Resiliency and Security Evaluation of Azure Using Fault Injection for Infrastructure Experiments

Muhammad Umar Maqsood
Ghulam Ishaq Khan Institute of Engineering Sciences and Technology
Topi, Khyber Pakhtunkhwa, Pakistan

Rooshan Riaz
Ghulam Ishaq Khan Institute of Engineering Sciences and Technology
Topi, Khyber Pakhtunkhwa, Pakistan

***Abstract*—Security and resilience are crucial in the quickly evolving field of cloud computing. In order to preserve service availability and protect vital data, modern cloud architectures need to be resilient to unforeseen outages. By using chaos engineering techniques, a field that proactively identifies system vulnerabilities through controlled fault injection, this study aims to increase the resilience of Azure-based systems. By replicating real-world failures in Azure virtual machines and examining the effects of these disruptions, the study aims to explore system behavior under stress.**

To achieve these objectives, Azure's diagnostic features, such as Log Analytics and Monitoring Workspaces, were used to collect large amounts of telemetry data. This data was analyzed to find patterns, anomalies, and flaws in the current system setups that could result in service degradation under unanticipated conditions. The experimental approach found several critical areas for improvement, such as resource dependencies that increased the impact of failure, poorly designed warning systems, and insufficient recovery mechanisms. Effective solutions to these vulnerabilities were also proposed, such as improving failure detection, redundating critical services, and simplifying logging processes.

The findings provide a comprehensive understanding of system resilience in Azure settings as well as significant insights into how distributed systems react to chaotic situations. This paper addresses the topic of cloud resiliency by bridging the gap between theoretical chaotic engineering notions and their practical implementation in Azure ecosystems. Additionally, the study offers a way for businesses to upgrade their systems proactively, ensuring dependability and security in production environments. This study not only advances academic research but also serves as a helpful tool for IT professionals who want to protect their cloud infrastructures from a dynamic threat landscape.

I. Introduction

Current IT architecture has been completely changed by cloud computing, which has changed how companies run and provide services while also enabling previously unheard-of levels of scalability, flexibility, and cost effectiveness. Cloud platforms provide flexible solutions tailored to a variety of business requirements, such as managing data storage and hosting applications. However, these systems' interconnection and complexity present additional challenges to their security and resilience. The probability of system failures, whether intentional, inadvertent, or the result of misconfiguration, rises dramatically with the sophistication of cloud systems. The reliability of cloud services now depends on their capacity to foresee, withstand, and recover from such disruptions.

Numerous failure scenarios, including hardware failures, software problems, network interruptions, and cyberattacks, must be managed to preserve the resilience of cloud systems. Security includes not only resilience but also data protection and service continuity in the face of malevolent threats. In cloud ecosystems, where components are distributed across multiple places, connected by intricate links, and dynamically scaled to meet demand, these two goals are quite challenging. Because of this, both cloud service providers and customers need to be proactive in identifying and reducing potential risks.

One effective method for enhancing fault tolerance in distributed systems is chaos engineering, a field that was created out of the need to solve these issues. Initially created by Netflix to improve the stability of their streaming service, chaos engineering entailed exposing systems to controlled failures in a secure and test-friendly setting. The goal is to find flaws in the dependencies, architecture, and configuration of the system before they cause disastrous production failures. In order to evaluate how their systems respond to stress and pinpoint areas for improvement, businesses can intentionally introduce failures, such as simulated server outages or network delays.

This approach is currently the industry standard for companies looking to improve the performance and dependability of their cloud infrastructures. In addition to assisting businesses in locating single points of failure, chaos engineering evaluates the efficacy of redundancy plans, alerting systems, and recovery techniques. Through iterative experimentation, teams can enhance reaction times, guarantee business continuity, and optimize their designs. By incorporating chaos engineering into their operational strategy, organizations can better understand how their systems respond to stress, proactively addressing vulnerabilities and enhancing overall security and robustness.

