TRƯỜNG ĐẠI HỌC CÔNG NGHỆ
KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO CÔNG CỤ KIỂM THỬ LOCUST

MÔN HỌC: KIỂM THỬ VÀ ĐẢM BẢO CHẤT LƯỢNG PHẦN MỀM

Giảng viên: ThS. Nguyễn Thu Trang

Sinh viên 1: Nguyễn Xuân Thịnh – 23020709

Sinh viên 2: Hoàng Duy Thịnh – 23020708

Sinh viên 3: Lê Duy Khánh Toàn – 23020702

Sinh viên 4: Đặng Đình Khang – 23020709

HÀ NỘI - 07/2025

Mục lục

1. Tổng quan

Trong quá trình phát triển phần mềm, kiểm thử phần mềm là một quá trình quan trọng trong việc phát triển phần mềm, nhằm đảm bảo rằng phần mềm hoạt động đúng theo yêu cầu và không có lỗi. Quá trình này bao gồm việc thực hiện các hoạt động kiểm tra để phát hiện và sửa chữa các lỗi trong phần mềm trước khi sản phẩm được phát hành đến người dùng cuối.

Công cụ kiểm thử phần mềm chính là trợ thủ đắc lực cho các nhà phát triển trong việc tự động hóa, tối ưu hóa quá trình kiểm tra, tiết kiệm thời gian, chi phí và nâng cao hiệu quả. Công cụ kiểm thử giảm thiểu khả năng xảy ra lỗi do con người, đảm bảo kết quả kiểm thử nhất quán và chính xác hơn; có thể chi phí đầu tư ban đầu cho công cụ kiểm thử cao, nhưng về lâu dài, nó giúp giảm chi phí phát sinh do lỗi phần mềm và các sự cố liên quan.

Trong bài báo cáo này, nhóm em tập trung vào công cụ kiểm thử Locust-công cụ kiểm thử hiệu năng được viết bằng Python.

1.1 Định nghĩa kiểm thử

Kiểm thử phần mềm là quá trình thực hiện các hành động có kế hoạch và có hệ thống để kiểm tra một phần mềm, nhằm xác định chất lượng của phần mềm đó. Mục tiêu của kiểm thử phần mềm là phát hiện lỗi, đảm bảo chất lượng và xác nhận rằng phần mềm đáp ứng các yêu cầu đã đặt ra.

1.2 Vai trò của kiểm thử

Kiểm thử phần mềm đóng vai trò quan trọng trong việc đánh giá chất lượng và là hoạt động chủ chốt trong việc đảm bảo chất lượng cao của sản phẩm phần mềm trong quá trình phát triển. Thông qua chu trình "kiểm thử -tìm lỗi - sửa lỗi" ta hy vọng chất lượng của sản phẩm phần mềm sẽ được cải tiến. Mặt khác, thông qua việc tiến hành kiểm thử mức hệ thống trước khi cho lưu hành sản phẩm, ta biết được sản phẩm của ta tốt ở mức nào. Vì thế, nhiều tác giả đã mô tả việc kiểm thử phần mềm là một quy trình kiểm chứng để đánh giá và tăng cường chất lượng

của sản phẩm phần mềm. [1] (N. H. Pham, A. H. Truong, and V. H. Dang, in GIÁO TRÌNH KIỂM THỬ PHẦN MỀM, 2014.)

1.3 Một số thuật ngữ liên quan

2. Công cụ kiểm thử Locust

2.1 Tổng quan

Locust là một công cụ kiểm thử tải mã nguồn mở được viết bằng Python, dùng để đánh giá **hiệu năng và khả năng chịu tải của hệ thống**. Theo *tài liệu chính thức của Locust*, công cụ này cho phép người dùng mô phỏng hàng nghìn hoặc hàng triệu người dùng ảo truy cập vào website hoặc API để kiểm tra phản ứng của hệ thống dưới các mức tải khác nhau.

Điểm đặc biệt của Locust là cho phép **viết kịch bản kiểm thử (test scenario)** bằng Python, nhờ đó người kiểm thử có thể mô phỏng hành vi thực tế của người dùng, chẳng hạn như: truy cập trang chủ, đăng nhập, tìm kiếm sản phẩm, thêm vào giỏ hàng hoặc gửi yêu cầu API.

Khác với nhiều công cụ truyền thống như JMeter hoặc LoadRunner, Locust **nhẹ, linh hoạt và dễ mở rộng**, đồng thời có **giao diện web trực quan** để theo dõi kết quả kiểm thử theo thời gian thực. Ngoài ra, nó hỗ trợ **chế độ phân tán (distributed mode)**, cho phép chạy nhiều worker trên nhiều máy để kiểm thử quy mô lớn.

2.2 Các tính năng

Theo tài liệu chính thức và cộng đồng người dùng Locust, công cụ này sở hữu các nhóm tính năng nổi bật sau:

2.2.1 Mô phỏng người dùng thực tế

- Các hành vi của người dùng được định nghĩa thông qua các class kế thừa từ HttpUser hoặc User.
- Cho phép sử dụng hàm @task để xác định hành vi cụ thể và trọng số cho từng tác vụ.
- Mỗi người dùng ảo thực thi các hành vi này theo chu kỳ, mô phỏng hoạt động thực tế của người thật.

