Task 1: System Performance Optimization and Automation

Objective: Analyze a hypothetical IT system facing performance issues, optimize it, and automate critical processes to ensure efficiency and reliability.

Scenario:
You are responsible for a multi-tier application deployed on a combination of on-premises servers and cloud instances. The system has been experiencing performance bottlenecks, particularly during peak usage times. Your task is to identify the issues, optimize the system, and automate key processes to enhance performance and reliability.

Samadhi Perera
Samadhi0830@gmail.com

Contents

1. Analysis

In a multi-layered application, where parts like the user interface, business logic, and database work together, performance problems can come from various sources. These systems are complex, with different components depending on each other across physical servers and cloud platforms. Each part, whether handling user actions, running processes, or storing data, can affect the system’s speed and reliability.

It's important to identify and fix these issues to ensure stability, scalability, and efficiency, especially during high-demand periods. Problems in any layer can cause widespread failures or a bad user experience. Performance issues can arise from resource shortages (like CPU, memory, or disk), slow networks, or poorly designed databases. Cloud environments bring extra challenges, such as managing the right number of resources, which requires careful monitoring and scaling.

1.1 Identify Bottlenecks

1.1.1 CPU bottlenecks

When a server's CPU, which manages different parts of an application (like the front-end, middleware, or database), becomes fully used during busy periods, it slows down the system’s ability to handle requests. This results in slower response times and delays throughout the application. For example, if the middleware or database is limited by CPU capacity, the entire process slows down, impacting the front-end and user experience. Common causes include high CPU usage, long process wait times, and delayed requests. Solutions include upgrading CPUs, adding more servers to share the load, improving code, balancing workloads, and optimizing heavy tasks.

1.1.2 Memory Bottlenecks

Memory bottlenecks happen when there isn't enough RAM to manage tasks, which often affects applications handling large data and using caching for speed. When RAM is full, the system turns to the hard drive as backup, but it's slower, leading to reduced performance and, in some cases, causing apps to crash or restart. Common reasons include high memory usage, increased disk activity from swapping, and running out of memory. To fix this, you can add more RAM, repair memory leaks, optimize data structures, or improve caching. Monitoring tools can spot issues, and autoscaling adjusts resources automatically to prevent bottlenecks.

1.1.3 Network Latency

Network latency is the delay that occurs when different parts of a system communicate, like between users and an application or between servers (web, application, and databases). This is especially important when these parts are in different places or layers. When the network is slow, the application also slows down, leading to longer wait times and delayed interactions. Latency issues are more noticeable when system components are hosted in different regions.

Common causes include long distances between servers, lost data that needs to be resent, and inefficient data routes. Solutions to reduce latency involve optimizing data routes, using CDNs for faster content delivery, improving traffic management with load balancers, minimizing back-and-forth data transfers, and using more efficient data formats like JSON.

1.1.4 Database contention

When multiple parts of an application, such as services or users, access the same database at once, it can slow down performance. This issue is typical in relational databases that use locks to maintain data accuracy, leading to delays, system deadlocks, and longer wait times, especially during heavy usage. Factors like long query times, timeouts, and retries can exacerbate the problem.

To address this, can improve performance by optimizing queries with indexes, minimizing full table scans, and implementing query caching. Distributing the workload through partitioning or sharding can also help. Switching to a more scalable database, such as NoSQL or NewSQL, might enhance performance. Additionally, using caching systems like Redis or Memcached can reduce the number of direct requests to the database, and employing connection pooling can improve connection management.

1.1.5 Disk I/O Bottlenecks

Disk I/O bottlenecks occur when reading or writing data to storage is slow, which can cause problems for systems that save data frequently, like databases or logging systems, or when dealing with large files. This slow access can affect the performance of applications that depend on fast storage, leading to delays and slower processing.
Common causes include long wait times for accessing data, frequent I/O warnings, and slow file or database access. To resolve this, you can switch to faster storage like SSDs or use high-performance cloud storage (e.g., AWS EBS or Azure Premium). Other fixes include adding caching, using asynchronous processing, optimizing the file system, and reducing unnecessary writes to the disk. These solutions help improve system performance.

