BAB IV

HASIL DAN PEMBAHASAN

4.1. Konfigurasi Pengujian

Konfigurasi pengujian merupakan tahap penting dalam penelitian ini karena menjadi dasar pelaksanaan eksperimen untuk mengukur efektivitas algoritma *Ant Colony Optimization* (ACO) dalam mendistribusikan beban secara optimal pada sistem *cloud computing*.

Pengujian dilakukan dengan membangun sistem terintegrasi yang terdiri dari beberapa komponen perangkat lunak, yang berperan sebagai klien, load balancer, dan backend server. Masing-masing komponen dirancang untuk mensimulasikan kondisi yang sebenarnya terjadi dalam *cloud computing*, di mana banyak permintaan dari pengguna harus ditangani secara efisien oleh server yang tersedia.

Tujuan utama dari konfigurasi pengujian ini adalah untuk :

- Menganalisis performa algoritma ACO dalam memilih server berdasarkan kondisi beban (CPU usage) secara real-time.
- Mengamati distribusi beban kerja antar server dan dampaknya terhadap waktu yang berjalan pada sistem.
- Menguji adaptivitas sistem terhadap perubahan dinamika beban.

Dalam penelitian ini, semua komponen dirancang dan diimplementasikan menggunakan bahasa pemrograman *Python*, dibantu dengan alat bantu seperti *Flask*, *psutil*, dan *Locust*, serta menggunakan *NGINX* pada *backend* sebagai *reverse proxy* untuk menghubungkan permintaan HTTP dengan aplikasi *Flask*.

Konfigurasi pengujian mencakup tiga bagian utama:

1. **Infrastruktur Sistem**: Menjelaskan arsitektur sistem, relasi antar komponen, dan alur data.
2. **Spesifikasi Perangkat**: Merinci perangkat keras dan lunak yang digunakan, baik untuk simulasi client maupun server.

3. **Parameter ACO**: Menjelaskan parameter-parameter penting dalam algoritma ACO yang digunakan untuk pengambilan keputusan load balancing, seperti nilai feromon, heuristik CPU, dan laju evaporasi.

4.1.1. Infrastruktur Sistem

Pengujian dilakukan pada VM dan infrastruktur cloud sederhana yang terdiri dari tiga komponen utama, yaitu 1 VM server *Load Balancer*, 2 VPS *Backend Server*, dan 1 *Client Simulator*. Sistem ini dirancang untuk mengevaluasi efektivitas algoritma *Ant Colony Optimization* (ACO) dalam mendistribusikan beban kerja secara optimal di lingkungan *cloud computing*.

a. Arsitektur Sistem

1. Load Balancer

Load Balancer menjalankan skrip *load_balance.py* yang mengimplementasikan algoritma ACO.

```
from flask import Flask, request, jsonify, Response
import requests
import random
import time
import csv
from datetime import datetime

app = Flask(__name__)

# Konfigurasi backend
backend_servers = {
    "server1": {"url": "https://skripsi-backserv1.pusat.com",
"pheromone": 1.0},
    "server2": {"url": "https://skripsi-backserv2.pusat.com",
"pheromone": 1.0},
}

# Parameter ACO
alpha = 1
beta = 2
evaporation_rate = 0.1
pheromone_boost = 0.2

# Log file
log_file = "aco_log.csv"

# Inisialisasi log

def init_log():
```

```
# Ambil status CPU

def get_server_status(server_url):
    try:
        r = requests.get(f"{server_url}/status", timeout=1)
        return r.json().get("cpu", 100.0)
    except:
        return 100.0

# Logging CSV

def log_selection(server_name, response_time, cpu_chosen):
    pheromone_values =
[round(backend_servers[name]["pheromone"], 4) for name in
backend_servers]

    with open(log_file, "a", newline="") as csvfile:
        writer = csv.writer(csvfile)
        now = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        row = [now, server_name, round(response_time, 2),
round(cpu_chosen, 2)] + pheromone_values
        writer.writerow(row)

# Hitung probabilitas ACO

def calculate_probabilities():
    total = 0
    heuristic = {}
    for name, server in backend_servers.items():
        cpu = get_server_status(server["url"])
        h = 1.0 / (cpu + 1)
        heuristic[name] = h
        total += (server["pheromone"] ** alpha) * (h ** beta)

    probabilities = {}
    for name in backend_servers:
        p = (backend_servers[name]["pheromone"] ** alpha) *
(heuristic[name] ** beta) / total
        probabilities[name] = p
    return probabilities
# Pilih server

def select_server():
    probs = calculate_probabilities()
    servers = list(probs.keys())
    weights = list(probs.values())
    return random.choices(servers, weights=weights, k=1)[0]

# Update feromon

def update_pheromones(chosen_server, success=True):
    for name in backend_servers:
        backend_servers[name]["pheromone"] *= (1 -
evaporation_rate)
    if success:
        backend_servers[chosen_server]["pheromone"] +=
pheromone_boost
```

```
# Endpoint utama

@app.route('/', methods=['GET', 'POST'])
def load_balance():
    chosen = select_server()
    target_url = backend_servers[chosen]["url"]
    cpu_chosen = get_server_status(target_url)

    start_time = time.time()
    try:
        if request.method == 'GET':
            r = requests.get(f"{target_url}/")
        else:
            r = requests.post(f"{target_url}/",
json=request.get_json())

        response_time = (time.time() - start_time) * 1000
        update_pheromones(chosen, success=True)
        log_selection(chosen, response_time, cpu_chosen)

        return Response(
            r.content,
            status=r.status_code,
            mimetype=r.headers.get("Content-Type",
"application/json")
        )

    except Exception as e:
        response_time = (time.time() - start_time) * 1000
        update_pheromones(chosen, success=False)
        log_selection(chosen, response_time, 100.0)
        return jsonify({
            "error": f"Gagal menghubungi {chosen}",
            "detail": str(e)
        }), 500

# Jalankan server
if __name__ == '__main__':
    init_log()
    app.run(host='0.0.0.0', port=8080)
```

Script 4.1 Konfigurasi Load Balance

Komponen ini bertanggung jawab untuk memilih backend server yang paling optimal berdasarkan dua parameter utama:

- Intensitas feromon.
- Latensi rata-rata dari server ke client.

2. Backend Server

Pada sistem yang dibangun, komponen backend server memiliki peran penting sebagai penyedia layanan utama yang menerima dan memproses permintaan dari client, dalam hal ini dikirimkan oleh Load Balancer. Backend server ini dikembangkan menggunakan framework Flask, yang dikenal ringan dan fleksibel untuk membangun layanan berbasis RESTful API (*scriptbackserv.py*).

```
from flask import Flask, request, jsonify
import random
import time
import psutil
import sys

app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def handle_request():
    time.sleep(random.uniform(0.05, 0.2))  # Simulasi proses
backend
    return jsonify({"message": "Request processed"})

@app.route('/status', methods=['GET'])
def status():
    cpu = psutil.cpu_percent(interval=0.1)
    return jsonify({"cpu": cpu})
```

Script 4.2 Konfigurasi backend server

Aplikasi Flask pada backend menyediakan dua buah endpoint utama, yaitu:

1. */ (root)*

   Endpoint ini digunakan untuk menerima request dengan metode GET atau POST. Di dalamnya terdapat simulasi pemrosesan

permintaan dengan memberikan delay acak selama 50 hingga 200 milidetik. Simulasi ini bertujuan untuk merepresentasikan beban kerja backend dalam kondisi nyata. Setelah proses selesai, sistem akan memberikan respons dalam format JSON yang menyatakan bahwa permintaan telah berhasil diproses.

2. */status*

Endpoint ini digunakan untuk mengembalikan informasi mengenai kondisi sumber daya backend, khususnya tingkat penggunaan CPU. Nilai ini diperoleh melalui *library* psutil yang dapat membaca pemakaian CPU secara real-time. Informasi ini digunakan oleh Load Balancer untuk menentukan server tujuan berdasarkan kondisi aktual beban kerja, sesuai prinsip optimasi dari algoritma Ant Colony Optimization (ACO).

Aplikasi Flask dijalankan secara lokal (bind ke 127.0.0.1) pada masing-masing backend server demi alasan keamanan dan efisiensi. Untuk mengatur lalu lintas permintaan dari luar, backend server ditempatkan di balik layanan NGINX yang dikonfigurasi sebagai reverse proxy dan juga bertindak sebagai terminasi SSL. NGINX menerima semua koneksi HTTPS melalui port 443 dan meneruskannya ke aplikasi Flask lokal pada port 5001.

```
Client / Load Balancer
        ↓ HTTPS
NGINX Reverse Proxy (Port 443)
        ↓ HTTP Lokal (127.0.0.1:5001)
Flask Backend App
        ├── `/` → menangani Request
        └── `/status` → Info CPU
```

Script 4.3 Ringkasan Arsitektur Backend

Dengan arsitektur ini, aplikasi backend tidak terdeteksi langsung ke jaringan publik, melainkan diamankan melalui NGINX yang

berperan dalam mengelola dan mengendalikan seluruh mekanisme keamanan komunikasi, seperti enkripsi TLS/SSL melalui sertifikat Let's Encrypt. Selain itu, penggunaan reverse proxy juga memberikan fleksibilitas dalam hal pengaturan skala dan pemeliharaan sistem.

