UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

MARCOS GUILLERMO DE SÁ CATÃO COSSON

PERFORMANCE EVALUATION OF OPENSTACK SWIFT

CAMPINA GRANDE - PB

2024

MARCOS GUILLERMO DE SÁ CATÃO COSSON

PERFORMANCE EVALUATION OF OPENSTACK SWIFT

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador : Thiago Emmanuel Pereira

CAMPINA GRANDE - PB

2024

MARCOS GUILLERMO DE SÁ CATÃO COSSON

PERFORMANCE EVALUATION OF OPENSTACK SWIFT

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Trabalho aprovado em: 17/05/2024

CAMPINA GRANDE - PB

PERFORMANCE EVALUATION OF OPENSTACK SWIFT

ABSTRACT

This study addresses the need for a comprehensive evaluation of Swift's performance in OpenStack environments. Despite its widespread adoption, there is a lack of in-depth analysis in this area. Through testing, including load and stress tests, we examined Swift's behavior across varying levels of demand. Our findings highlight key trends: as user load increased, response times showed a corresponding rise, reaching a maximum observed increase. Moreover, the variability in response times also expanded with higher user loads, emphasizing the importance of scalability and performance optimization for Swift in practical deployment scenarios.

Performance Evaluation of Openstack Swift

Marcos Guillermo Cosson
marcos.cosson@ccc.ufcg.edu.br

Thiago Emmanuel Pereira
temmanuel@computacao.ufcg.edu.br

ABSTRACT

This study addresses the need for a comprehensive evaluation of Swift's performance in OpenStack environments. Despite its widespread adoption, there is a lack of in-depth analysis in this area. Through testing, including load and stress tests, we examined Swift's behavior across varying levels of demand. Our findings highlight key trends: as user load increased, response times showed a corresponding rise, reaching a maximum observed increase. Moreover, the variability in response times also expanded with higher user loads, emphasizing the importance of scalability and performance optimization for Swift in practical deployment scenarios.

Keywords

Swift, OpenStack, object store, performance evaluation, system bottlenecks, cloud infrastructure, workload.

1. INTRODUCTION

Swift, an OpenStack service for object storage, operates within a model where each unit is treated as a distinct object, identified by a unique key. Its architecture comprises three main components: the Proxy Server, the Storage Cluster, and the Swift Rings. The Proxy Server manages client requests, directing them to the appropriate storage nodes, while the Storage Cluster stores and serves data objects, with the Swift Rings ensuring their scalable and resilient distribution[1].

Despite its significance, there remains a gap in the literature concerning Swift's performance metrics, impeding decision-making for OpenStack platform users. This study addresses this gap, beginning with a detailed overview of Swift's architecture and operational dynamics in the Background section. Here, we explore the roles of its components.

Following this, the Methodology section discusses the steps adopted and tools utilized to create a stress testing environment. The results yielded insights into Swift's behavior under varying demand levels. We observed an increase in median response time as demand increased. Additionally, wider quartile ranges in high-load scenarios suggested increased variability in response times, necessitating performance management strategies. The identification of outliers indicated potential system instabilities or bottlenecks, underscoring the need for actions to ensure consistent service delivery. Finally, in Section 5, we summarize the findings and discuss their implications for the practical deployment of Swift in production environments.

2. BACKGROUND

This section introduces Swift, an essential part of OpenStack, offering scalable object storage. It outlines Swift's architecture, including rings, partitions, proxy servers, and storage nodes. The section also covers key functionalities like data replication, consistency, and controlled distribution. Practical examples illustrate Swift's usage, such as retrieving and adding objects, showcasing its structure and versatility within cloud environments.

Swift is an object storage system, designed to store and retrieve large volumes of unstructured data across a variety of scenarios. It is one of the core components of the OpenStack cloud computing platform, providing a scalable and redundant storage solution. This capability is accessed through a Representational State Transfer (RESTful) API, enabling clients to interact programmatically with Swift.

