Devops Shack

DevOps Guide for Python

By DevOps Shack

[Click here for DevSecOps & Cloud DevOps Course]

DevOps Shack

DevOps Guide for Python

1. Introduction

Python has established itself as one of the most widely used languages in **DevOps** due to its powerful scripting capabilities and ease of automation. It plays a crucial role in **infrastructure automation, CI/CD pipelines, cloud management, and system administration**.

Why Python is Popular in DevOps?

1. **Versatility & Simplicity:** Python's easy-to-read syntax allows DevOps engineers to write efficient automation scripts without a steep learning curve.
2. **Cross-Platform Support:** Python runs seamlessly on Linux, Windows, and macOS, making it an ideal choice for managing diverse IT environments.
3. **Rich Ecosystem of Libraries:** Python offers specialized libraries like boto3 for AWS automation, paramiko for SSH-based tasks, and subprocess for shell scripting.
4. **Integration with DevOps Tools:** Many DevOps tools, such as Ansible, Kubernetes, and Jenkins, support Python for scripting and plugin development.

Python's Impact on DevOps Adoption

A **JetBrains survey** reports that **38% of Python usage** is in **DevOps, automation, and system administration**, highlighting its significance in managing modern infrastructure and workflows.

Python helps DevOps teams with:

✅ **Scripting:** Automating repetitive tasks like log management, server provisioning, and software deployment.

✅ **Automation:** Streamlining infrastructure provisioning, configuration management, and CI/CD workflows.

✅ **Platform Engineering:** Developing custom tools and internal platforms for DevOps teams to enhance efficiency.

With its adaptability and strong community support, Python remains an **essential language** for DevOps professionals aiming to improve automation and operational efficiency.

2. CI/CD, Infrastructure Provisioning & Configuration Management

Python plays a vital role in **Continuous Integration (CI), Continuous Deployment (CD), infrastructure provisioning, and configuration management**. While many open-source tools like **Terraform, Ansible, Jenkins, and Kubernetes** are available, Python extends their capabilities by enabling custom automation where native functionality is insufficient.

Why Python for CI/CD & Infrastructure Automation?

- **Bridges Gaps in Native DevOps Tools** – Python can enhance existing DevOps tools by adding custom scripts for deployment automation.
- **API-Driven Automation** – Python scripts can fetch secrets, interact with cloud APIs, and manage deployments dynamically.
- **Configuration Management** – Custom Python modules can extend tools like **Ansible** when default modules are unavailable.

Real-World Use Cases of Python in CI/CD & Infrastructure

✅ **Automating Secret Retrieval in CI/CD Pipelines:**
Before a deployment, Python can fetch **authentication tokens** from an API or a secret manager.

```
import requests


def get_auth_token():
    url = "https://api.example.com/get-token"
    response = requests.get(url)
    return response.json().get("token")
```

```
auth_token = get_auth_token()

print(f"Retrieved Token: {auth_token}")
```

This script ensures that deployments use dynamic authentication instead of hardcoded credentials.

✅ **Reading Data for Deployment Configuration:**
During application deployment, Python can read **CSV or JSON files** to retrieve necessary configurations.

```
import csv


with open('config.csv', mode='r') as file:
    reader = csv.DictReader(file)
    for row in reader:
        print(f"Deploying {row['service']} version {row['version']}")
```

This approach dynamically **fetches deployment parameters**, making infrastructure management more flexible.

✅ **Creating Custom Ansible Modules:**
When no built-in module exists for a particular task, Python allows DevOps engineers to **write custom Ansible modules** to extend automation capabilities.

Python in Configuration Management

- **Ansible Custom Modules:** Python scripts can define custom automation logic.
- **Terraform External Providers:** Python scripts can act as external data sources for Terraform.
- **Dynamic Inventory for Infrastructure Scaling:** Python scripts can **fetch live server IPs** dynamically in autoscaling groups.

Python enhances **CI/CD, infrastructure automation, and configuration management** by enabling **custom scripting and API-driven automation**, making it a powerful tool for DevOps engineers.

3. DevOps Platform Tooling

In modern DevOps practices, organizations often develop **internal DevOps platforms and tools** to streamline workflows, enforce security policies, and enhance automation. Python is widely used in **platform engineering** to build these custom tools, ensuring flexibility and scalability.

Why Python for DevOps Platform Engineering?

- **Custom Automation** – Python allows DevOps engineers to develop scripts and tools tailored to specific internal needs.
- **Seamless API Integration** – Python makes it easy to interact with APIs for managing cloud services, CI/CD pipelines, and monitoring systems.
- **Cross-Platform Compatibility** – Python-based DevOps tools work across Linux, Windows, and macOS environments.