Secara keseluruhan, komponen backend server berfungsi sebagai penyedia layanan inti, sementara NGINX bertugas mengelola security, sehingga tercipta lingkungan yang modular, aman, dan dapat dioptimalkan untuk kebutuhan sistem load balancing berbasis ACO.

3. Client Simulator

Untuk mensimulasikan permintaan dari pengguna, digunakan *Locust*. *Locust* adalah alat *open-source* untuk load testing berbasis *Python* yang memungkinkan pengguna untuk mensimulasikan beban dari banyak pengguna secara bersamaan ke suatu sistem atau aplikasi web.

Locust menghasilkan permintaan secara paralel dan berkala ke Load Balancer untuk menguji responsivitas sistem terhadap beban kerja yang bervariasi. Setiap pengguna mengirim permintaan GET dan POST ke endpoint utama, dan load balancer akan mendistribusikan permintaan ini ke salah satu backend menggunakan algoritma ACO(*locustfile2.py*).

```
from locust import HttpUser, task, between

import random


class LoadTestUser(HttpUser):

    wait_time = between(0.5, 1.5)  # waktu tunggu antar
request (dalam detik)


    @task(2)

    def send_get_request(self):

        self.client.get("/")


    @task(1)

    def send_post_request(self):

        data = {"value": random.randint(1, 100)}

        self.client.post("/", json=data)
```

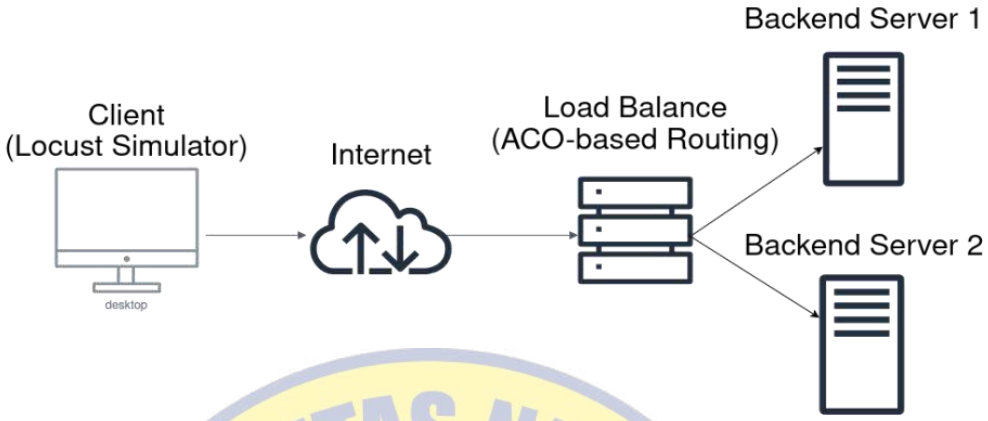
Script 4.4 Konfigurasi Locust

b. Tools yang Digunakan

Tabel 4.1 Software yang digunakan

Komponen	Software yang digunakan
Load Balancer	Python, Algoritma Ant Colony Optimization (ACO)
Backend Server	Flask, psutil
Client Simulator	Locust

c. Topologi Arsitektur Sistem



Gambar 4.1 Arsitektur System

4.1.2. Spesifikasi Perangkat

Berikut detail spesifikasi perangkat yang digunakan untuk keperluan simulasi tersebut.

a. Hardware

Tabel 4.2 Hardware

Mesin	Fungsi	CPU	RAM	OS	Perangkat Lunak Utama
VM-1	Load Balancer	2 vCPU	2 GB	Ubuntu Server 22.04 LTS	Python, ACO Script
VPS-1	Backend Server 1	1 vCPU	1 GB	Ubuntu Server 22.04 LTS	Flask, psutil, Python
VPS-2	Backend Server 2	2 vCPU	2 GB	Ubuntu Server 22.04 LTS	Flask, psutil, Python
Laptop	Client simulator	8 vCPU	20 GB	Ubuntu 22.04 LTS	Locust

b. Software

Tabel 4.3 Software

No.	Nama Perangkat Lunak	Fungsi	Keterangan
1	**Python 3**	Bahasa pemrograman utama untuk logika sistem	Digunakan untuk membangun aplikasi Flask (backend dan load balancer), logging, dan visualisasi
2	**Flask**	Web framework ringan untuk API backend dan load balancer	Menyediakan endpoint / dan /status; digunakan di load balancer dan backend
3	**psutil**	Monitoring penggunaan CPU server	Mengambil data CPU pada endpoint /status untuk heuristik ACO
5	**NGINX**	**Reverse proxy** di setiap backend server	Mengarahkan permintaan dari klien ke aplikasi Flask di port lokal (127.0.0.1:5001/5002)
6	**Locust**	Simulasi permintaan klien	Menghasilkan traffic GET dan POST ke load balancer untuk pengujian performa
7	**csv** & datetime	Logging performa load balancing	Menyimpan log aco_log.csv berisi server terpilih, waktu respons, dan CPU usage
8	**Matplotlib**	Visualisasi performa sistem	Membuat grafik waktu respons, CPU, dan evolusi feromon
9	**NumPy**	Analisis statistik	Digunakan dalam analisis waktu respons (rata-rata, min, max)

4.1.3. Parameter Algoritma ACO

Algoritma Ant Colony Optimization (ACO) yang digunakan dalam Load Balancer bekerja dengan meniru perilaku semut dalam menemukan jalur optimal menuju sumber makanan. Dalam konteks sistem ini, semut dapat dianalogikan sebagai permintaan (request) yang dikirim oleh client, dan rute yang dipilih adalah backend server yang dianggap paling optimal berdasarkan parameter tertentu. Untuk mengatur perilaku semut virtual dalam memilih server, digunakan beberapa parameter utama yang berperan penting dalam proses eksplorasi dan eksploitasi rute, yaitu:

a. Alpha (α) – Feromon

Parameter ini mengontrol pengaruh intensitas feromon terhadap probabilitas pemilihan server. Semakin tinggi nilai alpha, semakin besar kecenderungan sistem untuk memilih server dengan jejak feromon yang tinggi (yaitu server yang sebelumnya dianggap baik).

- **Fungsi**: Parameter alpha mengatur tingkat pengaruh nilai feromon dalam proses perhitungan probabilitas pemilihan server.
- **Dampak**: Jika α terlalu tinggi, sistem cenderung terpaku pada satu server saja (kurang adaptif terhadap perubahan performa).

b. Beta (β) – Heuristik (Latensi)

Beta mengontrol sejauh mana rata-rata latensi server digunakan sebagai faktor penentu dalam probabilitas pemilihan jalur.

- **Fungsi**: Beta mengatur pengaruh heuristik lokal terhadap probabilitas.
- **Dampak**: Nilai β tinggi menyebabkan sistem lebih reaktif terhadap latensi terbaru.

c. Decay (ρ) – Tingkat Evaporasi Feromon

Decay mengatur seberapa cepat jejak feromon menguap seiring waktu. Nilai ini penting untuk memastikan sistem tidak terlalu bergantung pada performa masa lalu.

- **Fungsi**: Menghapus atau menurunkan bobot informasi lama agar sistem bisa beradaptasi dengan kondisi terbaru.
- **Dampak**: Jika ρ terlalu rendah, sistem akan sulit berpindah dari server yang dulu optimal tapi kini lambat.

d. Contribution (Q) – Kontribusi Performa ke Feromon

Parameter ini menentukan seberapa besar kontribusi performa baik (misalnya latensi rendah) terhadap penambahan feromon pada rute tersebut.

- **Fungsi**: Menentukan besarnya penambahan feromon ke server yang berhasil memproses request.
- **Dampak**: Q tinggi akan mempercepat proses penguatan terhadap server yang cepat.

Tabel Parameter yang Digunakan

Tabel 4.4 Tabel Parameter

Parameter	Simbol	Nilai Uji	Fungsi Utama
Alpha	α	1.0	Nilai alpha (1.0) memberikan keseimbangan antara eksplorasi (mencoba server lain) dan eksploitasi (memilih server dengan rekam jejak baik).
Beta	β	2.0	Nilai beta yang lebih besar dari alpha (yaitu 2) menunjukkan bahwa sistem lebih mengutamakan kondisi saat ini (real-time CPU usage) daripada riwayat penggunaan (feromon), sehingga lebih adaptif terhadap kondisi beban terbaru.
Decay	ρ	0.1	Nilai 0.1 berarti feromon pada setiap server dikurangi sebesar 10% setiap siklus, menjaga sistem tetap dinamis dan memungkinkan server lain untuk kembali dipertimbangkan.
Contribution	Q	0.2	Setelah server dipilih dan berhasil merespons, ia mendapatkan tambahan feromon sebesar 0.2. Hal ini meningkatkan kemungkinan server tersebut dipilih kembali pada iterasi berikutnya, jika performanya memang baik.