Below, we provide a practical example of how to use the Swift API to create and retrieve an object:

```
# Step 1: Authentication and Token Retrieval
TOKEN=$(curl -X POST -d '{"username": "your username",
"password": "your-password"}'
htttp://your-end-point-of-authentication/auth | jq -r
'.token')

# Step 2: Creating a new Object
curl - X PUT -H "X-Auth-Token: $TOKEN" - T path/to/file.txt
http://your-swift-endpoint.com/v1/your-container/file.txt
#Step 3: Retrieving the Created Object
curl - X GET -H "X-Auth-Token: $TOKEN"
http://http://your-swift-endpoint.com/v1/your-container/my_
file.txt > my_retrieved_file.txt
```

Example of Swift API usage [2]

In this example, Swift is accessed through the RESTful API, enabling developers to seamlessly integrate Swift into their applications and data management systems.
The token is essential for authenticating and authorizing clients to access the Swift cluster. It acts as a unique authentication credential. After providing their user credentials, the authentication service generates and returns a unique token.

2.1. Rings and Partitions

In the context of Swift, a 'ring' functions as an organizational data structure that coordinates the distribution of data among storage nodes within a cluster. A ring contains details regarding the precise locations of each node and defines the 'partitions' of data that each node oversees, as illustrated in Figure 3.

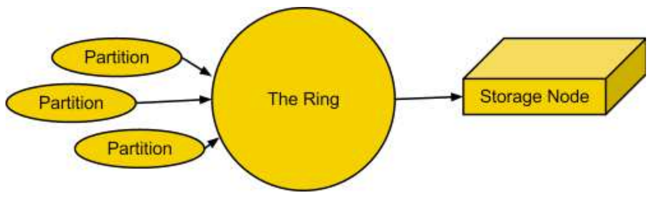

Figure 3: Partition and Ring Operation in the context of Swift[3]

In its turn, 'partitions,' are discrete data units with predetermined sizes, each one is designated to a particular storage node within the cluster. The efficacy of this approach resides in its capacity to streamline data distribution and enhance data management efficiency across the entirety of the cluster.

2.2 **Proxy Server**

The Proxy server acts as the gateway for clients to interface with the Swift cluster. It manages incoming requests, including read and write operations, and directs them to the relevant storage nodes. Moreover, the proxy server undertakes load balancing, authentication, and authorization validations, guaranteeing that clients engage with the appropriate nodes and possess requisite permissions.

2.3 **Storage Nodes**

Building upon the previous discussion of Swift's architecture, let's now shift our focus to the storage nodes. These nodes are responsible for storing and serving data objects, while simultaneously managing their assigned partitions. To ensure data resilience and persistence, they implement replication mechanisms. Each storage node utilizes a file system to house data objects. A key characteristic of Swift is its straightforward approach to object naming. Rather than employing complex hierarchical directories, Swift adopts a flat naming convention, resulting in simplified data organization. For a visual representation of storage node operations, please refer to Figure 4.

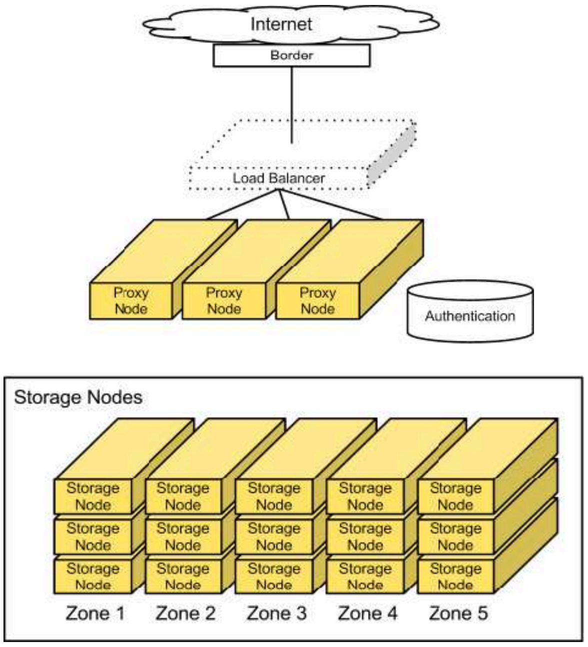

Figure 4: Object Storage Swift architecture[3]

2.4 **Data Replication and Distribution**

Data durability is a crucial aspect of Swift's architecture. The data in each partition is replicated across multiple storage nodes to ensure redundancy. Swift utilizes a distributed hash algorithm to determine which nodes should host replicas of a particular partition. Replication ensures that if a node becomes unavailable or a disk fails, the data remains accessible from other nodes. Figure 5 illustrates the behavior described above.