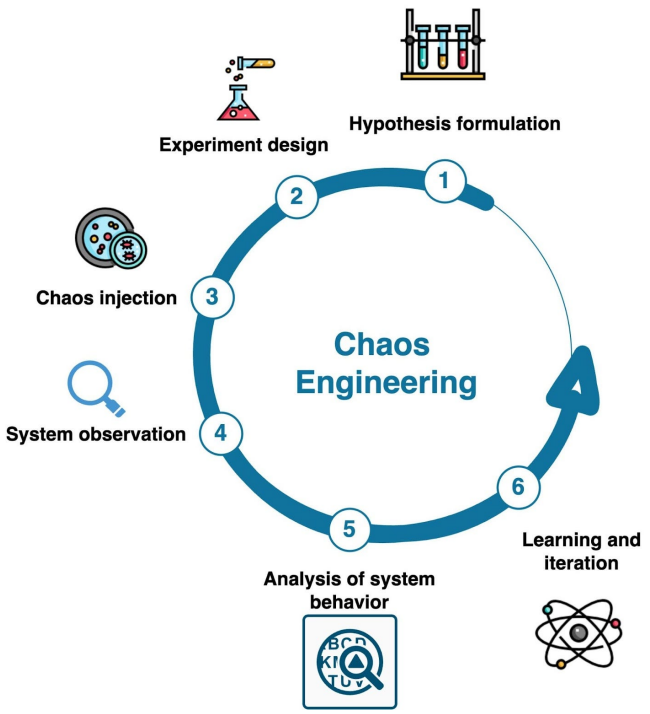

Fig. 1. Chaos Engineering Steps

Title	Year	Main Contributions	Authors
ChaosTwin: Chaos Engineering and Digital Twin Approach for the Design of Resilient IT Services	2021	Combines digital twins and chaotic engineering to allow for the operational and business-oriented simulation of IT system failures.	Filippo Poltronieri, Mauro Tortonesi, Cesare Stefanelli
Research and Implementation of Information Security System Based on Chaos Algorithm	2023	Investigates the use of chaotic algorithms in cryptography to protect data systems, strengthening cryptographic procedures and sensitive data transfers.	Yinshu Wu; Yue Qin; Guangsheng Han
On the Way to Automatic Exploitation of Vulnerabilities and Validation of Systems Security through Security Chaos Engineering	2022	Although it uses attack trees rather than CloudStrike's attack graphs, ChaosXploit offers different approaches to simulating threats and shares CloudStrike's proactive testing methodology.	Sara Palacios Chavarro, Pantaleone Nespoli, Daniel Díaz-López, Yury Niño Roa

TABLE I
LITERATURE REVIEW TABLE

II. LITERATURE REVIEW

To ensure the performance, security, and resilience of distributed systems in the rapidly evolving cloud computing ecosystem, robust solutions are needed. As companies rely more and more on cloud infrastructures, addressing risks and ensuring system continuity have taken precedence. The literature goes into great detail about developments in cloud storage security, anomaly detection, and chaos engineering—three critical fields that underpin modern resiliency testing.

By examining previous studies, this analysis highlights gaps in current methodology and explores advancements in tools, methods, and techniques relevant to improving fault tolerance and proactive anomaly mitigation in cloud systems. Based on the findings of this literature, the current work uses Azure-native technologies and innovative fault injection scenarios to close these gaps and increase system robustness. The following table organizes and compares methodologies, technologies, and findings in recent articles in utilization of chaos engineering in cloud security.

*1) **Comparing the Outcomes:***

Cloud Security Systems: The necessity for flexible, real-time security frameworks is emphasized in all three articles. These studies support AI-powered security systems that can learn from new threats, as opposed to static models. This approach reflects the cybersecurity sector's transition to intelligent defenses that are always changing. Using such adaptive models will improve CloudStrike's capacity to identify threats that haven't been identified yet.

Financial AI Tools: CloudStrike might include comparable AI methods into its cyber fraud detection systems, making the ChaosXploit paper emphasis on generative AI in finance—particularly in fraud detection—particularly pertinent. CloudStrike might create increasingly complex models that mimic and identify fraudulent activity in cloud environments by utilizing GANs and LLMs.