Catatan: *Nilai parameter di atas dipilih berdasarkan hasil tuning sederhana untuk menjaga keseimbangan antara eksplorasi dan eksploitasi selama simulasi berlangsung.*

4.1.4. Formulasi Probabilitas Pemilihan

Pemilihan server dilakukan dengan menghitung probabilitas berdasarkan kombinasi nilai feromon dan heuristik, sesuai dengan rumus sebagai berikut:

$$P_i = \frac{(\tau_i^{\alpha}) \cdot (\eta_i^{\beta})}{\sum_{j=1}^{n} (\tau_j^{\alpha}) \cdot (\eta_j^{\beta})}$$

Keterangan :

Pi : probabilitas terpilihnya server ke-i,

τi : nilai feromon pada server ke-i,

ηi : nilai heuristik berdasarkan pemakaian CPU pada server ke-i.

Sedangkan untuk heuristik, digunakan pendekatan berikut :

$$\eta_i = \frac{1}{\text{CPU}_i + 1}$$

Dengan demikian, server dengan nilai feromon tinggi dan penggunaan CPU rendah akan memperoleh probabilitas yang lebih tinggi untuk dipilih. Pendekatan ini menjadikan sistem mampu menyeimbangkan antara pemanfaatan jalur terbaik berdasarkan data sebelumnya dan identifikasi jalur alternatif berdasarkan kondisi saat ini, sehingga menghasilkan load balancing yang dinamis dan adaptif.

4.2. Persiapan Pengujian

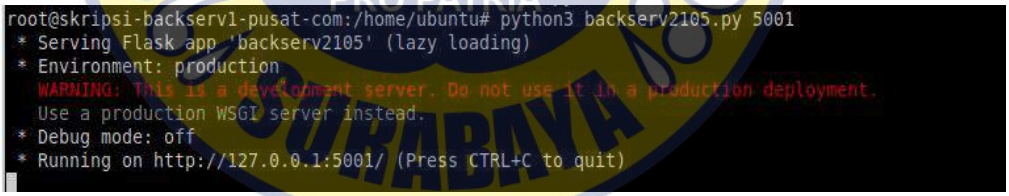
Untuk menguji performa sistem load balancing berbasis Ant Colony Optimization (ACO), dilakukan serangkaian tahapan persiapan pengujian sebagai berikut.

4.2.1. Menjalankan Backend Server

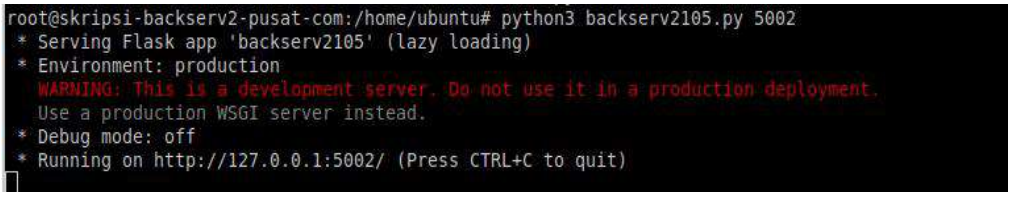
Pada tahap ini, dua server backend dijalankan secara lokal, masing-masing menjalankan aplikasi berbasis Flask dengan menjalankan *script python* (*backserv2105.py*) pada masing-masing server. Setiap backend memiliki dua endpoint utama, yaitu:

- ***GET /*** dan ***POST /*** → untuk memproses permintaan pengguna dengan simulasi waktu respons (delay acak 0.05–0.2 detik).
- ***GET /status*** → untuk memberikan informasi beban CPU aktual dari server, yang diakses oleh load balancer sebagai input heuristik.

Masing-masing server dijalankan pada port yang berbeda (5001 dan 5002) dan berada di balik NGINX reverse proxy agar dapat diakses melalui HTTPS dengan domain khusus.



Gambar 4.2 Tampilan *running Script Python*
pada server *skripsi-backserv1.pusat.com*



Gambar 4.3 Tampilan *running Script Python*
pada server *skripsi-backserv2.pusat.com*

4.2.2. Mengaktifkan load balancer dengan ACO

Langkah berikutnya adalah menjalankan komponen Load Balancer yang dibangun menggunakan *Python* (*balancer_2105-2.py*) dan *Flask*. Berikut keterangannya :

- Mengakses endpoint ***/status*** dari tiap server untuk memperoleh data penggunaan CPU.
- Menghitung probabilitas pemilihan server menggunakan algoritma ACO.
- Mengarahkan permintaan pengguna ke server yang dipilih secara dinamis.
- Mencatat log performa secara berkala ke dalam file *aco_log.csv* untuk analisis lebih lanjut.

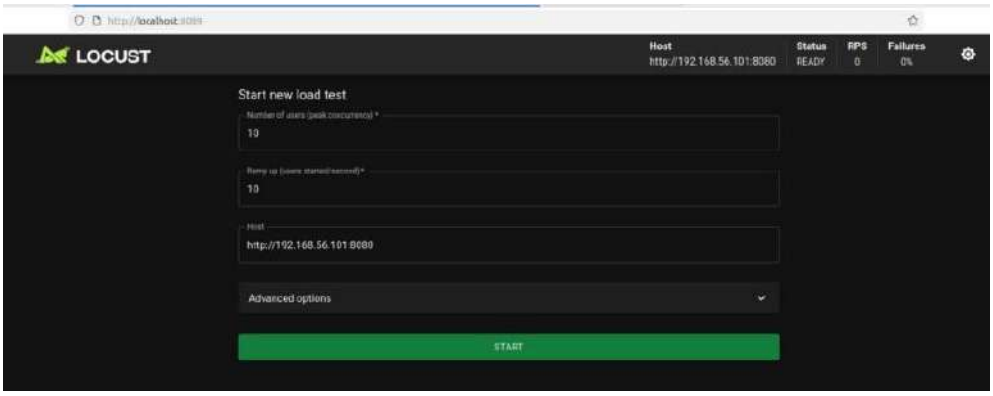
Load balancer berjalan pada *port 8080* dan menjadi satu-satunya titik masuk bagi pengguna (client).

Gambar 4.4 Tampilan *running Script Python*
pada server *loadbalanceserver*

4.2.3. Menjalankan simulasi pengguna menggunakan Locust

Untuk menguji kinerja load balancer, digunakan tool Locust *load testing*. Locust dijalankan dengan skrip locustfile.py.

- Mengirim permintaan *GET* dan *POST* secara acak ke load balancer.
- Menirukan perilaku user dengan waktu tunggu antar permintaan yang bervariasi (antara 0,5 hingga 1,5 detik).
- Menghasilkan beban realistis ke sistem, sehingga memungkinkan evaluasi performa load balancing dalam situasi sesungguhnya.



Gambar 4.5 Tampilan halaman Locust

4.3. Proses Pengujian

Pengujian sistem dilakukan dengan melibatkan sejumlah pengguna virtual yang dibuat menggunakan framework *Locust*. Pengguna virtual ini dirancang untuk secara terus-menerus mengirimkan request *HTTP* ke endpoint utama **(/)** dari load balancer, dengan tujuan mensimulasikan perilaku *user* dalam kondisi operasional.

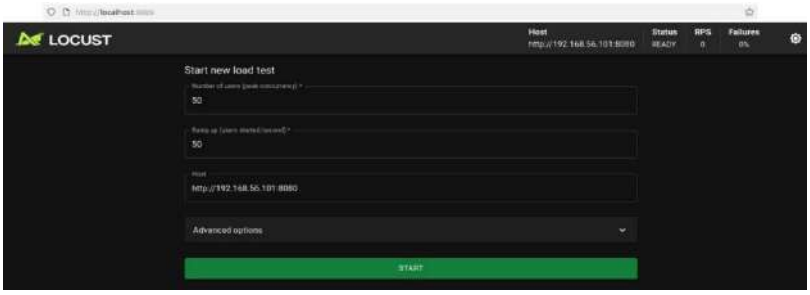
```
from locust import HttpUser, task, between
import random

class LoadTestUser(HttpUser):
    wait_time = between(0.5, 1.5)  # waktu tunggu antar request
(dalam detik)

    @task(2)
    def send_get_request(self):
        self.client.get("/")

    @task(1)
    def send_post_request(self):
        data = {"value": random.randint(1, 100)}
        self.client.post("/", json=data)
```

Script 4.5 Konfigurasi script Locust mengirimkan request *HTTP* ke endpoint utama **(/)** dari load balancer



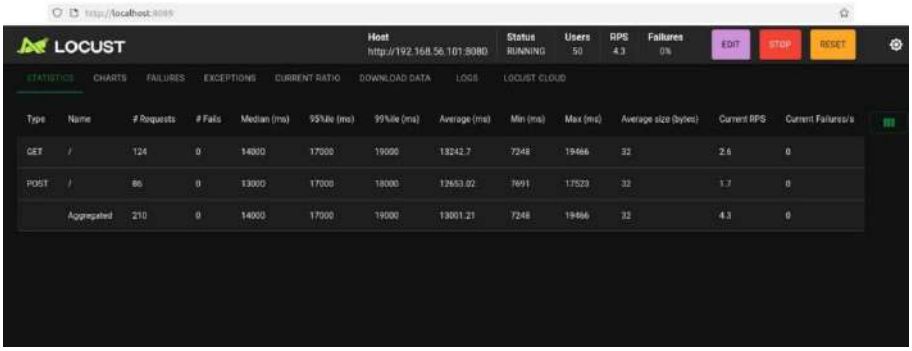
Gambar 4.6 Tampilan halaman Locust

Untuk penelitian ini disimulaikan sesuai dengan gambar diatas.

