Creating an flask and app and Grafana dashboard:

Some theory about the facts that we have covered within out app:

- **Latency:** How long it takes to serve a customer from when they order.
  In our script: We measure how long each web request takes to complete using app_request_latency_seconds. Just like timing how long it takes from a customer ordering food until they receive it.
- **Traffic:** How many customers are coming in per hour.
  In our script: We count incoming requests with app_request_count. This is like having a counter at the restaurant door that clicks each time someone enters.
- **Errors:** How many orders are returned or complaints received.
  In our script: We track HTTP error codes (like 500 errors) and explicit error counters. This is like tracking how many dishes are sent back to the kitchen or how many customers complain.
- **Saturation:** How close the restaurant is to maximum capacity.
  In our script: We use app_active_requests to see how many requests are being processed simultaneously. This is like counting how many tables are occupied at once in the restaurant.
- **Observability Foundations:**
  Think of operating a commercial airplane:
  - **Metrics:** Quantitative measurements like altitude, speed, and fuel level.
    In our script: Prometheus collects precise numbers about our system's performance, like counting requests or measuring response times.
  - **Logs:** The flight recorder (black box) that captures detailed events.

In our script: Our application writes detailed logs with timestamps and request IDs, collected by Loki. This is like recording every action taken by pilots and every system event.

- **Visualization:** The cockpit dashboard that displays important readings.
  In our script: Grafana creates visual dashboards that make it easy to understand system behavior at a glance.
- **Instrumentation:** The sensors throughout the aircraft that capture data.
  In our script: We've added code to the application to capture key information about its behavior.
- **Health Monitoring:**
  Think of a doctor checking a patient's health:
  Liveness Probes: Checking if the patient is alive (basic pulse check).
  In our script: The /health/liveness endpoint tells Kubernetes if our application is running at all.
- **Readiness Probes**: Determining if the patient can perform activities.
  In our script: The /health/readiness endpoint checks if our application can handle requests properly.
- **Health Endpoints:** Different types of medical tests (blood pressure, temperature, etc.).
  In our script: We have different endpoints that check different aspects of health.
- **Dependency Checks:** Making sure all organs are functioning together.
  In our script: We verify our "database connection" is working before declaring the app ready.

- **Service Level Objectives (SLO) Building Blocks**:
  Think of a pizza delivery promise:
  - **Error Rate Tracking:** Monitoring how many pizzas are delivered incorrectly.
    In our script: We track how many requests result in errors.
  - **Latency Histograms:** Recording delivery times across all orders.
    In our script: We track request durations in different "buckets" to understand response time patterns.
  - **Request Success Rate:** Calculating what percentage of pizzas are delivered correctly.
    In our script: We can determine what percentage of requests complete successfully.
  - These are the building blocks to create promises like "95% of pizzas will be delivered correctly within 30 minutes" (similar to "99.9% of requests will complete successfully in under 500ms").

