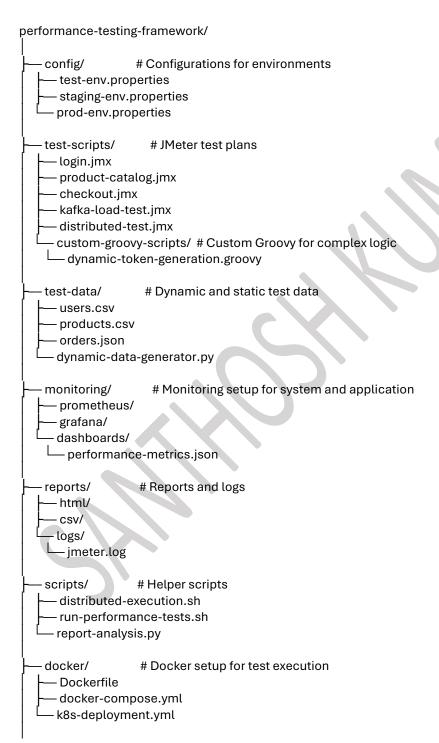
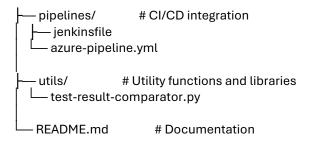
Sample Performance Testing Framework

1. Framework Architecture





2. Key Enhancements

2.1 Advanced Features

1. Distributed Load Testing:

- o Use Kubernetes to scale load generators.
- Dynamically allocate test resources using Kubernetes HPA (Horizontal Pod Autoscaler).

2. Dynamic Data Handling:

 Generate real-time test data for APIs, Kafka topics, and databases using Python scripts or Groovy.

3. Integration with Monitoring Tools:

 Integrate JMeter with Prometheus and Grafana for real-time visualization of metrics like TPS, response times, and error rates.

4. Custom Logic in Groovy:

 Use Groovy scripts for complex scenarios like token handling, conditional flows, and dynamic payload generation.

3. Setup Guide

3.1 Configuration Files

Example: test-env.properties

baseUrl=https://test-api.example.com threads=50 rampUpTime=30 duration=300 kafkaBroker=test.kafka.example.com:9092 grafanaUrl=http://grafana.example.com

3.2 Test Data

Dynamic Data Generator Script (dynamic-data-generator.py)

```
import csv
import random

def generate_users(output_file, count):
    with open(output_file, 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(['username', 'password'])
    for i in range(count):
        username = f'user{i}@example.com'
        password = f'password{i}'
        writer.writerow([username, password])

generate_users('test-data/users.csv', 100)
```

3.3 JMeter Test Scripts

1. **Custom Login Flow** (with Groovy Token Generation) **Groovy Script:** dynamic-token-generation.groovy

```
import groovy.json.JsonSlurper

def loginResponse = prev.getResponseDataAsString()
def jsonSlurper = new JsonSlurper()
def response = jsonSlurper.parseText(loginResponse)
vars.put("authToken", response.token)
```

2. Kafka Load Test Use JMeter's Kafka sampler to publish and consume messages:

```
<KafkaProducerSampler>
  <bootstrapServers>${kafkaBroker}</bootstrapServers>
  <topic>test-topic</topic>
  <key>${randomUUID()}</key>
  <message>{"orderId": ${orderId}, "status": "processing"}</message>
</KafkaProducerSampler>
```

3. Distributed Testing Setup

- Master and slave setup using Kubernetes.
- Configurable using distributed-test.jmx.

3.4 Dockerized Load Testing

Dockerfile

FROM alpine:latest RUN apk add --no-cache openidk11 curl bash

```
tar -xzf /opt/apache-jmeter.tgz -C /opt && \
rm /opt/apache-jmeter.tgz
ENV JMETER_HOME /opt/apache-jmeter-5.5
ENV PATH $JMETER_HOME/bin:$PATH
```

docker-compose.yml

```
version: '3.8'
services:
jmeter-master:
build: .
command: jmeter -n -t /scripts/distributed-test.jmx -R jmeter-slave-1,jmeter-slave-2
jmeter-slave:
image: jmeter:latest
```

4. CI/CD Integration

4.1 Jenkins Pipeline

```
pipeline {
 agent any
 environment {
   CONFIG_FILE = 'config/test-env.properties'
 stages {
   stage('Prepare Environment') {
     steps {
       script {
         sh 'python3 scripts/dynamic-data-generator.py'
   stage('Execute Tests') {
     steps {
       script {
         sh 'bash scripts/run-performance-tests.sh test'
   stage('Analyze Results') {
     steps {
       script {
         sh 'python3 scripts/report-analysis.py'
```

5. Monitoring and Reporting

5.1 Prometheus and Grafana

- Export JMeter metrics to Prometheus using jmeter-prometheus-plugin.
- Create Grafana dashboards with metrics like:
 - Average Response Time
 - Error Rate
 - 95th Percentile Latency
 - System Metrics (CPU, Memory, Disk)

Example Dashboard JSON (performance-metrics.json):

```
"title": "Performance Testing Metrics",

"panels": [
{
    "title": "Average Response Time",
    "type": "graph",
    "targets": [
        { "expr": "jmeter_response_time_avg", "legendFormat": "{{test_name}}" }
    ]
}
```

6. Execution Steps

1. Prepare Test Data:

python3 test-data/dynamic-data-generator.py

2. Run Tests:

bash scripts/run-performance-tests.sh test

- 3. Monitor Metrics: Access Grafana at http://grafana.example.com.
- 4. View Reports: Open reports/html/index.html.

7. Advanced Enhancements

1. Auto-Scaling for Distributed Load:

Use Kubernetes HPA to scale JMeter slaves based on CPU usage.

apiVersion: autoscaling/v2beta2 kind: HorizontalPodAutoscaler

spec:

scaleTargetRef: kind: Deployment name: jmeter-slave minReplicas: 2 maxReplicas: 10

metrics:

- type: Resource resource: name: cpu target:

type: Utilization averageUtilization: 80

2. Chaos Engineering Integration:

Use Gremlin to inject failures during load testing.

3. Custom Result Analysis:

Use scripts/test-result-comparator.py to compare new test results with baselines.

This sample framework provides a robust and scalable solution for advanced performance testing.