Designing and Evaluating a Resilience Testing Strategy for Cloud-Native DNS Systems

Georgios Daskalopoulos[1,2], Thomas Grübl[2], Burkhard Stiller[2]

[1]Swisscom AG
[2]Communication Systems Group CSG, Department of Informatics IfI, University of Zürich UZH
Binzmühlestrasse 14, CH—8050 Zürich, Switzerland
[1]georgios.daskalopoulos@swisscom.com [2]{gruebl,stiller}@ifi.uzh.ch

***Abstract*—Microservice applications rely heavily on DNS for internal service discovery in ephemeral environments. Unlike traditional public DNS, cloud-native DNS operates as a core infrastructure component with strict requirements for correctness, low latency, and adaptation to frequent record changes. There is a gap between existing DNS research focused mainly on performance in public environments and the unexplored resilience of internal, cloud-native DNS systems, especially those supporting private zones in containerized environments.**

This paper addresses this gap by examining the role of private DNS in cloud-native systems and applying resilience testing techniques to systematically evaluate its behavior. First, this work analyzes public cloud incident post-mortems and finds that DNS is rarely the primary root cause of outages, and incidents disproportionately affect private DNS over public DNS. Secondly, we design and evaluate a resilience testing strategy for a multi-cluster cloud-native DNS system published by Swisscom AG, the biggest telecommunications provider in Switzerland. The proposed strategy uses chaos injection and DNS traffic generation to evaluate availability, latency, accuracy, and recovery behavior under different disruption scenarios. We find that the system operates within acceptable limits in most scenarios and resilience testing can reliably capture the impact of caching, resolution inconsistencies, and failure-handling weaknesses across test cases.

***Index Terms*—Cloud-Native DNS, Private DNS, Resilience Testing, 5G.**

I. Introduction

The Domain Name System (DNS) has evolved from a static naming system to a commonly used service discovery mechanism [1]. Unlike traditional public DNS, cloud-native DNS is designed to support frequent, automated record updates and low-latency resolution within orchestration platforms, making it a core infrastructure component and a critical dependency for containerized workloads. Failures or performance degradation in DNS can directly impact service availability and system stability, even when application instances remain healthy [2].

This paper examines the role and behavior of DNS services in cloud-native environments, with a focus on private DNS used for internal service discovery. The paper then discusses the role of DNS in containerized platforms such as Kubernetes and introduces the term *cloud-native DNS* to describe DNS systems responsible for internal name resolution and record provisioning in dynamic, container-based environments.

While existing research provides valuable insights into performance, caching, and propagation behavior, largely derived from public DNS environments, it offers limited insights into how internal, cloud-native DNS systems behave under partial failures, dependency outages, or degraded conditions. As a result, resilience related questions remain largely unexplored, particularly for DNS services supporting private zones in containerized environments.

To provide an empirical basis for the study, we first analyze publicly available post-mortems from cloud provider incidents involving DNS. The dataset is analyzed with regards to whether DNS is the root cause of an incident or not, the affected DNS functionality (name resolution or record provisioning), and the scope of the affected zones (public or private). Subsequently, we introduce a DNS system published by Swisscom, the biggest Swiss telecommunication provider, for multi-cluster internal name resolution in 5G Core environments and examine its main components, resolution paths, record synchronization mechanisms, and fallback behavior. On this basis, we propose a resilience testing strategy, including the scope of injected faults, a set of test cases targeting individual components, multiple steady-state hypotheses describing expected system behavior, and a set of metrics used to evaluate resilience.

Finally, we describe the evaluation of a DNS traffic generation and validation approach, consisting of a custom client generating DNS traffic and DNS record updates, and a monitoring setup based on Prometheus and Grafana. We use a reproducible chaos testing toolchain to execute the predefined test cases.

In this work, we make the following **contributions**:

- We present a statistical analysis of publicly available DNS incident post-mortems, which quantifies the prevalence and roles of DNS in real-world outages.
- We identify a clear research gap, namely the lack of systematic research on the resilience of private DNS

ISBN 978-3-903176-82-9 © 2026 IFIP

infrastructures in cloud environments, despite evidence from public post-mortems that private DNS is disproportionately impacted in real-world incidents and plays an important role in service availability.
- We design a resilience testing strategy of a multi-cluster DNS solution for private name resolution.
- We evaluate the resilience testing strategy in a replicated environment using a cloud-native DNS solution published by Swisscom AG, the biggest Swiss telecommunication provider, and find that the solution behaves as expected and within acceptable operational limits across all our test cases.

II. Background and Related Work

While the term *cloud-native DNS* is (partially) adopted in industry, it can easily be confused with *cloud DNS*, which typically refers to managed DNS products provided by hyperscalers. Such products are mainly intended for publishing public-facing services to the global DNS and are not the focus of this work. Notable private DNS solutions are listed below.

kube-dns [3] was the original DNS service for Kubernetes and is still available and used in some environments for cluster-internal DNS resolution.

CoreDNS [4] is currently the default DNS server for Kubernetes. It is highly extensible through a plugin-based architecture and supports private DNS zones. CoreDNS can operate both as a recursive resolver (usually for global DNS) and as an authoritative DNS server (usually for the cluster internal resolutions), making it a central component of Kubernetes service discovery.

PowerDNS either as an authoritative server [5] or as a recursive resolver [6], is considered more of a traditional DNS solution which is usually deployed in standalone servers. However, it can be repurposed as a component in a cloud-native DNS architecture acting as an internal authoritative DNS service

External DNS [7], is a useful utility which detects workloads in containerized environments and publishes their domain names to authoritative DNS Servers. Depending on the nature of the workloads and the role of the target DNS servers, ExternalDNS can be used for either private or public DNS resolution.

Typical Kubernetes setups include a DNS Resolver Service exposed at a static IP, backed by CoreDNS, which is a general-purpose authoritative DNS. Each container receives responses originating from this authoritative server [8].

A. DNS Testing in Scientific Literature

The work by Liu et al. [9] is the only scientific publication proposing a DNS design specifically built for internal name resolution within Kubernetes and related container based environments. The authors examine how standard Kubernetes DNS components behave when responsible for resolving service names for containerized workloads, and they show the limitations of the existing solutions at the time. Their analysis points to issues such as latency instability, sensitivity to frequent record updates, and the operational effects of runtime behavior. Although the study is conducted in a large production deployment, the underlying challenges they highlight stem from the dynamic nature of internal service discovery rather than from scale alone. As a solution to those limitations, they propose *ContainerDNS*, an architecture designed to offer predictable behavior in systems where DNS records change frequently because workloads appear or disappear quickly.

