# Introduction to Cloud Computing Report

**Lab8: Istio Traffic Management Hands-on Lab**

Name: Yin Yuang
Student ID: 202383930027
Class: Software Engineering Class 1

Introduction to Cloud Computing
(Autumn,2025)

Nanjing University of Information Science and Technology, China
Waterford Institute

# I. Object

The objective of this lab is to gain hands-on experience with Istio's traffic management capabilities in a Kubernetes environment, including traffic shifting, request routing, fault injection, and circuit breaking. By completing this lab, students will be able to:

- Configure Istio VirtualService and DestinationRule objects to control traffic distribution between different service versions.
- Implement request routing rules based on headers, URI paths, and other attributes.
- Apply fault injection policies to simulate service failures and test system resilience.
- Enable circuit breaking to prevent cascading failures and manage service availability.
- Observe and verify the effects of traffic management policies through telemetry and Istio dashboards.

This lab aims to demonstrate how Istio decouples traffic management logic from application code, enabling fine-grained control over service-to-service communication in a microservices architecture.

# II. Experimental Procedure

This section describes the experimental procedures followed to evaluate Istio's traffic management capabilities. Each subsection corresponds to one task defined in the official Istio traffic management documentation, and outlines the configuration steps and execution workflow. Detailed experimental results and screenshots are presented in Section III.

## 1. Traffic Shifting

Traffic shifting was performed by configuring Istio `VirtualService` resources to control the distribution of requests among different service versions. Initially, all traffic was routed to a single version of the `reviews` service to establish a baseline behavior.

Subsequently, weighted routing rules were applied to gradually shift traffic between multiple versions of the service. During this process, the application behavior was observed through repeated access to the Bookinfo product page, allowing verification of dynamic traffic migration without redeploying services or modifying application code.

## 2. Request Routing

Request routing rules were configured to direct traffic based on request attributes such as HTTP headers and user identity. A user-based routing policy was implemented to route requests from a specific user to a designated version of the `reviews` service.

This procedure demonstrated how Istio can selectively route traffic for targeted users, enabling scenarios such as A/B testing, canary deployments, and feature validation under real traffic conditions.

## 3. Fault Injection

Fault injection experiments were conducted using Istio `VirtualService` configurations to introduce controlled failures into service communication. Both delay-based faults and HTTP abort faults were applied to simulate network latency and service unavailability.

Requests affected by the injected faults were monitored through the application interface and command-line tools. These steps allowed the evaluation of system behavior under partial failures and helped identify timeout and retry limitations in the application configuration.

## 4. Circuit Breaking

Circuit breaking was implemented using Istio `DestinationRule` resources to enforce strict connection pool limits and enable outlier detection. A backend service was deployed along with a load-testing client to generate concurrent requests.

Load tests were executed with increasing levels of concurrency to intentionally exceed the configured limits. The system behavior was observed as Istio began rejecting excessive requests, thereby protecting the backend service from overload and preventing cascading failures.

### 5. Observability and Verification

Throughout the experiment, system behavior was continuously verified using Istio telemetry data and command-line inspection tools. Metrics and proxy statistics were queried to confirm the effectiveness of traffic management policies such as request rejection and pending request overflows.

Screenshots were captured at each major step to document configuration actions, system responses, and experimental outcomes, providing visual evidence to support the analysis presented in the results section.

## III. Results

This section presents the experimental results of four Istio traffic management mechanisms: traffic shifting, request routing, fault injection, and circuit breaking. All experiments were conducted in the previously configured Istio-enabled Kubernetes environment using the Bookinfo sample application, except for the circuit breaking experiment, which employed the `httpbin` and `fortio` services for controlled load testing.

Each subsection reports the observed system behavior, followed by an interpretation that links the results to the corresponding Istio configuration.

### 1. Traffic Shifting

The objective of traffic shifting is to gradually migrate client requests from one service version to another in a controlled manner. Initially, all traffic was routed to `reviews:v1`.

