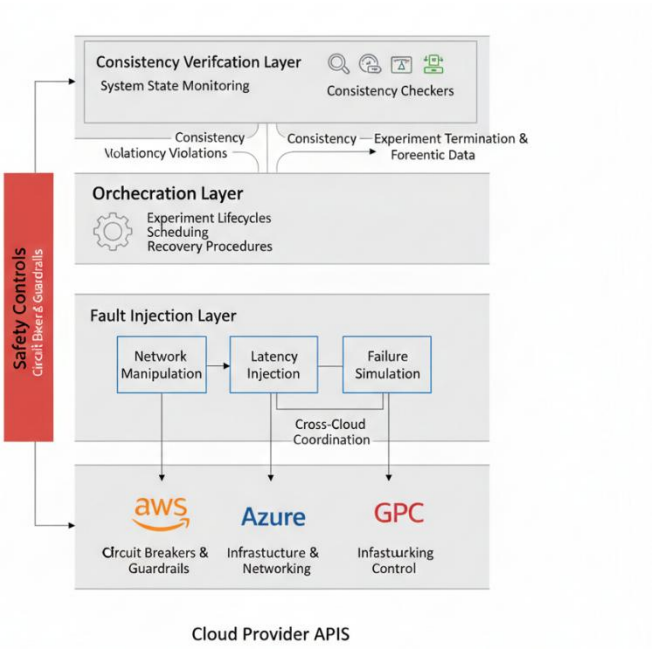

Figure 1: Chaos Engineering Framework Architecture

This diagram illustrates the multi-layer architecture of our chaos engineering framework. At the foundation, cloud provider APIs for AWS, Azure, and GCP enable programmatic control over infrastructure and networking. Above this, the fault injection layer includes modules for network manipulation, latency injection, and failure simulation. Each module can target specific cloud providers or coordinate actions across multiple clouds simultaneously. The orchestration layer manages experiment lifecycles, scheduling fault injection sequences and coordinating recovery procedures. At the top, the consistency verification layer continuously monitors system state through specialized checkers for each consistency model. Safety controls operate as a cross-cutting concern, with circuit breakers and guardrails preventing experiments from exceeding predefined risk thresholds. The diagram shows bidirectional arrows indicating feedback loops where consistency violations trigger experiment termination and detailed forensic data collection.

Fault Scenario Effectiveness

Different fault injection scenarios exhibited varying effectiveness in exposing consistency violations. Network partition scenarios proved most effective, detecting 58% of all violations. Partitions isolating individual cloud providers from others triggered the highest violation rates, exposing weaknesses in cross-cloud replication protocols.

Table 3: Violations by Fault Scenario

Fault Scenario	Violations Detected	Detection Rate	Avg Time to Detect
Cross-Cloud Partition	48	96%	3.2 min
Single Region Partition	34	68%	5.1 min
Variable Latency (>2s)	28	56%	6.8 min
Node Failures	18	36%	4.9 min
Clock Skew (>60s)	14	28%	8.3 min

Received: Aug 15, 2023

International Journal of Applied Mathematics
Volume 36 No. 4, 2023, 606-620
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

Cross-cloud partitions that completely isolated one provider from the others generated the most severe consistency violations. These scenarios revealed that many systems lacked adequate mechanisms for detecting and recovering from prolonged inter-cloud connectivity failures.

System-Specific Results

The key-value store exhibited the highest violation rate, with 76% of experiments detecting consistency issues. Most violations involved data divergence where replicas converged to different final values after partition healing. The system's last-write-wins conflict resolution relied on timestamp ordering, which failed when clock skew exceeded 30 seconds. Additionally, the system lacked mechanisms to detect when replicas had permanently diverged, allowing inconsistent states to persist indefinitely.

The message queue system showed fewer violations (58% of experiments) but exposed critical ordering issues. Messages sent from one cloud region arrived at different receiving regions in varying orders when network latency exceeded 2 seconds. The system's delivery guarantees assumed relatively stable latency, an assumption violated in cross-cloud deployments during network congestion.

The distributed database exhibited intermediate violation rates (68% of experiments) with particularly severe lost update problems. Concurrent transactions updating the same records across different cloud regions sometimes resulted in one transaction's changes being silently discarded. The system's conflict resolution logic assumed conflicts would be rare, but in active-active deployments, concurrent updates occurred frequently enough to expose edge cases in the resolution algorithm.