2.2.2 Kiểm thử tải và hiệu năng

- Đo **thời gian phản hồi (response time)**, **số lượng request/giây**, và **tỷ lệ lỗi**.
- Giúp phát hiện điểm nghẽn (bottleneck) của hệ thống khi số lượng người dùng tăng dần.
- Hỗ trợ giới hạn tốc độ tải hoặc đặt số lượng người dùng tăng theo thời gian (spawn_rate).

2.2.3 Giao diện web trực quan

- Giao diện tại địa chỉ http://localhost:8089 giúp điều chỉnh số người dùng, tốc độ sinh người dùng, và quan sát kết quả kiểm thử theo thời gian thực.
- Hiển thị thống kê như trung bình thời gian phản hồi, phần trăm lỗi, độ lệch chuẩn, và phân phối phần trăm phản hồi (percentiles).

2.2.4 Chạy phân tán

- Cho phép chạy theo mô hình **Master-Worker**, trong đó Master điều phối và tổng hợp kết quả từ nhiều Worker.
- Hữu ích khi cần kiểm thử hệ thống quy mô lớn với hàng trăm nghìn người dùng ảo.

2.2.5 Tích hợp và mở rộng

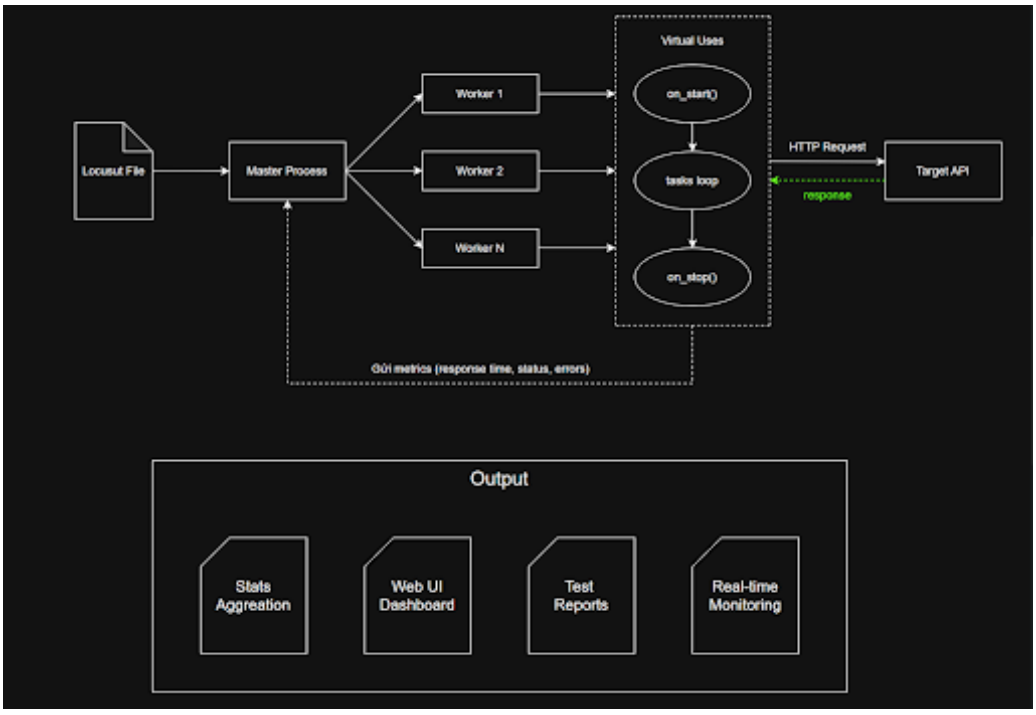
- Dễ dàng tích hợp vào pipeline CI/CD (Jenkins, GitHub Actions, GitLab CI).
- Có thể ghi log, xuất kết quả sang CSV hoặc gửi đến hệ thống giám sát như Grafana, Prometheus, InfluxDB.
- Hỗ trợ kiểm thử không chỉ HTTP mà còn WebSocket, MQTT, gRPC, và các giao thức tuỳ chỉnh thông qua lớp User.

2.3 Mục tiêu

Các mục tiêu chính của việc sử dụng Locust trong kiểm thử phần mềm bao gồm:

- **Đánh giá hiệu năng hệ thống:** Xác định khả năng chịu tải của hệ thống, tốc độ phản hồi và độ ổn định khi có nhiều người dùng đồng thời.
- **Phát hiện điểm nghẽn (bottlenecks):** Xác định thành phần gây ra độ trễ, như cơ sở dữ liệu, API chậm, hoặc quá tải CPU.
- **Tối ưu hóa tài nguyên:** Dựa trên kết quả kiểm thử, nhóm phát triển có thể tối ưu kiến trúc, cân bằng tải, hoặc nâng cấp phần cứng.
- **Đảm bảo độ tin cậy:** Kiểm thử khả năng phục hồi (resilience) và hành vi hệ thống trong các tình huống cực đoan.
- **Tự động hóa kiểm thử:** Tích hợp Locust vào quy trình CI/CD giúp đảm bảo hệ thống luôn duy trì hiệu năng ổn định qua các bản phát hành.

