The DORA Methodology: A Developer-First Approach for Resilience Testing

Matthew MacRae-Bovell
Carleton University
Ottawa, Canada
matthewmacraebovell@outlook.com

***Abstract*—Resilience testing and chaos engineering evaluate how a system holds up under adverse conditions such as network delays, dependency outages, or resource exhaustion. Even though downtime poses obvious technical and business risks, these practices remain uncommon outside large organizations. Most current tools require substantial operational expertise, complex environment setup, and workflows that diverge from how developers normally write tests. This paper introduces DORA (Declarative test Orchestration for Resilient Applications), a methodology that brings resilience testing into the test layer by abstracting away the underlying infrastructure and operational complexity. With DORA, developers describe services, faults, workloads, and behavior assertions using the same idioms as their unit and integration tests. Every DORA test creates a temporary environment that sets up services, applies faults, drives traffic, and cleans everything up on its own. We also introduce ChaosSpec, a reference implementation that shows how developers can spin up containers, inject failures, and check system behavior directly from their usual test suites. Our qualitative and quantitative results show that DORA noticeably reduces the work needed to write and run resilience tests, and that it behaves consistently in CI environments. By lowering expertise and setup requirements, DORA makes resilience testing practical for teams of any size and enables further integration into everyday software development.**

***Index Terms*—DORA, Declarative test Orchestration for Resilient Applications, Resilience Testing, Developer Experience, Chaos Engineering, Fault Tolerance, ChaosSpec**

I. Introduction

Today's software systems are expected to remain reliable under adverse conditions. No matter whether the system powers online stores, financial tools, or healthcare applications, outages and surprise failures can carry significant costs. Traditional functional testing and performance testing check correctness and efficiency under expected circumstances, but they do little to ensure whether distributed systems can withstand faults such as dependency outages, network latency, or resource exhaustion [1]. This gap has motivated the rise of resilience testing and chaos engineering[1], practices intended to systematically expose systems to failure scenarios before they occur in production [2]. By injecting controlled faults and observing system behavior, resilience testing gives teams evidence that their applications can degrade smoothly or recover quickly under disruption [2].

Yet, despite its technical value, resilience testing has not achieved widespread adoption. In reality, these practices are still mostly used inside large enterprise companies [3]. Companies like Netflix and Amazon adopted these ideas early on because their strict reliability targets require near-continuous availability [4]. Smaller teams and organizations rarely adopt similar practices [5]. This is not because failures matter less to them, but because today's resilience testing tools don't line up well with how most teams actually build and ship software.

Most tools on the market assume access to mature operational infrastructure, deep networking knowledge, and dedicated platform-engineering support. They often depend on Kubernetes, large observability setups, or specialized fault-injection frameworks [6]. As a result, setting up environments, wiring services through proxies, and handling cleanup usually takes more work than it's worth, especially for teams without dedicated SRE or DevOps support. This leads to a loop where enterprise-focused tools stay usable only inside enterprises, leaving the broader software community without practical, developer-friendly ways to check system resilience.

To help close this gap, we introduce DORA (Declarative test Orchestration for Resilient Applications), a methodology that reframes resilience testing as a developer-first activity rather than an infrastructure-driven one. DORA abstracts away the complex operational steps behind resilience testing and replaces them with simple, declarative interfaces that live directly inside normal test files.. By fitting resilience checks into the tools and workflows developers already use, DORA cuts down the expertise and setup effort usually needed for fault injection and makes resilience testing feel like a normal part of day-to-day development.

II. Background

A. Terminology

Resilience Testing: Is a form of software testing that evaluates how a system responds under adverse or failure conditions. Unlike functional tests that check behavior under normal conditions, resilience testing pushes the system with disruptions like hardware failures, network issues, or sudden load spikes [1]. The goal is to determine whether the system can maintain functionality, degrade gracefully, or recover quickly after disruption. In modern distributed and microservice-oriented architectures, resilience testing has become increasingly important because the number of potential failure points has

[1]Definitions will be provided in Background

grown alongside the complexity of service interactions [7]. By subjecting software to failure scenarios before they occur in production, resilience testing reduces downtime, improves user experience, and validates the effectiveness of recovery strategies [8].

Software Fault Injection: A testing technique in which faults are deliberately introduced into a running system to observe how it behaves under error conditions [9]. The goal is to assess system dependability by triggering failures in a controlled way and measuring whether the system continues to operate correctly, degrades gracefully, or recovers.

Chaos Engineering: Is a specialized discipline that extends beyond traditional resilience testing by introducing faults not just to verify recovery mechanisms but to discover weaknesses that only emerge under real operational conditions. While resilience testing checks whether a system behaves correctly under expected failure scenarios, chaos engineering deliberately injects uncertainty into live or production-like systems to reveal unexpected interactions and cascading failures [10]. It treats failure as a scientific experiment: teams form hypotheses about system behavior, inject faults in controlled real-world environments, and observe outcomes through monitoring, observability tooling, and rollback mechanisms [2]. This approach originated in large-scale distributed systems, most notably Netflix's Chaos Monkey, and has evolved into a broader methodology focused on building confidence in system resilience through empirical discovery rather than prescriptive validation [10].

Developer Experience (DX): Describes how straightforward, intuitive, and efficient it is for developers to work with a given tool, framework, or process. It encompasses factors such as ease of setup, clarity of documentation, integration with existing workflows, and the overall satisfaction a developer feels while using the tool. Good developer experience reduces friction, lowers the learning curve, and makes practices more likely to be adopted consistently. Poor developer experience, by contrast, can discourage developers from using even technically powerful tools [11].

Service Level Objective (SLO): A measurable target for the reliability or performance of a system, expressed as a percentage or threshold over a defined period of time. SLOs are derived from Service Level Agreements (SLAs) but are intended primarily for internal use, guiding engineering teams in setting and evaluating reliability goals [4]. For example, an SLO might state that "99.9% of requests must complete successfully within 200 milliseconds over a 30-day window." SLOs are commonly paired with Service Level Indicators (SLIs), which provide the quantitative metrics (e.g., latency, error rate, availability) needed to evaluate whether objectives are being met. By defining explicit reliability targets, SLOs help teams decide when to focus on resilience rather than feature development.

Image: In containerized environments, an image is a lightweight, standalone package that contains everything needed to run a piece of software, including the executable code, runtime, libraries, and dependencies [12]. Images are immutable and can be versioned, enabling reproducible deployments. When executed, an image becomes a container instance.

Container: A container is a lightweight, isolated runtime environment created from an image [12]. Containers share the host operating system kernel but provide process and filesystem isolation, making them faster to start and more resource-efficient than virtual machines. They are widely used for packaging applications and their dependencies consistently across environments.

Container Orchestration: Container orchestration refers to the automated management of container lifecycles, including deployment, scaling, networking, and health monitoring across clusters of machines [12]. Orchestration systems like Kubernetes ensure that distributed applications run reliably and efficiently by handling scheduling, recovery, and resource allocation.

Clusters: A cluster is a collection of interconnected machines (nodes) that work together to run containerized applications [12]. In Kubernetes, clusters manage and schedule workloads across nodes, providing high availability, scalability, and load balancing. Clusters abstract away the underlying hardware or cloud infrastructure, allowing applications to be deployed at scale.

Pods: In Kubernetes, a pod is the smallest deployable unit that can run in a cluster [12]. A pod encapsulates one or more containers that share the same network namespace, storage volumes, and lifecycle. Pods are typically used to run tightly coupled application components that must be co-located and share resources.

Proxies: A proxy is an intermediary service that sits between clients and servers [13], intercepting and forwarding requests. Proxies can be used for load balancing, caching, access control, or fault injection. In resilience testing, programmable proxies such as Toxiproxy are used to inject faults like latency, packet loss, or disconnections into network traffic.

B. Technologies

Docker: An open-source platform that automates the deployment of applications inside lightweight, portable containers [14]. Docker provides tooling and APIs for building images, running containers, and managing container lifecycles, making it foundational for modern DevOps and testing workflows.

Docker Compose: A tool for defining and running multi-container Docker applications [14]. Using a YAML configuration file, developers can specify services, networks, and volumes, enabling reproducible local environments and simplifying orchestration for small-scale systems.

Kubernetes: A container orchestration system originally developed by Google, now maintained by the CNCF [15], [16]. Kubernetes automates deployment, scaling, and management of containerized applications across clusters of machines. It introduces abstractions such as pods, services, and deployments, and has become the de facto standard for running distributed, cloud-native systems.

Redis: An open-source, in-memory data structure store used as a database, cache, and message broker [17]. Redis is valued for low latency and high throughput, often serving as a critical dependency in modern applications.

GitHub Actions: A CI/CD platform integrated into GitHub that allows developers to define workflows for building, testing, and deploying applications [18]. Workflows are defined as YAML files and can be triggered on events such as pushes, pull requests, or manual dispatch.

C. Fault Taxonomy

Resilience testing and chaos engineering experiments are frequently described using high-level categories of failure. The classification presented in this paper is neither exhaustive nor perflectly aligned with any particular industry taxonomy.

Network Faults: Failures that impair or disrupt communication between services.Examples include increased latency, packet loss, bandwidth throttling, network partitions, and DNS resolution failures.

Infrastructure Faults: Failures arising from underlying hardware or virtualization layers that reduce compute or storage availability. Examples include CPU exhaustion, memory exhaustion or out-of-memory termination, disk I/O throttling or full-disk conditions, and node or container crashes.

Dependency Faults: Failures caused by the degradation or unavailability of internal or external services on which the system depends. Examples include complete service outages, elevated error rates, slow or degraded responses, and rate limiting or quota exhaustion.

Application Faults: Failures originating within the application's own logic, configuration, or access-control mechanisms. Examples include configuration errors, uncaught exceptions, deadlocks or handler saturation, incorrect feature-flag activation, and authentication or authorization failures such as invalid tokens, revoked permissions, or certificate misconfiguration.

Concurrency Faults: Failures that emerge from interactions between threads or processes under concurrent execution. Examples include deadlocks, race conditions, thread or connection-pool exhaustion, and livelocks.