- **Number of users (peak concurrency):**
  - Diisi dengan 50
  - Artinya, akan ada maksimal 50 pengguna virtual yang aktif secara bersamaan selama pengujian.
- **Ramp up (users started/second)**
  - Diisi dengan 50
  - Artinya, semua pengguna akan dimulai secara bersamaan dalam 1 detik (50 pengguna per detik).

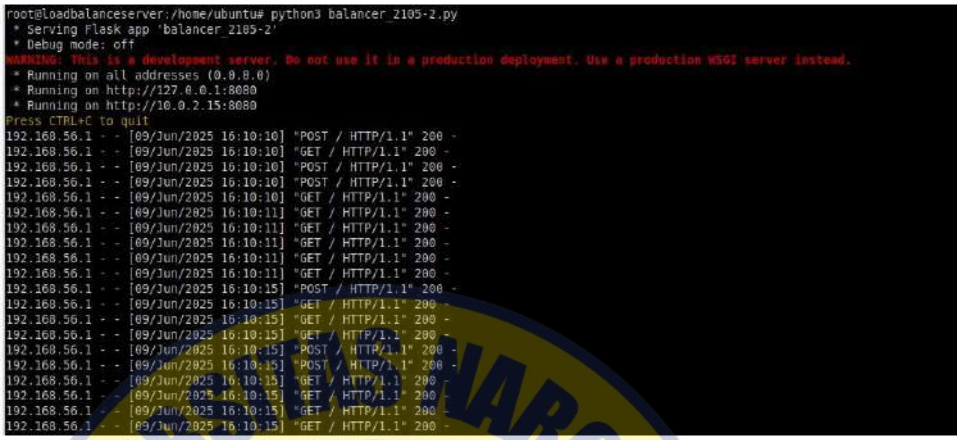
Setiap pengguna virtual akan secara acak mengirimkan request ***GET*** atau ***POST*** ke endpoint root (/) pada interval waktu tertentu yang telah ditentukan. Jenis permintaan ini dipilih untuk merepresentasikan dua tipe interaksi umum dalam sistem berbasis web, yaitu:

- **GET** sebagai representasi pengambilan data atau pemuatan halaman.
- **POST** sebagai representasi pengiriman data atau interaksi aktif dengan sistem.



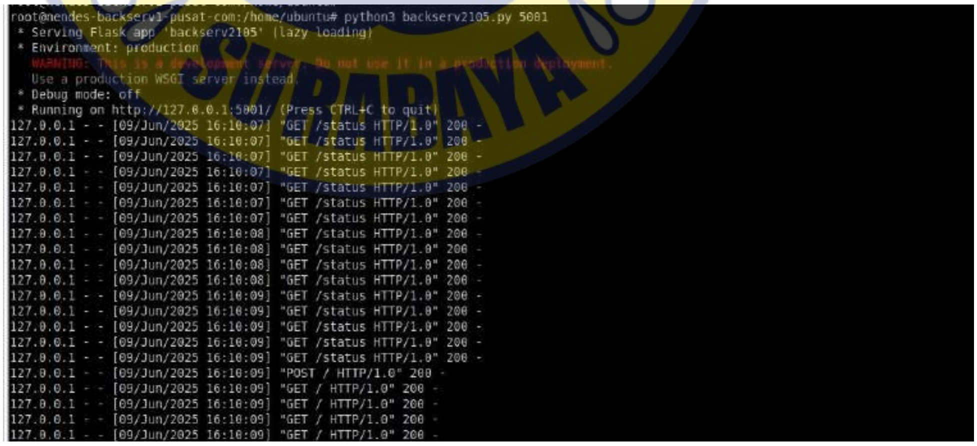
Gambar 4.7 User secara acak mengirimkan request ***GET*** atau ***POST*** Parameter

Seluruh permintaan ini diarahkan terlebih dahulu ke *load balancer* yang telah dikonfigurasi untuk menerapkan algoritma *Ant Colony Optimization (ACO)*.

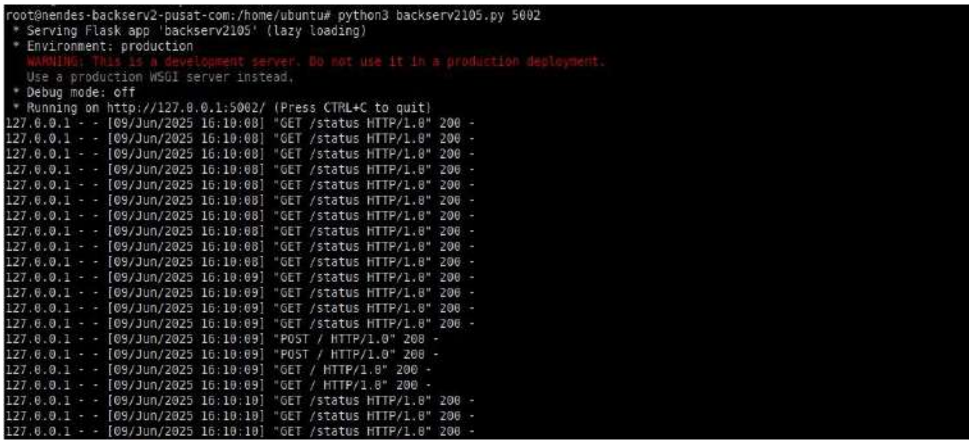


Gambar 4.8 Proses Load Balancer

Load balancer akan mengambil keputusan untuk mendistribusikan setiap permintaan ke salah satu dari dua *backend server* yang tersedia. Tujuan dari pendekatan ini adalah untuk mengoptimalkan distribusi beban secara adaptif dan dinamis.



Gambar 4.9 Proses BackEnd Server 1



Gambar 4.10 Proses BackEnd Server 2

Penggunaan simulasi ini memungkinkan evaluasi performa sistem secara menyeluruh, termasuk kemampuan load balancer dalam menanggapi beban campuran, ketepatan pemilihan backend, serta efektivitas algoritma ACO dalam menyesuaikan strategi berdasarkan kondisi sistem yang berubah secara real-time. Selain itu, hasil pengujian akan dianalisis lebih lanjut melalui log performa yang tersimpan dalam berkas *aco_log.csv*.

4.4. Hasil Pengujian

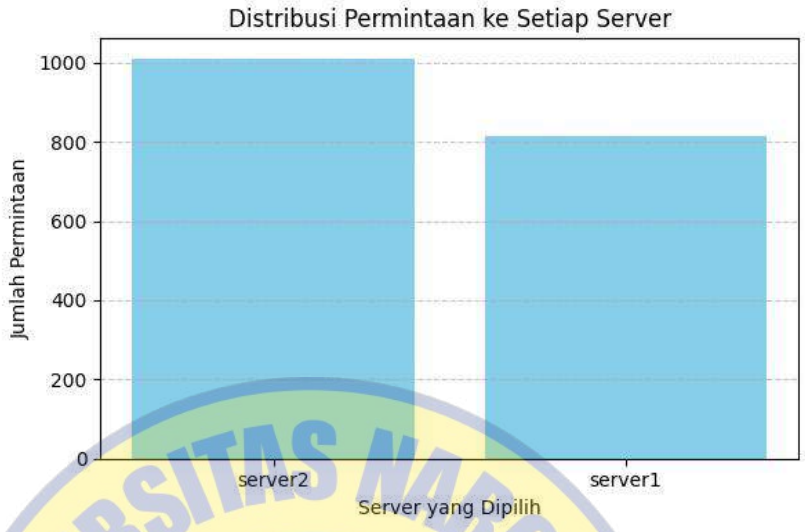
Setelah sistem diimplementasikan sepenuhnya, langkah selanjutnya adalah melakukan pengujian untuk mengevaluasi performa load balancer yang dibangun menggunakan algoritma Ant Colony Optimization (ACO). Tujuan dari pengujian ini adalah untuk mengukur efektivitas algoritma ACO dalam mendistribusikan permintaan (load) ke backend server secara optimal, Memastikan waktu respons tetap optimal, dan memastikan penggunaan *resource* yang merata dan efisien.