2.4 Cách thức hoạt động



Hình 2.1: Quy trình cho 1 ca kiểm thử với công cụ Locust

Khởi tạo: Locust đọc file cấu hình (`locustfile.py`) chứa các lớp `User` và `task` định nghĩa hành vi kiểm thử. `Master process` được khởi động và tạo

giao diện Web UI trên cổng 8089.

Kiến trúc Master-Worker: `Master process` điều phối và phân phối công việc đến các `worker processes`. Mỗi `worker` chạy độc lập và quản lý một nhóm người dùng ảo thông qua **greenlets (gevent)**, cho phép xử lý đồng thời hàng nghìn kết nối với overhead thấp.

- **Greenlet (gevent):** Lightweight coroutine (tiến trình nhẹ) cho phép xử lý đồng thời hàng nghìn kết nối trên một thread mà không tốn nhiều bộ nhớ, tạo ra hiệu suất cao hơn so với multithreading truyền thống.
- **Mô hình Greenlet (gevent):** Sử dụng coroutines để mô phỏng nhiều người dùng đồng thời trên một thread duy nhất. Khi một `greenlet` chờ response từ API, `gevent` tự động chuyển sang `greenlet` khác, tối ưu hóa việc sử dụng CPU.

Virtual User: Người dùng ảo được mô phỏng bởi `greenlet` để tạo ra tải truy cập đồng thời vào hệ thống, mỗi `user` thực hiện các `task` theo kịch bản định sẵn.

- **Vòng đời của mỗi *virtual user***: Mỗi virtual user trải qua 3 giai đoạn: khởi tạo với phương thức `on_start()`, thực thi vòng lặp tasks liên tục để gửi HTTP requests đến API đích, và kết thúc với `on_stop()` khi test hoàn tất.
- **Tasks Loop**: Vòng lặp thực thi liên tục các tác vụ (HTTP requests) để mô phỏng hành vi người dùng thực tế, bao gồm các hành động như đăng nhập, truy vấn dữ liệu, gửi form.

Thu thập và tổng hợp số liệu: Mỗi worker gửi *metrics* về master process thông qua message queue. Master tổng hợp dữ liệu từ tất cả workers thành module *Stats Aggregation*, tính toán các chỉ số như:

- **request count** (số lượng requests),
- **response times** (thời gian phản hồi),
- **failures** (lỗi),
- **RPS** (requests per second),
- **percentiles** (phân vị: P50, P95, P99).

Hiển thị kết quả: Master cung cấp 4 hình thức *output* chính:

- **Web UI Dashboard**: Giao diện web hiển thị biểu đồ thời gian thực (*real-time charts*), bảng thống kê chi tiết (*statistics table*), bảng điều khiển (*control panel*) để `start`/`stop` test, và khả năng tải xuống báo cáo.
- **Test Reports**: Các báo cáo chi tiết dạng *HTML report* và *CSV statistics*, bao gồm phân bố thời gian phản hồi (*response time distribution*) và nhật ký lỗi (*failure logs*).
- **Real-time Monitoring**: Giám sát trực tiếp số người dùng đang hoạt động (*active users*), RPS hiện tại (*current RPS*), tỷ lệ lỗi (*error rate*), và xu hướng thời gian phản hồi (*response time trends*).
- **Stats Aggregation**: Tổng hợp tất cả số liệu thống kê để phân tích tổng quan hiệu suất hệ thống.

2.5 Chi tiết hoạt động của LOCUST

2.5.1 Định nghĩa người dùng và tác vụ

Bắt đầu từ cốt lõi: mô tả "người dùng ảo" sẽ làm gì. Khi hành vi được khai báo rõ, mọi thứ tiếp theo (dữ liệu, kịch bản, tải) mới có nền tảng để triển khai.

- **Lớp người dùng ảo (`HttpUser`)**: Locust mô phỏng người dùng thông qua việc định nghĩa các lớp Python kế thừa `HttpUser` (hoặc User nếu không dùng HTTP). HttpUser cung cấp client HTTP tích hợp sẵn để gửi requests đến hệ thống đích. Mỗi instance của HttpUser đại diện cho một người dùng ảo độc lập với session riêng, thực hiện một tập hành vi cụ thể trên hệ thống.
- @task **decorator – Định nghĩa tác vụ bằng @task**: Bên trong lớp HttpUser, các hành vi (scenario) của người dùng được định nghĩa dưới dạng các phương thức được đánh dấu bằng decorator @task. Mỗi phương thức được trang trí bởi @task được coi là một *tác vụ* mà người dùng ảo sẽ thực hiện. Locust sẽ tạo nhiều instance của lớp người dùng và mỗi instance chọn ngẫu nhiên một tác vụ để chạy tại mỗi bước.
- **Trọng số task (task weight)**: Có thể gán trọng số cho tác vụ bằng cách truyền tham số vào @task, ví dụ @task(3). Tác vụ có trọng số cao sẽ được chọn thực thi thường xuyên hơn (ví dụ trọng số 3 nghĩa là xác suất chạy tác vụ đó cao gấp ~3 lần so với trọng số 1). Bên trong phương thức tác vụ, sử dụng self.client (HTTP client tích hợp) để gửi các yêu cầu HTTP (GET, POST, ...); Locust sẽ tự động thu thập thời gian thực thi và kết quả (status code, độ trễ, ...) của các request này.