1.1.6 Application Layer Bottlenecks (Code Inefficiencies)

Inefficient code and poor algorithms can slow down applications and negatively impact system performance. This often occurs due to unnecessary loops, inefficient task handling, or long wait times for external resources like third-party APIs. These issues can increase CPU and memory usage, delay user requests, and cause timeouts, making the application harder to scale.

Signs of inefficient code include slow response times, high CPU usage, and alerts from performance tools. Developers can improve performance by simplifying the code, optimizing data structures, and using asynchronous processing for tasks that require waiting. Regular code reviews and performance testing can help identify issues early. Additionally, breaking the application into smaller services (microservices) can enhance efficiency and resource management, improving overall performance.

1.1.7 Session Management Issues

In multi-layered applications, managing user sessions poorly can slow down performance, especially when all session data is on one server. This overloads the server, causing delays in logins and data access, particularly during peak times. If session data is lost or not shared properly, users may face transaction failures or data loss, leading to long login times and inconsistent data across servers. Solutions like distributed session management (using sticky sessions, session replication, or Redis) or stateless methods with tokens (such as JWT) can help improve scalability. Spreading session data across multiple servers eases the load on individual servers, enhancing performance and reliability during busy periods.

1.1.8 External API or Service Dependency Issues:

Many apps depend on outside services, such as payment systems or social media connections. If these services are slow or not working, the app can lag or even crash, resulting in a frustrating experience for users. Issues like delays or timeouts with these services can harm performance. To address this, you can use techniques like circuit breakers and backup methods to manage failures, retry failed requests with increasing wait times, and store previous responses to reduce the number of requests. Additionally, keeping an eye on the performance of these services and updating contracts with service providers can help ensure they work reliably and respond quickly.

1.2 Monitoring Setup

This script monitors important performance metrics in a multi-tier application, such as CPU usage, memory, disk activity, network delays, and session management. It logs this data in JSON format and sends alerts via Slack if any metrics exceed set limits. The goal is to quickly find performance issues and send real-time notifications to avoid slowdowns or outages. It also includes custom features for simulating performance issues and tracking session counts. Future updates may include real-world latency tests, flexible thresholds, and connections to more advanced monitoring tools.

```
!pip install psutil ping3 requests
```

```
Requirement already satisfied: psutil in /usr/local/lib/python3.10/dist-packages (5.9.5)
Collecting ping3
  Downloading ping3-4.0.8-py3-none-any.whl.metadata (13 kB)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (2.32.3)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests) (2.2.3)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests) (2024.8.30)
Downloading ping3-4.0.8-py3-none-any.whl (14 kB)
Installing collected packages: ping3
Successfully installed ping3-4.0.8
```

```python
import psutil
import time
import json
import requests
from datetime import datetime

LOG_FILE = "performance_metrics.json"
THRESHOLDS = {
    "cpu": 80,      # Hardcoded CPU usage percentage
    "memory": 80,   # Hardcoded Memory usage percentage
    "disk_io": 1000,  # Hardcoded Disk I/O operations per second
    "network_latency": 100,  # Hardcoded Latency in ms
    "application_bottlenecks": 0.5,  # Hardcoded Custom application logic
    "session_management": 200,  # Hardcoded Simulated session count
}

# Slack Webhook URL
SLACK_WEBHOOK_URL = "https://hooks.slack.com/services/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"

def send_alert(metric, value):
    """Send an alert to Slack when a metric exceeds its threshold."""
    message = {
        "text": f"Alert: {metric} exceeded threshold. Current value: {value}"
    }
```

```
+ Code + Text                                                    Reconnect  ✦ Gemini
```

```python
    }

    # Post the message to the Slack webhook URL
    response = requests.post(SLACK_WEBHOOK_URL, json=message)

    if response.status_code != 200:
        print(f"Failed to send alert to Slack: {response.status_code}, {response.text}")
    else:
        print(f"Alert sent to Slack: {metric} with value {value}")

def log_metrics(metrics):
    with open(LOG_FILE, "a") as log_file:
        json.dump(metrics, log_file)
        log_file.write("\n")

def check_application_bottlenecks():
    simulated_bottleneck = 0.4
    return simulated_bottleneck

def check_network_latency():
    return 20

def check_session_management():

    simulated_session_count = 150
    return simulated_session_count
```

```python
# Monitor system metrics for a defined number of iterations
num_iterations = 10
for _ in range(num_iterations):
    # Performance metrics collection
    metrics = {
        "timestamp": datetime.now().isoformat(),
        "cpu_usage": psutil.cpu_percent(),
        "memory_usage": psutil.virtual_memory().percent,
        "disk_io": psutil.disk_io_counters().read_count + psutil.disk_io_counters().write_count,
        "network_latency": check_network_latency(),
        "application_bottlenecks": check_application_bottlenecks(),
        "session_management": check_session_management(),
    }

    # Log metrics
    log_metrics(metrics)
```