Real-World Use Cases of Python in DevOps Tooling

✅ **Building Internal CLI Tools:**
Many organizations develop **command-line tools** to simplify DevOps workflows. Python's argparse module is useful for creating custom CLIs.

```
import argparse


parser = argparse.ArgumentParser(description="DevOps CLI Tool")

parser.add_argument("--deploy", help="Deploy application",
action="store_true")

args = parser.parse_args()


if args.deploy:

    print("Starting deployment...")
```

This script can be extended to automate deployment processes, manage configurations, or trigger CI/CD pipelines.

✅ **Developing API-Based Automation Tools:**
Python's Flask framework allows the creation of **custom API services** for DevOps automation.

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/health', methods=['GET'])
def health_check():
    return jsonify({"status": "OK"})

if __name__ == '__main__':
    app.run(port=5000)
```

This **lightweight monitoring service** can be integrated into infrastructure health checks or used as a webhook receiver for CI/CD events.

✅ **Automating Infrastructure Audits & Compliance Checks:**
Python can be used to **scan infrastructure configurations**, check for compliance violations, and generate reports.
Example: A script using boto3 to list **all unencrypted S3 buckets** in AWS.

```
import boto3

s3 = boto3.client("s3")
buckets = s3.list_buckets()

for bucket in buckets['Buckets']:
    encryption = s3.get_bucket_encryption(Bucket=bucket['Name'])
    if 'ServerSideEncryptionConfiguration' not in encryption:
        print(f"Bucket {bucket['Name']} is not encrypted!")
```

This helps **automate security audits** and enforce compliance policies.

Python's Role in DevOps Platform Engineering

- **Enhances productivity** by automating repetitive tasks.
- **Improves security** by enabling automated compliance checks.
- **Facilitates scalability** by allowing the development of **internal DevOps tools** that integrate seamlessly with existing infrastructure.

Python is a **key enabler** for **DevOps platform engineering**, making it an essential tool for organizations building internal automation solutions.

4. Cloud Automation

Cloud environments are an essential part of DevOps, and **Python is widely used for automating cloud infrastructure and services**. With cloud providers like AWS, Azure, and Google Cloud offering SDKs for Python, DevOps engineers can manage cloud resources programmatically, enabling **faster, more reliable, and scalable automation**.

Why Use Python for Cloud Automation?

- **Extensive Cloud SDK Support** – Libraries like boto3 (AWS), azure-mgmt (Azure), and google-cloud-sdk (GCP) allow seamless cloud interactions.
- **Infrastructure as Code (IaC) Automation** – Python scripts can provision and configure cloud infrastructure dynamically.
- **Serverless & Event-Driven Automation** – Python is widely used in cloud-based **Lambda functions, event triggers, and webhook-based automation**.

Real-World Use Cases of Python in Cloud Automation

✅ **Managing AWS EC2 Instances with Boto3**
Python's boto3 module allows DevOps engineers to automate AWS services like EC2, S3, and IAM.

```
import boto3


ec2 = boto3.client('ec2')
```

```
instances = ec2.describe_instances()

for reservation in instances['Reservations']:
    for instance in reservation['Instances']:
        print(f"Instance ID: {instance['InstanceId']}, State:
{instance['State']['Name']}")
```

This script fetches and displays the status of all EC2 instances, helping in **monitoring and managing cloud resources**.

✅ **Automating Storage Management (S3)**
Python can automate storage operations like creating and deleting S3 buckets.

```
s3 = boto3.client("s3")

bucket_name = "my-devops-automation-bucket"
s3.create_bucket(Bucket=bucket_name)
print(f"Bucket {bucket_name} created successfully!")
```

This is useful for **automated backups, data management, and storage provisioning**.

✅ **Deploying Infrastructure Using Python with AWS CDK**
Python can be used in **AWS Cloud Development Kit (CDK)** to define cloud infrastructure as code.

```
from aws_cdk import core, aws_s3 as s3

class MyStack(core.Stack):
    def __init__(self, scope: core.Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)
        s3.Bucket(self, "MyBucket")

app = core.App()
```

```
MyStack(app, "CloudAutomationStack")

app.synth()
```

This **IaC approach** ensures **repeatability, consistency, and version control** for cloud infrastructure.

✅ **Creating Serverless AWS Lambda Functions with Python**
Python is commonly used for **event-driven automation** via AWS Lambda functions. Example: Automatically stopping EC2 instances on weekends to save costs.

