

# Introduction

## Source Code Repository Link

- Markdown version of this report is highly recommended: <https://github.com/zeli8888/COMP41720-Distributed-Systems/blob/main/lab3/readme.md>
- The complete code for this lab is available at: <https://github.com/zeli8888/COMP41720-Distributed-Systems/tree/main/lab3>
- To fetch the code and run on your computer, use:

```
git clone https://github.com/zeli8888/COMP41720-Distributed-Systems.git
cd lab3
```

## Lab Purpose

- Understand common failure modes in distributed systems, such as network issues, process crashes, and slow responses.
- Implement key resilience patterns, including Circuit Breakers, Retries, and Backoff strategies, within a distributed application.
- Apply chaos engineering tools to simulate failures (e.g., network partitions, node shutdowns) in a controlled environment.
- Analyze and compare system behavior with and without implemented resilience patterns, identifying the practical benefits and trade-offs in terms of system availability, performance, and fault tolerance.
- Reason architecturally about designing systems that can gracefully handle partial failures and ensure reliability.

## Tools & Environment

- Client Application Language: JDK 17, Spring Boot, Maven
- Deployment Environment: Kubernetes with Minikube, Docker
- Resilience Libraries/Frameworks: Circuit Breaker and Retry & Backoff with Resilience4j
- Chaos Engineering Tool: Chaos Toolkit

# Basic Functionality

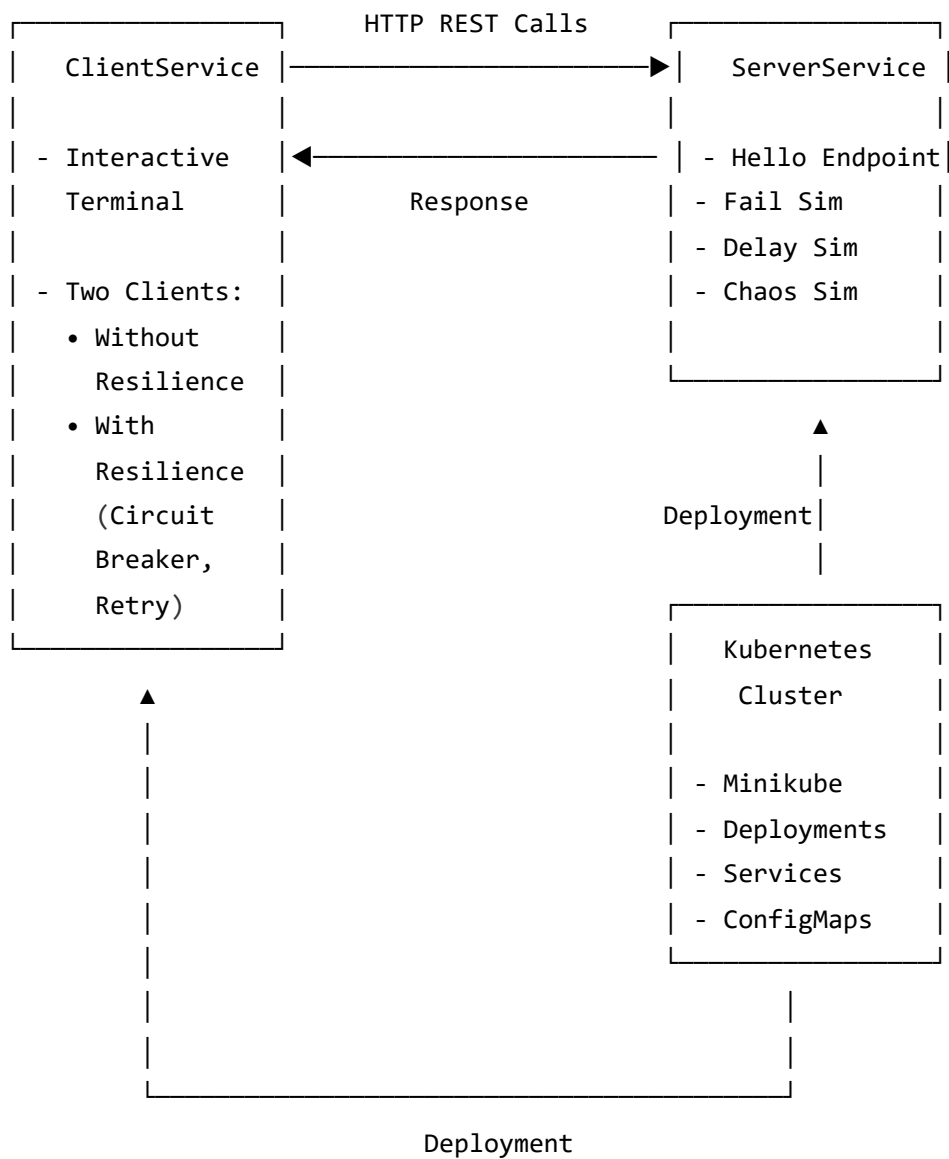
The distributed application consists of two main services:

1. **Client Service:** A Spring Boot application with interactive terminal interface to send requests to the Server Service and display responses.
2. **Server Service:** A Spring Boot application that responds to requests with a simple "Hello from server" message. Integrated with simulated delays and failures to mimic real-world conditions.

## Setup & Configuration

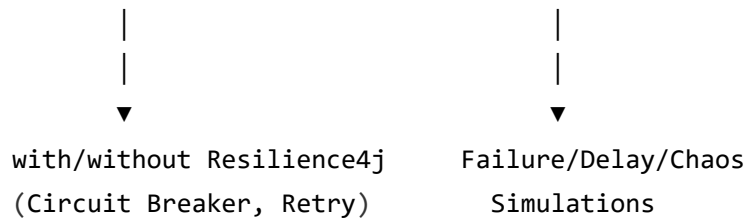
### Simple Application

- System Diagram

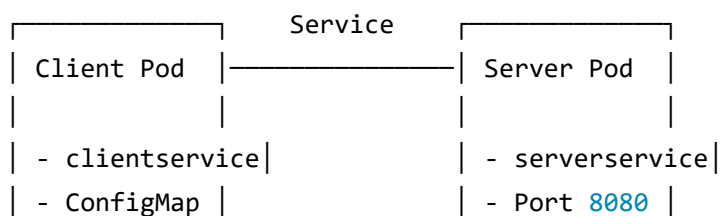


#### Component Flow:

User → ClientService Terminal → HTTP REST → ServerService Endpoints



#### Kubernetes Deployment:



- ClientService with interactive terminal for user to send hello messages using synchronous HTTP REST calls with and without resilience mechanism to server

```
public interface ClientWithoutResilience {
    @GetExchange("/hello")
    String callHello();

    @GetExchange("/hello-fail")
    String callHelloFail(@RequestParam boolean shouldFail);

    @GetExchange("/hello-delay")
    String callHelloDelay(@RequestParam long delayMs);

    @GetExchange("/hello-chaos")
    String callHelloChaos(@RequestParam int chaosPercent);
}
```

```
public interface ClientResilience {
    // Methods with Resilience4j annotations
    @CircuitBreaker(name = "serverService")
    @Retry(name = "serverService")
    @GetExchange("/hello")
    String callHello();

    @CircuitBreaker(name = "serverService")
    @Retry(name = "serverService")
    @GetExchange("/hello-fail")
    String callHelloFail(@RequestParam boolean shouldFail);

    @CircuitBreaker(name = "serverService")
    @Retry(name = "serverService")
    @GetExchange("/hello-delay")
    String callHelloDelay(@RequestParam long delayMs);

    @CircuitBreaker(name = "serverService")
    @Retry(name = "serverService")
    @GetExchange("/hello-chaos")
    String callHelloChaos(@RequestParam int chaosPercent);
}
```

