🚀 Scaling Observability for 100K Microservices: A Detailed Framework for Identifying and Resolving Performance Bottlenecks 🔍

Managing and maintaining a system with **100,000 microservices** is a formidable task, especially when one or more microservices exhibit high response times. Identifying and diagnosing these performance issues require a **comprehensive observability framework**. Below is a detailed framework that covers all aspects of observability, from metrics collection to automated diagnosis, complete with commands, configurations, examples, code snippets, and scenarios.

---

1. Introduction

Observability is the ability to measure the internal states of a system by examining its outputs. For microservices architectures, especially at the scale of 100K services, observability is crucial for:

- **Monitoring performance metrics**
- **Tracing requests across services**
- **Logging events and errors**
- **Automating issue detection and resolution**

This framework will guide you through setting up a robust observability pipeline to identify and resolve high response time issues in your microservices architecture.

---

2. Robust Observability Framework Overview

The observability framework consists of the following components:

1. **Metrics Collection**: Collect real-time performance data.
2. **Distributed Tracing**: Trace requests across microservices.
3. **Centralized Logging**: Aggregate and search logs.
4. **Instrumentation**: Embed observability into microservices.
5. **Automated Issue Detection**: Set up alerts and automated responses.
6. **Analysis and Resolution**: Tools and processes to analyze and fix issues.
7. **CI/CD Integration**: Automate diagnosis within the CI/CD pipeline.

Let's delve into each component in detail.

3. Setup for Metrics Collection

Prometheus is a popular open-source system monitoring and alerting toolkit, ideal for collecting metrics from microservices.

3.1. Installing Prometheus

First, install Prometheus on your monitoring server.

```
# Download Prometheus
wget https://github.com/prometheus/prometheus/releases/download/v2.44.0/prometheus-2.44.0.linux-amd64.tar.gz

# Extract the downloaded archive
tar xvf prometheus-2.44.0.linux-amd64.tar.gz

# Move to /usr/local/bin
sudo mv prometheus-2.44.0.linux-amd64/prometheus /usr/local/bin/
sudo mv prometheus-2.44.0.linux-amd64/promtool /usr/local/bin/

# Create Prometheus directories
sudo mkdir /etc/prometheus
sudo mkdir /var/lib/prometheus

# Move configuration files
sudo cp -r prometheus-2.44.0.linux-amd64/consoles /etc/prometheus
sudo cp -r prometheus-2.44.0.linux-amd64/console_libraries /etc/prometheus
```

3.2. Configuring Prometheus

Create a prometheus.yml configuration file to define scrape targets.

```
# /etc/prometheus/prometheus.yml
global:
  scrape_interval: 15s  # How frequently to scrape targets

scrape_configs:
  - job_name: 'microservices'
    kubernetes_sd_configs:
      - role: pod
    relabel_configs:
      - source_labels: [__meta_kubernetes_pod_label_app]
        action: keep
        regex: my-microservice
    metrics_path: /metrics
    scheme: http
```

Explanation:

- **global.scrape_interval**: Sets the default scraping interval.
- **scrape_configs**: Defines how to discover and scrape targets.
    - **kubernetes_sd_configs**: Discovers services in a Kubernetes cluster.
    - **relabel_configs**: Filters pods with the label app=my-microservice.
    - **metrics_path**: The HTTP endpoint where metrics are exposed.
    - **scheme**: The protocol used to access the metrics endpoint.

3.3. Starting Prometheus

Create a systemd service for Prometheus.

```
# /etc/systemd/system/prometheus.service
[Unit]
Description=Prometheus
Wants=network-online.target
After=network-online.target

[Service]
User=prometheus
Group=prometheus
Type=simple
ExecStart=/usr/local/bin/prometheus \
 --config.file=/etc/prometheus/prometheus.yml \
 --storage.tsdb.path=/var/lib/prometheus/ \
 --web.console.templates=/etc/prometheus/consoles \
 --web.console.libraries=/etc/prometheus/console_libraries

[Install]
WantedBy=multi-user.target
```

Commands to start Prometheus:

```
# Reload systemd configurations
sudo systemctl daemon-reload

# Start Prometheus service
sudo systemctl start prometheus

# Enable Prometheus to start on boot
sudo systemctl enable prometheus
```

3.4. Visualizing Metrics with Grafana

Grafana provides a powerful visualization layer for Prometheus metrics.