- wait_time: Định nghĩa thời gian chờ giữa các task để mô phỏng hành vi người dùng thực. Có thể sử dụng constant (cố định), between (ngẫu nhiên trong khoảng), hoặc constant_pacing (đảm bảo tần suất đều).
- **Hooks on_start/on_stop**: Có thể định nghĩa các phương thức đặc biệt on_start(self) và on_stop(self) trong lớp người dùng. on_start sẽ được Locust gọi **một lần** khi mỗi user ảo bắt đầu phiên chạy của nó – thường dùng để thực hiện bước khởi tạo, ví dụ đăng nhập hoặc thiết lập dữ liệu ban đầu. Tương tự, on_stop được gọi khi người dùng ảo kết thúc (khi test dừng hoặc người dùng bị loại bỏ), thường dùng để dọn dẹp, ví dụ đăng xuất hoặc giải phóng tài nguyên. Các hook này giúp mô phỏng chính xác hơn vòng đời của một phiên người dùng.
- **Stateful flow**: Cho phép duy trì trạng thái qua các request như session cookies, authentication tokens, hoặc dữ liệu context. Điều này quan trọng để mô phỏng luồng giao dịch liên tiếp của người dùng thực.

Đoạn code Python dưới đây định nghĩa một lớp người dùng ảo với hai tác vụ đơn giản, sử dụng trọng số, thời gian chờ và hook `on_start`/`on_stop`:

```
        from locust import HttpUser, task, between

        class WebsiteUser(HttpUser):
        wait_time = between(1, 5)

        def on_start(self):
        self.client.post("/login", json={"username": "test", "password": "
            test"})

        def on_stop(self):
        self.client.post("/logout")

        @task(3)
        def view_home(self):
        self.client.get("/")

        @task(1)
        def view_profile(self):
        self.client.get("/profile")
```

2.5.2 Chuẩn bị và tạo dữ liệu

Sau khi biết user ảo sẽ làm gì, ta cần chuẩn bị "dao cụ". Giai đoạn này nhằm tạo ra dữ liệu, tài khoản và môi trường để các hành vi được thực thi ổn định và tái lặp.

- **Chuẩn bị dữ liệu mẫu trước khi test:** Trước khi chạy tải bằng Locust, cần đảm bảo hệ thống có sẵn dữ liệu phù hợp. Nên có bộ dữ liệu thử riêng (fixtures) để phục vụ kiểm thử hiệu năng, ví dụ: tập tài khoản người dùng thử, danh sách sản phẩm mẫu, hoặc bản định sẵn cơ sở dữ liệu nhỏ phục vụ các thao tác cố định.

Tạo 100 user thử (với curl + shell):

```
for i in $(seq 1 100); do
curl -X POST https://api.example.com/admin/seed-user \
-H "Authorization: Bearer <ADMIN_TOKEN>" \
-H "Content-Type: application/json" \
-d '{"username":"test'$i'","password":"Pass@123","role":"customer"}
    '
done
```

Seed 1.000 sản phẩm mẫu:

```
curl -X POST https://api.example.com/admin/seed-products \
-H "Authorization: Bearer <ADMIN_TOKEN>" \
-H "Content-Type: application/json" \
--data-binary @products_seed.json
```

- **Sử dụng fixture và khởi tạo dữ liệu trong code:** Có thể nhúng dữ liệu mẫu ngay trong script Locust hoặc đọc từ file bên ngoài. Ví dụ: lưu danh sách tài khoản thử trong file CSV rồi đọc vào một biến toàn cục. Mỗi người dùng ảo khi `on_start` có thể lấy ngẫu nhiên một cặp username/password từ danh sách này để đăng nhập. Đảm bảo mỗi user ảo độc lập về dữ liệu (ví dụ mỗi user dùng một bộ định danh riêng) để kết quả test không bị sai lệch do các user tác động lẫn nhau.
- **Dữ liệu đăng nhập và người dùng thử:** Nếu kịch bản hiệu năng bao gồm bước đăng nhập, hãy tạo sẵn danh sách thông tin đăng nhập cho nhiều người dùng thử. Mỗi user ảo trong Locust nên sử dụng một tài khoản riêng (hoặc một tập hợp các tài khoản – *pool credentials*) để mô phỏng chân thực

việc nhiều người dùng đồng thời đăng nhập. Tránh việc tất cả user ảo dùng chung một thông tin đăng nhập, vì có thể dẫn đến tranh chấp (ví dụ: ghi đè *session* của nhau) hoặc không đúng hành vi thực tế. Ví dụ dưới đây minh họa việc: Đọc CSV và cấp phát tuần tự (round-robin) cho mỗi VU:

Đọc CSV và khởi tạo từ tệp:

```
import csv, itertools
from locust import HttpUser, task

with open("users.csv", newline="") as f:
USERS = list(csv.DictReader(f))  # cot: username,password
_pool = itertools.cycle(USERS)

class LoginUser(HttpUser):
def on_start(self):
creds = next(_pool)
self.username = creds["username"]
self.client.post("/login", json=creds)

def on_stop(self):
self.client.post("/logout", json={"username": self.username})

@task
def home(self):
self.client.get("/")
```

- **Seed dữ liệu và teardown:** Trong trường hợp kịch bản cần một lượng lớn dữ liệu (ví dụ hàng vạn bản ghi đơn hàng để truy vấn), nên thực hiện bước *seed* (tạo dữ liệu sẵn) trước khi chạy Locust, thay vì để user ảo tự tạo trong lúc chạy (điều này có thể làm nhiễu kết quả và tốn thời gian test). Sau khi kết thúc test, thực hiện *teardown* – tức là xóa hoặc dọn dẹp các dữ liệu thử đã sinh ra (ví dụ: xóa những đơn hàng test, người dùng test khỏi database) để môi trường trở lại trạng thái sạch cho lần chạy sau. Việc này giúp tránh *ô nhiễm môi trường test* (pollution) và đảm bảo mỗi lần test đều bắt đầu trên nền dữ liệu như nhau.

Sau khi Locust kết thúc (CI/CD pipeline step) teardown DB bằng script sau khi kết thúc (runner ngoài Locust)

```
python teardown_db.py --tag "loadtest-2025-10-18"
```

2.5.3 Luồng trạng thái và kịch bản

Khi dữ liệu đã sẵn, Locust cho phép ta thiết kế "đường đi" của người dùng theo luồng tuần tự, rẽ nhánh, và tỷ lệ pha trộn để phản ánh lưu lượng thực tế.

- **Kịch bản tuần tự/thành phần**: Locust cho phép mô phỏng các chuỗi hành động người dùng theo trình tự xác định. Mặc định, mỗi user sẽ bỏ chọn ngẫu nhiên các tác vụ đã định nghĩa, nhưng nếu muốn mô phỏng một luồng có định (ví dụ: *mở trang chủ → tìm sản phẩm → thêm vào giỏ → thanh toán*), ta có thể gọi các bước này vào trong một tác vụ duy nhất (một phương thức **@task** thực hiện tuần tự nhiều request) để đảm bảo trình tự. Locust cũng hỗ trợ lớp **SequentialTaskSet** cho phép khai báo nhiều phương thức tác vụ và đảm bảo chúng chạy tuần tự theo thứ tự định nghĩa, thay vì ngẫu nhiên.

- **Duy trì trạng thái giữa các bước**: Khi kịch bản có nhiều bước liên quan, ta có thể lưu trạng thái trong biến của user để dùng cho các bước sau. Ví dụ: sau khi login ở bước đầu (**on_start**), lưu lại token phiên (**self.token**). Hoặc nếu bước 1 lấy về một **order_id**, ta gán **self.order_id = ...** để bước 2 có thể sử dụng **self.order_id** trong request tiếp theo. Việc duy trì trạng thái trong đối tượng user (hoặc TaskSet) giúp mô phỏng đúng chuỗi giao dịch liên tục của một người dùng thật.

- **Phân nhánh luồng (branching)**: Không phải tất cả người dùng đều theo cùng một lộ trình; Locust cho phép mô phỏng các hành vi rẽ nhánh bằng cách sử dụng logic điều kiện hoặc xác suất trong code. Ví dụ: một tác vụ có thể viết

```
if random.random() < 0.2:
# ựthc ệhin nhánh A
else:
# ựthc ệhin nhánh B
```

  để 20% người dùng đi theo nhánh A (ví dụ: mua hàng) trong khi 80% đi theo nhánh B (ví dụ: chỉ xem sản phẩm). Cách khác, có thể định nghĩa nhiều tác vụ tương ứng với các nhánh khác nhau và gán trọng số để phản ánh tỷ lệ xảy ra của mỗi nhánh trong tập người dùng.

- **Think-time trong kịch bản**: Ngoài việc dùng **wait_time** để nghỉ giữa các tác vụ, ta có thể thêm thời gian chờ bên trong một kịch bản tuần tự để mô phỏng việc người dùng *lướt lại suy nghĩ hoặc đọc nội dung*. Ví dụ: sau khi tải trang sản phẩm, có thể dùng **time.sleep(2)** (hoặc **gevent.sleep(2)**) để giả lập người dùng xem thông tin trong 2 giây trước khi thêm sản phẩm vào giỏ. Việc đưa think-time vào luồng giúp hành vi người dùng đo sát thực tế

hơn và tránh tạo tải không tự nhiên (người dùng thật thường không spam request liên tục).

