International Journal on Advanced Science Engineering Information Technology

Vol.14 (2024) No. 6
ISSN: 2088-5334

Enhanced Chaos-Driven Automation: A Unique Resilience Testing Toolkit for Cloud-Native IoT Networks

Yu Weiyuan [a], Mohd Hafeez Osman [a,*], Rodziah Atan [a], Wan Nurhayati Wan Ab [a]. Rahman [a]

[a] *Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, Serdang, Selangor, Malaysia*
Corresponding author: [*]hafeez@upm.edu.my

Abstract— Conventional approaches, such as static load testing and synthetic monitoring, typically evaluate system performance under controlled conditions but do not fully capture the unpredictable scenarios encountered in real-world operations. For instance, static load testing involves applying a predetermined load to the system to measure performance metrics like response time and throughput, which may not reflect the variability and chaos of actual usage. Similarly, synthetic monitoring uses scripted transactions to check system availability and performance, but these scripts often lack the complexity and variability of real-world interactions. This research aims to overcome these limitations by utilizing advanced chaos engineering techniques to simulate a range of faults, including network latency, service crashes, resource exhaustion, message loss, and security attacks. The proposed tool integrates components for data generation, fault injection, storage, monitoring, and visualization, allowing for a thorough evaluation of system robustness. The methodology involves conducting controlled experiments within an AWS-based cloud-native IoT environment to assess the tool's effectiveness. These experiments demonstrate that the tool effectively identifies weaknesses in system resilience and improves overall robustness. By replicating real-world disruptions and analyzing system responses, the tool provides critical insights into the behavior of IoT devices under stress. The study concludes that this chaos engineering tool significantly enhances the ability to detect and address vulnerabilities, supporting creating more resilient IoT systems. Future work will expand the range of simulated faults, validate the tool across various cloud platforms, and incorporate additional real-time analysis features.

***Keywords*—Chaos engineering; IoT resilience; fault injection; cloud-native environments; Amazon Web Services (AWS).**

Manuscript received 16 Jan. 2024; revised 10 Sep. 2024; accepted 28 Oct. 2024. Date of publication 31 Dec. 2024.

I. INTRODUCTION

In recent years, the resilience of cloud-native IoT environments has gained increasing significance due to the complex integration of IoT devices across diverse applications. Critical infrastructure sectors such as smart grids, healthcare systems, and industrial automation increasingly rely on IoT. For instance, smart grids use IoT sensors for real-time monitoring and control of electrical distribution, while healthcare systems employ IoT devices for patient monitoring and data collection. Traditional testing methodologies often fall short when it comes to continuously operating, interconnected systems that require high reliability. Chaos engineering emerges as a powerful approach to enhance system resilience by intentionally injecting faults to test systems' ability to withstand unexpected conditions [1]-[6].

This study introduces a chaos engineering tool explicitly designed for IoT systems in cloud-native environments, employing advanced fault injection techniques to simulate real-world disruptions and assess system robustness. Conventional methods for testing IoT resilience often involve static tests that do not fully represent the dynamic nature of real-world operations. For example, standard testing may fail to account for sudden network outages or security breaches that can significantly impact system performance. The proposed tool addresses these limitations by integrating comprehensive fault injection capabilities, including network latency, service crashes, and security attacks. For instance, by simulating a Distributed Denial of Service (DDoS) attack, the tool can evaluate how well an IoT system withstands such a high-volume attack and recovers from it.

IoT systems' growing complexity and integration into critical infrastructure demand robust testing methods that can predict and prevent failures before they occur [7]-[9]. Chaos Engineering provides a proactive testing framework, contrasting with traditional reactive methods. By adopting chaos engineering principles, our tool aims to offer a detailed analysis of fault tolerance and resilience, paving the way for creating more reliable IoT systems [10]-[12]. For example, the tool's ability to simulate network latency allows it to test

how IoT devices manage delayed communications, which is crucial for maintaining functionality in smart grid systems where timely data is essential.

This paper proposes a structured methodology using a chaos-driven testing tool to evaluate IoT devices' resilience in cloud-native setups systematically. The tool leverages both simulated and real data to create a variety of test scenarios that mirror potential operational disruptions. This methodical injection of faults and subsequent monitoring allows for precisely assessing system capacities and recovery mechanisms.

The primary contributions of this study are threefold:

a. Innovative Fault Injection Techniques: The tool introduces advanced fault simulation methods tailored for cloud-native IoT environments, such as simulating network latency and service crashes to test system resilience under real-world conditions [13].

b. Comprehensive System Evaluation: Through continuous monitoring and data analysis, the tool provides insights into how IoT systems respond to and recover from disruptions like service failures and security attacks, thereby facilitating a deeper understanding of system vulnerabilities [14].

c. Enhanced Resilience Strategies: By identifying critical weaknesses and testing various fault scenarios, such as network partitions and security breaches, the tool helps develop effective mitigation strategies, significantly improving system robustness against future disruptions [15].

II. Materials and Method

A. Proposed Chaos Engineering Tool

The proposed chaos engineering tool for IoT systems in cloud-native environments leverages fault injection techniques to evaluate system resilience. The tool integrates various fault scenarios, such as network latency, service crashes, and security attacks, to simulate real-world disruptions and assess the robustness of IoT devices [16][17].

Fig. 1 illustrates the tool's architecture, showing its key components: fault injectors, monitoring modules, and resilience reporting. The fault injectors introduce disruptions into the system, the monitoring modules track the system's performance during these disruptions, and the resilience reporting component analyzes the data to provide insights and suggestions for improving system robustness. These components interact seamlessly within the cloud-native environment, ensuring comprehensive resilience testing.