Environmental Faults: Failures arising from the broader operating environment or physical infrastructure. Examples include datacenter or availability-zone outages, power loss, and cooling or thermal failures.

III. Related Work

While software resilience has been important as long as there has been software, Netflix was one of the first in recent history to reimagine resilience testing beyond traditional tooling. Chaos Monkey showed that deliberately terminating live instances could reveal hidden weaknesses in a system and its assumed SLO boundaries [19]. This mindset treated failure as a deliberate part of the engineering process, marking a practical shift in how teams approached system reliability [20]. That shift helped spark a broad ecosystem of tools and platforms, each differing in purpose, complexity, and required expertise.

A. Categorizations of Resilience Testing Tools

Within the field, resilience testing tools are often compared directly despite having immensely different scopes and goals. To support clearer discussion, this paper groups existing approaches into three broad categories. These categories are not formal industry classifications; they are simply a practical lens for comparing tools within the scope of this work. The categories are:

Primitive Fault Injection Tools: Low-level utilities that inject a specific class of fault (e.g., CPU stress, network latency) and require developers to manage orchestration and test environments manually. Examples include Stress-ng, which applies CPU, memory, filesystem, and kernel stress using OS-level primitives, and Toxiproxy, a programmable TCP proxy used to inject latency, bandwidth throttling, packet loss, and connection resets.

General Purpose Resilience Testing Tools: Higher-level tools that support multiple fault types and typically wrap one or more primitive fault injection tools behind CLI or API interfaces. Chaosd supports host-level faults (network, process, filesystem, resource) but leaves orchestration and probing largely to the developer. ChaosToolkit offers JSON/YAML-defined experiments that support probes, environment cleanup, and integrate with tools such as Stress-ng, Toxiproxy, and cloud services.

Chaos Engineering Platforms: Platforms built to inject faults through chaos experiments on real production systems. They frequently offer catalogues of predefined failures, experiment schedulers, and observability integrations. Today, most platforms are designed with enterprise infrastructure assumptions, particularly Kubernetes. Popular examples such as Chaos Mesh, LitmusChaos, and ChaosBlade all rely on Kubernetes as their underlying execution environment. Commercial tools like Gremlin, Steadybit, and Harness Chaos Engineering offer broader organizational features guided experiments, reliability scoring, and safety automation, but likewise expect mature platform engineering practices and container-orchestrated environments.

B. Academic Research

Academic research has long explored how to improve resilience testing and make fault reasoning systematic. Lineage-Driven Fault Injection (LDFI) by Alvaro et al. introduced a systematic method for identifying the minimal fault combinations that break an otherwise successful execution, framing fault discovery as a guided search problem [21]. DESTINI (Declarative Testing Specifications) further advanced this direction by modeling expected recovery behavior declaratively and verifying that systems follow their intended recovery paths [22]. Meiklejohn et al. introduced a technique called Service-level Fault Injection Testing and a prototype implementation called Filibuster, that can be used to systematically identify resilience issues early in the development of microservice applications using a novel dynamic reduction algorithm [23].

IV. Problem Description

Resilience testing remains uncommon outside large organizations, not because smaller teams care less about reliability, but because the current resilience testing ecosystem is fundamentally misaligned with everyday software development [5].

Most existing tools and practices grew out of environments with mature DevOps culture, dedicated reliability teams, extensive observability systems, and cluster orchestration platforms like Kubernetes. As a result, the expectations baked into these tools rarely match the skills, workflows, or constraints of typical development teams.

Prior work identifies several recurring obstacles that reinforce this misalignment and help explain why resilience testing is difficult to adopt in practice:

A. High Operational Expertise:

Each category of resilience test demands different specialized technical knowledge that many developers do not have. A network-related test, for example, may require experience with network proxies to simulate latency, dropped packets, or disconnections. A resource stress test may require familiarity with system-level utilities or custom load generators to overload CPU, memory, or disk. Even basic fault-injection experiments often assume a background in infrastructure automation, container orchestration, or monitoring systems. In other words, effective resilience testing often expects expertise across multiple domains that most application developers were never trained for.

B. High Setup and Maintenance Cost:

Configuring resilience testing requires nontrivial setup like provisioning test environments, wiring proxies, integrating observability, defining safe rollback rules, and maintaining the supporting infrastructure over time. Even when initial experiments are successful, sustaining the practice becomes an ongoing operational burden that smaller teams often cannot justify.

C. Lack of Accessible Resilience Tools:

Most resilience testing tools are built with enterprise organizations in mind. There are far more tools in the "Chaos Engineering Platforms" category than in the "General Purpose Resilience Testing Tools" category. Chaos engineering platforms are overkill for small teams as they often assume Kubernetes, advanced observability stacks, near-production environments, and mature DevOps practices [6], [8].

D. Unclear or Intermittent Business Value:

The benefits of resilience testing like reduced incident frequency, improved MTTR, and stronger SLO compliance are easy to justify at large scale but harder to quantify for smaller teams. When teams must choose between delivering new features or validating system resilience, feature work often takes priority. If resilience checks were easier to create and integrate into normal development workflows, they would be far more likely to be adopted consistently. Research shows that organizations without mature reliability practices often abandon chaos testing because results feel inconsistent, qualitative, or difficult to measure [6], [24], [25].

E. Poor Developer Ergonomics:

Most tools do not integrate with how developers write tests. Fault injection is often defined through YAML files, CRDs, CLI tools, or platform dashboards rather than within familiar testing frameworks. This forces developers to switch mental contexts, moving from writing a unit test to authoring a separate fault-injection workflow, which creates friction and discourages routine use.

V. The DORA Methodology

We propose "Declarative test Orchestration for Resilient Applications" (DORA) a methodology that aims to reduce the time, effort, and specialized knowledge traditionally required of developers for fault injection by abstracting away domain-specific infrastructure and tooling concerns.

This is accomplished through a declarative interface that shifts operational responsibility from the developer to an execution environment or engine that follows the DORA methodology.

Instead of requiring developers to configure proxies, manage containers, or script teardown logic, a DORA runtime automatically handles service orchestration, fault injection, workload execution, and environment cleanup.

By expressing experiments declaratively, developers describe *what* should happen during a resilience test rather than *how* it should be implemented. This abstraction eliminates the need for domain-specific infrastructure knowledge, reducing the time and therefore cost it takes to write resilience tests and makes resilience testing as natural as writing a standard automated test.

Here is a pseudocode representation of how a DORA test could appear:

```
describe("when Redis experiences an outage", () => {
  it("will returns a 500 response", () => {
    // 1. Setup environment and services under test
    redis = createService("redis:7")
    app = createService("app", { REDIS_URL: redis.url })

    // 2. Inject the fault (simulate Redis outage)
    injectOutageFault(redis)

    // 3. Execute the workload that depends on Redis
    response = GET(app, "/api/books")

    // 4. Assert the expected outcomes occur
    expect(response.status).toBe 500
    expect(response.body).toContain "unavailable"
  })
})
```

[2]

The Declarative Orchestration for Resilience Automation (DORA) methodology defines a set of principles that guide how tests and frameworks implementing DORA should be written, executed, and evaluated:

A. Co-located Test Definitions

Every DORA test should encapsulate the following components:

1) **The environment declaration** (e.g., a database, cache, or application service).
2) **The fault or failure to inject** (e.g., network latency, dependency outage, concurrency race).

[2]Further details and examples will be provided below

3) **The workload or traffic generation** (e.g., HTTP requests, background jobs, or concurrent clients).
4) **The behavioral assertions.** (e.g., meeting latency SLOs, graceful degradation, bounded error rates).

Co-locating all components ensures that DORA tests remain readable, reproducible, and self-contained. Each test serves as a complete description of a resilience experiment, capturing both the system configuration and the expected behavior under failure.

B. Declarative Service and Fault Model

Services and faults in DORA should be expressed declaratively through interfaces that describe intent rather than mechanism. The DORA implementation is responsible for mapping these operations to the appropriate underlying mechanisms, such as network proxies, hardware stressors, or container APIs.

C. Infrastructure Agnosticism Interface

Test frameworks implementing DORA should ensure interfaces do not expect any understanding of the DORA runtime's implementation infrastructure or tooling. For example, a network latency test should not require any understanding of how the underlying network proxy tool works, only the services directly declared in the test. By hiding implementation details behind a consistent, declarative interface, DORA allows any developer, regardless of operational expertise, to author, read, and maintain resilience tests with confidence.

D. Hermetic Per-Test Orchestration

Each test should execute within a hermetic environment that includes its own network, services, proxy layer, etc. The environment must be created before execution and torn down after completion. This per-test orchestration model guarantees isolation, determinism, and repeatability. It prevents shared-state interference between tests and removes the need for pre-provisioned clusters or persistent test infrastructure.

E. Automatic Provisioning and Cleanup

Service provisioning, fault injection, and teardown must be automated by the DORA runtime. Manual setup or persistent environments violate compliance. This guarantees that experiments are isolated, reproducible, and easy to execute as part of continuous integration pipelines.

VI. Implementation

To demonstrate the feasibility of DORA, we present **ChaosSpec**, a reference implementation that operationalizes the methodology into a prototype testing framework. This description of ChaosSpec's implementation is a simplification of what can be found on the Github repository which contains documentation, all source code, and example test cases: https://github.com/MathyouMB/chaos-spec

A. Context of the Implementation

This paper was originally motivated by the need to build resilience testing tooling for less complex infrastructure setups. The project was therefore deliberately designed around a less mature set of service orchestration technologies, prioritizing developer accessibility and minimal operational prerequisites[3]. By making resilience testing approachable in these contexts, the implementation demonstrates that meaningful reliability checks can be conducted even without enterprise-scale infrastructure.

Over the course of development, it became clear that the broader contribution was not merely a single tool but a methodology. Framing DORA as a general approach rather than a specific implementation allows its principles to be realized across diverse environments, from local developer machines to production-grade orchestration platforms.

For example, one could envision DORA being applied both in minimal technology stacks and in complex Kubernetes-based systems. In richer environments, the same declarative, test-native abstractions could operate atop existing orchestration layers, extending DORA's reach without changing its core philosophy. Such adaptability broadens the applicability of the methodology while preserving its central goal: making resilience testing faster to adopt, easier to write, and accessible to developers regardless of infrastructure maturity.