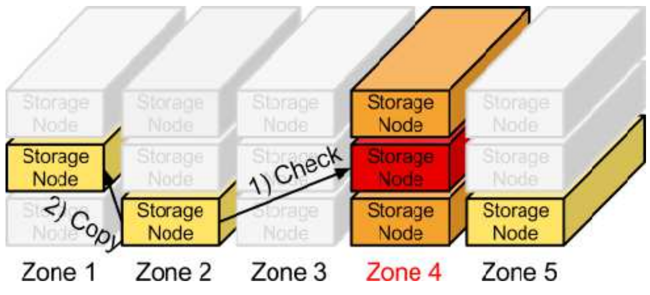

Figure 5: Swift replication scheme[3]

2.5 **Data Consistency**

Swift operates under an eventual consistency model, which implies that updates made to the system may not be instantly reflected in all replicas. Eventual consistency, in the context of Swift, means that when data updates occur, the replicas will, over time, adjust to achieve a coherent state. However, it is important to note that this gradual synchronization may take some time to materialize. Although this approach offers advantages in terms of high availability and scalability, it is crucial for applications using Swift to be designed considering this eventual consistency nature.

2.6 **Controlled Replication and Distribution**

In Swift, administrators can utilize Data Placement Groups (DPGs) to configure how data is replicated and placed in the cluster. These policies allow adjustments such as replica count and distribution based on hardware and performance considerations, optimizing storage efficiency and cluster performance.

3. METHODOLOGY

This section explains the methodology employed to perform a Load Test on Swift 4. The primary objective of this test was to assess Swift's responsiveness to diverse demands, encompassing scalability, stability, and response time. The testing environment was configured within a dedicated OpenStack infrastructure.

3.1 **Objective**

The primary objective of the test is to determine the behavior of the Swift service under varying workload, focusing on response time, for evaluating the time the service takes to respond to object download requests, and quantifying system latency.

3.2 **Experiment Design**

The experiment conducted was a load test, where the workload was synthetically generated, creating and storing 5MB files.

- **Independent Variable:** Generated workload, varying the number of virtual users (VUs) and test execution time to analyze the impact in different scenarios.
- **Dependent Variable:** Service response time for object download requests.

The testing process relied on a suite of tools and instruments. We used Locust to generate workload on the Swift service.

3.3 **Test Environment**

The test environment used was configured on a dedicated OpenStack infrastructure with the following characteristics:

Internet Bandwidth:

- 10Gbps network connection for Ceph
- Maximum observed bandwidth over the past two months: 800MB/s

Software:

- Operating System: Ubuntu 22.04.2 LTS (VM)
- Swift: python-swiftclient 3.13.1 stable version of the service to be tested.
- Locust: v. 2.24.1, load testing tool for simulating virtual users.

Hardware(VM):

CPU:

- Model: Intel Core Processor (Haswell, no TSX, IBRS)
- Cores: 4 (1 thread por core)
- Speed: 2297.340 MHz
- Cache: L1d: 128 KiB, L1i: 128 KiB, L2: 16 MiB, L3: 64 MiB

Storage:

- Total: 20 GB
- Partitions:
    - vda1: 19.9 GB (mounted on /)
    - vda14: 4 MB
    - vda15: 106 MB (mounted on /boot/efi)

RAM:

- Total: 7.8 GiB
- Used: 210 MiB
- Free: 5.5 GiB
- Available: 7.2 GiB

3.4. **Test Procedures**

3.4.1 **Authentication and Configuration**

The authentication process for the Swift service was conducted, leveraging a .rc configuration file provided by OpenStack to ensure secure access. Subsequently, the authentication token was exported for integration into the tests, thereby enabling communication with the service. Moreover, configuration of a CSV file was undertaken to record test results, thereby facilitating data collection and subsequent analysis.

3.4.2 **Test Execution**

1. Simulating VUs with Locust to perform downloads of objects from the Swift service, replicating the behavior of users.
2. Each VU executes a task that consists of downloading a random object from the list of objects, ensuring a test scenario.
3. Waiting time between tasks set between 1 and 2 seconds, simulating behavior between requests.
4. Each test is executed at a time (--run-time) and with a number of VUs (-u), allowing the analysis of the workload impact on different configurations.