Installation:

```
# Add Grafana APT repository
sudo apt-get install -y software-properties-common
sudo add-apt-repository "deb https://packages.grafana.com/oss/deb stable main"

# Install Grafana
sudo apt-get update
sudo apt-get install grafana

# Start and enable Grafana service
sudo systemctl start grafana-server
sudo systemctl enable grafana-server
```

Configuration:

1. **Access Grafana**: Navigate to http://<your-server-ip>:3000 and log in (default credentials: admin/admin).
2. **Add Prometheus Data Source**:
    - Go to **Configuration > Data Sources > Add data source**.
    - Select **Prometheus**.
    - Set the URL to http://localhost:9090.
    - Click **Save & Test**.
3. **Import Dashboards**:
    - Use pre-built dashboards or create custom ones to visualize metrics like response times, request rates, error rates, etc.

Example Dashboard: Response Time

Create a graph panel with the following PromQL query:

```
histogram_quantile(0.95, sum(rate(http_request_duration_seconds_bucket[5m])) by (le, service))
```

Explanation:

- **histogram_quantile**: Calculates quantiles (e.g., 95th percentile) from histogram buckets.
- **rate**: Calculates the per-second average rate of requests over a 5-minute window.
- **by (le, service)**: Aggregates metrics by bucket (le) and service.

---

4. Distributed Tracing with Jaeger

Jaeger is an open-source distributed tracing system, useful for monitoring and troubleshooting transactions in complex microservices environments.

4.1. Installing Jaeger

You can deploy Jaeger using Docker for simplicity.

```
# Pull Jaeger all-in-one image
docker pull jaegertracing/all-in-one:latest

# Run Jaeger all-in-one
docker run -d --name jaeger \
 -e COLLECTOR_ZIPKIN_HOST_PORT=:9411 \
 -p 5775:5775/udp \
 -p 6831:6831/udp \
 -p 6832:6832/udp \
 -p 5778:5778 \
 -p 16686:16686 \
 -p 14268:14268 \
 -p 14250:14250 \
 -p 9411:9411 \
 jaegertracing/all-in-one:latest
```

Explanation:

- **All-in-one**: Combines the Jaeger UI, agent, collector, and query services into a single Docker container.
- **Ports**:
  - `16686`: Jaeger UI
  - `14268`: Collector HTTP endpoint
  - `9411`: Zipkin compatible endpoint

4.2. Instrumenting Microservices for Tracing

To enable tracing, instrument your microservices with Jaeger clients. Below is an example in **Node.js** using the `jaeger-client` library.

Installation:

```
npm install jaeger-client opentracing
```

Code Snippet:

```
// tracer.js
const initTracer = require('jaeger-client').initTracer;

// Configuration for Jaeger Tracer
const config = {
 serviceName: 'my-microservice', // Name of the service
 reporter: {
  // Log the spans to console for debugging
```

```
    logSpans: true,
    // Jaeger agent host and port
    agentHost: 'localhost',
    agentPort: 6831,
  },
  sampler: {
    // Sample all traces for demonstration (not recommended for production)
    type: 'const',
    param: 1,
  },
};

// Initialize tracer
const options = {
  // Configuration options if any
};

const tracer = initTracer(config, options);

module.exports = tracer;
```