Erdenebat et al. [10] shows that similar weaknesses appear even at smaller scales when DNS configurations and service discovery patterns are not aligned with Kubernetes' internal mechanisms. Their case study highlights how default CoreDNS behavior, inefficient search domains, and heavy reliance on external services can lead to latency spikes, high error rates, and excessive memory consumption. This suggests that internal DNS issues are not exclusive to very large clusters, but could arise whenever the resolution flow becomes complex or misconfigured. To tackle this problem, Erdenebat et al. [10] propose a practical set of improvements including redesigned service objects, node DNS caching, refined `resolv.conf` search paths, and more appropriate scaling for CoreDNS. These changes reduce response times from seconds to milliseconds and stabilize CoreDNS behavior under load. Both studies focused on the performance aspect of cloud-native DNS, however, to the best of our knowledge, no existing work proposes and evaluates a resilience testing solution in cloud-native private DNS setups.

TABLE I: Sources of publicly available post-mortems reports used for the analysis of DNS incidents.

Reference	Source	Count
[11]	Google Cloud DNS Incidents	9
[12]	Google Cloud Incidents	11
[13]	AWS Post Event Summaries	7
[14]	AWS Route 53 Incidents	3
[15]	Azure Incidents	5
[16]	OSS Post-Mortem List	5
Total		40

III. Quantifying DNS Incidents From Post-Mortems

To further motivate this research, we analyzed the role of DNS in real-world failures through a systematic study of publicly available post-mortems from major cloud providers. This analysis quantifies how frequently DNS is involved in incidents, distinguishes between root causes and secondary effects, and examines which DNS functionalities and zone scopes are most commonly impacted, to identify recurring failure patterns and dependencies.

A. Incident Sourcing

Incident reports were sourced from the public incident reporting portals of major cloud providers, Google Cloud, Amazon Web Services (AWS), and from an open-source repository aggregating post-mortems across organizations. Relevant incidents were identified using a combination of product-based filtering and keyword-based search. An overview of the

TABLE II: Aggregated results from incident assessment.

Metric	Yes %	No %
Affected DNS Infrastructure	65	35
DNS as Root Cause	30	70
Record Provisioning Affected	47.5	52.5
DNS Resolution Affected	62.5	37.5
Private DNS	80	20
Public DNS	42.5	57.5

TABLE III: Distribution of incidents by affected DNS scope.

Incidents affecting ...	Observations	Percentage (%)
Private DNS	32	80
Public DNS	17	42.5
Private DNS only	23	57.5
Public DNS only	7	17.5
Both Private and Public DNS	10	25

TABLE IV: Comparison of affected DNS infrastructure and DNS as the root cause.

Case	Observations	Percentage (%)
Only DNS Infrastructure affected	23	57.5
Only DNS as Root Cause	9	22.5
Incidents where both are true	3	7.5
Incidents where neither is true	5	12.5

TABLE V: Comparison of incidents affecting record provisioning and DNS resolution.

Case	Observations	Percentage (%)
Only Record Provisioning affected	15	37.5
Only DNS Resolution affected	21	52.5
Incidents where both are affected	4	10
Incidents where neither is affected	0	0

sources examined and the number of incidents identified is shown in Table I.

B. Incident Assessing

After collecting the relevant incidents, each one was analyzed to understand the scope and impact of DNS within the incident. The assessment focused on identifying whether DNS components were directly affected, whether DNS was the root cause or an aftereffect, and what type of DNS functionality or zone scope (public or private) was impacted. The following attributes were defined and evaluated for every incident: *(i)* the affected DNS infrastructure, *(ii)* whether DNS was the root cause, *(iii)* affected record provisioning, *(iv)* affected DNS resolution, *(v)* private DNS, and *(vi)* public DNS.

The pairs of attributes *(i) & (ii)*, *(iii) & (iv)*, and *(v) & (vi)* may appear mutually exclusive, but this is not the case. Depending on the nature of the incident, both attributes in a pair can be true simultaneously, or none of them might be applicable. For example, both public and private DNS zones could be impacted in a large-scale incident.

C. Statistical Analysis of Results

The incident summary presented in Table II reveals a handful of trends. DNS was identified as the primary root cause in 30% of the incidents, indicating that while DNS failures have severe impact, they are usually secondary effects of broader system outages. This is also manifested from the 65% of the cases where DNS infrastructure was impacted during the incident. DNS resolution issues were observed more frequently than record provisioning failures. Finally, private DNS was affected in the majority of cases.

1) Incident Impact by DNS Zone Scope: Table III summarizes the distribution of incidents by affected DNS zone scope. While private DNS is impacted in the majority of incidents (80%), a significant portion of incidents (42.5%) also affect public DNS resolution, indicating that public DNS is not isolated from operational issues. Incidents affecting only public DNS are less frequent, whereas the overlap between private and public DNS failures shows that public DNS is not immune to disruption and may experience cascading effects originating from internal failures.

2) DNS Infrastructure Impact and DNS Root Cause Classification: As shown in Table IV, for the majority of cases, DNS functionality was affected as a result of other problems. This shows how dependent DNS is on the surrounding infrastructure and underlines the need for more resilient DNS service designs. Incidents where DNS itself was identified as the root cause are less frequent but still significant, highlighting the importance of proper testing and validation within DNS components. Cases where both the DNS infrastructure was affected and DNS was the root cause were rare, but it seems that DNS failures can occur concurrently with broader problems, potentially increasing the overall impact of an incident.

Finally, incidents where neither DNS infrastructure was affected nor DNS was the root cause were uncommon. These cases usually came from misconfigured DNS clients or wrong DNS records, rather than faults in the DNS system itself.

3) Incident Impact by DNS Functionality: Finally, as shown in Table V, DNS resolution was affected more often than record provisioning. This suggests that generic failures during name resolution are more likely to appear. Record provisioning issues were less common but still important, as they can stop new or updated records from being deployed, leading to stale DNS data, and DNS resolution failures for that particular records. The few cases where provisioning and resolution were both affected point to wider problems such as general unavailability of services and hardware failures because of external reasons (one example was a fire in the datacenter [17]).