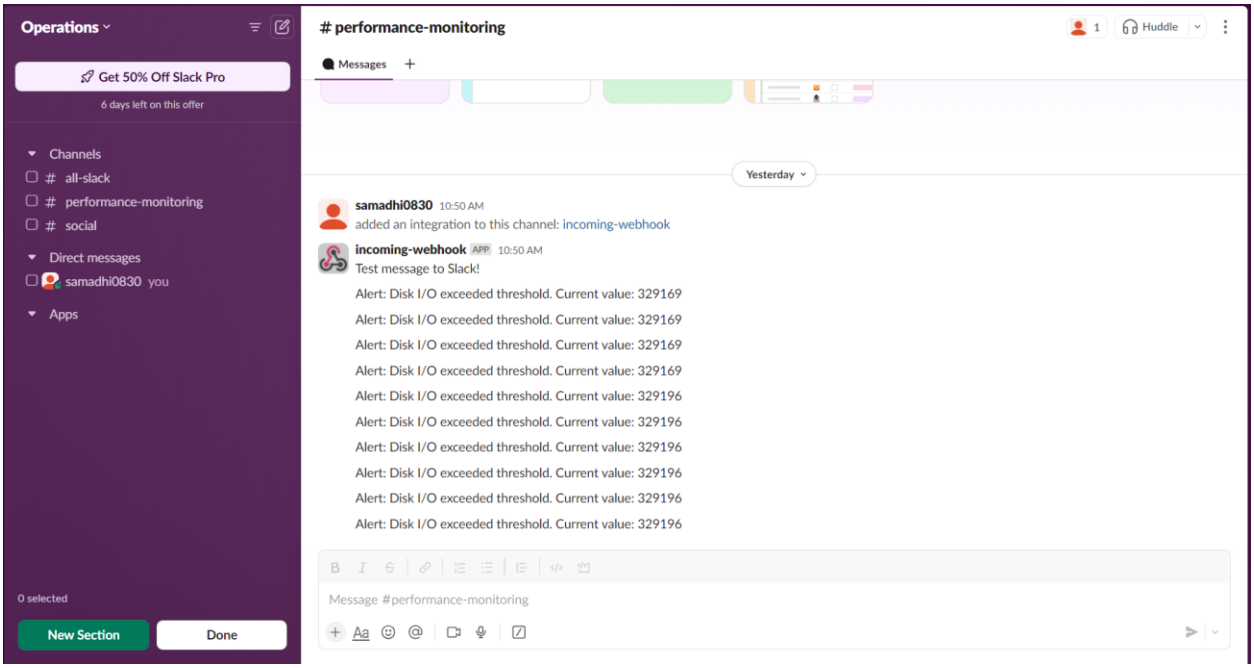
```python
    # Check against thresholds and print warnings if exceeded
    if metrics["cpu_usage"] > THRESHOLDS["cpu"]:
        print(f"Warning: CPU Usage exceeded threshold! Current value: {metrics['cpu_usage']}")
        send_alert("CPU Usage", metrics["cpu_usage"])
    if metrics["memory_usage"] > THRESHOLDS["memory"]:
        print(f"Warning: Memory Usage exceeded threshold! Current value: {metrics['memory_usage']}")
        send_alert("Memory Usage", metrics["memory_usage"])
    if metrics["disk_io"] > THRESHOLDS["disk_io"]:
        print(f"Warning: Disk I/O exceeded threshold! Current value: {metrics['disk_io']}")
        send_alert("Disk I/O", metrics["disk_io"])
    if metrics["network_latency"] > THRESHOLDS["network_latency"]:
        print(f"Warning: Network Latency exceeded threshold! Current value: {metrics['network_latency']}")
        send_alert("Network Latency", metrics["network_latency"])
    if metrics["application_bottlenecks"] > THRESHOLDS["application_bottlenecks"]:
        print(f"Warning: Application Bottlenecks exceeded threshold! Current value: {metrics['application_bottlenecks']}")
        send_alert("Application Bottlenecks", metrics["application_bottlenecks"])
    if metrics["session_management"] > THRESHOLDS["session_management"]:
        print(f"Warning: Session Management exceeded threshold! Current value: {metrics['session_management']}")
        send_alert("Session Management", metrics["session_management"])
```

