**Lab Guide: Define a Service with SLIs, SLOs, and SLAs**

**Lab Objective**

By the end of this lab, participants will be able to:

1. Define a service in simple business terms.
2. Identify appropriate **Service Level Indicators (SLIs)** for that service.
3. Write measurable **Service Level Objectives (SLOs)**.
4. Translate SLOs into business-facing **Service Level Agreements (SLAs)**.
5. Understand the difference between **internal reliability targets** (SLOs) and **external commitments** (SLAs).

**Pre-Lab Context**

Reliability engineering isn’t about chasing “100% uptime” — it’s about defining **how reliable is reliable enough** for a given service. We do this by:

* **SLI** → Measurement (e.g., “latency < 200ms”).
* **SLO** → Target (e.g., “99% of requests < 200ms over 30 days”).
* **SLA** → Contract (e.g., “99.5% uptime per month, otherwise credits to customer”).

This exercise uses a **Retail Checkout API** as the sample service.

**Lab Steps**

**Step 1: Define the Service**

* **Service Name:** Retail Checkout API
* **Purpose:** Processes customer orders (cart → payment → confirmation).
* **Criticality:** Core revenue-generating service (downtime directly = lost sales).

**Step 2: Identify User Journeys**

List what users actually care about (not what engineers think is “cool”):

1. Can I successfully place an order?
2. Is it fast enough that I don’t abandon my cart?
3. Is the system available when I need it?

**Step 3: Define Sample SLIs**

Pick **2–3 indicators per service**:

1. **Availability SLI:**
   * Definition: Ratio of successful checkout requests to total requests.
   * Formula: SuccessfulRequests / TotalRequests.
2. **Latency SLI:**
   * Definition: Time taken to complete checkout request.
   * Metric: 95th percentile latency in milliseconds.
3. **Error Rate SLI:**
   * Definition: Percentage of requests returning HTTP 5xx errors.

**Note:** SLIs must be *measurable* from monitoring data, not vague terms like “fast.”

**Step 4: Write SLOs**

Now, set **targets** for those SLIs:

1. **Availability SLO:**
   * Target: 99.9% successful checkout requests per calendar month.
   * Rolling window: 30 days.
2. **Latency SLO:**
   * Target: 95% of checkout requests complete < 300ms over 28 days. : 1000 total request within a month
3. **Error Rate SLO:**
   * Target: Less than 0.1% of checkout requests result in HTTP 5xx over 30 days.

**Step 5: Translate into SLA**

SLAs are written for the **business/customer**, not engineers:

* **Availability SLA:**
  + “We guarantee 99.5% monthly uptime for the Checkout API. If availability drops below this threshold, affected customers will receive a 10% service credit.”
* **Latency SLA:**

We guarantee that 90% of checkout requests will complete within 500ms per calendar month. If this target is not met, customers may request service credits.

* **Error Rate SLA:**

We guarantee that no more than 0.5% of checkout requests will return server errors in a given month. If exceeded, credits will be applied

**Key Point:**

* SLOs (internal) are usually stricter (e.g., 99.9%) than SLAs (external, 99.5%).
* This buffer protects the team while ensuring fairness to customers.

**Step 6: Document Your Service Contract**

| **Aspect** | **Definition** |
| --- | --- |
| **Service** | Retail Checkout API |
| **SLIs** | Availability (success ratio), Latency (95th percentile), Error Rate (5xx) |
| **SLOs** | 99.9% availability; 95% of requests <300ms; <0.1% 5xx errors |
| **SLA** | 99.5% monthly uptime; credits for violations |
| **Error Budget** | 0.1% of requests allowed to fail = 1,000 out of 1,000,000 requests within a month |

**Step 7: Questions you shud ask yourself**

* Which SLIs matter most for your business?
* Would you set tighter SLOs for internal teams than for customers? Why?
* How would your SLA change if the service was **internal-only** vs **external-facing**?

**🚖 Exercise: Define SLIs, SLOs, SLAs, and Error Budgets for a Ride-Hailing App**

**Scenario**

You are on the SRE/TPM team for a **ride-hailing service (like a mini-Uber)**.  
Your service connects **riders and drivers in real time**. Users quickly get frustrated if:

* They cannot book a ride,
* It takes too long to get a driver assigned, or
* The system is unavailable.

Your job is to define **SLIs, SLOs, SLAs, and Error Budgets** for this service.

**Tasks**

1. Identify **at least 3 SLIs** that represent what matters most to riders.
2. Define measurable **SLOs** for each SLI.
3. Translate these into **customer-facing SLAs**, including what happens if the SLA is breached.
4. Calculate **error budgets** based on your chosen SLOs.
   * Assume the system handles **1,000,000 ride requests per month**.

**Deliverable**

Prepare a table with:

| Metric | SLI | SLO | SLA | Error Budget |

**🔬 Lab Guide: Simulating SLO Tracking in a Spreadsheet**

**🎯 Learning Objectives**

By the end of this lab, trainees will be able to:

1. Track reliability using **SLIs** over a defined period.
2. Compare measured performance against an **SLO target**.
3. Calculate **error budget consumption**.
4. Interpret whether the service is **healthy** or if corrective action is needed.

**📘 Background for Trainees**

In real-world systems, monitoring tools (Prometheus, Datadog, CloudWatch, etc.) feed metrics into dashboards. But before learning tools, we’ll simulate this with a **spreadsheet**.

We will:

* Pretend we are tracking an **e-commerce product catalog service**.
* The **SLO** is: *99.9% availability per 30 days*.
* The **SLI** we care about is: *% of successful requests*.

**🛠️ Step-by-Step Instructions**

**Step 1: Simulate data**

| **Date** | **Total Requests** | **Failed Requests** | **Success Rate (%)** | **Target SLO (99.9%)** | **Error Budget Used (%)** | **Status** |
| --- | --- | --- | --- | --- | --- | --- |

**Step 2: Add Sample Monitoring Data**

Use 10–15 rows of **simulated daily data**

Example dataset:

| **Date** | **Total Requests** | **Failed Requests** |
| --- | --- | --- |
| Day 1 | 100,000 | 10 |
| Day 2 | 120,000 | 50 |
| Day 3 | 90,000 | 100 |
| Day 4 | 110,000 | 20 |
| Day 5 | 105,000 | 500 |
| Day 6 | 98,000 | 70 |
| Day 7 | 115,000 | 30 |

**Step 3: Calculate SLI (Success Rate)**

Formula (in spreadsheet):

Success Rate = (Total Requests - Failed Requests) / Total Requests

Convert to % (e.g., 99.99%).

**Step 4: Compare Against SLO**

* Target = **99.9%**
* If Success Rate < 99.9%, then SLO violated on that day.

**Step 5: Track Error Budget**

Error budget = (100% - SLO Target) = **0.1% allowed failures per month**

1. Daily allowed failures = Total Requests × 0.1%.
2. Actual failures = Failed Requests.
3. Error Budget Used (%) = (Actual Failures / Allowed Failures) × 100.

**Step 6: Add Status Column**

If **Error Budget Used > 100%** → “Breach”  
Else → “Healthy”

**Step 7: Visualize**

Create two charts:

* **Line chart of Success Rate (%) per day** with SLO threshold line at 99.9%.
* **Bar chart of Error Budget consumption per day**.

This gives a clear reliability picture.

**🧮 Example Calculation (Day 5)**

* Total Requests = 105,000
* Failed Requests = 500
* Success Rate = (105,000 - 500) / 105,000 = **99.52%**
* Allowed Failures = 105,000 × 0.001 = 105
* Error Budget Used = (500 / 105) × 100 = **476%** (breach 🚨)

**💡 Reflective questions**

1. On which days was the SLO violated?
2. How fast is the error budget being consumed?
3. If this were real production data, would you:
   * Slow down releases?
   * Investigate root cause?
   * Alert stakeholders?

**🔬Lab Guide: Basic Monitoring and Alerting with Prometheus & Grafana**

**🎯 Learning Objectives**

By the end of this lab, trainees will be able to:

1. Deploy **Prometheus** to collect metrics from a service.
2. Deploy **Grafana** to visualize metrics.
3. Set up **basic alerts** for critical metrics.
4. Interpret monitoring data and simulate alert scenarios.

**🛠️ Prerequisites**

* Trainees should have:
  + **Linux/MacOS** system or VM
  + Docker installed (Docker Desktop is fine)
  + Basic familiarity with Linux CLI
* Optional: Sample service to monitor (e.g., a simple web server or microservice)

**Deploy the sample application : app.py**

from flask import Flask, Response

from prometheus\_client import Counter, Histogram, generate\_latest, CONTENT\_TYPE\_LATEST, CollectorRegistry

import time

app = Flask(\_\_name\_\_)

# Create a custom registry to export only our metrics

registry = CollectorRegistry()

# Metrics

REQUEST\_COUNT = Counter(

"request\_count\_total",

"Total number of HTTP requests",

["endpoint", "status"], # 👈 added "status" label

registry=registry

)

REQUEST\_LATENCY = Histogram(

"request\_latency\_seconds",

"Request latency in seconds",

["endpoint"],

registry=registry

)

# Home endpoint

@app.route("/")

def home():

with REQUEST\_LATENCY.labels(endpoint="/").time():

time.sleep(0.1) # Simulate proc. delay (~100ms) to make it realistic

REQUEST\_COUNT.labels(endpoint="/", status="200").inc() # 👈 updated

return "Hello from Raman App!"

# Dynamic endpoint /<name>

@app.route("/<name>")

def greet(name):

endpoint\_label = f"/{name}"

with REQUEST\_LATENCY.labels(endpoint=endpoint\_label).time():

time.sleep(0.1) # Simulate processing delay (~100ms)

REQUEST\_COUNT.labels(endpoint=endpoint\_label, status="200").inc() # 👈 updated

return f"Hello, {name}!"

# Failing endpoint to simulate errors

@app.route("/fail")

def fail():

REQUEST\_COUNT.labels(endpoint="/fail", status="500").inc() # 👈 new for failures

return "Internal Error", 500

# Metrics endpoint

@app.route("/metrics")

def metrics():

return Response(generate\_latest(registry), mimetype=CONTENT\_TYPE\_LATEST)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host="0.0.0.0", port=5000)

**requirements.txt**

Flask==2.3.3

prometheus-client==0.21.0

sudo apt update

sudo apt install python3.12-venv

python3 -m venv venv

**# 2. Activate it**

source venv/bin/activate

**# 3. Install dependencies inside the venv**

pip install -r requirements.txt

**Step 1: Launch Prometheus using Docker**

1. Create a **raman-prom.yml** file with basic configuration:

global:

scrape\_interval: 10s

scrape\_configs:

- job\_name: 'raman-service'

static\_configs:

- targets: ['host.docker.internal:5000']

**Create a shared network:**

Sudo docker network create raman-observability-net

**Run Prometheus**:

sudo docker run -d \

--name raman-prometheus \

--network raman-observability-net \

--add-host=host.docker.internal:host-gateway \

-v "$(pwd)/raman-prom.yml":/etc/prometheus/prometheus.yml \

-p 9090:9090 \

prom/prometheus

**Run Grafana**:

sudo docker run -d \

--name raman-grafana \

--network raman-observability-net \

-p 3000:3000 \

grafana/grafana

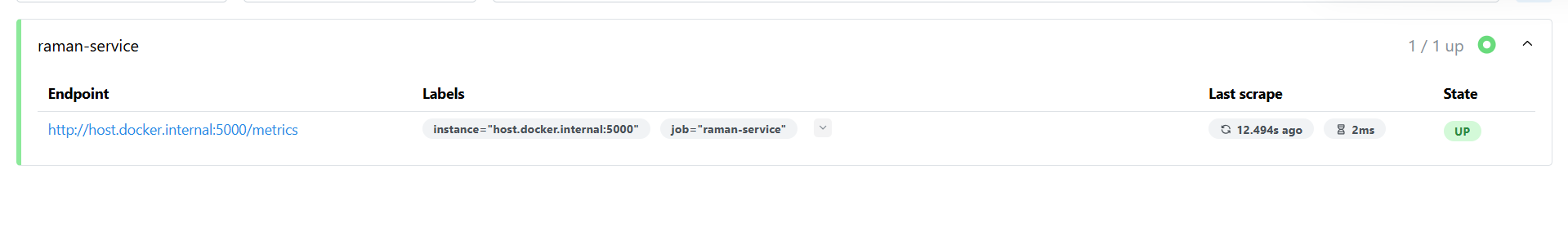
**Step 5 – Check Prometheus Target**

Go to:

http://<your-server-ip>:9090/targets

You should see:

raman-service – UP

****

**Configure Grafana**

1. Open Grafana:

http://<your-server-ip>:3000

(Default login: admin/admin)

1. Go to:

⚙️ → Data Sources → Add Data Source → Prometheus

Set URL:

http://raman-prometheus:9090

Click **Save & Test** → Should say “Data source is working”.

**Step 7 – Create Grafana Dashboard**

1. Go to:

Dashboards → New → New Dashboard

**Panel 1 – Total Requests**:

* Query:

request\_count\_total

* Title: **Total Requests**
* Visualization: Time series

**Panel 2 – Average Latency**:

* Query:

rate(request\_latency\_seconds\_sum[1m]) / rate(request\_latency\_seconds\_count[1m])

* Title: **Average Latency**
* Visualization: Time series

Save as:

raman-SRE-dashboard