D. Correlations Analysis

Next, we conducted a correlation analysis to identify how strongly different attributes in the dataset move together. It reveals which DNS-related factors tend to appear or exclude each other during incidents. Values more close to -1 or 1 indicate a stronger relationship, while values close to 0 suggest weak or no correlation. Figure 1 presents the pairwise correlations between the analyzed attributes.

a) Record Provisioning and DNS Resolution Affected: A strong inverse correlation of -0.81 is observed between record provisioning issues and DNS resolution failures. This

suggests that incidents affecting name resolution typically do not impact record provisioning, and vice versa.

b) Affected DNS Infrastructure and DNS as Root Cause: The correlation between affected DNS infrastructure and DNS being the root cause is moderately negative, with a value of -0.55. For example, when DNS infrastructure is affected, DNS is often not identified as the primary cause of the incident, but rather impacted as a consequence of other failures such as network disruptions or broader service unavailability.

c) Private DNS and Public DNS: The correlation between private and public DNS impact is weaker, with a value of -0.46. This suggests that incidents usually affect either private or public DNS zones, but not both.

d) DNS as Root Cause and DNS Zone Scope: An inverse correlation of -0.49 is observed between DNS being the root cause and the involvement of private DNS zones. This suggests that private DNS issues occur less frequently when DNS itself is the primary cause of an incident. In contrast, the correlation between DNS as the root cause and public DNS impact is close to zero (0.10), indicating no clear relationship. This behavior is further examined in the following section through conditional analysis.

E. Conditional Analysis

Conditional analysis explores if the likelihood of one event changes depending on another condition in the dataset. In this context, conditional probabilities are compared against the *a priori* probabilities observed across the entire dataset. Here, *a priori* refers to the baseline probability of an attribute occurring without conditioning on any specific factor. This type of analysis helps reveal which parts of the DNS system are more likely to fail under specific circumstances for example, when DNS is the root cause, when infrastructure is affected, or when private or public zones are involved.

1) DNS as Root Cause and Affected Functionality: Figure 2a illustrates the conditional probability of record provisioning and DNS resolution being affected, given whether DNS is identified as the root cause of an incident or not. When DNS is not the root cause, record provisioning and resolution are affected at similar rates, around 46% and 53% respectively. When DNS is the root cause, resolution is affected in 67% of incidents, while record provisioning drops to 33%. This shows that direct DNS failures mainly disrupt name resolution, whereas provisioning issues are more likely to result from secondary effects when DNS itself is not failing.

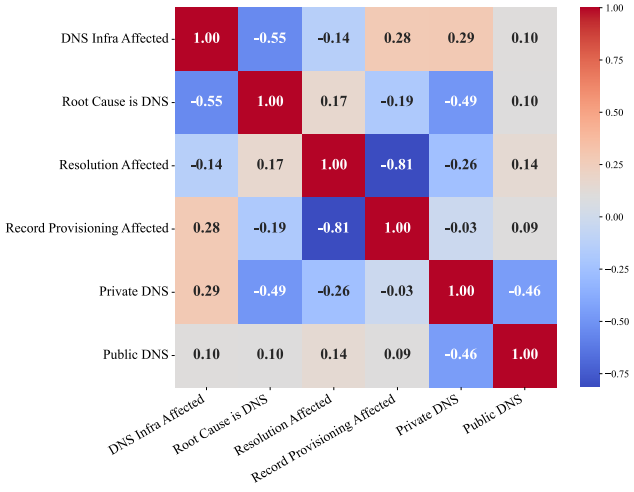


Fig. 1: Correlations between DNS incident attributes.

2) DNS as Root Cause vs Affected Zones: Figure 2b shows the conditional distribution of affected DNS zones based on whether DNS is the root cause. When DNS is not the root cause, private zones are almost always affected (92%).

When DNS is the root cause, private zone issues drop to 50% and public zone impact increases to 50%, showing that both zones can be impacted.

3) Impacted Functionality vs Affected Zones: Figure 2c shows the conditional probability of DNS zone impact given the affected DNS functionality. The results show that the affected functionality does not substantially alter differ across DNS types. When record provisioning is impacted, private DNS is affected in 79% of cases and public DNS in 47%. When resolution is impacted, private DNS remains dominant at 72% and public DNS involvement remains at 48%.

F. Takeaways

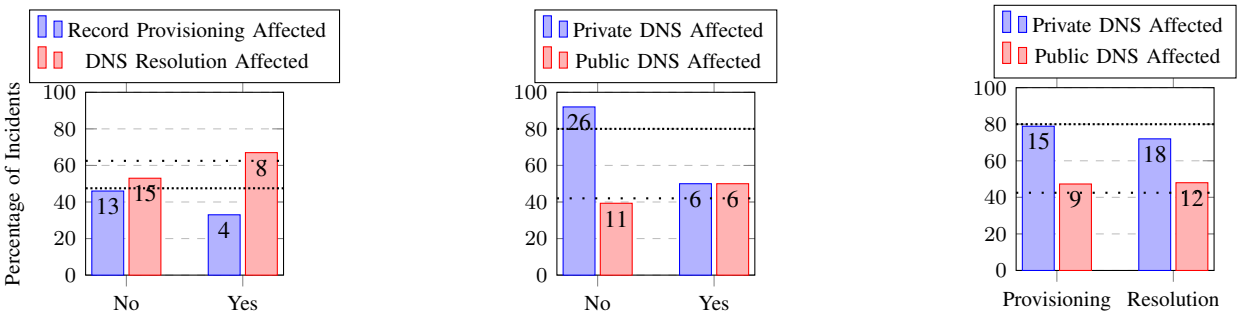
- DNS is typically not the root cause in incidents, and is involved in roughly one third of DNS-related cases. In the majority of cases other causes affect the DNS infrastructure.
- In incidents regarding cloud providers, private DNS is impacted in 80% of the cases, and public DNS is impacted in 42.5% of the cases. Both are impacted in 25% of the cases.
- DNS resolution is affected more often than record provisioning, while both functions are affected 10% of the time. This is also manifested in the correlation coefficient of -0.81.
- When DNS is not the root cause of an incident, record provisioning and resolution are similarly affected, unlike when DNS is the root cause.

