PRJ-DOP-025: Chaos Engineering with Full-Stack Observability

Certification: AWS Certified DevOps Engineer – Professional
Domain: Resiliency and Chaos Engineering

---

1. Project Overview

This project demonstrates how to proactively improve application resilience by combining **full-stack observability** with **chaos engineering**. Traditional testing often misses complex, real-world failure scenarios. Chaos engineering is the practice of intentionally injecting controlled faults into a system to identify weaknesses before they cause production outages. To do this safely and effectively, you need deep visibility into your system's behavior—this is where observability comes in.

We will build a complete observability stack using **OpenTelemetry**, sending traces to **AWS X-Ray**, metrics to **Amazon Managed Prometheus**, and logs to **CloudWatch**. With this in place, we will use the **AWS Fault Injection Simulator (FIS)** to run chaos experiments, such as terminating EC2 instances or throttling database performance. We will then analyze the impact of these faults on our system using our observability dashboards to validate our system's resilience and identify areas for improvement.

Key Objectives

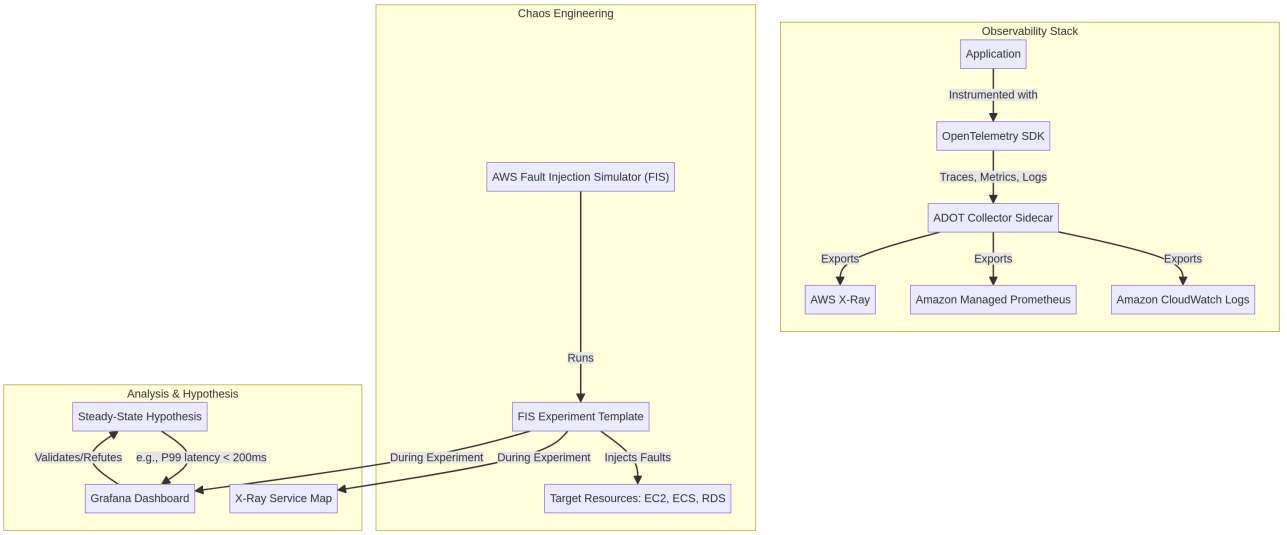
- Instrument an application with the OpenTelemetry SDK to generate traces, metrics, and logs.
- Configure the AWS Distro for OpenTelemetry (ADOT) Collector to send telemetry data to multiple AWS backend services (X-Ray, AMP, CloudWatch).
- Establish a baseline of the system's "steady-state" behavior using Grafana dashboards and X-Ray service maps.
- Create an AWS Fault Injection Simulator (FIS) experiment template to inject faults into the application environment.

- Run a chaos experiment and observe the system's response in real-time.
- Analyze the results to validate or refute a hypothesis about the system's resilience.

---

2. Architecture

The architecture combines a comprehensive observability pipeline with the fault injection capabilities of AWS FIS.



Architectural Components:

1. **Full-Stack Observability:**
    - The application is instrumented with the **OpenTelemetry SDK**, which generates the "three pillars" of observability: traces, metrics, and logs.
    - An **ADOT Collector**, running as a sidecar, receives this data.
    - The collector is configured with a pipeline that processes and exports the data to the appropriate AWS services:
        - **Traces** are sent to **AWS X-Ray** for end-to-end request tracing and service map visualization.
        - **Metrics** are sent to **Amazon Managed Prometheus (AMP)** for time-series analysis.
        - **Logs** are sent to **Amazon CloudWatch Logs**.