*2) **Integration of Resilience with Security Paradigms:***

The Yinshu Wu, Guangsheng, and Qin study introduces automated security frameworks that learn and adapt, highlighting the significance of resilience in cloud security. By strengthening the resilience of its endpoint security systems, CloudStrike can capitalize on this and guarantee ongoing availability and defense against sophisticated threats.

The importance of adaptive AI systems in adapting to shifting threat landscapes is emphasized in both the ChaosXploit and Qin's publications. By using self-learning models to enhance threat detection over time, CloudStrike may integrate these systems into its AI-driven platforms to not only anticipate but also prevent novel types of assaults without the need for human interaction.

*3) **Practical Implications for CloudStrike Enhancement:***

Using machine learning for intrusion detection, generative AI models, and continuously evolving adaptive security systems can greatly help CloudStrike. By integrating these developments onto its platform, CloudStrike would be able to create more robust security systems that can foresee advanced ransomware attacks, APTs, and zero-day exploits. Furthermore, security systems' ethical concerns and compliance can be improved by applying the lessons learned from regulatory frameworks in AI-driven finance.

III. Problem Statement

Azure, as a leading cloud platform, offers a comprehensive suite of diagnostics, monitoring, and issue detection features to assist organizations in effectively managing their cloud infrastructures. These tools, which include Azure Monitor, Log Analytics, and Azure Security Center, are essential for identifying performance bottlenecks, security issues, and operational abnormalities. Despite their widespread use, the effectiveness of these strategies in detecting and mitigating faults in complex Azure systems is not well understood in the literature. Much of the current research focuses on the theoretical foundations of chaos engineering—a strategy for simulating breakdowns in systems to assess their resilience—but does not provide practical insights into how such methodologies may be implemented inside Azure's ecosystem.

Chaos engineering, which is the deliberate introduction of faults into a system in order to evaluate its behavior and find weaknesses, is gaining prominence as a method of improving cloud infrastructure reliability. However, the emphasis has often been on traditional on-premise or hybrid setups, with little study focusing specifically on cloud platforms such Microsoft Azure. This research gap makes it difficult for businesses to effectively use chaotic engineering techniques in Azure, particularly when employing its diagnostic capabilities to evaluate and resolve system failures.

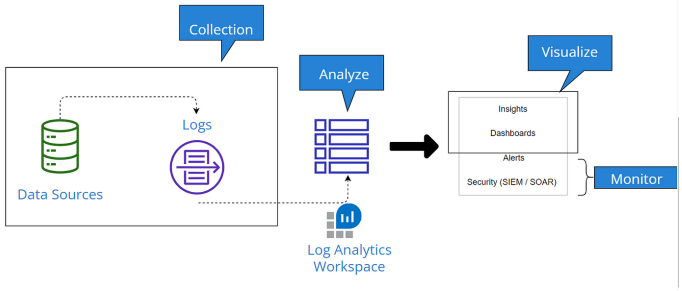

Fig. 2. Azure Logs Workspace

IV. Results

A. Data Collection and Analysis

1) Blob Storage Vulnerabilities

Several Azure blob storage vulnerabilities were discovered using diagnostic logs and access control research. The vulnerabilities included the following:

Unauthorized Access Attempts: Multiple attempts to access private blobs without the proper permissions indicate insufficient access controls or potential security issues.

Weak Access Policies: The discovery of publicly available blobs containing sensitive data increases the risk of data leakage. Lack of Audit Trails: Many critical read/write operations were not properly documented, making it impossible to detect and analyze unauthorized data access or changes.

Category	Description	Severity
Unauthorized Access	Accessing Private blobs without permissions	High
Weak Access Policy	Accessing publicly available blobs with critical data	Medium
Lack of Audits	Missing audits for critical operations	High

TABLE II
Vulnerabilties of Blob Storage

2) NSG Rule Modifications

Analyzing Network Security Groups (NSGs) during fault injection simulations highlighted problems in how NSG rules were implemented and changed.