IV. Resilience Testing Strategy Design

This section describes the high level testing strategy used to evaluate the resilience of a multi-cluster DNS solution for private name resolution. We provide details on the system architecture, delegation behavior, and caching settings. We then define the fault scenarios considered, introduce the steady-state hypotheses, expected behavior under disruption, and the metrics used to assess correctness, availability, and recovery.

A. Overview

The testing strategy presented in this section is based on a DNS solution developed and published by Swisscom AG [18]. The setup provides internal DNS resolution and service discovery across multiple Kubernetes clusters. The system is designed to authoritatively serve DNS records under



(a) Root Cause Per Affected Functionality. (b) Root Cause per DNS Type. (c) Impacted Functionality per DNS Type.

Fig. 2: Conditional analysis of DNS root causes and impacted functionality. The densely-dotted horizontal line (·····) in subfigure (a) represents the apriori provisioning fault rate of 47.5% and the loosely-dotted horizontal line (· · ·) represents the apriori resolution fault rate of 62.5%. In subfigures (b) and (c), the densely-dotted lines represent the overall private DNS failure rate of 80% and the loosely-dotted lines the overall public DNS failures of 42.5%.

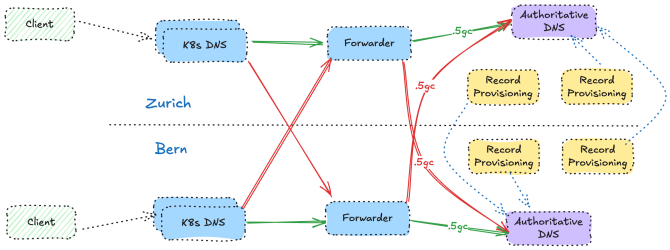


Fig. 3: Testbed setup with two Kubernetes clusters.

the *5gc.3gppnetwork.org* domain within the clusters, while forwarding public DNS queries to an upstream resolver. It continuously observes the state of each cluster and synchronizes the authoritative DNS servers accordingly, aiming to achieve eventual consistency across clusters over the DNS records that each authoritative server holds. The architecture includes fallback routing mechanisms between its components, allowing the system to tolerate a limited degree of disruption while maintaining service availability. This setup is extensible to be used in multiple clusters, but for the scope of our work a two cluster deployment is evaluated, illustrated in Figure 3.

B. Components

a) Kubernetes DNS (K8's DNS): Kubernetes DNS is a native service deployed in every cluster and based on the CoreDNS [2] project. It enables service discovery for workloads internal to the cluster. K8's DNS acts as an authoritative server for the *.cluster.local* domain and as a stub resolver for all other domains. In the evaluated setup, two replicas are deployed per cluster. For the *5gc.3gppnetwork.org* domain, K8's DNS is explicitly configured with forwarding rules that delegate resolution to the DNS Forwarder component.

b) DNS Forwarder: The DNS Forwarder is an additional component introduced by this design and is not provided by out-of-the-box Kubernetes. It is implemented using CoreDNS and applies specialized forwarding logic for the *5gc.3gppnetwork.org* domain, directing queries to authoritative DNS servers within the cluster or, if necessary, to fallback clusters. Each cluster runs a single instance of the DNS Forwarder.

c) Authoritative DNS Server: The Authoritative Server is responsible for serving DNS records for the private zone *5gc.3gppnetwork.org*. Each cluster hosts one authoritative instance, implemented using PowerDNS [5]. DNS records are persisted in a backend database to ensure durability.

d) Record Provisioning Service: DNS record creation and deletion are handled by the Record Provisioning Service, implemented using ExternalDNS [7]. This service monitors the Kubernetes control plane and reacts to changes in selected resources. When a matching resource is created, updated, or deleted, ExternalDNS synchronizes the corresponding DNS record with the authoritative servers. Each cluster runs multiple ExternalDNS instances, one for each authoritative server across all clusters. For example, Cluster A runs one instance targeting its local authoritative server and one additional instance targeting the authoritative server in Cluster B as illustrated in Figure 3.

C. Delegations and Fallback Routes

The delegation rules relevant to the private *5gc.3gppnetwork.org.* zone, which is the focus of the testing setup, are illustrated in Figure 4. The DNS components are configured to serve three distinct resolution scopes and to support fallback routes. A fallback route is only invoked if the primary delegation or a preceding fallback fails.

D. Caching Settings

K8's DNS and Forwarder DNS support two different caching configurations. The **No-Caching** setting disables caching in K8's DNS and forwarder DNS elements for the *5gc.3gppnetwork.org.* domain. Every request on the respective element is forwarded to the next element in the resolution chain. The **Standard-Caching** setting caches successful and NXDOMAIN responses of the *5gc.3gppnetwork.org.* domain for a duration (*i.e.,* time-to-live) of 5 seconds. The cache sizes are kept to the default number of 9,984 entries. The no-caching

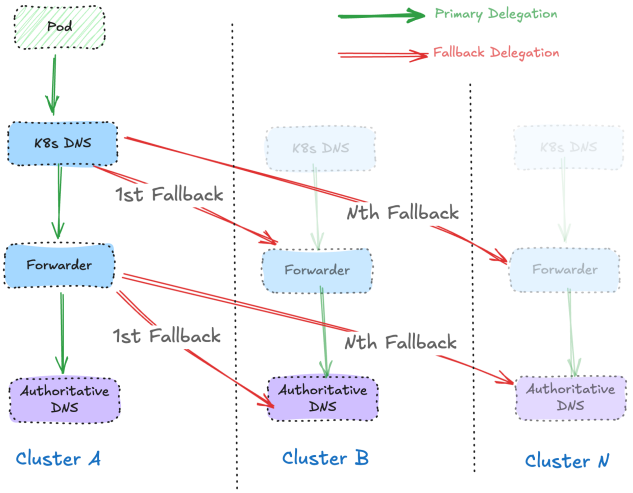


Fig. 4: 5GC Delegation Rules

profile is selected as an extreme example, which could help reveal resilience issues during the experimentation.

E. Fault Types

Chaos engineering is achieved with controlled fault injection to assess system resilience. Several tools support this practice, each with its own ecosystem of experiments and fault types. Notable examples include *LitmusChaos* [19], *Chaos Mesh* [20], and *Datadog Chaos Controller* [21]. Despite differences in implementation and feature sets, chaos engineering tools generally organize faults into common categories, such as **pod-level, node-level and network-level faults**.