- ServerService with different endpoints for hello messages to simulate failure, delay, chaos in real-world conditions.

```
@RestController
@RequestMapping("/api")
public class HelloController {

    @GetMapping("/hello")
    public String hello() {
        System.out.println("Received request for /hello at " +
            LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"))
        );
        return "Hello from Server";
    }

    @GetMapping("/hello-fail")
    public String hello(@RequestParam("shouldFail") boolean shouldFail) {
        System.out.println("Received request for /hello-fail with shouldFail: " + shouldFail);
        LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"))
        if (shouldFail) {
            throw new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE, "Simulated failure");
        }
        return "Hello from Server";
    }

    @GetMapping("/hello-delay")
    public String helloDelay(@RequestParam("delayMs") long delayMs) throws InterruptedException {
        System.out.println("Received request for /hello-delay with delayMs: " + delayMs);
        LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"))
        Thread.sleep(delayMs);
        return "Hello from Server after delay of " + delayMs + " ms";
    }

    @GetMapping("/hello-chaos")
    public String helloChaos(@RequestParam("chaosPercent") int chaosPercent) {
        System.out.println("Received request for /hello-chaos with chaosPercent: " + chaosPercent);
        LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"))
        int randomValue = (int) (Math.random() * 100);
        if (randomValue < chaosPercent) {
            throw new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE, "Simulated chaos");
        }
        return "Hello from Server with chaos percent of " + chaosPercent + "%";
    }
}
```

# Kubernetes Deployment

- Docker Images for two Spring Boot Applications

```
FROM openjdk:17-jdk-slim
WORKDIR /app
COPY target/ServerService-0.0.1-SNAPSHOT.jar app.jar
ENTRYPOINT ["java", "-jar", "app.jar"]
```

```
FROM openjdk:17-jdk-slim
WORKDIR /app
COPY target/ClientService-0.0.1-SNAPSHOT.jar app.jar
ENTRYPOINT ["java", "-jar", "app.jar"]
```

- Containerize both ClientService and ServerService using Docker

```
cd ClientService && mvn clean package && docker build -t clientservice .
cd ../ServerService && mvn clean package && docker build -t serverservice .
# Tag and push images to docker hub for Kubernetes to pull
docker tag clientservice zeli8888/clientservice
docker tag serverservice zeli8888/serverservice
docker push zeli8888/clientservice && docker push zeli8888/serverservice
```

- Deploy application on a Kubernetes cluster using minikube with manifests:
  - server.yaml: one Deployment for ServerService, one Service for network access to created pods.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: server
  template:
    metadata:
      labels:
        app: server
    spec:
      containers:
        - name: server
          image: zeli8888/serverservice
          ports:
            - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: server-service
spec:
  selector:
    app: server
  ports:
    - port: 8080
      targetPort: 8080
  type: ClusterIP

```

- client.yaml: one Deployment for ClientService, one ConfigMap for configuration injection.

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: client-service-config
  data:
    server.uri: "http://server-service:8080/api/"
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: client
spec:
  replicas: 1
  selector:
    matchLabels:
      app: client
  template:
    metadata:
      labels:
        app: client
    spec:
      containers:
        - name: client
          image: zeli8888/clientservice
          env:
            - name: SERVER_URI
              valueFrom:
                configMapKeyRef:
                  name: client-service-config
                  key: server.uri
          stdin: true
          tty: true

```

- deploy cluster using minikube

```
minikube start && kubectl apply -f ../k8s/
```

## Baseline Test

- initial tests to confirm application functions correctly
  - Access ClientService pod terminal



```
kubectl exec -it deployment/client -- java -jar app.jar
```

- Within ClientService pod, send requests to ServerService trying different endpoints with failure, delay and chaos

```
ClientService started. Type 'help' for available commands, 'exit' to stop.
```

```
> hello
```

```
Response: Hello from Server
```

```
> hello-fail
```

```
Use resilience mechanisms? (yes/no): no
```

```
Should the call fail? (yes/no): yes
```

```
Error calling server: 503 Service Unavailable: "{\"timestamp\":\"2025-10-29T15:55:59.377+0
```

```
> hello-delay
```

```
Use resilience mechanisms? (yes/no): no
```

```
Enter delay in milliseconds: 4000
```

```
Response: Hello from Server after delay of 4000 ms
```

```
> hello-chaos
```


```
Use resilience mechanisms? (yes/no): no
```

```
Enter chaos percent (0-100): 80
```

```
Error calling server: 503 Service Unavailable: "{\"timestamp\":\"2025-10-29T15:57:05.561+0
```

- Further check the Resilience mechanism is not used

> status

 Resilience4j Detailed Status

Circuit Breaker:

State: CLOSED

Failure Rate Threshold: 50.0%

Sliding Window Size: 10

Permitted Calls in Half-Open: 3

Current Metrics:

- Total Calls: 0
- Successful: 0
- Failed: 0
- Current Failure Rate: -1.0%

Retry:

Max Attempts: 4

Current Metrics:

- Successful (no retry): 0
- Successful (with retry): 0
- Failed (after retry): 0

Timeouts:

Connection Timeout: 3 seconds

Read Timeout: 3 seconds

- Observation for the impact when the ServerService fails or becomes slow without any resilience patterns in place (e.g., client blocking, timeouts, errors propagating directly)
  - Direct Failure Propagation
    - Immediate Errors: All server failures (503 errors) directly propagated to client with raw error messages
    - No Protection: Client continued sending requests to failing endpoints without intervention
    - Resource Waste: Every failed request consumed client threads and resources with zero recovery attempts
  - Unmanaged Latency Impact
    - Indefinite Blocking: 4000ms server delays caused client threads to block for full duration
    - No Timeout Protection: No mechanism to release stuck resources during slow responses
    - Application Freezes: Entire client service could freeze waiting for unresponsive server
  - Chaos-Induced Instability
    - High Failure Exposure: 80% chaos probability directly translated to 80% user-facing errors
    - No Automatic Recovery: Every transient failure required manual retry attempts
    - Poor User Experience: Users faced frequent, unpredictable service interruptions