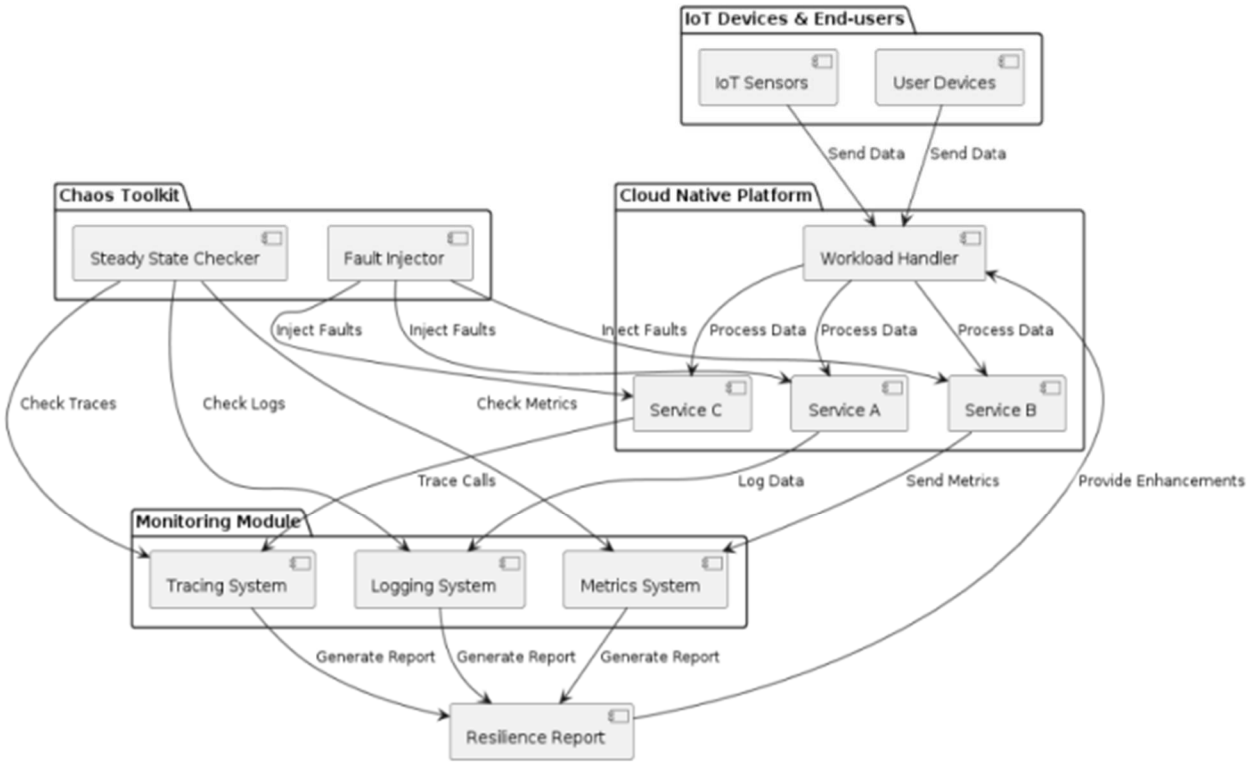

Fig. 1 The Architecture of the Proposed Chaos Engineering Tool

B. Fault Injection Techniques

In this study, we designed multiple fault injection techniques to test IoT systems' resilience comprehensively. These techniques include network latency injection, service crash simulation, and security attack emulation [18]. Their primary task is to create realistic fault scenarios that can challenge the system and reveal potential vulnerabilities [19], [20]. Fig. 2 illustrates the fault injection process, demonstrating how different fault scenarios are introduced and monitored within the system.

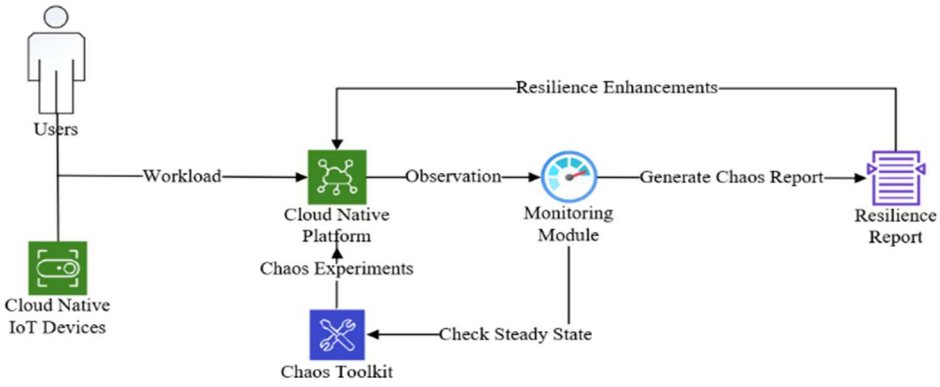

Fig. 2 Fault Injection Process

1) Network Latency Injection: This technique simulates delays in network communication to test the system's ability to handle slow or disrupted connectivity. By varying latency levels, we can assess the impact on data transmission and system performance [12], [16].

2) Service Crash Simulation: This method involves intentionally crashing specific services within the IoT system to evaluate the system's ability to detect, isolate, and recover from service failures. This helps identify critical dependencies and improve fault tolerance [19], [20].

3) Security Attack Emulation: This technique simulates different types of security attacks, such as Man-in-the-Middle (MitM) and Denial of Service (DoS), to assess the system's security measures and resilience against malicious activities.

C. Conceptual Model Design

This section focuses on designing the workflow for chaos engineering tools tailored for MQTT (Message Queuing Telemetry Transport)-based cloud-native IoT systems. The following diagram (Fig. 3) illustrates this system's primary workflow, including preparation, experiment design, and result analysis.

1) Connect IoT Devices: Connect the required devices to the cloud-native platform, such as AWS, using the appropriate communication protocol (focus on MQTT). For devices still in development, load the workload onto the cloud-native platform using manual simulation.

2) Ensure Observability: Integrate monitoring and logging components to ensure each IoT device service can be monitored. This allows the system to track the impact of chaos engineering fault injections in real time.

3) Define Steady State: According to chaos engineering principles, define the system's steady state, which should be inferred through measurable indicators such as workload and observability metrics.

4) Design Chaos Experiments: Design chaos experiments based on the fault injection models proposed earlier. Inject faults according to the fault types and compare the results with the steady state defined in step three.