Output

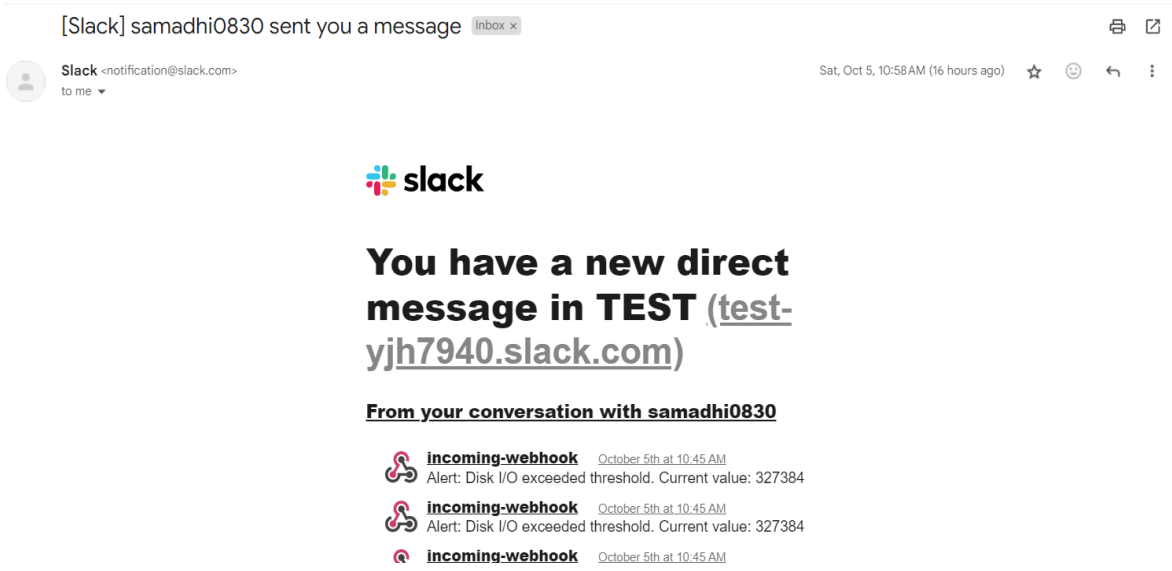
Through Script:

```
Warning: Disk I/O exceeded threshold! Current value: 329169
Alert sent to Slack: Disk I/O with value 329169
Warning: Disk I/O exceeded threshold! Current value: 329169
Alert sent to Slack: Disk I/O with value 329169
Warning: Disk I/O exceeded threshold! Current value: 329169
Alert sent to Slack: Disk I/O with value 329169
Warning: Disk I/O exceeded threshold! Current value: 329169
Alert sent to Slack: Disk I/O with value 329169
Warning: Disk I/O exceeded threshold! Current value: 329196
Alert sent to Slack: Disk I/O with value 329196
Warning: Disk I/O exceeded threshold! Current value: 329196
Alert sent to Slack: Disk I/O with value 329196
Warning: Disk I/O exceeded threshold! Current value: 329196
Alert sent to Slack: Disk I/O with value 329196
Warning: Disk I/O exceeded threshold! Current value: 329196
Alert sent to Slack: Disk I/O with value 329196
Warning: Disk I/O exceeded threshold! Current value: 329196
Alert sent to Slack: Disk I/O with value 329196
Warning: Disk I/O exceeded threshold! Current value: 329196
Alert sent to Slack: Disk I/O with value 329196
```

Slack Output



Email Notification



1.2.1 Logics Used for the implementation.

The python script uses the psutil library to gather system-level metrics like CPU usage, memory usage, and disk I/O operations. Custom functions are added for network latency, application bottlenecks, and session management.

