🧭 Understanding Google Cloud Monitoring (with Local Parallels)

This guide is designed to **translate a Google Cloud Monitoring Lab** into something meaningful — helping you understand:

- ✅ What each task in the lab *really does*
- 💡 Why it matters in real-world DevOps
- ⚙️ How to replicate it **locally** using free, open-source tools like Docker, Prometheus, Grafana, and Blackbox Exporter

---

🧩 Task 1: Creating Virtual Machines (VMs)

What you're doing:
Creating a Compute Engine instance (`instance2`) inside Project 2 — essentially a Linux server hosted by Google Cloud.

Compute engine: a virtual machine or bare metal server hosted on Google's infrastructure, providing on demand computing power to run apps and workload.

Why it matters:
Monitoring starts with something to observe. Having multiple instances lets you simulate different environments like `staging`, `testing`, and `production`.

Do this locally:
Using **Docker**, you can spin up two Linux-like containers:

```
docker run -d --name instance1 ubuntu sleep infinity

docker run -d --name instance2 ubuntu sleep infinity

sleep infinity: This is the command that runs inside the
container once it starts.
```

- `sleep is a Unix command that pauses for a specified amount of time.`
- `infinity is an argument telling it to sleep forever.`
- `This is a common technique used to keep a basic container running indefinitely. By default, a Docker container stops as soon as its main command finishes. Since sleep infinity never finishes, the container stays alive, allowing you to docker exec into it to run commands or use it for other purposes.`

Both containers now represent your "VMs."

🌐 Task 2: Setting Up a Monitoring Metrics Scope

What you're doing:
Creating a **Metrics Scope** that unifies metrics from multiple projects.

Why it matters:
Companies often have different GCP projects for different environments. Metrics Scopes give a *single dashboard* to monitor all.

Do this locally (Prometheus Federation):

```
scrape_configs:

  - job_name: 'project1'

    static_configs:

      - targets: ['localhost:9100']

  - job_name: 'project2'

    static_configs:

      - targets: ['localhost:9200']
```

This is like combining metrics from multiple Prometheus servers into one view.

👥 Task 3: Creating a Cloud Monitoring Group

What you're doing:
Defining a group (`DemoGroup`) that includes all VMs whose names contain "instance."

Why it matters:
Groups organize related resources for focused alerting or visualization — like grouping all frontend nodes.

Do this locally:
In **Grafana**, you can create a dashboard that filters instances using queries like:

```
up{job=~"instance.*"}
```

🩺 Task 4: Uptime Checks

What you're doing:
Configuring an automated **Uptime Check** to see if your instance's port 22 (SSH) is open.

Why it matters:
It's a fundamental "is this alive?" check for all services — helps detect global outages.

Do this locally (Blackbox Exporter):

```
modules:
  tcp_22:
    prober: tcp
    tcp:
      query_response:
```

Prometheus scrapes this exporter to verify if a target port or endpoint is reachable.

🚨 Task 5: Alerting Policies

What you're doing:
Creating an alert that triggers when the uptime check fails.

Why it matters:
Automation in alerting ensures downtime doesn't go unnoticed — critical for reliability.

Do this locally (Prometheus Rule):

```
groups:
  - name: example
    rules:
      - alert: InstanceDown
        expr: up == 0
        for: 1m
        labels:
          severity: critical
        annotations:
          summary: "Instance {{ $labels.instance }} is down"
```

How to see it:
When Prometheus detects a container stopped, this alert fires.
If connected to Alertmanager, it can notify via email, Slack, or Discord.

📊 Task 6: Custom Dashboards

What you're doing:
Building a dashboard in Cloud Monitoring to visualize uptime metrics.

Why it matters:
Dashboards help teams *see patterns and diagnose issues visually* — crucial for production observability.

Do this locally (Grafana):

1. Open Grafana (`http://localhost:3000`)
2. Add Prometheus as a data source.
3. Create a new dashboard → "Add Panel."

Use query:

```
up
```

4.
5. Select Line or Gauge visualization.

💣 Task 7: Causing an Incident (Simulating a Failure)

What you're doing:
Stopping your instance to intentionally fail an uptime check — generating an alert/incident.

Why it matters:
Testing failure conditions validates your monitoring system.
It's an early lesson in **chaos engineering** — breaking things to improve resilience.

Do this locally:

```
docker stop instance2
```

Prometheus marks it `up == 0`.
Restart to recover:

```
docker start instance2
```

🧠 Summary Table

Concept	Google Cloud Component	Local / Open Source Equivalent
Virtual Machines	Compute Engine	Docker / VirtualBox
Metrics Collection	Cloud Monitoring	Prometheus
Visualization	Dashboards	Grafana
Alerting	Alert Policy	Prometheus + Alertmanager
Uptime Check	Cloud Uptime	Blackbox Exporter
Incident Handling	Cloud Incidents	Alertmanager UI

🧰 Setting Up a Local Monitoring Stack (Full Hands-on)

Here's how to replicate **Google Cloud Monitoring** locally in one go, using Docker.

Step 1: Create a `docker-compose.yml` file

```
version: '3'

services:
  prometheus:
    image: prom/prometheus
    container_name: prometheus
```

```
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
    ports:
      - "9090:9090"

  grafana:
    image: grafana/grafana
    container_name: grafana
    ports:
      - "3000:3000"
    environment:
      - GF_SECURITY_ADMIN_USER=admin
      - GF_SECURITY_ADMIN_PASSWORD=admin

  blackbox:
    image: prom/blackbox-exporter
    container_name: blackbox
    ports:
      - "9115:9115"
    volumes:
      - ./blackbox.yml:/etc/blackbox_exporter/config.yml
```

Step 2: Create the Prometheus Config (`prometheus.yml`)

```yaml
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['prometheus:9090']

  - job_name: 'blackbox'
    metrics_path: /probe
    params:
      module: [tcp_22]
    static_configs:
      - targets:
        - instance1
        - instance2
    relabel_configs:
      - source_labels: [__address__]
        target_label: __param_target
      - target_label: instance
        replacement: blackbox
      - target_label: __address__
        replacement: blackbox:9115
```

Step 3: Create the Blackbox Config (`blackbox.yml`)

```
modules:
  tcp_22:
    prober: tcp
    timeout: 5s
```

Step 4: Launch Everything

Run:

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

Check:

- Prometheus → http://localhost:9090
- Grafana → http://localhost:3000
- Blackbox → http://localhost:9115

Step 5: Add Grafana Dashboard

1. Log in: `admin / admin`
2. Add data source → choose “Prometheus” → URL: `http://prometheus:9090`
3. Create Dashboard → Add Panel

Use queries like:

```
up
```

```
probe_success
```

4. These are your uptime metrics!

Step 6: Simulate a Failure

Stop one of your test containers:

```
docker stop instance2
```

Watch in:

- **Prometheus:** `up` value turns `0`
- **Grafana:** dashboard shows drop
- **Alerts:** trigger via Prometheus rule if configured

Restart:

```
docker start instance2
```

Within a minute, it clears — just like a closed incident in GCP.

🧩 Local Architecture Overview

```
[ Docker Containers ]
      ↓ metrics
[ Prometheus ] ← scrapes uptime
```

```
      ↓
[ Grafana ] ← visualizes metrics
      ↓
[ Alertmanager ] ← sends alerts (email/slack)
```

This replicates:

- GCP's Compute Engine → Docker containers
- Cloud Monitoring → Prometheus
- Cloud Dashboards → Grafana
- Uptime Checks → Blackbox Exporter
- Alert Policies → Prometheus alert rules
- Incidents → Alertmanager notifications

🎯 Final Takeaway

What you learned here mirrors **real-world SRE (Site Reliability Engineering)** concepts:

- Always **instrument** systems to emit metrics
- Build **visibility** using dashboards
- Automate **alerting** for failures
- Validate monitoring by **causing test incidents**

Cloud or local — the fundamentals are identical.
Once you grasp this, tools like GCP, AWS CloudWatch, or Azure Monitor all follow the same blueprint.