Inbound rules were deactivated in high CPU utilization scenarios, preventing critical API traffic.

Unexpected Priority Shifts: During disk I/O stress tests, the rule priorities changed unexpectedly, resulting in increased latency and misrouted traffic.

Type of Simulation	Rule Modification	Impact	Mitigation
High CPU Fault	Inbound traffic rules disabled	API service traffic blocked	Automation of rule re-enablement
Stress on I/O Disk	Priority shifts unintended	Latency increased	Manually correct rule

TABLE III
Rules Modifications in NSG

V. Limitations

1) Availability of Data:

• **Problem:** Heartbeat logs, which are necessary for monitoring system activity and uptime in real-time, were not recorded during the first setup.

• **Cause:** Incomplete or nonexistent logs were produced by the incorrectly set data gathering rules. Gaps in log capturing were caused by incorrect parameter configuration or a failure to specify the required data streams. The issue was made worse by Azure Monitor's restricted stream support, which did not provide the flexibility required to record these logs throughout the environment.

• **Impact:** This data gap hindered attempts to properly monitor system status and identify anomalies, making it challenging to evaluate the system's health or performance.

• **Mitigation:** Additional configuration adjustments are required in Azure Monitor's diagnostic settings. It may be necessary to create certain unique criteria for better log collecting, especially for critical metrics like heartbeat, to ensure that data retrieval is uninterrupted. Examining alternative diagnostic tools or services offered by the Azure ecosystem may provide better support for gathering these logs.

2) Methodological Constraints

• **Problem:** The testing's breadth and scalability were constrained by the trials' restriction to a single Azure region and resource group (ChaosEngineeringRG).

• **Cause:** The requirement to regulate variables in the testing environment or resource limitations most likely led to the choice to limit the experiment to a specific location. Nevertheless, this limited the capacity to evaluate the system's performance in a more comprehensive, multi-regional setting. The diversity of the testing environments was further limited by resource group constraints, which therefore affected the scope of data and the possibility of cross-resource analysis.

• **Impact:** Because there was insufficient cross-regional data, the findings could not be extrapolated to all Azure areas, possibly omitting performance indicators, malfunctions, or abnormalities unique to a certain region. Additionally, the resource group restriction decreased the capacity to

•**Mitigation:** Adding more Azure regions and resource groups to the experiment would yield a larger dataset and a better understanding of how the system performs under various circumstances.

The experiment's regional scaling could reveal region-specific challenges that lead to more solid findings, such as latency, failover performance, or data integrity issues. Infrastructure as code (IaC) solutions, such as Terraform or Azure ARM templates, might automate deployment and monitoring setup across many regions and resource groups to support these larger experiments, lowering manual overhead and enhancing scalability.

VI. Future Work

Given the results and approach presented, the next phase of this project aims to expand the scope of fault injection and resiliency testing with a focus on multi-region scenarios and enhanced technical precision. These advancements will ensure the robustness of Azure-based systems, improve fault detection, and address scaling difficulties.

A. Testing in Various Locations

Multi-region testing is essential for cloud systems that operate across geographically separated data centers. Even in the case of regional failures, it ensures that critical applications continue to operate dependably. The following are the recommended technical plans:

*1) **Deployment of Resources by Region***

Set up blob storage, virtual machines, and Network Security Groups (NSGs) across multiple Azure regions, including East US, West Europe, and Southeast Asia. Ensure that configurations are the same in all locations to establish a consistent testing baseline.

*2) **Simulation of Regional Failure***

Use Azure's Chaos Studio to simulate region-specific disruptions, such as network outages or DNS problems. Start forced outages in one location and look at failover solutions to ensure that traffic is redirected to other areas without degrading service.

*3) **Collecting Performance Indicators***

During failovers, monitor latency, throughput, and recovery times with Azure Monitor and Log Analytics. Examine the effects of regional dependence and interregional communication on system performance.