3.4.3. **Test Strategy**

An approach was employed, commencing with a workload and augmenting the number of virtual users (VUs) and execution time to pinpoint service breakpoints. This method entailed an increment of VUs while sustaining a generation rate (-r 1), ensuring an increment in the workload. Furthermore, the extension of execution time in each iteration facilitated system stabilization and assessment of service behavior over periods, affording an understanding of performance under load. The objective was to subject the service to a range of load scenarios to discern performance constraints and bottlenecks, thereby ensuring the service's resilience.

3.4.4 **Test Scenario Table**

It is important to highlight that the tests followed a load progression in accordance with the processing, reading, and storage capacity of the VM used to run the experiments. The endeavor sought to stress Swift within the defined test time frame while avoiding potential interruptions due to CPU, RAM, and storage overload on the VM executing the tests. Furthermore, the tests were conducted at a scheduled time agreed upon with the responsible cloud engineer to enable monitoring of the infrastructure and prompt response to any incidents. Subsequently, we will present in detail a table containing the test scenarios, the number of simultaneous virtual users, and the corresponding execution times.

Tests Scenarios		
Scenario	**Number of VUs**	**Execution Time**
1	10	70
2	50	130
3	100	150
4	150	170

Figure 6: A scenario test, number of virtual users and execution time

4. RESULTS

In this section, the results obtained from the analysis of the tests conducted with the Swift service are presented.

4.1. **Analysis of Response Time**

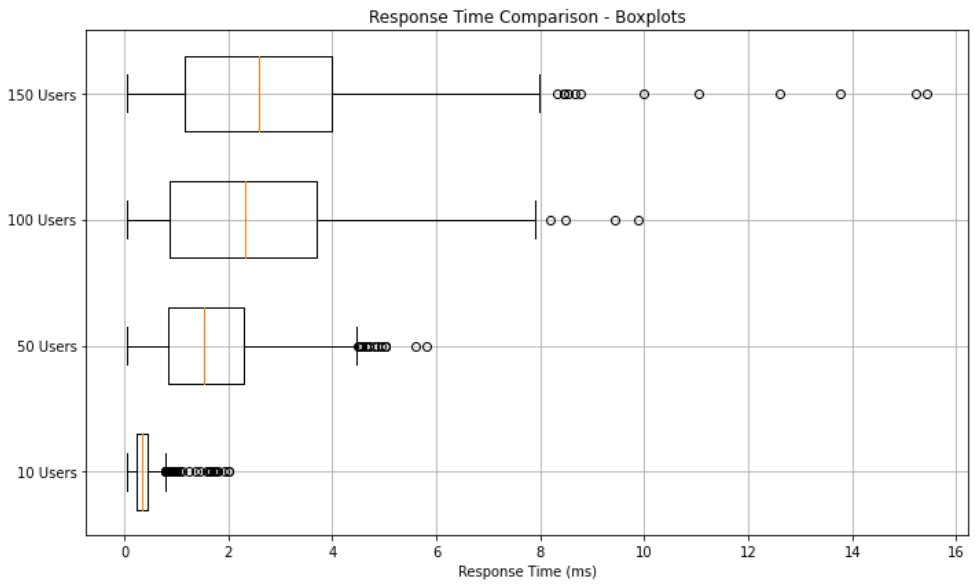

Figure 7: Response time for the Load Test scenarios

Figure 7, illustrates the response time of the Swift service in different load scenarios, ranging from 10 to 150 virtual users (VUs). Each entry in the table represents a specific metric, including the mean, standard deviation, minimum and maximum values, as well as the 25th, 50th (median), and 75th quartiles. The summary of findings indicates that the performance of the Swift service gradually declines as the workload escalates, particularly noticeable in scenarios 3 and 4, suggesting limitations in handling higher demands. Furthermore, the presence of outliers in these scenarios signifies potential instabilities in the system under heavy load, potentially compromising user experience and service reliability.

Next, we will proceed with an exploration of the patterns and conclusions revealed by these data, emphasizing the relevance of each metric in evaluating the performance of the Swift service, as depicted in the table in Figure 8.