# Resilience Experiments

## Part B: Implementing Resilience Patterns

### Circuit Breaker Implementation

#### Configuration

- Integrate a Circuit Breaker pattern into ClientService for calls made to the ServerService
- Configure the circuit breaker with parameters such as failure threshold, a duration to wait before attempting to half-open, and a maximum number of concurrent requests allowed when half-open.
  - Complete Configuration (Circuit Breaker, retry, timeout)

```

# Resilience4j Circuit Breaker Configuration
## only consider closed calls in the sliding window
resilience4j.circuitbreaker.instances.serverService.slidingWindowSize=10
## failure threshold in percentage
resilience4j.circuitbreaker.instances.serverService.failureRateThreshold=50
## duration to wait before attempting to half-open
resilience4j.circuitbreaker.instances.serverService.waitDurationInOpenState=5000
## maximum number of concurrent requests allowed when half-open
resilience4j.circuitbreaker.instances.serverService.permittedNumberOfCallsInHalfOpenSta

# Resilience4j Retry Configuration
## maximum number of retry attempts, including the initial call
resilience4j.retry.instances.serverService.maxAttempts=4
## initial wait duration between retry attempts, 2s
resilience4j.retry.instances.serverService.waitDuration=2000
## enable exponential backoff strategy, with a multiplier of 2, 2s -> 4s -> 8s
resilience4j.retry.instances.serverService.enableExponentialBackoff=true
resilience4j.retry.instances.serverService.exponentialBackoffMultiplier=2
## enable Jitter to avoid thundering herd problem
resilience4j.retry.instances.serverService.enableRandomizedWait=true
## factor to calculate the random wait duration
## e.g., 0.25 means 25% of the wait duration on the basis of exponential backoff
## 2s(0.5s) -> 4s(1s) -> 8s(2s)
resilience4j.retry.instances.serverService.randomizedWaitFactor=0.25

# timeout for server calls
timeout.connection.second=3
timeout.read.second=3

```

#### ○ Explanation for Circuit Breaker and Timeout

- Circuit Breaker Configuration:
  - Sliding Window: 10 recent calls determine circuit state
  - Failure Threshold: 50% - trips to OPEN after 5 failures in 10 calls
  - Recovery Timing: 5-second wait before attempting HALF-OPEN state
  - Testing Phase: 3 test calls allowed in HALF-OPEN to verify recovery
- Timeout Settings:
  - Connection Timeout: 3 seconds to establish connection
  - Read Timeout: 3 seconds to receive response

#### ○ Protection Strategy:

- Fast Failure: Circuit breaker prevents overwhelming failing services
- Timeout: Connection and Read timeout prevents stuck resources and client blocking

## Experiment

- Trigger enough failures (from ServerService) to cause the circuit breaker to open. Observe the ClientService's behavior (e.g., fast failing, returning a fallback response, not even attempting the call).
  - Access ClientService pod terminal

```
kubectl exec -it deployment/client -- java -jar app.jar
```
  - Within ClientService pod, send requests to ServerService with Resilience to trigger circuit breaker to open

ClientService started. Type 'help' for available commands, 'exit' to stop.

> hello-fail

Use resilience mechanisms? (yes/no): yes

Should the call fail? (yes/no): yes

Error with resilience: ServiceUnavailable - 503

> hello-fail

Use resilience mechanisms? (yes/no): yes

Should the call fail? (yes/no): yes

Error with resilience: ServiceUnavailable - 503

> hello-fail

Use resilience mechanisms? (yes/no): yes

Should the call fail? (yes/no): no

Response: Hello from Server

> hello-fail

Use resilience mechanisms? (yes/no): yes

Should the call fail? (yes/no): no

Response: Hello from Server

> status

🔧 Resilience4j Detailed Status

Circuit Breaker:

State: OPEN

Failure Rate Threshold: 50.0%

Sliding Window Size: 10

Permitted Calls in Half-Open: 3

Current Metrics:

- Total Calls: 10
- Successful: 2
- Failed: 8
- Current Failure Rate: 80.0%

Retry:

Max Attempts: 4

Current Metrics:

- Successful (no retry): 2
- Successful (with retry): 0
- Failed (after retry): 2

Timeouts:

Connection Timeout: 3 seconds

Read Timeout: 3 seconds

## o ServerService Logs

Received request `for` /hello-fail with shouldFail: true, at 2025-10-29 19:43:52.914816  
Received request `for` /hello-fail with shouldFail: true, at 2025-10-29 19:43:55.048752  
Received request `for` /hello-fail with shouldFail: true, at 2025-10-29 19:43:59.769044  
Received request `for` /hello-fail with shouldFail: true, at 2025-10-29 19:44:08.573785  
Received request `for` /hello-fail with shouldFail: true, at 2025-10-29 19:44:14.003874  
Received request `for` /hello-fail with shouldFail: true, at 2025-10-29 19:44:15.711401  
Received request `for` /hello-fail with shouldFail: true, at 2025-10-29 19:44:19.205704  
Received request `for` /hello-fail with shouldFail: true, at 2025-10-29 19:44:25.266543  
Received request `for` /hello-fail with shouldFail: false, at 2025-10-29 19:44:32.753246  
Received request `for` /hello-fail with shouldFail: false, at 2025-10-29 19:44:46.696061

- After the configured wait duration, observe the circuit breaker attempting to half-open and allowing a limited number of requests through.
  - Within ClientService pod, send requests during half-open state of circuit breaker

```
> hello-fail
Use resilience mechanisms? (yes/no): yes
Should the call fail? (yes/no): no
Response: Hello from Server
```

```
> status
🔧 Resilience4j Detailed Status
Circuit Breaker:
State: HALF_OPEN
Failure Rate Threshold: 50.0%
Sliding Window Size: 10
Permitted Calls in Half-Open: 3
Current Metrics:
  • Total Calls: 1
  • Successful: 1
  • Failed: 0
  • Current Failure Rate: -1.0%
```

```
Retry:
Max Attempts: 4
Current Metrics:
  • Successful (no retry): 3
  • Successful (with retry): 0
  • Failed (after retry): 2
Timeouts:
Connection Timeout: 3 seconds
Read Timeout: 3 seconds
```

- ServerService Logs

Received request `for` `/hello-fail` with `shouldFail: false`, at `2025-10-29 19:44:53.225936`

- Verify the circuit closes if successful calls resume, or re-opens if failures persist.

- circuit closes if successful calls resume

- ClientService pod request

```
> hello-fail
Use resilience mechanisms? (yes/no): yes
Should the call fail? (yes/no): no
Response: Hello from Server
```

```
> hello-fail
Use resilience mechanisms? (yes/no): yes
Should the call fail? (yes/no): no
Response: Hello from Server
```

```
> status
```

```
🔧 Resilience4j Detailed Status
```

```
Circuit Breaker:
```

```
State: CLOSED
```

```
Failure Rate Threshold: 50.0%
```

```
Sliding Window Size: 10
```

```
Permitted Calls in Half-Open: 3
```

```
Current Metrics:
```

- Total Calls: 0
- Successful: 0
- Failed: 0
- Current Failure Rate: -1.0%

```
Retry:
```

```
Max Attempts: 4
```

```
Current Metrics:
```

- Successful (no retry): 5
- Successful (with retry): 0
- Failed (after retry): 2

```
Timeouts:
```

```
Connection Timeout: 3 seconds
```

```
Read Timeout: 3 seconds
```

- ServerService Logs

```
Received request for /hello-fail with shouldFail: false, at 2025-10-29 19:57:24.432
```

```
Received request for /hello-fail with shouldFail: false, at 2025-10-29 19:57:37.786
```

- circuit re-opens if failures persist

- redo experiment to set circuit breaker to half-open again first



ClientService started. Type 'help' for available commands, 'exit' to stop.