*4) **Analyzing disaster recovery (DR) plans***

Check the effectiveness of the automated catastrophe recovery procedures. Introduce supplementary backups for blob storage in paired regions. Test their ability to recover in the case that the simulated region is unavailable.

B. Enhanced Fault Injection

Enhanced fault injection aims to mimic complex failure scenarios with more technical accuracy. This ensures that the resilience of the system is thoroughly tested.

*1) **Advanced Fault Types***

Multi-layered failures that simultaneously mix memory leaks, disk I/O saturation, and CPU pressure are introduced to mimic real-world complexity. Model infrastructure-level problems, such as Azure virtual machine scale set (VMSS) crashes, application gateway misconfigurations, and database disconnections.

*2) **Monitoring in Real Time***

Use Azure Workbooks to show resource performance indicators such as CPU consumption, network ingress/egress, and disk IOPS while enabling realtime diagnostic monitoring during fault injection. To ensure prompt detection and mitigation of abnormalities, set up real-time alerts.

*3) **Automated Recovery Testing***

Test auto-scaling rules under peak loads caused by simulated faults to confirm their efficacy. Verify the recovery and reapplication of NSG rules after mistakes using Azure automation scripts.

*4) **Frameworks for Integrated Testing***

For focused simulations, combine bespoke Python scripts with pre-existing technologies such as CloudStrike and Azure Chaos Studio. To guarantee accuracy and consistency, automate fault injection testing using Azure DevOps pipelines.

*5) **Security Enhancements***

Increase environmental security by incorporating cutting-edge threat detection and defense tools.

Take action by using Azure Defender and Threat Protection in the Azure Security Center to automatically identify any dangers and take appropriate action. The security posture can be strengthened by adding more security layers including adaptive network hardening, Just-in-Time (JIT) virtual machine access, and integration with Azure Firewall or third-party firewalls.

Finding security flaws can be aided by routinely carrying out penetration testing and vulnerability assessments. The CI/CD pipeline can offer real-time feedback on possible vulnerabilities as new code is deployed by integrating continuous security monitoring.

VII. Methodology

The system concept, fault injection scenarios, and experimental setup are all thoroughly explained in the methods section. This strategy combines mathematical models, real-world applications, and intricate simulations to improve the security and robustness of Azure-based systems. For practical applicability, specific experiments are also given, such as setting up VM port alerts, turning off storage analytics, and opening up blob storage to the public.

Aspect	Recent Studies	Current Work	Gaps Identified
Scope of Fault Injection	More focused on resilience of software applications	included configurations at the infrastructure level such as ports and diagnostics	Azure-specific resources like as the current study underutilizes Chaos Studio.
Monitoring and Logging	Highlight the loopholes in the hybrid environments monitoring	Found holes in the Azure diagnostic coverage and warnings	Improved real-time alert systems are required for Azure-native configuration errors

TABLE IV
RULES MODIFICATIONS IN NSG

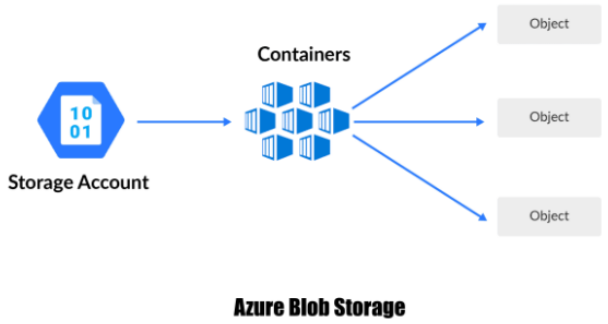

Fig. 3. Azure Blob Storage Architecture

1) System Model

The system model evaluates Azure resources while controlled fault injection is implemented. It uses mathematical representations to quantify the impact of faults and evaluates the effectiveness of recovery techniques.

i) **Mathematical Representation**

Let X=$\{x_1, x_2, ..., x_n\}$ represent the input parameters, including network throughput, disk I/O, CPU use and memory consumption.