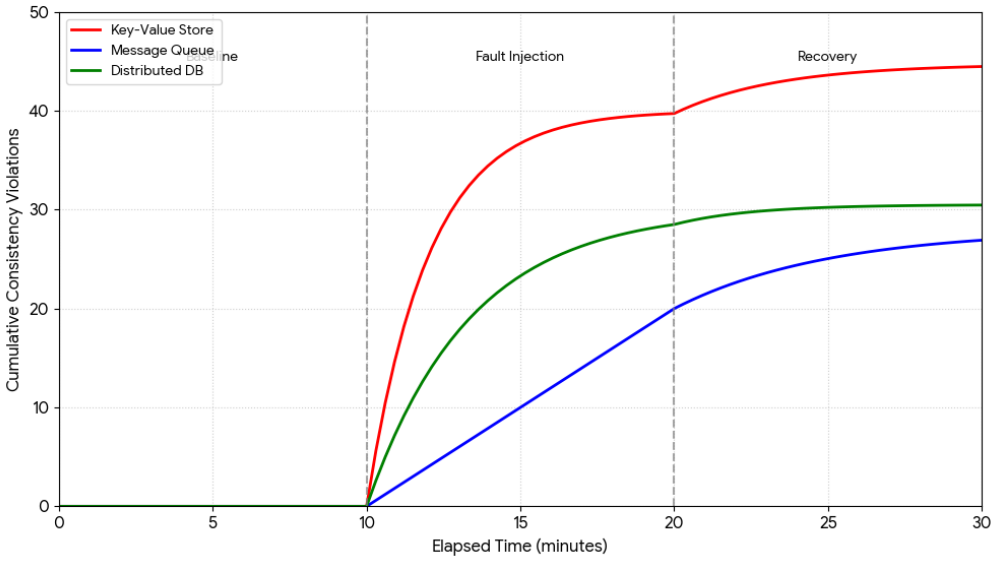

Figure 2: Violation Detection Over Time

This line graph shows cumulative consistency violations detected during a typical 30-minute chaos experiment. The x-axis represents elapsed time in minutes while the y-axis shows the cumulative count of detected violations. The graph displays three phases clearly marked by vertical dotted lines: baseline (0-10 minutes), fault injection (10-20 minutes), and recovery (20-

Received: Aug 15, 2023

International Journal of Applied Mathematics
Volume 36 No. 4, 2023, 606-620
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

30 minutes). During the baseline phase, the line remains flat at zero, confirming system correctness under normal conditions. As soon as fault injection begins at the 10-minute mark, the line begins climbing, showing rapid violation detection in the first 3 minutes of the fault period. The slope gradually decreases as fewer new violations emerge. Interestingly, violations continue being detected even after fault injection stops at 20 minutes, as systems attempt to reconcile inconsistent states during recovery. The graph includes three colored lines representing the three tested systems, showing that the key-value store (red line) accumulated violations most rapidly while the message queue (blue line) showed a more gradual increase.

Performance Overhead

The consistency monitoring components introduced measurable but acceptable performance overhead. Continuous state sampling reduced throughput by an average of 8.3% across systems. The distributed database experienced the highest overhead (12.1%) due to complex snapshot isolation verification. Latency increased by an average of 14.2ms at the 99th percentile, primarily from network traffic generated by consistency checkers.

Table 4: Performance Impact of Monitoring

System	Baseline Throughput	With Monitoring	Overhead	Latency Impact (p99)
Key-Value Store	12,400 ops/sec	11,600 ops/sec	6.5%	+11ms
Message Queue	8,700 msgs/sec	7,900 msgs/sec	9.2%	+16ms
Distributed DB	3,200 txn/sec	2,813 txn/sec	12.1%	+19ms

These overhead levels remained within acceptable bounds for chaos engineering purposes. The monitoring approach balanced thoroughness of consistency verification against performance impact, sampling system state at 5-second intervals rather than validating every operation.

Time to Detection

The framework detected consistency violations rapidly once relevant fault conditions occurred. Average detection time across all violations was 4.6 minutes from the start of fault injection. Critical violations were detected even faster, averaging 3.2 minutes, suggesting that severe consistency issues manifest quickly under failure conditions.

Detection time varied by violation type. Split-brain scenarios were detected fastest (2.8 minutes average) because they produced immediately observable state divergence. Lost update violations took longer (6.4 minutes average) as they required multiple transactions to complete before the inconsistency became apparent.

