Journal of Applied Informatics and Computing (JAIC)
Vol.10, No.1, February 2026, pp. 910~922
e-ISSN: 2548-6861

Performance Evaluation of Multi-Cloud Failover Using Domain Name System

Cahya Zaelani [1]*, Galura Muhammad Suranegara [2]*
* Department of Telecommunication Systems, Universitas Pendidikan Indonesia
cahyaazaelanii@upi.edu [1], galurams@upi.edu [2]

Article Info

Article history:

Received 2025-11-17
Revised 2026-01-28
Accepted 2026-01-30

Keyword:

Multi-Cloud,
Cloud Computing,
Failover,
Domain Name System,
High Availability
Nginx.

ABSTRACT

This research implements and analyzes a multi-cloud failover system using DNS failover via AWS Route53 and Nginx reverse proxy load balancers on Google Cloud (primary) and Herza Cloud (backup), with AWS EC2 as shared backend web servers. An Ubuntu control node orchestrates deployments across these providers, enabling automatic traffic rerouting from the primary to secondary load balancer upon failure detection via health checks. Performance testing employed wrk benchmarking (4 threads, 250 connections, 300s) and Python monitoring scripts under baseline and failover scenarios with DNS TTLs of 30s, 60s, and 120s. Baseline yielded 2,291.81 req/s throughput, 108.42ms average latency, and 231.15ms p99 latency. Failover results showed TTL 30s optimal for reliability (152.65s downtime, 48.62% failed requests, 30.53s average recovery), outperforming TTL 60s (243.92s downtime, 83.48% failures due to health check mismatch) and TTL 120s (186.88s downtime) and TTL 30s is recommended for high availability in low-budget SMEs, balancing reduced downtime against DNS overhead. However, this approach is limited to small-scale infrastructure.

I. INTRODUCTION

The rapid adoption of cloud computing among small and medium enterprises (SMEs) has transformed how these organizations manage IT resources, enabling scalability, operational flexibility, and cost efficiency without requiring heavy upfront investments in physical infrastructure. This operational transformation is particularly significant in developing economies, where [1] reported that cloud-adopting SMEs achieved 2.5 times improvement in operational efficiency (OR=2.5, p<0.001), while [2] identified 50–75% reduction in time and effort for product development. Empirical studies indicate that cloud services help SMEs streamline business processes, improve data accessibility, and strengthen business continuity, which are critical for maintaining competitiveness in dynamic market environments [3]. As cloud-based applications increasingly support critical business operations, service availability has become a primary concern, where even short periods of downtime may lead to financial loss and degradation of user trust [4], [5]. Consequently, ensuring high availability (HA) is a fundamental requirement in cloud infrastructure design, particularly for web services that demand continuous accessibility under varying network conditions and traffic loads [6], [7].

To address availability challenges, various high availability mechanisms have been widely adopted, including server clustering, load balancing, and automated failover strategies. Load balancing distributes incoming traffic across multiple backend servers to improve performance and fault tolerance, while failover mechanisms ensure service continuity when a primary node becomes unavailable [8], [9]. Most existing implementations rely on single-cloud architectures, where redundancy is confined within one cloud provider. Although effective in mitigating local failures, such approaches remain vulnerable to provider-level outages, regional disruptions, and vendor lock-in issues [10], [11]. These limitations have encouraged the adoption of multi-cloud architectures that leverage resources across different cloud providers to enhance resilience and availability [12].

Among various failover techniques, Domain Name System (DNS)-based failover is widely used due to its simplicity,

scalability, and compatibility with heterogeneous cloud environments. DNS failover redirects client requests to alternative endpoints when service degradation is detected, typically controlled by Time To Live (TTL) parameters that determine DNS cache duration [13], [14]. Several studies highlight that TTL configuration plays a crucial role in failover responsiveness and service continuity, as shorter TTL values enable faster redirection at the cost of increased DNS query overhead [15], [16]. However, prior research largely focuses on functional validation or single-parameter observation, providing limited insight into how different TTL values quantitatively affect performance and reliability in real multi-cloud scenarios [17]. While cloud computing adoption has been extensively studied from business perspectives with 68.1% of SMEs prioritizing cost avoidance in hardware investments [1] and research [3] emphasizing infrastructure cost reduction as primary driver for Indonesian SMEs the technical optimization of failover mechanisms through DNS TTL configuration remains underexplored, particularly in low-budget multi-cloud architectures.

Furthermore, performance evaluation in DNS-based failover studies often lacks comprehensive metrics. Many works emphasize availability status without analyzing network quality indicators such as latency, jitter, throughput, and tail latency, which are essential for understanding user experience during failover events [18], [19]. Benchmarking tools such as wrk have been recognized as effective for generating sustained HTTP workloads and capturing detailed latency distributions, including p95 and p99 values [20], [21]. Nevertheless, systematic performance analysis combining DNS failover behavior with fine-grained reliability metrics such as downtime duration, recovery time, and failed request percentage remains limited, particularly under controlled TTL variations.

This study implements DNS-based multi-cloud failover using Nginx reverse proxy, with AWS EC2 backend, Google Cloud primary load balancer, Herza Cloud backup, and AWS Route53 orchestration. Performance evaluated across baseline performance measurement and TTL 30s, 60s, and 120s scenarios using wrk benchmarking and custom monitoring for network quality and recovery metrics and prioritizes high availability over throughput stability, which aligns with SME operational requirements.

II. METHODS

This research focuses on the implementation and performance analysis of a multi-cloud failover system based on DNS and Nginx load balancing mechanisms. The purpose of this methodology is to design, deploy, and evaluate a reliable web service infrastructure that can automatically maintain service availability during node or cloud provider failures. The experiment was conducted using three different cloud environments: Amazon Web Services (AWS) as the backend layer, Google Cloud as the primary load balancer 1 (LB1), and Herza Cloud as the secondary backup load balancer 2 (LB2). All configurations and management tasks were performed from a Control Node, which runs on Ubuntu Server within a VirtualBox virtual machine environment. This setup provides a centralized control and monitoring station for establishing secure SSH connections to all remote instances. Figure 1 shows the flow of implementation methodology and performance analysis of DNS and Nginx-based failover multi-cloud systems.

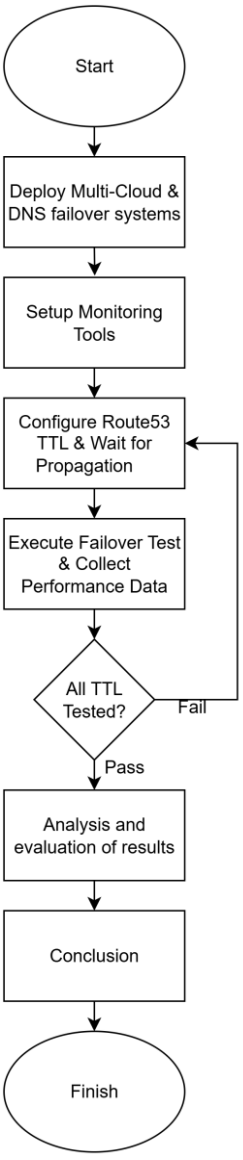

Figure 1. Multi-cloud failover methodology flow

A. Control Node Preparation

The control node serves as the central orchestration point responsible for managing, configuring, and monitoring all remote cloud instances deployed across multiple cloud providers during this research. It provides a unified environment for executing SSH connections and administrative tasks while automating the provisioning and

instantiation of virtual machines across distributed cloud platforms. This enables seamless deployment of identical server configurations on Google Cloud, AWS, and Herza Cloud.