B. Technologies

ChaosSpec is written in TypeScript on Node.js. It integrates with Jest's test runner, lifecycle hooks, and assertion syntax so resilience checks look and behave like ordinary unit and integration tests. Testcontainers is used for environment orchestration. It provisions hermetic, per-test Docker networks and services on demand, enabling isolation, reproducibility, and clean teardown. Network-related faults are injected via Toxiproxy, which is interposed between services to introduce degradations such as latency, bandwidth caps, packet loss, and artificial disconnects. Stress-ng is used for all hardware-related stressors, allowing ChaosSpec to apply controlled CPU load and emulate resource-constrained environments.

It is worth noting that the technologies used in the prototype were selected primarily due to developer familiarity and the goal of rapidly producing a proof-of-concept. These choices reflect implementation convenience rather than inherent constraints of the DORA methodology, which remains independent of ChaosSpec's underlying tooling.

[3]Limitations will be further described in the Discussion section

C. Developer API

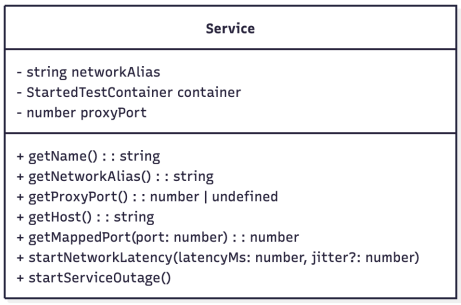
ChaosSpec provides a developer-first API designed to make resilience testing feel as natural as writing unit tests.

Key Library functions:

- `createService` – Launches a containerized service inside the test's isolated network. Supports options such as container image, exposed ports, environment variables, and proxy configuration.
- `buildLocalDockerImage` – Builds a Docker image from local source code so it can be used directly in tests.
- `startSimultaneously` – Executes multiple actions at the same time, enabling tests for race conditions, concurrency issues, or ordering faults.
- `timedHttpRequest` – Wraps an HTTP request and returns both the response and measured duration, making SLO checks straightforward.
- `startTrafficExperiment` – Runs controlled load tests by generating concurrent requests under specified strategies, while collecting metrics such as latency percentiles and error rates.

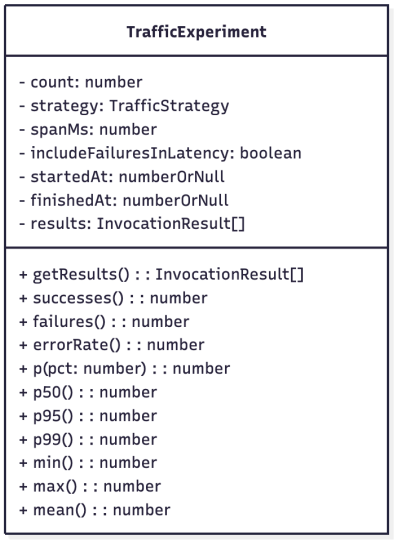
Service Object:

The `createService` function exposes fault injection methods via the returned `Service` object



- `service.startNetworkLatency` – Injects network latency into a specific service, simulating slow or degraded dependencies.
- `service.startServiceOutage` - Simulates a total service outage (the service stops responding or the proxy returns connection errors/timeouts).
- `service.startCpuStress` - Applies controlled CPU load inside the service container, simulating conditions where processing capacity is saturated.

TrafficExperiment Object:



`TrafficExperiment` is an object that generates concurrent traffic and aggregates results into test-grade metrics such as latency percentiles, error rates, and extrema. When invoked through `startTrafficExperiment`, the framework captures the workload (e.g., an HTTP request), schedules a specified `count` of invocations across a time window `spanMs` using a configurable `TrafficStrategy`, records each invocation as an `InvocationResult` (including timestamps, durations, statuses, and errors), and returns a completed experiment object exposing both raw results and aggregate statistics. Configuration parameters include the total request count, the scheduling window, the chosen traffic strategy (such as even spacing, bursts, or ramp-up), and whether failed invocations should be included in latency distributions. The result model allows developers to query successes, failures, and error rates, as well as latency summaries through functions such as `p50()`, `p95()`, `p99()`, `min()`, `max()`, and `mean()`. Developers are encouraged to assert on percentiles and error rates rather than raw wall-time to avoid misleading results.

D. Network Latency Fault Test Example

This test below validates how the application responds when its Redis dependency is slowed by injected latency. The test provisions both Redis and the application service in isolated containers, injects a 500 ms delay into Redis, then issues a read request through the service. The assertion checks that the request still succeeds and completes within a 2000 ms latency SLO.

```
import {
 createService,
 timedHttpRequest,
} from "@chaosspec";

describe("when Redis experiences 500ms of network latency", () => {
 it("will still meet the SLO of 2000ms for a read request", async () => {
   const SLO_MS = 2000;

   // 1. Setup the services under test
   const redis = await createService("redis", {
     image: "redis:7.0-alpine",
     portToProxy: 6379,
   });
   const service = await createService("example-service", {
     image: "example-service:dev",
     portToExpose: 5000,
     environment: {
       REDIS_URL: `redis://${redis.getHost()}:${redis.getProxyPort()}`,
     },
   });

   // 2. Inject 500ms latency into Redis
   await redis.startNetworkLatency({ latencyMs: 500 });

   // 3. Send HTTP request and measure duration
   const serviceUrl = `http://localhost:${service.getMappedPort(5000)}/api`;
   const [response, duration] = await timedHttpRequest(serviceUrl, {
     method: "GET",
     headers: { "Content-Type": "application/json" },
   });

   // 4. Assert that the measured duration is within the SLO
   expect(response.ok).toBe(true);
   expect(duration).toBeLessThanOrEqual(SLO_MS);
 });
});
```
[4]

E. Service Outage Fault Test Example

This test evaluates system behavior when Redis is completely unavailable. After provisioning Redis and the application, the test triggers a simulated outage on the Redis service. An HTTP request is then issued to the application, which is expected to fail gracefully by returning a 500 status. The assertion confirms that the failure is handled as expected, preventing silent errors or incorrect data from being returned.

```
describe("when redis is completely unavailable", () => {
  it("will return a 500 error on /api/books", async () => {
    // 1. Setup the services under test
    const redis = await createService("redis", {...});
    const service = await createService("example-service", {...});

    // 2. Prevent the example service from connected to Redis
    await redis.startServiceOutage();

    // 3. Send HTTP request
    const serviceUrl = `${service.getMappedPort(5000)}/api/books`;
    const response = await fetch(serviceUrl, { method: "GET" });

    // 4. Assert that response is a 500
    expect(response.status).toBe(500);
  });
});
```
[5]

F. CPU Stress Fault Test Example

This test demonstrates how DORA can test resilience under infrastructure stress. The application service is constrained to one CPU core and 0.5 GB of memory, and then placed under simulated 80% CPU load. A read request is issued while the service is stressed, and the test asserts that the response still succeeds within the configured SLO. This provides evidence that the application maintains acceptable performance even under compute resource exhaustion.

```
describe("when limited to 1 CPU and 0.5GB mem, under 80% CPU load", () => {
  it("will respond within SLO while stressed", async () => {
    const SLO_MS = 2000;

    // 1. Setup the services under test (with cpu and memory limits)
    const redis = await createService("redis", {...});
    const service = await createService("example-service", {...,
      resources: {
        cpuCores: 1,
        memoryGb: 0.5,
      },
    });

    // 2. Inject cpu stress
    await service.startCpuStress({ loadPercent: 80 });

    // 3. Send HTTP request
    const serviceUrl = `${service.getMappedPort(5000)}/api/books`;
    const [res, dur] = await timedHttpRequest(serviceUrl, {...});

    // 4. Assert the response is ok
    expect(res.ok).toBe(true);
    expect(dur).toBeLessThanOrEqual(SLO_MS);
  });
});
```
[6]

G. Concurrency Ordering Fault Test Example

In addition to fault injection on individual services, ChaosSpec also provides first-class support for concurrency and ordering faults. To make such scenarios testable, ChaosSpec exposes the `startSimultaneously` helper function, which allow multiple actions to be triggered at the same instant. The following example illustrates a concurrency experiment. Two users attempt to check out the last copy of a book at the same time.

```
describe("when two users take out the last copy of a book simultaneously", () => {
  it("will inform one user that the book has already been taken", async () => {
    const BOOK_WITH_ONE_COPY_ID = "book-with-one-copy-left";

    // 1. Setup the services under test
    const redis = await createService("redis", {...});
    const service = await createService("example-service", {...});

    // 2. Trigger two HTTP requests at the same time
    const serviceUrl = `${service.getMappedPort(5000)}/api/loans`;
    const [firstResponse, secondResponse] = await startSimultaneously([
      () =>
        fetch(serviceUrl, {
          method: "POST",
          body: JSON.stringify(
            { userId: "user-1", bookId: BOOK_WITH_ONE_COPY_ID }
          ),
        }),
      () =>
        fetch(serviceUrl, {
          method: "POST",
          body: JSON.stringify(
            { userId: "user-2", bookId: BOOK_WITH_ONE_COPY_ID }
          ),
        }),
    ]);

    // 3. Assert that one of the users received book was no longer available.
    const messages = [firstResponseJson.message, secondResponseJson.message];
    expect(messages).toContain("Book is no longer available");
  });
});
```
[7]

H. Load Testing Support

ChaosSpec also supports developer-friendly load testing. Its built-in traffic engine can generate concurrent requests during fault scenarios and produce test-grade metrics, such as latency percentiles and error rates that developers can directly assert against.