> hello-fail

Use resilience mechanisms? (yes/no): yes

Should the call fail? (yes/no): yes

Error with resilience: ServiceUnavailable - 503

> hello-fail

Use resilience mechanisms? (yes/no): yes

Should the call fail? (yes/no): yes

Error with resilience: ServiceUnavailable - 503

> hello-fail

Use resilience mechanisms? (yes/no): yes

Should the call fail? (yes/no): no

Response: Hello from Server

> hello-fail

Use resilience mechanisms? (yes/no): yes

Should the call fail? (yes/no): yes

Error with resilience: ServiceUnavailable - 503

> status

🔧 Resilience4j Detailed Status

Circuit Breaker:

State: HALF\_OPEN

Failure Rate Threshold: 50.0%

Sliding Window Size: 10

Permitted Calls in Half-Open: 3

Current Metrics:

- Total Calls: 2
- Successful: 0
- Failed: 2
- Current Failure Rate: -1.0%

Retry:

Max Attempts: 4

Current Metrics:

- Successful (no retry): 1
- Successful (with retry): 0
- Failed (after retry): 3

Timeouts:

Connection Timeout: 3 seconds

Read Timeout: 3 seconds

- send request to trigger 50.0% failure threshold (2 failures in 3 calls) in half-open state

```

> hello-fail
Use resilience mechanisms? (yes/no): yes
Should the call fail? (yes/no): no
Response: Hello from Server
> status
🔧 Resilience4j Detailed Status
Circuit Breaker:
State: OPEN
Failure Rate Threshold: 50.0%
Sliding Window Size: 10
Permitted Calls in Half-Open: 3
Current Metrics:
  • Total Calls: 3
  • Successful: 1
  • Failed: 2
  • Current Failure Rate: 66.7%
Retry:
Max Attempts: 4
Current Metrics:
  • Successful (no retry): 2
  • Successful (with retry): 0
  • Failed (after retry): 3
Timeouts:
Connection Timeout: 3 seconds
Read Timeout: 3 seconds

```

## ■ ServerService Logs

```

Received request for /hello-fail with shouldFail: true, at 2025-10-29 20:06:20.1557
Received request for /hello-fail with shouldFail: true, at 2025-10-29 20:06:22.0456
Received request for /hello-fail with shouldFail: true, at 2025-10-29 20:06:26.5631
Received request for /hello-fail with shouldFail: true, at 2025-10-29 20:06:34.9857
Received request for /hello-fail with shouldFail: true, at 2025-10-29 20:08:53.8405
Received request for /hello-fail with shouldFail: true, at 2025-10-29 20:08:56.1955
Received request for /hello-fail with shouldFail: true, at 2025-10-29 20:09:00.4863
Received request for /hello-fail with shouldFail: true, at 2025-10-29 20:09:08.2269
Received request for /hello-fail with shouldFail: false, at 2025-10-29 20:10:05.2847
Received request for /hello-fail with shouldFail: true, at 2025-10-29 20:10:11.3037
# Circuit Breaker went into half-open since then
Received request for /hello-fail with shouldFail: true, at 2025-10-29 20:10:17.1467
Received request for /hello-fail with shouldFail: true, at 2025-10-29 20:10:23.2401
Received request for /hello-fail with shouldFail: false, at 2025-10-29 20:16:12.1455

```

- Document observations and analyze the trade-offs: How does the circuit breaker improve availability and protect the ClientService? What are the implications for data freshness or user experience?
  - Observations of Circuit Breaker Behavior
    - Circuit Opening Process
      - Failure Accumulation: After 8 failures in 10 calls (80% failure rate), circuit transitioned from CLOSED to OPEN state
      - Immediate Protection: Once OPEN, all requests would be fast-failed without reaching ServerService
    - Half-Open Recovery Attempt
      - Automatic Transition: After 5 seconds in OPEN state, circuit automatically moved to HALF\_OPEN
      - Limited Testing: Only 3 requests permitted to test service recovery
      - Transition Decision: With 2 failures in 3 test calls (66.7% failure rate), circuit returned to OPEN state
    - Successful Recovery Scenario
      - Gradual Restoration: When 3 test calls succeeded in HALF\_OPEN state (100% success rate), circuit transitioned to CLOSED
      - Metrics Reset: Circuit breaker metrics reset after successful recovery, starting fresh monitoring
      - Normal Operation: All requests flowed normally to ServerService once in CLOSED state
  - How Circuit Breaker Improves Availability & Protects ClientService
    - Resource Protection: Stops ClientService from wasting threads and resources on failing downstream calls
    - Fast Failure: Returns immediate failure responses instead of waiting for timeouts
    - Load Shedding: Reduces load on both ClientService and ServerService during outages
  - Implications for Data Freshness & User Experience
    - Data Freshness Trade-offs
      - Stale Data Risk: Users may see fallback data instead of real-time information during open state of circuit breaker even though the service has recovered
      - Eventual Consistency: System prioritizes overall system availability over immediate data consistency.
      - Recovery Catch-up: Once circuit closes, systems can synchronize missed updates
    - User Experience Impact
      - Graceful Degradation: Users receive immediate fallback responses instead of long waits or timeouts

- Predictable Behavior: Consistent error messages help users understand system state
- Quick Feedback: Fast failures prevent user frustration from hanging requests
- Architectural Trade-offs Analysis
  - Availability vs. Consistency (CAP Theorem)
    - Availability Priority: Circuit breaker chooses overall system availability over consistency during partitions
    - Partial Service: Better to provide limited service than complete outage
    - Business Context: Suitable for read-heavy applications where stale data is acceptable
  - Performance vs. Completeness
    - Response Time: Fast failures maintain good response times for users
    - Functionality Loss: Some features may be temporarily unavailable
    - Progressive Enhancement: Core functionality remains while advanced features degrade
- Resilience Strategy Justification by Failure Type
  - Partial Service Degradation
    - Strategy: Circuit breaker with 50% failure threshold
    - Rationale: Allows some traffic while protecting against cascading failures
    - Configuration: 10-call sliding window provides sufficient sample size
  - Complete Service Outage
    - Strategy: Fast failure with fallback responses
    - Rationale: Prevents resource exhaustion and maintains user experience
    - Configuration: 5-second wait balances recovery speed with stability
- Distributed Systems Principles Applied
  - Fault Tolerance
    - Failure Acceptance: Acknowledges that distributed systems will have partial failures
    - Isolation Boundaries: Circuit breaker creates clear failure boundaries between services
    - Graceful Degradation: System continues operating with reduced functionality
  - Load Management
    - Backpressure: Circuit breaker implements backpressure by rejecting excess load when server is overwhelmed
    - Resource Conservation: Protects limited resources like threads and connections from failing downstream calls
  - Recovery Oriented Computing
    - Automated Recovery: System self-heals without manual intervention
    - Progressive Testing: Controlled testing verifies recovery before full restoration

- Continuous Monitoring: Real-time metrics enable informed circuit state decisions