For our multi-cluster DNS setup, the primary focus is on fault types that target specific components and render them unavailable or unreachable, because such failures most directly affect DNS resolution and record provisioning behavior. Therefore, the following classes of faults are considered relevant: **Pod deletion or container termination** affecting DNS pods in the resolution chain or provisioning services and **network disruption** selectively blocking or degrading traffic between DNS components. From an observable perspective at the DNS client level, both mechanisms have equivalent effects and therefore either of them is suitable for resilience evaluation.

F. Testing Strategy

This section presents a high-level description of the testing strategy applied to the *5gc.3gppnetwork.org* zone. It defines the considered fault scenarios, establishes the steady-state behavior of the system in the absence of disruption, and specifies the expected system behavior under each test case. Finally, it introduces the set of observable metrics used to assess system resilience, correctness, and recovery throughout the experiments.

1) Test Cases: Since the testing scope is limited to the zone *5gc.3gppnetwork.org*, the meaningful test cases for this setup are the following:

TABLE VI: Steady state under different caching configurations. QPS → Queries Per Second; Comp. → Components; Res. → Resolution; Auth. → Authoritative.

Metric	No Caching	Standard Caching
Requests successfully resolved	51 QPS	51 QPS
Accuracy of DNS Responses	99.3%	98.5%
Client Observed DNS Res. Latency	0.85 ms	0.45 ms
Client Observed DNS Res. Jitter	0.12 ms	0.09 ms
K8's DNS Comp. Requests	62 QPS	62 QPS
K8's DNS Comp. Resp. Latency	0.5 ms	0.25 ms
K8's DNS Comp. Response Jitter	0.03 ms	0.013 ms
DNS Forwarder Requests	51 QPS	14.9 QPS
DNS Forwarder Response Latency	0.33 ms	0.42 ms
DNS Forwarder Response Jitter	0.03 ms	0.02 ms
Auth DNS Server Requests	51 QPS	7.8 QPS
Auth DNS Server Resp. Latency	1.6 ms	4.5 ms
Auth DNS Server Response Jitter	0.008 ms	0.009 ms

- **TC1**: Disruption targeting the K8's-DNS
- **TC2**: Disruption targeting the DNS Forwarder
- **TC3**: Disruption targeting the Authoritative Server
- **TC4**: Disruption targeting the Record Provisioning

The first three test cases aim to reveal potential problems on the resolution path, while the latter aims to reveal potential problems in the accuracy of DNS resolutions.

2) Steady-State Hypothesis: Chaos engineering experiments are driven by a Steady-State Hypothesis (SSH), which defines the expected system behavior under normal operating conditions and serves as a baseline for evaluating the impact of injected faults [22]. The hypothesis focuses on observable system behavior rather than internal implementation details.

For the DNS system under evaluation, the steady state corresponds to correct and stable resolution of records under the private zone *5gc.3gppnetwork.org*.

The steady-state hypotheses are defined as follows:

- **SSH1**: DNS resolution for *5gc.3gppnetwork.org* is continuously available to internal workloads.
- **SSH2**: Resolution latency remains within a stable and predictable range.
- **SSH3**: DNS responses are accurate with respect to the cluster state, allowing for limited staleness when caching is enabled.

3) Monitoring Metrics: To evaluate whether the system maintains or returns to steady state during chaos experiments, the measurements for each caching configuration are presented in Table VI. Request rates are measured in queries per second (QPS). Latencies represent the average response latency measured in milliseconds (ms). Jitter is defined as the standard deviation of response latency, also measured in ms. The request rate observed at the K8's DNS components is higher than the traffic generated by the test clients. This is expected, as K8's DNS also serves DNS queries from internal cluster workloads not related to the 5GC zone.

DNS response accuracy can be further classified into the following categories:

- **True Positives (TP)**: queries resolved successfully (return code 0) and response matches the current cluster

state.

- **False Positives (FP)**: queries resolved successfully (return code 0) but returns stale records or records that no longer exist.
- **True Negatives (TN)**: queries returning `NXDOMAIN` (return code 3) correctly because the record does not exist.
- **False Negatives (FN)**: queries returning `NXDOMAIN` even though the record exists but has not yet propagated through the resolution chain.

This classification enables finer-grained analysis of resolution correctness and helps to directly identify subtle failure modes, such as stale negative caching or delayed record propagation. Formally, resolution accuracy is defined as:

$$\text{Resolution Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (1)$$

Resolution accuracy (Equation 1) is used as the primary correctness indicator. This metric is sufficient to detect resilience issues related to record provisioning for the examined disruption cases.

4) Test Case Execution: Chaos injection using Datadog Chaos [21] was integrated into Chainsaw [23] test cases to orchestrate the execution of the defined scenarios. Chainsaw is a testing utility that enables declarative sequencing of test steps and assertions over Kubernetes resources [23]. Each test case follows a structured execution sequence: First, annotate the test case in the observability stack, then inject a fault targeting a single pod of the selected component in the DNS resolution chain or record provisioning path, and maintain the disruption for a bounded duration of two minutes. Secondly, observe the system recovery and annotate the return to steady state.

V. Evaluation

This section evaluates the system behavior during the execution of test cases **TC1–TC4**. The experiments are conducted under two caching profiles: *Standard-Caching* (enabled) and *No-Caching* (disabled). For each test case and configuration, system behavior is analyzed using the 13 collected metrics presented in Table VI. The experiments are executed on a two-cluster setup. For convenience, the clusters are named *Zürich* and *Bern*. In practice, both clusters are Kind-based Kubernetes clusters [24] running on the same physical host.

In each cluster, a dedicated DNS client generates traffic consisting of 50 requests per second for existing domains. Additionally, each client performs six DNS record updates per minute of randomly selected test domains from the pre-populated domains in the cluster.

K8s DNS metrics include both test and in-cluster queries, so totals slightly exceed the generated 5gc load by a stable 10% baseline. Metrics from the downstream forwarder and authoritative servers, which serve only 5gc, match the generated traffic. The uplift affects only the traffic volume. The timing and size of drops and recoveries are still clear, so the metric remains reliable. A more granular instrumentation configuration in the CoreDNS pod would remove this. It was not configured during our runs but is a feasible adaptation.