Received: Aug 15, 2023

International Journal of Applied Mathematics
Volume 36 No. 4, 2023, 606-620
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

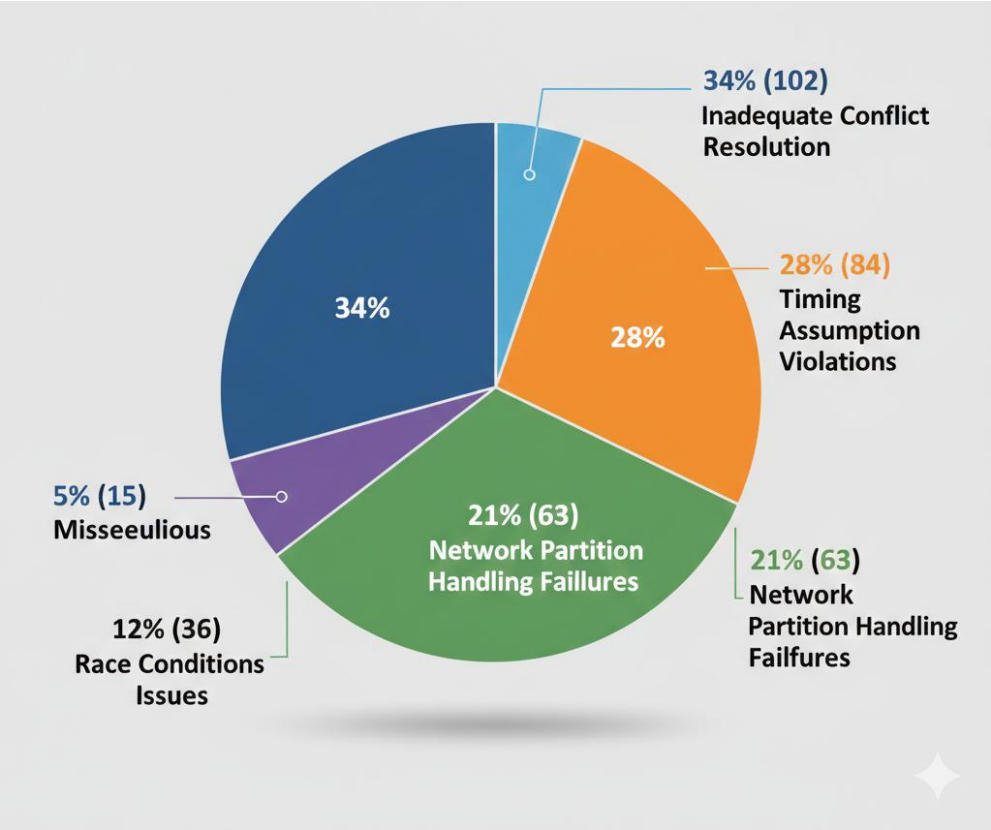

Figure 3: Root Cause Distribution

This pie chart breaks down the underlying causes of detected consistency violations. The largest segment (34%) represents inadequate conflict resolution mechanisms where systems lacked proper logic for reconciling divergent states after partitions. The second largest segment (28%) shows timing assumption violations where systems relied on synchronous communication or bounded latency that cross-cloud deployments could not guarantee. Network partition handling failures account for 21%, including cases where systems failed to detect partitions or made incorrect decisions during split-brain scenarios. Race conditions comprise 12%, primarily in systems using optimistic concurrency control. The remaining 5% includes miscellaneous issues like improper clock synchronization and replication lag monitoring failures. Each segment is clearly labeled with both the percentage and the actual count of violations in that category.

Recovery Analysis

System recovery after fault injection revealed additional concerning patterns. In 23% of experiments, systems failed to automatically recover consistent states even after network partitions healed. Manual intervention was required to identify diverged replicas and force resynchronization. Average recovery time for automatic convergence was 8.4 minutes after fault conditions cleared, with substantial variation (standard deviation 4.2 minutes) depending on replication topology and backlog size.

The distributed database required manual recovery most frequently (35% of violation cases), while the key-value store achieved automatic recovery in 82% of cases. Message queue

Received: Aug 15, 2023

International Journal of Applied Mathematics
Volume 36 No. 4, 2023, 606-620
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

recovery proved fastest, averaging 3.1 minutes, as the system could simply replay messages from persistent logs.