CPU usage (psutil.cpu_percent()): Directly impacts application performance, especially for compute-heavy processes.

Memory usage (psutil.virtual_memory().percent): High memory usage can lead to paging/swapping, slowing down applications.

Disk I/O (psutil.disk_io_counters()): Disk throughput bottlenecks can delay access to essential data, especially for I/O-bound processes.

Network latency (custom function): Latency can degrade user experience and cause delays in service response times, especially in distributed architectures.

Application bottlenecks (custom simulation): This function simulates performance bottlenecks within the application layer, like database locks or inefficient code execution.

Session management (custom simulation): Simulates session handling, relevant for multi-user systems like web servers, where higher session counts can lead to resource contention.

- **Threshold-based Alerts**

The script defines thresholds for each key metric (CPU, memory, disk I/O, network latency, etc.) and triggers Slack alerts via a webhook if thresholds are exceeded.

Slack integration - A simple but effective way to notify the team of performance issues in real time. This can be extended to other alerting mechanisms like email, SMS, or integrated monitoring solutions (e.g., Prometheus, Grafana, CloudWatch).

Logging - Metrics are logged in a structured JSON format for future analysis. This helps in creating historical data logs and trends for performance tuning and debugging. JSON is preferred due to its flexibility and ease of integration with log management tools.

Custom Monitoring Logic - Network latency and application bottlenecks are simulated as placeholders for real-world monitoring. These can be expanded by integrating real network latency checks (e.g., pinging external services).

1.2.2 Assumptions Made for the implementation.

Fixed Thresholds

- Threshold values are hardcoded for CPU (80%), memory (80%), disk I/O (1000 ops), network latency (100ms), application bottlenecks (0.5), and session management (200 sessions).

- These thresholds may not be representative of the actual system environment. They are assumed based on general performance characteristics but should be adjusted per the specifics of your application's workload, infrastructure, and SLAs (Service Level Agreements).

Simulated Bottlenecks

- Both application bottlenecks and session management logic are simulated rather than derived from real-time data. This assumes that any bottleneck logic exists and can be replicated, which might not always be accurate in real-world situations.

Basic Latency Check

- The network latency check is a static value (20ms), which is unrealistic in production systems where network performance fluctuates based on multiple factors (e.g., network congestion, service endpoint load).

1.2.3 Future Improvements for the Solution

Real-Time Visualization

Connect with tools like Prometheus, Grafana, or ELK (Elasticsearch, Logstash, Kibana) to create detailed dashboards. This will help analyze trends and visualize performance data over time.

Integration with Cloud Services

Update the script to also monitor cloud resources, such as those on AWS CloudWatch, in addition to local resources.

Improved Network Monitoring

Set up real-time checks for network delays using ICMP ping or services like Pingdom. Also, track packet loss and jitter to gain better insights into network performance.

Application-Level Monitoring

Integrate with Application Performance Management (APM) tools like New Relic to identify more issues within applications, such as slow database queries, memory leaks, and unoptimized code.

Session Management Enhancements

Improve session tracking by monitoring application or web server logs to see active sessions in real-time. This information can help with scaling decisions in environments that automatically adjust resources.

2. Optimization

The generation of a weekly performance report (e.g., using a cron job or scheduled task) summarizing the system's performance, including any incidents or bottlenecks.

This solution automates a weekly report that tracks important system metrics like network traffic, disk I/O, and application performance. It also connects with ServiceNow to gather incidents related to system performance, ensuring all relevant issues are included in the report.