The Script code:

```
#!/bin/bash

# Exit on error
set -e

echo "============================================================="
echo "  LIGHTWEIGHT SRE MONITORING SETUP FOR MINIKUBE/WSL"
echo "============================================================="
echo "This script will:"
echo "  1. Reset minikube for a clean environment"
echo "  2. Create a lightweight Python Flask API with SRE instrumentation"
echo "  3. Deploy it to Kubernetes with appropriate waits between steps"
echo "  4. Set up minimal but effective monitoring"
echo "============================================================="

# Function to check if a command exists
check_command() {
  if ! command -v "$1" &> /dev/null; then
    echo "Error: $1 is required but not installed."
    exit 1
  fi
}

# Function to wait with a countdown
wait_with_message() {
  local seconds=$1
  local message=$2

  echo -n "$message "
  for (( i=$seconds; i>0; i-- )); do
    echo -n "$i... "
    sleep 1
  done
  echo "Done!"
}

# Check for required commands
echo "Checking prerequisites..."
check_command kubectl
check_command docker
check_command minikube
check_command helm

# Step 1: Clean up any existing setup
echo "Cleaning up previous setup if it exists..."
minikube delete || true
```

```
wait_with_message 3 "Waiting for minikube to fully shut down..."

# Step 2: Start minikube with appropriate resources
echo "Starting minikube with optimized settings for WSL..."
minikube start --driver=docker --cpus=2 --memory=3072 --disk-size=10g

# Verify minikube is running
echo "Verifying minikube status..."
minikube status
echo "Kubernetes version:"
kubectl version

# Step 3: Create a project directory
echo "Creating project directory..."
mkdir -p sre-flask-app
cd sre-flask-app

# Step 4: Create a smaller, optimized Python Flask application
echo "Creating Python Flask application with SRE instrumentation..."

# Create main app file
cat > app.py << 'EOF'
import os
import time
import random
import logging
from datetime import datetime
from flask import Flask, request, jsonify
from prometheus_client import Counter, Histogram, Gauge, generate_latest,
CONTENT_TYPE_LATEST
from werkzeug.middleware.dispatcher import DispatcherMiddleware
from prometheus_client import make_wsgi_app

# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s [%(levelname)s] %(message)s -
request_id=%(request_id)s',
    datefmt='%Y-%m-%d %H:%M:%S'
)

# Initialize Flask app
app = Flask(__name__)

# Add Prometheus WSGI middleware
app.wsgi_app = DispatcherMiddleware(app.wsgi_app, {
    '/metrics': make_wsgi_app()
})
```

```
# Define Prometheus metrics
REQUEST_COUNT = Counter(
    'app_request_count',
    'Application Request Count',
    ['endpoint', 'method', 'http_status']
)
REQUEST_LATENCY = Histogram(
    'app_request_latency_seconds',
    'Application Request Latency',
    ['endpoint', 'method'],
    buckets=[0.01, 0.05, 0.1, 0.5, 1, 2, 5]
)
ERROR_COUNTER = Counter(
    'app_error_count',
    'Application Error Count',
    ['error_type', 'endpoint']
)
ACTIVE_REQUESTS = Gauge(
    'app_active_requests',
    'Active Requests Currently Being Processed'
)

# Sample database - kept small for memory efficiency
users_db = [
    {"id": 1, "name": "Alice", "email": "alice@example.com"},
    {"id": 2, "name": "Bob", "email": "bob@example.com"}
]

# Simulated database query with variable latency
def simulate_db_query(query_type="read"):
    # Simulate different query times based on query type
    time_ranges = {
        "read": (0.001, 0.05),
        "write": (0.005, 0.1),
        "complex": (0.01, 0.2)
    }

    min_time, max_time = time_ranges.get(query_type, (0.001, 0.05))

    # Occasionally simulate slow queries
    if random.random() < 0.05:  # 5% chance
        query_time = random.uniform(max_time, max_time * 4)
        app.logger.warning(
            f"Slow {query_type} query detected, took {query_time:.4f}s",
            extra={"request_id": getattr(request, 'request_id', 'unknown')}
        )
    else:
```

```
        query_time = random.uniform(min_time, max_time)

    time.sleep(query_time)
    return query_time

# Request logging middleware
@app.before_request
def before_request():
    request.start_time = time.time()
    ACTIVE_REQUESTS.inc()

    # Add request ID for tracking
    request.request_id = str(random.randint(1000, 9999))

@app.after_request
def after_request(response):
    ACTIVE_REQUESTS.dec()

    # Calculate request processing time
    request_time = time.time() - request.start_time

    # Record request latency
    REQUEST_LATENCY.labels(
        endpoint=request.path,