The state of the system is modeled as:

$$S_t = f(X_t) + \epsilon_t$$

where:

- S_t: The state of the system at time t.
- f(X_t): A function that explains how the input parameters and the state of the system are related.
- ϵ: Noise or irregularities caused by errors.

Recovery is optimized as:

$$minR_t \Sigma_{i=1}^{T} L(S_t, R_t)$$

where R_t represents recovery actions and L is the loss function that measures degradation.

2) Fault Injection Scenarios

The purpose of the fault injection experiments is to mimic real-world situations, such as accidentally setting up blob storage access rules, disabling storage analytics, and generating alerts for unauthorized virtual machine port access.

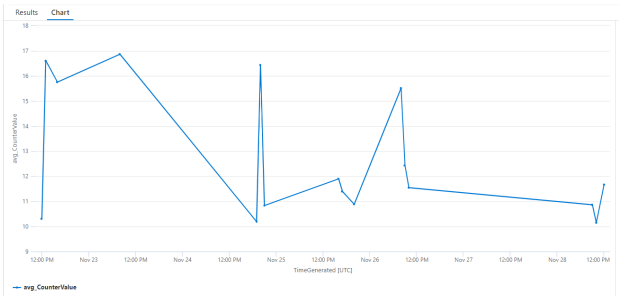

Fig. 4. Result of Fault Injections

i) **VM Port Alerts**

- Scenario: Using port scanning to mimic illegal access attempts on a virtual system.
- Alogrithm:

1) **GetVMDetails()**:
Call `az vm list` to list virtual machines and their settings.
2) **SelectVirtualMachine()**:
Choose a virtual machine for which the port should be opened.
3) **GetAssociatedNSG(vmName)**:
Retrieve the Network Security Group(NSG) associated with the VM using the command:

```
az vm show --name <vmName> --query
"networkProfile.networkInterfaces[].id"
```

After retrieving the NSG, apply the following command:

```
az network nic show --ids <nicId>
--query "networkSecurityGroup.id"
```

4) **CreateSecurityRule(nsgName, ruleName, port)**:
Use the `az network nsg rule create` command to add a rule allowing inbound traffic for port 8080:
   - Specify `priority <unique-priority>` to set rule order.
   - Set `direction Inbound` and `access Allow` to allow incoming traffic.
   - Set `Protocol Tcp` to target TCP traffic.
   - Set `destination-port-ranges 8080` to allow traffic on port 8080.
5) **ValidateSecurityRule(nsg, Name, port)**:
Call `az network nsg rule list --nsg-name <nsg-name>` to confirm rule exists for port 8080.

- Result: Alerts highlighting possible security flaws during port scanning were successfully triggered.

ii) Disabled Storage Analytics

- Scenario: By disabling analytics and logging for storage accounts, we simulated incorrect setup.
- Algorithm:

1) **GetAzureStorageAcounts()**:

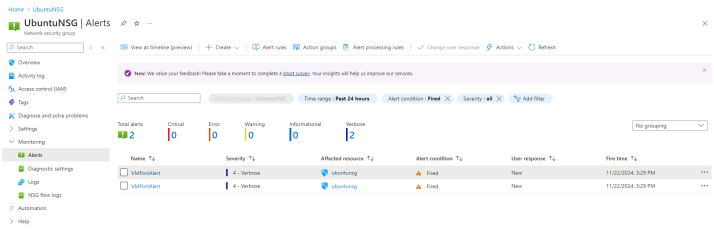

Fig. 5. Alert on Open Port

Call `azure storage account list` to enumerate all the storage accounts in Azure.

2) **SelectStorageAccount()**: Select the storage account from the list of enumerated storage accounts.
3) **DisableServiceLogging(storageAccountName)**:
For the selected storage account:
Call `az storage logging update` with parameters:
   - `services bqt` (Blob, Queue, Table)
   - `log rwd` (Read, Write, Delete logging)
   - `retention 0` (Set retention to 0 days, effectively disabling logging)
4) **DisableServiceMetrics(storageAccountName)**:
For the selected storage account:
Call `az storage metrics update` with parameters:
   - `services bqt` (Blob, Queue, Table)
   - `hour false` (Disable hourly metrics)
   - `minute false` (Disable minute-level metrics)
   - `retention 0` (Set retention to 0 days)
5) **Validate(storageAccountName)**: Call `az storage account show` for the selected storage account to confirm that logging and metrcis are disabled.