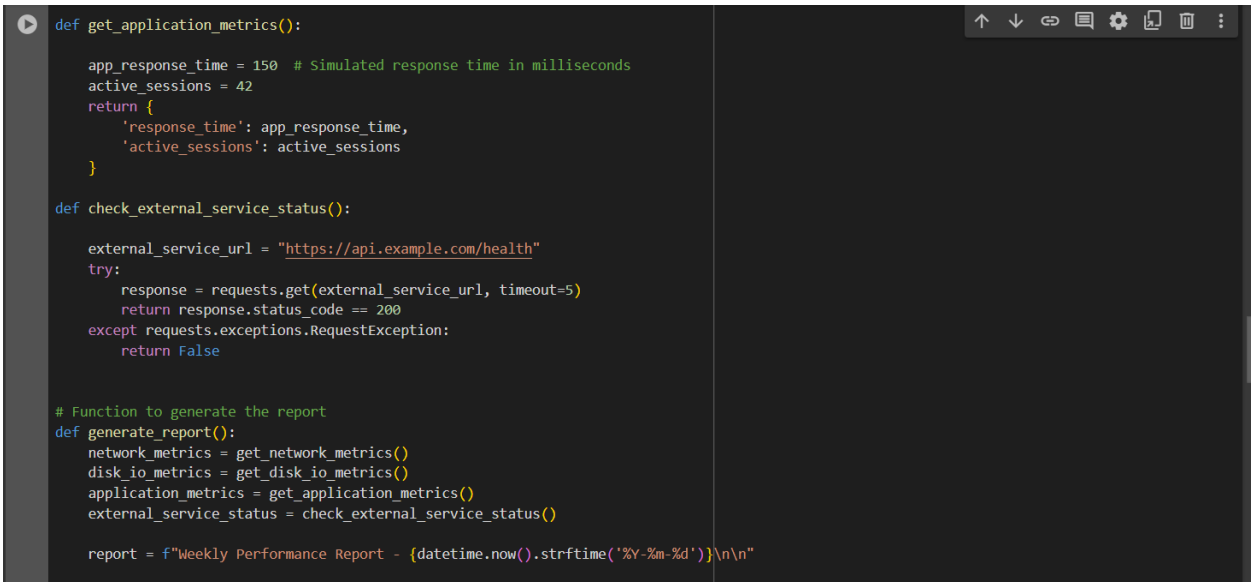
This offers a full overview of the system's health and any potential bottlenecks from the week.

```python
import os
import requests
import psutil
from datetime import datetime
from google.colab import drive

# Mount Google Drive
drive.mount('/content/drive')

# Function to get network metrics
def get_network_metrics():
    net_io = psutil.net_io_counters()
    return {
        'bytes_sent': net_io.bytes_sent,
        'bytes_recv': net_io.bytes_recv,
        'packets_sent': net_io.packets_sent,
        'packets_recv': net_io.packets_recv
    }

# Function to get Disk I/O metrics
def get_disk_io_metrics():
    disk_io = psutil.disk_io_counters()
    return {
        'read_bytes': disk_io.read_bytes,
        'write_bytes': disk_io.write_bytes,
        'read_count': disk_io.read_count,
        'write_count': disk_io.write_count
    }
```



```python
def get_application_metrics():

    app_response_time = 150  # Simulated response time in milliseconds
    active_sessions = 42
    return {
        'response_time': app_response_time,
        'active_sessions': active_sessions
    }

def check_external_service_status():

    external_service_url = "https://api.example.com/health"
    try:
        response = requests.get(external_service_url, timeout=5)
        return response.status_code == 200
    except requests.exceptions.RequestException:
        return False


# Function to generate the report
def generate_report():
    network_metrics = get_network_metrics()
    disk_io_metrics = get_disk_io_metrics()
    application_metrics = get_application_metrics()
    external_service_status = check_external_service_status()

    report = f"Weekly Performance Report - {datetime.now().strftime('%Y-%m-%d')}\n\n"
```

```
    # Network Metrics
    report += "Network Metrics:\n"
    report += f"Bytes Sent: {network_metrics['bytes_sent']}\n"
    report += f"Bytes Received: {network_metrics['bytes_recv']}\n"
    report += f"Packets Sent: {network_metrics['packets_sent']}\n"
    report += f"Packets Received: {network_metrics['packets_recv']}\n\n"

    # Disk I/O Metrics
    report += "Disk I/O Metrics:\n"
    report += f"Read Bytes: {disk_io_metrics['read_bytes']}\n"
    report += f"Write Bytes: {disk_io_metrics['write_bytes']}\n"
    report += f"Read Count: {disk_io_metrics['read_count']}\n"
    report += f"Write Count: {disk_io_metrics['write_count']}\n\n"

    # Application Metrics
    report += "Application Metrics:\n"
    report += f"Response Time: {application_metrics['response_time']} ms\n"
    report += f"Active Sessions: {application_metrics['active_sessions']}\n\n"

    # External Service Dependency Status
    report += "External Service Dependency Status:\n"
    report += f"External Service is {'UP' if external_service_status else 'DOWN'}\n\n"


    return report
```

