CloudPathFI: Uncovering Cross-Layer Vulnerabilities in Cloud Networks via Path-Aware Fault Injection

Yinqin Zhao*†, Gaoxu Guo*, Chang Liu*, Long Wang*
*REASONS Lab, Institute for Network Sciences and Cyberspace, Tsinghua University
†China Telecom Cloud Technology Co Ltd
{zyq21, ggx21, chang-liu22}@mails.tsinghua.edu.cn, longwang@tsinghua.edu.cn

IEEE INFOCOM 2026 - IEEE Conference on Computer Communications | 979-8-3315-4961-9/26/$31.00 ©2026 IEEE | DOI: 10.1109/INFOCOM59046.2026.11571766

***Abstract*—Cloud networks are critical to modern services, but failures in these networks still happen frequently and are hard to diagnose. To improve reliability, we need to understand which parts of a network are most likely to fail and how different types of faults affect end-to-end communication. Fault injection (FI) is a useful way to study cloud network failures, but current FI tools focus on a single VM/container, APIs, or applications, and cannot test faults along the full network path from source to destination. To address this limitation, we propose CloudPathFI, a fault injection tool that takes actual network paths as input and orchestrates targeted fault injection campaigns along these paths. It supports 15 common fault types based on real cloud incident reports and can test faults across control, data, management, and physical layers. We evaluated CloudPathFI in 924 fault injection experiments, and the results show that CloudPathFI achieves full path coverage, far exceeding baseline tools (6%-60%), while maintaining high reliability (99.78% FI success rate) and low overhead (avg. 4.3% CPU), and uncovering several key findings, such as the fragility of virtual-physical boundaries.**

I. Introduction

Virtualized and cloud-native environments have become the backbone of modern computing platforms, powering web services, enterprise IT, big data, high performance computing clusters, and deep learning frameworks [3], [18], [22]. The rapid adoption of cloud computing is driven by flexible scalability and cost efficiency: the global cloud market is projected to more than double between 2022 and 2027 [2]. However, with the increasing complexity of network layers, from container or VM network interfaces to virtual switches and routers, tunnels (e.g., VXLAN), and the physical underlay, ensuring reliable communication and service delivery has become a significant challenge [26], [37].

In practice, cloud systems are highly sensitive to diverse failures: hardware faults, network disruptions, configuration errors, and software upgrade mistakes can all trigger major outages. Notable examples include multi-hour outages on AWS in 2021, service disruptions on Alibaba Cloud in 2023, and global Microsoft 365 downtime in 2024 [1], [8], [28]. The subtle and dynamic nature of these failures, often propagating through complex virtual network paths, makes understanding, diagnosing, and preventing them a long-term challenge.

While extensive research has tried to improve cloud reliability [33], [42], [48], traditional failure analysis still faces limitations. Most methods rely on static log analysis or post-mortem incident review, which struggle to capture complex and dynamic failure propagation. Against this backdrop, *fault injection* (FI) has emerged as a powerful technique for proactively evaluating system reliability. By purposely introducing faults to simulate real-world failure scenarios, FI can verify a system's error detection and recovery capabilities, providing a foundation for automated reliability testing [32], [47].

The problem addressed in this paper is *how to systematically study the reliability and vulnerability of complex cloud networks using fault injection techniques*. Specifically, we seek to answer: How can one efficiently and realistically inject faults into multi-layer, dynamic, and heterogeneous cloud network environments, so as to reveal hidden weaknesses, gray failures, and vulnerabilities that are difficult to uncover with existing methods?

A variety of fault injection frameworks have been developed. Early tools like ChaosMonkey [23] and Gremlin [34] focused on coarse-grained, VM-level failures like terminating VM instances. More improved methods emerged to improve coverage: NEAT explores network partitions [21], Legolas uses code instrumentation to find partial failures [50], Rainmaker targets the REST API layer [24], and ThorFI injects faults within tenant virtual networks in OpenStack [25]. These frameworks are highly effective for their designed purposes, such as verifying instance-level redundancy (ChaosMonkey), testing application-level error handling against API failures (Rainmaker), or securing tenant-level virtual networks (ThorFI). However, their focus is, by design, on the health of isolated VMs/containers, APIs, or single network layers, lacking explicit path awareness and the ability to test along real end-to-end communication paths systematically. As a result, these approaches cannot fully simulate or inject cascading failures and subtle interactions that occur across multiple layers and components, especially in modern clouds, where overlay networks, virtual switches, and physical infrastructure are deeply mixed together. From a user's perspective, however, what they ultimately experience is the failure of the entire end-to-end path. A user's request either succeeds by completing this journey or fails somewhere along it. Therefore, a better way to test reliability is to *use the complete network path as the basic unit of testing*.

In this paper, we propose **CloudPathFI**, a path-aware fault injection framework designed to systematically evaluate the reliability and vulnerability of cloud networks. CloudPathFI

This work was supported by the National Natural Science Foundation of China (NSFC) under Grant 62132009. Long Wang is the corresponding author.

leverages network path information, obtained through tracing/telemetry systems and user input, to orchestrate fine-grained fault injection along real communication paths in cloud environments. Specifically, our approach (1) collects the network path for a target application or service; (2) determines the injection sequence based on the path, supporting various dependencies and scheduling policies; (3) executes fault injection on selected nodes or links to emulate realistic failures; (4) collects detailed data to assess the impact of each fault; (5) supports automated recovery actions; and (6) analyzes the results to provide actionable insights into network resilience.

By making the end-to-end path a first-class target, CloudPathFI enables systematic and comprehensive fault injection that addresses the limitations of VM-based or protocol-specific tools. This global, cross-layer methodology allows us to uncover path-dependent, multi-domain vulnerabilities and cascading failure scenarios that were previously hidden, offering a much deeper understanding of the true resilience of cloud network systems. Our main **contributions** are as follows:

i) We conduct a comprehensive analysis of representative networking scenarios in cloud and similar virtualized environments, including the end-to-end paths that packets traverse at device port granularity. This path analysis establishes the foundation for path-aware fault injection.

ii) We propose CloudPathFI, to the best of our knowledge, the first fault injection framework that systematically targets *end-to-end network paths* in cloud environments, which in turn enables fine-grained fault injection across multiple network layers, protocols, and tenant domains. To quantify vulnerabilities along these paths, we also design an algorithm that computes a node-specific Impact Score (Section V-E).

iii) We implement CloudPathFI with a library of 15 automated fault injection types based on public cloud incident reports and production experience from China Telecom Cloud, enabling reproducible emulation of a wide range of failure scenarios.

iv) We demonstrate through a comprehensive evaluation of 924 experiments that CloudPathFI achieves full path coverage (100%), substantially outperforming existing methods (6%–60%) with high reliability (99.78% FI success rate) and low overhead (avg. 4.3% CPU). CloudPathFI also enables the discovery of previously hidden path-dependent failures. For example, we find that faults at virtual–physical boundaries, such as VXLAN tunnel endpoints, can degrade service throughput by 78.4% on average, a vulnerability that existing VM- or API-level tools cannot reveal.

II. Related Work and Motivation

While significant progress has been made in cloud reliability, existing fault injection methods often fail to reveal failures that arise along the complex communication paths in modern clouds.

