**🔬 Lab Guide 11: Run a Chaos Experiment to Kill Pods in Kubernetes**

**🎯 Learning Objectives**

By the end of this lab, participants will:

* Understand the basics of **chaos engineering**.
* Run an experiment to **kill Kubernetes pods** in a deployment.
* Observe how Kubernetes self-heals by restarting pods.
* Collect logs and document system behavior.

**📘 Background for Trainees**

Modern distributed systems must tolerate unexpected failures.

* **Chaos Engineering** = practice of deliberately injecting failures to verify resilience.
* Kubernetes is designed with **self-healing**:
  + If a pod dies, the ReplicaSet/Deployment controller will automatically start a replacement.
* By killing pods on purpose, we validate that our system continues to meet **availability SLOs**.

**🛠️ Prerequisites**

* Kubernetes cluster (Minikube, Kind, or EKS/GKE/AKS).
* kubectl configured to talk to the cluster.
* A sample app deployed (we’ll use a simple Nginx/Flask app).

**🛠️ Step-by-Step Instructions**

**Step 1 – Deploy a Sample Application**

Deploy an Nginx-based web service with 3 replicas:

# app-deployment.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: chaos-demo

spec:

replicas: 3

selector:

matchLabels:

app: chaos-demo

template:

metadata:

labels:

app: chaos-demo

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

name: chaos-demo-svc

spec:

type: NodePort

selector:

app: chaos-demo

ports:

- port: 80

targetPort: 80

nodePort: 30080

Apply it:

kubectl apply -f app-deployment.yaml

Check status:

kubectl get pods -l app=chaos-demo

👉 You should see **3 running pods**.

Open in browser:

http://PubIP:8080

👉 Confirms the app is working.

**Step 3 – Run a Chaos Experiment (Kill Pods)**

Let’s delete pods randomly and see what happens:

# Kill one pod

kubectl delete pod -l app=chaos-demo --grace-period=0 --force

OR kill multiple pods:

kubectl delete pod chaos-demo-xxxxx

kubectl delete pod chaos-demo-yyyyy

Observe:

kubectl get pods -l app=chaos-demo -w

👉 You’ll see **new pods being created automatically** to maintain 3 replicas.

**Step 4 – Automate Chaos (Optional: Chaos Monkey Style)**

Run a loop that kills pods every 10s:

while true; do

POD=$(kubectl get pod -l app=chaos-demo -o jsonpath='{.items[0].metadata.name}')

echo "Killing pod: $POD"

kubectl delete pod $POD --grace-period=0 --force

sleep 10

done

Observe in another terminal:

kubectl get pods -l app=chaos-demo -w

👉 Kubernetes continuously spins up replacements.

**Step 5 – Observe System Behavior**

Check logs during experiment:

kubectl describe deployment chaos-demo

kubectl get rs

kubectl get events --sort-by=.metadata.creationTimestamp

Questions to ask:

* How quickly did Kubernetes replace killed pods?
* Was there any downtime (check by refreshing http://pubIP:8080)?
* How did service availability look from a user perspective?

**Step 6 – Document Findings**

Prepare a short **chaos experiment log**:

| **Event** | **Time Observed** | **System Behavior** | **Notes** |
| --- | --- | --- | --- |
| Deleted pod chaos-demo-xyz | 10:15 | New pod scheduled within 2s | No downtime noticed |
| Deleted 2 pods | 10:17 | ReplicaSet spun up 2 replacements | Service stable |
| Continuous chaos loop | 10:20 | Pods recycled continuously | System stayed healthy |

**📊 Analysis (SLIs & SLOs)**

* **SLI:** % of successful HTTP responses during chaos.
* **SLO:** 99.9% availability per month.
* **Observation:** Even under pod failures, the service kept responding → system resilient.

**Lab Guide 12 :Distributed Tracing with OpenTelemetry + Jaeger**

**🎯 Scenario**

We have an **Online Bookstore** with 3 services:

* **Frontend Service** → handles user requests (UI).
* **Catalog Service** → fetches book info from DB.
* **Payment Service** → charges customer.