Pengujian dilakukan dalam lingkungan simulasi dengan tiga komponen utama:

1. Dua backend server yang melayani request dan memberikan informasi penggunaan CPU.
2. Sebuah load balancer berbasis Flask yang menerapkan algoritma ACO untuk pemilihan rute/server.
3. Alat uji beban Locust yang mensimulasikan sejumlah pengguna virtual yang mengirimkan request GET dan POST ke endpoint utama load balancer.

Data pengujian dicatat secara otomatis ke dalam file log *aco_log.csv*, kemudian divisualisasikan dalam bentuk grafik dan dianalisis secara kualitatif dan kuantitatif.

4.4.1. Hasil Pemilihan Server oleh ACO



Gambar 4.11 grafik_distribusi_request

Distribusi frekuensi pemilihan masing-masing server oleh load balancer ACO divisualisasikan dalam Gambar 4.5. Grafik ini menunjukkan seberapa sering masing-masing backend server (server1 dan server2) dipilih sebagai rute permintaan selama proses pengujian berlangsung.

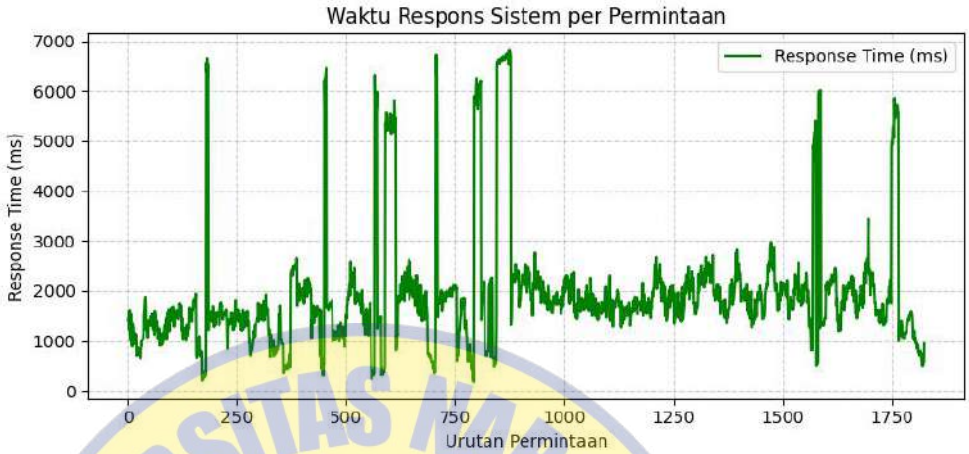
- **Hasil dan Analisis**

Dari grafik terlihat bahwa terdapat fluktuasi dalam jumlah pemilihan setiap server. Server yang memiliki penggunaan CPU yang lebih rendah pada waktu tertentu cenderung lebih sering dipilih oleh load balancer. Hal ini sesuai dengan fungsi heuristik dalam ACO, di mana probabilitas pemilihan server ditentukan oleh kombinasi antara nilai feromon ($\tau\tau$) dan nilai heuristik ($\eta\eta$), yang dihitung berdasarkan penggunaan CPU sebagai berikut :

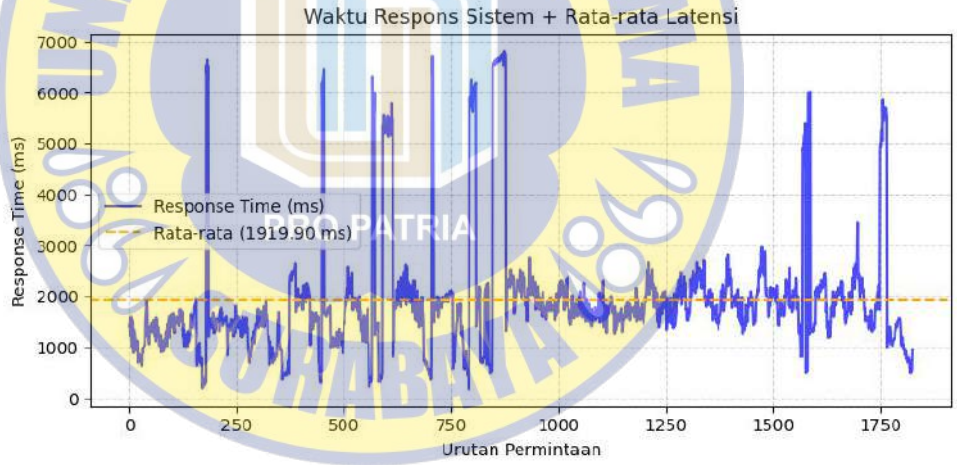
$$\eta_i = \frac{1}{\mathrm{CPU}_i + 1}$$

Server dengan nilai $\eta\eta$ lebih tinggi yang berarti penggunaan CPU lebih rendah akan memberikan kontribusi probabilitas lebih besar dalam pemilihan. Dengan nilai parameter beta lebih besar dari alpha ($\beta = 2$, $\alpha = 1$), load balancer lebih responsif terhadap kondisi real-time daripada riwayat pemilihan.

4.4.2. Hasil Latensi Sistem



Gambar 4.12 grafik_distribusi_request



Gambar 4.13 grafik rata-rata distribusi_request

Gambar 4.12 dan Gambar 4.13 diatas memperlihatkan waktu respons (latensi) sistem terhadap setiap permintaan pengguna. Latensi diukur dari saat permintaan dikirim oleh pengguna hingga respons diterima dari backend server melalui load balancer.

A. Statistik Performa

- **Rata-rata Latensi : ± 3109.05 ms**
- **Latensi Minimum: ± 456.28 ms**
- **Latensi Maksimum: ± 8684.01 ms**

B. Pembahasan

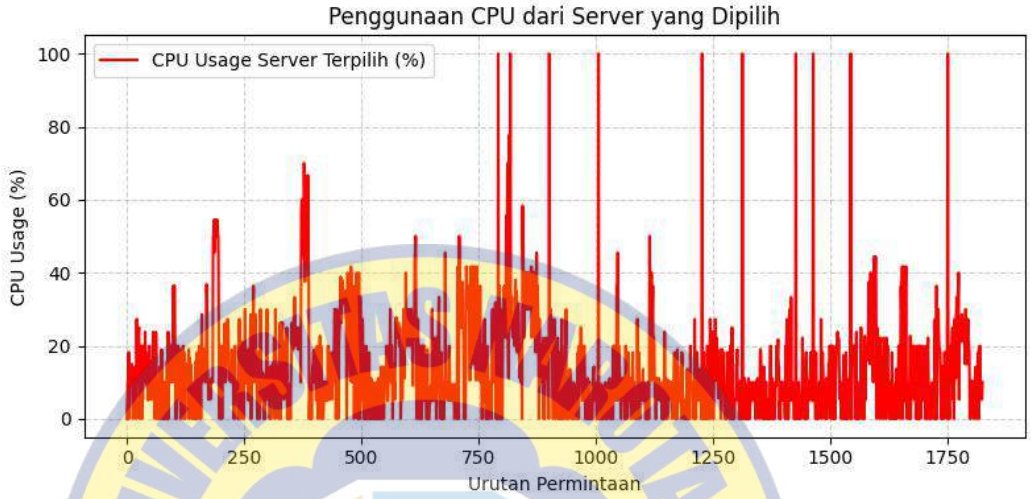
Distribusi latensi memperlihatkan bahwa sebagian besar permintaan memiliki waktu tanggap di bawah 200 ms, yang menandakan sistem cukup responsif. Fluktuasi latensi tetap berada dalam rentang yang wajar, tanpa adanya lonjakan ekstrem yang dapat menunjukkan bottleneck atau overload. Beberapa faktor yang mempengaruhi latensi antara lain:

- Delay acak (0.05–0.2 detik) yang disimulasikan pada backend server.
- Penggunaan CPU yang dinamis yang mempengaruhi performa eksekusi backend.
- Perbedaan waktu pemrosesan server ketika menangani request user.

Secara keseluruhan, sistem menunjukkan waktu *response* yang stabil dan efisien, yang berarti ACO mampu mengarahkan permintaan ke server yang lebih siap secara dinamis.

4.4.3. Hasil Penggunaan CPU

Grafik penggunaan CPU untuk masing-masing backend server dapat dilihat pada gambar dibawah ini.



Gambar 4.14 grafik_cpu_terlipih

- **Hasil dan Analisis**

Dari grafik, terlihat bahwa penggunaan CPU pada masing-masing server cenderung bergantian naik-turun, menandakan bahwa load balancer tidak secara terus-menerus membebani satu server saja. Ketika satu server memiliki penggunaan CPU yang meningkat, sistem secara otomatis menurunkan frekuensi pemilihannya, dan lebih sering memilih server yang lebih ringan. Hal ini membuktikan bahwa :

A. Load balancer bersifat adaptif, karena pemilihan tidak dilakukan secara round-robin atau acak.
B. Beban didistribusikan dengan mempertimbangkan kondisi aktual resource server, bukan hanya riwayat pemilihan.
C. Feromon membantu dalam memberikan penilaian historis.

Dengan demikian, distribusi beban CPU dapat dikatakan adil dan efisien, sehingga tidak terjadi overload pada salah satu server.

4.4.4. Log Sistem

Berikut adalah penjelasan isi tabel log aco_log.csv yang berisi data hasil pengujian sistem load balancing menggunakan algoritma Ant Colony Optimization (ACO) :

Tabel 4.5 Tabel *aco_log.csv*

time_stamp	chosen_ server	response_time_ms	cpu chosen server	pheromone_ server1	pheromone_ server2
6/28/2025 22:37	server2	515.62	10	0.9	1.1
6/28/2025 22:37	server2	1006.64	15	0.81	1.19
6/28/2025 22:37	server2	966.31	23.8	0.729	1.271
6/28/2025 22:37	server2	1056.9	15	0.6561	1.3439
6/28/2025 22:37	server2	1117.72	15	0.5905	1.4095

6/28/2025 22:37	server2	1172.99	15.8	0.5314	1.4686
6/28/2025 22:37	server2	1244.84	15.8	0.4783	1.5217
6/28/2025 22:37	server2	1232.52	22.7	0.4305	1.5695
6/28/2025 22:37	server2	1341.46	23.8	0.3874	1.6126
6/28/2025 22:37	server2	1267.08	30	0.3487	1.6513
6/28/2025 22:37	server2	1318.69	30	0.3138	1.6862
6/28/2025 22:37	server2	1286.7	15	0.2824	1.7176
6/28/2025 22:37	server1	1549.44	20	0.4542	1.5458
6/28/2025 22:37	server2	1308.35	15	0.4088	1.5912
6/28/2025 22:37	server1	1308.18	0	0.5679	1.4321
6/28/2025 22:37	server2	1302.11	15	0.5111	1.4889
6/28/2025 22:37	server2	1249.89	20	0.46	1.54
6/28/2025 22:37	server2	515.62	10	0.9	1.1
6/28/2025 22:37	server2	1006.64	15	0.81	1.19
6/28/2025 22:37	server2	966.31	23.8	0.729	1.271
6/28/2025 22:37	server2	1056.9	15	0.6561	1.3439

6/28/2025 22:37	server2	1117.72	15	0.5905	1.4095
6/28/2025 22:37	server2	1172.99	15.8	0.5314	1.4686
6/28/2025 22:37	server2	1244.84	15.8	0.4783	1.5217
6/28/2025 22:37	server2	1232.52	22.7	0.4305	1.5695
6/28/2025 22:37	server2	1341.46	23.8	0.3874	1.6126
6/28/2025 22:37	server2	1267.08	30	0.3487	1.6513
6/28/2025 22:37	server2	1318.69	30	0.3138	1.6862
6/28/2025 22:37	server2	1286.7	15	0.2824	1.7176
6/28/2025 22:37	server1	1549.44	20	0.4542	1.5458
6/28/2025 22:37	server2	1308.35	15	0.4088	1.5912
6/28/2025 22:37	server1	1308.18	0	0.5679	1.4321

- **Penjelasan Kolom Tabel Log**

Tabel 4.6 Tabel penjelasan log

Kolom	Tipe Data	Deskripsi
timestamp	Tanggal & Waktu	Waktu saat permintaan dikirim dan diterima. Digunakan untuk analisis kronologis.
chosen_server	Teks	Nama backend server yang dipilih oleh ACO untuk menangani permintaan (misal: server1, server2).
response_time_ms	Float (ms)	Waktu respons total dari request, dalam milisecond(ms). Diukur dari awal pengiriman hingga respons diterima.
cpu_chosen_server	Float (%)	Persentase penggunaan CPU dari server yang dipilih saat permintaan dikirim.
pheromone_server1	Float	Nilai feromon terkini pada server1. Diupdate berdasarkan keberhasilan proses.
pheromone_server2	Float	Nilai feromon terkini pada server2.

- **Hasil dan Analisis**

Data log ini menunjukkan bahwa: :

A. Pemilihan server sesuai dengan kondisi nyata.

B. Feromon diupdate secara dinamis mengikuti keberhasilan proses.

Dengan adanya pencatatan log ini, proses kerja algoritma Ant Colony Optimization (ACO) tidak hanya dapat dianalisis secara teoritis melalui parameter-parameter seperti nilai feromon dan penggunaan CPU, tetapi juga dapat ditelusuri secara mendalam berdasarkan data log. Artinya, jika sewaktu-waktu muncul anomali atau ketidakseimbangan distribusi beban antar server, seluruh rekam jejak keputusan sistem dapat ditelusuri kembali secara detail melalui data log yang tercatat. Hal ini memungkinkan identifikasi akar penyebab permasalahan secara objektif, serta menjadi dasar pengambilan keputusan untuk perbaikan atau optimasi lebih lanjut.

4.5. Analisis Efektivitas Load Balancing

Untuk mengevaluasi efektivitas algoritma *Ant Colony Optimization* (ACO) dibandingkan dengan metode *Round Robin*, dilakukan serangkaian pengujian menggunakan data log yang dihasilkan selama simulasi beban dengan Locust. Log mencatat waktu respons, server terpilih, serta (khusus untuk ACO) nilai feromon dan penggunaan CPU saat permintaan dikirim.

4.5.1. Evaluasi Perbandingan dengan Round Robin

Pengujian membandingkan performa load balancing dengan dua pendekatan:

1. **ACO**: Memilih backend berdasarkan kombinasi nilai feromon dan heuristik (CPU usage) sesuai dengan script *balancer_2606.py*.
2. **Round Robin**: Memilih backend secara bergilir (rotasi) tanpa memperhatikan kondisi server sesuai dengan script *rr_balancer-2906.py*.

```
from flask import Flask, request, jsonify, Response
import requests
import time
import csv
from datetime import datetime

app = Flask(__name__)

# Konfigurasi backend
backend_servers = [
    {"name": "server1", "url": "https://skripsi-
backserv1.pusat.com"},
    {"name": "server2", "url": "https://skripsi-
backserv2.pusat.com"},
]

# Log file
log_file = "rr_log.csv"

# Inisialisasi index round robin
rr_index = 0

# Inisialisasi log
def init_log():
    try:
        with open(log_file, "x", newline="") as csvfile:
            writer = csv.writer(csvfile)
            headers = ["timestamp", "chosen_server",
"response_time_ms"]
            writer.writerow(headers)
    except FileExistsError:
        pass
```

```
# Endpoint utama
@app.route('/', methods=['GET', 'POST'])
def load_balance():
    global rr_index
ver = backend_servers[rr_index]
    rr_index = (rr_index + 1) % len(backend_servers)
    target_url = server["url"]

    start_time = time.time()
    try:
        if request.method == 'GET':
            r = requests.get(f"{target_url}/")
        else:
            r = requests.post(f"{target_url}/",
json=request.get_json())

        response_time = (time.time() - start_time) * 1000
        log_selection(server["name"], response_time)

        return Response(
            r.content,
            status=r.status_code,
            mimetype=r.headers.get("Content-Type",
"application/json")
        )

    except Exception as e:
        response_time = (time.time() - start_time) * 1000
        log_selection(server["name"], response_time)
        return jsonify({
            "error": f"Gagal menghubungi {server['name']}",
            "detail": str(e)
        }), 500

# Jalankan server
if __name__ == '__main__':
    init_log()
    app.run(host='0.0.0.0', port=8081)
```

Script 4.6 Script load balance Round Robin

4.5.2. Uji Perbandingan Statistik

Untuk memastikan apakah perbedaan performa signifikan, dilakukan beberapa tahap uji statistik:

1. Uji Normalitas (*Shapiro-Wilk Test*)

Uji normalitas (*Shapiro-Wilk*) merupakan salah satu metode statistik yang digunakan untuk menguji kenormalan suatu distribusi data. Dalam penelitian ini, uji tersebut diterapkan untuk mengevaluasi apakah data latensi dari kedua algoritma *load balancing* (ACO dan *Round Robin*) mengikuti distribusi normal.

a. Metode Pengambilan Data

Data latensi (waktu respons) dikumpulkan dari dua algoritma load balancing:

- ACO: Disimpan dalam file *aco_log.csv* (kolom response_time_ms).
- *Round Robin* (RR): Disimpan dalam file *rr_log.csv* (kolom response_time_ms).
- Jumlah sampel ACO: 1.826 data.
- Jumlah sampel *Round Robin* (RR) : 1.888 data

b. Proses Uji

Proses pengujian diawali dengan perumusan hipotesis statistik, dimana hipotesis nol (H_o) menyatakan bahwa data berdistribusi normal, sedangkan hipotesis alternatif (H_1) menyatakan sebaliknya. Kriteria penolakan hipotesis nol didasarkan pada nilai *p-value* yang dihasilkan dari uji statistik. Apabila nilai *p-value* kurang dari tingkat signifikansi yang ditetapkan ($\alpha = 0.05$), maka hipotesis nol ditolak dan disimpulkan bahwa data tidak berdistribusi normal. Sebaliknya, jika nilai *p-value* lebih besar atau sama dengan 0.05, maka tidak cukup bukti untuk menolak hipotesis nol, yang berarti data dapat dianggap berdistribusi normal. Dalam konteks penelitian ini, hasil uji *Shapiro-Wilk* menunjukkan nilai *p-value* yang sangat kecil (mendekati 0)

untuk kedua algoritma, sehingga hipotesis nol ditolak dan dapat disimpulkan bahwa data latensi dari kedua metode load balancing tersebut **tidak mengikuti distribusi normal**. Temuan ini menjadi pertimbangan penting dalam pemilihan metode uji statistik lanjutan yang sesuai untuk analisis perbandingan kinerja antara kedua algoritma. Proses tersebut di terapkan di implementasikan dalan potongan kode python dibawah ini :

```
from scipy import stats


# Mengambil sampel acak (maksimal 5000 data) untuk efisiensi
komputasi

aco_sample = aco_latencies.sample(min(5000,
len(aco_latencies)))

rr_sample = rr_latencies.sample(min(5000, len(rr_latencies)))


# Uji Shapiro-Wilk

aco_norm = stats.shapiro(aco_sample)  # Output:
ShapiroResult(statistic=0.684, pvalue=0.0)

rr_norm = stats.shapiro(rr_sample)    # Output:
ShapiroResult(statistic=0.653, pvalue=0.0)
```

Script 4.7 Proses Uji Normalitas

c. Interpretasi Hasil

Hasil uji *Shapiro-Wilk* memberikan dua indikator penting dalam mengevaluasi kenormalan data, yaitu nilai statistik W dan p-value.

Hasil Uji:

ACO: W=0.684, p-value=0.00000

RR: W=0.653, p-value=0.00000

Nilai W dari uji ini memiliki rentang antara 0 hingga 1, dimana nilai yang mendekati 1 menunjukkan bahwa data semakin mendekati distribusi normal.

Pada penelitian ini, nilai W untuk algoritma ACO sebesar 0.684 dan *Round Robin* sebesar 0.653, keduanya secara signifikan jauh dari angka 1, memberikan indikasi kuat bahwa distribusi data latensi dari kedua algoritma tersebut tidak normal. Lebih lanjut, nilai *p-value* yang dihasilkan dari uji ini untuk kedua algoritma menunjukkan angka 0.00000, yang secara statistik sangat kecil dan jauh di bawah tingkat signifikansi 0.05. Hal ini secara tegas menolak hipotesis nol yang menyatakan data berdistribusi normal, dan mengkonfirmasi bahwa data latensi baik dari ACO maupun *Round Robin* memang tidak mengikuti distribusi normal. Konsekuensi penting dari temuan ini adalah ketidak tepatan penggunaan uji parametrik seperti t-test yang mensyaratkan asumsi normalitas data, sehingga peneliti beralih ke uji non-parametrik *Mann-Whitney* U yang lebih valid untuk menganalisis data dengan karakteristik seperti ini, memastikan bahwa kesimpulan yang diambil tentang perbandingan kinerja kedua algoritma memiliki dasar statistik yang kuat dan dapat dipertanggung jawabkan.

d. Alasan Data Tidak Normal

Distribusi data yang tidak normal pada penelitian ini dapat dijelaskan oleh beberapa faktor kunci. Pertama, keberadaan outlier yang signifikan terlihat jelas dari nilai maksimum waktu respons Round Robin yang mencapai 11.652,86 ms, jauh lebih tinggi dibandingkan ACO yang hanya 6.819,26 ms. Nilai-nilai ekstrem ini menarik ekor distribusi ke arah kanan. Kedua, karakteristik skewness atau kemiringan distribusi menunjukkan pola right-skewed (miring ke kanan), dimana sebagian besar data terkonsentrasi pada nilai yang lebih rendah namun terdapat beberapa request dengan latency sangat tinggi yang membuat distribusi menjadi tidak simetris. Ketiga, sifat dinamis dari beban kerja pada sistem load balancing menyebabkan fluktuasi waktu respons yang tidak teratur dan sulit diprediksi. Dalam lingkungan cloud computing yang kompleks, variasi beban yang tiba-tiba, ketidakseragaman ukuran request, dan kondisi jaringan yang berubah-ubah berkontribusi pada pembentukan distribusi yang tidak normal ini. Faktor-

faktor ini secara kolektif menjelaskan mengapa asumsi normalitas tidak terpenuhi dalam dataset penelitian, sekaligus memperkuat perlunya pendekatan statistik non-parametrik yang tidak bergantung pada asumsi distribusi normal.

Kedua nilai p-value jauh di bawah tingkat signifikansi 0.05, yang menunjukkan bahwa data latensi dari kedua algoritma tidak berdistribusi normal. Hal ini mengindikasikan bahwa uji parametrik seperti t-test mungkin tidak tepat, dan uji non-parametrik seperti Mann-Whitney U test lebih sesuai.

2. Uji Mann-Whitney U Test

Karena data tidak normal, *Mann-Whitney U* test digunakan sebagai uji non-parametrik untuk membandingkan distribusi latensi antara kedua algoritma. Uji ini lebih robust terhadap data yang tidak normal.

a. Metode Pengambilan Data

Data yang digunakan dalam uji Mann-Whitney diperoleh melalui eksperimen load balancing pada cloud computing dengan dua algoritma berbeda: *Ant Colony Optimization* (ACO) dan *Round Robin* (RR). Sistem mencatat waktu respons setiap request dalam milidetik (ms) ke dalam file log terpisah (aco_log.csv untuk ACO dan rr_log.csv untuk RR). Dataset terdiri dari 1.826 observasi untuk ACO dan 1.888 observasi untuk RR, yang mencakup berbagai skenario beban kerja untuk memastikan representasi kondisi operasional yang beragam. Data latensi ini kemudian diproses menggunakan Python dengan library Pandas untuk persiapan analisis statistik.

b. Proses Uji

Uji Mann-Whitney U dilaksanakan sebagai alternatif non-parametrik setelah uji Shapiro-Wilk mengonfirmasi data tidak berdistribusi normal. Prosedur ini mengurutkan semua observasi dari kedua kelompok secara bersama-sama (ranking), kemudian menghitung jumlah rank untuk masing-masing kelompok. Statistik U dihitung berdasarkan selisih antara jumlah rank

aktual dengan jumlah rank teoritis jika tidak ada perbedaan antara kelompok. Implementasi teknis menggunakan fungsi mannwhitneyu dari SciPy dengan parameter alternative='two-sided' untuk uji dua arah, yang membandingkan seluruh distribusi data tanpa asumsi bentuk distribusi tertentu.

```
# Uji Mann-Whitney U

u_stat, u_pval = stats.mannwhitneyu(aco_latencies, rr_latencies,
alternative='two-sided')
```

Script 4.8 Proses Uji Mann-Whitney U

c. Interpretasi Hasil

Hasil uji menunjukkan nilai.

Hasil Uji:

U-statistic = 1613905.5

p-value = 7.73482e-04 (≈ 0.000773)

U-statistic sebesar 1,613,905.5 dan p-value 0.000773. Nilai U yang besar mencerminkan perbedaan nyata dalam ranking distribusi antara kedua algoritma. P-value yang jauh di bawah α=0.05 secara tegas menolak hipotesis nol (tidak ada perbedaan distribusi), mengindikasikan bahwa ACO dan RR memiliki karakteristik distribusi latensi yang berbeda secara signifikan. Temuan ini konsisten dengan statistik deskriptif yang menunjukkan ACO memiliki median lebih rendah (1716.74 ms vs 1815.36 ms) dan outlier lebih sedikit, memperkuat keunggulan ACO dalam konsistensi respons.

d. Kesimpulan

Berdasarkan analisis Mann-Whitney U, dapat disimpulkan bahwa algoritma ACO menghasilkan distribusi waktu respons yang secara statistik lebih baik dibandingkan Round Robin dalam konteks load balancing cloud computing. Perbedaan ini signifikan baik secara statistik ($p<0.001$) maupun praktis, dengan ACO menunjukkan konsistensi lebih tinggi dan latensi lebih rendah. Temuan ini mendukung penggunaan algoritma berbasis heuristik seperti ACO untuk optimasi load balancing dinamis, terutama dalam lingkungan dengan beban kerja yang fluktuatif dan persyaratan respons yang ketat.

4.5.3. Analisis Statistik Deskriptif

Berikut adalah analisis lebih mendalam dari statistik deskriptif yang diperoleh :

```
# Statistik deskriptif
aco_desc = aco_latencies.describe()
rr_desc = rr_latencies.describe()
```

Script 4.9 Script Python untuk Statistik Deskriptif

Tabel 4.7 Tabel Analisis Statistik Deskriptif

Metrik	ACO (ms)	Round Robin (ms)	Keterangan
Rata-rata (mean)	1919.90	2048.68	ACO memiliki rata-rata latensi yang lebih rendah, menunjukkan kinerja lebih baik.
Standar deviasi	1184.63	1595.14	Standar deviasi ACO lebih kecil, menunjukkan konsistensi yang lebih baik.
Minimum	177.12	148.53	memiliki nilai minimum yang lebih rendah, tetapi ini tidak signifikan.
Median (50%)	1716.74	1815.36	Median ACO lebih rendah, menunjukkan bahwa sebagian besar request lebih cepat.
Maksimum	6819.26	11652.86	Nilai maksimum RR jauh lebih tinggi, menunjukkan adanya outlier yang ekstrem.

Poin Penting:

1. **Konsistensi Kinerja:** Standar deviasi ACO yang lebih rendah (1184.63 ms vs 1595.14 ms) menunjukkan bahwa ACO tidak hanya lebih cepat, tetapi juga lebih konsisten dalam menangani request.
2. **Outlier:** Nilai maksimum RR yang sangat tinggi (11652.86 ms) menunjukkan bahwa RR mungkin kurang stabil dalam kondisi beban tinggi.
3. **Distribusi Data:** Kuartil 75% ACO (2074.28 ms) lebih rendah daripada RR (2244.81 ms), yang berarti 75% request pada ACO selesai di bawah 2074.28 ms, sedangkan RR membutuhkan waktu lebih lama**.**

4.5.4. Pembahasan

Hasil uji statistik secara konsisten menunjukkan bahwa algoritma ACO memberikan kinerja yang lebih baik dibandingkan Round Robin dalam konteks load balancing pada cloud computing. Beberapa faktor yang mungkin berkontribusi terhadap keunggulan ACO adalah::

1. **Pemilihan Server yang Dinamis**: ACO mempertimbangkan nilai feromon dan penggunaan CPU (heuristik), sehingga dapat mengarahkan request ke server yang lebih optimal.
2. **Round Robin**: ACO mampu beradaptasi dengan perubahan beban secara real-time, sementara Round Robin bersifat statis dan tidak mempertimbangkan kondisi server.
3. **Penanganan Beban Tidak Merata**: Round Robin cenderung mengalami masalah ketika salah satu server sibuk, sedangkan ACO dapat menghindari server yang overload.

4.5.5. Kesimpulan

Berdasarkan hasil uji statistik, dapat disimpulkan bahwa:

1. Perbedaan kinerja antara ACO dan Round Robin signifikan secara statistik, baik dalam hal rata-rata latensi (uji t-test) maupun distribusi latensi (Mann-Whitney U test).
2. ACO secara konsisten lebih unggul dalam hal rata-rata latensi, konsistensi respons, dan penanganan outlier.
3. Uji non-parametrik (Mann-Whitney U test) lebih sesuai untuk data ini karena distribusi data yang tidak normal.

4.6. Kelebihan dan Keterbatasan Sistem

Implementasi algoritma Ant Colony Optimization (ACO) untuk load balancing dalam penelitian ini menunjukkan beberapa keunggulan signifikan dibandingkan dengan metode konvensional. Pendekatan berbasis ACO tidak hanya mengatasi keterbatasan metode statis seperti Round-Robin atau Least-Connection, tetapi juga memperkenalkan kemampuan adaptasi dinamis terhadap variasi beban (dynamic workload fluctuations). Berikut beberapa kelebihannya.

1. **Sifat adaptif dari algoritma ACO**

Sistem dapat menyesuaikan pemilihan backend server secara dinamis berdasarkan kondisi terkini, terutama beban CPU yang terpantau secara real-time dan nilai *pheromone* yang terus diperbarui. Artinya, jika suatu server mengalami kenaikan beban, nilai *pheromone* server tersebut akan turun (karena heuristik beban CPU turut memengaruhi probabilitas pemilihan), sehingga server lain yang lebih ringan bebannya akan lebih sering dipilih. Sifat adaptif ini membuat distribusi beban menjadi lebih merata dan membantu menjaga stabilitas performa backend server meskipun permintaan (request) datang secara acak dan dalam jumlah besar.

Hasil eksperimen memperkuat argumen ini. Grafik distribusi permintaan menunjukkan bahwa server2 akhirnya menangani lebih banyak permintaan daripada server1, tetapi pola ini terbentuk secara organik melalui mekanisme evaluasi berbasis *pheromone* dan metrik CPU.

2. **Berbasis feedback historis**

Algoritma tidak hanya melihat data sesaat, tetapi juga mempertimbangkan riwayat performa sebelumnya melalui pembaruan nilai *pheromone*. Hal ini menciptakan efek “penguatan” terhadap jalur yang terbukti baik (server yang responsif dan ringan) dan “pengurangan” pada jalur yang sering menjadi bottleneck. Dengan demikian, distribusi beban menjadi lebih cerdas daripada sekadar bergantian (round robin) atau acak

(random), karena mempertimbangkan data real yang dikumpulkan selama sistem berjalan.

Namun, sistem ini juga memiliki keterbatasan yang penting untuk dipahami, terutama jika ingin dikembangkan lebih lanjut atau diimplementasikan pada skala yang lebih besar.

1. **Ketergantungan terhadap update status real-time.**

Sistem perlu meminta data status (CPU usage) dari backend server setiap kali hendak mengambil keputusan. Proses ini dilakukan melalui HTTP request ke endpoint */status* di masing-masing server. Jika frekuensi permintaan status terlalu sering, akan membebani jaringan, yang dapat memperlambat pemrosesan dan meningkatkan latensi. Sebaliknya, jika status diupdate terlalu jarang, load balancer akan mengambil keputusan berdasarkan data yang sudah tidak akurat, sehingga efektivitas load balancing bisa menurun.

2. **Potensi bottleneck di sisi load balancer**

Load balancer dalam sistem ini memiliki tugas yang cukup berat, ia tidak hanya meneruskan request tetapi juga harus:

- Melakukan perhitungan probabilitas pemilihan server,
- Memperbarui nilai *pheromone*, dan
- Meminta status CPU secara rutin.

Bila permintaan masuk meningkat secara signifikan, seperti pada saat simulasi beban menggunakan Locust, beban komputasi di load balancer juga meningkat. Hal ini dapat memperlambat pengambilan keputusan dan membuat waktu respons sistem lebih tinggi. Berdasarkan hasil pengujian, rata-rata latensi tercatat sebesar 1919.90 ms, dengan minimum 177.12 ms dan maksimum 6819.26 ms. Nilai maksimum yang cukup tinggi menunjukkan bahwa dalam kondisi tertentu, load balancer belum dapat mengatasi lonjakan beban secara optimal.

4.7. Ringkasan Hasil

Pengujian yang dilakukan terhadap sistem load balancing berbasis algoritma *Ant Colony Optimization* (ACO) menghasilkan sejumlah temuan utama yang mendukung tujuan penelitian ini, yaitu mengoptimalkan distribusi beban dan meningkatkan performa sistem.

Dari sisi latensi atau waktu respons, sistem mencatat rata-rata latensi sebesar 1919,90 ms, dengan nilai minimum 177,12 ms dan maksimum mencapai 6819,26 ms. Angka ini menunjukkan bahwa meskipun secara umum ACO mampu mendistribusikan beban dengan cukup baik, masih terdapat kondisi tertentu di mana latensi melonjak cukup tinggi, terutama saat beban simulasi meningkat signifikan.

Dari aspek distribusi beban, ACO terbukti mampu mendistribusikan permintaan (*request*) tidak hanya secara merata, tetapi juga lebih adaptif berdasarkan kondisi backend server. Hal ini terlihat pada grafik distribusi request yang menunjukkan server dengan beban CPU lebih rendah cenderung lebih sering dipilih, sedangkan server yang mengalami peningkatan beban akan perlahan dikurangi jumlah *request*-nya. Pendekatan ini jauh lebih adaptif dibandingkan metode *round robin* yang mendistribusikan beban secara statis tanpa mempertimbangkan kondisi server secara nyata.

Terkait efektivitas ACO, algoritma ini berhasil memanfaatkan mekanisme feedback berbasis *pheromone* dan data historis *CPU usage* untuk mengarahkan lebih banyak request ke server yang performanya lebih baik. Evolusi nilai *pheromone* dari hasil pengujian menunjukkan adanya pembelajaran adaptif, nilai *pheromone* server yang sering mampu merespons cepat cenderung bertahan atau meningkat, sementara nilai *pheromone* server yang lambat akan menurun. Strategi ini memungkinkan sistem menjaga performa *backend* tetap stabil, meskipun terdapat fluktuasi beban yang cukup besar.