A. Limitations of Existing Fault Injection Techniques

Fault injection is a well-accepted method for evaluating system robustness. Existing tools, however, can be broadly categorized by their limited operational scopes. VM-level tools like ChaosMonkey [23] inject failures by directly terminating VMs or containers to simulate crashes. Code-Level tools like Legolas [50] instrument system code and inject faults such as exceptions or delays at instruction-level granularity. Similarly, API-Level tools like Rainmaker [24] inject faults by intercepting outbound REST API calls at the HTTP layer and returning crafted errors or inducing timeouts. Even more advanced, network-aware tools have fundamental blind spots. ThorFI [25] operates at the IaaS-Level, injecting faults by modifying virtual network components such as flow tables and router ports through OpenStack control APIs. However, it lacks the cross-layer orchestration needed to target the physical underlay or trace faults across virtual and physical boundaries. The core limitation of these methods is their focus on single components or layers. Our analysis of over 40 real-world cloud outage reports (e.g., [1], [28], [30]) confirms that many critical failures propagate through multi-layer paths involving virtual tunnels and misconfigured flow rules, a reality these tools are not designed to test.

B. The Opportunity: Leveraging Network Path Visibility

Recent progress in network telemetry makes path-aware fault injection possible. While the mechanisms for discovering paths are an active research area, our work is not a new network telemetry system. Instead, we leverage the fact that a variety of mature technologies now make it practical to obtain detailed, end-to-end network paths. Techniques such as traceroute, in-band network telemetry (INT) [38], sFlow [49], X-Trace [27], DeTector [46], and Zoonet [52] can extract detailed paths across virtual and physical network components, including tunnels, switches, and routers.

C. Bridging the Gap with a Path-Aware Approach

Existing fault injection tools are limited to single components or layers. This creates a critical gap between how failures are tested and how they actually occur in production. The core of this gap is the lack of a unified context that connects all the individual hops such as VMs, virtual switches, tunnels and physical routers that a request traverses.

Therefore, bridging this gap requires shifting the focus from individual components to the complete network communication path. This is critical in today's complex cloud networks, where network components such as vNICs, virtual switches, VXLAN tunnels, and so on form long dependency chains. If a fault injection framework could use this complete path as its input, it could inject faults at specific hops along that path to directly study the reliability of each hop in the path. With this approach, we can systematically find complex, hidden problems deep within the network that traditional tools cannot.

III. Analysis of Network Scenarios in Cloud Environments

To build a fault injection framework that is "path-aware", it is essential to first understand what a network path in a cloud environment entails. This section analyzes several representative scenarios to illustrate not only the concrete structure of these

paths but also the rich, multi-layer information they contain. This analysis forms the foundation for our entire approach.

In cloud environments, many physical machines (PMs) are connected by physical switches and routers, and many more VMs hosted in the PMs are connected by virtual switches and routers. Figure 1 illustrates an example network topology that connects three PMs (PM1, PM2 and PM3) and their hosted VMs. This topology is a network design recommended by the OpenStack project for cloud networking [17], [44], [45]. We use it as the example in this analysis because network designs in other cloud environments (cloud or non-cloud) share many common basic connection types with it, though actual scenarios may be different in certain aspects: some VMs may be replaced with containers, the number of physical/virtual routers/switches/machines may be different, there may be no virtual routers but only virtual switches, etc.

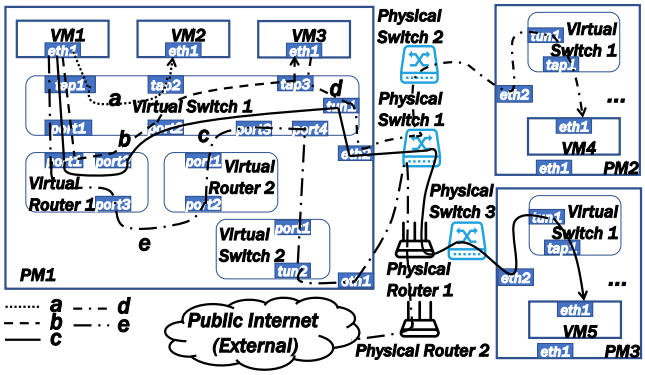

Fig. 1: 5 Example Kinds of Network Scenarios

In Figure 1, a PM has two virtual routers, VR1 (Virtual Router 1) for routing within the computing environment and VR2 for routing to external networks. There is also a physical router for traffic within the environment (PR1) and one connecting to external networks (PR2). PMs are connected to physical switches and routers via physical links while VMs are connected by means of software of virtual network devices (switches, routers, NICs, etc.) and virtual links, which are provided by operating systems or network software tools. Popular implementations of virtual switches and routers include Linux bridges [39], [40] or Open vSwitch (OVS) bridges [43]. Typical virtual links are implemented as the veth-pair mechanism [36] or the tap mechanism [41]. The veth-pair is a data channel with two endpoints, similar to a pipe in POSIX with the difference that veth-pair is bidirectional and POSIX pipe is unidirectional. The tap mechanism is provided by the operating system, i.e., at certain points of the OS kernel's TCP/IP stack's processing, data from a data structure (*a tap device*) are read in or data are put into the data structure; then a VM's network card is virtualized by having its network data flowing in or out of this data structure with the support of the hypervisor and the physical host. For example, the port 1 of Virtual Switch 1 connects with the port 1 of VR1 in PM1 in Figure 1 via the veth-pair mechanism, and *VM1.eth1* connects with *VS1.tap1* via the tap mechanism.

There are many kinds of network scenarios in cloud environments. Here we analyze a few kinds of example scenarios to illustrate the network paths in them. Each path consists of multiple network segments (a segment is the part of network traffic between two adjacent devices/ports along the path). We select these example scenarios so that the segments of the network traffic in these scenarios are representative of network segments in most network traffic. Other kinds of network scenarios on the example topology or other topologies are usually various combinations and/or cascadings of the representative network segments of these example kinds.

Figure 1 illustrates 5 network scenarios (a $\sim$ e). The 6th scenario (f, not shown in the figure) has multi-layer virtual network devices: two PMs (PM1 and PM2) have a virtual switch each, VM1 and VM2 are in PM1 and PM2, respectively, and the two VMs have a virtual switch each, too. We launched the network traffic of these scenarios, inspected them manually and wrote down their network paths, as listed in Table I.

Due to the page limit, here we just briefly describe one scenario and analyze the network paths. In scenario b, the segments *PM1.VS1.tap1* $\rightarrow$ *PM1.VS1.port1*, *PM1.VR1.port1* $\rightarrow$ *PM1.VR1.port2* and *PM1.VS1.port2* $\rightarrow$ *PM1.VS1.tap3* are conducted dynamically by a virtual switch/router according to configured forwarding rules, and the other segments are along virtual links (a tap or veth-pair mechanism). The network traffic goes across two subnets at VR1, i.e., gets forwarded to VR1.port2 by VR1, and the packet header, *ethernet: (VM1.eth1.mac, VR1.port1.mac), IP: (VM1.eth1.IP, VM2.eth1.IP)*, gets changed to *ethernet: (VR1.port2.mac, VM3.eth2.mac), IP: (VR1.port2.IP, VM3.eth1.IP)* by VR1's SNAT, too.