```
import boto3

import datetime


def lambda_handler(event, context):

    ec2 = boto3.client('ec2')

    if datetime.datetime.today().weekday() in [5, 6]:  # Saturday & Sunday

        instances = ec2.describe_instances()

        for reservation in instances['Reservations']:

            for instance in reservation['Instances']:

                ec2.stop_instances(InstanceIds=[instance['InstanceId']])

                print(f"Stopped Instance: {instance['InstanceId']}")


    return "Completed"
```

This script helps in **cost optimization** by shutting down instances when they are not needed.

Python's Role in Cloud Automation

- **Simplifies cloud management** through API-driven automation.
- **Enhances efficiency** by reducing manual intervention in cloud operations.

- **Improves scalability** by enabling Infrastructure as Code (IaC) deployments.

Python's integration with cloud providers makes it an **essential tool** for DevOps engineers to automate cloud infrastructure and services efficiently.

5. Monitoring & Alerting

Monitoring and alerting are **critical components** of DevOps to ensure system reliability, detect anomalies, and proactively resolve issues. Python is widely used to create **custom monitoring scripts, alerting mechanisms, and automated incident responses** in DevOps workflows.

Why Use Python for Monitoring & Alerting?

- **Custom Monitoring Solutions** – Python allows DevOps teams to build monitoring tools tailored to their infrastructure and applications.
- **API-Based Integrations** – Python can fetch logs, query metrics, and trigger alerts using APIs from tools like Prometheus, Grafana, and CloudWatch.
- **Automated Incident Response** – Python scripts can automatically **trigger alerts, scale infrastructure, or restart services** based on monitoring data.

Real-World Use Cases of Python in Monitoring & Alerting

✅ **Fetching System Metrics using psutil**
Python's psutil module provides real-time CPU, memory, and disk usage metrics.

```
import psutil


cpu_usage = psutil.cpu_percent(interval=1)
memory_usage = psutil.virtual_memory().percent
disk_usage = psutil.disk_usage('/').percent


print(f"CPU Usage: {cpu_usage}%")
```

```
print(f"Memory Usage: {memory_usage}%")
print(f"Disk Usage: {disk_usage}%")
```

This script helps monitor system health and **detect high resource utilization**.

✅ **Log Monitoring with Python**

Python can parse system logs and detect anomalies. Example: Checking for ERROR messages in application logs.

```
import re

log_file = "/var/log/app.log"

with open(log_file, "r") as f:
    logs = f.readlines()

for line in logs:
    if re.search(r"ERROR", line):
        print(f"Alert! Found an error: {line.strip()}")
```

This is useful for **log analysis, error detection, and troubleshooting**.

✅ **Custom Auto-Scaling Based on Alerts**

Python can listen to monitoring alerts and take automated actions, such as **scaling resources dynamically**.

Example: Using Flask to create a webhook that listens for alerts and scales EC2 instances.

```
from flask import Flask, request
import boto3

app = Flask(__name__)
ec2 = boto3.client('ec2')
```

```
@app.route('/scale', methods=['POST'])
def scale():
    alert_data = request.json
    if alert_data['metric'] == "high_cpu":
        ec2.start_instances(InstanceIds=['i-1234567890abcdef'])
        return "Scaling triggered!", 200
    return "No action taken.", 200

if __name__ == '__main__':
    app.run(port=5000)
```

This webhook can be **integrated with monitoring tools** like **Prometheus, CloudWatch, or Datadog** to trigger auto-scaling events.

✅ **Integrating Python with Slack for Alerts**

Python can send real-time alerts to **Slack, Teams, or email** when an issue is detected.

Example: Sending a Slack message when CPU usage is too high.

```
import requests

SLACK_WEBHOOK_URL = "https://hooks.slack.com/services/your/slack/webhook"