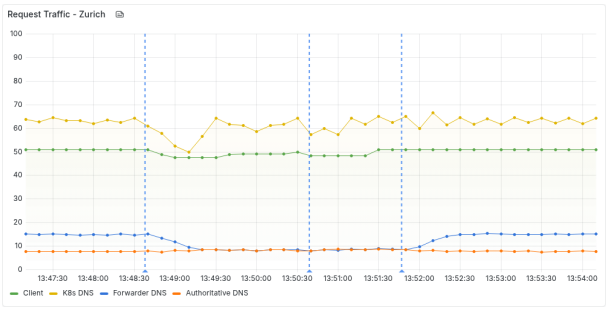


Fig. 5: TC1 — Standard Caching - DNS Traffic Patterns

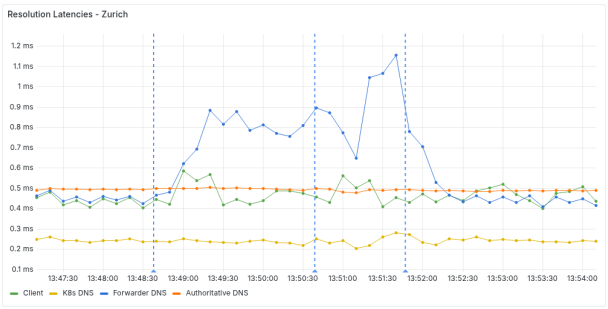


Fig. 6: TC1 — Standard Caching - Latency per Component

For each test execution, only metrics that differ substantially from the steady state are discussed. If a metric is not explicitly mentioned, it is assumed to remain within steady-state bounds. All chaos injections are performed in the Zürich cluster and last for two minutes. After the disruption ends, the system requires approximately 1–2 additional minutes to return to its original steady state. Given the symmetry of the setup, equivalent behavior is expected if the same disruptions were applied in the Bern cluster. In this setup, given the current TTL settings and the chosen fault type, a two-minute injection is enough for all effects to surface in the metrics. Longer outages would not add insight here, though with longer TTLs or different disruption types, longer disruptions may reveal delayed effects in system operation and recovery.

Figures 5 – 11 include three vertical annotations (blue dotted lines): the first and second blue lines denote the start and end of the chaos injection, and the third denotes the point at which the disrupted component has fully recovered and the system returns to its steady state.

A. TC1 — Targeting K8's DNS component

1) Standard Caching: In **TC1** Standard-Caching, **SSH1** does not hold. As shown in Figure 5, the client-side request rate briefly drops to a minimum of approximately 47 QPS shortly after the disruption begins and again after the end of it. These drops correspond to DNS queries that fail to resolve within the timeout period after being forwarded to the unavailable K8's DNS pod.

During the disruption, the request rate processed by the forwarder in the Zürich cluster decreases to approximately 8 QPS. This effect is caused by improved cache efficiency. Un-

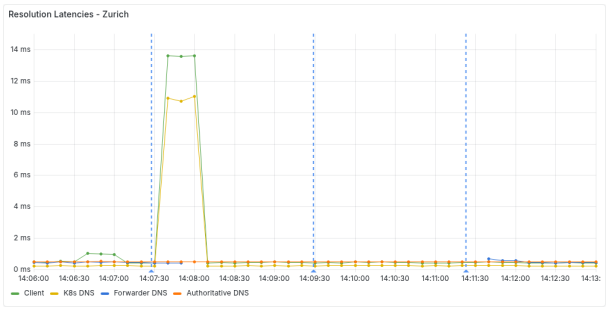


Fig. 7: TC2 — Standard Caching - DNS Latency per Component

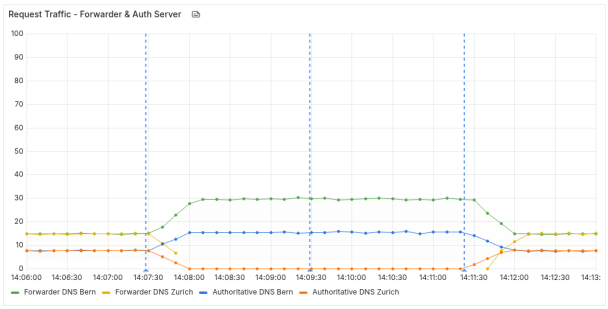


Fig. 8: TC2 — Standard Caching - DNS Traffic Patterns

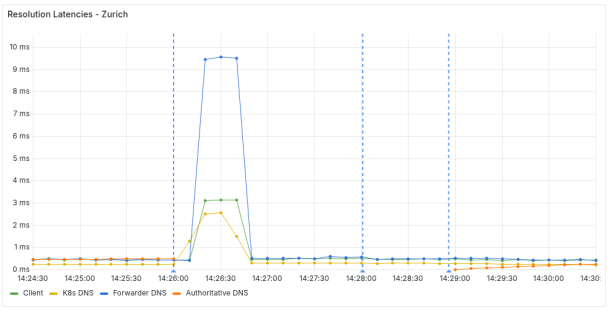


Fig. 9: TC3 — Standard Caching - DNS Latency per Component

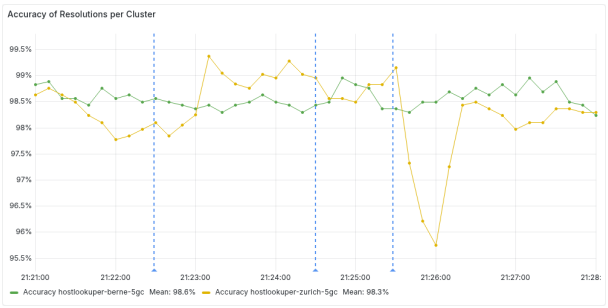


Fig. 10: TC3 — Standard Caching - Accuracy Drop

der steady-state conditions, K8's DNS runs with two replicas, each maintaining its own cache. However, only a single replica remains active during disruption, effectively consolidating the cache. This increases cache hit rates and reduces the number of queries to the Forwarder layer.

Latency measurements for the client, K8's DNS, and DNS Forwarder components are shown in Figure 6. In this case, the DNS Forwarder experiences marginally higher latency throughout the disruption period. With caching configured to five seconds at both the K8's DNS and Forwarder layers, the cache on the Forwarder during disruption becomes ineffective leading to a larger fraction of queries being forwarded to the authoritative DNS servers, resulting to increased response latency.

