Reducing Adoption Barriers in Resilience Testing Through Enhanced Developer Experience

Honours Project Proposal

Matthew MacRae-Bovell

(student number redacted)

August 27th 2025

Motivation

Modern software systems increasingly rely on distributed architectures [1], where individual services interact over networks that are inherently unreliable [2]. While these systems provide scalability and flexibility, they are also highly vulnerable to faults such as latency, packet loss, disconnections, concurrency issues, and more.

Resiliency testing is the process of intentionally injecting such faults to evaluate whether a system can maintain stability and recover gracefully [3]. Despite its importance, current resiliency testing practices are often manual, infrastructure-heavy, and disconnected from the workflows most developers use. Many existing tools (e.g., Gremlin, Chaos Mesh) [4] require specialized knowledge or cloud-native integration, making them accessible primarily to Site Reliability Engineers or platform teams.

Additionally, existing tools rarely support simple, assertion-based tests, the core model developers are already familiar with when writing unit, integration, and e2e tests. As a result, resilience testing remains disconnected from standard development practices and is difficult to include in day-to-day workflows or continuous integration pipelines.

I believe that if resiliency testing could align more closely with the developer experience [5] already used for automated testing, it would lower the barrier to entry and encourage much wider adoption.

Project Scope

This project aims to examine the shortcomings in the current developer experience of resilience testing and determine how they can be addressed to make the practice more accessible and effective for developers [6].

The key deliverable of the project will be a prototype test framework that would allow developers to create simple automated assertion-based resiliency tests for distributed systems. This framework will serve as a practical application of improving the developer experience in resilience testing and could be compared against existing work in the field.

What the project will address:

- Framing resiliency testing as a developer experience problem rather than purely an infrastructure or operations concern.
- Identifying and studying a representative subset of distributed system failure modes (e.g., latency, packet loss, service unavailability, and simple concurrency conflicts) that can be meaningfully expressed through assertions.
- Comparing existing approaches in order to identify gaps and trade offs in terms of accessibility, reproducibility, and adoption.

- Producing a prototype framework as a deliverable to demonstrate and validate findings.

What the project will not address:

- It will not attempt to provide a comprehensive taxonomy of distributed system failures or fully generalize resiliency testing across all contexts.
- It will not replace or compete directly with large-scale chaos engineering platforms (e.g., Gremlin, Chaos Mesh) designed for production environments.
- It will not contribute new theoretical or mathematical models of resiliency. Instead, it complements formal work by investigating the practical developer experience of implementing resiliency testing.

Points of comparison:

The findings of this project will be evaluated against these two existing approaches:

1. Theoretical models of resiliency: Prior research on latency, packet loss, consensus, and fault tolerance in the form of mathematical models or formal proofs.

2. Existing chaos engineering tools: Tools such as Gremlin and Chaos Mesh implementing resiliency testing at scale in production environments.

Deliverables

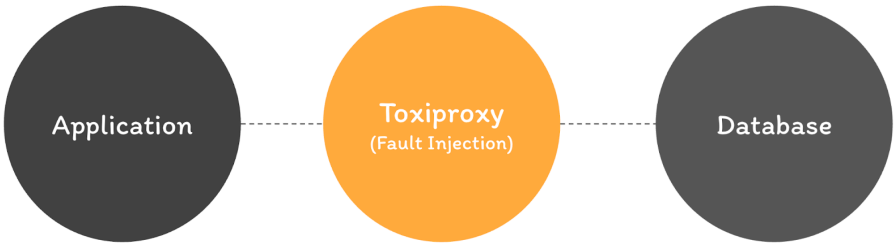
1. The Test Framework / Library

One of the outcomes of this project will be a prototype testing framework that would allow developers to create simple automated assertion-based resiliency tests for distributed systems.

Each test would expect the developer to define:

- The environment and services under test,
- The faults to inject (e.g., latency, packet loss),
- The test trigger (e.g., HTTP request, system event),
- and the expected outcomes or assertions.

The testing framework would use Toxiproxy (an open source tool for simulating network and system conditions) to inject network-level faults between services, simulating real-world conditions like latency or disconnections.



Once a fault is injected, the system is triggered by the defined action, and ChaosSpec would assert if the system behaved as expected.

The project would also leverage Testcontainers to manage the creation and cleanup of test environments, ensuring that each test runs on isolated services and infrastructure.

Additionally, the testing framework will offer utilities to support hard-to-test scenarios closely related to system robustness, such as concurrency correctness. This includes the two specific cases identified during feedback with Professor Corriveau: (1) two simultaneous requests for the last available item, and (2) three simultaneous requests for the last two available items.

To validate this is possible with this set of technologies, I've created a working prototype with an example test that demonstrates the usage of Toxiproxy and Testcontainers.

Below is a hypothetical example of the kind of test the final product would enable:

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

describe("when payment service is completely unavailable", () => {
  it("will trigger circuit breaker and return a 503 response", async () => {
    const payments = await createService("payments", {
      image: "myorg/payment-service:latest",
      proxy: true,
      ports: [8080],
    });

    const app = await createService("app", {
      image: "myorg/web-app:latest",
      environment: {
        PAYMENTS_URL: payments.url(8080),
      },
      ports: [3000],
    });

    await injectServiceOutage(payments);

    const response = await fetch(`${app.url(3000)}/checkout`, {
      method: "POST",
    });

    expect(response.status).toBe(503);
  });
});
```

The library would also include utilities to combine resiliency scenarios with other hard-to-test conditions, such as concurrency correctness.

For example, here's what a test could look like when two users request the last item while the database is experiencing latency:

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

describe("when two users request the last item under DB latency", () => {
  it("will allow only one checkout to succeed and respond within limits", async () => {
    const db = await createService("db", {
      image: "postgres:16-alpine",
      proxy: true,
      ports: [5432],
    });

    const app = await createService("app", {
      image: "myorg/app:latest",
      environment: {
        DATABASE_URL: db.url(5432),
        CHECKOUT_TIMEOUT_MS: "2000",
      },
      ports: [3000],
    });

    // Simulate a slow database
    await injectLatency("db", 1200);

    const [a, b] = await concurrent([
      () =>
        timedHttpRequest({
          method: "POST",
          url: `${app.url(3000)}/checkout`,
          headers: { "Content-Type": "application/json" },
          body: { bookId: 123, userId: "A" },
        }),
      () =>
        timedHttpRequest({
          method: "POST",
          url: `${app.url(3000)}/checkout`,
          headers: { "Content-Type": "application/json" },
          body: { bookId: 123, userId: "B" },
        }),
    ]).startSimultaneously();

    // Exactly one success despite degraded conditions
    const statuses = [a.response.status, b.response.status];
    expect(statuses.filter((s) => s === 200).length).toBe(1);
    expect(statuses.filter((s) => s === 409 || s === 423).length).toBe(1);

    // Resiliency assertion: requests still complete within the app's timeout budget
    expect(a.duration).toBeLessThanOrEqual(2000);
    expect(b.duration).toBeLessThanOrEqual(2000);
  });
});
```

2. Example System Repository

A GitHub repository containing:

- A simple intentionally fault-prone distributed system
- A collection of example tests targeting the system
- Demonstrations of typical failure scenarios and assertions

This will serve both as a proof-of-concept and as documentation for how the test framework is intended to be used.

3. Developer-Facing White Paper

A separate white paper to summarize the test framework's purpose and usage for a broader developer audience. This is not the project report itself but a companion artifact suitable for sharing within the software engineering community.

Schedule

Period	***Milestones***
Sept 7th - Sept 21st	**Goal:** Background Familiarization ● Survey existing chaos testing tools (Chaos Mesh, Gremlin, Toxiproxy). ● Review theoretical work and formal methods related to resiliency ● Catalogue common distributed system failure modes. ● Identify key resiliency properties developers aim to test (availability, retries, timeouts).
Sept 21st - Oct 2nd	**Goal:** Initial Prototype Development ● Implement a minimal prototype of the testing framework using Toxiproxy and Testcontainers. ● Create a first simple resiliency test (e.g., inject latency, assert response time). ● Begin designing an intentionally fault-prone distributed system to serve as a test target.
Oct 2nd - Oct 16th	**Goal:** Support Multiple Faults ● Extend prototype to handle multiple fault types (latency, service unavailability) ● Implement utilities for developer-facing assertions. ● Write initial tests against the example system. ● Begin recording strengths/weaknesses compared to existing tools.
Oct 16th - Oct 30th	**Goal:** Concurrency Testing and Mid-Project Checkpoint ● Add concurrency correctness scenarios (e.g., multiple simultaneous checkouts).

	• Mid-term checkpoint with supervisor: review progress, refine scope if necessary.
Oct 30th - Nov 9th	**Goal:** Polishing Prototype and Drafting Deliverables • Expand example system repository with multiple failure scenarios. • Prepare draft of the project report for early feedback
Nov 9th - Nov 23rd	**Goal:** Testing, Analysis & Report Writing • Incorporate feedback from supervisor into project report • Continue working on test framework implementation
Nov 23rd - Dec 7th	**Goal:** Testing, Analysis & Report Writing • Run and document end-to-end example tests. • Evaluate framework against original objectives (ease of use, reproducibility, adoption potential). • Finalize written project report and package deliverables (framework, example system, documentation)
Dec 7th - Dec 21st	**Goal:** Submit Final Project Submission

References

[1] CNCF. 2023. *CNCF Annual Survey 2023*. Cloud Native Computing Foundation, 2023. Retrieved from https://www.cncf.io/reports/cncf-annual-survey-2023/

[2] D. P. Reed. 2018. *Fallacies of Distributed Computing Explained*. ResearchGate. Retrieved from https://www.researchgate.net/publication/322500050_Fallacies_of_Distributed_Computing_Explained

[3] A. Nagarajan and A. Vaddadi, "Automated Fault-Tolerance Testing," 2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Chicago, IL, USA, 2016, pp. 275-276, https://ieeexplore.ieee.org/document/7528973

[4] Gremlin, Inc. 2020. “Bring Chaos Engineering to your CI/CD pipeline.” Retrieved from https://www.gremlin.com/blog/bring-chaos-engineering-to-your-ci-cd-pipeline

[5] Fabian Fagerholm and Jürgen Münch. 2012. Developer experience: Concept and definition. In *Proceedings of the 2012 International Conference on Software and System Process* (ICSSP ’12). IEEE Press, Zurich, Switzerland, 73–77. https://ieeexplore.ieee.org/document/6225984

[6] Peter Alvaro and Séverine Tymon. 2017. Abstracting the geniuses away from failure testing. *Queue* 15, 5 (October 2017), 29–53. https://dl-acm-org.proxy.library.carleton.ca/doi/10.1145/3155112.3155114