In the example below, we inject 500 ms latency into Redis and then drive 500 simultaneous reads against the service. We assert that the p95 latency stays within an SLO of 2000 ms, and that the error rate ≤ 2%:

[4]See Appendix: “Network Latency Test Example” for line-by-line explanation

[5]See Appendix: “Service Outage Resilience Test Example” for line-by-line explanation

[6]See Appendix: “CPU Stress Fault Test Example” for line-by-line explanation

[7]See Appendix: “Concurrency Ordering Fault Test Example” for line-by-line explanation

```
describe("when redis experiences 500ms latency", () => {
  it("will maintain p95 ≤ SLO for 500 simultaneous reads", async () => {
    const SLO_MS = 2000;

    // 1. Setup the services under test
    const redis = await createService("redis", {...};
    const service = await createService("example-service", {...});

    // 2. Inject 500ms latency into Redis
    await redis.startNetworkLatency({ latencyMs: 500 });

    // 3. Send 500 HTTP requests over 1000ms
    const serviceUrl = `${service.getMappedPort(5000)}/api/books`;
    const experiment = await startTrafficExperiment(
      () => fetch(serviceUrl, { method: "GET" }),
      {
        count: 500,                // total requests
        strategy: strategies.even, // evenly fire the requests
        spanMs: 1000,              // spread requests over 1s
      },
    );

    // 4. Assert p95 was within SLO and error rate is below 2%
    expect(experiment.errorRate()).toBeLessThanOrEqual(0.02);
    expect(experiment.p95()).toBeLessThanOrEqual(SLO_MS);
  });
});
```
[8]

I. *Intended CI Usage*

ChaosSpec tests are often long-running and may significantly increase pipeline execution time. To prevent them from blocking regular developer workflows, the recommended approach is to run ChaosSpec as a separate automated test suite, distinct from your standard functional tests.

To make this separation straightforward, ChaosSpec enforces a naming convention: all resilience test files must end with `.chaos.test.ts`. This allows you to configure your project so that functional and chaos tests can be executed independently.

A simple example using `package.json` scripts is shown below:

```
{
  "name": "example-node-project",
  "version": "1.0.0",
  "scripts": {
    "test":
      "jest --testPathPattern='^(?!.*\\.chaos\\.test\\.ts$)'",
    "test:chaos":
      "npx jest --testPathPattern='\\.chaos\\.test\\.ts$'",
  },
}
```

To integrate ChaosSpec into continuous integration pipelines we recommend not triggering job automatically on every commit. Instead, using manually queued workflows allow teams to run resilience experiments on demand. This ensures that long-running chaos tests can be executed when needed without slowing down the standard development workflow.

A minimal Github Actions example is shown below. The job provisions Docker, installs dependencies, and executes the ChaosSpec suite via `npm run test:chaos`:

```
name: Manual ChaosSpec CI Workflow
on:
  workflow_dispatch: {}
jobs:
  chaos-tests:
    runs-on: ubuntu-latest
    timeout-minutes: 60
    permissions:
      contents: read
    steps:
      - name: Checkout
        uses: actions/checkout@v4
      - name: EnsureDockerInstalledRunning
        shell: bash
        run: |
          if ! command -v docker >/dev/null 2>&1; then
            sudo apt-get update
            sudo apt-get install -y docker.io
            sudo systemctl enable --now docker
          else
            sudo systemctl start docker || true
          fi
          docker --version
          sudo docker info
      - name: SetupNode
        uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'
          cache-dependency-path: package-lock.json
      - name: InstallDeps
        run: npm ci
      - name: RunChaosSpecs
        env:
          CI: true
        run: npm run test:chaos
```
[9]

J. *Runtime Architecture*

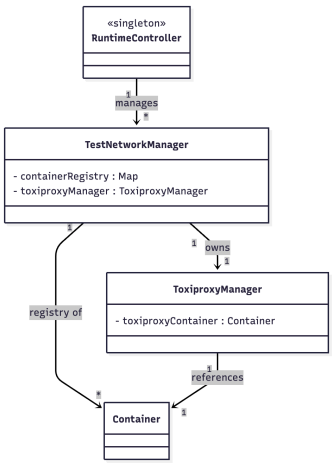
Functions in the Developer API that create new nodes in the network or inject faults ultimately delegate to the RuntimeController.

The RuntimeController is a singleton that acts as the central entry point into the runtime module. It maintains a registry of TestNetworkManagers, with each one corresponding to a specific test case. Every test block is assigned a unique key, which the RuntimeController uses to associate it with its corresponding TestNetworkManager.

A TestNetworkManager is responsible for provisioning and managing all infrastructure required for its test. This includes maintaining a registry of containers and coordinating the test's network environment. Each TestNetworkManager also holds a reference to a single ToxiproxyManager, which provides the interface for injecting faults via the underlying Toxiproxy container.

[8] See Appendix: "Load Testing Example" for line-by-line explanation

[9] See Appendix: "GitHub Actions Example" for line-by-line explanation



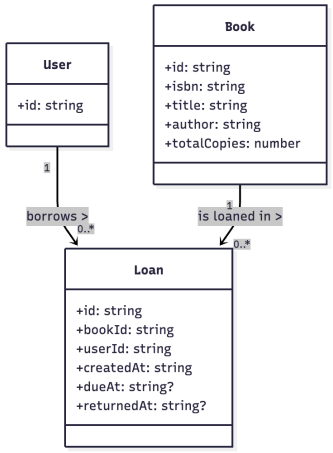
K. Example Fault Prone Service

While building ChaosSpec, we also developed an intentionally fault-prone service used throughout the test examples above. This service, called the **Library Service**, provides a realistic but lightweight target for resilience validation. It is a small Express.js and TypeScript application that exposes REST endpoints for listing books, creating loans, and viewing a user's loan history. The service stores all state in Redis, including book availability and active loans, making it naturally sensitive to latency, dependency outages, and concurrency faults.

The Library Service is intentionally simple: it does not implement retries, circuit breakers, or distributed locks. This absence of safeguards makes it ideal for demonstrating how the DORA methodology exposes typical failure modes in ordinary applications. When Redis becomes unavailable, network latency increases, or multiple users attempt to borrow the same book simultaneously, these conditions propagate into realistic errors such as slow responses, duplicate loans, or failed requests.

An overview as well as the source code for the Library service can be viewed on https://github.com/MathyouMB/chaos-spec/blob/master/example-service

Service Domain Model:



- `User` – Represents an individual user of the library system. Each user can hold zero or more active loans, identified by a unique `id`.
- `Book` – Represents a book available in the catalog. Each book includes metadata such as `isbn`, `title`, `author`, and `totalCopies`. The field `totalCopies` determines the maximum number of concurrent loans that can exist for that book.
- `Loan` – Represents a record of a user borrowing a book. Includes references to `bookId` and `userId`, along with timestamps for `createdAt`, `dueAt`, and `returnedAt`. Loans without a `returnedAt` timestamp are considered active.

Service RESTful API:

- `GET /api/books` – Lists all available books in the catalog.
- `GET /api/books/:id` – Retrieves a single book by its ID. Returns `404` if the book is not found.
- `POST /api/loans` – Creates a loan for a specified `{ bookId, userId }`. Returns `201` on success, or `409` if the book is no longer available.
- `GET /api/users/:id/loans` – Returns all active loans associated with a given user.
- `GET /api/` – Returns health information or a greeting message

VII. Evaluation Methodology

To evaluate whether DORA provides a practical developer-first approach to resilience testing, we conducted both qualitative and quantitative analyses. Qualitatively, we compared DORA and its reference implementation, ChaosSpec, against two existing resilience testing tools, Chaosd and ChaosToolkit, to understand differences in developer effort, workflow overhead, and required operational expertise. Quantitatively, we evaluated ChaosSpec in isolation by measuring execution time, stability, and CI feasibility across repeated runs. Together, these analyses provide a holistic understanding of how DORA compares to adjacent tooling and how it behaves when exercised as a standalone testing framework.

A. Research Questions

The primary research questions this paper explores are as follows:

RQ1: What are the execution costs and stability characteristics of running DORA-based resilience tests in continuous integration environments? This question examines the runtime behavior of ChaosSpec under CI conditions, including wall-clock duration, run duration of each example test and the frequency of nondeterministic test failures.

RQ2: How much developer effort is required to author resilience tests using DORA compared to adjacent tools? We compared the developer experience of authoring equivalent resilience scenarios using ChaosSpec, Chaosd, and

ChaosToolkit. For each tool, we implemented two representative fault scenarios starting from a minimal project setup.

B. Quantitative Analysis

DORA can be quantitatively evaluated by examining how its reference implementation, ChaosSpec, behaves when executed as part of a continuous integration pipeline. In this setting, the primary concern is whether a DORA based resilience suite is practical to run on shared CI infrastructure, and how its execution cost is distributed across different fault scenarios. This analysis directly addresses RQ1 by treating runtime, stability, and scenario level cost as proxies for execution feasibility.

To construct a realistic workload, we used the Library Service introduced in the Implementation section and assembled a ChaosSpec test suite that covers the main fault categories supported by the prototype. The suite includes one test for each supported fault type: network latency, service outage, CPU stress, and concurrency fault. Each test is written as a DORA style specification that declares services, faults, workloads, and assertions in a single Jest test file.

For the CI environment, we configured a GitHub Actions workflow that provisions Docker, installs Node.js and project dependencies, and runs only the ChaosSpec suite by selecting files that match the `.chaos.test.ts` naming convention. Each job starts a fresh `ubuntu-latest` runner, performs repository checkout and Docker setup, and then invokes Jest to execute all resilience tests. This workflow mirrors how a typical team would integrate ChaosSpec as a separate on demand job alongside their regular test suite.

During each workflow run, we collected two classes of measurements. At the CI level, we recorded the total job duration reported by GitHub Actions from job start to completion, which captures both environment setup time and ChaosSpec execution time. At the test level, we recorded the Jest reported duration for the overall suite and for each individual scenario, allowing us to break execution cost down by fault type. These values serve as our quantitative proxies for the runtime overhead introduced by DORA based resilience testing.

To account for normal variability on shared runners, we executed the same workflow ten times under identical configuration. This produced a small sample of repeated runs from which we computed descriptive statistics, such as minimum, maximum, and mean durations for both the CI job and each test scenario. We also examined how much of the total job time was spent in ChaosSpec itself compared to environment setup activities such as dependency installation and Docker initialization.