A network path in a virtualized environment is much more than a simple chain of IP addresses or device names. Each path explicitly enumerates the sequence of devices, spanning PMs, virtual switches, virtual routers, and even per-device ports such as tap, veth, tun, and physical NICs. This level of granularity enables a path to uniquely capture not only the logical and physical topology, but also the exact port-to-port context where packets traverse. Moreover, the path naturally bridges information across layers and protocols. For example, it can cross multiple forwarding domains (overlay and underlay), involve various encapsulation and decapsulation points, and record protocol header transformations such as NAT, SNAT, DNAT, VXLAN or GENEVE headers, and MAC or IP rewrites. As a result, the path abstraction contains substantially more actionable information than what can be observed from RESTful APIs, source code alone, or single-point monitoring. It enables both precise localization and context-aware fault injection, down to a specific port, device, or cross-domain boundary along the path.

IV. Fault Model

As analyzed in Section III, cloud network paths span multiple layers of abstraction and traverse a diverse set of virtual and physical components, including vNICs, virtual switches/routers, tunnels, and physical infrastructure. To effectively design CloudPathFI, which aims to uncover path-related faults,

TABLE I: Network Paths in Certain Network Scenarios

Scenario	Description	Network Path
a	Intra-Node Traffic within the Same Subnet	*PM1.VM1.eth1* → *PM1.VS1.tap1* → *PM1.VS1.tap2* → *PM1.VM2.eth1*
b	Intra-Node Traffic in Different Subnets	*PM1.VM1.eth1* → *PM1.VS1.tap1* → *PM1.VS1.port1* → *PM1.VR1.port1* → *PM1.VR1.port2* → *PM1.VS1.port2* → *PM1.VS1.tap3* → *PM1.VM3.eth1*
c	Inter-Node Traffic in Different Subnets	*PM1.VM1.eth1* → *PM1.VS1.tap1* → *PM1.VS1.port1* → *PM1.VR1.port1* → *PM1.VR1.port2* → *PM1.VS1.port2* → encapsulation → *PM1.VS1.tun1* → *PM1.eth2* → *PS1.port1* → *PS1.port2* → *PR1.port1* → *PR1.port2* → *PS3.port1* → *PS3.port2* → *PM3.eth2* → *PM3.VS1.tun1* → decapsulation → *PM3.VS1.tap1* → *PM3.VM5.eth1*
d	Inter-Node Traffic within the Same Subnet	*PM1.VM3.eth1* → *PM1.VS1.tap3* → encapsulation → *PM1.VS1.tun1* → *PM1.eth2* → *PS1.port1* → *PS1.port2* → *PS2.port1* → *PS2.port2* → *PM2.eth2* → *PM2.VS1.tun1* → decapsulation → *PM2.VS1.tap1* → *PM2.VM4.eth1*
e	Traffic to External Network	*PM1.VM1.eth1* → *PM1.VS1.tap1* → *PM1.VS1.port1* → *PM1.VR1.port1* → *PM1.VR1.port3* → *PM1.VR1.port2* → *PM1.VR1.port1* → *PM1.VS1.port3* → *PM1.VS1.port4* → *PM1.VS2.port1* → *PM1.VS2.port2* → *PM1.eth1* → *PS1.port1* → *PS1.port2* → *PR1.port1* → *PR1.port2* → *PR2.port1* → *PR2.port2* → *Public Internet*
f	Multi-Overlay Traffic	*Container1.eth1* → *VM1.VS1.veth1* → encapsulation → *VM1.VS1.tun1* → *VM1.eth1* → *PM1.VS1.tap1* → encapsulation → *PM1.VS1.tun1* → *PM1.eth1* → *PS1* → *PR1* → *PS2* → *PM2.eth1* → *PM2.VS1.tun1* → decapsulation → *PM2.VS1.tap1* → *VM2.eth1* → *VM2.VS1.tun1* → decapsulation → *VM2.VS1.veth1* → *Container2.eth1*

it is critical to establish a systematic and representative fault model grounded in real-world incidents. This section addresses the fundamental challenge of comprehensively characterizing cloud network faults, especially their path-centric impact.

A. Constructing a Taxonomy of Cloud Network Failures

To better understand the many types of failures in cloud networks, we focus on handling different sources of fault data and the complex ways failures can spread. We collect fault data from three main sources: (1) public incident reports and bug databases from major cloud providers (such as AWS, Azure, Google Cloud, Alibaba Cloud, and Huawei Cloud), the CNCF community, and the OpenStack Bugzilla database, offering broad industry coverage; (2) internal operations and maintenance logs from China Telecom Cloud, covering real production scenarios. These include incident and ticket logs, failure reports, and maintenance records for key network devices, providing data that closely reflects real-world production environments; (3) active faults in our own experimental cloud platform, capturing logs, monitoring metrics, and packet traces under controlled failure scenarios. All collected data is standardized using a unified template that records key attributes such as occurrence time, device type, failure symptoms, impact scope, diagnostic process, and recovery strategy. In total, we analyzed 457 fault descriptions and incident tickets from various sources. We use entity extraction based on large language models (LLM) to structure information from tickets and case descriptions, with manual verification for ambiguous samples, thus constructing a retrievable and model-ready fault knowledge base.

For classification and modeling, we use a two-dimensional taxonomy that considers the *trigger mechanism* and the *network environment*. The first dimension divides into hardware and software faults. Hardware faults include physical device defects, aging, and external factors (e.g., NIC chip failure, physical link disconnection, power instability). Software faults include defects in virtualization components, configuration conflicts, or protocol stack anomalies, and can be further categorized into control plane (e.g., SDN controller inconsistency), data plane (e.g., OVS flow table error), and management plane (e.g., Neutron API timeout). The second dimension distinguishes failures in the underlay (physical network) and overlay (virtual network). Physical network failures concern TOR switches, core routers, and their physical interfaces; virtual network failures focus on logical paths and software-defined functions, such as VXLAN MTU mismatch or vSwitch flow table overflow.

B. Fault Models

Based on this fault classification and knowledge base, we selected a set of typical fault models for CloudPathFI to use in fault injection. Table II shows these fault models, which are drawn from real cloud incidents, public reports, and our own experiments. Each model was chosen to cover important types of network path failures at different layers and caused by different triggers, giving us a practical and realistic foundation for testing and evaluation.

In the control plane, SDN controller faults often cause flow tables to be incorrectly issued, resulting in downstream switches losing forwarding rules and making VMs unreachable, as seen in both AWS and Google Cloud incident reports (FM-01). In the data plane, VM routing misconfiguration (FM-02) or VPC peering errors (FM-05) break connectivity across hosts or subnets, as documented in Azure and AWS failures. Management plane failures, such as wrong port settings (FM-04) or Neutron API timeouts, disrupt compute and network management interfaces. Security group misconfigurations, especially on basic protocols (ICMP, TCP/22), commonly lead to base connectivity loss and diagnostic challenges (FM-03). Performance degradation faults include packet loss and delay injections (e.g., on OVS bridges (FM-07), (FM-08)), which may induce application timeouts or slowdowns, as well as traffic surges that cause resets or service interruptions (FM-10). In cross-subnet or cross-tenant paths, gateway flow table errors (FM-11) and load balancer misconfigurations (FM-09) can break connectivity or return errors (such as 504 Gateway Timeout). At the physical layer, switch errors (FM-12) or hardware failures such as power loss (FM-13), fiber cut (FM-14), and NIC failure (FM-15) lead to broad and hard-to-diagnose outages.