message = {"text": "🚨 High CPU usage detected on server! Please investigate."}

requests.post(SLACK_WEBHOOK_URL, json=message)
```

This helps **improve incident response times** by notifying the DevOps team instantly.

Python's Role in Monitoring & Alerting

- **Enhances observability** by collecting system and application metrics.

- **Improves reliability** by automating incident detection and resolution.
- **Integrates seamlessly** with monitoring tools to create **custom dashboards and alerting systems**.

Python's flexibility makes it an ideal choice for **real-time monitoring, proactive alerting, and automated incident response** in DevOps environments.

6. MLOps (Machine Learning Operations)

MLOps (Machine Learning Operations) is the practice of integrating machine learning models into the **DevOps pipeline**. As DevOps engineers work alongside data scientists and ML teams, Python plays a key role in automating machine learning workflows and ensuring that ML models are properly integrated, tested, and deployed.

Why Use Python for MLOps?

- **Widespread Adoption in Data Science** – Python is the most commonly used language for machine learning, making it an ideal choice for MLOps automation.
- **Integration with ML Frameworks** – Python integrates seamlessly with popular ML libraries like **TensorFlow, PyTorch, Scikit-Learn**, and **XGBoost**, enabling full automation of model deployment.
- **Automation of Model Training & Deployment** – Python can automate the process of training, testing, and deploying machine learning models as part of the CI/CD pipeline.

Real-World Use Cases of Python in MLOps

✅ **Automating Model Training with Python**

Python is used to automate the process of **training machine learning models** on new datasets, ensuring that models are continually updated as fresh data is available.

Example: A Python script that trains a **Scikit-learn model** and saves it to a file.

```
from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.datasets import load_iris
```

```
import joblib

# Load dataset and split
data = load_iris()
X_train, X_test, y_train, y_test = train_test_split(data.data, data.target, test_size=0.2)

# Train model
model = RandomForestClassifier()
model.fit(X_train, y_train)

# Save model
joblib.dump(model, 'iris_model.pkl')
```

This automates model training and ensures that the latest model is always available for deployment.

✅ **Automating Model Deployment using Flask & Docker**

After training, Python can automate the deployment of ML models by serving them through APIs. For example, using **Flask** to create a REST API that serves the trained model.

```
from flask import Flask, request, jsonify
import joblib

app = Flask(__name__)
model = joblib.load('iris_model.pkl')

@app.route('/predict', methods=['POST'])
def predict():
    data = request.get_json()
```

```
    prediction = model.predict([data['features']])
    return jsonify({'prediction': prediction.tolist()})

if __name__ == '__main__':
    app.run(debug=True)
```

This allows machine learning models to be **exposed as APIs**, making them easy to integrate into larger systems or applications.

✅ **Automating Model Monitoring**

Python is also used to **monitor deployed models** by tracking their performance over time. If a model's performance deteriorates, Python scripts can trigger alerts or re-train the model.

Example: A Python script that checks model accuracy and triggers an alert if accuracy falls below a threshold.

```
import joblib
import numpy as np

model = joblib.load('iris_model.pkl')
accuracy = model.score(X_test, y_test)

if accuracy < 0.85:
    print("Alert! Model accuracy below threshold!")
```

This ensures that the **ML models remain performant** and up-to-date throughout their lifecycle.

✅ **Orchestrating ML Pipelines with Apache Airflow**

Airflow is a widely used tool in MLOps for managing complex ML pipelines. Python scripts are often used to define tasks in Airflow, such as **data preprocessing, model training, and model evaluation**.

```
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
```

```
from datetime import datetime


def train_model():
    # Your model training code here
    pass


default_args = {'owner': 'airflow', 'start_date': datetime(2022, 1, 1)}


dag = DAG('ml_pipeline', default_args=default_args,
schedule_interval='@daily')


train_task = PythonOperator(task_id='train_model',
python_callable=train_model, dag=dag)
```

Airflow allows DevOps engineers to automate end-to-end machine learning workflows, making it an essential tool in **MLOps automation**.

Python's Role in MLOps

- **End-to-End Automation** – Python automates **training, testing, deployment, and monitoring** of machine learning models, integrating them into DevOps workflows.
- **Collaboration Between Teams** – Python helps DevOps engineers and data scientists collaborate on **deploying, scaling, and monitoring ML models** in production.
- **Streamlined Model Deployment** – By integrating Python with APIs, Docker, and orchestration tools like Airflow, the deployment and management of ML models are streamlined and automated.

Python is **indispensable in MLOps**, enabling **continuous integration, continuous delivery, and continuous training** of machine learning models in production environments.

7. Important Python Modules for DevOps Automation

In DevOps, Python's versatility is enhanced by a wide range of modules and libraries that simplify automation, system administration, cloud management, and integration tasks. Below is a list of essential Python modules that DevOps engineers frequently use for automation and other DevOps tasks.

1. os and sys

- **Use**: These modules are part of Python's standard library and are primarily used for interacting with the operating system.
- **Use Cases**: File system manipulation, process management, environment variable handling, and path management.
    - Example: Running shell commands, getting environment variables.

```
import os

print(os.environ['HOME'])

os.system('ls -l')
```

2. subprocess

- **Use**: To spawn new processes, connect to their input/output/error pipes, and obtain their return codes.
- **Use Cases**: Running shell commands and capturing output from command-line tools. This is especially useful for automating tasks that involve system-level commands or scripts.