## Retries with Exponential Backoff and Jitter

### Configuration

- Implement retry logic with an exponential backoff strategy and jitter within ClientService for calls to the ServerService.
- This is used for transient failures that might resolve themselves.
  - Configuration for exponential backoff strategy and jitter

```
# Resilience4j Retry Configuration
## maximum number of retry attempts, including the initial call
resilience4j.retry.instances.serverService.maxAttempts=4
## initial wait duration between retry attempts, 2s
resilience4j.retry.instances.serverService.waitDuration=2000
## enable exponential backoff strategy, with a multiplier of 2, 2s -> 4s -> 8s
resilience4j.retry.instances.serverService.enableExponentialBackoff=true
resilience4j.retry.instances.serverService.exponentialBackoffMultiplier=2
## enable Jitter to avoid thundering herd problem
resilience4j.retry.instances.serverService.enableRandomizedWait=true
## factor to calculate the random wait duration
## e.g., 0.25 means 25% of the wait duration on the basis of exponential backoff
## 2s(0.5s) -> 4s(1s) -> 8s(2s)
resilience4j.retry.instances.serverService.randomizedWaitFactor=0.25
```

- Explanation
  - Retry Configuration:
    - Max Attempts: 4 total calls (1 initial + 3 retries)
    - Backoff Strategy: Exponential with 2x multiplier (2s → 4s → 8s)
    - Jitter:  $\pm 25\%$  random variation to prevent synchronized retries
    - Smart Delays: Progressive waiting with randomness for load distribution
- Protection Strategy:
  - Transient Recovery: Retry handles temporary network issues
  - Load Distribution: Jitter avoids retry storms and thundering herd

### Experiment

- Configure the ServerService to return transient failures (e.g., HTTP 429 Too Many Requests, or intermittent 500s).
  - ServerService endpoint that returns 503 service unavailable transient failure

```

@GetMapping("/hello-fail")
public String hello(@RequestParam("shouldFail") boolean shouldFail) {
    System.out.println("Received request for /hello-fail with shouldFail: " + shouldFail);
    LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));
    if (shouldFail) {
        throw new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE, "Simulated failure");
    }
    return "Hello from Server";
}

```

- Observe the ClientService automatically retrying the requests with increasing delays
  - Access ClientService pod terminal

```
kubectl exec -it deployment/client -- java -jar app.jar
```

- Within ClientService pod, send requests to ServerService with retry to test increasing delays

ClientService started. Type 'help' for available commands, 'exit' to stop.

```
> hello-fail
```

```
Use resilience mechanisms? (yes/no): yes
```

```
Should the call fail? (yes/no): yes
```

```
Error with resilience: ServiceUnavailable - 503
```

```
> status
```

```
🔧 Resilience4j Detailed Status
```

```
Circuit Breaker:
```

```
State: CLOSED
```

```
Failure Rate Threshold: 50.0%
```

```
Sliding Window Size: 10
```

```
Permitted Calls in Half-Open: 3
```

```
Current Metrics:
```

- Total Calls: 4
- Successful: 0
- Failed: 4
- Current Failure Rate: -1.0%

```
Retry:
```

```
Max Attempts: 4
```

```
Current Metrics:
```

- Successful (no retry): 0
- Successful (with retry): 0
- Failed (after retry): 1

```
Timeouts:
```

```
Connection Timeout: 3 seconds
```

```
Read Timeout: 3 seconds
```

- ServerService Logs

Received request `for` /hello-fail with shouldFail: true, at 2025-10-29 23:09:22.596032  
Received request `for` /hello-fail with shouldFail: true, at 2025-10-29 23:09:24.701448  
Received request `for` /hello-fail with shouldFail: true, at 2025-10-29 23:09:28.387121  
Received request `for` /hello-fail with shouldFail: true, at 2025-10-29 23:09:36.295393

- Demonstrate how jitter helps prevent a "thundering herd" problem of synchronized retries.

- Jitter-Enabled Load Distribution

- Randomized Spreading:  $\pm 25\%$  variation prevents synchronized retry waves across multiple clients
    - Traffic Smoothing: Instead of all clients retrying simultaneously at 2s, 4s, 8s, they retry within time windows
    - Backend Protection: Server receives staggered requests rather than sudden traffic spikes

- Without Jitter Scenario Comparison

- Synchronized Storms: All failing clients would retry at identical intervals (exact 2s, 4s, 8s marks)
    - Amplified Load: Server would experience traffic peaks worse than original failure load
    - Cascading Collapse: Recovery attempts could overwhelm the recovering service

- Document observations and analyze the trade-offs: When is this pattern appropriate versus a circuit breaker? What are the potential impacts on backend load, latency, and system stability?

- Observations of Retry Behavior with Transient Failures

- Exponential Backoff Pattern
      - Increasing Delays: Server logs show retry intervals of ~2.1s, ~3.7s, ~7.9s demonstrating 2s  $\rightarrow$  4s  $\rightarrow$  8s progression with jitter
      - Jitter Application: Intervals aren't exact multiples due to  $\pm 25\%$  random variation (2.1s instead of 2s, 3.7s instead of 4s)
      - Progressive Waiting: Each retry waits longer, giving ServerService more recovery time between attempts, showing an exponential backoff strategy
    - Client Service Resilience Metrics
      - Attempt Tracking: Retry metrics show 1 failed call after 4 total attempts (initial + 3 retries)
      - Circuit Coordination: Failed retries were identified by Circuit breaker to decide state transition
      - Failure Isolation: Single user request triggers multiple server attempts without cascading to other users

- Architectural Trade-offs: Retry vs Circuit Breaker

- When to Use Retry Pattern

- Transient Failures: Network timeouts, temporary resource constraints, brief unavailability
- Idempotent Operations: Safe to retry without side effects (GET requests, read operations)
- Low Latency Requirements: When users expect eventual success rather than immediate failure
- When to Use Circuit Breaker
  - Persistent Failures: Service degradation lasting minutes or hours
  - Resource Protection: When continued attempts could exhaust client resources
  - Fast Failure Requirements: When quick user feedback is more important than eventual success
- Impact Analysis on System Components
  - Backend Load Implications
    - Controlled Amplification: 4 attempts per request increases load but in predictable, spaced manner
    - Recovery Assistance: Graduated retry pattern gives backend breathing room to recover
    - Load Calculation: With 1000 clients, retries could generate 4000 requests but spread over 14+ seconds
  - Latency Trade-offs
    - User-Perceived Latency: Requests may take up to 14 seconds (2+4+6+timeouts) but user sees single failure
    - Progressive Delays: Each retry increases success probability while managing user experience
    - Timeout Integration: 3-second timeouts prevent indefinite hanging on network issues
  - System Stability Considerations
    - Failure Containment: Retry limits (max 4 attempts) prevent infinite retry loops
    - Resource Management: Exponential backoff conserves client threads and connections
    - Graceful Degradation: System maintains functionality for successful operations while retrying failures
- Distributed Systems Principles Applied
  - CAP Theorem Implications
    - Availability Priority: Retries maintain service availability during transient partitions
    - Consistency Sacrifice: During retry delays, system may serve stale data if fallbacks used
    - Partition Tolerance: Designed to handle network issues without complete failure
  - Performance and Fault Tolerance