        method=request.method
    ).observe(request_time)

    # Record request count
    REQUEST_COUNT.labels(
        endpoint=request.path,
        method=request.method,
        http_status=response.status_code
    ).inc()

    # Log request details
    app.logger.info(
        f"Processed {request.method} {request.path} in {request_time:.4f}s
with status {response.status_code}",
        extra={"request_id": getattr(request, 'request_id', 'unknown')}
    )

    return response

# Health check endpoint (for Kubernetes probes)
@app.route('/health/liveness')
def liveness_check():
    # Simple liveness check - just verify the app is running
```

```
    return jsonify({"status": "alive", "timestamp":
datetime.now().isoformat()}), 200

@app.route('/health/readiness')
def readiness_check():
    # More complex readiness check - verify dependencies
    try:
        simulate_db_query("read")
        return jsonify({"status": "ready", "checks": {"database":
"connected"}}), 200
    except Exception as e:
        ERROR_COUNTER.labels(error_type="dependency_failure",
endpoint="/health/readiness").inc()
        return jsonify({"status": "not ready", "checks": {"database":
str(e)}}), 503

# API endpoints
@app.route('/')
def root():
    return jsonify({
        "service": "SRE Demo API",
        "version": "1.0.0",
        "endpoints": [
            "/api/users",
            "/api/echo",
            "/api/error",
            "/api/slow",
            "/metrics"
        ]
    })

@app.route('/api/users')
def get_users():
    # Simulate DB query
    simulate_db_query("read")

    # Occasionally simulate an error
    if random.random() < 0.02:  # 2% error rate
        ERROR_COUNTER.labels(error_type="database_error",
endpoint="/api/users").inc()
        app.logger.error(
            "Database error occurred: Failed to retrieve users",
            extra={"request_id": getattr(request, 'request_id', 'unknown')}
        )
        return jsonify({"error": "Database error"}), 500

    return jsonify(users_db)
```

```
@app.route('/api/echo', methods=['POST'])
def echo():
    # Echo back the JSON received
    data = request.get_json(silent=True)
    if data is None:
        ERROR_COUNTER.labels(error_type="invalid_input",
endpoint="/api/echo").inc()
        return jsonify({"error": "Invalid JSON"}), 400

    # Simulate a write operation
    simulate_db_query("write")

    return jsonify(data)

@app.route('/api/error')
def simulate_error():
    # Deliberately cause an error for testing
    ERROR_COUNTER.labels(error_type="simulated_error",
endpoint="/api/error").inc()

    error_type = request.args.get('type', 'server')

    if error_type == 'client':
        app.logger.warning(
            "Client error simulated",
            extra={"request_id": getattr(request, 'request_id', 'unknown')}
        )
        return jsonify({"error": "Bad Request Simulation"}), 400
    else:
        app.logger.error(
            "Server error simulated",
            extra={"request_id": getattr(request, 'request_id', 'unknown')}
        )
        return jsonify({"error": "Internal Server Error Simulation"}), 500

@app.route('/api/slow')
def slow_response():
    # Simulate a slow API response
    delay = min(float(request.args.get('delay', 1)), 5)  # Cap at 5 seconds
for resource efficiency

    app.logger.info(
        f"Processing slow request with {delay}s delay",
        extra={"request_id": getattr(request, 'request_id', 'unknown')}
    )

    # Simulate a complex query
    simulate_db_query("complex")
```

```
    # Additional delay
    time.sleep(delay)

    return jsonify({"message": f"Slow response completed after {delay}
seconds"})

# Run the application
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=int(os.environ.get('PORT', 5000)))
EOF

# Create requirements.txt
cat > requirements.txt << 'EOF'
Flask==2.2.3
prometheus-client==0.16.0
gunicorn==20.1.0
Werkzeug==2.2.3
EOF

# Create Dockerfile optimized for size
cat > Dockerfile << 'EOF'
FROM python:3.9-slim as builder

WORKDIR /app
COPY requirements.txt .
RUN pip wheel --no-cache-dir --no-deps --wheel-dir /app/wheels -r
requirements.txt

FROM python:3.9-slim
WORKDIR /app
COPY --from=builder /app/wheels /wheels
RUN pip install --no-cache /wheels/*

COPY app.py .
ENV PORT=8080
ENV PYTHONUNBUFFERED=1

HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
  CMD curl -f http://localhost:${PORT}/health/liveness || exit 1

# Use gunicorn with fewer workers for resource efficiency
CMD ["gunicorn", "--bind", "0.0.0.0:8080", "--workers", "1", "--threads", "2",
"app:app"]

EXPOSE 8080
EOF
```

```