C. Qualitative Analysis

The qualitative analysis evaluates the developer effort involved in writing DORA-style resilience tests. To address RQ2, we compare the effort required to author equivalent scenarios using ChaosSpec, Chaosd, and ChaosToolkit.

We selected Chaosd and ChaosToolkit because both occupy the same space as DORA: they are general-purpose, free, and open-source resilience testing tools that do not position themselves as full chaos-engineering platforms. This makes them the most appropriate baselines for comparing the developer effort required to write resilience tests, since all three tools aim to support lightweight, developer-driven experimentation rather than organization-wide chaos programs.

For this comparison we used the same Library Service from the Implementation section and implemented two representative resilience scenarios with each tool: a network latency experiment that slows Redis while validating a latency SLO, and a CPU stress experiment that constrains and saturates the application's compute resources while asserting continued responsiveness. For each tool, the same researcher started from an empty folder containing only the Library Service and performed whatever setup was necessary to produce a fully passing resilience test for that scenario.

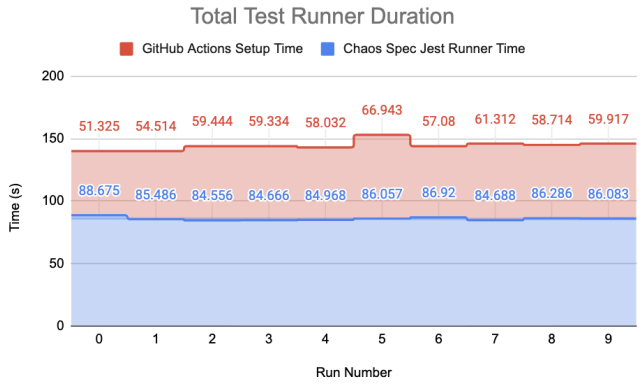
During each implementation session, we recorded several indicators of developer effort such as how many configuration files were required, how much custom scripting was necessary, and how many external tools had to be installed or orchestrated manually.

VIII. Evaluation Results

All evaluation data and code can be found in the ChaosSpec repository in the `evaluation` folder: https://github.com/MathyouMB/chaos-spec/tree/master/evaluation

A. Quantitative Results

The figure below summarizes the total duration of each CI job. The results show that overall job times remain tightly clustered, indicating that the test suite behaves consistently even on shared runners. The stability of these repeated executions suggests that DORA based resilience tests do not introduce significant nondeterminism into CI pipelines.

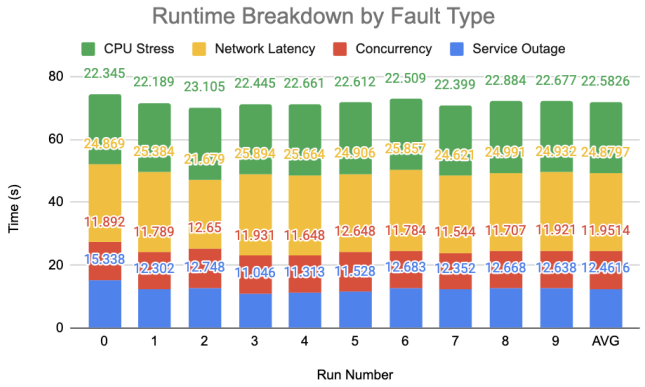


Across the ten CI executions, the total runner duration ranged from 140s to 153s, with a mean of 144.5s. The above figure illustrates that the ChaosSpec test suite accounts for the majority of total runtime, averaging 85.84 seconds per run, while GitHub Actions setup tasks (repository checkout, Docker initialization, Node setup, and dependency installation) take an additional 51–67 seconds, with a mean of 58.48 seconds.

The setup duration is noticeably more variable than the ChaosSpec execution time. Jest execution fluctuated within

a narrow band of roughly 84.5 to 88.7 seconds, which is a difference of less than five seconds. In contrast, setup time showed greater variability, ranging from 51.33 seconds at its fastest to 66.94 seconds at its slowest. This variation aligns with known sources of shared-runner nondeterminism, including container startup overhead and package installation caching effects. Despite this, ChaosSpec execution remains highly stable across runs, which allows overall CI time to remain predictable even when setup time fluctuates. We did not observe any test failures or flakes across the ten runs, suggesting that the suite is stable enough for repeated CI execution on shared runners, with variability dominated by normal GitHub Actions noise rather than test non-determinism.

To better understand where this time is spent, the figure below breaks down the Jest reported durations by fault type for each run.



Across all executions, Network Latency (μ = 24.88 s) and CPU Stress (μ = 22.58 s) consistently represented the largest share of execution time, while Service Outage (μ = 12.46 s) and Concurrency (μ = 11.95 s) accounted for smaller portions of the total budget. This behavior is expected since the latency test intentionally introduces delay into the request path, and the CPU stress test must maintain load long enough for performance assertions to become meaningful. By contrast, the service outage and concurrency scenarios complete more quickly and with less variance. All four scenarios were consistent in their run duration across the 10 runs.

Taken together, these quantitative results show that the DORA based suite is both stable and predictable in CI, and that the majority of execution time is contributed by fault types that inherently require sustained or time based workloads. This profile indicates that DORA is feasible to integrate into CI pipelines and provides a clear understanding of the cost associated with each category of resilience test.

B. Qualitative Results

Across tools, we observed substantial differences in the amount and type of work required.

Chaosd provided a direct command line interface for injecting faults including a network proxy, but left probing, orchestration, and cleanup entirely to the developer. This required the construction of a custom script capable of issuing HTTP requests, capturing response times, applying and removing faults, and coordinating the experiment lifecycle.

ChaosToolkit offered a more structured, declarative model for resilience scenarios, but still required environment preparation before an experiment could run. In the network latency case, for example, the developer had to edit Docker Compose, introduce a network proxy container, and reroute Redis traffic before the JSON experiment definition could reference these components. Although ChaosToolkit handled probing and teardown automatically once configured, the up-front configuration cost and verbosity of the experiment file created some friction.

ChaosSpec provided a markedly different authoring experience. All services, faults, probes, and traffic workloads were declared directly within a single test file, without modifying Docker Compose or invoking external scripts. The runtime automatically provisioned containers, attached proxies, injected faults, and performed cleanup, allowing the developer to focus solely on specifying system behavior rather than orchestrating infrastructure. This consolidation removed much of the operational burden found in the other tools. However, the current prototype is implemented as a Jest based framework and is therefore tied to the JavaScript ecosystem. This limitation is not inherent to DORA itself, but it does mean that the present implementation cannot be used natively in projects that rely on other languages or testing ecosystems without additional tooling or wrappers.

Capability	**Chaosd**	**ChaosToolkit**	**ChaosSpec**
Network Latency Fault Injection	✓	✓	✓
Hardware Stress Injection	✓	✓	✓
Automatic Network Proxy Orchestration	✓	✘	✓
Automatic service orchestration	✘	✘	✓ (fully abstracted from the developer)
Automatic probes & measurement	✘ (developer scripts)	✓	✓
Automatic cleanup & teardown	✘	✓ (teardown is configurable)	✓ (fully abstracted from the developer)
Ecosystem & Language Agnostic	✓	✓	✘ JS-only (Jest)
Overall Developer Effort	Highest (just a CLI, needs custom scripting)	Middle (still requires some manual orchestration)	Lowest

Overall, the qualitative analysis shows that DORA materially reduces the cognitive and operational overhead associated with authoring resilience tests. By abstracting network

configuration, orchestration, and cleanup into the runtime, ChaosSpec enables developers to express their tests at the level of service behavior and expected outcomes rather than infrastructure mechanics.

C. Answers to Research Questions

While we presented the detailed results in the previous section, we now highlight and summarize the answers to our research questions.

Answer to RQ1: *What are the execution costs and stability characteristics of running DORA-based resilience tests in continuous integration environments?* The quantitative analysis shows that DORA based resilience tests execute with stable and predictable runtimes across repeated CI runs. The total job time varies only slightly between executions, and the per scenario durations remain consistent for each fault type. Network latency and CPU stress account for the majority of execution time, which is expected given their time based workloads, while other scenarios complete more quickly. These results indicate that DORA tests introduce a manageable and well understood cost to CI pipelines and can be executed reliably without significant nondeterminism.

Answer to RQ2: *How much developer effort is required to author resilience tests using DORA compared to adjacent tools?* The qualitative comparison highlights substantial differences in the developer experience across tools. Chaosd requires developers to manually orchestrate probes, apply faults, measure performance, and handle cleanup, which increases effort and shifts focus toward mechanics rather than intent. ChaosToolkit provides a structured declarative model, but requires environment preparation and produces verbose experiment definitions. ChaosSpec reduces this overhead by integrating service definitions, fault injection, probing, and orchestration into a single test file, allowing developers to express resilience behavior directly. However, the current ChaosSpec prototype is tied to the JavaScript ecosystem via Jest, which limits language flexibility even though the DORA methodology itself is not bound to JavaScript. Overall, DORA substantially lowers the effort required to author resilience tests while maintaining expressiveness.

IX. Discussion

The DORA methodology reframes resilience testing from an infrastructure-driven activity into a developer-oriented testing workflow. By expressing faults, environments, and workloads declaratively within standard test definitions, DORA reduces the time and expertise needed to validate system behavior under failure. Instead of configuring proxies, orchestrators, and stress tools manually, developers define what should happen, and the DORA runtime automates how it happens.

The ChaosSpec prototype demonstrates successful DORA implementations of network faults, resource stress, service outages, and concurrency ordering scenarios; however, it does not demonstrate application-level exception injection or regional and environmental disruptions.

Importantly, DORA does not replace large-scale or production-facing chaos engineering platforms. It complements them. DORA excels in developer-controlled environments where the scope of faults is narrow and well defined. It is particularly effective at verifying behavior at the service or integration level, ensuring that local and pre-merge tests can surface issues long before deployment. Broader organizational goals such as region failovers and incident rehearsal remain within the domain of platform and operations teams. A balanced resilience strategy may use DORA for pre-deployment validation of localized failure modes, while using traditional chaos engineering for validating systemic and cross-service behaviors.