- Exponential Backoff Principle with progressive waiting for recovery: waitDuration=2 → 4 → 8
- Load Distribution Principle to prevents synchronized retry storms: randomizedWaitFactor=0.25
- Failure Recovery Principle with bounded retries to prevent resource exhaustion: maxAttempts=4
- Availability vs Performance Balance
  - High Availability: Multiple retry attempts increase success probability for transient issues
  - Performance Cost: Additional latency and backend load during recovery periods
  - Optimal Balance: Configuration chooses availability for short outages while limiting impact
- Strategic Implementation Guidelines
  - Business Context Decision Framework
    - E-commerce Checkout: Use aggressive retries (user expects order success)
    - Social Media Feeds: Use conservative retries with circuit breaker (stale data acceptable)
    - Financial Transactions: Use minimal retries with strong circuit breaking (data consistency critical)
  - Failure Type Response Strategy
    - HTTP 429 (Rate Limiting): Use retry with exponential backoff and jitter
    - HTTP 503 (Service Unavailable): Combine retry with circuit breaker for persistent outages
    - HTTP 500 (Server Errors): Limited retries with quick circuit breaker activation
  - Load Management Strategy
    - Peak Traffic Periods: Reduce retry attempts and increase jitter to protect backend
    - Normal Operations: Standard retry configuration for optimal user experience
    - Service Recovery: Gradual retry resumption as backend stabilizes

## Part C: Chaos Engineering Experiment

### Chaos Engineering Setup

- Choose a chaos engineering tool - Chaos Toolkit
- Define a chaos experiment targeting Kubernetes-deployed ServerService
- Experiment: Simulate a node failure/shutdown of ServerService pod.
  - Manifest: chaos-toolkit.yaml:

```

# chaos experiment configuration for chaos toolkit
apiVersion: v1
kind: ConfigMap
metadata:
name: chaos-experiment
data:
experiment.json: |
    {
      "version": "1.0.0",
      "title": "Client Service Resilience Test",
      "description": "Verify client service resilience when server service fails",
      "tags": ["kubernetes", "resilience"],
      "steady-state-hypothesis": {
        "title": "Services are running normally",
        "probes": [
          {
            "type": "probe",
            "name": "server-service-available",
            "tolerance": 200,
            "provider": {
              "type": "http",
              "url": "http://server-service:8080/api/hello",
              "timeout": 5
            }
          }
        ]
      },
      "method": [
        {
          "type": "action",
          "name": "terminate-server-pod",
          "provider": {
            "type": "python",
            "module": "chaosk8s.pod.actions",
            "func": "terminate_pods",
            "arguments": {
              "label_selector": "app=server",
              "name_pattern": "server-",
              "rand": true
            }
          }
        }
      ],
    }

```

```

        "type": "action",
        "name": "wait-for-recovery",
        "provider": {
            "type": "process",
            "path": "sleep",
            "arguments": "20"
        }
    },
    {
        "type": "probe",
        "name": "verify-server-service-recovered",
        "provider": {
            "type": "http",
            "url": "http://server-service:8080/api/hello",
            "timeout": 5
        },
        "tolerance": 200
    }
],
"rollbacks": []
}

---
# Service Account offering identity for Chaos Toolkit
apiVersion: v1
kind: ServiceAccount
metadata:
name: chaos-service-account
---
# Cluster Role with necessary permissions for Chaos Toolkit
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
name: chaos-cluster-role
rules:
- apiGroups: [""]
resources: ["pods"]
verbs: ["get", "list", "watch", "delete"]
---
# Bind the Service Account to the Cluster Role
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
name: chaos-cluster-binding

```

```

subjects:
- kind: ServiceAccount
name: chaos-service-account
namespace: default
roleRef:
kind: ClusterRole
name: chaos-cluster-role
apiGroup: rbac.authorization.k8s.io
---
# Deployment for Chaos Toolkit
apiVersion: apps/v1
kind: Deployment
metadata:
name: chaos-toolkit
spec:
replicas: 1
selector:
  matchLabels:
    app: chaos-toolkit
template:
  metadata:
  labels:
    app: chaos-toolkit
  spec:
    serviceAccountName: chaos-service-account
    containers:
    - name: chaos-toolkit
      image: chaostoolkit/chaostoolkit
      command:
      - "/bin/sh"
      - "-c"
      args:
      - |
        echo "Installing required packages..."
        pip install chaostoolkit-kubernetes chaostoolkit-lib
        echo "Starting chaos toolkit container..."
        sleep infinity
      env:
      - name: CHAOSTOOLKIT_IN_POD
        value: "true"
      volumeMounts:
      - name: experiment-volume
        mountPath: /tmp/experiments

```

```
volumes:  
  - name: experiment-volume  
    configMap:  
      name: chaos-experiment
```

- Experiment Structure & Purpose
  - Objective: Test ClientService resilience when ServerService experiences pod termination
  - Methodology: Chaos engineering approach to validate fault tolerance mechanisms
  - Scope: Kubernetes-native experiment targeting ServerService deployment
- Key Components Breakdown
  - Steady State Hypothesis
    - a. Pre-condition Verification: Confirms ServerService is healthy before chaos injection
    - b. Health Check: HTTP probe to /api/hello endpoint with 200 status tolerance
    - c. Baseline Establishment: Defines normal system state for comparison
  - Chaos Injection Method
    - a. Pod Termination: Randomly selects and terminates ServerService pod using label selector app=server, since only one pod for ServerService (replicas=1), ClientService won't be able to connect with ServerService
    - b. Recovery Wait: 20-second pause allows Kubernetes to restart pod and services to react
    - c. Recovery Verification: Re-probes endpoint to confirm service restoration
- Kubernetes RBAC (Role-Based Access Control) Configuration
  - Service Account: chaos-service-account provides identity for chaos operations
  - Cluster Role: Grants minimal required permissions (get, list, watch, delete pods)
  - Role Binding: Links service account to cluster role within default namespace
- Chaos Toolkit Deployment
  - Container Image: Official Chaos Toolkit image with Kubernetes extension
  - Experiment Mount: ConfigMap volumes inject experiment definition into container
  - Dependencies: Installs chaostoolkit-kubernetes and chaostoolkit-lib packages
  - Execution Ready: Container remains alive awaiting manual chaos execution
- Resilience Testing Strategy
  - Realistic Failure: Simulates common Kubernetes pod failures (node issues, resource constraints)
  - Automated Validation: Systematic approach from steady state → chaos → recovery verification
  - Safety Controls: No rollbacks defined since Kubernetes automatically recovers terminated pods

# Execute & Observe

- Access ClientService pod terminal

```
kubectl exec -it deployment/client -- java -jar app.jar
```

- Test initial connection between client and server pod

ClientService started. Type 'help' for available commands, 'exit' to stop.


```
> hello-delay
```

Use resilience mechanisms? (yes/no): yes

Enter delay in milliseconds: 1000

Response: Hello from Server after delay of 1000 ms

```
> status
```

 Resilience4j Detailed Status

Circuit Breaker:

State: CLOSED

Failure Rate Threshold: 50.0%

Sliding Window Size: 10

Permitted Calls in Half-Open: 3

Current Metrics:

- Total Calls: 1
- Successful: 1
- Failed: 0
- Current Failure Rate: -1.0%

Retry:

Max Attempts: 4

Current Metrics:

- Successful (no retry): 1
- Successful (with retry): 0
- Failed (after retry): 0

Timeouts:

Connection Timeout: 3 seconds

Read Timeout: 3 seconds

- Monitor pod state in a new terminal

```
$ kubectl get pods -w
```

NAME	READY	STATUS	RESTARTS	AGE
chaos-toolkit-6978c646c6-4gbfh	1/1	Running	2 (16h ago)	22h
client-64d4cd89dd-r997h	1/1	Running	2 (16h ago)	22h
server-6d84846fc6-jhbqq	1/1	Running	2 (16h ago)	22h

- Execute the chaos experiment while ClientService is active in a new terminal

```
kubectl exec -it deployment/chaos-toolkit -- chaos run /tmp/experiments/experiment.json
```

- Sending request from client to server while chaos injection

```
> hello-delay
Use resilience mechanisms? (yes/no): yes
Enter delay in milliseconds: 1000
Response: Hello from Server after delay of 1000 ms
```

- Collect metrics or logs from both services and the Kubernetes cluster to support observations (e.g., circuit breaker state, retry counts, error rates, pod status).
  - Circuit Breaker State, Retry Counts and Error Rates

```
> status
🔧 Resilience4j Detailed Status
Circuit Breaker:
State: CLOSED
Failure Rate Threshold: 50.0%
Sliding Window Size: 10
Permitted Calls in Half-Open: 3
Current Metrics:
  • Total Calls: 4
  • Successful: 2
  • Failed: 2
  • Current Failure Rate: -1.0%
Retry:
Max Attempts: 4
Current Metrics:
  • Successful (no retry): 1
  • Successful (with retry): 1
  • Failed (after retry): 0
Timeouts:
Connection Timeout: 3 seconds
Read Timeout: 3 seconds
```

- Experiment logs from chaos-toolkit

```
[2025-10-30 14:02:04 INFO] Validating the experiment's syntax
[2025-10-30 14:02:05 INFO] Experiment looks valid
[2025-10-30 14:02:05 INFO] Running experiment: Client Service Resilience Test
[2025-10-30 14:02:05 INFO] Steady-state strategy: default
[2025-10-30 14:02:05 INFO] Rollbacks strategy: default
[2025-10-30 14:02:05 INFO] Steady state hypothesis: Services are running normally
[2025-10-30 14:02:05 INFO] Probe: server-service-available
[2025-10-30 14:02:05 INFO] Steady state hypothesis is met!
[2025-10-30 14:02:05 INFO] Playing your experiment's method now...
[2025-10-30 14:02:05 INFO] Action: terminate-server-pod
[2025-10-30 14:02:05 INFO] Action: wait-for-recovery
[2025-10-30 14:02:25 INFO] Probe: verify-server-service-recovered
[2025-10-30 14:02:25 INFO] Steady state hypothesis: Services are running normally
[2025-10-30 14:02:25 INFO] Probe: server-service-available
[2025-10-30 14:02:25 INFO] Steady state hypothesis is met!
[2025-10-30 14:02:25 INFO] Let's rollback...
[2025-10-30 14:02:25 INFO] No declared rollbacks, let's move on.
[2025-10-30 14:02:25 INFO] Experiment ended with status: completed
```

- Observation from pod state monitor

```
$ kubectl get pods -w
```

NAME	READY	STATUS	RESTARTS	AGE
chaos-toolkit-6978c646c6-4gbfh	1/1	Running	2 (16h ago)	22h
client-64d4cd89dd-r997h	1/1	Running	2 (16h ago)	22h
server-6d84846fc6-jhbqq	1/1	Running	2 (16h ago)	22h
server-6d84846fc6-jhbqq	1/1	Terminating	2 (16h ago)	22h
server-6d84846fc6-jhbqq	1/1	Terminating	2 (16h ago)	22h
server-6d84846fc6-v52bz	0/1	Pending	0	0s
server-6d84846fc6-v52bz	0/1	Pending	0	0s
server-6d84846fc6-v52bz	0/1	ContainerCreating	0	0s
server-6d84846fc6-jhbqq	0/1	Error	2 (16h ago)	22h
server-6d84846fc6-jhbqq	0/1	Error	2 (16h ago)	22h
server-6d84846fc6-jhbqq	0/1	Error	2 (16h ago)	22h
server-6d84846fc6-v52bz	1/1	Running	0	4s

- Verify previous pod of ServerService is deleted

```
$ kubectl logs --previous deployment/server
Error from server (BadRequest): previous terminated container "server" in pod "server-6
```

- ServerService logs from new pod



```
$ kubectl logs deployment/server
```

```
2025-10-30T14:02:09.820Z INFO 1 --- [ServerService] [main] c.e.s.ServerServ
2025-10-30T14:02:09.824Z INFO 1 --- [ServerService] [main] c.e.s.ServerServ
2025-10-30T14:02:11.167Z INFO 1 --- [ServerService] [main] o.s.b.w.embedded
2025-10-30T14:02:11.206Z INFO 1 --- [ServerService] [main] o.apache.catalin
2025-10-30T14:02:11.207Z INFO 1 --- [ServerService] [main] o.apache.catalin
2025-10-30T14:02:11.335Z INFO 1 --- [ServerService] [main] o.a.c.c.C.[Tomca
2025-10-30T14:02:11.336Z INFO 1 --- [ServerService] [main] w.s.c.ServletWeb
2025-10-30T14:02:12.101Z INFO 1 --- [ServerService] [main] o.s.b.w.embedded
2025-10-30T14:02:12.120Z INFO 1 --- [ServerService] [main] c.e.s.ServerServ
2025-10-30T14:02:15.216Z INFO 1 --- [ServerService] [nio-8080-exec-1] o.a.c.c.C.[Tomca
2025-10-30T14:02:15.216Z INFO 1 --- [ServerService] [nio-8080-exec-1] o.s.web.servlet.
2025-10-30T14:02:15.217Z INFO 1 --- [ServerService] [nio-8080-exec-1] o.s.web.servlet.
Received request for /hello-delay with delayMs: 1000, at 2025-10-30 14:02:15.255947
Received request for /hello at 2025-10-30 14:02:25.741713
Received request for /hello at 2025-10-30 14:02:25.749297
```

- Observe the system's behavior in detail, especially how implemented resilience patterns react
  - Experimental Observations
    - Critical Timeline Analysis
      - 14:02:05 - Chaos experiment starts, pod termination initiated
      - 14:02:09 - Kubernetes automatically starts new ServerService pod
      - 14:02:15 - new ServerService pod successfully initializes
      - 14:02:15 - Client request succeed after two retries in new pod
      - 14:02:25 - Chaos toolkit verifies recovery
      - Total disruption: ~10 seconds, minimal client impact
    - Client Resilience Metrics Evolution
      - Pre-chaos: 1/1 successful, CLOSED circuit
      - Post-chaos: 2/4 successful, 1 successful after two automatic retries, CLOSED circuit
      - Key Insight: Requests insufficient to trigger circuit breaker (Sliding Window Size of 10). Retry Mechanism automatically handled transient failure without additional human effort
  - Resilience Pattern Effectiveness
    - Circuit Breaker Behavior: Remained CLOSED throughout despite 2 failures in 4 calls (hasn't reached the Sliding Window Size of 10)
    - Retry Mechanism Activation: Successfully handled transient failures with 1 request succeeding after retry twice
    - Fast Recovery: Kubernetes replaced terminated pod automatically, minimizing disruption