TABLE II: Representative Fault Models and Sources

ID	Fault Name	Description	Source	Type
FM-01	SDN controller software fault	Controller pushes incorrect flows, causing VM communication disruption	AWS [20], GCP [16]	Control plane / Disconnect
FM-02	VM routing misconfiguration	Missing or incorrect default route blocks external access	Azure [7]	Data plane / Disconnect
FM-03	Security group misconfiguration	Incorrect rules block traffic, VM unreachable	AWS [19]	Mgmt plane / Disconnect
FM-04	Component port misconfiguration	Component listens on wrong port, affecting manageability	GCP [14]	Mgmt plane / Disconnect
FM-05	VPC peer IP misconfiguration	Boundary host IP error breaks cross-host communication	AWS [6]	Virtual net / Disconnect
FM-06	Virtual router error	Traffic forwarded to wrong next hop, subnet isolated	GCP [10]	Virtual net / Disconnect
FM-07	OVS bridge packet loss injection	Injected loss destabilizes or breaks connections	GCP [11]	Data plane / Packet loss
FM-08	Network delay	Burst latency impacts response	GCP [13]	Data plane / Delay
FM-09	Load balancer misconfiguration	Traffic forwarded to invalid backend, returns 504	GCP [9]	Control plane / Disconnect
FM-10	Traffic burst congestion	Surges trigger resets, degrade performance	AWS [4]	Data plane / Loss, Delay
FM-11	Gateway flow table error	Cross-subnet traffic matching fails, communication breaks	GCP [12]	Control plane / Disconnect
FM-12	Physical switch error	Physical switch malfunction	GCP [15]	Physical net / Disconnect
FM-13	Physical host power outage	Host power loss, all VMs unreachable	AWS [5]	Physical net / Disconnect
FM-14	Fiber link disconnection	Fiber cut, total communication loss	GCP [10]	Physical net / Disconnect
FM-15	NIC hardware failure	NIC chip fault causes communication failure	Internal case	Physical net / Disconnect

V. The CloudPathFI Framework

We now present the design of our solution, building upon our foundational analysis of what cloud network paths entail (Section III) and how they fail (Section IV), we designed and implemented CloudPathFI. It is a path-aware fault injection framework that uses the end-to-end network path as the primary target for testing. By leveraging path data from telemetry and injecting our realistic fault models, CloudPathFI is proposed to systematically discover the complex, cross-layer failures that traditional approaches miss.

A. Architectural Overview

The overall architecture of CloudPathFI is shown in Figure 2. The framework is organized into four modules that manage the lifecycle of a fault injection: (A) Cloud Network Path Discovery, (B) Fault Orchestration, (C) Fault Injection Execution, and (D) Fault Observability & Impact Analysis.

The workflow begins with *Path Discovery*, where the network paths to be tested are identified from telemetry systems, monitoring tools, or user input. These paths serve as a foundational input for the *Fault Orchestration* module. This module acts as the strategic control center, combining the discovered path (the "where") with a declarative *Fault Specification* (the "what") to generate an executable *Fault Campaign*. This campaign is then passed to the *Fault Injection Execution* module, which contains a library of fault primitives to perform the actual injection and subsequent recovery on the cloud infrastructure. Finally, throughout the process, the *Fault Observability & Impact Analysis* module collects telemetry and monitoring data to analyze the system's response, quantify the impact of the injected faults, and store the results for diagnosis and evaluation.

B. Cloud Network Path Discovery

A path is defined as a detailed, ordered sequence of physical and virtual network hops from a source to a destination. While paths can be represented at various granularities, the ideal representation is at the device port level (e.g., as illustrated by the examples in Table I), as this enables the most systematic and fine-grained fault injection. CloudPathFI can acquire these paths from three primary sources. First, it can collect data from fine-grained sources such as advanced telemetry systems (e.g., sFlow [49], In-band Network Telemetry (INT) [38]) and system-level traces (e.g., Open vSwitch datapath traces), which provide real-time traffic paths. Alternatively, in environments without advanced telemetry, paths can be discovered using standard network monitoring tools and cloud-specific APIs. For instance, using traceroute combined with queries to cloud provider APIs (e.g., OpenStack Neutron API for virtual networks or AWS EC2 APIs for instance topology) can reveal the sequence of virtual routers, gateways, and network interfaces, thereby providing sufficient information about the end-to-end path. Finally, for targeted testing or the reproduction of known incidents, users can manually define a path.

C. Fault Orchestration

The Fault Orchestration module is the strategic core of CloudPathFI, responsible for planning and scheduling the fault injection experiments. It translates high-level reliability goals into a specific, executable fault campaign by intelligently combining a fault's definition with its target path. This is primarily achieved through two key components: the Fault Specification and the Fault Scheduler.

The first component, the *Fault Specification*, is a declarative template that defines the *what* of a fault. This allows for the reuse of fault definitions across many different paths and experiments. The specification defines a fault through its fundamental properties: its *Fault Types*, which reference one or more models from our taxonomy (e.g., FM-06: Virtual router error); its configurable *Fault Parameters*, which control intensity and behavior, such as the duration of a network delay or the percentage of packets to drop; and optionally, a *Custom Fault Injection Script* for complex user-defined scenarios. This entire specification is typically defined in a human-readable format like YAML and JSON, allowing for easy management and version control of test scenarios.

The second component, the *Fault Scheduler*, acts as the adaptive engine that translates the abstract specification into a crafted, executable *Fault Campaign*, by determining *where* the

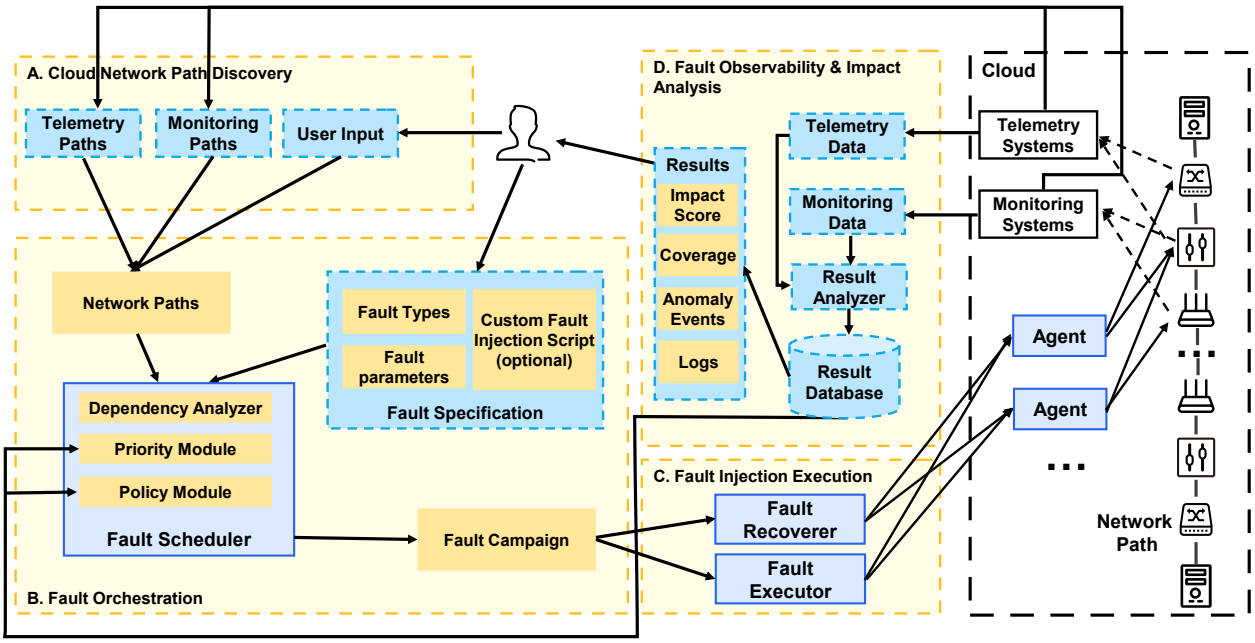

Fig. 2: The overall architecture of the CloudPathFI framework.