```
import subprocess

result = subprocess.run(['ls', '-l'], stdout=subprocess.PIPE)

print(result.stdout.decode())
```

3. psutil

- **Use**: Provides an interface for retrieving information on running processes and system utilization (CPU, memory, disks, network, and sensors).
- **Use Cases**: Monitoring and managing system resources, creating alerts based on system health metrics.

python

CopyEdit

```
import psutil

print(f"CPU Usage: {psutil.cpu_percent()}%")

print(f"Memory Usage: {psutil.virtual_memory().percent}%")
```

4. boto3

- **Use**: The Amazon Web Services (AWS) SDK for Python, used to interact with AWS services like EC2, S3, and Lambda.
- **Use Cases**: Automating AWS infrastructure provisioning, managing resources, deploying applications, and scaling cloud infrastructure.

```
import boto3

ec2 = boto3.resource('ec2')

for instance in ec2.instances.all():

    print(instance.id, instance.state)
```

5. requests and urllib3

- **Use**: These modules are used for making HTTP requests. requests is a simpler and more popular library for making HTTP requests in Python, while urllib3 is more low-level.
- **Use Cases**: API integrations, sending data to monitoring systems, managing configurations, making HTTP requests to third-party services.

```
import requests

response = requests.get('https://api.example.com/data')

print(response.json())
```

6. paramiko

- **Use**: A module for working with SSH to execute commands on remote machines.
- **Use Cases**: Automating SSH access, running commands remotely, managing servers, and securely transferring files.

```
import paramiko

ssh = paramiko.SSHClient()

ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())

ssh.connect('example.com', username='user', password='password')

stdin, stdout, stderr = ssh.exec_command('uptime')

print(stdout.read().decode())
```

7. PyYAML

- **Use**: A module used for parsing and writing YAML files, which are commonly used for configuration management.
- **Use Cases**: Reading/writing configuration files, automating tasks related to system configuration, managing infrastructure as code (e.g., Ansible playbooks).

```
import yaml

with open('config.yaml', 'r') as file:

    config = yaml.load(file, Loader=yaml.FullLoader)

print(config)
```

8. json

- **Use**: Part of the Python standard library for parsing JSON data. JSON is widely used for configuration files and data interchange in DevOps pipelines.
- **Use Cases**: Reading and writing JSON files, integrating with APIs, managing configuration data, and passing data between services.

```
import json

data = {'name': 'DevOps', 'language': 'Python'}

json_data = json.dumps(data)

print(json_data)
```

9. logging

- **Use**: Python's built-in library for logging, which is critical for tracking the execution of scripts and diagnosing errors.
- **Use Cases**: Setting up logging in automation scripts to track execution, failures, and system events.

```
import logging

logging.basicConfig(level=logging.INFO)

logging.info('This is an info message')

logging.error('This is an error message')
```

10. re (Regular Expressions)

- **Use**: Provides regular expression matching operations.
- **Use Cases**: Parsing log files, validating data, extracting information, filtering logs, or automating tests.

```
import re

match = re.search(r"error", "Log file with an error message")

if match:

  print("Error found!")
```

11. pandas

- **Use**: A data analysis library, often used for working with structured data such as CSV or Excel files.
- **Use Cases**: Analyzing logs, processing CSV files, and working with structured data for DevOps automation (e.g., managing deployments, configuration data).

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df.head())
```

12. smtplib

- **Use**: A Python module for sending emails using the Simple Mail Transfer Protocol (SMTP).

- **Use Cases**: Sending automated alerts, notifications, or reports as part of the automation process.

```
import smtplib
from email.mime.text import MIMEText

msg = MIMEText('Your server is down!')
msg['Subject'] = 'Server Alert'
msg['From'] = 'admin@example.com'
msg['To'] = 'user@example.com'

with smtplib.SMTP('smtp.example.com') as server:
    server.login('admin', 'password')
    server.sendmail('admin@example.com', ['user@example.com'], msg.as_string())
```

13. flask

- **Use**: A web framework for building APIs and web applications.
- **Use Cases**: Creating custom web-based dashboards, integrating with tools to display monitoring data, or serving machine learning models as APIs in MLOps pipelines.

```
from flask import Flask, jsonify
app = Flask(__name__)

@app.route('/status', methods=['GET'])
def status():
    return jsonify({"status": "up and running"})

if __name__ == '__main__':
```

```
app.run(debug=True)
```

14. Kubernetes Python Client

- **Use**: A Python client for interacting with Kubernetes clusters.
- **Use Cases**: Managing Kubernetes resources, scaling deployments, monitoring pods, and integrating with CI/CD pipelines.

```
from kubernetes import client, config


config.load_kube_config()

v1 = client.CoreV1Api()

pods = v1.list_pod_for_all_namespaces(watch=False)

for pod in pods.items:

    print(f"Pod Name: {pod.metadata.name}")
```

Project: Server Health Monitoring and Alerting System

Objective:

Automate the process of monitoring system health (CPU, memory, and disk usage) and send email alerts if these exceed certain thresholds (e.g., CPU usage > 80%, Memory usage > 75%).