First, destination rules for all Bookinfo services were applied to define available subsets:

```
kubectl apply -f samples/bookinfo/networking/destination-rule-all.yaml
```

Figure 1 confirms that the destination rules were successfully applied.

```
C:\Users\21534\Downloads\istio-1.27.2-win-amd64\istio-1.27.2>kubectl apply -f samples/bookinfo/networking/virtual-service-all-v1.yaml
virtualservice.networking.istio.io/productpage created
virtualservice.networking.istio.io/reviews created
virtualservice.networking.istio.io/ratings created
virtualservice.networking.istio.io/details created

C:\Users\21534\Downloads\istio-1.27.2-win-amd64\istio-1.27.2>kubectl apply -f samples/bookinfo/networking/virtual-service-reviews-50-v3.yaml
virtualservice.networking.istio.io/reviews configured

C:\Users\21534\Downloads\istio-1.27.2-win-amd64\istio-1.27.2>kubectl get virtualservice reviews -o yaml
apiVersion: networking.istio.io/v1
kind: VirtualService
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"networking.istio.io/v1","kind":"VirtualService","metadata":{"annotations":{},"name":"reviews","namespace":"default"},"spec":{"hosts":["
reviews"],"http":[{"route":[{"destination":{"host":"reviews","subset":"v1"},"weight":50},{"destination":{"host":"reviews","subset":"v3"},"weight":50}]}]}}
  creationTimestamp: "2025-10-21T07:06:20Z"
  generation: 2
  name: reviews
  namespace: default
  resourceVersion: "54866"
  uid: 7c4ad6d0-915b-4d48-ad00-c2836c97416a
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v1
      weight: 50
    - destination:
        host: reviews
        subset: v3
      weight: 50
```

Figure 1: Applying destination rules for all Bookinfo services.

Next, a virtual service was applied to route 100% of traffic to version 1 of each service:

```
kubectl apply -f samples/bookinfo/networking/virtual-service-all-v1.yaml
```

As shown in Figure 2, the product page displays reviews without rating stars. This is expected behavior because `reviews:v1` does not invoke the ratings service.