Using the Tracer:

```
// app.js
const express = require('express');
const tracer = require('./tracer');
const opentracing = require('opentracing');

const app = express();

// Middleware to start a span for each incoming request
app.use((req, res, next) => {
  const wireCtx = tracer.extract(opentracing.FORMAT_HTTP_HEADERS, req.headers);
  const span = tracer.startSpan(req.path, { childOf: wireCtx });

  // Add span to request object for further usage
  req.span = span;

  // Finish the span when response is finished
  res.on('finish', () => {
    span.setTag(opentracing.Tags.HTTP_STATUS_CODE, res.statusCode);
    span.finish();
  });

  next();
});

app.get('/endpoint', async (req, res) => {
  const span = req.span;

  // Start a child span for a database call
  const dbSpan = tracer.startSpan('db_query', { childOf: span });
```

```
  // Simulate database query
  await fakeDbQuery();
  dbSpan.finish();

  res.send('Response from endpoint');
});

function fakeDbQuery() {
  return new Promise((resolve) => setTimeout(resolve, 100)); // Simulate 100ms delay
}

app.listen(3000, () => {
  console.log('Service listening on port 3000');
});
```

Explanation:

- **initTracer**: Initializes the Jaeger tracer with configuration.
- **Middleware**: Extracts tracing information from incoming requests and starts a new span.
- **Child Spans**: Creates child spans for internal operations like database queries.
- **Span Finishing**: Ensures spans are finished when the response is sent.

4.3. Viewing Traces in Jaeger UI

Navigate to http://<your-server-ip>:16686 to access the Jaeger UI.

1. **Select Service**: Choose your service (e.g., my-microservice).
2. **Set Time Range**: Define the period for trace search.
3. **Search**: Click **Find Traces** to view traces.
4. **Trace Details**: Click on individual traces to see the flow across microservices, durations, and spans.

---

5. Centralized Logging with ELK Stack

The **ELK Stack** (Elasticsearch, Logstash, Kibana) provides centralized logging, enabling powerful search and visualization capabilities.

5.1. Installing ELK Stack

For large-scale deployments, it's recommended to use Docker or Kubernetes. Below is a simplified installation using Docker Compose.

docker-compose.yml:

```
version: '7.6'
services:
 elasticsearch:
  image: docker.elastic.co/elasticsearch/elasticsearch:8.7.0
  environment:
   - discovery.type=single-node
   - ES_JAVA_OPTS=-Xms512m -Xmx512m
  ports:
   - "9200:9200"

 logstash:
  image: docker.elastic.co/logstash/logstash:8.7.0
  volumes:
   - ./logstash.conf:/usr/share/logstash/pipeline/logstash.conf
  ports:
   - "5044:5044"

 kibana:
  image: docker.elastic.co/kibana/kibana:8.7.0
  ports:
   - "5601:5601"
```

logstash.conf:

```
input {
 beats {
  port => 5044
 }
}

filter {
 json {
  source => "message"
 }
}

output {
 elasticsearch {
  hosts => ["elasticsearch:9200"]
  index => "microservices-logs-%{+YYYY.MM.dd}"
 }
}
```

Explanation:

- **Elasticsearch**: Stores and indexes logs.
- **Logstash**: Processes incoming logs from Beats (e.g., Filebeat) and forwards them to Elasticsearch.

- **Kibana**: Provides a UI for searching and visualizing logs.

Start ELK Stack:

```
docker-compose up -d
```

5.2. Installing and Configuring Filebeat

Filebeat is a lightweight shipper for forwarding and centralizing log data.

Installation:

```
# Download and install Filebeat
curl -L -O https://artifacts.elastic.co/downloads/beats/filebeat/filebeat-8.7.0-amd64.deb
sudo dpkg -i filebeat-8.7.0-amd64.deb
```

Configuration:

Edit the filebeat.yml file to define log paths and output.

```
# filebeat.yml

filebeat.inputs:
- type: log
  enabled: true
  paths:
    - /var/log/microservices/*.log

output.logstash:
  hosts: ["<logstash-server-ip>:5044"]
```

Explanation:

- **filebeat.inputs**: Specifies log files to monitor (e.g., /var/log/microservices/*.log).
- **output.logstash**: Sends logs to Logstash for processing.