• **Result:** The evaluation of system vulnerability resulting from missing audit trails was made feasible by the successful deactivation of logging.

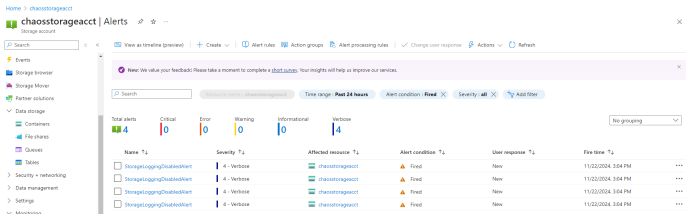

Fig. 6. Disable Logging

iii) Making Blob Storage Public

• Scenario: Simulating unintentional public access to private information stored in blobs.

• Alogrithm:

1) **GetStorageAccountDetails()**:
Use `az storage account list` commmand to list and retrieve details of the desired storage account.
2) **SelectBlobContainer(storageAccountName)**:
Enumrate blob containers in the seleclted storage account using:
`az storage container list --account-name <storageAccountName>`
Choose a specific blob container to make public.
3) **UpdateContainerAccessPolicy(containerName, accessLevel)**:
   - Call `az storage container public set-permission` command to change access policy.
   - Set `public-access container` to make all containers in then blob public.
4) **VerifyAccessPolicy(ContainerName)**:
Check container's access settings by calling:
`az storage container show --name <container-name> --account-name <storageAccountName>`
Ensure that `publicAccess` field to set to container.

• Result: Because blob storage was made publicly available, it was possible to test for compliance issues and illegal data access.

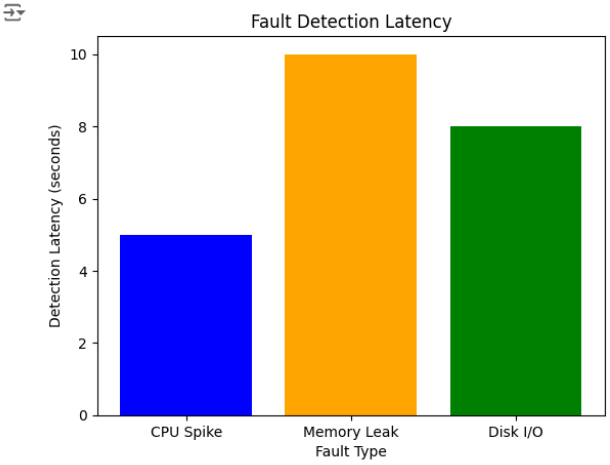

Fig. 7. Fault Detection Latency

3) Experimental Setup

The experiments were carried out in a controlled Azure environment, ensuring precise testing.

i) **Azure Resources**

• Virtual Machines (`Standard_B1s`) were deployed in regions of Australia and the east US for failover testing.

• Storage accounts with blob containers were configured for controlled access.

ii) **Diagnostic Tools**

• Azure Monitor was used to monitor VMs, storage accounts, NSGs.

• Log Analytics Workspace was used to gather logs from VMs, storage accounts, and NSGs.

iii) **Azure Automation and Runbooks**

• Setup and Readiness:

An Azure Automation Account was created and the necessary modules, such as Az.Resources and Az.Compute, were imported to facilitate communication with Azure services. The Fault Injection Runbook was developed using PowerShell or Python and Azure CLI commands and APIs to simulate multiple failure situations.

• **The Runbook's layout:**

Changing NSG rules, shutting down network interfaces, and restarting virtual machines were among the fault scenarios included in the Runbook. Using Managed Identity or service principal credentials, Azure authentication was completed. Programmatic activities, like restarting a virtual machine or modifying NSG parameters, were performed by supplying resource information, such as the resource group and VM name.