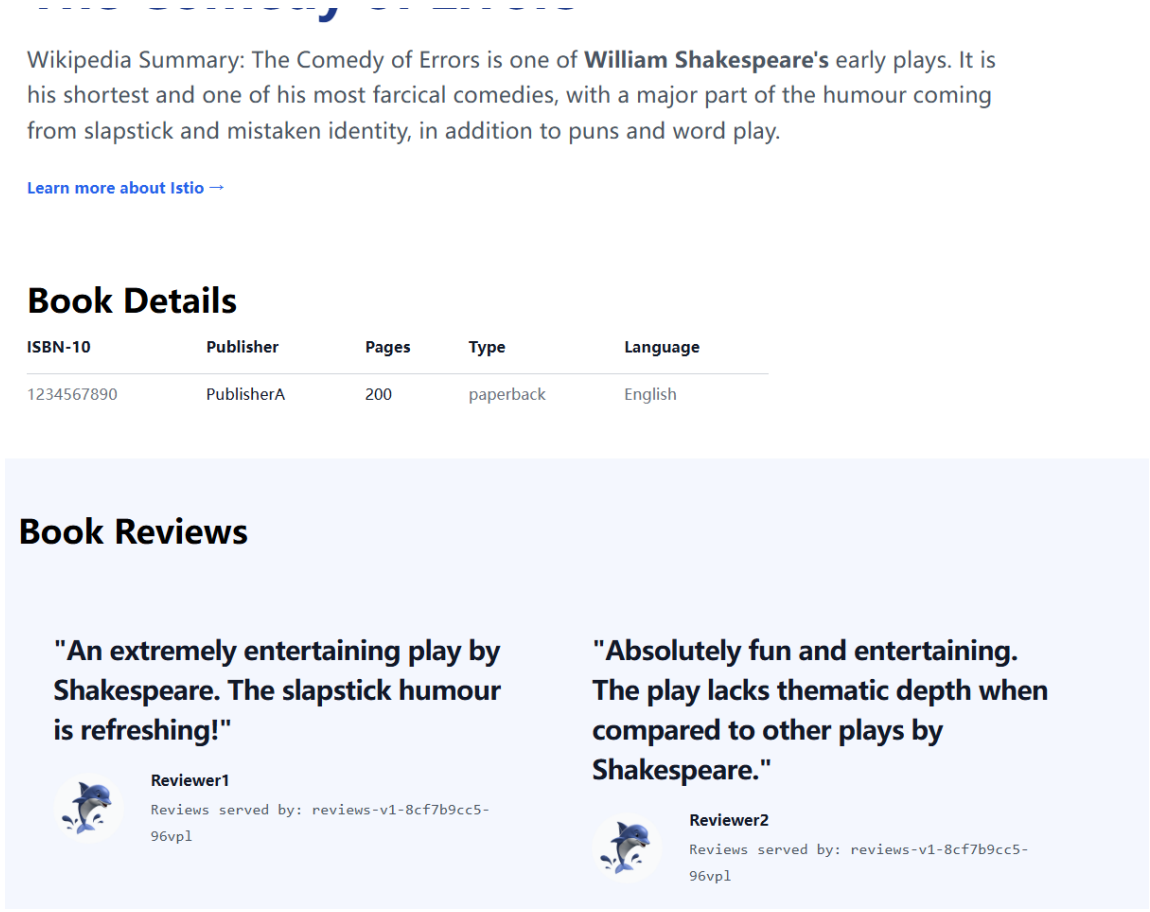


Figure 2: All traffic routed to `reviews:v1`; rating stars are not displayed.

To demonstrate gradual traffic migration, traffic was split evenly between `reviews:v1` and `reviews:v3`:

```
kubectl apply -f samples/bookinfo/networking/virtual-service-reviews-50-v3.yaml
```

With this configuration, refreshing the product page alternates between responses served by different service versions, as illustrated below.