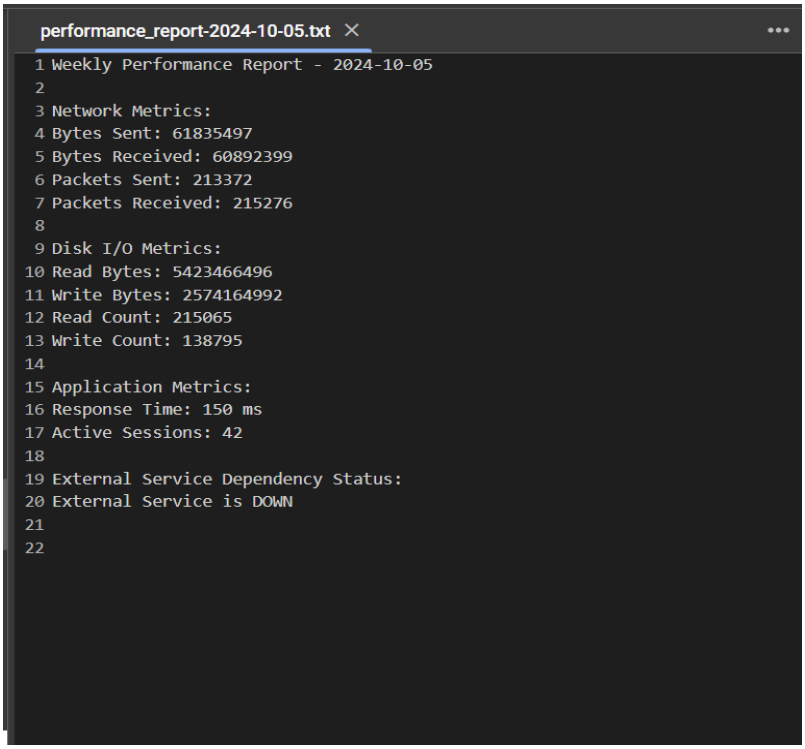
```
+ Code  + Text                                                    Reconnect  ✦ Gemini

# Function to save the report to Google Drive
def save_report_to_drive(report):
    file_name = f"/content/drive/My Drive/performance_report-{datetime.now().strftime('%Y-%m-%d')}.txt"
    with open(file_name, 'w') as file:
        file.write(report)
    print(f"Report saved to {file_name}")

# Main execution
if __name__ == "__main__":
    report = generate_report()
    save_report_to_drive(report)
```

Output

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
Report saved to /content/drive/My Drive/performance_report-2024-10-05.txt
```



This script is designed to collect and report key performance metrics from a multi-tier application, focusing on network, disk I/O, application, and external service status.

get_network_metrics(): Leverages the psutil library to gather network traffic metrics like bytes sent/received and packet counts.

get_disk_io_metrics(): Uses psutil to collect disk I/O operations, including read/write bytes and counts.

get_application_metrics(): Simulates collection of application-level metrics, such as response time and active session counts. This would typically come from a custom application performance management (APM) tool in a real-world scenario.

check_external_service_status(): Verifies the health of an external dependency by checking the status code of a predefined health-check URL.

Report Generation: The generate_report() function compiles all gathered metrics into a human-readable format, detailing the system's weekly performance. It aggregates the information in sections for network, disk I/O, application metrics, and external service status.

2.1 Assumptions Made

The script simulates application metrics like response time and active sessions, which are usually gathered by monitoring tools such as New Relic or Datadog. In a real environment, these metrics would come from a monitoring system's API.

While the script focuses on disk and network I/O metrics for assessing system performance, it overlooks CPU and memory usage, which can also cause performance issues.

Integrating the script with ServiceNow would allow automatic collection of incident data related to performance problems. Key details for each incident include:

- **Incident ID -** A unique identifier.
- **Description-** A brief summary of the performance issue.
- **Timestamp-** When the incident was reported.
- **Status-** Current status of the incident (open, in progress, on hold, or resolved).

This information can be added to the weekly performance report to give a complete view of any system incidents that occurred during the week.

Python code for the enhancement

