Practical work 6 - Automation in SRE: Capacity Planning and Load Testing

Objective: Understand the significance of automation in Site Reliability Engineering (SRE) with a focus on capacity planning and load testing. By the end of this practical work, students should be able to implement automated solutions for scaling infrastructure and evaluating its performance under different loads.

Task 1. Enumerate three major incidents in the tech industry that could have been prevented with better automation. For each incident, explain how automation could have helped.

Task 2. Discuss the ROI (Return on Investment) of automation in SRE. Highlight the long-term benefits versus the initial investment in setting up automation.

For Tasks 3 and 4, we can use a web application or an API service as the project for analysis.
For both tasks, it's important to choose a project that has real or simulated logs for analysis. If you don't have a real project at hand, you can use traffic and log-generating tools to simulate a real-world environment.

Task 3: Predictive Analysis Using Automation

Objective: Automate the process of analyzing server logs to predict future traffic growth.
Project: A simple web application such as a blog or an e-commerce store.

Logs: A web server, like Apache or Nginx, will generate logs each time users visit the site or perform actions on the site. These logs will contain details about the access time, user IP address, requested page, and other details.
Analysis: By analyzing these logs, you can discern traffic patterns like peak activity hours or popular pages. This information can be used to predict future traffic growth.

Steps:

1. Configuring the server for regular log extraction
- Determine where your server logs are stored. The usual location is /var/logs/.
- Use tools such as cron on Linux to automatically extract and save logs to your chosen location.
2. Using ELK stack for log analysis
- Install and configure the ELK stack (Elasticsearch, Logstash, Kibana).
- Use Logstash to import your logs into Elasticsearch.
- With Kibana, create dashboards to visualize and analyze your logs.
3. Calculation of the average monthly traffic growth rate
- Using data from Elasticsearch, calculate the monthly traffic growth by comparing the current month with the previous one.
- This can be done using a Python script or Kibana tools.
4. Traffic forecasting
- Using the calculated growth rate, forecast traffic for the next 6 months.
- Consider possible anomalies or events that may affect traffic.
5. Documentation

- Write down all your steps, tools used, and codes.
- Save the charts and dashboards from Kibana that demonstrate your analysis.

Let's look at the examples of scripts for each of the stages:

1) **Importing the Required Library:**

```
from elasticsearch import Elasticsearch
```

The script starts by importing the Elasticsearch class from the elasticsearch library. This class allows us to connect to an Elasticsearch instance and perform various operations on it.

2) **Configuring the server for regular log extraction:**

```
# Script for regular copying of logs from the server to the local directory
# log_copy.sh
#!/bin/bash
cp /var/logs/server.log /path/to/local/directory/
```

To run this script automatically, use cron:

```
crontab -e
# Add the following line to run the script every day at midnight0 0 * * * /path/to/log_copy.sh
```

3) **Using ELK stack for log analysis**

Logstash configuration (logstash.conf):

```
input {
  file {
    path => "/path/to/local/directory/server.log"
    start_position => "beginning"
  }
}

output {
  elasticsearch {
    hosts => ["localhost:9200"]
  }
}
```

Run Logstash with this configuration:

```
logstash -f /path/to/logstash.conf
```

Calculation of the average monthly traffic growth rate

Python script:

```
from elasticsearch import Elasticsearch

def get_monthly_growth():
    es = Elasticsearch([{'host': 'localhost', 'port': 9200}])
```

```
    last_month_logs = es.count(index="server_logs", body={
        "query": {
            "range": {
                "timestamp": {
                    "gte": "now-1M/M",
                    "lt": "now/M"
                }
            }
        }
    })['count']

    current_month_logs = es.count(index="server_logs", body={
        "query": {
            "range": {
                "timestamp": {
                    "gte": "now/M",
                    "lt": "now+1M/M"
                }
            }
        }
    })['count']

    growth_rate = ((current_month_logs - last_month_logs) / last_month_logs) * 100
    return growth_rate

print(f"Monthly traffic growth rate: {get_monthly_growth()}%")
```

Traffic forecasting
Continuation of the previous Python script:

```
def predict_traffic_for_next_six_months(current_month_logs, growth_rate):
    predictions = []
    for _ in range(6):
        current_month_logs += current_month_logs * (growth_rate / 100)
        predictions.append(current_month_logs)
    return predictions

growth_rate = get_monthly_growth()
current_logs = es.count(index="server_logs", body={
    "query": {
        "range": {
            "timestamp": {
                "gte": "now/M",
                "lt": "now+1M/M"
            }
        }
    }
})['count']
```

```
predictions = predict_traffic_for_next_six_months(current_logs, growth_rate)
print("Forecasts for the next 6 months:", predictions)
```