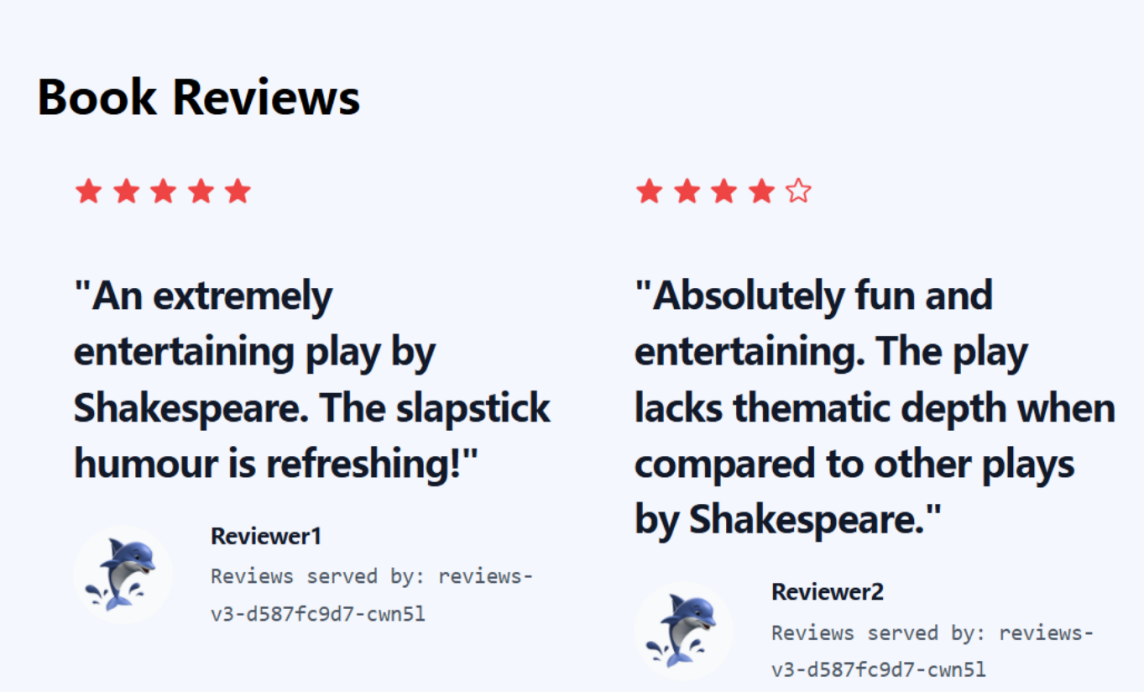


Figure 3: 50% traffic routed to `reviews:v1`, displaying no rating stars.

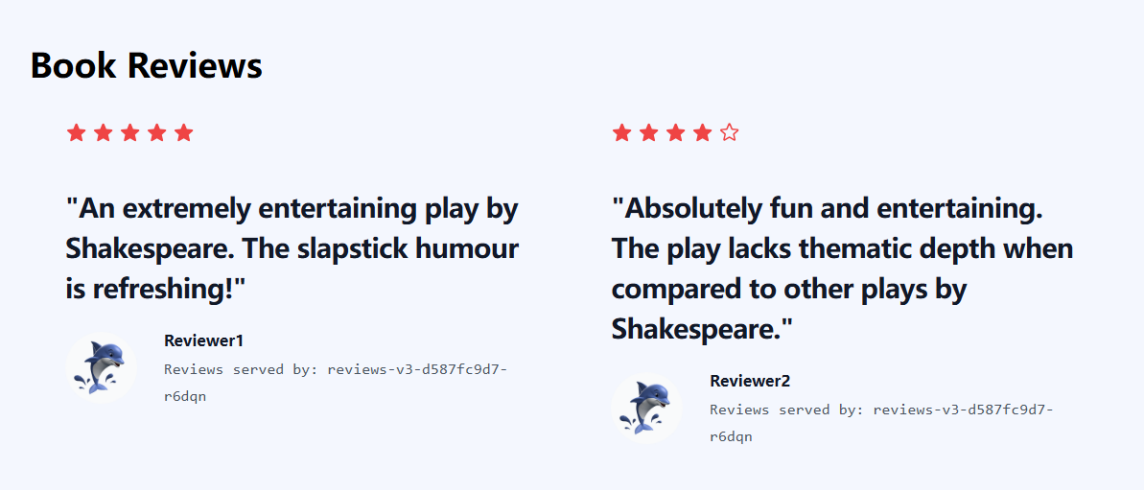


Figure 4: 50% traffic routed to `reviews:v3`, displaying red rating stars.

After confirming the correctness and stability of `reviews:v3`, all traffic was migrated to this version:

```
kubectl apply -f samples/bookinfo/networking/virtual-service-reviews-v3.yaml
```

Figure below confirms that the updated routing rule was successfully applied.


Figure 5: Routing 100% of traffic to `reviews:v3`.

Finally, the traffic routing rules were removed to restore the default configuration:

```
kubectl delete -f samples/bookinfo/networking/virtual-service-all-v1.yaml
```

```
C:\Users\21534\Downloads\istio-1.27.2-win-amd64\istio-1.27.2>kubectl delete -f samples/bookinfo/networking/virtual-service-all-v1.yaml
virtualservice.networking.istio.io "productpage" deleted from default namespace
virtualservice.networking.istio.io "reviews" deleted from default namespace
virtualservice.networking.istio.io "ratings" deleted from default namespace
virtualservice.networking.istio.io "details" deleted from default namespace
```

Figure 6: Deletion of traffic shifting rules.