```
def get_servicenow_incidents():

    servicenow_url = "https://your_instance.servicenow.com/api/now/table/incident"

    headers = {

        "Content-Type": "application/json",

        "Authorization": "Bearer YOUR_API_TOKEN"

    }

    params = {

        "sysparm_query": "category=system_performance^state!=resolved",

        "sysparm_fields": "number,short_description,opened_at"

    }

    try:

        response = requests.get(servicenow_url, headers=headers, params=params)

        response.raise_for_status()
```

```
        incidents = response.json().get('result', [ ])
        return incidents
    except requests.exceptions.RequestException as e:
        print(f"Error fetching incidents from ServiceNow: {e}")
        return [ ]
```

Integrating Incident Data into the Report: Once the incidents are retrieved, they can be formatted and added to the report:

```
def append_incident_data_to_report(report):
    incidents = get_servicenow_incidents()
    if incidents:
        report += "ServiceNow Incident Data:\n"
        for incident in incidents:
            report += f"Incident ID: {incident['number']}\n"
            report += f"Description: {incident['short_description']}\n"
            report += f"Opened At: {incident['opened_at']}\n\n"
    else:
        report += "No performance-related incidents were raised in ServiceNow during this period.\n\n"
    return report
```

3. Optional Enhancements

```python
import os
import time
import shutil
from zipfile import ZipFile
from datetime import datetime, timedelta

# Path to the log directory
log_dir = '/content/logs'  # You can create this directory in Colab to test
archive_dir = '/content/archived_logs'  # Directory where archived logs are stored

# Retention period for logs (in days)
log_retention_days = 7

# Ensure the archive directory exists
os.makedirs(archive_dir, exist_ok=True)

def archive_old_logs():
    # Get the current time
    now = time.time()

    # Define the retention threshold
    retention_threshold = now - log_retention_days * 86400  # 86400 seconds in a day

    for log_file in os.listdir(log_dir):
        log_path = os.path.join(log_dir, log_file)
```

```python
        # Check if the log file is older than the retention period
        if os.path.isfile(log_path) and os.path.getmtime(log_path) < retention_threshold:
            # Create a zip archive of the log file
            zip_name = f"{os.path.splitext(log_file)[0]}.zip"
            zip_path = os.path.join(archive_dir, zip_name)

            with ZipFile(zip_path, 'w') as zipf:
                zipf.write(log_path, os.path.basename(log_path))
                print(f"Archived: {log_file} -> {zip_name}")

            # Remove the original log file after archiving
            os.remove(log_path)
            print(f"Deleted: {log_file}")

# Function to delete archives older than retention period
def clean_archived_logs():
    now = time.time()
    retention_threshold = now - log_retention_days * 86400

    for archive_file in os.listdir(archive_dir):
        archive_path = os.path.join(archive_dir, archive_file)

        # Check if the archive is older than the retention period
        if os.path.isfile(archive_path) and os.path.getmtime(archive_path) < retention_threshold:
            os.remove(archive_path)
            print(f"Deleted archive: {archive_file}")
```

```python
# Run the log management process
print("Archiving old logs...")
archive_old_logs()

print("\nCleaning old archives...")
clean_archived_logs()

print("\nLog rotation and cleanup completed.")
```

Output through the script

```
Archiving old logs...

Cleaning old archives...

Log rotation and cleanup completed.
```

The above script is for the log rotation process.