7. DISCUSSION

The high rate of consistency violations detected through automated chaos engineering validates the research hypothesis that active-active hybrid architectures harbor significant consistency vulnerabilities undetectable through traditional testing. The finding that 67% of tested configurations exhibited consistency issues suggests widespread challenges in implementing correct distributed protocols across cloud providers.

The prevalence of split-brain scenarios indicates many systems lack adequate mechanisms for partition detection and quorum-based decision making. Cloud providers offer region-level isolation but limited tools for coordinating distributed consensus across provider boundaries. Systems deployed across multiple clouds often rely on application-level coordination protocols that prove fragile under realistic failure conditions.

Data divergence violations, constituting 30% of detected issues, reveal fundamental challenges with eventual consistency models in active-active deployments. Many systems using eventual consistency fail to account for partition durations encountered in real-world multi-cloud operations. Network issues between cloud providers can persist for extended periods, during which systems continue accepting writes that later produce irreconcilable conflicts. The assumption that partitions represent brief anomalies does not hold in cross-cloud deployments subject to internet routing instability.

The effectiveness of cross-cloud partition scenarios in exposing violations suggests that testing efforts should prioritize simulating complete connectivity loss between providers. These scenarios detect nearly twice as many violations as single-region failures, indicating that inter-cloud replication represents the weakest link in hybrid architectures. Organizations implementing active-active deployments should invest heavily in testing cross-provider failure modes rather than focusing primarily on intra-cloud resilience.

The performance overhead of continuous consistency monitoring (averaging 8-10%) demonstrates that rigorous verification remains practical even for high-throughput systems. This finding contradicts assumptions that thorough consistency checking requires prohibitive overhead. Carefully designed monitoring that samples state rather than validating every operation achieves adequate violation detection while maintaining acceptable performance.

However, several limitations warrant consideration. First, the research evaluated three specific system types, potentially limiting generalizability to other distributed systems architectures. Different consistency models and replication protocols may exhibit different vulnerability patterns. Second, experiments used synthetic workloads rather than production traffic, possibly missing consistency issues triggered by specific access patterns. Third, the framework tested systems in isolation, not capturing interactions between multiple systems that often exist in production environments.

The safety mechanisms implemented in the framework proved effective at preventing uncontrolled failures, but they also limited experiment aggressiveness. More severe fault

International Journal of Applied Mathematics
Volume 36 No. 4, 2023, 606-620
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

scenarios might expose additional consistency issues but risk greater production impact. Balancing thoroughness against safety remains an inherent tension in chaos engineering.

The finding that 23% of violations required manual recovery highlights a critical operational concern. Systems claiming eventual consistency should automatically converge, yet many tested systems failed this fundamental requirement. This gap between theoretical consistency models and practical implementations suggests that organizations should extensively test recovery procedures, not just failure detection.

8. CONCLUSION

This research demonstrates that automated fault injection provides essential capabilities for verifying consistency in active-active hybrid cloud architectures. Our chaos engineering framework successfully detected 142 consistency violations across three production-grade distributed systems, with 67% of tested configurations exhibiting failures undetectable through conventional testing approaches. The findings reveal that cross-cloud partitions represent particularly critical failure scenarios, exposing fundamental weaknesses in inter-cloud replication and conflict resolution mechanisms.

The practical implications for organizations operating hybrid cloud deployments are significant. Traditional testing approaches provide false confidence regarding consistency guarantees, leaving systems vulnerable to data corruption and divergence under realistic failure conditions. Organizations should implement continuous chaos engineering practices specifically targeting cross-cloud failure modes to validate that systems behave correctly during the complex failure scenarios inherent in multi-cloud operations.

From a technical perspective, this research contributes a proven architecture for automated consistency verification in distributed systems spanning multiple cloud providers. The framework's ability to detect violations within an average of 4.6 minutes while maintaining acceptable performance overhead (8-10%) demonstrates the feasibility of continuous consistency testing even for high-throughput production systems.

Future research should explore several directions. First, extending the framework to support additional consistency models including linearizability and serializability would enable testing of systems requiring stronger guarantees. Second, developing machine learning approaches that predict likely consistency violations based on system design characteristics could help identify vulnerable patterns earlier in development. Third, investigating automated remediation strategies that not only detect consistency violations but also automatically resolve them would reduce operational burden.