5) Analyze Results and Generate Reports: Analyze the results and generate reports to provide recommendations for resilience enhancement.

D. Model Development

Fig. 1 illustrates the various system components and their interactions, providing a comprehensive overview of how different modules collaborate to enhance the resilience of the cloud-native IoT environment [21], [22], [23], [24], [25], [26], [27], [28], [29], [30].

1) IoT Devices & End-users: Generate workloads and send them to the cloud-native platform.

2) Cloud Native Platform: Includes workload processors and multiple services (Service A, B, C).

3) Chaos Toolkit: Contains fault injectors and steady-state checkers for injecting faults and checking system steady-state.

4) Monitoring Module: This includes a log system, indicator system, and tracing system, as well as Prometheus and Grafana for real-time analysis and monitoring.

5) Resilience Report: Generates reports and suggestions for improving system resilience.

The interaction flow design section (Fig. 3) outlines the system's interaction flow, illustrating how users interact with the system and how data moves through different components. This flowchart clarifies the roles of each element and the sequence of operations that occur during typical usage scenarios.

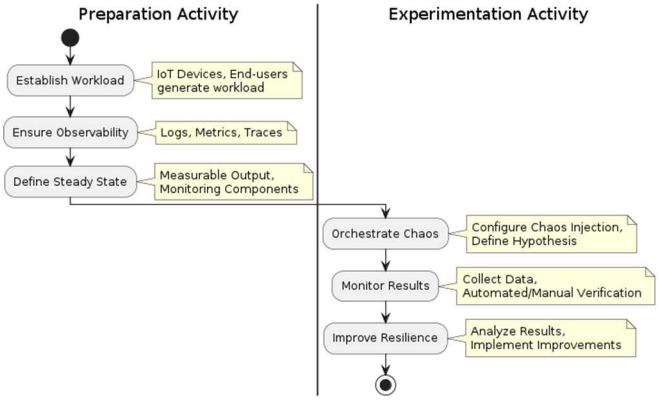

Fig. 3 Interaction Flow Diagram

The sequence diagram (Fig. 4) provides a step-by-step illustration of the proposed chaos engineering model's activities, detailing the components' interactions during a

resilience test. Users generate workloads and send data to the cloud platform through IoT devices and end users.

1) Workload Generation: Users generate workloads through IoT devices and end-users, which send data to the cloud platform for processing.

2) Data Processing: The workload processor receives and processes the data, distributing it to each service for further processing according to their specialized functions.

3) Monitoring: Each service sends logs, performance indicators, and call link data to the corresponding systems within the monitoring module. This ensures that all activities are tracked and can be analyzed for performance and issues.

4) Fault Injection: The Chaos Toolkit injects faults into the system, simulating different failure scenarios. Steady-state checkers within the toolkit monitor the system's response, ensuring it continues operating effectively despite disruptions.

5) Data Collection & Analysis: Prometheus collects data from the monitoring systems, including logs, indicators, and traces. This data is then provided to Grafana for real-time analysis and visualization.

6) Resilience Reporting: Based on the collected data, Grafana generates comprehensive resilience reports. These reports include detailed analyses of how the system responded to the injected faults and provide actionable suggestions for improving system resilience.

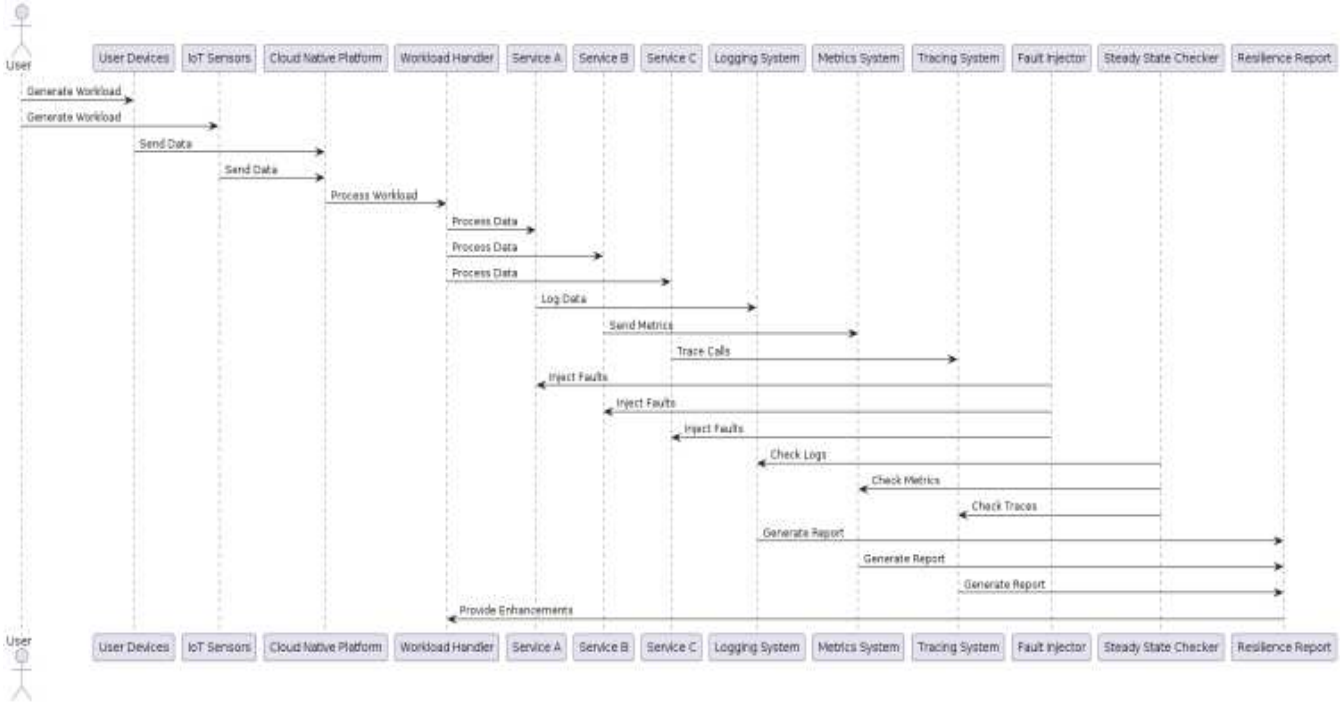

Fig. 4 Sequence Diagram

E. Validation of Model

After the model was developed, several experts in cloud computing, IoT, and chaos engineering reviewed the model [21], [22], [23]. The main steps include:

1) Model Introduction: Introduce the chaos engineering model in detail to the experts.