2) No Caching: A similar pattern of failed DNS lookups is observed during the disruption (**SSH1** does not hold). In this case, all components in the resolution chain experience similar drops in traffic. Since caching is disabled, no upstream component can absorb load, and the traffic reduction propagates uniformly through the entire resolution path. Also, since K8's DNS does not cache responses, no sustained reduction in traffic is observed at the forwarder layer.

In **TC1**, **SSH2** and **SSH3** are not impacted.

B. TC2 — Targeting DNS Forwarder

1) Standard Caching: As seen in the Figure 7, there is some **latency degradation (SSH2)** observed right after the disruption begins, but it returns quickly to the steady state even during the disruption. The same latency degradation is observed from the K8's DNS components, but not from the Forwarder Components.

As observed in Figure 8, all the traffic that was supposed to reach the Zürich Forwarder component, is handled by the Bern cluster Forwarder component. The same effect is observed for the traffic towards the Bern Authoritative Server, since the Bern Forwarder uses only its primary delegation which is the Bern Authoritative Server.

2) No Caching: The same latency effects are observed as in the standard caching case, but they are amplified. Average latency peaks at almost 70 ms at the start of disruption, instead of 14 ms when caching was enabled. Therefore, **SSH2** is impacted only for the beginning of the disruption. The traffic patterns of the Forwarder and Authoritative Server shift in the same way as the previous execution TC2 Standard-Caching. The difference is the number of requests handled in each layer, reaching 102 QPS.

In **TC2**, **SSH1** and **SSH3** are not impacted.

C. TC3 — Targeting the Authoritative Server

1) Standard Caching: As seen in Figure 9, there is an increase in latency observed from the client side. which impacts **SSH2** only at the beginning of the disruption. The effect on latency is less severe compared to the TC2 standard caching case (peak latency is 3 ms, instead of 14 ms), possibly because the disruption is affecting a more distant component from the client. Furthermore, the same effects on latency are observed in the K8's DNS component. Whereas in the Forwarder DNS component, the impact on latency is more severe, with a peak of 9.5 ms. Despite this spike, within the

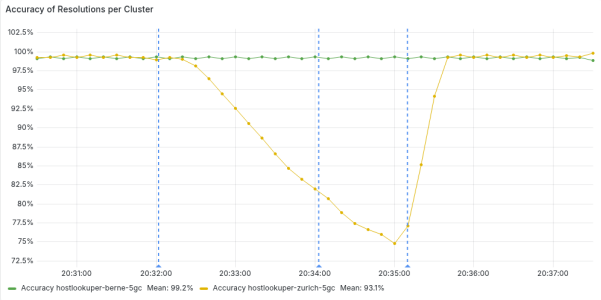


Fig. 11: TC4 — No Caching - Findings

first minute of disruption, the latency approaches the steady state expectations.

As a result of this disruption, an unexpected effect on **resolution accuracy (SSH3)** is observed, which is slightly decreased to **95.5%** after the system is restored. During the disruption, the Authoritative Server is unavailable and therefore does not accept any record updates from the Record Provisioning component. As a result, when the Authoritative Server is restored, it serves the records as they were stored in its database at the moment the disruption began. Some records drifted from the intended cluster state during this period, leading to a temporary decrease in accuracy until the Authoritative Server is fully synchronized with the cluster state.

2) No Caching: Latency is impacted in the Zürich Cluster, so **SSH2** does not hold. In contrast with the cases where caching is used, the effect of the disruption does not seem to wear off depending on how "far" the client is from the outage. The latency hit observed is similar to TC2 No-Caching, climbing up to 63ms for the client side. Traffic patterns are shifted as expected, with the Bern Authoritative DNS server handling all the traffic. Also, the drop in the **resolution accuracy (SSH3)** is observed, as in the *TC3 Standard-Caching* case, with a drop to **93%** right after system restoration. In **TC3**, **SSH1** is not impacted.

D. TC4 — Targeting Record Provisioning

1) Standard Caching: In this test case, no traffic pattern change is either expected or observed. The expected effect of the **resolution accuracy (SSH3)** drop is detected during the disruption, continuously falling during the duration of the disruption, reaching a low of 68%, since the Authoritative Server is no longer updated to reflect the intended state. After the system recovery, the **resolution accuracy** returns to its steady state target after 35 seconds. Additionally, due to the absence of updates, the latency of the Authoritative Server is slightly reduced during disruption.

2) No Caching: As observed in Figure 11, the same effects are observed as in the TC4 Standard-Caching case.

In **TC4**, **SSH1** and **SSH2** are not impacted.

E. Findings

Overall, the system behaves as expected and within acceptable operational limits across all test cases. When caching is enabled, the system demonstrates slightly stronger resilience, particularly when disruptions occur further away from the client in the resolution chain. In these scenarios, caching absorbs the impact of failures, limiting latency degradation at the client. Failover mechanisms also operate as expected. In **TC1**, **TC2**, and **TC3**, traffic is rerouted to the healthy expected fallback component, and the additionally load is handled without steady performance degradation. The Record Provisioning disruption (**TC4**) behaves as anticipated. While accuracy degrades during the outage due to missing updates, the system recovers automatically after restoration, and accuracy converges back to steady-state levels within a short time window.

In the **TC2** and **TC3** No-Caching scenarios, client-side latency degradation is slightly worse. Under the no-caching configuration, the perceived impact of disruptions is independent of the failure location. Failures close to or far from the client produce comparable latency penalties, indicating that the absence of caching eliminates any latency smoothing effect along the resolution path.

When a caching layer uses the same Time-to-Live (TTL) as the upstream component, the benefit of caching is reduced. Caching could be more effective when TTL values differ across layers, with layers closer to the Authoritative Server having higher TTLs. Finally, when caching is disabled, resolution accuracy is slightly higher, while latency is consistently worse. Accuracy never reaches 100% within the observation window because the record-provisioning path is pull-based and eventually consistent. Kubernetes controllers discover changes and reconcile them over successive loops, so updates propagate with short, inherent delays. The instrumentation mechanism is able to expose and quantify this architectural behavior.

Client-side metrics alone are sufficient to validate system's resilience. However, while internal system metrics are not strictly required to determine whether the system is resilient, they are essential for explaining observed behavior and understanding the root causes of deviations during disruptions.