Start Filebeat:

```
sudo systemctl start filebeat
sudo systemctl enable filebeat
```

5.3. Viewing Logs in Kibana

1. **Access Kibana**: Navigate to http://<your-server-ip>:5601.
2. **Configure Index Pattern**:
    - Go to **Management > Stack Management > Index Patterns > Create index pattern**.

- Enter microservices-logs-* as the index pattern.
- Select the timestamp field and create the index pattern.

3. **Explore Logs**:
   - Use **Discover** to search and filter logs.
   - Create visualizations and dashboards as needed.

Example Search Query:

To find logs with high response times:

```
{
 "query": {
  "range": {
   "response_time_ms": {
    "gte": 1000
   }
  }
 }
}
```

---

6. Instrument Microservices

Instrumentation involves embedding observability hooks within your microservices to emit metrics, traces, and logs. Below are examples in **Java** using **Spring Boot** and **Micrometer**.

6.1. Adding Dependencies

Add the following dependencies to your pom.xml for metrics and tracing.

```
<dependencies>
  <!-- Micrometer Prometheus registry -->
  <dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
  </dependency>

  <!-- Spring Boot Actuator -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>

  <!-- Jaeger Tracing -->
  <dependency>
    <groupId>io.opentracing.contrib</groupId>
    <artifactId>opentracing-spring-jaeger-cloud-starter</artifactId>
    <version>3.3.0</version>
```

```
  </dependency>
</dependencies>
```

6.2. Configuring Micrometer and Jaeger

application.yml:

```
# application.yml

management:
 endpoints:
  web:
   exposure:
    include: "prometheus,health,info"
 metrics:
  export:
   prometheus:
    enabled: true

spring:
 application:
  name: my-microservice

jaeger:
 service-name: my-microservice
 udp-sender:
  host: localhost
  port: 6831
 sampler:
  type: const
  param: 1
```

Explanation:

- **management.endpoints.web.exposure.include**: Exposes /actuator/prometheus for Prometheus scraping.
- **metrics.export.prometheus.enabled**: Enables Prometheus metrics export.
- **spring.application.name**: Names the service for tracing and logging.
- **Jaeger Configuration**: Sets up Jaeger tracer with service name and sampler.

6.3. Exposing Metrics Endpoint

With the above configuration, Spring Boot Actuator exposes the metrics at /actuator/prometheus.

Example:

Access http://<service-host>:8080/actuator/prometheus to view Prometheus metrics.

6.4. Logging with Structured Logs

Ensure logs are structured (e.g., JSON) for better parsing in ELK.

Logback Configuration (logback-spring.xml):

```
<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder class="net.logstash.logback.encoder.LogstashEncoder" />
  </appender>

  <root level="INFO">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

Explanation:

- **LogstashEncoder**: Formats logs in JSON, which Logstash can easily parse.

---

7. Automate Issue Detection

Automate the detection of issues using Prometheus alerting rules and Alertmanager.

7.1. Configuring Prometheus Alerting Rules

Add alerting rules to prometheus.yml or a separate rules.yml file.

rules.yml:

```
groups:
- name: microservices-alerts
 rules:
 - alert: HighResponseTime
  expr: histogram_quantile(0.95, sum(rate(http_request_duration_seconds_bucket[5m])) by (le, service)) >
1.0
  for: 2m
  labels:
   severity: warning
  annotations:
   summary: "High response time detected for service {{ $labels.service }}"
   description: "95th percentile response time is above 1 second."
```

Explanation:

- **alert**: Name of the alert (HighResponseTime).
- **expr**: PromQL expression to trigger the alert (95th percentile > 1 second).
- **for**: Duration the condition must be true before firing.
- **labels**: Metadata (e.g., severity).
- **annotations**: Additional information for notifications.

Include Rules in Prometheus Configuration:

```
# prometheus.yml

rule_files:
  - "rules.yml"
```

7.2. Setting Up Alertmanager

Alertmanager handles alerts sent by Prometheus and can route them to various receivers like email, Slack, or PagerDuty.

Installing Alertmanager:

```
# Download Alertmanager
wget https://github.com/prometheus/alertmanager/releases/download/v0.25.0/alertmanager-0.25.0.linux-amd64.tar.gz

# Extract
tar xvf alertmanager-0.25.0.linux-amd64.tar.gz

# Move binaries
sudo mv alertmanager-0.25.0.linux-amd64/alertmanager /usr/local/bin/
sudo mv alertmanager-0.25.0.linux-amd64/amtool /usr/local/bin/
```

Configuration (alertmanager.yml):

```
global:
  resolve_timeout: 5m

route:
  receiver: 'slack-notifications'

receivers:
- name: 'slack-notifications'
  slack_configs:
  - send_resolved: true
    text: "{{ range .Alerts }}{{ .Annotations.summary }}\n{{ .Annotations.description }}\n{{ end }}"
    api_url: 'https://hooks.slack.com/services/T00000000/B00000000/XXXXXXXXXXXXXXXXXXXXXXXX'
```

Explanation:

- **route.receiver**: Defines the default receiver (slack-notifications).
- **receivers**: Configures where to send alerts.
- **slack_configs**: Sends alerts to a Slack channel using a webhook URL.

Starting Alertmanager:

```
alertmanager --config.file=alertmanager.yml --storage.path=/tmp/alertmanager
```

7.3. Integrating Prometheus with Alertmanager

Add Alertmanager configuration to prometheus.yml.

```
# prometheus.yml

alerting:
 alertmanagers:
  - static_configs:
   - targets:
     - 'localhost:9093'  # Adjust if Alertmanager is on a different host/port
```

Restart Prometheus to apply changes.

```
sudo systemctl restart prometheus
```

---

8. Analyzing and Resolving Issues

Once alerts are in place, you can analyze and resolve high response time issues using collected metrics, traces, and logs.

8.1. Identifying the Culprit Microservice

1. **Alert Triggered**: Receive an alert indicating high response time for a specific service.
2. **Prometheus Metrics**: Check Prometheus/Grafana for detailed metrics on response times, request rates, error rates, etc.
3. **Jaeger Traces**: Use Jaeger to trace requests and identify where delays occur.
4. **ELK Logs**: Search logs in Kibana for error messages, stack traces, or unusual patterns.

8.2. Drilldown Steps

Step 1: Check Metrics in Grafana

- **Dashboard**: Open the response time dashboard.
- **Identify Service**: Locate the service with spikes in response time.
- **Correlate with Other Metrics**: Check CPU, memory, network usage to see if resource constraints are causing delays.

Step 2: Analyze Distributed Traces in Jaeger

- **Find Traces**: Search for recent traces during the alert period.
- **Examine Spans**: Identify which spans (operations) have high latency.
- **Service Dependencies**: Determine if downstream services are contributing to delays.

Step 3: Investigate Logs in Kibana

- **Filter Logs**: Search logs for the affected service during the alert period.
- **Error Patterns**: Look for error messages, exceptions, or warnings.
- **Performance Logs**: Identify slow database queries, external API calls, or other performance bottlenecks.

8.3. Resolving the Issue

Based on the analysis:

- **Code Optimization**: Optimize inefficient code paths identified in traces or logs.
- **Resource Scaling**: Scale up resources (CPU, memory) if metrics indicate resource exhaustion.
- **Database Tuning**: Optimize database queries or indexes if database latency is the issue.
- **Network Optimization**: Improve network configurations if network latency is detected.
- **Retry Mechanisms**: Implement or adjust retry strategies for failed external calls.

---

9. Real-World Scenario: Database Latency

Let's walk through a real-world scenario where a microservice experiences high response times due to database latency.

9.1. Scenario Overview

- **Issue**: Service OrderService has high response times.

- **Potential Cause**: Database queries within OrderService are slow.

9.2. Detection

1. **Alert**: Prometheus triggers HighResponseTime alert for OrderService.
2. **Metrics**: Grafana shows increased 95th percentile response time for OrderService.
3. **Tracing**: Jaeger traces indicate high latency in the db_query span.
4. **Logs**: Kibana logs show slow_query warnings.

9.3. Detailed Analysis

Step 1: Metrics Analysis

- **Grafana Dashboard**: View OrderService response times.
- **CPU and Memory**: No unusual spikes; resource usage is normal.
- **Database Metrics**: Check database server metrics (e.g., CPU, I/O).

Step 2: Distributed Tracing

- **Jaeger Traces**: Identify that the db_query span consistently takes ~500ms, whereas normal queries take ~100ms.
- **Service Dependencies**: OrderService depends on InventoryService and PaymentService, but the delay is isolated to database operations.

Step 3: Log Investigation

- **Kibana Search Query**:

```
{
 "query": {
  "bool": {
   "must": [
    { "match": { "service": "OrderService" }},
    { "match": { "log_level": "WARN" }},
     { "match": { "message": "slow_query" }}
   ],
   "filter": {
    "range": { "timestamp": { "gte": "now-5m" }}
   }
  }
 }
}
```

- **Findings**: Logs contain entries like:

  {"timestamp":"2024-12-31T23:58:00Z","service":"OrderService","log_level":"WARN","message":"slow_query","query":"SELECT * FROM orders WHERE user_id = ?","duration_ms":600}

9.4. Resolving the Database Latency

Step 1: Identify Slow Queries

- **Analyze Query Patterns**: Review the queries logged as `slow_query`.
- **Database Profiling**: Use database profiling tools to analyze query execution plans.

Step 2: Optimize Queries

- **Indexing**: Ensure that columns used in `WHERE` clauses are properly indexed.

  ```
  -- Adding index on user_id
  CREATE INDEX idx_orders_user_id ON orders(user_id);
  ```

- **Query Refactoring**: Optimize the query structure to reduce execution time.

Step 3: Database Configuration

- **Connection Pooling**: Ensure that the microservice uses an optimal number of database connections.

  Example (Spring Boot `application.yml`):

  ```
  spring:
   datasource:
    url: jdbc:mysql://db-host:3306/orders
    username: user
    password: pass
    hikari:
     maximum-pool-size: 20
  ```

- **Caching**: Implement caching for frequently accessed data to reduce database load.

Step 4: Deploy and Monitor

- **Deploy Changes**: Update the microservice with optimized queries and configurations.
- **Monitor Metrics**: Use Grafana and Jaeger to ensure that response times have improved.

- **Verify Logs**: Check Kibana to confirm that slow_query logs have decreased.

---

10. Automating Diagnosis with CI/CD

Integrating observability checks into the CI/CD pipeline ensures that performance issues are detected early during the development lifecycle.

10.1. Setting Up CI/CD Pipeline with Jenkins

Assuming you're using **Jenkins** as your CI/CD tool.

Jenkins Pipeline Script (Jenkinsfile):

```
pipeline {
  agent any

  stages {
    stage('Checkout') {
      steps {
        // Checkout code from repository
        git 'https://github.com/your-repo/microservice.git'
      }
    }

    stage('Build') {
      steps {
        // Build the application
        sh './gradlew build'
      }
    }

    stage('Unit Tests') {
      steps {
        // Run unit tests
        sh './gradlew test'
      }
    }

    stage('Integration Tests') {
      steps {
        // Run integration tests with observability
        sh './gradlew integrationTest'
      }
    }

    stage('Performance Tests') {
      steps {
```

```
            // Run performance tests and collect metrics
            sh './performance-tests.sh'
        }
    }