fault should be injected. This scheduler operates within a feedback loop, leveraging historical data from the *Result Database* to dynamically refine its decisions over successive injection cycles. The process is initiated by its *Dependency Analyzer*, which moves beyond simple event sequencing to determine the optimal path and order of injections. This analysis begins by dynamically constructing and maintaining a high-fidelity *Service Dependency Graph (SDG)* of the system, leveraging real-time distributed tracing data to create a "live map" of actual service interactions [51]. Upon this graph, the analyzer applies a suite of graph theory algorithms to identify critical choke points and likely failure paths. For instance, it employs topological analysis such as calculating the *Betweenness Centrality* of each node to pinpoint architecturally critical components (e.g., API gateways) that act as "bridges" for inter-service communication [29]. Furthermore, by modeling fault propagation probabilities, it can simulate how failures cascade through the SDG, allowing it to construct multi-stage injection sequences. Following this structural analysis, the *Priority Module* functions as the core risk assessment engine. It ingests the candidates from the Dependency Analyzer and correlates them with historical *Impact Scores* and *Anomaly Events* from the Result Database. It then assigns a dynamic priority score to each potential target based on a multi-faceted criteria, which include: (1) *Observed Vulnerability*, where a high score is given to targets whose past injections led to significant service degradation; (2) *Architectural Criticality*, as determined by the *Dependency Analyzer*'s output (e.g., high centrality score); (3) *Injection Coverage*, which gives a high score to hops that have not been targeted in previous experiments to ensure test coverage expands over time; and (4) *Resilience Strength*, where a high score means the target was barely affected in past fault injection tests. Finally, the *Policy Module* consumes this prioritized list and based on a user-defined strategic goal executes a specific injection policy. Key policies now correspond to the priority criteria: (i) the *Exploitation Policy*, which selects targets with the highest *Observed Vulnerability* scores to deeply probe known system weaknesses; (ii) the *Chokepoint Policy*, which prioritizes targets with the highest *Architectural Criticality* score to assess the overall impact of failures at vital system components; (iii) the *Coverage-Expansion Policy*, which selects targets with the highest *Injection Coverage* score to systematically expand testing into unexplored areas of the system; and (iv) the *Resilience-Probing Policy*, which targets components with the highest *Resilience Strength* to challenge assumptions about system stability and uncover novel failure modes.

The output of the *Fault Scheduler* is the *Fault Campaign*, which is a detailed, step-by-step plan that tells the agents exactly *what* faults to inject, *where* to inject them on the network path, and in what sequence. It serves as the direct instruction set for fault injection and recovery, turning high-level scheduling decisions into concrete actions.

D. Fault Injection Execution

This module is the operational component of CloudPathFI, responsible for executing the actions defined in the *Fault Campaign*. The execution is handled by a distributed system of lightweight *Agents* deployed across the cloud environment, as illustrated in Figure 2. An agent resides on each relevant node within the testbed (e.g., hypervisors, network devices, or virtual machines) and acts as the remote endpoint for executing scripts. The central *Fault Executor* translates the *Fault Campaign* into

specific instructions and dispatches them to the appropriate agents. These instructions specify precisely *what* fault to inject, *where* to apply it, and the corresponding *parameters*. These parameters are directly derived from user-specified fault configurations. For example, an agent on a specific compute node may receive an instruction to inject a 200ms latency (*what*) on the virtual interface vnet0 (*where*) for 60 seconds (a specific *parameter* defining duration). Upon receiving a command, the agent performs the specified fault injection action. These actions are organized into a library that use standard Linux and cloud tools, directly matching our fault models. Table III provides a complete list of these injecting actions.

For every injection action, there is a corresponding, automated recovery procedure. Once the specified injection duration has elapsed, the central *Fault Recoverer* component issues a command to the remote agent to trigger the recovery process.

TABLE III: A comprehensive list of fault injection primitives and their corresponding recovery methods

ID	Injection Method	Recovery Method
FM-01	Inject error flow rule via *ovs-ofctl add-flow*	Delete flow rule & restart controller service
FM-02	Modify default route via *ip route del/add*	Restore static route or reconfigure via DHCP
FM-03	Delete security group TCP rule via *openstack* CLI	Restore security group rule configuration
FM-04	Modify port number in configuration file and restart service	Restore configuration and restart service
FM-05	Inject incorrect peer by modifying tunnel *remote_ip* parameter	Restore *remote_ip* and rebuild OVS tunnel
FM-06	Configure incorrect next-hop via *openstack* CLI	Restore correct routing table configuration
FM-07	Inject packet loss via *tc netem loss*	Clear *tc* injection rule
FM-08	Inject delay via *tc netem delay*	Clear *tc* delay rule
FM-09	Modify backend IP in Nginx configuration	Restore IP and reload Nginx
FM-10	Script a high-concurrency request burst	Traffic shaping + elastic load scaling
FM-11	Add *drop* flow rule to a specific OVS	Clear malicious flow rule + BGP/OSPF reconvergence
FM-12	Inject OpenFlow rule to drop GRE traffic	Delete *drop* rule + verify tunnel recovery
FM-13	Power off the target physical machine	Automatic migration + restore power
FM-14	Disconnect the network cable	Manual repair or enable backup link
FM-15	Disable interface via *ip link set* $< if >$ *down*	Enable backup NIC or restore interface

E. Fault Observability & Impact Analysis

The *Fault Observability & Impact Analysis* module monitors system behavior to analyze the impact of injected faults. During a fault injection, it collects *Telemetry Data* and *Monitoring Data* from the cloud environment, such as service latency, throughput metrics, system logs, resource utilization, and network error rates. This data is processed by the *Result Analyzer*, which correlates the observed system behavior with the fault injection events. The analyzer generates a structured result set for storage in the *Result Database*, as shown in Table IV.