## 2. Request Routing

Request routing enables traffic to be directed based on request attributes, such as user identity. Initially, all requests were routed to `reviews:v1`:

```
kubectl apply -f samples/bookinfo/networking/virtual-service-all-v1.yaml
```

Figure below confirms that the product page displays no rating stars.

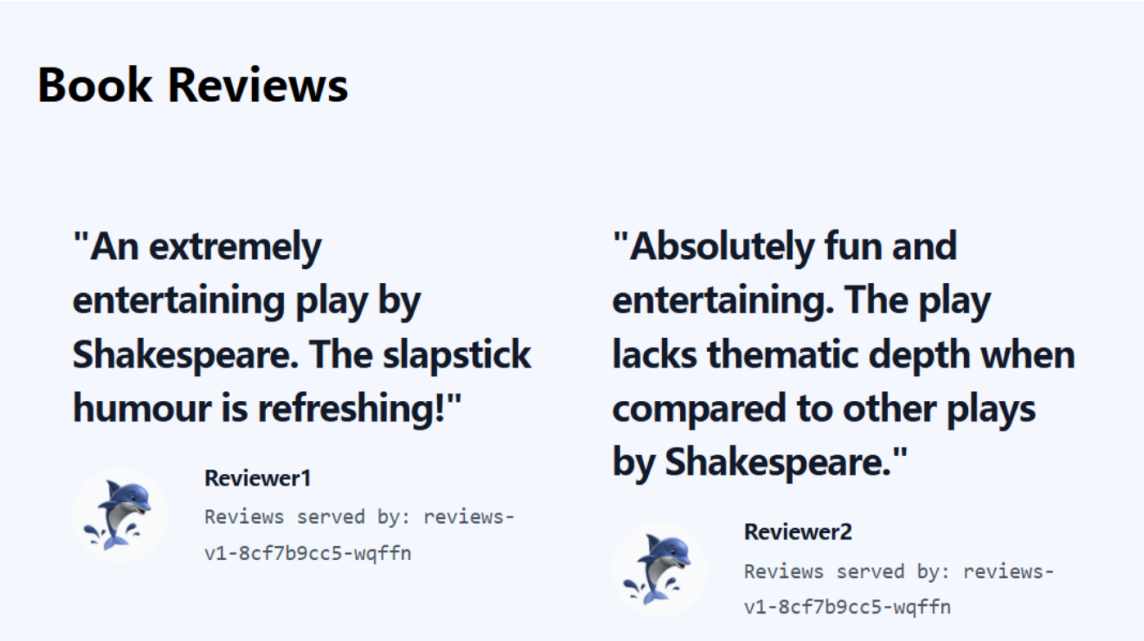


Figure 7: Default routing to `reviews:v1`.

Subsequently, a user-based routing rule was applied so that requests from user `jason` were routed to `reviews:v2`, which includes rating stars.

Figure 8 shows the login operation used to trigger this rule.

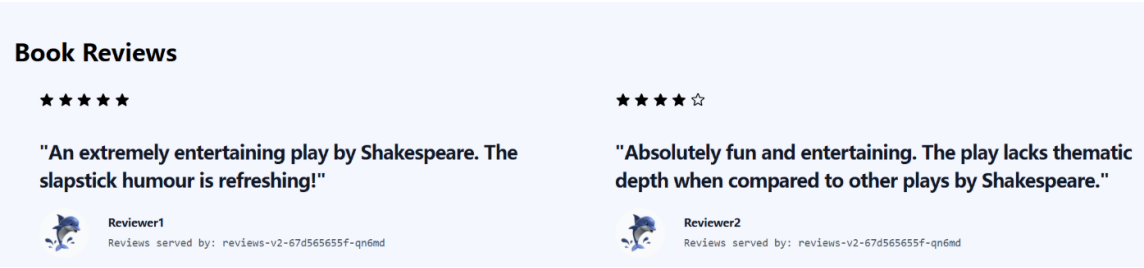


Figure 8: User “Jason” logging in to trigger user-specific routing.