    stage('Static Code Analysis') {
        steps {
            // Analyze code for potential performance issues
            sh 'sonar-scanner'
        }
    }

    stage('Deploy to Staging') {
        steps {
            // Deploy the microservice to staging environment
            sh './deploy.sh staging'
        }
    }

    stage('Automated Diagnosis') {
        steps {
            script {
                // Trigger automated diagnosis scripts
                sh './diagnose-issues.sh'
            }
        }
    }

    stage('Approval') {
        steps {
            // Await manual approval
            input 'Approve Deployment to Production?'
        }
    }

    stage('Deploy to Production') {
        steps {
            // Deploy to production environment
            sh './deploy.sh production'
        }
    }
  }

  post {
    always {
      // Cleanup or notify
      echo 'Pipeline completed.'
    }
  }
}
```

Explanation:

- **Stages**:
  - **Checkout**: Pulls the latest code.
  - **Build**: Compiles the application.
  - **Unit Tests**: Runs unit tests.
  - **Integration Tests**: Runs integration tests with observability enabled.
  - **Performance Tests**: Executes performance benchmarks.
  - **Static Code Analysis**: Analyzes code quality and potential issues.
  - **Deploy to Staging**: Deploys to a staging environment for further testing.
  - **Automated Diagnosis**: Runs scripts to diagnose any detected issues automatically.
  - **Approval**: Waits for manual approval before production deployment.
  - **Deploy to Production**: Deploys the microservice to production.

10.2. Automating Issue Diagnosis

Create scripts that analyze metrics and logs to automatically detect and possibly remediate issues before deployment.

diagnose-issues.sh:

```
#!/bin/bash

# Fetch metrics from Prometheus
response_time=$(curl -s 'http://prometheus-
server:9090/api/v1/query?query=histogram_quantile(0.95,sum(rate(http_request_duration_seconds_bucket
[5m])) by (le, service)) > 1.0' | jq '.data.result | length')

if [ "$response_time" -gt 0 ]; then
  echo "High response time detected. Failing the build."
  exit 1
fi

echo "No high response time issues detected."
exit 0
```

Explanation:

- **Fetch Metrics**: Queries Prometheus for any services with 95th percentile response time > 1 second.
- **Check Results**: If any results are found, the script exits with a non-zero status, failing the build.
- **Automation**: Prevents deployment of services with known performance issues.

Integration in Jenkinsfile:

The Automated Diagnosis stage runs this script. If the script fails, the pipeline halts, preventing deployment.

---

11. Conclusion

Managing a massive microservices architecture with **100,000 services** requires a robust and comprehensive observability framework. By implementing the following components, you can effectively monitor, trace, log, detect, and resolve performance issues:

1. **Metrics Collection**: Use **Prometheus** and **Grafana** for real-time metrics.
2. **Distributed Tracing**: Implement **Jaeger** to trace requests across services.
3. **Centralized Logging**: Utilize the **ELK Stack** for log aggregation and analysis.
4. **Instrumentation**: Embed observability into your microservices using libraries like **Micrometer** and **Jaeger clients**.
5. **Automated Issue Detection**: Set up alerts with **Prometheus Alertmanager** and integrate them into your CI/CD pipeline.
6. **Analysis and Resolution**: Leverage collected data to perform deep dives and resolve issues promptly.
7. **CI/CD Integration**: Automate performance checks and issue diagnosis within your deployment pipelines.

By following this detailed framework, you can maintain high performance and reliability in your microservices ecosystem, ensuring that any high response time issues are swiftly identified and addressed.

Additional Resources

- **Prometheus Documentation**: https://prometheus.io/docs/
- **Grafana Documentation**: https://grafana.com/docs/
- **Jaeger Documentation**: https://www.jaegertracing.io/docs/
- **ELK Stack Documentation**: https://www.elastic.co/what-is/elk-stack
- **Micrometer Documentation**: https://micrometer.io/docs
- **Spring Boot Actuator**: https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html