Prerequisites:

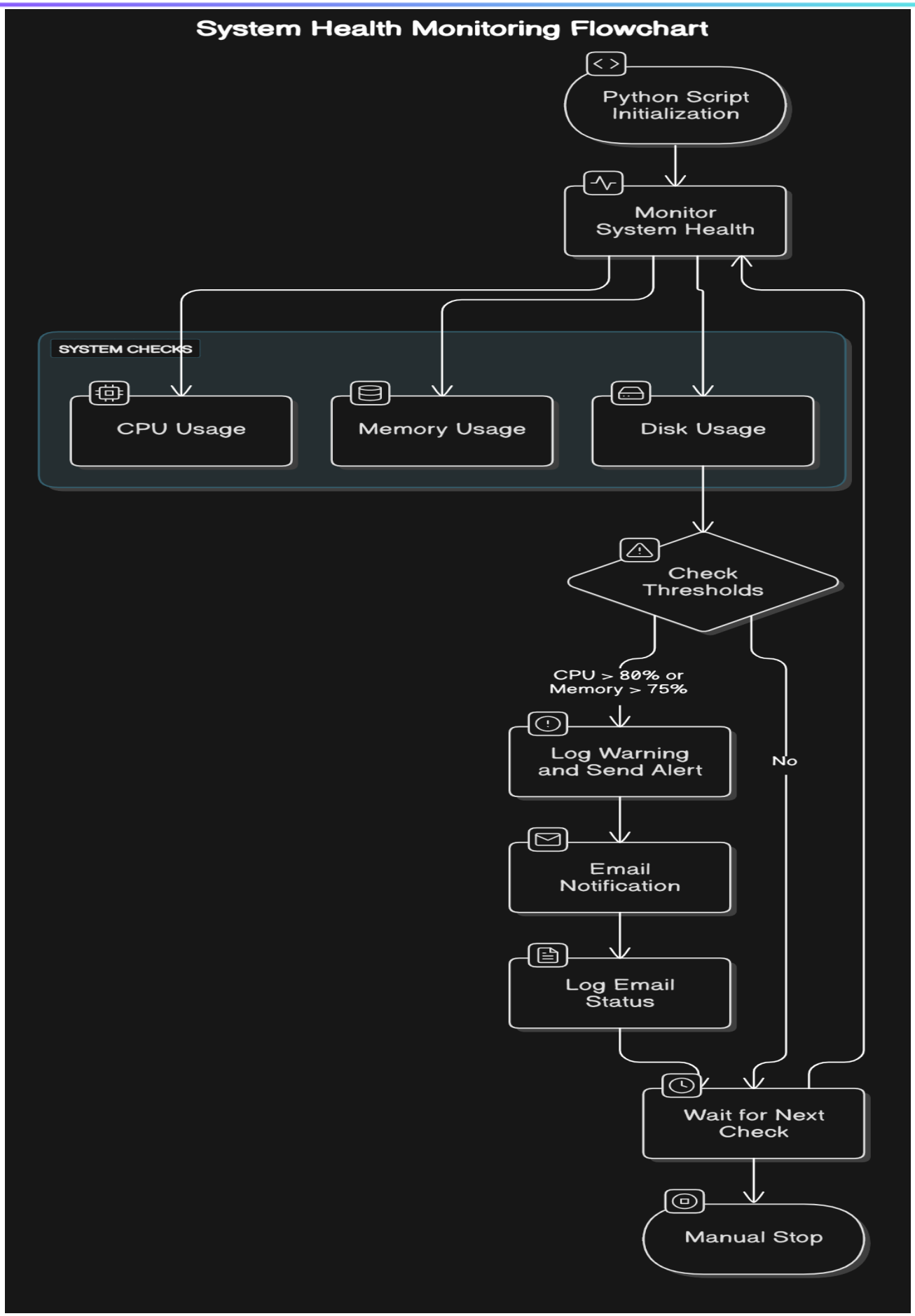
- **Python**: Ensure Python is installed on your system (Python 3.x recommended).
- **Libraries**: We'll need several Python libraries, mainly psutil for system monitoring, smtplib for sending emails, and logging for logging events.
- **Email**: A Gmail or SMTP-compatible email account to send alerts (you will need to allow access to less secure apps or use an application-specific password for Gmail).