2) Expert Feedback: Collect feedback and suggestions on the model's rationality, practicality, and innovation.

3) Problem Discussion: Discuss potential problems and deficiencies in the model with experts and seek improvement solutions.

4) Review Summary: Summarize the results of the expert review and provide a basis for model improvement.

The experts involved in reviewing the model are the following:

1) Expert A: Scholar in chaos engineering, Tsinghua University.

2) Expert B: Expert in IoT and cloud computing, Peking University.

3) Expert C: Senior engineer at Alibaba Cloud.

4) Expert D: Senior researcher at Huawei.

The feedback from the experts is the following:

1) Expert A: Suggested further refinement of the fault injection methods.

2) Expert B: Recommended considering more user behaviors and usage scenarios when defining the system's steady state.

3) Expert C: Suggested detailed simulation schemes for cloud service provider failures.

4) Expert D: Recommended incorporating more real-time analysis tools into the monitoring module.

From the expert feedback, improvements were made to the model (illustrated in Fig 5):

1) Enhanced Fault Injection: Tailored fault injection methods were refined for specific failure scenarios like network partitioning and CPU exhaustion, broadening the model's ability to simulate diverse conditions.

2) Expanded Steady State Definition: The system's steady state was expanded to include various user behaviors and peak usage scenarios, offering a more accurate baseline for resilience assessment.

3) Detailed Cloud Failure Simulations: New simulation schemes for cloud service provider failures, such as database outages, were incorporated to test system recovery under critical conditions.

4) Added Real-Time Analysis: The monitoring module now includes additional real-time tools like distributed tracing and anomaly detection, improving the system's ability to quickly identify and respond to issues.

Integrating these enhancements posed challenges, such as compatibility issues with new fault injection methods and complexities monitoring expanded steady-state definitions. These were resolved through infrastructure updates and refined data processing capabilities. The experiment aims to validate the effectiveness of the designed tool through simulated workloads. It covers three types of fault injection: external faults, internal faults, and cloud service provider faults. The experiments will be divided into three groups to verify the tool's applicability and effectiveness.