- **Kết hợp nhiều luồng người dùng**: Để mô phỏng *mix traffic* – trộn kết hợp nhiều loại người dùng với hành vi khác nhau trong cùng một test. Có thể sử dụng nhiều lớp **HttpUser** khác nhau cho mỗi loại kịch bản. Mỗi lớp người dùng có thể đại diện cho một vai trò hoặc trường hợp sử dụng (ví dụ: người dùng khách, người dùng đăng nhập mua hàng). Sử dụng thuộc tính **weight** ở cấp lớp người dùng để điều chỉnh tỷ lệ giữa các kịch bản này khi chạy đồng thời (ví dụ: **class Buyer(HttpUser): weight = 3** và **class Guest(HttpUser): weight = 1** để có 75% người dùng là người mua, 25% là khách vãng lai). Locust sẽ phân bổ số lượng user ảo theo trọng số đã chỉ định.
- **Sử dụng tag cho kịch bản**: Locust cho phép gắn nhãn cho tác vụ hoặc TaskSet bằng decorator **@tag("tên_nhãn")**.
  - Tính năng này hữu ích khi muốn chạy một phần cụ thể của kịch bản. Ví dụ: có thể đánh dấu các tác vụ liên quan đến luồng "thanh toán" bằng tag **"checkout"**. Khi chạy test, ta có thể dùng tham số dòng lệnh **–tags checkout** để chỉ chạy những tác vụ có nhãn đó (hoặc **–exclude-tags** để loại trừ). Điều này giúp linh hoạt trong việc thử nghiệm từng phần của kịch bản hoặc tập trung vào điểm nóng cụ thể mà không cần tách hẳn thành file Locust khác.
  - Ngoài ra, gắn tags cho các tasks hoặc requests để phân loại và lọc kết quả. Ví dụ: tag **"read"** cho GET requests, **"write"** cho POST/PUT, giúp phân tích riêng performance của từng nhóm operations.

Ví dụ 2.1: Mô phỏng hành vi người dùng với tuần tự, nhánh, think-time, tag và mix traffic

```
        import random
        import gevent
        from locust import HttpUser, SequentialTaskSet, task, tag, between

        # ----------------------------
        # 1) TaskSet tuan tu, giu trang thai, re nhanh & think-time
        # ----------------------------
        class BrowseFlow(SequentialTaskSet):
        def on_start(self):
        # (Tuy chon) giu trang thai phien: token, cart_id...
        # Vi duj gia dinh login ben ngoai hoac khong can login
        self.selected_product_id = None

        @task
```

```
        @tag("read", "list")
        def list_products(self):
        # Buoc 1 : Xem danh sach san pham
        res = self.user.client.get("/products")
        if res.ok:
        items = res.json().get("items", [])
        if items:
        # Giu trang thai de dung cho buoc sau
        self.selected_product_id = items[0]["id"]
        # Think-time tu nhien : nguoi dung dung doc 1.5s
        gevent.sleep(1.5)

        @task
        @tag("read", "detail")
        def view_detail(self):
        # Buoc 2 : Xem chi tiet san pham da chon o Buoc truoc
        if self.selected_product_id:
        self.user.client.get(f"/products/{self.selected_product_id}")
        gevent.sleep(0.8)

        @task
        @tag("write", "checkout")
        def maybe_checkout(self):
        # Buoc 3 : Re nhanh co xac suat (20%) thuc hien mua hang
        if self.selected_product_id and random.random() < 0.2:
        self.user.client.post("/cart", json={"id":
            self.selected_product_id})
        self.user.client.post("/checkout")
        # Ket thuc 1 vong flow, quay lai buoc dau (SequentialTaskSet lap
            lai)
        gevent.sleep(0.5)

        # ----------------------------
        # 2) Nhieu lop HttpUser de mo phong mix traffic
        # ----------------------------
        class Guest(HttpUser):
        """
        Khach vang lai: Chi duyet trang chu hoac danh sach
        weight = 1  -> Chiem ti le nho hon (Vi du 25% neu Buyer = 3)
        """
        wait_time = between(1, 3)
        weight = 1

        @task(3)
        @tag("read", "home")
        def home(self):
        self.client.get("/")

        @task(1)
        @tag("read", "list")
        def list(self):
        self.client.get("/products")

        class Buyer(HttpUser):
        """
        Nguoi mua : chay flow tuan tu BrowseFlow.
        weight = 3 -> chiem ti le cao hon( vi du 75% )
        """
```

```
        wait_time = between(1, 2)
        tasks = [BrowseFlow]
        weight = 3
```

2.5.4 Cấu hình tải và thực thi

- **Số lượng user ảo và tốc độ sinh user:** Cấu hình tải trong Locust chủ yếu dựa vào hai tham số chính: *số lượng người dùng đồng thời và tốc độ tạo người dùng mới*. Số lượng user (thường ký hiệu -u hoặc –users) xác định có tối đa bao nhiêu người dùng ảo hoạt động cùng lúc trong test. Tốc độ spawn rate (ký hiệu -r hoặc –spawn-rate) xác định số user ảo khởi tạo mỗi giây cho đến khi đạt tổng số yêu cầu. Ví dụ: -u 1000 -r 50 nghĩa là tăng dần đến 1000 user ảo với tốc độ 50 user mỗi giây. Việc điều chỉnh spawn rate giúp mô phỏng kịch bản ramp-up dần dần thay vì đột ngột tạo toàn bộ user (tránh gây sốc tải tức thời trừ khi đó là tình huống cần test).