# Step 5: Create minimal Kubernetes manifests
echo "Creating Kubernetes manifests..."

# Create namespace
cat > k8s-namespace.yaml << 'EOF'
apiVersion: v1
kind: Namespace
metadata:
  name: sre-demo
EOF

# Create ConfigMap for app configuration
cat > k8s-configmap.yaml << 'EOF'
apiVersion: v1
kind: ConfigMap
metadata:
  name: sre-flask-app-config
  namespace: sre-demo
data:
  LOG_LEVEL: "INFO"
EOF

# Create Service
cat > k8s-service.yaml << 'EOF'
apiVersion: v1
kind: Service
metadata:
  name: sre-flask-app
  namespace: sre-demo
  labels:
    app: sre-flask-app
  annotations:
    prometheus.io/scrape: "true"
    prometheus.io/path: "/metrics"
    prometheus.io/port: "8080"
spec:
  ports:
  - port: 80
    targetPort: 8080
    name: http
  selector:
    app: sre-flask-app
EOF

# Create Deployment with minimal resource requirements
cat > k8s-deployment.yaml << 'EOF'
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: sre-flask-app
  namespace: sre-demo
  labels:
    app: sre-flask-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: sre-flask-app
  template:
    metadata:
      labels:
        app: sre-flask-app
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/path: "/metrics"
        prometheus.io/port: "8080"
    spec:
      containers:
      - name: sre-flask-app
        image: sre-flask-app:latest
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8080
        envFrom:
        - configMapRef:
            name: sre-flask-app-config
        resources:
          requests:
            cpu: "50m"
            memory: "64Mi"
          limits:
            cpu: "200m"
            memory: "128Mi"
        livenessProbe:
          httpGet:
            path: /health/liveness
            port: 8080
          initialDelaySeconds: 10
          periodSeconds: 30
          timeoutSeconds: 3
        readinessProbe:
          httpGet:
            path: /health/readiness
            port: 8080
          initialDelaySeconds: 5
          periodSeconds: 10
```

```
          timeoutSeconds: 2
EOF

# Step 6: Create a lightweight monitoring setup script
echo "Creating monitoring setup script..."

cat > setup-monitoring.sh << 'EOF'
#!/bin/bash
set -e

echo "Setting up minimal monitoring stack..."

# Create monitoring namespace
kubectl create namespace monitoring --dry-run=client -o yaml | kubectl apply -
f -

# Add Helm repositories if they don't exist
if ! helm repo list | grep -q "prometheus-community"; then
  echo "Adding Prometheus Helm repository..."
  helm repo add prometheus-community https://prometheus-
community.github.io/helm-charts
fi

if ! helm repo list | grep -q "grafana"; then
  echo "Adding Grafana Helm repository..."
  helm repo add grafana https://grafana.github.io/helm-charts
fi

helm repo update

# Install a minimal Prometheus (without many components to save resources)
echo "Installing Prometheus (lightweight configuration)..."
cat > prometheus-values.yaml << 'EOPROMETHEUS'
alertmanager:
  enabled: false
pushgateway:
  enabled: false
nodeExporter:
  enabled: false
server:
  persistentVolume:
    enabled: false
  resources:
    requests:
      cpu: 100m
      memory: 256Mi
    limits:
      cpu: 200m
```

```
      memory: 512Mi
EOPROMETHEUS

helm upgrade --install prometheus prometheus-community/prometheus \
  --namespace monitoring \
  --values prometheus-values.yaml \
  --wait

echo "Waiting for Prometheus to be fully ready..."
kubectl wait --for=condition=available --timeout=120s deployment/prometheus-
server -n monitoring

# Install a lightweight version of Loki
echo "Installing Loki (lightweight configuration)..."
cat > loki-values.yaml << 'EOLOKI'
loki:
  persistence:
    enabled: false
  resources:
    requests:
      cpu: 50m
      memory: 128Mi
    limits:
      cpu: 100m
      memory: 256Mi
promtail:
  resources:
    requests:
      cpu: 20m
      memory: 64Mi
    limits:
      cpu: 50m
      memory: 128Mi
  config:
    snippets:
      pipelineStages:
        - docker: {}
EOLOKI

helm upgrade --install loki grafana/loki-stack \
  --namespace monitoring \
  --set grafana.enabled=false \
  --set prometheus.enabled=false \
  --values loki-values.yaml \
  --wait

echo "Waiting for Loki to be fully ready..."
kubectl rollout status statefulset/loki -n monitoring --timeout=120s
```

```
# Install Grafana with minimal resources
echo "Installing Grafana (lightweight configuration)..."
cat > grafana-values.yaml << 'EOGRAFANA'
persistence:
  enabled: false
resources:
  requests:
    cpu: 50m
    memory: 128Mi
  limits:
    cpu: 100m
    memory: 256Mi
datasources:
  datasources.yaml:
    apiVersion: 1