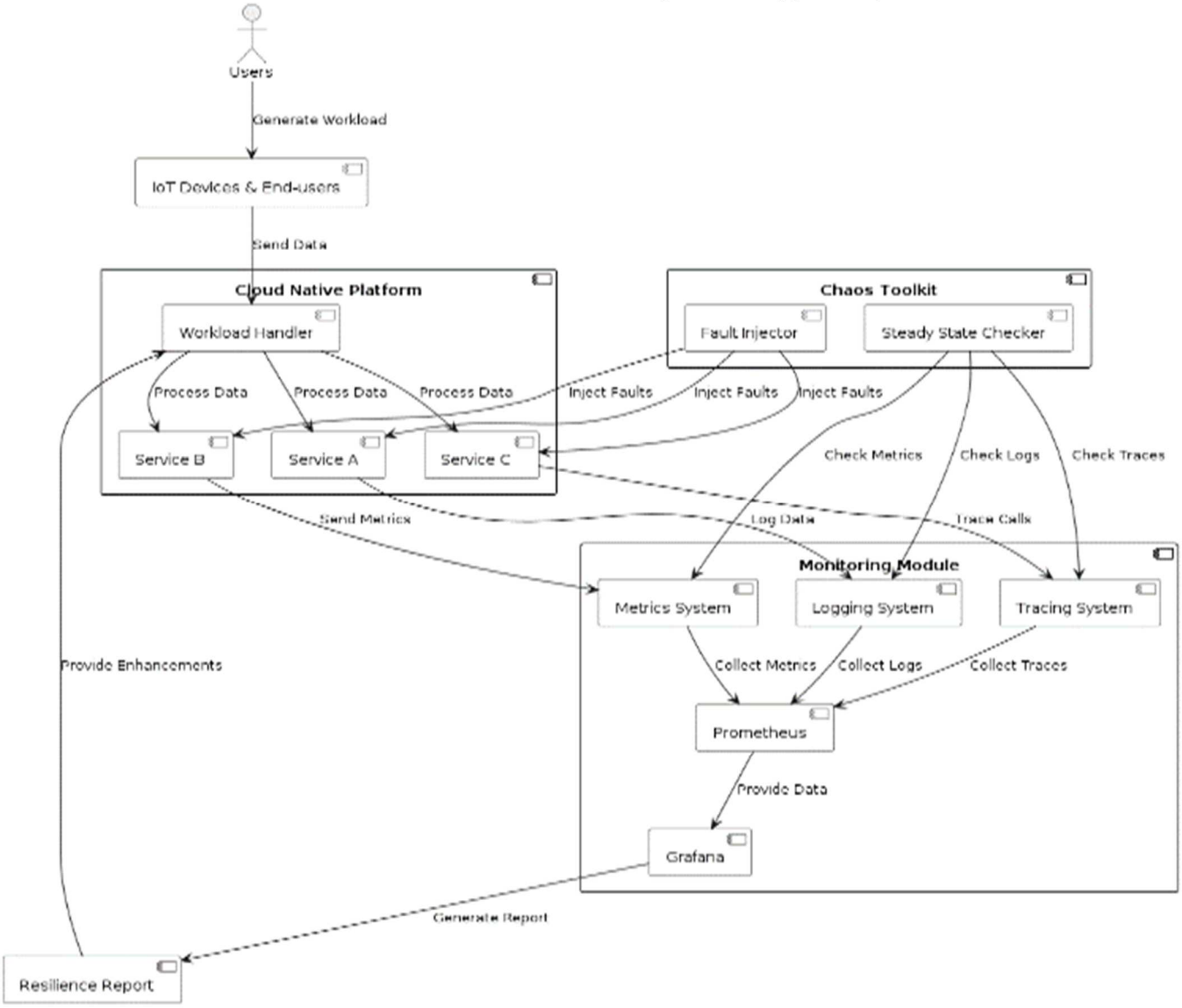

Fig. 5 Improvement of the proposed model after expert review

F. External Service Fault Injection

This experiment tests the tool's ability to inject faults from external services. User and gateway services handle user requests through an API gateway and communicate with an external database service. The primary goal is to demonstrate the tool's impact on the system's steady state by injecting DNS failures and blocking traffic to the external database service. The simulation of user requests is the following:

a. Configured to send requests at a rate of two user threads per second.
b. Maximum request throughput set to two requests per second.
c. Each user thread executes 300 requests each time, for a total of 600 requests.

G. Internal Service Fault Injection

These experiments simulate issues within an internally managed Redis deployment. Redis operates on a master-slave architecture, with master instances handling read-write operations and slave replicas handling read-only operations. The deployment uses Redis Sentinel for high availability by enabling automatic failovers and master reelection.

a. Terminate different combinations of master/slave pods to observe the failover process initiated by Redis Sentinel.
b. Stress the memory of all master/slave pods to test the system's resilience against memory exhaustion issues.

H. Cloud Provider Service Fault Injection

These experiments simulate issues related to the user-side MQTT proxy. The first part blocks all network traffic between the stream-service microservice and the user-side MQTT proxy. The second part introduces communication delays with the user-side MQTT proxy. The goal is to verify the system's stability in case the MQTT proxy is unavailable.

Observable Metrics include Success Rate, Error Rate, Average Response Time, CPU Usage, Memory Usage, Disk Usage, Event Messages Sent, and Event Messages Received. The tool generates workloads through test cases designed to test the event pipeline, running automatically every ten minutes through croon scheduling. This setup ensures stable workloads for evaluating system performance under different fault injection conditions.

III. Result and Discussion

This section explains the experimental results of External Fault Injection, Internal Fault Injection, and Cloud Provider Fault Injection.

A. Result of External Fault Injection

For the external fault injection experiment, we propose two main hypotheses: success rate and response time. Success rate hypothesis: during DNS request error injection, the success rate will drop significantly by more than five percentage points. Response time hypothesis: during DNS request error injection, the response time of successful requests will dramatically change, with a p-value below 0.000001.

1) Experimental Group 1: DNS Injection: We used chaos engineering tools to inject DNS errors and collected the following data for the control and experimental groups after DNS injection. The results showed a significant drop-in success rate from 97.6% to 87.2% and an increase in average response time from 788 milliseconds to 1325 milliseconds. The t-test results support these observations, indicating that DNS chaos injection significantly affects the system's steady state (Table 1).

TABLE I
TEST FOR DNS INJECTION EXPERIMENT

Metric	T-test Statistic	p-value
Success Rate	19.8	2.20E-35
Average Response Time	-25.6	8.50E-48

2) Experimental Group 2—Network Partition: In the network partition experiment, the system's success rate dropped from 97.6% to 85.3%, and the average response time increased from 788 milliseconds to 1400 milliseconds. The t-test results confirm that network partition failures significantly impact the system's steady state (Fig. 6).

Metric	T-test Statistic	p-value
Success Rate	22.4	1.10E-39
Average Response Time	-28.3	2.70E-52

Fig. 6: Test for Network Partition Experiment

B. Result of Internal Fault Injection

This section presents the design and results of an experiment conducted using our chaos engineering tool to test the resilience of Redis deployments, focusing on a master pod crash. The experiment was run ten times to thoroughly assess its impact on the system. The hypothesis was that if the master pod fails, Redis Sentinel should promote a slave pod to the master, and the system should continue to operate with minimal disruption.

The chaos engineering tool terminated the master pod for 10 seconds. The results were compared with a control group running without induced failures (Table 2). In the experimental group, the success rate dropped to 75%, the error rate increased to 25%, and the average response time increased to 450 milliseconds, indicating a significant impact on system performance due to the master node failure Fig. 7.

TABLE II
COMPARISON OF RESULTS FOR CONTROL GROUP AND EXPERIMENTAL GROUP

Metric	Control Group	Experimental Group (Master Node Failure)
Success Rate (%)	100%	75%
Error Rate (%)	0%	25%
Average Response Time (ms)	200 ms	450 ms
CPU Usage (%)	50%	70%
Memory Usage (%)	60%	80%
Disk Usage (%)	40%	60%
Redis Commands Executed (commands/sec)	1000	600
Redis Cache Hits	950	700
Redis Cache Misses	50	300

C. Result of Cloud Provider Fault Injection

This section presents the results of two key experiments designed to evaluate the impact of network faults on an MQTT-based cloud-native IoT system using Mosquitto as the MQTT broker with different QoS levels.

1) Experiment Group 1- Network Partition: This experiment blocked traffic to the user-facing MQTT broker for 1 minute and 30 minutes. The hypothesis was falsified, as the connection was reestablished after the partition, but events during the disruption were lost. The results showed a drop-in success rate and an increase in error rate and average response time across all QoS levels (as shown in Fig. 7).