B. AWS Backend Web Server Infrastructure

The second phase involves provisioning two independent web servers on Amazon Web Services (AWS) EC2 to serve as the backend application layer. These instances were deployed through AWS Management Console with configurations optimized for load balancing and high availability. AWS was selected for its global infrastructure, predictable performance, and suitability for cross-cloud redundancy mechanisms.

Table I presents the detailed specifications for Web Server 1 and Web Server 2.

TABLE I
WEB SERVER 1 & 2 SPECIFICATIONS

Component	Specification
Cloud Provider	Amazon Web Services (AWS)
Compute Service	EC2 (Elastic Compute Cloud)
Instance Type	t3.micro
vCPU	2 vCPU
RAM	1 GB
Storage	8 GB
Operating System	Ubuntu Server 22.04 LTS
Public IP	Elastic IP

SSH key pairs were configured for secure authentication by generating EC2 RSA keys through AWS Management Console. The private key aws-key.pem was transferred to the control node via SCP, with permissions restricted to read-only for owner following security best practices. The following is to change the access permissions:

```
# chmod 400 aws-key.pem
```

Figure 2. AWS key pair generation for secure ssh authentication

Two separate index pages were configured for Web Server 1 and Web Server 2 to uniquely identify backends and validate load distribution during failover testing. The following is how to install Nginx and change the index page:

```
# sudo apt install nginx -y
#echo "Hello from Webserver1 (AWS)" | sudo tee /var/www/html/index.html
#echo "Hello from Webserver2 (AWS)" | sudo tee /var/www/html/index.html
```

Figure 3 shows that these unique responses enable real-time detection of routing behavior and verification of load balancing, while also helping monitor DNS propagation and reveal traffic inconsistencies that provide insight into overall failover performance.

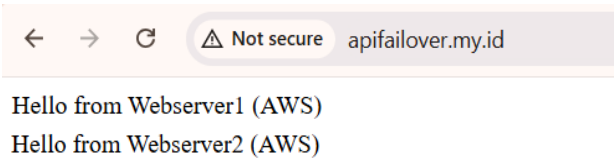

Figure 3. Web server 1 and Web server 2 response output

C. Google Cloud Primary Load Balancer Configuration

The primary load balancing tier utilizes Google Cloud Compute Engine deployed in the Jakarta region (asia-southeast2-a), designated as Load Balancer 1 (LB1). The Virtual Machine instance was provisioned through the Google Cloud Console with specifications detailed in Table II.

TABLE II
LOAD BALANCER 1 SPECIFICATIONS

Component	Specification
Cloud Provider	Google Cloud Platform
Compute Service	Compute Engine
Instance Role	Primary Load Balancer
Zone	Jakarta, asia-southeast2-a
Machine Type	e2-medium (2 vCPU, 4 GB RAM)
Storage	20 GB
Operating System	Ubuntu Server 22.04 LTS
Web Server Software	Nginx 1.18.0

SSH access was configured by first generating an RSA key pair locally through the ssh-keygen command in the Linux terminal. The resulting public key was then added to the instance's authorized keys by inserting it into GCP metadata settings under the SSH keys field. The following is how to create an rsa key:

```
# mkdir GCP
# cd GCP
# ssh-keygen -t rsa -f gcp -C gcpkey
```

After the RSA key pair was generated on the control node for SSH authentication to Google Cloud VM, where Nginx 1.18.0 was installed as reverse proxy. Nginx was installed using the following commands:

```
# sudo apt install nginx -y
# sudo systemctl enable nginx
# sudo systemctl start nginx
```

Once Nginx web server (version 1.18.0) was installed and running, the load balancing logic was implemented using the upstream module. The backend server pool and reverse proxy directives were defined in a configuration file at "/etc/nginx/conf.d/loadbalancer.conf". The following shows how to configure this file:

```
# sudo nano /etc/nginx/conf.d/loadbalancer.conf
```

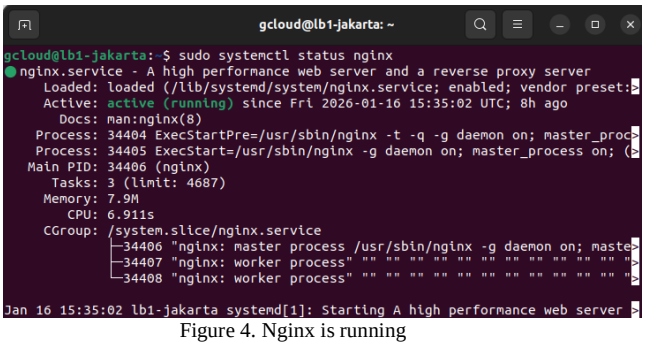

Figure 4. Nginx is running

The upstream block configures round-robin load balancing across two AWS EC2 instances on port 80, preserving client headers and isolating failed backends through health monitoring. Configuration syntax was validated using Nginx built-in testing before implementation.

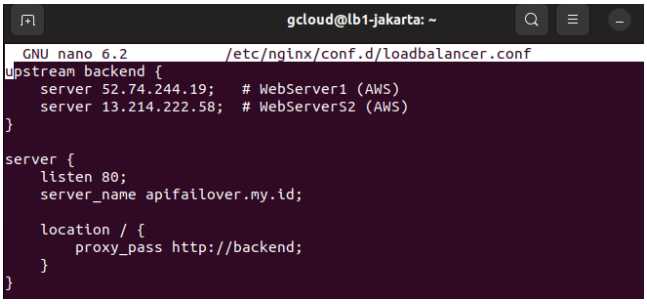

Figure 5. Google Cloud upstream configuration

Figure 6 shows successful Nginx configuration syntax validation command:

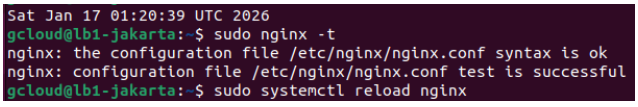

Figure 6. Nginx configuration syntax validation

D. Herza Cloud Backup Load Balancer Deployment

Load Balancer 2 (LB2) utilizes Herza Cloud KVM VPS in Indonesian infrastructure, selected for full root access and cost-effective pricing ensuring data residency. Deployment mirrors GCP configuration, with the key difference being SSH key generation performed locally on Windows command prompt and transferred to the control node server via SCP protocol. Table III shows the specifications of the backup load balancer.