Server Health Monitoring and Alerting System Workflow

1. **Initialization**
    - Start Python Script.
    - Set up logging for tracking events (CPU, memory usage, and alerts).
2. **System Health Check**
    - Monitor **CPU usage**, **Memory usage**, and **Disk usage** using the psutil library.
    - Collect metrics every **X seconds** (e.g., every 5 minutes).
3. **Threshold Check**
    - **CPU Usage**: Check if CPU usage exceeds **80%**.
    - **Memory Usage**: Check if memory usage exceeds **75%**.
4. **Trigger Alert**
    - If any threshold is exceeded:

- Log the event as a **warning** (e.g., high CPU or memory usage).
- Call the **Send Alert** function.

5. **Send Alert via Email**
   - Use smtplib to send an email with system health details (CPU and memory usage) to the designated recipient.
   - Log success or failure of sending the alert.
6. **Repeat Monitoring**
   - After sending an alert or completing the check, wait for the next cycle (e.g., **5 minutes**) and repeat the monitoring process.
7. **Exit/Stop**
   - The monitoring continues indefinitely until manually stopped.


System Health Monitoring Flowchart
Python Script Initialization
Monitor System Health
SYSTEM CHECKS
CPU Usage
Memory Usage
Disk Usage
Check Thresholds
CPU > 80% or Memory > 75%
No
Log Warning and Send Alert
Email Notification
Log Email Status
Wait for Next Check
Manual Stop

Step-by-Step Implementation

1. Set Up Python Environment

Install necessary Python libraries using pip:

```
pip install psutil
```

```
pip install smtplib
```

These libraries will allow you to:

- psutil: Access system resource stats (CPU, memory, disk usage).
- smtplib: Send emails via an SMTP server (like Gmail).
- logging: Log events related to system health checks.

2. Import Libraries and Define Functions

```
import psutil
```

```
import smtplib
```

```
from email.mime.text import MIMEText
```

```
import time
```

```
import logging
```

3. Configure Logging

You can use the logging module to keep track of events, such as when thresholds are exceeded or when health checks occur. Set up logging to write to a file:

```
logging.basicConfig(filename='server_health.log', level=logging.INFO,
                    format='%(asctime)s - %(levelname)s - %(message)s')
```

This log file will track all health check results and alert actions.

4. Create Functions to Monitor System Resources

We'll use psutil to monitor system resources such as CPU usage, memory usage, and disk usage.

Function to get system health stats:

```
def get_system_health():
```

```
    cpu_usage = psutil.cpu_percent(interval=1)  # Returns CPU usage percentage over 1 second

    memory = psutil.virtual_memory()  # Returns memory stats (total, available, used, percent)

    memory_usage = memory.percent  # Memory usage percentage

    return cpu_usage, memory_usage
```

This function will return the CPU and memory usage.

5. Define the Alerting Function

Using smtplib and MIMEText, we can set up email alerts. This function will send an email whenever a threshold is exceeded.

```
def send_alert(cpu_usage, memory_usage):

    # Email setup

    sender = "your_email@gmail.com"

    recipient = "recipient_email@example.com"

    subject = "Server Health Alert"

    body = f"Warning! The server has exceeded its thresholds.\n\nCPU Usage: {cpu_usage}%\nMemory Usage: {memory_usage}%"


    msg = MIMEText(body)

    msg['Subject'] = subject

    msg['From'] = sender

    msg['To'] = recipient


    try:

        with smtplib.SMTP_SSL('smtp.gmail.com', 465) as server:

            server.login(sender, 'your_password')  # Use your Gmail app password

            server.sendmail(sender, [recipient], msg.as_string())
```

```
            logging.info("Alert sent successfully!")
    except Exception as e:
        logging.error(f"Failed to send alert: {e}")
```

6. Define the Threshold Check Function

Next, we'll define the function that checks if CPU or memory usage exceeds the defined thresholds. If the usage exceeds these thresholds, we call the send_alert() function.

```
def check_thresholds():
    cpu_usage, memory_usage = get_system_health()

    # Define thresholds (e.g., CPU > 80%, Memory > 75%)
    cpu_threshold = 80
    memory_threshold = 75

    if cpu_usage > cpu_threshold:
        logging.warning(f"CPU usage is high: {cpu_usage}%")
        send_alert(cpu_usage, memory_usage)

    if memory_usage > memory_threshold:
        logging.warning(f"Memory usage is high: {memory_usage}%")
        send_alert(cpu_usage, memory_usage)
```