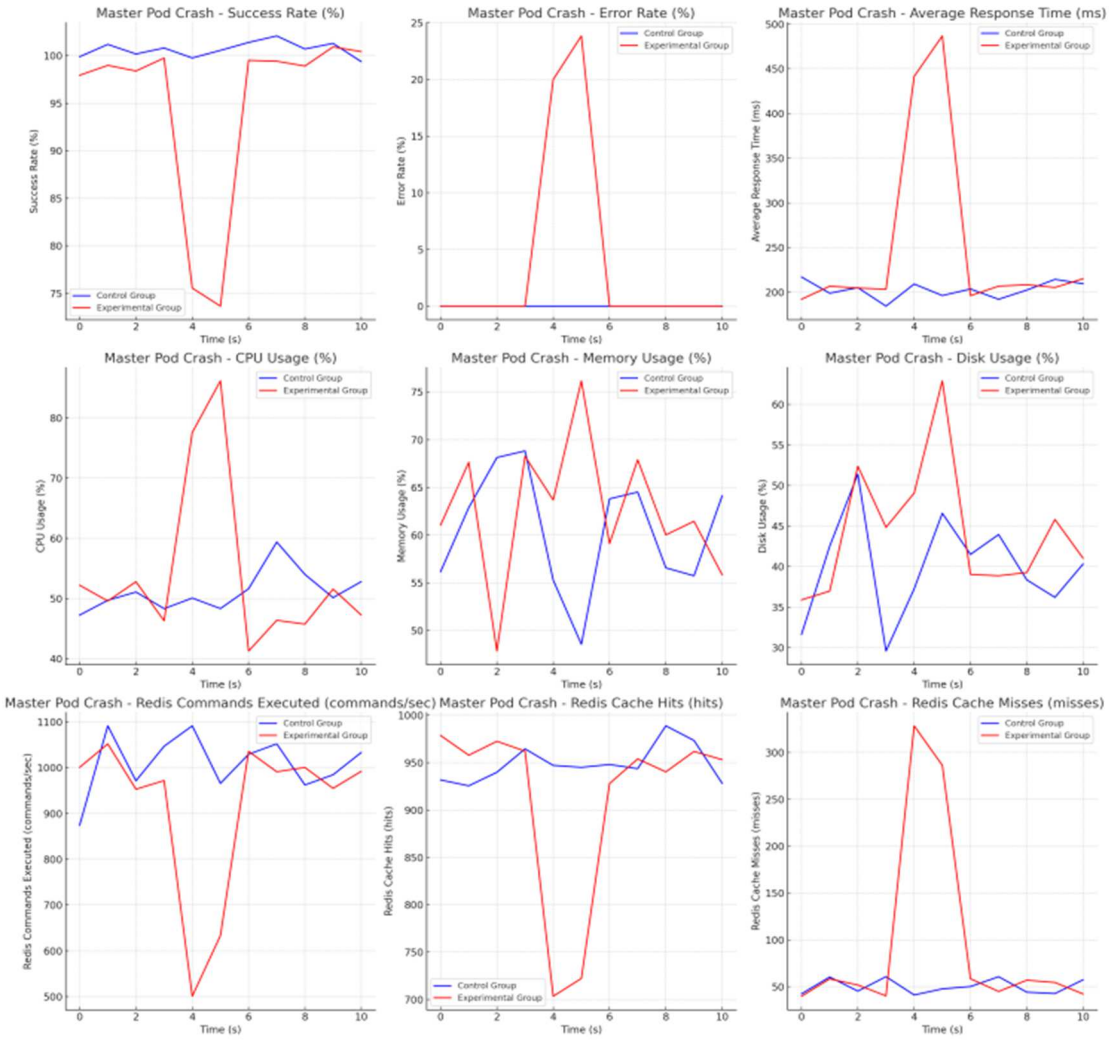

Fig. 7 Line chart of the comparison of results for control group and experimental group

2) Experiment Group 2-Network Latency: This experiment introduced network latencies of 5 seconds and 10 seconds to the traffic to the user-facing MQTT broker. The hypothesis was partially verified. With 5-second latency, the connection remained stable, and no events were lost. However, with a 10-second latency, some events were lost. The results indicated significant impacts on success rate, error rate, and average response time across all QoS levels (as shown in Fig. 8).

The experimental results from the three key fault injection scenarios-external service faults, internal service faults, and cloud provider faults—provide valuable insights into the resilience of the MQTT-based cloud-native IoT system.

1) External Service Fault Injection: The experiments injecting DNS errors and simulating network partitions significantly impacted system performance. During DNS request errors, the success rate dropped from 97.6% to 87.2%, and the average response time increased from 788 milliseconds to 1325 milliseconds. Network partitions reduced the success rate to 85.3% and increased the average response time to 1400 milliseconds. These results validate the hypothesis that external service faults significantly affect system stability, highlighting the need for robust fault tolerance mechanisms to handle such disruptions.

2) Internal Service Fault Injection: In the internal fault injection scenario, focusing on Redis master pod crashes, the experiments demonstrated a notable decrease in system performance. The success rate dropped to 75%, and the error rate increased to 25%, with the average response time increasing from 200 milliseconds to 450 milliseconds. These findings underscore the critical role of internal service stability in maintaining overall system performance and the effectiveness of tools like Redis Sentinel in mitigating such faults.

3) Cloud Provider Fault Injection: The experiments involving network partitions and network latencies on MQTT brokers showed that these faults significantly affect system performance. Network partitions resulted in a success rate drop to 65% for QoS 0 and an error rate increase to 35%, with an average response time of 550 milliseconds. Network latencies caused the success rate to drop to 75% with a 10-second latency for QoS 0, and the average response time increased to 450 milliseconds. These results validate the hypothesis that network faults substantially impact system stability and response times, especially under higher latency conditions.