TABLE III
LOAD BALANCER 2 SPECIFICATIONS

Component	Specification
Cloud Provider	Herza Cloud (Indonesia)
Compute Service	Virtual Private Server (KVM)
Instance Role	Secondary Load Balancer (Backup)
Region	Jakarta, Indonesia
Instance Type	1 vCPU, 1 GB RAM
Storage	20 GB SSD
Operating System	Ubuntu Server 22.04 LTS
Firewall	Allow: 22 (SSH), 80 (HTTP)
Web Server Software	Nginx 1.18.0

After instance provisioning SSH key workflows differed. GCP keys generated on Linux control node due to Windows issues, while Herza keys were successfully generated on Windows and transferred via SCP. The following shows the SSH key pair generation on Windows CMD and subsequent transfer via SCP:

```
# ssh-keygen -t rsa -b 4096 -C "herza-lb2" -f herza-key
# cat herza-key
# scp "C:\Users\Cahya\Downloads\herza-key"
username@IP_Public:~
```

The public key must then be uploaded manually to the Herza Cloud Panel under the SSH Keys section, where it is registered as an authorized key for later VPS provisioning. This process allows the VPS to be deployed with pre-configured key-based authentication, ensuring secure, passwordless access once the instance is created. After the public key is added, the private key is transferred securely to the server control node and restricted to comply with SSH security requirements before connecting via ssh. The following is how to change the RSA key permissions and access the VM:

```
# chmod 400 herza-key
# ssh -i ~/name_folder/herza-key username@IP_Public
```

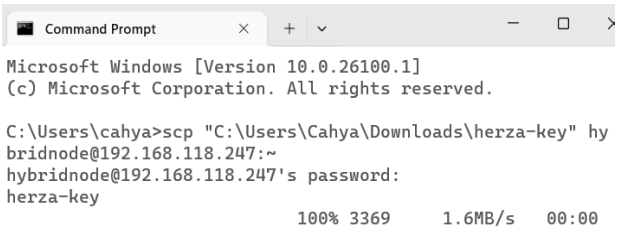

Figure 7. Sends the rsa key file to the control node server

After successfully establishing SSH connection to the Herza Cloud instance, Nginx installation and upstream configuration were performed following the same procedures as Load Balancer 1 (LB1). Nginx upstream config defines two AWS EC2 backends with round-robin, validated using “sudo nginx -t” then applied “via sudo systemctl restart nginx”.

E. DNS-Based Failover Orchestration

AWS Route53 implements DNS failover through hosted zone configuration, health check deployment, and primary-secondary routing policy implementation. Figure 8 shows the system flow of multi-cloud failover.

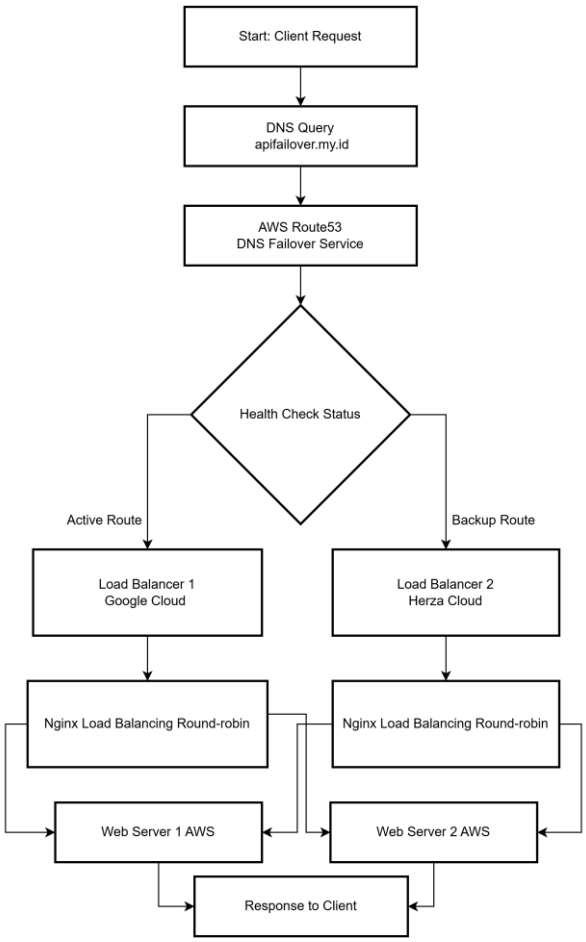

Figure 8. System flow from multi-cloud failover

The following section details the Hosted Zone configuration and domain registration that enables this DNS-based traffic management across multi-cloud environments.

1) Configuration Domain Registration: AWS Route53 hosted zone was created for apifailover.my.id with nameserver delegation configured at DomaiNesia using four AWS-assigned nameservers.

2) AWS Route53 Health Check Implementation for Load Balancer Monitoring: Automated failover relies on continuous health monitoring of Google Cloud Load Balancer 1 (LB1) and Herza Load Balancer 2 (LB2) using AWS Route53 HTTP health checks, with one targeting Load Balancer 1 (LB1) public IP 34.50.111.48:80. Figure 9 illustrates the health check configuration for Load Balancer 1 (LB1) in the AWS console, specifying protocol, endpoint path, check interval, timeout, and threshold values to ensure backend instances remain healthy and reliably serve user traffic.

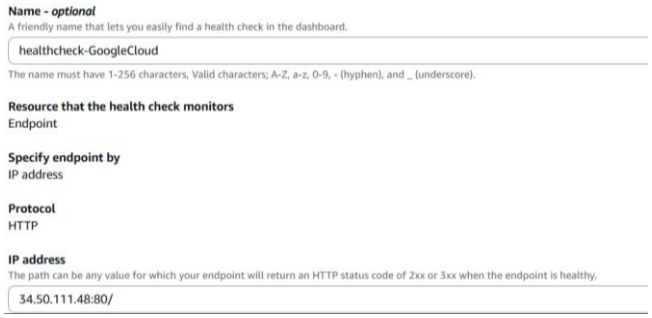

Figure 9. Configuration on health check Load Balancer 1

While the second monitors the Herza Cloud load balancer 2 (LB2) IP address 103.168.146.162 on the same port, both executing HTTP GET requests to the root path / to verify load balancer operational status.

3) Failover Routing Policy Configuration with Primary and Secondary Records: The DNS failover policy is implemented in Route53 by creating a hosted zone for apifailover.my.id with two A records using failover routing. The primary A record points to Google Cloud load balancer IP 34.50.111.48, marked as Primary, attached to a health check, with 60-second TTL.

The secondary failover record points to Herza Cloud load balancer IP 103.168.146.162 and is marked as Secondary with a 60-second TTL, remaining passive until the primary health check fails, after which Route53 directs traffic to this backup endpoint.

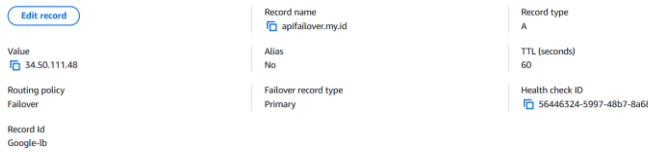

Figure 10. Configuring Google Cloud records in hosted zones

4) DNS Failover Configuration Verification: After completing the Route53 failover configuration, DNS propagation was verified using DNSChecker.org to ensure proper domain resolution across the global DNS infrastructure. The verification process tested the resolution of (apifailover.my.id) from multiple geographic locations, including San Francisco and Mountain View, California, representing different DNS resolver networks worldwide. As shown in Figure 11, the domain consistently resolved to

the primary load balancer IP address 34.50.111.48 across all tested locations. This result confirms that the Route53 nameserver delegation, health checks, and failover routing policies were correctly implemented and successfully propagated globally, ensuring reliable domain accessibility and high availability.

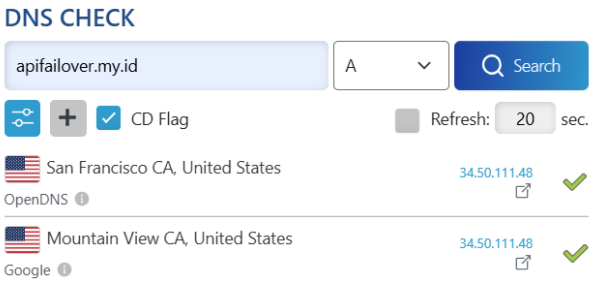

Figure 11. Domain has successfully propagated across global DNS

F. Performance Testing and Metrics Collection

Performance testing evaluates network quality and failover reliability using a dual-methodology approach. wrk (HTTP benchmarking tool) measures latency distribution, jitter, tail latency (p95/p99), and throughput under sustained load, while a custom Python monitoring script captures downtime duration, recovery time, and failed request percentages during simulated infrastructure failures.