    datasources:
    - name: Prometheus
      type: prometheus
      url: http://prometheus-server.monitoring.svc.cluster.local
      access: proxy
      isDefault: true
    - name: Loki
      type: loki
      url: http://loki.monitoring.svc.cluster.local:3100
      access: proxy
adminUser: admin
adminPassword: admin
EOGRAFANA

helm upgrade --install grafana grafana/grafana \
  --namespace monitoring \
  --values grafana-values.yaml \
  --wait

echo "Waiting for Grafana to be fully ready..."
kubectl rollout status deployment/grafana -n monitoring --timeout=120s

echo "Monitoring stack has been set up successfully!"
echo "Access Grafana:"
echo "  kubectl port-forward svc/grafana 3000:80 -n monitoring"
echo "  Default credentials: admin / admin"
EOF

chmod +x setup-monitoring.sh

# Step 7: Create a simple load testing script (very lightweight)
echo "Creating load testing script..."
```

```
cat > load-test.sh << 'EOF'
#!/bin/bash

# Simple load test script that uses minimal resources
API_URL="http://$(kubectl get svc -n sre-demo sre-flask-app -o
jsonpath='{.spec.clusterIP}')"

echo "Starting minimal load test against $API_URL"
echo "Press Ctrl+C to stop..."

# Function to make a request
make_request() {
  local endpoint="$1"
  local method="$2"
  local data="$3"

  if [ "$method" = "POST" ]; then
    curl -s -X POST -H "Content-Type: application/json" -d "$data"
"$API_URL$endpoint" > /dev/null
  else
    curl -s "$API_URL$endpoint" > /dev/null
  fi

  echo "$(date +%H:%M:%S) - Made $method request to $endpoint"
}

# Main loop - very lightweight to avoid resource contention
while true; do
  # Choose a random endpoint
  endpoints=("/api/users" "/api/error" "/api/slow")
  endpoint=${endpoints[$RANDOM % ${#endpoints[@]}]}

  case "$endpoint" in
    "/api/echo")
      make_request "$endpoint" "POST" '{"test":"data"}'
      ;;
    "/api/error")
      make_request "$endpoint" "GET"
      ;;
    "/api/slow")
      make_request "$endpoint?delay=0.5" "GET"  # Short delay to avoid
resource issues
      ;;
    *)
      make_request "$endpoint" "GET"
      ;;
  esac
```

```
  # Sleep longer between requests to use fewer resources
  sleep 3
done
EOF

chmod +x load-test.sh

# Step 8: Create instruction file for dashboards
echo "Creating Grafana dashboard instructions..."

cat > grafana-dashboard-instructions.md << 'EOF'
# SRE Flask App Grafana Dashboard Setup Instructions

This guide will help you create monitoring dashboards in Grafana to visualize
the SRE metrics from your Flask application.

## Basic SRE Dashboard

### Step 1: Create a New Dashboard

1. Click the "+" icon in the left sidebar
2. Select "Dashboard" from the dropdown menu
3. Click "Add new panel"

### Step 2: Request Rate Panel (Traffic)

1. In the query editor, select "Prometheus" as the data source
2. Enter this PromQL query:
   ```
   sum(rate(app_request_count{namespace="sre-demo"}[1m])) by (endpoint)
   ```
3. Set visualization to "Time series"
4. Title: "Request Rate by Endpoint"
5. Under "Standard options", set unit to "requests/sec"
6. Click "Apply"

### Step 3: Error Rate Panel (Errors)

1. Click "Add panel"
2. Select "Prometheus" as the data source
3. Enter this query:
   ```
   sum(rate(app_request_count{namespace="sre-demo", http_status=~"5.."}[1m]))
/ sum(rate(app_request_count{namespace="sre-demo"}[1m])) * 100
   ```
4. Set visualization to "Stat"
5. Title: "Error Rate (%)"
```

````
6. Under "Standard options", set unit to "Percent (0-100)"
7. Add thresholds: 0-1 green, 1-5 orange, 5-100 red
8. Click "Apply"

### Step 4: Latency Panel (Latency)

1. Click "Add panel"
2. Select "Prometheus" as the data source
3. Enter this query:
   ```
   histogram_quantile(0.95,
sum(rate(app_request_latency_seconds_bucket{namespace="sre-demo"}[5m])) by
(le, endpoint))
   ```
4. Set visualization to "Time series"
5. Title: "95th Percentile Latency by Endpoint"
6. Under "Standard options", set unit to "seconds"
7. Click "Apply"

### Step 5: Active Requests Panel (Saturation)

1. Click "Add panel"
2. Select "Prometheus" as the data source
3. Enter this query:
   ```
   app_active_requests{namespace="sre-demo"}
   ```
4. Set visualization to "Stat"
5. Title: "Active Requests"
6. Click "Apply"

### Step 6: Save the Dashboard

1. Click the save icon in the top-right corner
2. Name the dashboard "SRE Basic Dashboard"
3. Click "Save"

## Log Analysis Dashboard

### Step 1: Create a New Dashboard

1. Click the "+" icon in the left sidebar
2. Select "Dashboard" from the dropdown
3. Click "Add new panel"

### Step 2: Create Log Panel

1. Select "Loki" as the data source
````