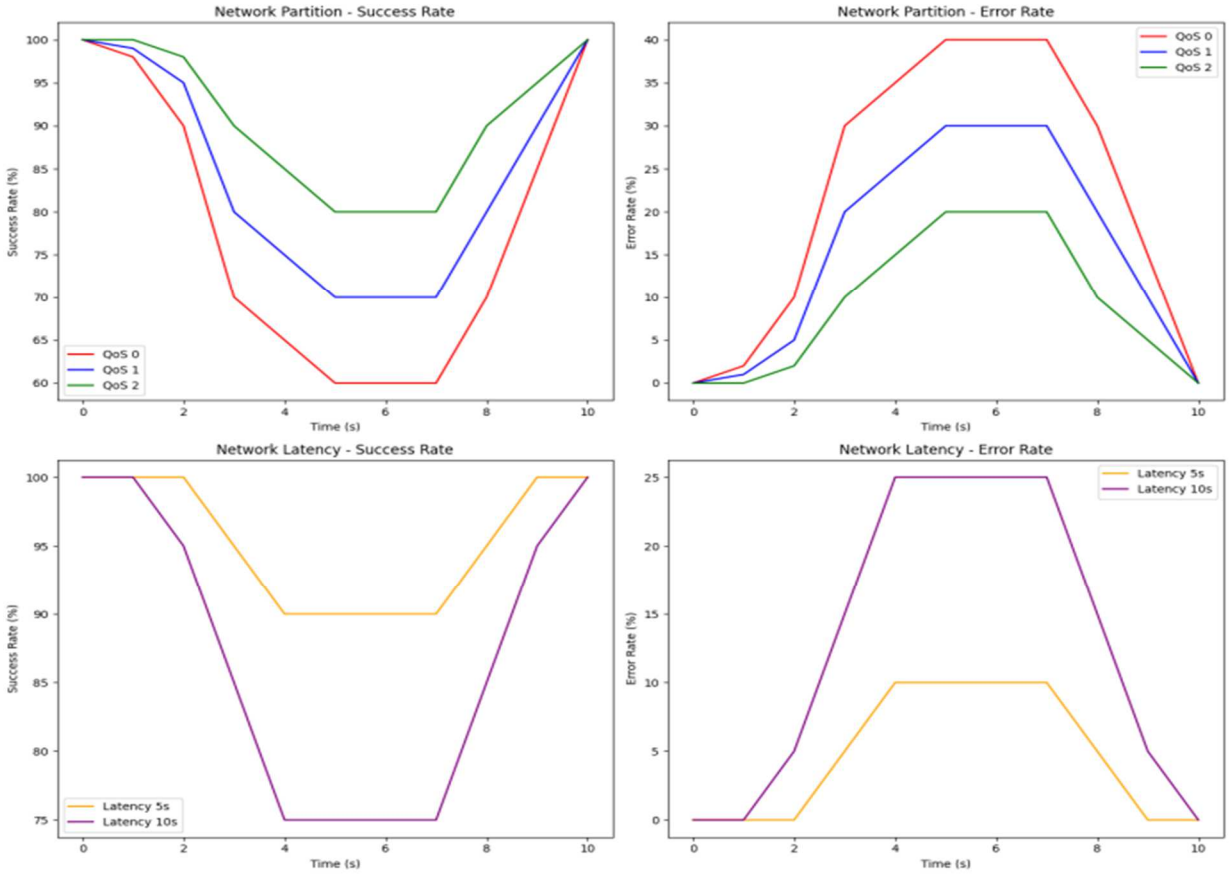

Fig. 8 Line graph of the result of cloud provider fault injection experiment

IV. Conclusion

This study investigates the application of a chaos engineering-based assessment model and tools in MQTT cloud-native IoT systems by evaluating the impact of external fault injection, internal fault injection, and cloud service provider fault injection on system performance. Despite limitations such as specific cloud-native environments (e.g., AWS) and a focus on MQTT protocols, the developed chaos engineering model and toolkit were validated through extensive empirical research, demonstrating their effectiveness in revealing system vulnerabilities and enhancing fault tolerance, resilience, and adaptability. The contributions include developing a comprehensive chaos engineering assessment model, designing and implementing assessment tools, validating tool effectiveness, and providing actionable improvement suggestions. Future work aims to expand fault types and scenarios, validate tools across multiple cloud platforms, enhance automation, and apply the tools to other protocols and system architectures. These efforts will further improve the robustness and reliability of cloud-native IoT systems.

Acknowledgment

The authors gratefully acknowledge the support provided by Universiti Putra for financial assistance.

References

[1] Z. Shu and G. Yan, "IoTInfer: Automated Blackbox Fuzz Testing of IoT Network Protocols Guided by Finite State Machine Inference," *IEEE Internet of Things Journal*, vol. 9, no. 22, pp. 22737–22751, Nov. 2022, doi: 10.1109/jiot.2022.3182589.

[2] D. Silva, L. I. Carvalho, J. Soares, and R. C. Sofia, "A Performance Analysis of Internet of Things Networking Protocols: Evaluating MQTT, CoAP, OPC UA," *Applied Sciences*, vol. 11, no. 11, p. 4879, May 2021, doi: 10.3390/app11114879.

[3] S. V. Mukherji, R. Sinha, S. Basak, and S. P. Kar, "Smart Agriculture using Internet of Things and MQTT Protocol," *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*, Feb. 2019, doi:10.1109/comitcon.2019.8862233.

[4] S. Qazi, B. A. Khawaja, and Q. U. Farooq, "IoT-Equipped and AI-Enabled Next Generation Smart Agriculture: A Critical Review, Current Challenges and Future Trends," *IEEE Access*, vol. 10, pp. 21219–21235, 2022, doi: 10.1109/access.2022.3152544.

[5] M. Pyingkodi et al., "Sensor Based Smart Agriculture with IoT Technologies: A Review," *2022 International Conference on Computer Communication and Informatics (ICCCI)*, pp. 1–7, Jan. 2022, doi: 10.1109/iccci54379.2022.9741001.

[6] A. Basiri et al., "Chaos Engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, May 2016, doi: 10.1109/ms.2016.60.

[7] P. Dedousis, G. Stergiopoulos, G. Arampatzis, and D. Gritzalis, "Enhancing Operational Resilience of Critical Infrastructure Processes Through Chaos Engineering," *IEEE Access*, vol. 11, pp. 106172–106189, 2023, doi: 10.1109/access.2023.3316028.

[8] S. Nikolovski and P. Mitrevski, "Data Protection and Recovery Performance Analysis of Cloud-Based Recovery Service," 2023 58th *International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*, Jun. 2023, doi:10.1109/icest58410.2023.10187249.