F. Limitations

Our evaluation is constrained by scale and topology. It uses two clusters co-located on a single host with *Kind*, which cannot reproduce the heterogeneous latency, jitter, packet loss, and cross region partitions found in global multi-cluster environments. Hence, the observed convergence, failover, and recovery may look more favorable than they would in real-world deployments.

We exercised only two caching configurations and we focused on disruptions that fully remove a component rather than on brownouts or partial degradations. This leaves corners of the behavior surface unexplored.

Another limitation is that the strategy is evaluated only on a single architecture. This process of evaluation is proved useful for the specific architecture, but some aspects of it might not be transferable to different architectures.

VI. Conclusions and Future Work

In containerized environments, private DNS is often used for service discovery, which leads to more frequent record updates and higher accuracy expectations. These characteristics introduce testing needs that are not adequately addressed by evaluation approaches derived from public DNS. This study highlights that while DNS is rarely the primary root cause of public cloud outages, private, cloud-native DNS systems are disproportionately affected. By applying systematic chaos-based testing to a multi-cluster deployment, we demonstrate that resilience testing effectively uncovers failure propagation patterns, configuration inefficiencies, and recovery dynamics.

Future work could extend the evaluation to larger, geo-distributed multi cluster topologies deployed on separate hosts to better capture real world latency, jitter, and concurrent load. The caching design space could be broadened to include heterogeneous TTLs across layers and the fault model expanded beyond hard failures to include brownouts and partial degradations of varying duration. Such extensions would enable a more accurate characterization of failure propagation, cascading effects, and recovery dynamics at production scale.

Further validation across different cloud-native DNS architectures, including alternative configurations of the current stack and architectures proposed from other organizations in the future, would help assess portability of the testing methodology and surface provider specific sensitivities. As additional private DNS architectures are documented, applying the testing framework to them could also support a more formal taxonomy of acceptable operational limits, improving comparability among different architectures.

Acknowledgments

This work was partially supported by (a) the University of Zürich UZH, Switzerland, (b) the Horizon Europe Framework Program's project Certify, Grant Agreement No. 101069471, funded by the Swiss State Secretariat for Education, Research, and Innovation SERI, under Contract No. 22.00165, and (c) the Swisscom AG. We acknowledge the use of generative AI solely for grammar, sentence structuring, and basic editing.

References

[1] T. D. N. Encyclopedia, "DNS for Microservices Architectures Challenges and Patterns," https://dn.org/dns-for-microservices-architectures-challenges-and-patterns/?utm_source=chatgpt.com, 2025, [Accessed 16-02-2026].

[2] J. Belamaric, "Introduction to CoreDNS Cloud Native DNS," https://www.cncf.io/wp-content/uploads/2020/08/Introduction-to-CoreDNS-1.pdf, 2020, [Accessed 16-02-2026].

[3] G. Cloud, "Using kube-dns — GKE networking — Google Cloud," https://cloud.google.com/kubernetes-engine/docs/how-to/kube-dns, [Accessed 24-10-2025].

[4] Kubernetes, "DNS for Services and Pods," https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/, [Accessed 04-08-2025].

[5] PowerDNS, "PowerDNS Authoritative Server," https://www.powerdns.com/powerdns-authoritative-server, [Accessed 24-10-2025].

[6] ——, "PowerDNS Recursor," https://www.powerdns.com/powerdns-recursor, [Accessed 24-10-2025].

[7] Kubernetes-Sigs, "External-DNS, Configure external DNS servers dynamically from Kubernetes resources," https://github.com/kubernetes-sigs/external-dns, [Accessed 17-12-2025].

[8] Kubernetes, "Customizing DNS Service," https://kubernetes.io/docs/tasks/administer-cluster/dns-custom-nameservers/, [Accessed 10-08-2025].

[9] H. Liu, H. Wang, Y. Le, J. Liang, Y. Liang, and C. Wu, "A high performance, scalable dns service for very large scale container cloud platforms," in *Proceedings of the 19th International Middleware Conference*. ACM, 2018, pp. 39–51. [Online]. Available: https://dl.acm.org/doi/10.1145/3284028.3284034

[10] B. Erdenebat, B. Bud, and T. Kozsik, "Challenges in service discovery for microservices deployed in a kubernetes cluster," *Infocommunications journal*, vol. 15, pp. 69–75, 01 2023.

[11] G. Cloud, "Google Cloud Service Health," https://status.cloud.google.com/products/TUZUsWSJUVJGW97Jq2sH/history, [Accessed 08-01-2026].

[12] ——, "Google Cloud Service Health," https://status.cloud.google.com/, [Accessed 08-01-2026].

[13] A. W. Services, "AWS Post-Event Summaries," https://aws.amazon.com/premiumsupport/technology/pes/, [Accessed 08-01-2026].

[14] ——, "Aws services health report," https://health.aws.amazon.com/health/status, [Accessed 08-01-2026].

[15] M. Azure, "Azure status history," https://azure.status.microsoft/en-us/status/history, [Accessed 08-01-2026].

[16] danluu, "A collection of postmortems." github.com/danluu/post-mortems, [Accessed 08-01-2026].

[17] G. Cloud, "Multiple Google Cloud services in the europe-west9-a zone are impacted," https://status.cloud.google.com/incidents/dS9ps52MUnxQfyDGPfkY, 2023, [Accessed 27-02-2026].

[18] Swisscom, "Cloud-native telco transformation," https://github.com/swisscom/cloud-native-telco, [Accessed 19-12-2025].

[19] LitmusChaos, "Litmus Experiments," https://litmuschaos.github.io/litmus/experiments/categories/contents, [Accessed 20-12-2025].

[20] C. Mesh, "Chaos Mesh Overview," https://chaos-mesh.org/docs/, [Accessed 20-12-2025].

[21] Datadog, "Chaos-controller, chaos injection examples," https://github.com/DataDog/chaos-controller/blob/main/docs/examples.md, [Accessed 20-12-2025].

[22] C. Commmunity, "PRINCIPLES OF CHAOS ENGINEERING - Principles of chaos engineering," https://principlesofchaos.org/, [Accessed 11-08-2025].

[23] Kyverno, "Chainsaw documentation," https://kyverno.github.io/chainsaw/0.2.3, [Accessed 23-12-2025].

[24] Kind, "Kind documentation," https://kind.sigs.k8s.io/, [Accessed 26-12-2025].