This experiment verifies that Istio can perform fine-grained traffic control based on request metadata.

## 3. Fault Injection

### 3.1 Delay Fault Injection

To evaluate service resilience, a 7-second artificial delay was injected into requests from user `jason` between `reviews:v2` and the ratings service.

As shown in Figure 9, the product page failed to load properly for Jason.

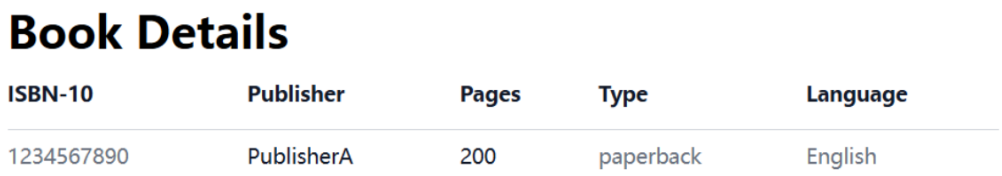


Figure 9: Injected delay causing timeout for user “Jason”.

CLI-based measurements further confirmed the increased response latency.

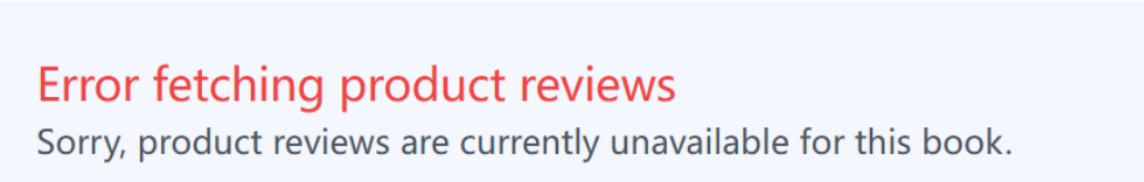


Figure 10: Measured response latency under injected delay fault.

This behavior exposes a timeout mismatch: `productpage` enforces a 3-second timeout with one retry (approximately 6 seconds), which is shorter than the injected delay, resulting in request failure.

### 3.2 HTTP Abort Fault Injection

In addition to delay faults, an HTTP abort fault was injected to simulate service unavailability:

```
kubectl apply -f virtual-service-ratings-test-abort.yaml
```

As expected, the product page displayed a ratings service error.