2. Enter this LogQL query:
   ```
   {namespace="sre-demo"}
   ```
3. Set visualization to "Logs"
4. Title: "Application Logs"
5. Enable "Show time" under Options
6. Click "Apply"

Step 3: Create Error Logs Panel

1. Click "Add panel"
2. Select "Loki" as the data source
3. Enter this query:
   ```
   {namespace="sre-demo"} |= "error"
   ```
4. Set visualization to "Logs"
5. Title: "Error Logs"
6. Click "Apply"

Step 4: Create Log Volume Panel

1. Click "Add panel"
2. Select "Loki" as the data source
3. Enter this query:
   ```
   sum(count_over_time({namespace="sre-demo"}[1m]))
   ```
4. Set visualization to "Time series"
5. Title: "Log Volume"
6. Click "Apply"

Step 5: Save the Dashboard

1. Click the save icon in the top-right corner
2. Name the dashboard "Log Analysis"
3. Click "Save"

Creating Dashboard Variables (Optional)

To make your dashboards more flexible:

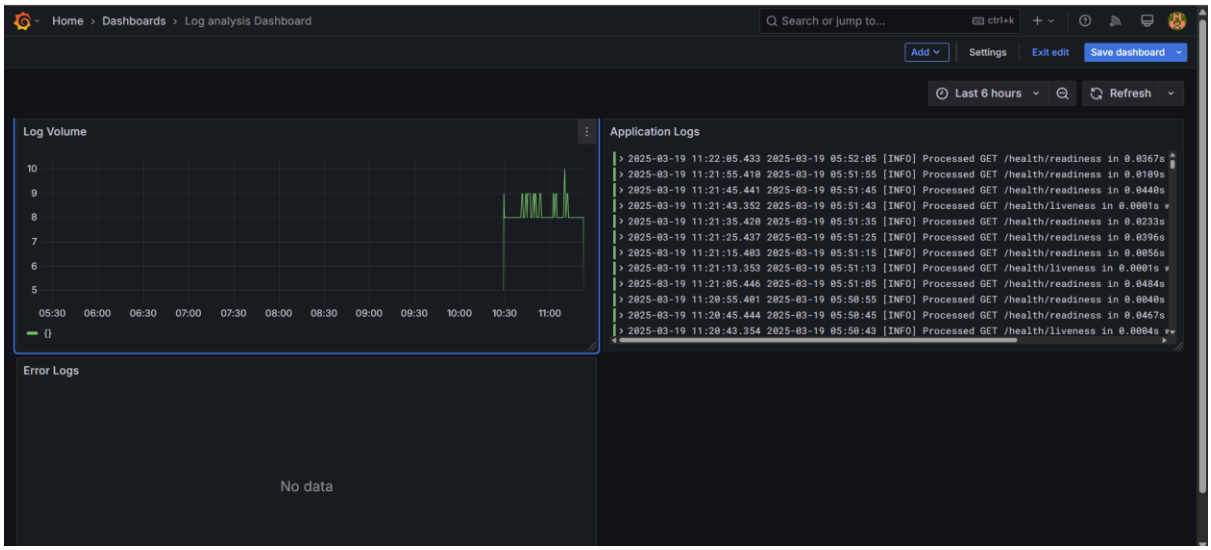
1. Click the gear icon in the top-right corner
2. Select "Variables" from the left menu
3. Click "Add variable"
4. Configure a namespace variable:
   - Name: namespace

```
   - Type: Query
   - Data source: Prometheus
   - Query: `label_values(kube_namespace_labels, namespace)`
5. Click "Add" then "Save"
6. Update your queries to use the variable:
   - Change `namespace="sre-demo"` to `namespace="$namespace"`

This allows you to switch between different namespaces if you deploy multiple
versions of the application.
EOF

# Step 9: Build the Docker image with minikube's Docker environment
echo "Building Docker image in minikube's Docker environment..."
eval $(minikube docker-env)
docker build -t sre-flask-app:latest .

# Step 10: Deploy to Kubernetes
echo "Deploying application to Kubernetes..."
kubectl apply -f k8s-namespace.yaml
kubectl apply -f k8s-configmap.yaml
kubectl apply -f k8s-service.yaml
kubectl apply -f k8s-deployment.yaml

# Wait for deployment to be ready
echo "Waiting for deployment to be ready..."
kubectl rollout status deployment/sre-flask-app -n sre-demo --timeout=120s

echo "Application deployed successfully!"

# Step 11: Set up monitoring with proper waits
echo "Setting up monitoring stack..."
./setup-monitoring.sh

# Step 12: Start load generator in background
echo "Starting minimal load generator in background..."
./load-test.sh > load-test.log 2>&1 &
LOAD_TEST_PID=$!
echo $LOAD_TEST_PID > load-test.pid
echo "Load generator started with PID $LOAD_TEST_PID. Logs in load-test.log"

# Step 13: Set up port forwarding for Grafana
echo "Setting up port forwarding for Grafana..."
kubectl port-forward svc/grafana 3000:80 -n monitoring &
GRAFANA_PID=$!
echo $GRAFANA_PID > grafana.pid
echo "Grafana port-forwarding started with PID $GRAFANA_PID"

echo "================================================================="
```