The selection of TTL intervals (30s, 60s, 120s) was based on industry practices and AWS Route53 health check threshold (90s) [22]. The 250 concurrent connections simulate typical SME traffic patterns, aligning with empirical evidence from [1] showing that surveyed SMEs averaged 25 employees with moderate-scale operations, while avoiding infrastructure saturation on low-tier instances commonly deployed by budget-constrained organizations. Research [2] further validated that SMEs prioritize scalability (dynamic resource provisioning) and cost efficiency over peak performance capacity, justifying our conservative load testing approach.

1) Network Quality Metrics Testing: Network performance characteristics were measured using wrk (HTTP benchmarking tool), an open-source, high-performance HTTP load generator capable of producing sustained traffic loads while capturing detailed latency statistics. wrk was selected for its efficiency in measuring request-level metrics under controlled load conditions and its widespread adoption in cloud computing research for performance benchmarking. The testing protocol executed 5-minute (300s) benchmark sessions against the DNS-managed endpoint (http://apifailover.my.id) Packet loss was measured using ICMP echo requests with the following configuration:

```
# ./wrk -t4 -c250 -d300s --latency http://apifailover.my.id
#  ping -c 300 apifailover.my.id (packet loss)
```

Testing was conducted across four distinct scenarios to evaluate both baseline performance and DNS TTL impact on failover behavior:

- Scenario 1 Baseline Performance Measurement

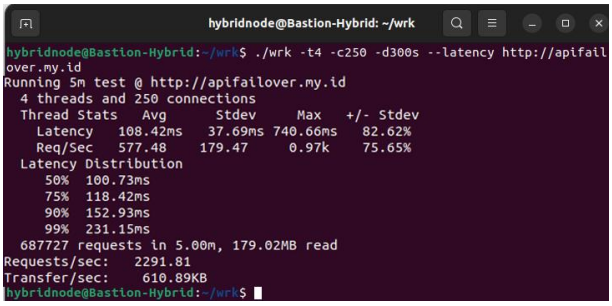

Figure 12. Results from baseline performance

All infrastructure components operational with no induced failures. This scenario establishes baseline performance metrics for comparison with failover scenarios. Figure 12 baseline wrk testing achieved 2,291 req/s throughput with 108 ms mean latency and 231 ms p99. Results indicate stable multi-cloud performance under normal conditions, showing zero packet loss, no connection errors, and consistent latency behavior for quantitative failover benchmarking.

- Scenario 2 DNS Failover with TTL 30 seconds

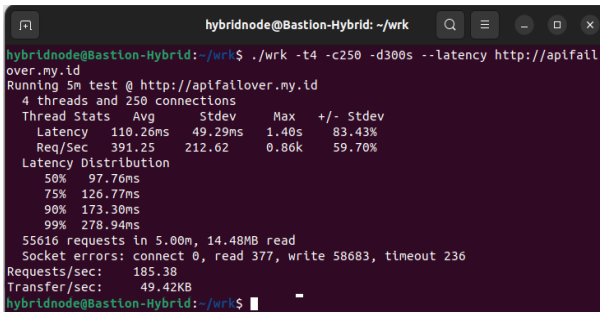

Figure 13. Results from TTL 30

Figure 13 generating a total of 55,616 requests. The observed performance metrics include a jitter of 49.29 ms, throughput of 185.38 requests per second, an average latency of 110.26 ms, and a p99 latency of 278.94 ms. A substantial number of socket errors were recorded, mainly triggered by frequent DNS re-resolution and repeated failover cycles during the test period. These conditions caused connection instability, increased timeout events, and reduced request success rates. As a result, overall throughput degraded by

approximately 91.9% compared to the baseline performance of 2,291.81 req/s, highlighting the adverse impact of overly aggressive DNS TTL configurations on system stability and performance.

- Scenario 3 DNS Failover with TTL 60 seconds

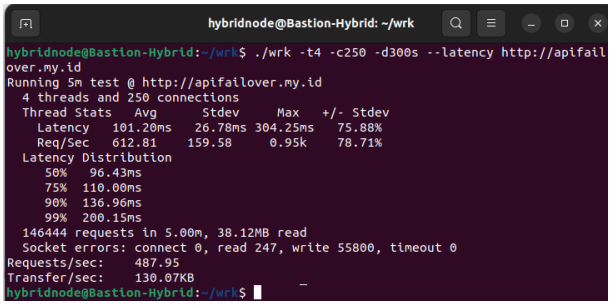

Figure 14. Results from TTL 60

Figure 14 shows TTL 60s testing with 250 concurrent connections over 5 minutes, completing 146,444 requests at 487.95 req/s, 101.20 ms average latency, Jitter 26.78ms, and 200.15 ms p99. Although 56,047 socket errors occurred, throughput was 2.6× higher than TTL 30s but still 78.7% below baseline due to DNS failover.

- Scenario 4 DNS Failover with TTL 120 seconds

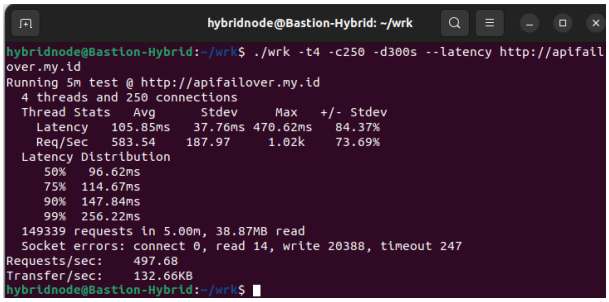

Figure 15. Results from TTL 120

Figure 15 shows TTL 120s testing using wrk with 250 concurrent connections over 5 minutes. TTL 120s and 60s reach ~498 and ~488 req/s with lower jitter, while TTL 30s suffers 185 req/s, highest jitter, and many socket errors, showing shorter TTLs worsen instability and overhead.

2) Failover Reliability Metrics Testing: Failover behavior assessment required continuous endpoint availability monitoring to detect downtime duration, recovery time, and failed request rates during infrastructure failures. Failover behavior was assessed using a custom Python monitoring script implementing polling-based availability checking. The script configuration defines four key parameters:

```
# url = "http://apifailover.my.id"
# interval = 1        # 1-second polling interval
# duration = 300      # 5-minute observation window
# timeout = 2         # 2-second request timeout
```

These parameters enable granular 1-second resolution monitoring over 5-minute windows, sufficient to capture complete DNS TTL propagation cycles (30-120 seconds) while the 2-second timeout distinguishes network delays from actual service unavailability. The monitoring loop executes HTTP GET requests at fixed intervals, recording timestamp-precise success/failure status for subsequent analysis.

- DNS Failover Reliability with TTL 30s

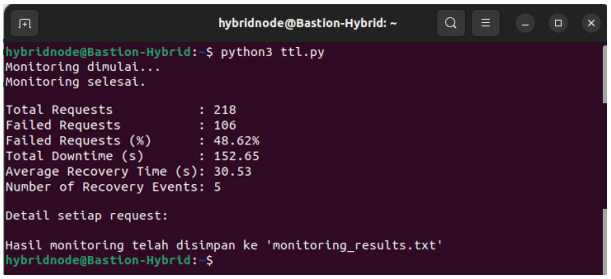

Figure 16. Results of the TTL 30 test

Figure 16 shows temporal failover behavior for TTL 30s with 5 recovery cycles across 218 requests in 5 minutes, achieving 51.38% success and 152.65s downtime, where frequent DNS cache expiration enables rapid repeated recovery, improving overall reliability despite slower individual recoveries.