Building upon this collected data, the module then computes a comprehensive *Impact Score* for each node on a path. This score serves to quantify a node's vulnerability to injected faults and the severity of the degradation observed on the path due to a fault originating in that specific node. The score for a given node n in the path during a specific fault injection event, denoted as $S(n)$, is calculated as:

$$S(n) = \sum_{m \in \text{Metrics}} W_m \cdot \text{ND}(m,n) \cdot \text{NDur}(m,n) + \sum_{a \in \text{Anomalies}_n} W_a \cdot \text{Sev}(a) + W_{prop} \cdot \text{PropScore}(n)$$

Here, Metrics refers to key performance indicators like service latency, throughput, error rates, and resource utilization. W_m is a predefined weight for each metric m, reflecting its criticality. $\text{ND}(m,n) \in [0,1]$ (Normalized Degradation) quantifies the severity of performance degradation for metric m on node n, calculated as deviation from baseline and normalized (e.g., for latency, (*ObservedLatency* - *BaselineLatency*) / *BaselineLatency*; for throughput, (*BaselineThroughput* - *ObservedThroughput*) / *BaselineThroughput*)). *NDur*$(m,n) \in [0,1]$ (Normalized Duration) quantifies the persistence of degradation for metric m on node n, representing the duration for which the metric remained outside acceptable bounds, normalized against experiment duration (duration of metric m being beyond 2σ from *baseline / total experiment time*). Anomalies_n is the set of distinct anomaly events observed on node n. W_a is a predefined weight for anomaly type a, and $\text{Sev}(a)$ is its numerical severity score (based on predefined impact levels for specific log errors or system alerts). W_{prop} is a weight for fault propagation. Finally, $\text{PropScore}(n)$ is a composite score designed to capture the extent and severity of fault propagation originating from node n. This score is derived from the SDG and quantifies propagation in two ways: it considers both the sum of criticality scores of directly affected downstream nodes and the count of critical nodes that report observed anomalies linked to node n during the fault event. The analyzer also computes test *Coverage* metrics (e.g., percentage of nodes or variety of fault types tested) and extracts a timeline of *Anomaly Events* and detailed *Logs*.

VI. Evaluation

A. Experimental Setup

Our testbed is a cloud built with OpenStack Yoga on three physical machines (PMs), each with dual Intel Xeon Silver 4214R CPUs and 192GB RAM. Core OpenStack services run on two PMs, while the CloudPathFI orchestrator runs on the third. To generate traffic workloads, we deployed two benchmark applications onto virtual machines (VMs) across PM1 and PM2: an Nginx-based chat system [35] and the Google Online Boutique [31], a multi-service e-commerce storefront. The testbed is monitored using sFlow for traffic analysis, Prometheus for metrics, and standard probing tools (*ping*, *traceroute* and *iperf*). For our experiments, end-to-end network paths were constructed using a semi-automated approach, combining traceroute output, sFlow monitoring, OVS trace data and topological information from the OpenStack

TABLE IV: Result Database Structure

Category	Description
Experiment Metadata	IDs, timestamps, status of experiments.
Fault Injection Details	Target node, fault type, parameters, injection/recovery times and status.
Node-Specific Impact Scores	Calculated $S(n)$ and its contributing components for each affected node.
Performance Metrics	Timestamped latency, throughput, error rates, resource utilization by phase.
Anomaly Events	Detected anomalies: timestamp, type, severity, description, source.
System Logs & Traces	Relevant logs and distributed traces for root cause analysis.
Test Coverage Metrics	Quantitative measures of testing breadth and depth (e.g., node/path/fault type coverage).
System State Snapshots (Optional)	Configuration and process state at key moments for reproducibility.

Neutron (via API queries and configuration files) to produce the detailed, port-level paths as shown in Table I. We implemented and deployed CloudPathFI on the two PMs, with agents running on these machines to perform the fault injections.

B. Framework Validation and Robustness

To validate CloudPathFI end-to-end functionality and robustness, we conducted a large-scale experimental campaign. We created a matrix of 90 unique scenarios by combining the 6 network topologies from Table I with our 15 fault types from Table II. For each scenario in this matrix, we performed 10 to 15 independent fault injection runs with varying parameters (such as the target device, fault configuration, and timing), resulting in a total of 924 fault injection experiments.

Across the 924 injection experiments, CloudPathFI demonstrated high robustness, with a 99.78% success rate. Only two injections failed: one due to a race condition with network topology convergence, and another due to a temporary host privilege issue. The successful runs consistently validated the entire workflow. For example, when injecting a "VPC Peer IP Misconfiguration" fault (FM-05), the workflow followed the steps shown in Figure 2. First, in the *Fault Orchestration* module, the *Fault Scheduler* combined the target path with the fault specification to generate an executable *Fault Campaign*. This campaign was then passed to the *Fault Injection Execution* module, where the *Fault Executor* dispatched the command to the appropriate *Agent*. The *Agent* on the host then correctly executed the primitive to alter the peer IP. Finally, the *Fault Observability & Impact Analysis* module captured the immediate impact; its *Result Analyzer* processed monitoring data to confirm a 100% *ping* failure rate. Our subsequent path analysis verified the connection broke at the intended VXLAN entry point. This successful end-to-end execution repeated consistently for other fault types.

C. Comparative Analysis of Fault Injection Paradigms

To evaluate the effectiveness of our path-aware approach, we conducted a comparison with four baseline injectors across their operational *Scope & Coverage* and *Granularity & Fidelity*. The results are summarized in Table V.

Both the VM-Level (ChaosMonkey) and Code-Level (Legolas) injectors are confined to the VM or container endpoints of a path. While their injection mechanisms differ—one terminates instances, the other instruments code—their path impact is identical. Our experiment, conducted over 924 runs on an average path length of 13.83 hops, shows they can only target an average of 2.17 component hops. This yields a limited *Network Path Coverage* of just 15.69% for both. The API-Level injector (Rainmaker) is even more restrictive. It operates by intercepting REST API calls, meaning it can only inject faults in nodes that actively run a client application. For any network path without such an application, Rainmaker has no injection target, resulting in the lowest Network Path Coverage of 6.00%. The IaaS-Level injector (ThorFI) represents an improvement because it uses hypervisor-level agents to test virtual network segments. This allows it to see much more of the virtual network and reach 60.23% coverage. However, its design lacks the orchestration to target the physical underlay and the fine-grained control to inject faults at a specific device port or flow level.

Traditional tools like ChaosMonkey, Rainmaker, and Legolas are limited to coarser-grained injections (VM-level, API-level, Code-level), making them unable to precisely target network elements or replicate complex fault scenarios listed in Table II. As shown in Table V, their ability to reproduce real-world fault scenarios is very low (e.g., 13.3% for ChaosMonkey and Legolas). ThorFI improves this with 'Tenant Path-level' granularity, allowing for 53.3% scenario reproduction.

D. Key Findings from CloudPathFI

Our experiments with CloudPathFI reveal several key insights into cloud network reliability, supported by quantitative data from our 924 experiments.

Finding 1: Components at virtual–physical boundaries are most fragile. In our experiments, nodes that connect the virtual and physical layers caused much more disruption than others. Our result shows that faults injected at these boundary components, such as OVS bridges (FM-07) and VXLAN tunnel endpoints (FM-05), caused an average service throughput degradation of 78.4%, whereas faults within a single domain (e.g., inside a VM) resulted in only a 31.6% average degradation. For instance, in all 92 experiments of the "VPC Peer Misconfiguration" fault (FM-05), traffic dropped at the VXLAN tunnel ingress, resulting in a 100% ping failure rate.

Finding 2: The layer of a component affects how serious a failure is. We found that a component's layer is just as important as its location for predicting the impact of a fault. Problems in the physical layer (such as FM-14 and FM-15) and in the control plane (such as FM-01 and FM-11) were the most damaging, causing 85.7% of all complete connection losses. By contrast, faults in the data or management planes usually led to slower performance, not total outages.