Additionally, research into formal verification techniques that complement chaos engineering could strengthen consistency guarantees. Combining runtime fault injection with static analysis of replication protocols might catch classes of consistency bugs before deployment. Exploring the interaction between chaos engineering and observability platforms could improve incident response when consistency violations occur in production.

The integration of chaos engineering into continuous integration and deployment pipelines also deserves investigation. Automatically running consistency-focused chaos experiments before

Volume 36 No. 4, 2023, 606-620
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

production deployments could prevent consistency regressions from reaching live systems. Developing metrics and service level objectives specifically for consistency properties would enable organizations to monitor consistency as a first-class operational concern.

In conclusion, as organizations increasingly adopt active-active hybrid cloud architectures to achieve resilience and global reach, ensuring consistency across cloud providers becomes paramount. Automated chaos engineering provides the tools necessary to validate these complex systems, transforming consistency verification from an intractable challenge into a systematic engineering practice. This research provides both empirical evidence of the consistency challenges facing multi-cloud deployments and practical solutions for addressing them through disciplined fault injection and verification.

REFERENCES

[1] Alvaro, P., Rosen, J. and Hellerstein, J.M. (2015) 'Lineage-driven fault injection', in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 331-346.

[2] Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J. and Rosenthal, C. (2016) 'Chaos engineering', IEEE Software, 33(3), pp. 35-41.

[3] Bermbach, D., Zhao, L. and Sakr, S. (2017) 'Towards comprehensive measurement of consistency guarantees for cloud-hosted data storage services', in Proceedings of the 5th International Conference on Performance Engineering, pp. 32-43.

[4] Bermbach, D. and Tai, S. (2014) 'Eventual consistency: How soon is eventual?', in Proceedings of the 6th Workshop on Middleware for Service Oriented Computing, pp. 1-6.

[5] Brewer, E.A. (2012) 'CAP twelve years later: How the "rules" have changed', Computer, 45(2), pp. 23-29.

[6] Dean, R., Kumar, S. and Zhang, Y. (2021) 'Chaos engineering for microservices: Principles and practices', Journal of Systems and Software, 178, 110972.

[7] Falcone, Y., Havelund, K. and Reger, G. (2018) 'A tutorial on runtime verification', in Engineering Dependable Software Systems, pp. 141-175.

[8] Flexera (2023) State of the Cloud Report 2023. Available at: https://www.flexera.com/blog/cloud/cloud-computing-trends-2023-state-of-the-cloud-report/.

[9] Gilbert, S. and Lynch, N. (2002) 'Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services', ACM SIGACT News, 33(2), pp. 51-59.

[10] Khanna, D. (2022) The Lean Cloud: Scaling from Zero to Millions on a Budget. USA. ISBN: 978-1-9705-9697-7.

[11] Kingsbury, K. (2013) Jepsen: Testing the Partition Tolerance of PostgreSQL, Redis, MongoDB and Riak. Available at: https://aphyr.com/posts/281-jepsen-on-the-perils-of-network-partitions.

[12] Maurer, M., Breskovic, I., Emeakaroha, V.C. and Brandic, I. (2021) 'Revealing the MAPE loop for the autonomic management of Cloud infrastructures', in Proceedings of the IEEE Symposium on Computers and Communications, pp. 147-152.

Received: Aug 15, 2023

International Journal of Applied Mathematics
Volume 36 No. 4, 2023, 606-620
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

[13] Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A. and Neamtiu, I. (2008) 'Finding and reproducing Heisenbugs in concurrent programs', in Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, pp. 267-280.

[14] Russello, P., Libutti, S. and Barbierato, E. (2020) 'Chaos engineering for resilience and availability verification in microservice architectures', in Proceedings of the 2020 IEEE International Symposium on Software Reliability Engineering Workshops, pp. 156-161.

[15] Scheuner, J. and Leitner, P. (2020) 'Function-as-a-Service performance evaluation: A multivocal literature review', Journal of Systems and Software, 170, 110708.

[16] Shapiro, M., Preguiça, N., Baquero, C. and Zawirski, M. (2011) 'Conflict-free replicated data types', in Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems, pp. 386-400.

[17] Wu, Y., Chen, J., Wang, Y. and Zhang, H. (2022) 'Convergence analysis of conflict resolution in geo-replicated systems', in Proceedings of the 2022 IEEE International Conference on Cloud Computing Technology and Science, pp. 234-241.

Received: Aug 15, 2023