Ví dụ 2.2: Kịch bản ba giai đoạn tải: ramp-up → giữ ổn định → ramp-down

```
        # locustfile.py
        from locust import HttpUser, task, between, LoadTestShape

        class ShopUser(HttpUser):
        wait_time = between(1, 3)

        @task
        def browse(self):
        self.client.get("/products")

        # Stages/steps: ramp-up → hold → ramp-down
        class StagesShape(LoadTestShape):
        stages = [
        {"duration": 60, "users": 100, "spawn_rate": 20},  # 1) ramp-up 100 user trong 60s
        {"duration": 360, "users": 100, "spawn_rate": 20}, # 2) giu on dinh 5 phut (tong 6 phut)
        {"duration": 420, "users": 0,   "spawn_rate": 50}, # 3) ramp-down trong 1 phut
        ]

        def tick(self):
        run_time = self.get_run_time()
        for s in self.stages:
        if run_time < s["duration"]:
        return (s["users"], s["spawn_rate"])
        return None  # End test
```

- **Thời gian chạy test:** Locust cho phép đặt thời gian giới hạn cho bài test. Mặc định, nếu không chỉ định, test sẽ chạy cho đến khi người vận hành dừng thủ công. Để tự động dừng sau một khoảng thời gian, có thể dùng tuỳ

chọn -t hoặc –run-time, ví dụ: -t 5m để chạy trong 5 phút rồi kết thúc. Cần chọn thời gian đủ dài để thu thập số liệu ổn định (thường vài phút trở lên), nhưng cũng không nên quá dài trừ khi cần kiểm tra độ bền (soak test).

- **Stages/steps:** Định nghĩa các giai đoạn tải khác nhau trong test. Ví dụ: ramp-up 100 users trong 1 phút, giữ ổn định 5 phút, ramp-down về 0. Locust cung cấp `LoadTestShape` class cho phép tạo load patterns phức tạp.

- **Thực thi qua giao diện Web UI:** Khi chạy Locust mà **không** dùng chế độ headless, Locust sẽ khởi động một web UI (mặc định tại **http://localhost:8089**). Qua giao diện này, người dùng có thể nhập số users, spawn rate, thời gian chạy và nhấn `Start` để bắt đầu test. Web UI cung cấp bảng số liệu và biểu đồ thời gian thực về thông lượng, độ trễ, tỉ lệ lỗi... giúp quan sát trực quan trong khi test đang chạy. Đây là cách thuận tiện để thực hiện thử nghiệm thủ công hoặc phân tích tương tác thời gian thực.

- **Thực thi bằng dòng lệnh (CLI) / chế độ headless:** Đối với chạy tự động (ví dụ trong pipeline CI/CD) hoặc khi không cần giao diện đồ hoạ, Locust hỗ trợ chế độ headless. Sử dụng cờ –headless kèm các tham số cần thiết (`-u`, -r, -t, -H cho host, v.v.) để Locust chạy ngay bài test từ dòng lệnh. Ở chế độ này, kết quả sẽ được log ra console theo chu kỳ (mặc định mỗi 5s in thống kê một lần). Chế độ headless cho phép tích hợp dễ dàng với script và thu thập kết quả bằng file hoặc log mà không cần sự can thiệp của con người. Lưu ý khi chạy headless, nên chỉ định rõ ràng các tham số, và có thể dùng thêm –csv để xuất kết quả ra file.

Ví dụ 2.3: Ví dụ chạy Locust ở chế độ headless và xuất file CSV

```
# 1000 users, spawn 50/s, run 10 phút, ghi CSV
locust -f locustfile.py --headless -H https://api.example.com \
-u 1000 -r 50 -t 10m --csv run_2025_10_22
```

- **Chế độ phân tán (master–worker):** Locust được thiết kế để có thể sinh tải phân tán trên nhiều máy. Điều này hữu ích khi một máy đơn lẻ không đủ tài nguyên tạo lượng user mong muốn. Cấu hình master–worker: khởi chạy một tiến trình Locust làm **Master** (ví dụ: locust -f test.py –master), sau đó khởi chạy một hoặc nhiều tiến trình Locust khác làm **Worker** với cờ –worker –master-host=<địa_chỉ_master>. Master sẽ đóng vai trò điều phối: nó nhận cấu hình test (số user, spawn rate) và phân phối người dùng cho các worker, đồng thời thu thập số liệu từ chúng để tổng hợp báo cáo

thống nhất. Các worker chỉ lo việc mô phỏng user và gửi kết quả về master. Giao tiếp qua ZeroMQ. Mô hình này cho phép dễ dàng mở rộng quy mô test bằng cách tăng số lượng worker, vượt quá giới hạn tài nguyên của một máy đơn lẻ.

```
  # Master (điều phối + UI)
locust -f locustfile.py --master --web-port 8089

  # Worker (1 hoặc nhiều máy)
locust -f locustfile.py --worker --master-host 10.0.0.1
```