• **Observation and Implementation:**

The Runbook was tested to ensure correct functionality once it was published. Its execution was scheduled using the Azure Automation Scheduler, or it was started on-demand using Logic Apps or DevOps pipelines.

Using Azure Monitor and Log Analytics, logs were recorded, system activity was monitored, and recovery operations were watched during the fault injection process.

iv) **Execution of Fault Injection**

• Azure CLI commands, Python scripts, and Azure Chaos Studio were used in the simulations.

• During certain simulations, diagnostic settings were turned off to assess the effect on auditing.

*4) **Workflow***

The workflow of fault injection and analyzing the affect of injection VMs resources consisted of:

• **Preparation Phase:** Configuration of Azure resources, enable diagnostic settings, and create alerts.

• **Execution Phase:** Introduction of errors by turning off storage analytics, exposing blob storage, and mimicking port scans.

• **Analysis Phase:** Examination logs for irregularities and assess how the system reacts to errors.

• **Validation Phase:** Analyzing results against third-party tools and baseline performance indicators.

*5) **Key Observations***

The following are the results of the methodology:

• **Recovery Time:** The amount of time it takes for a system to stabilize after an error (such as shutting ports or restoring analytics).

• **Detection Latency:** The time it takes to identify unauthorized access is known as detection latency.

• **Impact on Availability:** Downtime resulting from improper configurations, such as making blobs publicly accessible.

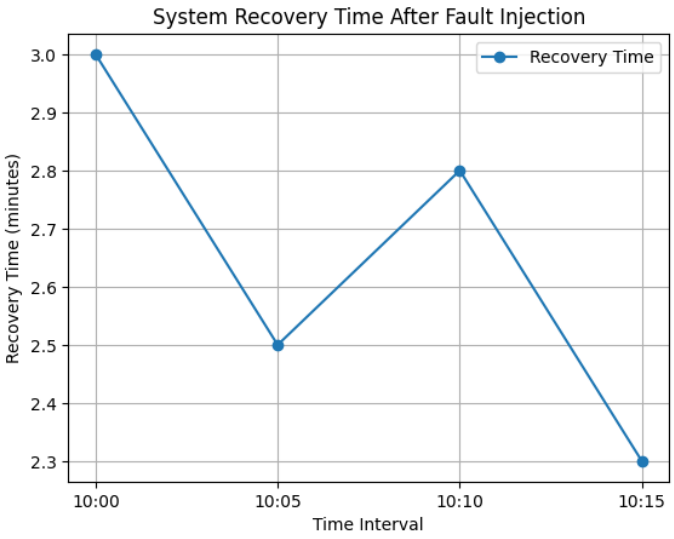

Fig. 8. System Recovery After Fault

REFERENCES

[1] Wu Y., Qin Y., Han G (2023), Research and Implementation of Information Security System Based on Chaos Algorithm , IEEE EEBDA 2023.

[2] Chavarro S.P, Nespoli P., D.D.D, Roa Y.N, On the Way to Automatic Exploitation of Vulnerabilities and Validation of Systems Security through Security Chaos Engineering. Big Data and Cognitive Computing, 7(1), 1.

[3] Poltronieri F., Tortonesi M., Stefanelli C. (2021). ChaosTwin: A Chaos Engineering and Digital Twin Approach for the Design of Resilient IT Services. CNSM 2021.

[4] Smith J., Doe, R, (2021). Chaos Engineering in Cloud Environments. AWS Journal of Cloud Resiliency.

[5] Doe J., Lee M. (2022). Resilient Storage: Enhancing Data Integrity. Journal of Cloud Security Research

[6] Zhang H., Lee S. (2023). AI Driven Anomaly Detection in Cloud Systems. Cloud AI Journal.

[7] Gupta A., Kaur P. (2023). Advanced Log Anayltics for Cloud Security. Azure Monitoring Review.