Finding 3: Faults at key points in the network affect many more services. Our results show that faults at components with a high network centrality (based on the *Architectural Criticality*

TABLE V: Comparison of Fault Injection Tools

Metric	ChaosMonkey [23]	Rainmaker [24]	Legolas [50]	ThorFI [25]	CloudPathFI (Ours)
Scope & Coverage					
Injectable Target: VM/Container Node	✓	✓	✓	✓	✓
Injectable Target: Virtual Network Devices	✗	✗	✗	✓	✓
Injectable Target: Physical Network Devices	✗	✗	✗	✗	✓
End-to-End Network Path Coverage (%)	15.69%	6.00%	15.69%	60.23%	**100.00%**
Granularity & Fidelity					
Injection Target Granularity	VM-level	API-level	Code-level	Tenant-level	**Path-level**
# of Reproducible Fault Scenarios	2 / 15	3 / 15	2 / 15	8 / 15	**15 / 15**
Fault Reproduction Rate (%)	13.3%	20.0%	13.3%	53.3%	**100.0%**

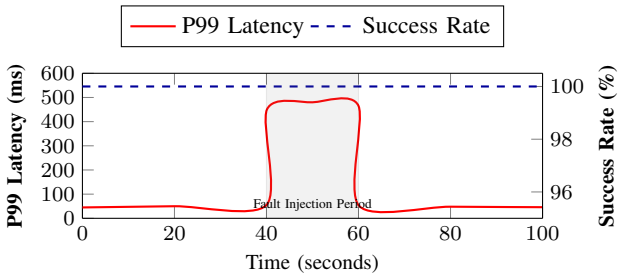

Fig. 3: A gray failure: success rate remains at 100% while P99 latency surges during fault injection.

score from *Dependency Analyzer*), such as a VXLAN gateway (FM-11), consistently received the highest *Impact Scores* from our *impact Analysis* mechanism. Specifically, faults at the top 10% most central nodes affected over 3.24 times as many unrelated services compared to faults at less central points.

Finding 4: Some faults slow down services even when everything looks normal. CloudPathFI was very good at testing "gray failures," which are problems where the system seems healthy because all requests succeed, but the performance is actually much worse. As shown in Figure 3, when we injected 25% packet loss at an OVS bridge in the middle of the path (FM-07), the application's success rate stayed at 100%, but the P99 response time (which measures the slowest 1% of requests) jumped from 50ms to almost 500ms. This means users would experience much slower responses, even though the system shows no alerts.

Finding 5: Localized network faults amplify at the application layer. We found that a fault in just one part of the network can have a much larger effect on the whole application. For example, when we added a 500ms delay (FM-08) to the database server's network link, the P95 response time at the web server (which shows how slow the slowest 5% of requests are) increased by more than 2000ms. This means the problem became 4x worse as it moved through the system, showing that even with extra protection in other parts of the network, some faults can still spread and seriously hurt performance.

E. System Overhead Analysis

To evaluate whether CloudPathFI operates in a lightweight and non-intrusive manner, we measured the system overhead introduced by the framework itself, excluding the intended behavioral changes of the faults (e.g., packet loss or delay). Table VI reports the overhead for each of the 15 fault models. For each test, we ran CloudPathFI's agents without introducing fault logic to isolate the cost of components like the scheduler, agent communication, and log collection.

Our measurements show that CloudPathFI introduces minimal system impact. The maximum CPU overhead was 6.2% (FM-15), but most fault types incur less than 5.0% CPU overhead and under 2.0% memory overhead. The average RTT increase due to framework operations was 2.0 ms, with bandwidth reductions typically within 4.0%.

TABLE VI: System Overhead per Fault Model

Model ID	CPU (%)	Mem (%)	RTT (ms)	Bandwidth (%)
FM-01	5.5	1.8	2.5	-4.5
FM-02	2.1	0.9	0.8	-1.2
FM-03	1.8	0.8	1.0	-1.0
FM-04	2.0	0.7	0.7	-0.8
FM-05	3.9	1.2	1.5	-3.1
FM-06	4.1	1.1	1.7	-3.5
FM-07	4.2	1.5	1.2	-3.0
FM-08	3.8	1.3	2.0	-1.5
FM-09	4.5	1.6	3.0	-6.4
FM-10	5.0	1.7	2.8	-5.7
FM-11	4.7	1.4	2.2	-4.8
FM-12	5.8	2.0	2.9	-6.1
FM-13	5.6	1.9	2.6	-5.6
FM-14	5.4	1.8	2.7	-5.9
FM-15	6.2	2.0	2.2	-7.0
Average	**4.3**	**1.4**	**2.0**	**-4.0**

VII. Conclusion

We presented CloudPathFI, a path-aware fault injection framework for evaluating the reliability and vulnerability of complex cloud networks. By targeting end-to-end network paths rather than isolated APIs, application code, VMs, containers, or IaaS-level components, CloudPathFI injects faults along end-to-end paths, covers different layers and domains, and supports 15 types of faults. We validated our framework with 924 experiments across 90 scenarios. The results show complete path coverage (compared to only 6%–60% for existing tools), high reproducibility (99.78% success), low system overhead, and valuable insights for improving cloud reliability.

REFERENCES

[1] Alibaba cloud fault announcement. https://www.aliyun.com/noticelist/arti-cleid/1064981333.html, 2023. Accessed: 2025-07-19.

[2] Cloud computing platform market by service model, by deployment model, organization size, vertical and region - global forecast to 2027. https://www.reportlinker.com/p05749258/, 2023. Accessed: 2025-07-19.

[3] Kubernetes documentation. https://kubernetes.io, 2023.

[4] Aws service notification 073024. https://health.aws.amazon.com/health/status, 2024. Accessed: 2025-07-19.

[5] Aws service notification 4372t8. https://health.aws.amazon.com/health/status, 2024. Accessed: 2025-07-19.

[6] Aws vpc peering troubleshooting guide. aws documentation. https://docs.aws.amazon.com/vpc/latest/peering/troubleshoot-vpc-peering-connections.html, 2024. Accessed: 2025-07-19.

[7] Diagnose a virtual machine routing problem. learn azure. https://learn.microsoft.com/en-us/azure/virtual-network/diagnose-network-routing-problem, 2024. Accessed: 2025-07-19.

[8] Everything you need to know about the microsoft outage. https://www.itpro.com/software/everything-you-need-to-know-about-the-microsoft-outage, 2024. Accessed: 2025-07-19.

[9] Google cloud incident 4skcabqhef84rnvvyj43. google cloud status dashboard. https://status.cloud.google.com/incidents/4skcabqhef84rnvvyj43, 2024. Accessed: 2025-07-19.

[10] Google cloud incident 58jrpvzppto1s6xjoimj. google cloud status dashboard. https://status.cloud.google.com/incidents/58jrpvzppto1s6xjoimj, 2024. Accessed: 2025-07-19.

[11] Google cloud incident clr48jptqmugcm4xbfwj. google cloud status dashboard. https://status.cloud.google.com/incidents/clr48jptqmugcm4xbfwj, 2024. Accessed: 2025-07-19.

[12] Google cloud incident dtjpnaho1ivrdnccaqku. google cloud status dashboard. https://status.cloud.google.com/incidents/dtjpnaho1ivrdnccaqku, 2024. Accessed: 2025-07-19.