- DNS Failover Reliability with TTL 60s

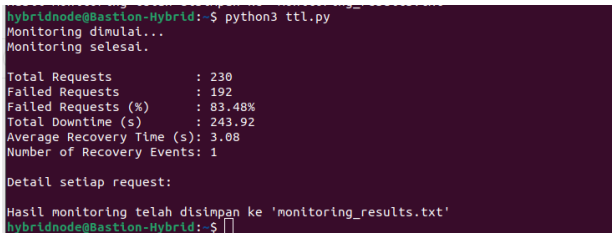

Figure 17. Results of the TTL 60 test

Figure 17 presents DNS failover monitoring results for the TTL 60s configuration, based on 230 requests executed over a 5-minute observation period. The experiment reveals a single prolonged recovery cycle with a very low success rate of only 16.52%, indicating that most requests failed during the failover process. The total recorded downtime reached 243.92 seconds, which is approximately 60% higher than the downtime observed in the TTL 30-second scenario. Although the measured average recovery time per event was relatively short at 3.08 seconds, this metric alone did not translate into

reliable service availability. The mismatch between the DNS cache expiration interval (60 seconds) and the health check detection window (90 seconds) resulted in a worst-case cache coherence condition, where clients continued to resolve unhealthy endpoints.

- DNS Failover Reliability with TTL 120s

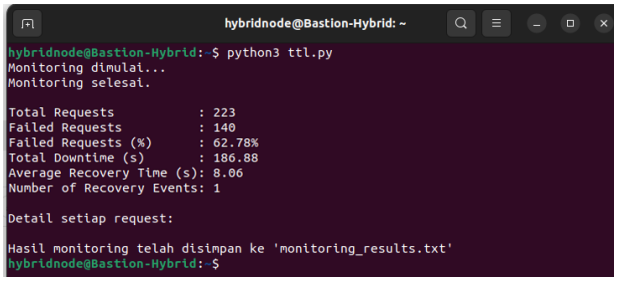

Figure 18. Results of the TTL 120 test

Figure 18 shows TTL 120s failover with 1 recovery cycles, 186.88s downtime, and 38.12% success over 230 requests, where longer TTL reduces recovery frequency yet avoids TTL 60s race conditions, providing intermediate, more stable reliability during infrastructure failures.

III. Results and Discussions

The tests were conducted across four scenarios representing different operational conditions: Scenario 1 (Baseline) establishes reference performance under baseline performance, while Scenarios 2-4 evaluate DNS failover with TTL configurations of 30s, 60s, and 120s during simulated primary server failures. Each scenario employed dual-methodology testing wrk for performance metrics (throughput, latency, jitter) and Python monitoring for reliability metrics (downtime, recovery cycles, success rates) over 5-minute windows with 250 concurrent connections. This section presents experimental findings from performance testing and failover reliability analysis across multiple DNS TTL configurations. Results are organized into three subsections: baseline performance characteristics, performance under failover conditions, and failover reliability metrics, followed by comparative analysis and discussion.

A. *Baseline Performance Characteristics*

Baseline performance testing establishes reference metrics for system behavior under normal operating conditions without any infrastructure failures and serves as a critical control scenario for subsequent experiments. Figure 19 demonstrates stable baseline performance, achieving a throughput of 2,291.81 requests per second, with an average latency of 108.42 ms and jitter of 37.69 ms, indicating consistent and predictable request processing. Although minor packet loss and socket errors of 2.33% were observed, these anomalies remained within acceptable limits and did not significantly impact service availability or responsiveness. The p99 latency of 231.15 ms defines the upper-bound performance experienced by 99% of user requests, capturing worst-case delay under healthy conditions. Collectively, these metrics establish a reliable performance baseline that reflects optimal system behavior. This baseline is essential for accurately quantifying the extent of performance degradation, increased latency, throughput reduction, and availability loss introduced by DNS failover mechanisms and recovery events evaluated in subsequent testing scenarios.

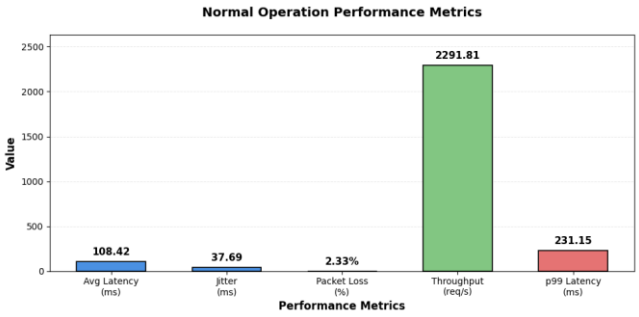

Figure 19. Baseline performance measurement test results

B. *Performance Under Failover Conditions*

This section evaluates network performance metrics during simulated primary infrastructure failures across three DNS TTL configurations. Each metric is analyzed independently with measurement methodologies, and comparative results.

1) Average Latency Analysis: Average latency represents the mean time required for request-response cycles, calculated as:

$$\text{Average Latency} = \frac{\sum_{i=1}^{n} \text{Latency}_i}{n}$$

Based on the measurements results in the Table IV, the average latency for each was calculated as follows:

$$\textbf{TTL 30s: } 110.26 \text{ ms} = \frac{6.132.295.16}{55.616}$$

$$\textbf{TTL 60s: } 101.20 \text{ ms} = \frac{14.820.132.80}{146.444}$$

$$\textbf{TTL 120s: } 105.85 \text{ ms} = \frac{15.811.241.15}{149.339}$$

Average latency calculations reveal paradoxical performance metrics across TTL configurations. TTL 60s achieves the lowest latency (101.20 ms) with 146,444 successful requests. However, latency and jitter metrics

during failover cannot be interpreted independently from failure rates, as they represent only surviving requests, while TTL 120s records 105.85ms across 149.339 requests both processing 2.6-2.7× more requests than TTL 30s (110.26ms, 55,616 requests). However, these favorable latency values reflect survival bias: only successful requests contribute to calculations, masking underlying 83.48% and 62.78% failure rates respectively, thereby invalidating apparent performance superiority during failover conditions.

2) Jitter Latency Analysis: Jitter quantifies latency variability as standard deviation of samples. TTL 30s exhibited highest jitter (49.29ms, +30.8% vs baseline 37.69ms) due to synchronized 30-second DNS cache expiration creating periodic spikes. TTL 60s achieved lowest (26.78ms, -28.9%), paradoxically masking 83.48% failure rate through survival bias. TTL 120s maintained near-baseline variability (35.46ms, -5.9%), confirming longer cache durations stabilize latency variance.