[9] X. Tang, "Reliability-Aware Cost-Efficient Scientific Workflows Scheduling Strategy on Multi-Cloud Systems," *IEEE Transactions on Cloud Computing*, vol. 10, no. 4, pp. 2909–2919, Oct. 2022, doi:10.1109/tcc.2021.3057422.

[10] A. S. Shaikh, "A Survey on Exchanging Data Using MQTT Protocol in Arduino," *International Journal for Research in Applied Science and Engineering Technology*, vol. 9, no. VII, pp. 3081–3082, Jul. 2021, doi: 10.22214/ijraset.2021.37007.

[11] A. R. Alkhafajee, A. M. A. Al-Muqarm, A. H. Alwan, and Z. R. Mohammed, "Security and Performance Analysis of MQTT Protocol with TLS in IoT Networks," *2021 4th International Iraqi Conference on Engineering Technology and Their Applications (IICETA)*, pp. 206–211, Sep. 2021, doi: 10.1109/iiceta51758.2021.9717495.

[12] A. Awajan, "A Novel Deep Learning-Based Intrusion Detection System for IoT Networks," *Computers*, vol. 12, no. 2, p. 34, Feb. 2023, doi: 10.3390/computers12020034.

[13] Y. Chen, Y. Sun, C. Wang, and T. Taleb, "Dynamic Task Allocation and Service Migration in Edge-Cloud IoT System Based on Deep Reinforcement Learning," *IEEE Internet of Things Journal*, vol. 9, no. 18, pp. 16742–16757, Sep. 2022, doi: 10.1109/jiot.2022.3164441.

[14] E. Gómez-Marín, L. Parrilla, G. Mauro, A. Escobar-Molero, D. P. Morales, and E. Castillo, "RESEKRA: Remote Enrollment Using SEaled Keys for Remote Attestation," *Sensors*, vol. 22, no. 13, p. 5060, Jul. 2022, doi: 10.3390/s22135060.

[15] G. Peralta, P. Garrido, J. Bilbao, R. Agüero, and P. M. Crespo, "On the Combination of Multi-Cloud and Network Coding for Cost-Efficient Storage in Industrial Applications," *Sensors*, vol. 19, no. 7, p. 1673, Apr. 2019, doi: 10.3390/s19071673.

[16] F. Poltronieri, M. Tortonesi, and C. Stefanelli, "ChaosTwin: A Chaos Engineering and Digital Twin Approach for the Design of Resilient IT Services," *2021 17th International Conference on Network and Service Management (CNSM)*, Oct. 2021, doi:10.23919/cnsm52442.2021.9615519.

[17] M. Rozsíval and A. Smrčka, "NetLoiter: A Tool for Automated Testing of Network Applications using Fault-injection," *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 207–210, Jun. 2023, doi:10.1109/dsn-w58399.2023.00057.

[18] L. Nurfiqin, "Analisis Quality Of Service (QoS) Protokol MQTT dan HTTP Pada Sistem Smart Metering Arus Listrik," *Jurnal Repositor*, vol. 3, no. 1, Dec. 2020, doi: 10.22219/repositor.v3i1.1084.

[19] R. Zitouni, J. Petit, A. Djoudi, and L. George, "IoT-Based Urban Traffic-Light Control: Modelling, Prototyping and Evaluation of MQTT Protocol," *2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. 182–189, Jul. 2019, doi: 10.1109/ithings/greencom/cpscom/smartdata.2019.00051.

[20] D. Borsatti, W. Cerroni, F. Tonini, and C. Raffaelli, "From IoT to Cloud: Applications and Performance of the MQTT Protocol," 2020 *22nd International Conference on Transparent Optical Networks (ICTON)*, Jul. 2020, doi: 10.1109/icton51198.2020.9203167.

[21] D. Eridani, K. T. Martono, and A. A. Hanifah, "MQTT Performance as a Message Protocol in an IoT based Chili Crops Greenhouse Prototyping," *2019 4th International Conference on Information Technology, Information Systems and Electrical Engineering (ICITISEE)*, pp. 184–189, Nov. 2019, doi:10.1109/icitisee48480.2019.9003975.

[22] B. Mishra, B. Mishra, and A. Kertesz, "Stress-Testing MQTT Brokers: A Comparative Analysis of Performance Measurements," *Energies*, vol. 14, no. 18, p. 5817, Sep. 2021, doi: 10.3390/en14185817.

[23] S. Arora and A. Ksentini, "Dynamic Resource Allocation and Placement of Cloud Native Network Services," *ICC 2021 - IEEE International Conference on Communications*, Jun. 2021, doi:10.1109/icc42927.2021.9500276.

[24] D. Breitgand, V. Eisenberg, N. Naaman, N. Rozenbaum, and A. Weit, "Toward True Cloud Native NFV MANO," *2021 12th International Conference on Network of the Future (NoF)*, Oct. 2021, doi:10.1109/nof52522.2021.9609908.

[25] W. Liao, & J. Draper, "Cloud Computing and Docker Containerization: A Survey", *Proceedings of the 2019 Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2019, doi:10.1109/PRDC47759.2019.8997375

[26] H. Jernberg, "Building a Framework for Chaos Engineering", LU-CS-EX, 2020.

[27] D. Craveiro, & J. Barreiros, "Chaos Engineering Tool Analysis", 2023.

[28] A. Gangolli, Q. H. Mahmoud, and A. Azim, "A Systematic Review of Fault Injection Attacks on IoT Systems," *Electronics*, vol. 11, no. 13, p. 2023, Jun. 2022, doi: 10.3390/electronics11132023.

[29] A. Pierce, J. Schanck, A. Groeger, R. Salih, and M. R. Clark, "Chaos engineering experiments in middleware systems using targeted network degradation and automatic fault injection," *Open Architecture/Open Business Model Net-Centric Systems and Defense Transformation* 2021, p. 8, Apr. 2021, doi: 10.1117/12.2584986.

[30] L. Zhang, " Application-Level Chaos Engineering". *PhD thesis*, KTH Royal Institute of Technology, 2022.