Load Test Results Summary Table							
Users (VU's)	Mean (ms)	STD (ms)	Min (ms)	25% (ms)	50% (ms)	75% (ms)	Max (ms)
10	0.37	0.21	0.05	0.23	0.34	0.46	2.01
50	1.63	1.01	0.04	0.83	1.55	2.29	5.81
100	2.64	1.87	0.06	1.08	2.43	3.76	11.16
150	3.26	2.33	0.05	1.16	3.12	4.86	20.15

Figure 8: Summary of Test Results(in milliseconds)

Firstly, a gradual increase in median response time is evident as the number of virtual users (VUs) escalates from 10 to 150. This consistent upward trend underscores the growing demand placed on the Swift service, resulting in an increase in the average time taken to process requests. Such observations emphasize the critical importance of understanding system responsiveness and scalability across diverse workloads. Secondly, alongside the rise in median response time, there is a widening range of quartiles observed in scenarios with higher load levels. This widening quartile range indicates a greater dispersion of data points, reflecting increased variability in response times. The broader quartile range accentuates the necessity for performance monitoring and management strategies to address potential fluctuations in service performance effectively. Furthermore, the identification of outliers in scenarios with 100 and 150 VUs. These outliers represent instances where the Swift service experienced unusually high response times, potentially indicative of system instabilities or bottlenecks under heavier loads. Addressing these outliers is imperative to ensure consistent and reliable service delivery to users, underscoring the importance of proactive performance optimization measures.

4.2 **Validity threats to experiments**

Our experiments face some validity threats that may affect the reliability and generalizability of the results. These threats include internal validity, external validity, construct validity, and conclusion validity.

Internal validity threats include confounding variables, selection bias, and instrumentation. Confounding variables, such as other processes running on the same virtual machine (VM) or network, can interfere with performance measurements. A dedicated testing environment can mitigate this. Selection bias can occur if the VM configurations and workload patterns are not representative. While we believe our microbenchmark was a good choice for a load test, experimenting with different system configurations would improve our work (we considered only the configuration used in production in our testbed).

External validity threats include environmental differences, population validity, and temporal validity. Environmental differences, such as specific hardware and network conditions in our test environment, may not match other environments where

Swift is used. Conducting experiments in varied environments can enhance external validity.

By addressing these threats through careful experimental design, execution, and analysis, we aim to ensure that our findings on OpenStack Swift's performance are reliable and applicable to various scenarios.

5. CONCLUSION AND FUTURE WORK

Challenges and implications encompass the complexity introduced by integrating authentication during tests, potentially impacting other services besides Swift. Additionally, the tests were designed to simulate stress scenarios, with results indicating satisfactory performance in scenarios 1 and 2. However, the significant increase in outliers in scenarios 3 and 4 underscores potential scalability limitations under more intense loads.

Further considerations include the necessity for detailed outlier analysis to discern underlying causes of response time spikes and the importance of benchmarking comparison to evaluate competitiveness and identify improvement areas. Scalability assessment through tests on different infrastructures and configurations is also recommended.

In conclusion, the performance tests underscore a gradual decline in average response time with escalating workloads, signaling potential scalability challenges. It is essential to explore underlying issues, implement corrective measures, and establish monitoring mechanisms to uphold optimal Swift service performance.

Another aspect that needs attention is scalability testing across infrastructures and configurations. By scrutinizing its scalability across environments, ranging from cloud platforms to on-premises hardware, organizations can ascertain the service's adaptability and resilience in meeting demands.

6. ACKNOWLEDGMENTS

I would like to express my sincere gratitude to all who have contributed to the completion of this work. Firstly, I thank myself for persevering through various challenges and successfully concluding this journey alongside my undergraduate studies in computer science. Next, I am immensely grateful to my advisor, Thiago Emmanuel Pereira, for his guidance, collaboration in this work, and sharing of knowledge and experience. I deeply thank God and my grandmother, Maria Marlene de Sá, for providing exceptional familial upbringing and unwavering support. I also acknowledge the positive influence of my grandfather, Anésio Figueira Catão, and my mother, Maria das Neves de Sá Catão, whose resilience and guidance have been invaluable throughout my academic journey.

To my friends Hiarly Souto, Jonatas Ferreira, Kleber Reudo, João Vitor, and Luiz Gustavo, I am deeply grateful for their constant support. Without them, this graduation journey wouldn't have been as enriching. I also extend my gratitude to Victor, Samuel, Paola, and Eduarda — friends I made during this period who were also indispensable on this journey — among others. Additionally, I am thankful to the LSD support team for their patience and assistance in imparting crucial knowledge for the completion of this work.