Final script:
log_analysis.py

```
import os
import datetime
from elasticsearch import Elasticsearch

# 1. Automated script for regular extraction of server logs
def get_logs_from_server():
    # Assuming the server logs are stored in the file /var/logs/server.log
    with open('/var/logs/server.log', 'r') as file:
        logs = file.readlines()
    return logs

# 2. Use tools like ELK stack for automatic log analysis
def ingest_logs_to_elk(logs):
    es = Elasticsearch([{'host': 'localhost', 'port': 9200}])
    for log in logs:
        es.index(index="server_logs", doc_type="log", body={"message": log, "timestamp":
datetime.datetime.now()})

# 3. Implement an algorithm to calculate the average monthly traffic growth rate
def calculate_growth_rate():
    es = Elasticsearch([{'host': 'localhost', 'port': 9200}])
    last_month_logs = es.search(index="server_logs", body={
        "query": {
            "range": {
                "timestamp": {
                    "gte": "now-1M/M",
                    "lt": "now/M"
                }
            }
        }
    })
    current_month_logs = es.search(index="server_logs", body={
        "query": {
            "range": {
                "timestamp": {
                    "gte": "now/M",
                    "lt": "now+1M/M"
                }
            }
        }
    })
```

```
    last_month_count = last_month_logs['hits']['total']['value']
    current_month_count = current_month_logs['hits']['total']['value']

    growth_rate = ((current_month_count - last_month_count) / last_month_count) * 100
    return growth_rate

# 4. Automate the prediction of expected traffic for the next six months
def predict_traffic(growth_rate, current_traffic):
    predictions = {}
    for i in range(1, 7):
        current_traffic += current_traffic * (growth_rate / 100)
        predictions[f"Month {i}"] = current_traffic
    return predictions

if __name__ == "__main__":
    logs = get_logs_from_server()
    ingest_logs_to_elk(logs)
    growth_rate = calculate_growth_rate()
    current_traffic = len(logs)
    predictions = predict_traffic(growth_rate, current_traffic)
    print(predictions)
```

Make sure that you have Elasticsearch running on the local machine on port 9200.
Run the script:

```
python log_analysis.py
```

The script will automatically extract logs, index them in Elasticsearch, calculate the average monthly traffic growth rate and provide a forecast for the next 6 months.

Task 4 - Infrastructure Scaling Automation

Project: An API service that provides information about products in an e-commerce store.

Scaling: If there's an anticipated spike in traffic, like during a sale or holiday season, you'd need to scale the infrastructure to handle the increased load.
Automation: Using tools like Kubernetes, you can auto-scale the number of servers based on the load. For example, if the CPU load exceeds 80%, Kubernetes can automatically add another server to handle the requests.

1. Setting Thresholds for Scaling:
Based on the traffic predictions from Task 3, you can set thresholds for when to scale up or down. For instance, if your server can handle 10,000 requests per minute (RPM) and the predicted traffic for a month is 15,000 RPM, you might want to scale up.

Python:

```
def set_scaling_thresholds(predicted_traffic):
    BASE_RPM = 10000  # The number of requests per minute your server can handle without scaling.
    SCALE_UP_THRESHOLD = 0.8  # Scale up when traffic is at 80% capacity.
    SCALE_DOWN_THRESHOLD = 0.5  # Scale down when traffic drops to 50% capacity.

    scale_up = BASE_RPM * SCALE_UP_THRESHOLD
    scale_down = BASE_RPM * SCALE_DOWN_THRESHOLD

    if predicted_traffic > scale_up:
        return "Scale Up"
    elif predicted_traffic < scale_down:
        return "Scale Down"
    else:
        return "Maintain Current Infrastructure"
```

2. Using Kubernetes for Deployment and Scaling:

Kubernetes provides a robust solution for deploying and scaling applications. To scale based on traffic, you'll primarily interact with Deployments and Horizontal Pod Autoscalers (HPA).

Here's a brief outline:

Deploy your web application on Kubernetes: This involves creating a Docker image of your app, pushing it to a container registry, and then deploying it to Kubernetes using a Deployment.
Use HPA for auto-scaling: Horizontal Pod Autoscaler automatically scales the number of pods in a deployment or replica set based on observed CPU utilization or other select metrics.
Here's a sample configuration for HPA:

Yaml:

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: webapp-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: webapp-deployment
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 80
```

In this configuration, the HPA will increase the replicas of the webapp-deployment when the CPU utilization exceeds 80%.

3. Automation Scripts for Scaling:
While HPA handles auto-scaling, you may want custom scripts for notifications or logging scaling events. This could be achieved using Kubernetes' API and Python clients.

4. Testing the Scaling Solution:
For testing, tools like locust or Apache Benchmark (ab) can be used to simulate traffic spikes. Monitor the number of pods during the test to see if the scaling occurs as expected.

5. Documentation:
Document all steps meticulously:

- Scaling Thresholds: Clearly mention the thresholds for scaling up and down and the rationale behind choosing them.
- Kubernetes Configuration: Document the configurations used, including deployment configurations and HPA settings.
- Automation Scripts: If any custom scripts are used, provide their details, code, and usage instructions.
- Testing Methodology: Describe the tools and methods used for testing, the scenarios tested, and the results observed.
- Observations and Conclusions: Note down how the system behaved during scaling tests, any bottlenecks identified, and potential improvements.