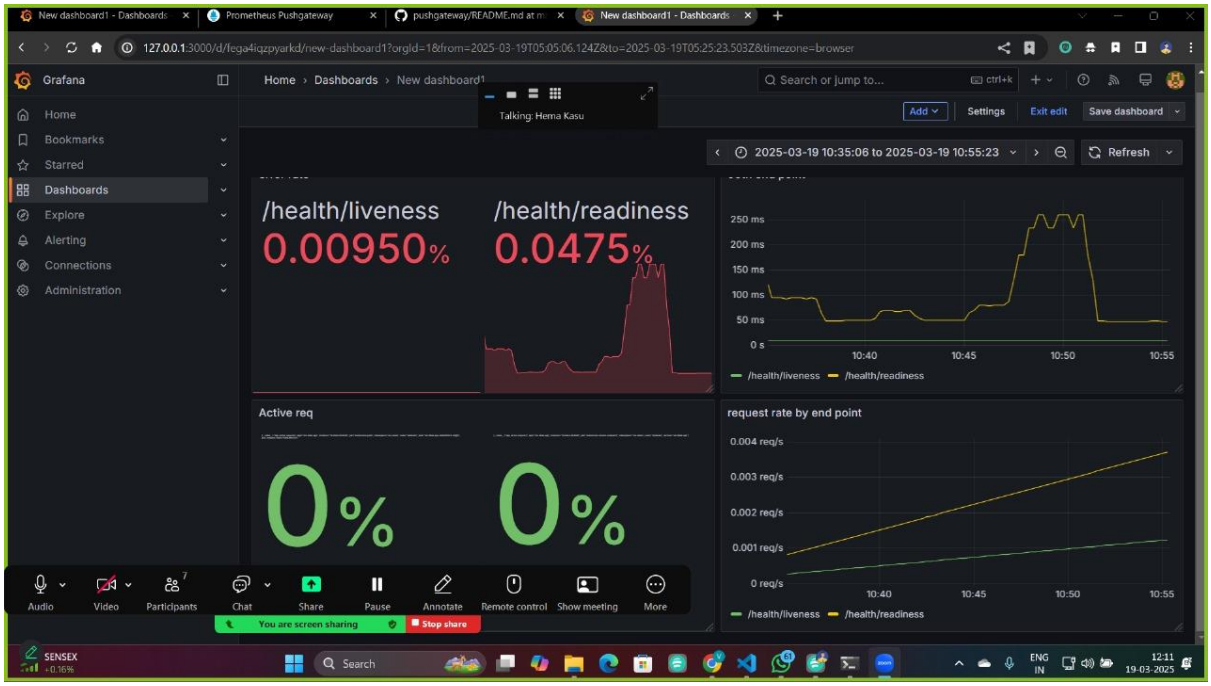
```
echo "SETUP COMPLETE!"
echo "========================================================================"
echo "Your SRE Flask App and monitoring are now running!"
echo ""
echo "Access your application:"
echo "  kubectl port-forward svc/sre-flask-app 8080:80 -n sre-demo"
echo ""
echo "Access Grafana:"
echo "  Grafana URL: http://localhost:3000"
echo "  Username: admin"
echo "  Password: admin"
echo ""
echo "Follow the instructions in grafana-dashboard-instructions.md"
echo "to set up your Grafana dashboards."
echo ""
echo "To clean up all resources, run:"
echo "  kill \$(cat load-test.pid)"
echo "  kill \$(cat grafana.pid)"
echo "  kubectl delete namespace sre-demo"
echo "  kubectl delete namespace monitoring"
echo "  minikube stop"
echo "========================================================================"
```