We want to see what happens in tracing when a user clicks **“Buy Book ID=42.”**

**🔎 Step-by-Step Detailed Flow**

**Step 1. The User Clicks “Buy Book”**

* Action: User sends HTTP request to **Frontend**.
* OpenTelemetry library inside Frontend does:
  1. Creates a **new trace** (like a journey ticket).
  2. Assigns a trace\_id = abc123.
  3. Starts a **root span** = "Buy Book Request".

👉 This span measures how long the entire request lasts inside Frontend.

**Step 2. Frontend Calls Catalog Service**

* Frontend needs book info.
* Before making the HTTP call:
  + OpenTelemetry **injects context** into request headers.
  + Header contains trace\_id=abc123 and span\_id for the parent span.

Example HTTP header:

traceparent: 00-abc123-frontendspan-01

👉 Think of this as writing the **tracking number** on the parcel before sending it to Catalog.

**Step 3. Catalog Service Receives the Request**

* Catalog gets the HTTP request.
* OpenTelemetry **extracts the context** from header.
* It sees: “Ah, this belongs to trace abc123.”
* Catalog does **not** start a new trace — it joins the existing one.
* It creates its own span: "Fetch Book Details".

This span records:

* Time to query the database.
* Attributes like db.query = SELECT \* FROM books WHERE id=42.

👉 Now the trace has 2 spans:

* "Buy Book Request" (Frontend)
* "Fetch Book Details" (Catalog)

**Step 4. Catalog Calls Database**

* Catalog queries DB.
* This DB call can also be a span if instrumented.
* Example span: "SELECT book by ID".

👉 If DB is slow, this span highlights it.

**Step 5. Frontend Calls Payment Service**

* After getting book details, Frontend now calls Payment Service.
* Again, OTel injects the trace\_id=abc123 into the HTTP header.
* Payment extracts it → creates its own span "Charge Credit Card".

This span records:

* API call to payment gateway.
* Response code (success/failure).
* Latency of the transaction.

**Step 6. All Spans Exported**

* Each service (Frontend, Catalog, Payment) sends spans to:
  + Directly → Jaeger
  + Or → OpenTelemetry Collector → Jaeger

Data sent includes:

* trace\_id (abc123)
* span\_id (unique per service step)
* Parent-child relationships
* Timing + metadata

👉 This is like each courier stop updating a central tracking system.

**Step 7. Jaeger Builds the Full Picture**

In Jaeger’s UI, you now see **one trace (abc123)** stitched together:

Trace: abc123

├─ Frontend: Buy Book Request [0–200ms]

│ ├─ Catalog: Fetch Book Details [20–70ms]

│ │ └─ Database: SELECT book [25–60ms]

│ └─ Payment: Charge Credit Card [100–180ms]

👉 You immediately see:

* The Catalog step took 50ms.
* DB took 35ms inside Catalog.
* Payment step took 80ms.
* Total user request: 200ms.

If Payment was slow, the waterfall clearly shows it.

**🎓 Key Takeaways**

1. **Trace = Full Request Journey**  
   One trace ID links all steps together.
2. **Span = One Operation**  
   Each service creates spans, nested like a family tree.
3. **Context Propagation = Passing the Ticket**  
   The trace\_id is passed in headers from service to service.
4. **OpenTelemetry = Instrumentation**  
   OTel libraries record spans, propagate IDs, and export data.
5. **Jaeger = Visualization**  
   Jaeger shows the journey as a timeline (waterfall view).

**🍕 Analogy (Pizza Order)**

* Customer places order at counter (Frontend span).
* Kitchen prepares pizza (Catalog span → DB span).
* Cashier takes payment (Payment span).
* The order slip (trace\_id) is carried across all steps.
* Jaeger shows you where the pizza spent time (waiting, cooking, billing).

**📊 Why This Matters**

* You see latency hotspots.
* You see root cause of failures (e.g., slow DB query).
* You get an **end-to-end X-ray** of your system.

**🔬 Lab Guide 13: Enable Auto Scaling with AWS Auto Scaling Groups (ASG)**

**🎯 Learning Objectives**

By the end of this lab, participants will:

* Launch an **Auto Scaling Group (ASG)** to manage EC2 instances automatically.
* Attach the ASG to an existing **Application Load Balancer (ALB)**.
* Configure scaling policies to automatically add/remove capacity.
* Simulate traffic spikes to observe **scale-out** and **scale-in** in real time.

**📘 Background**

In **Lab 10**, you deployed three static EC2 instances behind an ALB.  
That gave you **load balancing** and **high availability**, but the number of instances was fixed.

**Problem:**

* Traffic spikes → existing servers overload.
* Idle periods → you pay for unused servers.

**Solution:**  
An **Auto Scaling Group (ASG)** dynamically adjusts capacity:

* Ensures at least *min instances* are always running.
* Launches new instances when demand rises.
* Terminates extra instances when demand falls.
* Integrates with ALBs to keep traffic routed only to healthy instances.

This enables **elastic scaling** while still maintaining SLOs on availability and latency.

**🛠️ Step-by-Step Instructions**

**Step 1 – Create a Launch Template**

A Launch Template defines *how new EC2 instances should look* (AMI, size, bootstrap script, etc.).

1. Go to **EC2 → Launch Templates → Create launch template**.
2. Name: asg-web-template
3. AMI: Amazon Linux 2 (latest)
4. Instance type: t2.micro
5. Key pair: select your existing one
6. Security group: allow HTTP (80) + SSH (22)
7. **User Data** (bootstrap Apache web server):

#!/bin/bash

yum update -y

yum install -y httpd

systemctl start httpd

systemctl enable httpd

echo "<h1>Hello from ASG instance $(hostname)</h1>" > /var/www/html/index.html

1. Save template.

👉 Every new instance from this template will serve a unique “Hello” page with its hostname.

**Step 2 – Create an Auto Scaling Group**

1. Go to **EC2 → Auto Scaling Groups → Create ASG**.
2. Name: web-asg
3. Choose **Launch Template**: asg-web-template
4. Select your VPC + **at least 2 subnets** across different AZs (for resilience).
5. Load Balancing:
   * Attach to **existing Target Group** you created in Lab 10.
   * Health Check Type: EC2 + ELB (ensures ASG uses ALB health checks).
6. Group Size:
   * Desired capacity: 2
   * Minimum capacity: 1
   * Maximum capacity: 4
7. Scaling Policies:
   * Choose **Target Tracking Scaling Policy**
   * Metric: **Average CPU Utilization**
   * Target value: 50%
   * Cooldown: 120 seconds
8. Review & Create.

👉 Your ASG is now linked with the ALB and will keep between 1 and 4 instances alive, scaling as needed.

**Step 3 – Verify Initial Setup**

* Go to **EC2 → Instances** → you should see 2 running instances created by the ASG.
* Open the ALB DNS name in your browser (from Lab 10).
* Refresh several times → responses should alternate:
* Hello from ASG instance i-0abcd...
* Hello from ASG instance i-123efg...

**Step 4 – Simulate Traffic Spike**

1. Launch a temporary load generator instance (EC2, same VPC).  
   SSH into it and install Apache Benchmark:

sudo yum install -y httpd-tools

1. Run a load test against your ALB:

ab -n 200000 -c 200 http://<ALB-DNS>/

1. Monitor scaling:

watch -n 30 "aws autoscaling describe-auto-scaling-groups --auto-scaling-group-names web-asg --query 'AutoScalingGroups[0].Instances[\*].LifecycleState'"

Or in the AWS Console → you should see new instances being launched.

👉 After a few minutes, ASG will scale out (2 → 3 → 4 instances).

**Step 5 – Observe Scale-In**

* Stop the load generator.
* Wait a few minutes.
* ASG should gradually terminate extra instances and scale back toward the desired count (2, or min=1 if idle).

**📊 Analysis (SLIs/SLOs)**

* **Availability SLI:** % of successful responses via ALB.
* **Latency SLI:** Request completion time from ALB.
* **SLO Example:** 99.9% availability, 95% requests <300 ms.
* **Error Budget Impact:** If an instance fails, ASG replaces it automatically, avoiding downtime.

👉 With this lab, you’ve gone from **fixed HA with ALB** to **elastic autoscaling with ASG**, which is one of the most powerful patterns in AWS for reliability + cost efficiency.