3) Packet Loss Analysis: Packet loss rate quantifies failed connection attempts relative to total requests. The following is the formula:

$$\text{Packet Loss (\%)} = \frac{\text{Socket Errors} + \text{Timeouts}}{\text{Total Requests}} \times 100$$

The packet loss metrics across all test scenarios were obtained as follows:

$$\textbf{Normal:}\quad \text{Packet Loss (\%)} = \frac{7}{300} \times 100 = 2.33\ \%$$

$$\textbf{TTL 30s:}\quad \text{Packet Loss (\%)} = \frac{48}{300} \times 100 = 16.00\ \%$$

$$\textbf{TTL 60s:}\quad \text{Packet Loss (\%)} = \frac{12}{300} \times 100 = 4.00\ \%$$

$$\textbf{TTL 120s:}\quad \text{Packet Loss (\%)} = \frac{60}{300} \times 100 = 20.00\ \%$$

Baseline 2.33% packet loss during baseline performance measurement was caused by resource saturation of AWS t3.micro with burst limit of only ~1,788 req/s while wrk testing achieved 2,291 req/s (28% overload), wrk aggressiveness with 4 threads generating 1,000 req/s instant burst (vs SME gradual ramp-up averaging 25 users), and Nginx keep-alive timeout 65s causing connection pool exhaustion at 250 concurrent connection.

4) Throughput Analysis: Throughput measures successfully completed requests per unit time. The following is the formula:

$$\text{Throughput (req/s)} = \frac{\text{Total Successful Requests}}{\text{Test Duration (s)}}$$

The request rate was measured in requests per second (req/s). The throughput metrics across all test scenarios were obtained as follows:

$$\textbf{TTL 30s:} = \frac{55.616}{300} = 185.38\ \text{req/s}$$

$$\textbf{TTL 60s:} = \frac{146.444}{300} = 487.95\ \text{req/s}$$

$$\textbf{TTL 120s:} = \frac{149.339}{300} = 497.68\ \text{req/s}$$

Throughput calculations reveal catastrophic performance degradation during DNS failover: TTL 30s achieves only 185.38 req/s (91.9% reduction from baseline 2,291.81 req/s), while TTL 60s and 120s attain 487.95 and 497.68 req/s respectively (78.7-78.3% degradation). Despite TTL 60s/120s processing 2.6× more requests than TTL 30s, this apparent superiority masks critical reliability trade-offs their higher throughput coincides with 83.48% and 62.78% failure rates, indicating severe system instability under failover conditions.

5) p99 Tail Latency Analysis: p99 latency represents the 99th percentile latency value, indicating maximum latency experienced by 99% of requests. The following is the formula:

$$\text{P99 Latency} = L_{0.99} \text{ where } P(L \le L_{0.99}) = 0.99$$

The p99 tail latency, representing the latency threshold below which 99% of requests were completed, was measured across all test scenarios as follows:

TTL 30s:
n = 55.616 requests (1)
k = [0.99 × 55,616] = 55.059 (2)
P99 = L(55,059) = 278.94 ms (3)

TTL 60s:
n = 146.444 requests (1)
k = [0.99 × 146,444] = 144.979 (2)
P99 = L(144,979) = 200.15 ms (3)

TTL 120s:
n = 149.339 requests (1)
k = [0.99 × 149,339] = 147.845 (2)
P99 = L(147,845) = 256.22 ms (3)

p99 latency represents the 99th percentile value from sorted latency samples. The calculation involves three steps: (1) sorting all latency values, (2) computing the index k = ⌈0.99 × n⌉ where n is the total number of samples, and (3) extracting P99 = L_k. Results reveal TTL 60s achieved best tail latency (200.15ms), TTL 120s intermediate (256.22ms), and TTL 30s worst (278.94ms), indicating longer TTL intervals reduce tail latency despite lower overall throughput during DNS failover.

Table IV presents the aggregate results from all Performance Under Failover Condition experiments that have been implemented, showing comprehensive system performance metrics obtained through systematic observation of system behavior under failover conditions across various testing scenarios.

TABLE IV
NETWORK QUALITY METRICS

Test Scenario	Avg Latency (ms)	Jitter (ms)	Packet Loss (%)	Throughput (req/s)	p99 Latency (ms)
Baseline	108.42	37.69	2.33	2291.81	231.15
30 s	110.26	49.29	16.0	185.38	278.94
60 s	101.20	26.78	4.0	487.95	200.15
120 s	105.85	37.76	20.0	497.68	256.22

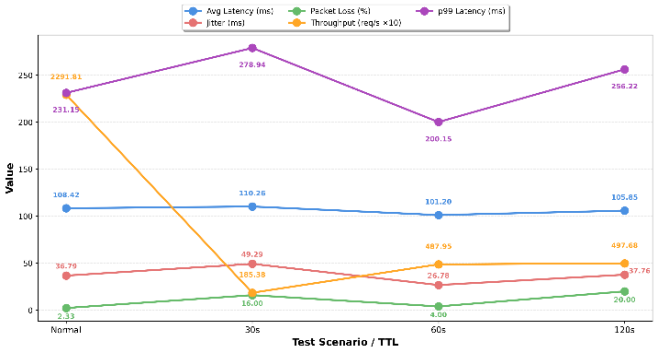

Figure 20. Network quality metrics results

Figure 20 Performance metrics comparison across DNS TTL configurations during simulated failover conditions. Metrics include average latency (blue), jitter (red), packet loss percentage (green), p99 latency (purple), and throughput (orange). Y-axis scaled to maximum observed value (p99 latency: 278.94 ms). Throughput values: Baseline=2291.81, TTL 30s=185.38, TTL 60s=487.95, TTL 120s=497.68 req/s. Data aggregated from wrk benchmarking (n=4 threads, 250 concurrent connections, 300s duration) against multi-cloud infrastructure endpoint, revealing DNS cache interval impact on system performance.

C. *Failover Reliability Analysis*

This section evaluates system availability and recovery characteristics during simulated infrastructure failures. The analysis encompasses critical reliability metrics, including recovery time, system availability, and failure detection latency. Each metric is mathematically defined and systematically measured across different TTL configurations to quantify the impact of DNS caching parameters on system resilience under failover conditions. Table V presents the experimental results from each session and shows that shorter TTLs improve failover recovery and reduce downtime but increase DNS overhead, quantifying key trade-offs in DNS-based failover reliability.

TABLE V
FAILOVER RELIABILITY METRICS

DNS TTL (S)	Downtime	Revovery Time (s)	Failed Requests (%)
30	152.65	30.53	48.62
60	243.92	243.92	83.48
120	186.88	186.88	62.78

Overall, this analysis evaluates three key metrics total downtime, average recovery time, and failed request percentage to compare the effectiveness of TTL 30s, 60s, and 120s in multi-cloud DNS failover scenarios.

1) Total Downtime Analysis: Total downtime quantifies cumulative service unavailability duration during the monitoring window. The following is the formula:

$$\text{Total Downtime (s)} = \sum_{i=1}^{k} (t_{\text{end},i} - t_{\text{start},i})$$

The downtime metrics across all test scenarios were obtained as follows:

TTL 30s:
$$\text{Total Downtime} = (t_{end,1} - t_{start,1}) + (t_{end,2} - t_{start,2}) + \ldots + (t_{end,6} - t_{start,6}) \quad (1)$$
$$= 128.91 + 3.16 + 8.01 + 8.03 + 4.54 + 0.00 \quad (2)$$
$$= 152.65 \text{ s} \quad (3)$$

TTL 60s:
$$\text{Total Downtime} = (t_{end,1} - t_{start,1}) + (t_{end,2} - t_{start,2}) \quad (1)$$
$$= 3.08 + 240.85 \quad (2)$$
$$= 243.92 \text{ s} \quad (3)$$

TTL 120s:
$$\text{Total Downtime} = (t_{end,1} - t_{start,1}) + (t_{end,2} - t_{start,2}) \quad (1)$$
$$= 8.06 + 178.82 \quad (2)$$
$$= 186.88 \text{ s} \quad (3)$$

Based on calculations, TTL 30s achieves lowest downtime 152.65s, 5 recovery cycles, 51.38% success. TTL 60s performs worst 1 cycle, 243.92s downtime, 16.52% success due to Route53 90s health check timing mismatch causing destructive race conditions. TTL 120s moderate 1 cycle, 186.88s, 37.22% success with predictable behavior. For low-budget SMEs using cost-effective infrastructure (AWS t3.micro, GCP e2-medium, Herza 1vCPU), TTL 30s recommended for high availability. Avoid TTL 60s.

2) Recovery Time Analysis: Average recovery time measures mean duration of individual downtime-to-recovery cycles. The following is the formula:

$$\text{Avg Recovery Time (s)} = \frac{\text{Total Downtime (s)}}{\text{Number of Recovery Events}}$$

Multi-cloud failover system testing demonstrates the significant during primary server failures. The recovery time metrics across all test scenarios were obtained as follows:

$$\textbf{TTL 30s:}\quad \text{Avg Recovery Time} = \frac{152.65}{5} = 30.53\ \text{s}$$

$$\textbf{TTL 60s:}\quad \text{Avg Recovery Time} = \frac{243.92}{1} = 243.92\ \text{s}$$

$$\textbf{TTL 120s:}\quad \text{Avg Recovery Time} = \frac{186.88}{1} = 186.88\ \text{s}$$

Average recovery time calculation involves dividing total recovery time by the number of trials. For TTL 30s, total time of 152.65 seconds across 5 trials yields an average of 30.53 seconds, demonstrating stable failover consistency. Meanwhile, TTL 60s with 1 trial recorded 243.92 seconds, and TTL 120s recorded 186.88 seconds in 1 trial. Lower TTL enables faster DNS propagation, allowing clients to switch to backup servers with minimal delay.

3) Failed Requests Analysis: Failed request percentage quantifies proportion of unsuccessful availability checks. The following is the formula:

$$\text{Failed Requests}\ (\%) = \frac{\text{Failed Requests}}{\text{Total Requests}} \times 100$$

The failed requests metrics across all test scenarios were obtained as follows:

$$\textbf{TTL 30s:}\quad \text{Failed Requests}\ (\%) = \frac{106}{218} \times 100 = 48.62\ \%$$

$$\textbf{TTL 60s:}\quad \text{Failed Requests}\ (\%) = \frac{192}{230} \times 100 = 83.48\ \%$$

$$\textbf{TTL 120s:}\quad \text{Failed Requests}\ (\%) = \frac{140}{223} \times 100 = 62.78\ \%$$

Based on the calculation results TTL 30s shows lowest failure rate (48.62%) and downtime (152.65s across 5 cycles), outperforming TTL 60s (83.48% failures, 243.92s single-cycle outage) and TTL 120s (62.78% failures, 186.88s). Lower TTL enables faster DNS propagation via frequent cache invalidation. TTL 60s fails due to resonance with Route53's 90s health check threshold. Empirical U-shaped distribution confirms timing-dependent cache-health check interference.

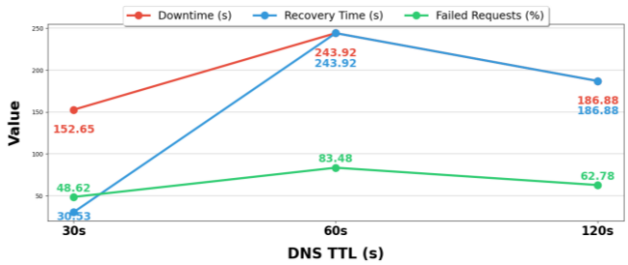

Figure 21. Failover reliability results

D. Research Limitations and Discussion

This study has several limitations that should be considered when interpreting results and practical applications.

1) Limited Scale and Infrastructure: The multi-cloud failover system implementation used infrastructure with limited capacity, specifically AWS t3.micro instances (2 vCPU, 1 GB RAM) and Google Cloud e2-medium (2 vCPU, 4 GB RAM). This configuration simulates small to medium enterprises with low-budget capabilities. Enterprises with high traffic may experience significantly different performance characteristics.

2) Testing Load Constraints: Performance testing used wrk with 4 threads and 250 concurrent connections over 300 seconds. While representative for small-medium applications, these parameters do not reflect enterprise real-world traffic loads of thousands to millions of requests per second. Network quality metrics such as average latency of 108.42 ms and p99 latency of 231.15 ms under normal operation may not generalize to high-traffic production environments.

3) Geographic and Cloud Vendor Scope: Experiments were limited to three cloud providers (AWS, Google Cloud, Herza Cloud) focused on Southeast Asia, particularly Jakarta region. Vendor and region selection aligned with Indonesian data residency requirements and cost affordability for budget-constrained local organizations. Failover reliability results such as 152.65s downtime for TTL 30s and 243.92s for TTL 60s may differ substantially across other cloud providers or global multi-region deployments with higher network latency.

4) Practical Implications for Low-Budget SMEs: The implemented architecture uses lowest-tier cloud services Google Cloud (e2-medium, 2 vCPU, 4 GB RAM), AWS (t3.micro, 2 vCPU, 1 GB RAM), and Herza Cloud (1 vCPU, 1 GB RAM) to minimize monthly operational costs. This infrastructure sizing is empirically justified, whose regression analysis of 400 SMEs demonstrated that training and support programs increased effective technology leverage by 1.75 times ($p=0.0003$), while cloud integration with existing systems doubled efficiency gains (OR=2.0, $p<0.001$) [1]. Our \$45–60/month cost structure aligns with [3] findings that Indonesian SMEs prioritize pay-per-use models to avoid capital expenditure in hardware and infrastructure, research and [2] PEST analysis confirming that limited financial resources make cost factors more relevant to SMEs than strategic competitiveness . Compared to [1] reported 30% cost reduction and 20% security improvement through cloud adoption, our TTL 30s configuration achieving 152.65s

downtime represents a practical implementation of high availability within comparable budget constraints.

5) Failover Observation Duration: Failover monitoring spanned 5-minute windows with 1-second polling intervals. While sufficient to capture DNS TTL propagation cycles (30-120 seconds), this duration does not evaluate long-term system stability, daily traffic pattern behaviors, or corner cases like simultaneous multi-availability zone cascading failures.

IV. CONCLUSION

This research successfully implemented and evaluated a DNS-based multi-cloud failover system using AWS Route53, Nginx reverse proxy on Google Cloud (primary) and Herza Cloud (backup), with shared AWS EC2 backends orchestrated from an Ubuntu control node. Performance testing via wrk (4 threads, 250 connections, 300s) and Python monitoring scripts across baseline and failover scenarios (TTL 30s, 60s, 120s) showed baseline throughput of 2,291.81 req/s, average latency 108.42 ms, and p99 latency 231.15 ms.

Under failover conditions, TTL 30s performed best with 152.65s downtime, 30.53s average recovery time, and 48.62% failures, despite 91.9% throughput drop to 185.38 req/s and 49.29 ms jitter from high DNS overhead. TTL 60s was worst (243.92s downtime, 83.48% failures) due to health check (90s) interference, while TTL 120s was intermediate (186.88s, 62.78%). TTL 30s configuration is recommended for low-budget SMEs ($45–60/month), balancing high availability with operational costs.