Beyond technical value, DORA also influences engineering culture. Co-locating resilience tests with functional and integration tests clarifies ownership: reliability becomes a shared responsibility of application developers, not just platform engineers. By using familiar syntax and workflows, DORA embeds resilience awareness directly into the development cycle, making fault tolerance an everyday engineering activity rather than an occasional exercise.

A. Limitations

Even with the improvements introduced by DORA, a number of practical constraints remain.

Implementation Complexity and Organizational Adoption: Building a production-quality DORA runtime requires substantial engineering investment and organizational commitment. Implementations must integrate with existing build systems, container orchestration platforms, and continuous integration pipelines, each of which varies widely across organizations. As a result, no single reference implementation can easily satisfy all infrastructure contexts. Open-source tooling could help reduce this barrier by providing common orchestration primitives, fault adapters, and runtime abstractions, but enterprise environments often impose unique security, compliance, or networking constraints that limit direct adoption. Building a DORA testing tool in practice therefore depends on company-specific engineering effort and sustained buy-in from platform and DevOps teams.

Data Seeding and Test State: Scenarios that depend on complex data initialization, such as multi-service fixtures, prepopulated caches, or synchronized clocks, can be difficult to encode directly within a declarative test. Capturing these states often requires custom setup scripts, versioned datasets, or specialized image entrypoints. Practical mitigations include supporting external configuration files, environment variables for dataset selection, or scenario flags consumed at service startup. However, moving initialization logic outside the test definition reduces readability and breaks the principle of co-located configuration, making it harder for developers to understand the complete scope of the experiment at a glance. These approaches also shift complexity into the build and boot processes, which can reintroduce some of the operational overhead that DORA aims to remove.

SLO External Validity: DORA evaluates service behavior within disposable, isolated test environments rather than the production infrastructure where service-level objectives are

measured and enforced. As a result, passing DORA assertions does not guarantee that those SLOs will hold under real operating conditions. Production factors such as multi-tenant contention, noisy neighbors, cross-zone and cross-region latency, autoscaling delays, load balancer routing policies, production data volumes, cache heat, background workloads, and organization-wide failover playbooks can all influence observed performance.

In contrast, chaos testing conducted in staging or production environments directly validates SLOs under realistic load and topology conditions. DORA complements these practices but operates at a different level of abstraction. Assertions within DORA tests can provide early feedback on resilience behavior, yet they may also create a false sense of completeness if interpreted as evidence of system-wide reliability. A test that meets a p95 latency target in a hermetic environment may still fail to capture regressions that emerge only under production traffic and scale.

To mitigate this, teams should treat DORA results as local indicators of service robustness rather than proofs of SLO compliance. Combining DORA with environment-aware validation, such as scheduled resilience drills or controlled chaos experiments in production, provides stronger assurance that reliability goals are truly met across deployment contexts.

Not a Replacement for Observability: DORA does not replace tracing, metrics, logging, or production-level experiments that reveal systemic issues and emergent behaviors. Its purpose is to validate specific resilience properties within isolated, reproducible environments, not to provide continuous insight into live systems. Future DORA implementations could extend support for observability by emitting standardized artifacts such as latency histograms, percentile distributions, and error timelines, enabling post-test analysis, comparison across runs, and integration with external monitoring tools.

Inherent Non-Determinism: Injected faults, scheduler variance, and container timing introduce flakiness. Preferred mitigations include using statistical thresholds such as percentiles and error-rate bounds, along with stabilized clocks, bounded retries, and time budgets, but some nondeterminism remains unavoidable.

Execution Time and Resource Cost: Spinning up containers, injecting faults, and driving traffic are slower and more resource-intensive than unit tests. Typical mitigations: isolate suites (e.g., test:chaos), run on demand, cap parallelism, and tune workloads. Even so, CI wall-time and runner contention can increase.

B. Future Work

While this paper demonstrates the feasibility and usability of DORA through a reference implementation, the next challenge is understanding how it affects actual developer behavior and real-world reliability outcomes.

One direction is to run developer-centered studies that compare DORA with existing resilience testing tools under controlled conditions. These studies could measure time to complete a scenario, number of mistakes made, perceived workload, ease of use, and overall success rates. Such measurements would help determine whether DORA reduces the cognitive and operational effort normally required to write resilience tests. These studies could also examine how developers respond to declarative fault definitions and to the automatic orchestration of services and environments.

Another direction is to evaluate DORA in real software teams over longer periods of time. This type of field study would reveal whether DORA leads to meaningful changes in actual development practice. Relevant observations include:

- Whether developers write more resilience tests after adopting DORA
- Whether teams begin covering a wider range of fault scenarios
- Whether on boarding new developers into resilience testing becomes easier
- Whether the frequency or severity of production incidents involving resilience failures decreases

Longer term deployments in real organizations would also reveal practical issues that may not be visible in short-term experiments, such as continuous integration cost, compatibility with existing tooling, and organizational constraints.

Another direction for future work could be to study how DORA fits within larger reliability programs. This includes exploring how DORA interacts with observability tools, service level objectives, and existing chaos engineering processes. It also includes studying whether DORA-based resilience tests influence how teams prepare for incidents, respond to failures, or perform postmortem analysis.

Overall, empirical studies are needed to determine whether DORA actually changes developer behavior and improves real-world resilience outcomes. These studies would validate whether a declarative and test-native approach results in more resilience tests being written and fewer resilience issues appearing in production.

X. Conclusion

This paper introduced DORA (Declarative test Orchestration for Resilient Applications), a developer-first methodology that reframes resilience testing as an ordinary part of software testing rather than a practice that depends on operational expertise or specialized infrastructure.

The ChaosSpec implementation demonstrated that this model can be implemented with familiar developer technologies and applied across a wide range of resilience scenarios.

The evaluation showed that DORA based tests run in a predictable and repeatable manner and that the effort required to author equivalent scenarios is substantially lower than with general purpose resilience tools. These findings suggest that developers are more likely to write and maintain resilience tests when the required abstractions are simple, local, and aligned with their existing workflows.

While limitations remain, including upfront engineering investment, data setup complexity, runtime cost, and infrastructure variability, DORA provides a concrete path toward making resilience testing more inclusive, efficient, and routine.

XI. Appendix

A. Network Latency Test Example

```
import {
 createService,
 timedHttpRequest,
} from "@chaosspec";

describe("when Redis experiences 500ms of network latency", () => {
 it("will still meet the SLO of 2000ms for a read request", async () => {
   const SLO_MS = 2000;

   // 1. Setup the services under test
   const redis = await createService("redis", {
     image: "redis:7.0-alpine",
     portToProxy: 6379,
   });
   const service = await createService("example-service", {
     image: "example-service:dev",
     portToExpose: 5000,
     environment: {
       REDIS_URL: `redis://${redis.getHost()}:${redis.getProxyPort()}`,
     },
   });

   // 2. Inject 500ms latency into Redis
   await redis.startNetworkLatency({ latencyMs: 500 });

   // 3. Send HTTP request and measure duration
   const serviceUrl = `http://localhost:${service.getMappedPort(5000)}/api`;
   const [response, duration] = await timedHttpRequest(serviceUrl, {
     method: "GET",
     headers: { "Content-Type": "application/json" },
   });

   // 4. Assert that the measured duration is within the SLO
   expect(response.ok).toBe(true);
   expect(duration).toBeLessThanOrEqual(SLO_MS);
 });
});
```

Lines 1–4: Import the required helpers from the ChaosSpec library. `createService` provisions containerized services within the isolated test network, while `timedHttpRequest` performs an HTTP request and returns both the response and its duration in milliseconds.

Lines 6–7: Define the test case using `describe` and `it`. The test scenario describes Redis experiencing 500 ms of network latency, and the expectation is that the application will still satisfy the 2000 ms service-level objective (SLO) for a read request.

Line 8: Declare the constant `SLO_MS = 2000`, setting the upper latency limit for successful operation.

Lines 11–14: Initialize the Redis service container using `createService`. The `image` field specifies the Redis version (`redis:7.0-alpine`), and `portToProxy: 6379` ensures all Redis communication passes through a fault-injectable proxy layer.

Lines 15–21: Create the application service container using `createService`. It runs the `example-service:dev` image, exposes port 5000, and defines an environment variable `REDIS_URL` that points to the proxied Redis endpoint. This configuration allows injected latency to propagate to the application naturally.

Lines 23–24: Inject the network fault by calling `redis.startNetworkLatency({ latencyMs: 500 })`. This command introduces a 500 ms artificial delay for all Redis traffic handled through the proxy.

Lines 26–31: Construct the request URL using `service.getMappedPort(5000)` so the host system can reach the running container. Then, perform a timed GET request with `timedHttpRequest`, capturing both the HTTP response object and total request duration.

Lines 33–35: Validate outcomes with assertions. `expect(response.ok).toBe(true)` confirms functional success, while `expect(duration).toBeLessThanOrEqual(SLO_MS)` ensures total latency remains within the defined 2000 ms SLO even under fault conditions.

Lines 36–37: Close the `it` and `describe` blocks, marking the end of the test case definition.

B. Service Outage Test Example

```
describe("when redis is completely unavailable", () => {
  it("will return a 500 error when trying to fetch books", async () => {
    // 1. Setup the services under test
    const redis = await createService("redis", {
      image: "redis:7.0-alpine",
      portToProxy: 6379,
    });
    const service = await createService("example-service", {
      image: EXAMPLE_SERVICE_IMAGE,
      portToExpose: 5000,
      environment: {
        REDIS_URL: `redis://${redis.getHost()}:${redis.getProxyPort()}`,
      },
    });

    // 2. Prevent the example service from connected to Redis
    await redis.startServiceOutage();

    // 3. Send HTTP request
    const serviceUrl = `http://localhost:${service.getMappedPort(5000)}/api/books`;
    const response = await fetch(serviceUrl, {
      method: "GET",
      headers: { "Content-Type": "application/json" },
    });

    // 4. Assert that response is a 500
    expect(response.ok).toBe(false);
    expect(response.status).toBe(500);
  });
});
```

Lines 1–2: Define the test scenario and expected outcome. The `describe` block declares the condition where Redis becomes completely unavailable, while the nested `it` block specifies that the service should respond with an HTTP 500 error when attempting to fetch data.

Lines 4–7: Create a Redis service container using `createService`. The container runs the `redis:7.0-alpine` image, and `portToProxy: 6379` ensures that Redis traffic passes through a proxy layer to allow fault injection.

Lines 8–13: Provision the application service container using `createService`. It runs `EXAMPLE_SERVICE_IMAGE`, exposes port `5000`, and includes an environment variable `REDIS_URL` that points to the Redis proxy endpoint. This configuration ensures that Redis faults propagate to the application as connection errors.

Line 17: Simulate a complete Redis outage using `redis.startServiceOutage()`. This operation prevents the application from establishing or maintaining any active connection to Redis, effectively emulating a dependency failure.

Lines 20–24: Build the target URL using the mapped application port and send an HTTP GET request to the `/api/books` endpoint. The `fetch` call simulates a client request, while the `Content-Type` header ensures a consistent JSON response format.

Lines 27–28: Validate outcomes with assertions. `expect(response.ok).toBe(false)` confirms that the request failed, and `expect(response.status).toBe(500)` verifies that the application responds with the proper 500 status code rather than timing out or returning invalid data.

Lines 29–30: Close the `it` and `describe` blocks, finalizing the dependency-fault test definition.

C. CPU Stress Fault Test Example

```
describe("when service is limited to 1 CPU core and 0.5GB memory, under 80% CPU load", () => {
  it("will respond within SLO while stressed", async () => {
    // 1. Setup the services under test
    const SLO_MS = 2000;
    const redis = await createService("redis", {
      image: "redis:7.0-alpine",
      portToProxy: 6379,
    });
    const service = await createService("example-service", {
```

```
      image: EXAMPLE_SERVICE_IMAGE,
      portToExpose: 5000,
      environment: {
        REDIS_URL: `redis://${redis.getHost()}:${redis.getProxyPort()}`,
      },
      resources: {
        cpuCores: 1,
        memoryGb: 0.5,
      },
    });

    // 2. Inject cpu stress
    await service.startCpuStress({ loadPercent: 80 });

    // 3. Send HTTP request
    const serviceUrl = `http://localhost:${service.getMappedPort(5000)}/api/books`;
    const [response, duration] = await timedHttpRequest(serviceUrl, {
      method: "GET",
      headers: { "Content-Type": "application/json" },
    });

    // 4. Assert that response is within SLO
    expect(response.ok).toBe(true);
    expect(duration).toBeLessThanOrEqual(SLO_MS);
  });
});
```

Lines 1–2: Define the test scenario and expected outcome. The `describe` block states that the example service is running with constrained CPU and memory under 80% CPU load, while the nested `it` block asserts that the service should still respond within the configured SLO.

Line 4: Declare `SLO_MS = 2000`, setting a 2000 ms upper bound for acceptable response time under stress.

Lines 5–8: Create a Redis container with `createService`. The container runs the `redis:7.0-alpine` image, and `portToProxy: 6379` routes Redis traffic through a proxy so other experiments could inject network faults if needed.

Lines 9–19: Create the application service container using `createService`. It runs `EXAMPLE_SERVICE_IMAGE`, exposes port `5000`, and sets `REDIS_URL` to point at the proxied Redis endpoint. The `resources` block constrains the container to a single CPU core (`cpuCores: 1`) and 0.5 GB of memory (`memoryGb: 0.5`), ensuring the test runs under predictable resource limits.

Lines 21–22: Inject CPU stress into the application container via `service.startCpuStress({ loadPercent: 80 })`. This call drives CPU utilization to roughly 80%, emulating a high-load scenario without changing the incoming traffic pattern.

Lines 24–29: Build the service URL using `service.getMappedPort(5000)` so the host test runner can reach the container's `/api/books` endpoint. Use `timedHttpRequest` to send a GET request and capture both the HTTP `response` and the total `duration` of the call in milliseconds.

Lines 31–33: Validate behavior under stress: `expect(response.ok).toBe(true)` checks that the request succeeds functionally, while `expect(duration).toBeLessThanOrEqual(SLO_MS)` ensures the response time stays within the 2000 ms SLO despite the CPU load.

Lines 34–35: Close the `it` and `describe` blocks, completing the CPU-stress resilience test definition.

D. Concurrency Ordering Fault Test Example

```
describe("when two users both try take out the last copy of a book simultaneously", () => {
  it("will inform at least one user that the book has already been taken", async () => {
    // 1. Setup the services under test
    const BOOK_WITH_ONE_COPY_ID = "book-with-one-copy-left";
    const redis = await createService("redis", {
      image: "redis:7.0-alpine",
    });

    const service = await createService("example-service", {
      image: EXAMPLE_SERVICE_IMAGE,
      portToExpose: 5000,
      environment: {
        REDIS_URL: `redis://${redis.getNetworkAlias()}:6379`,
      },
    });
```

```
16
17      // 2. Trigger two HTTP requests at the same time
18      const serviceUrl = `http://localhost:${service.getMappedPort(5000)}/api/loans`;
19      const [firstResponse, secondResponse] = await startSimultaneously([
20        () =>
21          fetch(serviceUrl, {
22            method: "POST",
23            body: JSON.stringify({ userId: "user-1", bookId: BOOK_WITH_ONE_COPY_ID }),
24            headers: { "Content-Type": "application/json" },
25          }),
26        () =>
27          fetch(serviceUrl, {
28            method: "POST",
29            body: JSON.stringify({ userId: "user-2", bookId: BOOK_WITH_ONE_COPY_ID }),
30            headers: { "Content-Type": "application/json" },
31          }),
32      ]);
33
34      // 3. Assert that one of the users was informed the book was no longer available.
35      const messages = [firstResponseJson.message, secondResponseJson.message];
36      expect(messages).toContain("Book is no longer available");
37
38    });
39  });
```

Lines 1–2: Define the test scenario and its expected outcome. The `describe` block frames a race for the last copy of a book; the `it` block asserts that at least one user is informed that the item is unavailable.

Line 4: Declare `BOOK_WITH_ONE_COPY_ID`, a fixture identifier that ensures contention over a single remaining item.

Lines 5–7: Start a Redis container with `createService("redis", { image: "redis:7.0-alpine" })`. This provides shared state the application will use to coordinate inventory.

Lines 9–15: Create the application service container. It runs `EXAMPLE_SERVICE_IMAGE`, exposes port `5000`, and sets `REDIS_URL` to `redis://${redis.getNetworkAlias()}:6379`, using the in-network alias so the app can reach Redis inside the isolated test network.

Line 18: Build the public URL for the loans endpoint using `service.getMappedPort(5000)` so the host test runner can reach the container.

Lines 19–31: Invoke `startSimultaneously([...])` with two functions that each perform a `POST` to `/api/loans`. Both bodies encode `{ userId: "...", bookId: BOOK_WITH_ONE_COPY_ID }`, creating concurrent attempts to loan the same last item. The helper coordinates the start time to maximize the chance of exposing races (lost updates, missing locks, etc.). The result is a pair of `Response` objects: `[firstResponse, secondResponse]`.

Lines 33–35: Aggregate user-facing messages and assert that at least one response indicates the item is no longer available: `expect(messages).toContain("Book is no longer available");`. (Implementation note: parse each `Response` to JSON before reading `.message`, for example `const firstResponseJson = await firstResponse.json()`.)

Lines 36–37: Close the `it` and `describe` blocks, completing the concurrency-fault test.

E. Load Testing Example

```
1  describe("when redis experiences 500ms latency", () => {
2    it("will maintain p95 ≤ SLO for 500 simultaneous reads", async () => {
3      // 1. Setup the services under test
4      const SLO_MS = 2000;
5      const redis = await createService("redis", {
6        image: "redis:7.0-alpine",
7        portToProxy: 6379,
8      });
9      const service = await createService("example-service", {
10       image: EXAMPLE_SERVICE_IMAGE,
11       portToExpose: 5000,
12       environment: {
13         REDIS_URL: `redis://${redis.getHost()}:${redis.getProxyPort()}`,
14       },
15     });
16
```

```
    // 2. Inject 500ms latency into Redis
    await redis.startNetworkLatency({ latencyMs: 500 });

    // 3. Send 500 HTTP requests over 1000ms
    const serviceUrl = `http://localhost:${service.getMappedPort(5000)}/api/books`;
    const experiment = await startTrafficExperiment(
      () => fetch(serviceUrl, { method: "GET" }),
      {
        count: 500,                  // total requests
        strategy: strategies.even,   // evenly fire the requests over window
        spanMs: 1000,                // spread requests over 1s window
      },
    );

    // 4. Assert p95 was within SLO and error rate is below 2%
    expect(experiment.errorRate()).toBeLessThanOrEqual(0.02);
    expect(experiment.p95()).toBeLessThanOrEqual(SLO_MS);
  });
});
```

Lines 1–2: Define the scenario and goal. `describe` states Redis has 500 ms latency. `it` asserts that under this condition the application maintains p95 latency within the SLO for 500 simultaneous reads.

Line 4: Declare `SLO_MS = 2000` as the latency objective for this test.

Lines 5–8: Create a Redis container with `createService`. Uses image `redis:7.0-alpine` and `portToProxy: 6379` so traffic is routed through a proxy for fault injection.

Lines 9–15: Create the application service container. Runs `EXAMPLE_SERVICE_IMAGE`, exposes port `5000`, and sets `REDIS_URL` to the Redis **proxy** endpoint so injected latency influences the app's calls.

Line 18: Apply the network fault: `redis.startNetworkLatency({ latencyMs: 500 })`.

Line 21: Build the public URL for `/api/books` using `service.getMappedPort(5000)` so the host test runner can reach the container.

Lines 22–29: Start a traffic experiment that generates concurrent requests: `startTrafficExperiment(() => fetch(serviceUrl, { method: "GET" }), { ... })`

- `count: 500` sets total requests to 500.
- `strategy: strategies.even` spreads requests evenly across the window.
- `spanMs: 1000` schedules them across a 1 second window.

The returned `experiment` aggregates results and exposes latency percentiles and error statistics.

Lines 32–34: Validate error rate and tail latency: `experiment.errorRate()` must be $\leq$ 0.02 (2 percent). `experiment.p95()` must be $\leq$ `SLO_MS` (2000 ms).

Line 35: Close the `it` and `describe` blocks, completing the load-test specification.

F. GitHub Actions Example

```
name: Manual ChaosSpec CI Workflow
on:
  workflow_dispatch: {}
jobs:
  chaos-tests:
    runs-on: ubuntu-latest
    timeout-minutes: 60
    permissions:
      contents: read
    steps:
      - name: Checkout
        uses: actions/checkout@v4
      - name: EnsureDockerInstalledRunning
        shell: bash
        run: |
          if ! command -v docker >/dev/null 2>&1; then
            sudo apt-get update
            sudo apt-get install -y docker.io
            sudo systemctl enable --now docker
          else
```

```
21              sudo systemctl start docker || true
22            fi
23            docker --version
24            sudo docker info
25        - name: SetupNode
26          uses: actions/setup-node@v4
27          with:
28            node-version: '20'
29            cache: 'npm'
30            cache-dependency-path: package-lock.json
31        - name: InstallDeps
32          run: npm ci
33        - name: RunChaosSpecs
34          env:
35            CI: true
36          run: npm run test:chaos
```

Lines 1–2: Define the workflow name (`Manual ChaosSpec CI Workflow`) and its trigger (`workflow_dispatch`). `workflow_dispatch` makes the workflow manually triggered, ensuring long-running resilience tests are executed only on demand.

Lines 4–5: Begin the `jobs` section and declare a job named `chaos-tests`. Each job runs in isolation within its own GitHub Actions virtual environment.

Lines 6–7: Specify the environment: `runs-on: ubuntu-latest` sets the base runner image. `timeout-minutes: 60` limits total job execution time to one hour to prevent indefinite runs.

Line 8: Define permissions for the job. `contents: read` grants minimal read-only repository access, sufficient for checkout.

Lines 11–12: First step: check out the repository using `actions/checkout@v4`. This retrieves the project's code so later steps can access tests, scripts, and configuration files.

Line 13: Second step: declare a custom step named `EnsureDockerInstalledRunning`. This ensures Docker is present and active before any container-based testing begins.

Lines 14–15: Specify that the command should run in a Bash shell. `run: |` starts a multi-line script block.

Lines 16–23: Conditional installation logic for Docker:

- The `if` block checks whether Docker is already installed.
- If missing, it runs `apt-get update`, installs `docker.io`, enables and starts the service.
- If Docker exists, it attempts to start the daemon (`sudo systemctl start docker || true`) to guarantee it is running.

Lines 24–25: Verification commands: `docker --version` prints the Docker version, and `sudo docker info` confirms the daemon is responsive and properly configured.

Line 26: Next step: `SetupNode` — configures the Node.js environment using `actions/setup-node@v4`.

Lines 27–29: Specify configuration options for the Node.js setup:

- `node-version: '20'` installs Node 20.
- `cache: 'npm'` enables dependency caching.
- `cache-dependency-path: package-lock.json` ties the cache to dependency versions.

Line 30: New step: `InstallDeps`, installing project dependencies.

Line 31: Run command `npm ci` to install dependencies exactly as specified in `package-lock.json`, ensuring reproducible builds.

Line 32: Begin the `RunChaosSpecs` step — executes the DORA/ChaosSpec test suite.

Lines 33–35: Set the `CI` environment variable to `true` to indicate a CI environment. `run: npm run test:chaos` executes the resilience tests defined in `package.json` that match the `.chaos.test.ts` naming pattern.

Line 36: End of workflow definition. This complete job provisions Docker, sets up Node.js, installs dependencies, and executes ChaosSpec tests manually via GitHub Actions.

References

[1] A. Rotem-Gal-Oz, "Fallacies of Distributed Computing Explained," *Doctor Dobbs Journal*, p. , 2008, [Online]. Available: https://www.researchgate.net/publication/322500050_Fallacies_of_Distributed_Computing_Explained

[2] P. Alvaro and S. Tymon, "Abstracting the geniuses away from failure testing," *Commun. ACM*, vol. 61, no. 1, pp. 54–61, Dec. 2017, doi: 10.1145/3152483.

[3] J. Owotogbe, I. Kumara, W.-J. V. D. Heuvel, and D. A. Tamburri, "Chaos Engineering: A Multi-Vocal Literature Review." [Online]. Available: https://arxiv.org/abs/2412.01416

[4] B. Treynor, M. Dahlin, V. Rau, and B. Beyer, "The calculus of service availability," *Commun. ACM*, vol. 60, no. 9, pp. 42–47, Aug. 2017, doi: 10.1145/3080202.

[5] S. L. Dorton, G. J. Lematta, and K. J. Neville, "The Tough Sell of Resilience Engineering," *IEEE Transactions on Technology and Society*, vol. 6, no. 1, pp. 47–53, 2025, doi: 10.1109/TTS.2024.3484176.

[6] L. Petersson, "An Empirical Investigation of the Acceptance of Chaos Engineering," Master's thesis, 2022. [Online]. Available: https://www.diva-portal.org/smash/get/diva2%3A1684983/FULLTEXT01.pdf

[7] R. R. Karn, R. Das, D. R. Pant, J. Heikkonen, and R. Kanth, "Automated Testing and Resilience of Microservice's Network-link using Istio Service Mesh," in *2022 31st Conference of Open Innovations Association (FRUCT)*, 2022, pp. 79–88. doi: 10.23919/FRUCT54823.2022.9770890.

[8] P. K. Adapala, "Integrating Chaos Engineering with CI/CD for Resilience Testing," *International Journal of Innovative Research in Computer and Communication Engineering*, vol. 10, no. 11, 2022, doi: 10.15680/IJIRCCE.2022.1011053.

[9] L. Feinbube, L. Pirl, and A. Polze, "Software Fault Injection: A Practical Perspective," in *Dependability Engineering*, F. P. G. Márquez and M. Papaelias, Eds., London: IntechOpen, 2017. doi: 10.5772/intechopen.70427.

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

[11] F. Fagerholm and J. Münch, "Developer Experience: Concept and Definition," in *Proceedings of the 2012 International Conference on Software and System Process (ICSSP)*, 2012, pp. 73–77. doi: 10.1109/ICSSP.2012.6225984.

[12] M. A. Rodriguez and R. Buyya, "Container-based cluster orchestration systems: A taxonomy and future directions," *Software: Practice and Experience*, vol. 49, no. 5, pp. 698–719, 2019, doi: https://doi.org/10.1002/spe.2660.

[13] D. Malandrino and V. Scarano, "A TAXONOMY OF PROGRAMMABLE HTTP PROXIES FOR ADVANCED EDGE SERVICES," in *Proceedings of the First International Conference on Web Information Systems and Technologies - WEBIST*, SciTePress, 2005, pp. 231–238. doi: 10.5220/0001233802310238.

[14] P. Muzumdar, A. Bhosale, G. P. Basyal, and G. Kurian, "Navigating the Docker Ecosystem: A Comprehensive Taxonomy and Survey," *Asian Journal of Research in Computer Science*, vol. 17, no. 1, pp. 42–61, Jan. 2024, doi: 10.9734/ajrcos/2024/v17i1411.

[15] M.-N. Tran, D.-D. Vu, and Y. Kim, "A Survey of Autoscaling in Kubernetes," in *2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN)*, 2022, pp. 263–265. doi: 10.1109/ICUFN55119.2022.9829572.

[16] K. Rzadca *et al.*, "Autopilot: workload autoscaling at Google," in *Proceedings of the Fifteenth European Conference on Computer Systems*, in EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. doi: 10.1145/3342195.3387524.

[17] S. Chen, X. Tang, H. Wang, H. Zhao, and M. Guo, "Towards Scalable and Reliable In-Memory Storage System: A Case Study with Redis," in *2016 IEEE Trustcom/BigDataSE/ISPA*, 2016, pp. 1660–1667. doi: 10.1109/TrustCom.2016.0255.

[18] T. Kinsman, M. Wessel, M. A. Gerosa, and C. Treude, "How Do Software Developers Use GitHub Actions to Automate Their Workflows?," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 420–431. doi: 10.1109/MSR52588.2021.00054.

[19] Netflix TechBlog, "The Netflix Simian Army." [Online]. Available: http://techblog.netflix.com/2011/07/netflix-simian-army.html

[20] P. Alvaro, K. Andrus, C. Sanden, C. Rosenthal, A. Basiri, and L. Hochstein, "Automating Failure Testing Research at Internet Scale," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, in SoCC '16. Santa Clara, CA, USA: Association for Computing Machinery, 2016, pp. 17–28. doi: 10.1145/2987550.2987555.

[21] P. Alvaro, J. Rosen, and J. M. Hellerstein, "Lineage-driven Fault Injection," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, in SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 331–346. doi: 10.1145/2723372.2723711.

[22] H. S. Gunawi *et al.*, "FATE and DESTINI: A Framework for Cloud Recovery Testing," in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA: USENIX Association, Mar. 2011. [Online]. Available: https://www.usenix.org/conference/nsdi11/fate-and-destini-framework-cloud-recovery-testing

[23] C. S. Meiklejohn, A. Estrada, Y. Song, H. Miller, and R. Padhye, "Service-Level Fault Injection Testing," in *Proceedings of the ACM Symposium on Cloud Computing*, in SoCC '21. Seattle, WA, USA: Association for Computing Machinery, 2021, pp. 388–402. doi: 10.1145/3472883.3487005.

[24] C. Camacho, P. C. Cañizares, L. Llana, and A. Núñez, "Chaos as a Software Product Line—A platform for improving open hybrid-cloud systems resiliency," *Software: Practice and Experience*, vol. 52, no. 7, pp. 1581–1614, 2022, doi: https://doi.org/10.1002/spe.3076.

[25] C. Krueger, "Eliminating the adoption barrier," *IEEE Software*, vol. 19, no. 4, pp. 29–31, 2002, doi: 10.1109/MS.2002.1020284.