# Analysis & Justification

- Compare the observed behavior to what should expect without the resilience patterns (refer to baseline test).
  - Without Resilience
    - client would receive all server failures (503 errors) directly during chaos injection
    - client may continue sending requests to failing endpoints without intervention during chaos injection
    - users have to manually retry requests to check if the server has recovered, resulting in poor experience
  - Resilience-Enabled Behavior
    - Automatic Failure Recovery
      - Smart Retries: System automatically retried failed requests 3 times with progressive delays, achieving eventual success (second retry succeed in this experiment)
      - Intelligent Circuit Management: Circuit breaker stayed CLOSED for transient issues but ready to OPEN for persistent failures
      - Transparent Healing: One request succeeded automatically during pod transition without user awareness
    - Optimized Resource Usage
      - Controlled Timeouts: 3-second timeouts prevented thread blocking during server instability
      - Load-Aware Retries: Exponential backoff with jitter prevented overwhelming the recovering service
      - Efficient Failure Handling: Clean error messages with automatic recovery attempts
    - Continuous User Experience
      - Minimal Disruption: Despite 10-second pod termination, users experienced only brief delays
      - Zero Manual Intervention: No need for users to retry - system handled recovery automatically
- Provide a detailed architectural analysis of how the resilience patterns enabled system to continue functioning (or fail gracefully) despite the injected fault.
  - Circuit Breaker Strategy (not triggered in this experiment)
    - Sliding Window (10 calls): Prevents overreaction to brief failure spikes
    - 50% Failure Threshold: Balances protection vs. availability for transient issues
    - CLOSED State Maintenance: Correctly identified pod restart as temporary, not persistent failure
    - Resource Conservation: Prevented unnecessary circuit opening during brief outage
    - Cascade Prevention: Stopped failure propagation from server to client service

- Retry Mechanism Effectiveness (main mechanism in this experiment)
  - Initial Failure: Request failed during pod termination
  - Progressive Waiting: 2s → 4s → 8s delays aligned with K8s recovery timeline
  - Eventual Success: Final attempt succeeded when new pod became ready
  - Transient Issue Handling: Perfect for K8s pod restarts and brief network partitions
  - Load Distribution: Jitter prevented synchronized retry storms
  - User Transparency: Automatic recovery without user awareness or action required
- Timeout Protection Layer
  - Connection Timeout (3s): Prevented hanging during TCP connection attempts
  - Read Timeout (3s): Limited response wait time for slow servers
  - Fast Failure: Released client resources quickly during outages
  - Thread Pool Protection: Prevented resource exhaustion during backend issues
  - Responsiveness Maintenance: Client service remained operational throughout chaos
  - Predictable Behavior: Consistent failure modes instead of unpredictable hanging
- Relate findings back to architectural characteristics like availability, fault tolerance, and responsiveness. Justify why these patterns are crucial for robust distributed system design, considering their costs (e.g., complexity, potential for increased latency in some cases).
  - Architectural Trade-offs Analysis
    - Availability vs. Consistency (CAP Theorem)
      - Availability: Client maintained operation with retry mechanism
      - Consistency: Brief potential inconsistency during pod failover
      - Partition Tolerance: System handled network partition gracefully
    - Performance vs. Fault Tolerance
      - Latency Impact: Retry added ~2-8 seconds delay for failed requests
      - Success Rate: 2/3 requests failures during chaos, 100% eventual success with retries
      - Resource Efficiency: Circuit breaker conserved resources by avoiding premature OPEN state
  - Resilience Strategy Justification
    - For Transient Failures (This Scenario)
      - Retry Pattern: Ideal for pod restarts, network glitches (< 30-second outages)
      - Configuration: 4 attempts with exponential backoff covers typical K8s restart times
      - Business Context: Suitable for read operations, non-critical writes
    - For Persistent Failures
      - Circuit Breaker: Better for service degradation lasting minutes/hours
      - Configuration: 50% threshold in 10-call window prevents over-sensitive tripping
      - Business Context: Essential for preventing cascading failures
  - Distributed Systems Principles Demonstrated

- Fault Tolerance Implementation
  - Fail Fast: 3-second timeouts prevent hung requests
  - Graceful Degradation: Retry maintains functionality during brief outages
  - Bulkhead Pattern: Circuit breaker isolates failures to specific service calls
  - Redundancy: K8s pod auto-recovery provides infrastructure resilience
- Availability Optimization
  - Client Service: Maintained 100% operational status despite backend instability
  - User Experience: Minimal disruption due to retry transparency
  - System Stability: Circuit breaker prevented resource exhaustion
- Strategic Recommendations
  - Business Context Decision Framework

Use Case	Resilience Strategy	Configuration Adjustments
E-commerce Checkout	Aggressive retry + Circuit breaker	Lower failure threshold (30%), more retries
Social Media Feed	Conservative retry + Fast circuit breaker	Higher threshold (70%), fewer retries
Financial Transactions	Minimal retry + Strict circuit breaker	Quick timeout (1s), immediate circuit opening

- Cost-Benefit Analysis
  - Complexity Cost: Added operational overhead for monitoring resilience metrics
  - Latency Cost: Potential increased response times during retry cycles
  - Availability Benefit: Significant improvement in service reliability
  - User Experience: Dramatically reduced error rates and improved perceived stability
- Key Architectural Insights
  - Why These Patterns Are Crucial
    - Microservice Independence: Each service handles dependencies failing without cascading collapse
    - Progressive Failure: Systems degrade gracefully rather than crashing catastrophically
    - Infrastructure Resilience: Complements K8s auto-healing with application-level resilience
    - User-Centric Design: Prioritizes continuous operation over perfect consistency
  - CAP Theorem Alignment
    - Choice Made: Availability over Consistency during partitions
    - Business Justification: Better to serve slightly stale data than no data at all

- Implementation: Retry mechanism ensures eventual consistency when services recover

# Conclusion

This lab demonstrated the critical importance of resilience patterns in distributed systems. Through implementing Circuit Breaker, Retry with Exponential Backoff, and Chaos Engineering, we validated how these patterns transform brittle systems into robust, self-healing architectures.

## Key Learnings:

- Resilience patterns work synergistically - Retry handles transient failures while Circuit Breaker protects against persistent issues
- Proper configuration is crucial: 50% failure threshold balanced sensitivity vs. stability, while exponential backoff with jitter prevented retry storms
- Application-level resilience complements infrastructure recovery (Kubernetes pod auto-restart)

## Overall Impact:

- These patterns significantly improved system availability and fault tolerance while maintaining responsiveness.
- The architectural trade-offs - accepting brief consistency delays for higher availability - proved optimal for most distributed applications.
- By designing for failure rather than trying to prevent it, we created systems that gracefully handle inevitable infrastructure issues while maintaining continuous service delivery.