Limitations include small-scale instances (t3.micro/e2-medium), 250-connection loads, Jakarta region focus, and 300s observation, limiting generalization to high-traffic or multi-region setups. Future research should explore enterprise-scale testing, ML-based load prediction, and hybrid anycast failover for further optimization.

REFERENCES

[1] H. Kalinaki, T. Joshua, and A. University, “Cloud Computing and Operational Efficiency: A Case Study of SMEs in Kampala,” vol. 3, pp. 711–723, Oct. 2024.

[2] M. A. Javaid, “Implementation of Cloud Computing for SMEs,” *World J. Comput. Appl. Technol. Publ.*, vol. 2, no. 3, pp. 66–72, Mar. 2014, doi: 10.13189/wjcat.2014.020302.

[3] D. R. Rahadian, V. Y. Mahendra, R. D. Yuliyanto, and M. A. Sholihin, “Manajemen Resiko Cloud Computing Pada UMKM,” *Pros. Semin. Nas. Teknol. Inf. Dan Bisnis*, pp. 135–141, 2023.

[4] A. F. Kasmar, W. Wahyuna, F. Sukma, and S. Amalia, “Implementasi sistem keamanan dan high availability pada cloud server menggunakan Amazon Web Services (AWS),” *J. Teknoif Tek. Inform. Inst. Teknol. Padang*, vol. 13, no. 1, pp. 40–47, Apr. 2025, doi: 10.21063/jtif.2025.V13.1.40-47.

[5] A. A. Rotib, “Pusat Data Dan Layanan Cloud Center : Jaringan Protokol Dan Manajemen,” vol. 1, no. 1, 2024.

[6] Muhajirin, “Optimalisasi Web Server Menggunakan System Failover Clustering Berbasis Cloud Computing,” *J. Ilm. Ilmu Komput. Fak. Ilmu Komput. Univ. Al Asyariah Mandar*, vol. 3, no. 2, pp. 35–42, 2017, doi: 10.35329/jiik.v3i2.58.

[7] H. Y. Prabowo, A. R. Mukti, Suryayusra, and T. Ariyadi, “Analisa Desain High Availability dan Uji Reabilitas Cloud Storage,” *J. Indones. Manaj. Inform. Dan Komun.*, vol. 5, no. 1, pp. 262–270, Jan. 2024, doi: 10.35870/jimik.v5i1.467.

[8] S. Sumarna, H. Nurdin, and F. W. Handono, “Perancangan N-Clustering High Availability Web Server Dengan Load Balancing Dan Failover,” *JITK J. Ilmu Pengetah. Dan Teknol. Komput.*, vol. 4, no. 2, pp. 149–154, Feb. 2019, doi: 10.33480/jitk.v4i2.287.

[9] A. Fadila, M. Nasir, and S. Safriadi, “Implementasi Sistem Load Balancing Web Server Pada Jaringan public Cloud Computing Menggunakan Least Connection,” *J. Artif. Intell. Softw. Eng.*, vol. 3, no. 2, pp. 50–55, Oct. 2023, doi: 10.30811/jaise.v3i2.4578.

[10] Prinafsika, A. Junaidi, and M. Muharrom Al Haromainy, “Cloud-Based High Availability Architecture Using Least Connection Load Balancer and Integrated Alert System,” *Bit-Tech*, vol. 8, no. 1, pp. 263–274, Aug. 2025, doi: 10.32877/bt.v8i1.2520.

[11] D. Siregar, A. Ariangga, S. Sarudin, H. Harahap, and R. Liza, “Load Balancing untuk Lalu Lintas Tinggi pada Lingkungan Cloud Menggunakan Metode Round Robin,” *J. Inform. Univ. Pamulang*, vol. 9, no. 2, pp. 38–45, Jul. 2024, doi: 10.32493/informatika.v9i2.42662.

[12] Fauzan Prasetyo Eka Putra, Noviyani Dwi Saputri, Fathur Rosi, and Rohilia Loati, “Optimalisasi Infrastruktur Cloud Networking melalui Integrasi SDN, NFV, dan Multi-Cloud,” *J. Inform. Dan Tekonologi Komput. JITEK*, vol. 5, no. 1, pp. 118–125, Mar. 2025, doi: 10.55606/jitek.v5i1.6099.

[13] Y. Afek and A. Litmanovich, “Decoupling DNS Update Timing from TTL Values,” Sep. 16, 2024, *arXiv*: arXiv:2409.10207. doi: 10.48550/arXiv.2409.10207.

[14] M. F. Darmawan and S. Risnanto, “Implementasi Failover Gateway Recursive Dan Load Balancing Menggunakan Metode Per Connection Classifier,” *Infotronik J. Teknol. Inf. Dan Elektron.*, vol. 8, no. 2, pp. 56–66, Dec. 2023, doi: 10.32897/infotronik.2023.8.2.1887.

[15] N. M. K. Koneru, “Disaster Recovery In The Cloud: Implementing Dr Sites And Blue/Green Deployments In Aws,” *Int. J. Appl. Math.*, vol. 38, no. 10s, pp. 2441–2461, Nov. 2025, doi: 10.12732/ijam.v38i10s.1135.

[16] I. P. A. E. Pratama, P. V. Andreyana, and P. R. Nurjana, “Pengujian High Availability pada Asynchronous DNS Berbasis Restknot menggunakan Algoritma Round Robin,” *J. Indones. Manaj. Inform. Dan Komun.*, vol. 5, no. 1, pp. 1019–1032, Jan. 2024, doi: 10.35870/jimik.v5i1.582.

[17] M. P. Hapsari, A. B. Prasetijo, and A. Fauzi, “Analisa Kinerja pada Standalone Server dan Clustering Server Teknologi RAC (Real Application Clustering) dengan Algoritma DNS (Domain Name System) Round Robin Berbasis Oracle Linux 6.4 di Lingkungan Virtual,” *J. Sist. Komput.*, vol. 10, no. 2, 2020.

[18] L. Izhikevich *et al.*, “ZDNS: A Fast DNS Toolkit for Internet Measurement,” in *Proceedings of the 22nd ACM Internet*

Measurement Conference, Oct. 2022, pp. 33–43. doi: 10.1145/3517745.3561434.

[19] R. Annisa, A. R. Makarim, M. Afif, W. E. Sulistiono, and S. Ferbangkara, "Analisis Kinerja Layanan Cloud Computing dalam Sistem Cerdas Rekomendasi Tanaman Perkebunan," *SINTA*, vol. 7, Jun. 2025.

[20] W. Wicoksono, H. A. Mustaqhim, P. P. Anwas, and L. N. L. Badratul, "Performance Comparison of NGINX, Apache, and Lighttpd Using WRK on a Debian," *Bit-Tech*, vol. 8, no. 1, pp. 670–680, Aug. 2025, doi: 10.32877/bt.v8i1.2661.

[21] A. A. Nizar, S. A. Karimah, and E. M. Jadied, "Analysis of Virtualization Performance on Resource Efficiency Using Containers and Unikernel," in *2024 International Conference on Artificial Intelligence, Blockchain, Cloud Computing, and Data Analytics (ICoABCD)*, Aug. 2024, pp. 125–130. doi: 10.1109/ICoABCD63526.2024.10704518.

[22] M. Willetts and A. S. Atkins, "Performance measurement to evaluate the implementation of big data analytics to SMEs using benchmarking and the balanced scorecard approach," *J. Data Inf. Manag.*, vol. 5, no. 1, pp. 55–69, Jun. 2023, doi: 10.1007/s42488-023-00088-8.