- **Amazon Managed Grafana (AMG)** is used to create a unified dashboard, visualizing metrics from AMP and logs from CloudWatch.

2. **Chaos Engineering (AWS FIS):**
    - **Hypothesis:** We start by defining a hypothesis about our system's resilience. For example: "If one of the three EC2 instances in the web tier fails, the P99 latency will not exceed 200ms, and there will be no increase in 5xx errors."
    - **FIS Experiment Template:** We create a template in AWS FIS that defines the chaos experiment. The template specifies:
        - **Targets:** The resources to inject faults into (e.g., a specific Auto Scaling Group, an RDS database).
        - **Actions:** The faults to inject (e.g., `aws:ec2:terminate-instances`, `aws:rds:reboot-db-instances`).
        - **Stop Conditions:** A critical safety mechanism. We define a CloudWatch alarm that, if triggered, will immediately stop the experiment. For example, an alarm that fires if the overall 5xx error rate exceeds 1%.

3. **Analysis:**
    - We run the FIS experiment.
    - During the experiment, we monitor our Grafana dashboards and X-Ray service map.
    - We observe how the system reacts to the fault. Does the Auto Scaling Group correctly replace the terminated instance? Does the load balancer reroute traffic? Does latency spike? Do errors occur?
    - Based on the observations, we can **validate** our hypothesis (the system behaved as expected) or **refute** it (the system did not handle the fault gracefully), revealing a weakness that needs to be fixed.

---

3. Prerequisites

- An AWS account with administrative permissions.

- An application deployed on AWS (e.g., on an EC2 Auto Scaling Group).
- A configured observability stack with AMP and AMG (you can use the one from PRJ-DOP-023).

---

4. Step-by-Step Implementation Guide

Step 4.1: Establish the Steady-State Baseline

Before injecting faults, you must understand what "normal" looks like for your system.

1. **Instrument Your Application:** Ensure your application is fully instrumented with OpenTelemetry and is sending traces, metrics, and logs to your observability backend.
2. **Build a Resilience Dashboard:** In Grafana, create a dashboard that shows the key performance indicators (KPIs) of your application. This should include:
    - Request rate (requests per second).
    - Error rate (percentage of 5xx errors).
    - Latency (average, P95, P99).
    - Resource utilization (CPU, memory) of your servers.
3. **Define Your Hypothesis:** Based on this dashboard, define a clear, measurable hypothesis. **Example Hypothesis:** "If 33% of the ECS tasks in the web service are terminated, the user-facing P99 latency will remain below 300ms, and the error rate will not exceed 0.5%."

Step 4.2: Create the FIS Stop Condition

This is the most important safety mechanism.

1. Go to the **CloudWatch Console -> Alarms -> Create alarm**.
2. Create an alarm based on a key business metric. Using our hypothesis, we would create an alarm that triggers if the `P99 Latency > 300ms` or `Error Rate > 0.5%`.
3. Name this alarm `Chaos-Experiment-Stop-Condition`.

Step 4.3: Create the FIS Experiment Template

1. Go to the **AWS Fault Injection Simulator console -> Experiment templates -> Create experiment template**.
2. **Description:** Describe your hypothesis.
3. **IAM role:** Create a new IAM role that gives FIS permission to perform the actions you will define (e.g., `ec2:TerminateInstances`).
4. **Actions:**
    - **Add action**.
    - **Name:** `Terminate-One-Third-Of-Instances`
    - **Action type:** `aws:ec2:terminate-instances`
    - **Targets:** Click **Edit** on the `Instances-Target-1`.
        - **Target method:** Resource tags and filters.
        - Select the tag that identifies your application's Auto Scaling Group.
        - **Selection mode:** Count
        - **Number of resources:** 1 (assuming you have 3 instances).
    - Save the action.
5. **Stop conditions:**
    - **Add stop condition**.
    - Select the `Chaos-Experiment-Stop-Condition` CloudWatch alarm you created.
6. Create the experiment template.

Step 4.4: Run the Experiment and Observe

1. **Start the experiment:** Select your template and click **Start experiment**.
2. **Monitor your dashboards:** Have your Grafana resilience dashboard and your X-Ray service map open on your screen.
3. **Observe the impact:**

- You will see the FIS action begin. In the EC2 console, you will see one of your instances move to the `shutting-down` state.
- On your Grafana dashboard, you should see:
  - The number of healthy hosts in your load balancer drop from 3 to 2.
  - A potential brief spike in latency or errors as traffic is rerouted.
  - The Auto Scaling Group should automatically start launching a new instance to replace the terminated one.
  - The number of healthy hosts should return to 3.
- On your X-Ray service map, you can see if any requests are failing and pinpoint the exact component causing the issue.

Step 4.5: Analyze the Results

- **Did the P99 latency stay below 300ms?**
- **Did the error rate stay below 0.5%?**
- **Did the Auto Scaling Group behave as expected?**

If the answer to all these questions is yes, your hypothesis is **validated**. Your system is resilient to this specific type of failure.

If the latency spiked to 500ms or the error rate hit 5%, your hypothesis is **refuted**. This reveals a weakness. Perhaps your remaining two instances couldn't handle the load, or your health checks were too slow. You now have a concrete data point to justify improving your system (e.g., by adjusting your Auto Scaling policy or using larger instance types).

---

5. Expanding the Experiments

Chaos engineering is a continuous process. You should build a library of experiments that test different failure modes:

- **API Throttling:** Use FIS to inject API throttling errors to see how your application handles them. Does it have a retry mechanism with exponential backoff?

- **Latency Injection:** Inject latency between services to simulate network degradation.
- **Database Failure:** Use the `aws:rds:reboot-db-instances` action to test how your application handles a database reboot.
- **Dependency Failure:** If your application calls a third-party API, use a tool to block access to that API to see if your application degrades gracefully.

---

6. Cleanup

1. **Delete the FIS experiment template.**
2. **Delete the CloudWatch alarm** used as the stop condition.
3. Clean up the resources from your observability stack (AMP, AMG, etc.) if desired.
4. Ensure your application's Auto Scaling Group has returned to its normal, normal, healthy state.

Business Context

The Problem

Development teams face slow deployment cycles, manual testing bottlenecks, and security vulnerabilities introduced during the CI/CD process. Traditional deployment methods lack security scanning, leading to vulnerabilities reaching production. Manual processes create inconsistency and human error.

The Solution

Secure CI/CD pipeline with integrated security scanning, automated testing, and infrastructure as code. Implements shift-left security with SAST, DAST, and dependency scanning. Automates deployments while enforcing security gates and compliance checks at every stage.

Business Value

- **Faster Time to Market:** Automated pipelines reduce deployment time from weeks to hours
- **Improved Security Posture:** Catches vulnerabilities before production deployment
- **Reduced Manual Effort:** Eliminates 80%+ of manual deployment tasks
- **Audit Trail:** Complete deployment history and security scan results for compliance

Risk Mitigation

Prevents deployment of vulnerable code, hardcoded secrets, misconfigured infrastructure, and non-compliant resources to production environments.

GRC Mapping

Compliance Frameworks

- **NIST CSF:** PR.IP-1 (Baseline configuration), PR.DS-6 (Integrity checking), DE.CM-4 (Code analysis)
- **ISO 27001:** A.12.1 (Operational procedures), A.14.2 (Security in development), A.12.4 (Logging)
- **CIS Controls:** Control 16 (Application Software Security), Control 11 (Secure Configuration)
- **OWASP DevSecOps:** Security testing integration, Secret management, Dependency checking

Security Controls Implemented

- Static Application Security Testing (SAST)
- Dynamic Application Security Testing (DAST)
- Software Composition Analysis (SCA)
- Secrets scanning and prevention

- Infrastructure as Code security validation

Audit Evidence

- Pipeline execution logs with security scan results
- Code commit and approval records
- Security test reports (SAST/DAST/SCA)
- Deployment approval and rollback records

Regulatory Alignment

- **SOX:** Section 404 (Change management controls)
- **PCI DSS:** Requirement 6.3 (Secure development), Requirement 6.5 (Common vulnerabilities)
- **GDPR:** Article 25 (Data protection by design), Article 32 (Security of processing)
- **SOC 2:** CC8.1 (Change management), CC7.2 (System monitoring)