7. Implement a Continuous Monitoring Loop

To continuously monitor the server's health, we will use a loop with a delay (e.g., check every 5 minutes). The loop will run the health check and log the results.

```
def monitor():
    while True:
```

```
        check_thresholds()
        logging.info(f"CPU: {psutil.cpu_percent()}% | Memory: {psutil.virtual_memory().percent}%")
        time.sleep(300)  # Check every 5 minutes
```

8. Running the Monitoring System

Now, we need to execute the monitoring function in the script's main section:

```
python
CopyEdit
if __name__ == '__main__':
    monitor()
```

When this script runs, it will monitor the system's health, check the usage of resources, and send an email alert if the usage exceeds the specified thresholds. It will also log all checks and alerts.

Additional Enhancements

1. Customizable Thresholds

Instead of hardcoding thresholds, consider making them configurable via a settings file or environment variables. This will make the tool more flexible.

```
import os


cpu_threshold = int(os.getenv('CPU_THRESHOLD', 80))  # Default threshold is 80%
memory_threshold = int(os.getenv('MEMORY_THRESHOLD', 75))  # Default threshold is 75%
```

2. Monitor Additional Resources

You can extend the functionality to monitor more system resources, such as disk usage, network activity, etc. Example for disk usage:

```
def get_disk_usage():
```

```
    disk = psutil.disk_usage('/')
    return disk.percent
```

3. Integration with Other Monitoring Tools

You can integrate this script with popular monitoring tools like Prometheus or Grafana for more advanced visualizations and alerting.

4. Handling Edge Cases

Make sure your script handles any edge cases, such as when psutil cannot retrieve data or the SMTP server fails.

5. Notifications via Multiple Channels

Instead of just email, you could use APIs like Slack or Telegram to send alerts for more timely notifications.

Testing the Project

1. **Run the Script**:
   Run the script on your local machine or a test server. Ensure that it continuously checks system resources and sends email alerts when thresholds are crossed.
2. **Test Alerting**:
   Manually trigger high CPU or memory usage using tools like stress or memtester, or simulate it in a test environment to verify that the alerting system works.

Conclusion.

By implementing this Server Health Monitoring and Alerting System, you will:

- Gain experience using Python for system monitoring and automation.
- Learn how to interact with system APIs (e.g., CPU, memory) using psutil.
- Automate real-time alerts for system health, which is a key DevOps responsibility.
- Understand how to implement a simple but powerful monitoring tool for both personal and production environments.

This project is scalable, and as you gain more experience, you can extend it with more features such as integrating with cloud platforms (AWS, GCP), using Kubernetes, and handling different types of system alerts.

Conclusion

Python's power in DevOps automation is largely enhanced by these essential libraries and modules. They offer a wide variety of tools for **system administration, automation, monitoring, cloud management, and data processing**. Understanding and using these modules enables DevOps engineers to **automate repetitive tasks**, **integrate with cloud services**, **monitor system health**, and build **custom solutions** for complex DevOps workflows.

Throughout this guide, we have explored how Python plays a crucial role in DevOps automation and system administration. By delving into various use cases such as CI/CD, infrastructure provisioning, cloud automation, and monitoring, we have highlighted Python's versatility and power in automating repetitive tasks and enhancing DevOps workflows.

Python's ease of use, extensive libraries, and seamless integration with popular tools make it a valuable asset for DevOps engineers. Whether you're managing cloud environments with **Boto3**, monitoring system health, or automating deployment pipelines, Python provides the flexibility and reliability needed to improve efficiency, reduce manual intervention, and ensure robust system performance.

By learning the fundamental concepts, key Python modules, and real-world implementation strategies, you can successfully integrate Python into your DevOps processes. From basic scripting tasks to more advanced automation like cloud management, monitoring, and continuous integration, Python enables DevOps engineers to build and scale solutions efficiently.

As DevOps continues to evolve, mastering Python for automation not only enhances your capabilities but also makes you a more competitive and effective engineer in the rapidly growing DevOps field. By undertaking projects such as the **Server Health Monitoring and Alerting System**, you gain hands-on experience and deepen your understanding of practical automation in real-world scenarios.

In conclusion, mastering Python for DevOps is an essential skill for any engineer looking to automate workflows, streamline operations, and drive more efficient, scalable systems. With the tools, techniques, and best practices outlined in this guide, you’re well-equipped to embark on a rewarding journey into Python-driven DevOps automation.