[13] Google cloud incident mqnbag7gz7pgrh9ydsnh. google cloud status dashboard. https://status.cloud.google.com/incidents/mqnbag7gz7pgrh9ydsnh, 2024. Accessed: 2025-07-19.

[14] Google cloud incident ragcw4n9jhrkrajnx2v7. google cloud status dashboard. https://status.cloud.google.com/incidents/ragcw4n9jhrkrajnx2v7, 2024. Accessed: 2025-07-19.

[15] Google cloud incident reports. https://status.cloud.google.com/incidents, 2024. Accessed: 2025-07-19.

[16] Google cloud service health. https://status.cloud.google.com/, 2024. Accessed: 2025-07-19.

[17] *Neutron DVR*, 2024. Accessed: 2024-02-01.

[18] Openstack documentation. https://www.openstack.org, 2024.

[19] Security group setup stopped working (smb shares). aws re:post. https://repost.aws/questions/QUAHR4J0STQiGlQOJU5J1NKA/security-group-setup-stopped-working-smb-shares, 2024. Accessed: 2025-07-19.

[20] Service health - apr 10, 2025. https://health.aws.amazon.com/health/status, 2025. Accessed: 2025-07-19.

[21] ALQURAAN, A., TAKRURI, H., ALFATAFTA, M., AND AL-KISWANY, S. An analysis of {Network-Partitioning} failures in cloud systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018), pp. 51–68.

[22] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I., ET AL. A view of cloud computing. *Communications of the ACM 53*, 4 (2010), 50–58.

[23] CHANG, M. A., TSCHAEN, B., BENSON, T., AND VANBEVER, L. Chaos monkey: Increasing sdn reliability through systematic network destruction. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), pp. 371–372.

[24] CHEN, Y., SUN, X., NATH, S., YANG, Z., AND XU, T. {Push-Button} reliability testing for {Cloud-Backed} applications with rainmaker. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (2023), pp. 1701–1716.

[25] COTRONEO, D., DE SIMONE, L., AND NATELLA, R. Thorfi: a novel approach for network fault injection as a service. *Journal of Network and Computer Applications 201* (2022), 103334.

[26] DI MAURO, M., CERRONI, W., POSTIGLIONE, F., TORNATORE, M., AND TRIVEDI, K. S. Reliability and availability in virtualized networks: A survey on standards, modeling approaches, and research challenges. *arXiv preprint arXiv:2503.22034* (2025).

[27] FONSECA, R., PORTER, G., KATZ, R. H., AND SHENKER, S. {X-Trace}: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)* (2007).

[28] FORBES TECHNOLOGY COUNCIL. 15 actions businesses must consider in light of the recent aws outages. https://www.forbes.com/sites/forbestechcouncil/2022/02/08/15-actions-businesses-must-consider-in-light-of-the-recent-aws-outages/, 2022. Accessed: 2025-07-19.

[29] GAIDELS, E., AND KIRIKOVA, M. Service dependency graph analysis in microservice architecture. In *International Conference on Business Informatics Research* (2020), Springer, pp. 128–139.

[30] GOOGLE CLOUD. Google cloud incident 4skcabqhef84rnvvyj43, 2025. https://status.cloud.google.com.

[31] GOOGLECLOUDPLATFORM. Online boutique microservices demo. https://github.com/GoogleCloudPlatform/microservices-demo, 2023.

[32] GUNAWI, H. S., DO, T., JOSHI, P., ALVARO, P., HELLERSTEIN, J. M., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., SEN, K., AND BORTHAKUR, D. {FATE} and {DESTINI}: A framework for cloud recovery testing. In *NSDI 11* (2011).

[33] GUNAWI, H. S., HAO, M., SUMINTO, R. O., LAKSONO, A., SATRIA, A. D., ADITYATAMA, J., AND ELIAZAR, K. J. Why does the cloud stop computing? lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (2016), pp. 1–16.

[34] HEORHIADI, V., RAJAGOPALAN, S., JAMJOOM, H., REITER, M. K., AND SEKAR, V. Gremlin: Systematic resilience testing of microservices. In *ICDCS* (2016), IEEE, pp. 57–66.

[35] INC., N. Microservices march demo architecture, 2023. GitHub repository containing NGINX gateway and Spring Boot-based microservices.

[36] KERRISK, M. *veth(4) - Linux manual page*, 2023. Accessed: 2024-02-01.

[37] KHAN, H. M., CERVEIRA, F., CRUZ, T., AND MADEIRA, H. Network failures in cloud management platforms: A study on openstack. 228–235.

[38] KIM, C., SIVARAMAN, A., KATTA, N., BAS, A., DIXIT, A., WOBKER, L. J., ET AL. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM* (2015), vol. 15, pp. 1–2.

[39] LINUX FOUNDATION. networking:bridge [wiki], 2024. Accessed: 2024-1-12.

[40] LINUX KERNEL CONTRIBUTORS. *Ethernet Bridging — The Linux Kernel documentation*, 2024. Accessed: 2024-02-01.

[41] LINUX KERNEL CONTRIBUTORS. *Universal TUN/TAP device driver — The Linux Kernel documentation*, 2024. Accessed: 2024-02-01.

[42] LIU, H., LU, S., MUSUVATHI, M., AND NATH, S. What bugs cause production cloud incidents? In *Proceedings of the Workshop on Hot Topics in Operating Systems* (2019), pp. 155–162.

[43] OPEN VSWITCH PROJECT. Open vswitch, 2024. Accessed: 2024-1-12.

[44] OPENSTACK CONTRIBUTORS. *OpenStack Docs: Open vSwitch: High availability using DVR*, 2018. Accessed: 2024-02-01.

[45] OPENSTACK FOUNDATION. *Open Source Cloud Computing Infrastructure - OpenStack*, 2024. Accessed: 2024-02-01.

[46] PENG, Y., YANG, J., WU, C., GUO, C., HU, C., AND LI, Z. {deTector}: a topology-aware monitoring system for data center networks. In *USENIX ATC 17* (2017), pp. 55–68.

[47] RAIYAT ALIABADI, M., AND PATTABIRAMAN, K. Fidl: A fault injection description language for compiler-based sfi tools. In *International Conference on Computer Safety, Reliability, and Security* (2016), Springer, pp. 12–23.

[48] SCHROEDER, B., AND GIBSON, G. A. A large-scale study of failures in high-performance computing systems. *IEEE transactions on Dependable and Secure Computing 7*, 4 (2009), 337–350.

[49] WANG, M., LI, B., AND LI, Z. sflow: Towards resource-efficient and agile service federation in service overlay networks. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.* (2004), IEEE, pp. 628–635.

[50] WU, H., PAN, J., AND HUANG, P. Efficient exposure of partial failure bugs in distributed systems with inferred abstract states. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)* (2024), pp. 1267–1283.

[51] ZHANG, Z., RAMANATHAN, M. K., RAJ, P., PARWAL, A., SHERWOOD, T., AND CHABBI, M. {CRISP}: Critical path analysis of {Large-Scale} microservice architectures. In *USENIX ATC 22* (2022), pp. 655–672.

[52] ZHU, S., LU, J., LYU, B., PAN, T., JIA, C., CHENG, X., KANG, D., LV, Y., YANG, F., XUE, X., ET AL. Zoonet: a proactive telemetry system for large-scale cloud networks. In *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies* (2022), pp. 321–336.