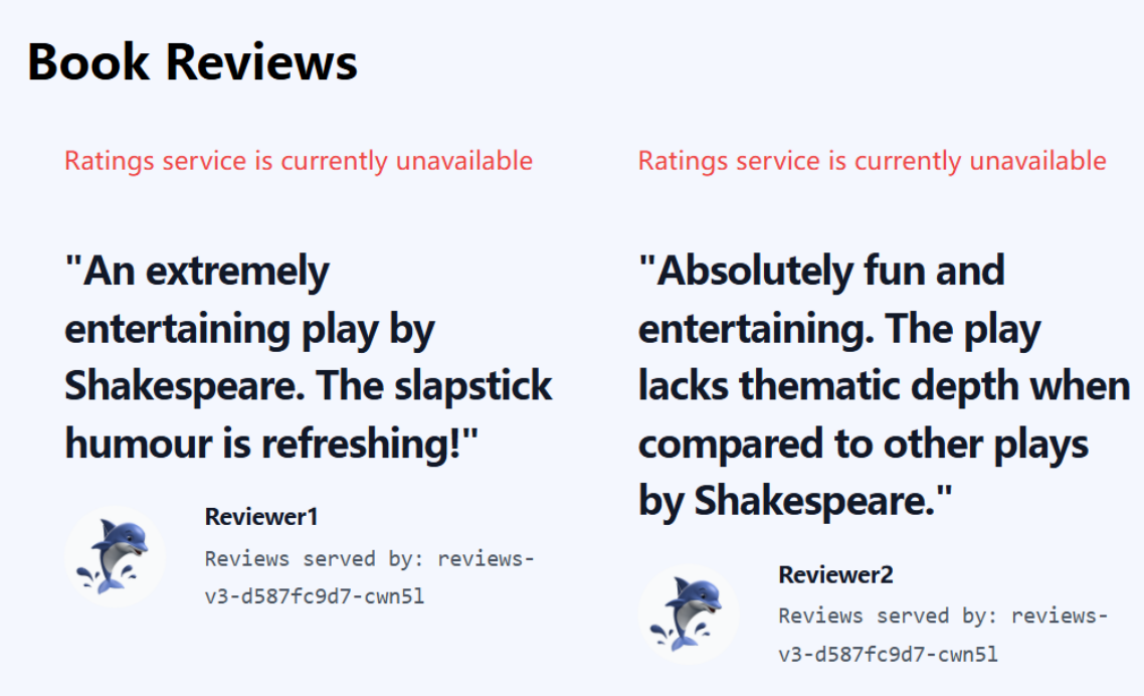


Figure 11: HTTP abort fault injection causing ratings service failure.

## 4. Circuit Breaking

The circuit breaking experiment evaluates Istio's ability to protect backend services by limiting concurrent connections and rejecting excessive requests.

### 4.1 Experimental Setup

The `httpbin` service was deployed as the backend, and `fortio` was used as the load generator.

Listing 1: Deployment Commands

```
kubectl apply -f samples/httpbin/httpbin.yaml
kubectl apply -f samples/httpbin/sample-client/fortio-deploy.yaml
```

Both services were verified to be running with Istio sidecar proxies injected.

```
C:\Users\yin\Downloads\istio-1.28.1-win-amd64\istio-1.28.1>kubectl get pods
NAME                                    READY   STATUS    RESTARTS      AGE
bookinfo-gateway-istio-5876fff84-dmnls  1/1     Running   3 (12m ago)   47h
details-v1-77d6bd5675-vkw9z             2/2     Running   2 (15m ago)   2d
fortio-deploy-7647467df4-8k2kj          2/2     Running   1 (15m ago)   44h
httpbin-654df84766-krqpr                2/2     Running   1 (15m ago)   45h
```

Figure 12: httpbin and fortio pods running with Istio sidecars.

A strict `DestinationRule` was then applied to enforce circuit breaking.

### 4.2 Load Testing Results

A baseline request returned `HTTP 200 OK`, confirming normal operation.

As load increased to two concurrent connections, a small portion of requests failed with `HTTP 503`, indicating partial circuit breaker activation.

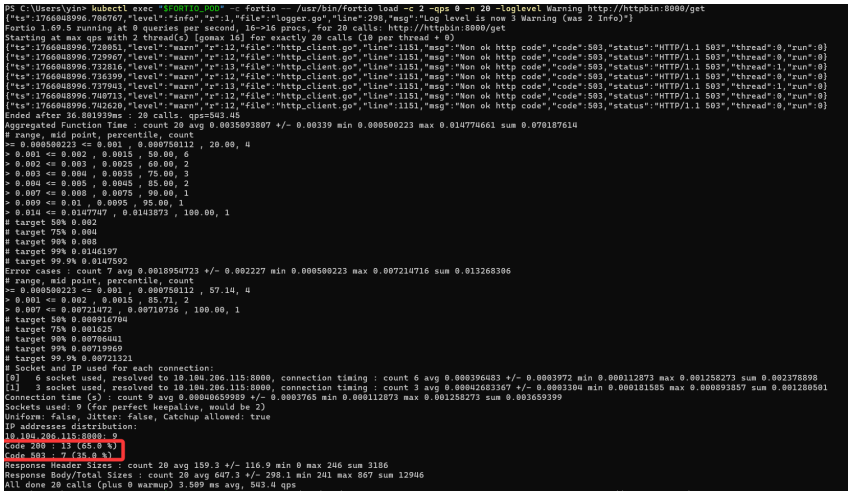


Figure 13: Initial circuit breaker activation under moderate load.