Then after creating dashboard in Grafana:

Dashboard for error log analysis:

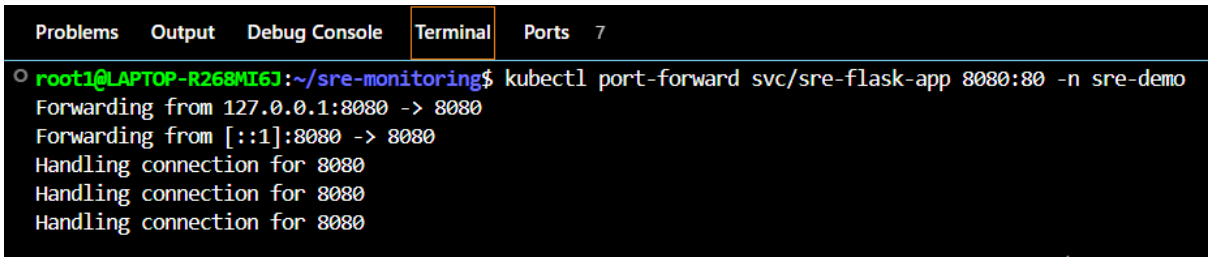


And the dashboard for Basic SRE analysis:

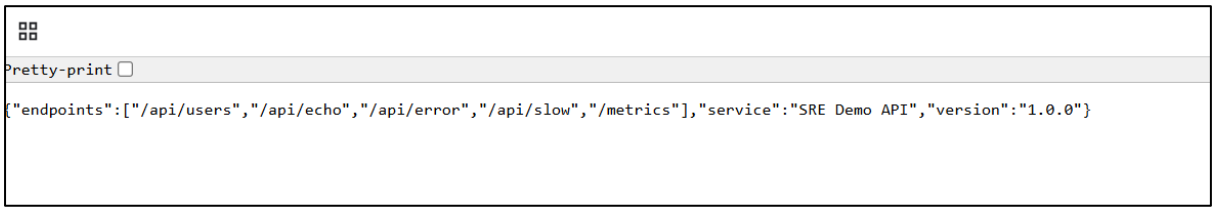


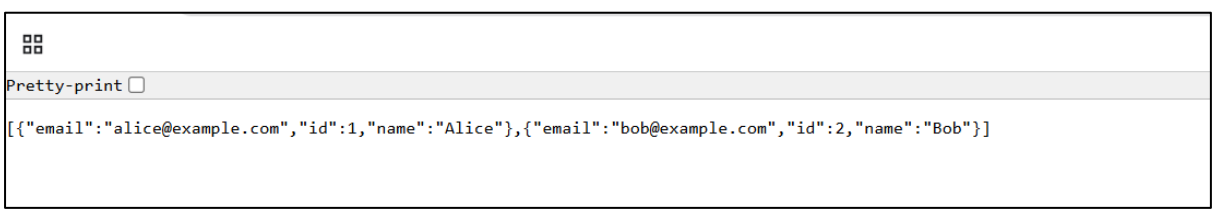
Now if we use another way of visualizing the 4th graph using time series then it would show data like:

And when we run the flask app after getting its server using this command:



In our browser without css it would be like:





And similarly we can access other url requests as well.

But after adding the static files like html and css: