Tutorial: Flask Monte Carlo com Docker e Kubernetes

Me. Luis Vinicius Costa Silva

September 11, 2025

1 Objetivo

Este tutorial mostra como criar uma aplicação Flask que realiza simulação Monte Carlo, empacotá-la em Docker e rodá-la em Kubernetes local (Minikube ou cluster local). A aplicação terá endpoints:

- `/docker-info` - informações do container Docker
- `/montecarlo/<n>` - simulação Monte Carlo em CPU única
- `/montecarlo-distributed/<n>` - simulação distribuída em múltiplos pods

Atividade prática NF2

1. Fazer o **deploy da aplicação** usando Docker e Kubernetes.
2. Testar todos os endpoints:
   - `/docker-info` para verificar informações do container.
   - `/montecarlo/<n>` para testar a simulação em CPU única.
   - `/montecarlo-distributed/<n>` para testar a simulação distribuída em múltiplos pods.
3. Modificar o código para adicionar um novo endpoint que execute uma variação da simulação Monte Carlo (exemplo: estimativa de área de uma função ou outra forma geométrica).
4. Utilizar o script `montecarlo_aggregator.py` ou criar sua própria lógica para agregar resultados distribuídos.

Critérios de avaliação

- Correto deploy da aplicação em Docker e Kubernetes.
- Funcionamento correto de todos os endpoints.
- Capacidade de modificação do código para criar um novo endpoint funcional.
- Clareza e organização do código e testes.
- Uso correto das ferramentas de container e cluster (Docker, Minikube/Kubernetes).

Modificação no código

Criar um endpoint `/montecarlo-square/<n>` que estima a área de um quadrado inscrito em um círculo usando Monte Carlo.

Retornar tanto a estimativa da área quanto a proporção de pontos dentro da área esperada.

Garantir que o endpoint funcione tanto em single CPU quanto em modo distribuído nos pods.

2 Aplicação Flask

Crie o arquivo `app.py`:

```
import os
import socket
import random
from flask import Flask, jsonify

app = Flask(__name__)

# Fun   o Monte Carlo
def monte_carlo_pi(num_samples):
    inside = 0
    for _ in range(num_samples):
        x, y = random.random(), random.random()
        if x*x + y*y <= 1.0:
            inside += 1
    return (4.0 * inside) / num_samples

@app.route('/')
def index():
    return jsonify({
        "message": "Hello Flask + Kubernetes",
        "endpoints": [
            "/docker-info",
            "/montecarlo/<n>",
            "/montecarlo-distributed/<n>"
        ]
    })

@app.route('/docker-info')
def docker_info():
    return jsonify({
        "hostname": socket.gethostname(),
        "cwd": os.getcwd(),
        "env": dict(list(os.environ.items())[:10])
    })

@app.route('/montecarlo/<int:n>')
def montecarlo_single(n):
    pi_est = monte_carlo_pi(n)
    return jsonify({
        "samples": n,
        "pi_estimate": pi_est,
        "mode": "single-cpu"
    })
```

```
44
45 @app.route('/montecarlo-distributed/<int:n>')
46 def montecarlo_distributed(n):
47     replicas = int(os.getenv("POD_REPLICAS","1"))
48     pod_index = int(os.getenv("POD_INDEX","0"))
49     per_pod = n // replicas
50     pi_est = monte_carlo_pi(per_pod)
51     return jsonify({
52         "samples_total": n,
53         "samples_this_pod": per_pod,
54         "replicas": replicas,
55         "pod_index": pod_index,
56         "pi_partial": pi_est
57     })
58
59 if __name__ == '__main__':
60     app.run(host='0.0.0.0', port=8080, debug=True)
```

3 Docker

3.1 Dockerfile

Crie o arquivo Dockerfile:

```
1  FROM python:3.11-slim
2
3  WORKDIR /app
4
5  COPY requirements.txt requirements.txt
6  RUN pip install --no-cache-dir -r requirements.txt
7
8  COPY . .
9
10 EXPOSE 8080
11
12 CMD ["python", "app.py"]
```

3.2 requirements.txt

```
1 flask
```

3.3 Build e Run

```
1 docker build -t flask-montecarlo:latest .
2 docker run -p 8080:8080 flask-montecarlo
```

3.4 Testando endpoints

```
1 curl http://127.0.0.1:8080/docker-info
2 curl http://127.0.0.1:8080/montecarlo/1000000
```

4 Kubernetes

4.1 Deployment + Service

Crie o arquivo `kube-flask-montecarlo.yaml`:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flask-montecarlo-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: flask-montecarlo
  template:
    metadata:
      labels:
        app: flask-montecarlo
    spec:
      containers:
      - name: flask-montecarlo
        image: flask-montecarlo:latest
        imagePullPolicy: Never
        ports:
        - containerPort: 8080
        env:
        - name: POD_REPLICAS
          value: "3"
        - name: POD_INDEX
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
---
apiVersion: v1
kind: Service
metadata:
  name: flask-montecarlo-service
spec:
  selector:
    app: flask-montecarlo
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
  type: LoadBalancer
```

4.2 Aplicando no cluster

```
kubectl apply -f kube-flask-montecarlo.yaml
kubectl get pods
kubectl get svc
```

4.3 Acessando endpoints

- **Port-forward:**

```
kubectl port-forward service/flask-montecarlo-service 8080:8080
curl http://127.0.0.1:8080/montecarlo-distributed/1000000
```

- **Minikube service:**

```
minikube service flask-montecarlo-service --url
curl http://<URL>/montecarlo-distributed/1000000
```

5 Observações

- Cada pod calcula apenas sua parte da simulação distribuída.
- Para obter o valor final de π, é necessário agregar os resultados (ex: script Python externo).
- Em Docker local, você pode testar apenas ‘montecarlo‘ single CPU.