When the load increased further, the majority of requests were rejected to protect the backend service.

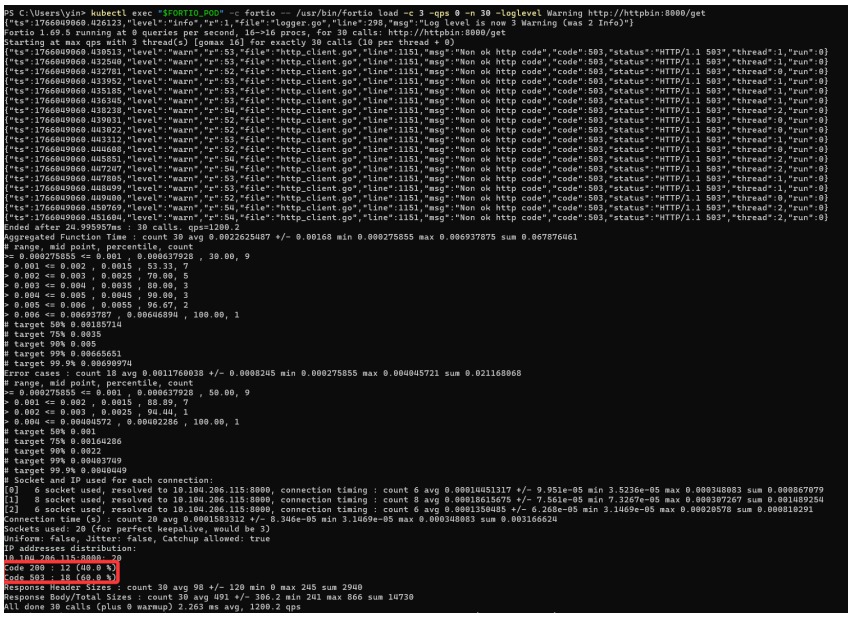


Figure 14: Circuit breaker fully triggered under high concurrency.

### 4.3 Analysis and Verification

Envoy proxy metrics were inspected to confirm that request failures were caused by circuit breaking rather than backend crashes.

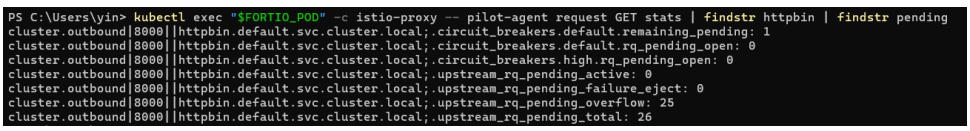


Figure 15: Envoy metrics showing pending request overflows due to circuit breaking.

The metric `upstream_rq_pending_overflow` increased significantly, verifying that Istio successfully enforced connection limits and protected the backend service under overload conditions.

# IV. Discussion and Conclusion

This laboratory experiment systematically demonstrated Istio's traffic management capabilities in a microservices-based Kubernetes environment. Through traffic shifting and request routing experiments, Istio proved capable of dynamically controlling traffic flow between service versions without requiring application redeployment or code changes.

Fault injection experiments highlighted how controlled failures can be introduced to evaluate system resilience and expose limitations in timeout and retry configurations. Circuit breaking experiments further demonstrated Istio's ability to protect backend services under high load by proactively rejecting excessive requests and preventing cascading failures.

Overall, the experiment illustrates how Istio decouples traffic management logic from application implementation, enabling flexible, fine-grained, and resilient service-to-service communication. These capabilities are essential for managing complex microservices architectures, supporting safe deployments, improving fault tolerance, and maintaining system stability in production environments.