Finally, I would like to pay a special tribute to my dear friend Adriano dos Santos Lira, whose memory will be forever cherished. Although he is no longer with us, his remarkable friendship and the moments we shared will remain etched in my heart.

7. REFERENCES

[1] https://avcourt.github.io/tiny-cluster/2019/04/20/openstack_swift.html

[2] Cosson, Marcos Guillermo de Sá Catão. (2023). Tutorial: How to Integrate with Swift Component - OpenStack Documentation. [Online]. Available at: https://github.com/MarcosDaNight/automated_load_transfer. Accessed on: February 20, 2024.

[3] https://docs.openstack.org/swift/latest/admin/objectstorage-components.html

[4] GREGG, Brendan. Systems Performance: Enterprise and the Cloud. 3. ed. Upper Saddle River, Nj: Prentice Hall, 2014. 15 - 82 p.

8. APPENDIX

APPENDIX 1 - Payload Generation

```
import os
import random

def generate_payload_file(file_name, size_mb):
    with open(file_name, 'wb') as file:
        file.write(os.urandom(size_mb * 1024 * 1024))

def generate_simulated_payloads(output_dir, num_files,
min_size_mb, max_size_mb):
    for i in range(num_files):
        size_mb = random.randint(min_size_mb, max_size_mb)
        file_name = os.path.join(output_dir,
f'payload_{i}.txt')
        generate_payload_file(file_name, size_mb)

# Example usage:
generate_simulated_payloads('simulated_loads', 10, 1, 10)
```

APPENDIX 2 - Swift Object Upload Script

```
#!/bin/bash

SWIFT_CONTAINER_IP=”<your container ip>”
SWIFT_CONTAINER_NAME="<your container name>"
SWIFT_URL="http://${SWIFT_CONTAINER_IP}:<Port>/v1//${SWIFT_
AUTH_TOKEN}/${SWIFT_CONTAINER_NAME}"

for file in simulated_loads/*; do
    if [ -f "$file" ]; then
        echo "Sending $file"
        curl -X PUT -H "X-Auth-Token: $(openstack token
issue -f value -c id)" \
             -T "$file" \
             "${SWIFT_URL}$(basename "$file")"
    fi
done
```

APPENDIX 3 - Swift Download Stress Test Script

```
from locust import HttpUser, task, between
import requests
import time
import os
import atexit

# Creation of the downloads folder
downloads_dir = os.path.join(os.path.dirname(__file__),
"downloads")
if not os.path.exists(downloads_dir):
         os.makedirs(downloads_dir)

class SwiftDownloader(HttpUser):
         wait_time = between(1, 2)
         host = 'your swift url'

         @task
         def download_payload(self):
         with open('list_swift.txt', 'r') as file:
         objects = [line.strip() for line in file]

         for object_name in objects:
         url = f'{self.host}/{object_name}'
         headers = {'X-Auth-Token':
os.getenv('OS_AUTH_TOKEN')}

         try:
                  start_time = time.time()
                  response = self.client.get(url,
headers=headers)
                  end_time = time.time()

                  download_time = end_time - start_time

                  if response.status_code == 200:
                  timestamp = int(time.time() * 1000)  #
Milliseconds timestamp
                  filename =
f"{object_name}_{timestamp}.txt"  # Filename with timestamp
                  with open(os.path.join(downloads_dir,
filename), 'wb') as f:
                           f.write(response.content)
                  else:
                  print(f"Error downloading object:
{object_name} - {response.status_code}")

                  # Error logging to file
                  with open('errors.log', 'a') as f:

f.write(f"{object_name},{response.status_code}\n")

                  # Statistics logging to file
                  with open('statistics.csv', 'a') as f:

f.write(f"{object_name},{download_time},{response.status_co
de}\n")

         except requests.exceptions.RequestException as e:
                  print(f"Error downloading object:
{object_name} - {e}")

                  # Error logging to file
                  with open('errors.log', 'a') as f:
                  f.write(f"{object_name},{e}\n")

# Cleanup function
def cleanup():
         for filename in os.listdir(downloads_dir):
         os.remove(os.path.join(downloads_dir, filename))

atexit.register(cleanup)  # Register function to be
executed when exiting the script
```