- **Triển khai bằng Docker/Kubernetes:** Để thuận tiện và linh hoạt, có thể chạy Locust trong Docker containers. Locust cung cấp image Docker chính thức; người dùng có thể dùng một container làm **master** và nhiều container khác làm **worker**. Ví dụ:

Ví dụ 2.4: Chạy master và worker với Docker

```
docker run -p 8089:8089 locustio/locust:latest -f /mnt
    /locustfile.py --master
docker run locustio/locust:latest -f /mnt/locustfile.
    py --worker \
--master-host="master_container_ip"
```

Tương tự, trên Kubernetes có thể triển khai Locust master và worker dưới dạng Deployment/Pod, triển khai thông qua Helm chart có sẵn. Việc container hóa giúp dễ dàng tích hợp hệ thống và hạ tầng hiện có, chỉnh cấu hình, cũng như tăng/giảm số lượng worker nhanh chóng phù hợp với kịch bản tải.

Ví dụ docker-compose.yml:

Ví dụ 2.5: Cấu hình docker-compose Locust

```
services:
master:
image: locustio/locust
command: -f /mnt/locustfile.py --master --web-
    port 8089
ports: ["8089:8089"]
volumes: ["/locustfile.py:/mnt/locustfile.py"]

worker:
image: locustio/locust
command: -f /mnt/locustfile.py --worker --master
    -host master
depends_on: [master]
volumes: ["/locustfile.py:/mnt/locustfile.py"]
deploy:
replicas: 4  # scale nhanh s ố worker
```

Tóm tắt các tham số:

- **Users/Spawn-rate:** kiểm soát mức đồng thời và tốc độ nở tải.
- **Run-time:** dùng từ dòng lệnh hoặc dựa vào `LoadTestShape`.
- **Stages/steps:** mô tả ramp-up/hold/ramp-down bằng `LoadTestShape`.
- **Web UI:** thao tác thủ công, quan sát realtime.
- **CLI/headless:** chạy tự động, xuất CSV.

2.5.5 Thu thập số liệu & tiêu chí vượt/không vượt

- **Các số liệu hiệu năng chính:** Locust sẽ tự động thu thập và hiển thị nhiều metric quan trọng trong quá trình test. Bao gồm:
  - **Requests per second (RPS):** số lượng request gửi đi mỗi giây (càng nhiều là thông lượng hệ thống xử lý tốt).
  - **Tỷ lệ lỗi (error rate):** phần trăm số request thất bại (mã HTTP >= 400).
  - **Thời gian phản hồi trung bình (average response time)** và các phân vị thời gian phản hồi như **P50 (median)**, **P90**, **P95**, **P99** – cho biết phân bố độ trễ (ví dụ P95 = 500ms nghĩa là 95% số request có thời gian $\leq$ 500ms).
  - Ngoài ra còn có số lượng kết nối người dùng đang hoạt động, tổng số request đã thực hiện, v.v.

  Các metric này đặc biệt quan trọng để đánh giá trải nghiệm người dùng và hiệu năng hệ thống.
- **Xác định tiêu chí Pass/Fail:** Trước khi test, cần đặt ra tiêu chuẩn chấp nhận hiệu năng (SLA hoặc SLO). Ví dụ: *99% request phải dưới 1 giây, throughput tối thiểu 200 req/s, và tỷ lệ lỗi không quá 0.5%.* Sau khi chạy, dựa trên số liệu thu thập, đánh giá xem hệ thống vượt hay không vượt tiêu chí. Nếu bất kỳ chỉ số nào vượt quá ngưỡng cho phép (ví dụ thời gian P99 cao hơn 1 giây, hoặc error rate vượt 0.5%) thì bài test bị coi là thất bại (fail về hiệu năng). Ngược lại, nếu tất cả tiêu chí đều thỏa, hệ thống được xem là đạt yêu cầu dưới tải kiểm thử.
- **Thiết lập ngưỡng và giám sát khi chạy:** Locust cho phép người viết kịch bản thiết lập các sự kiện xử lý động dựa trên số liệu thống kê. Ví dụ: có thể sử dụng hook events.request_failure để đếm số lỗi và nếu tỷ lệ lỗi vượt quá 5% thì tự động dừng test sớm. Hoặc dùng hook events.quitting để kiểm tra

các thống kê tổng thể khi kết thúc: nếu environment.stats.total_fail_ratio (tỷ lệ thất bại) vượt quá một ngưỡng nào đó,
ta có thể đặt environment.process_exit_code = 1 để báo hiệu test không đạt (trả mã exit code khác 0 cho CI/CD). Những cơ chế này giúp tự động hóa đánh giá pass/fail thay vì kiểm tra thủ công.

- **Báo cáo và xuất khẩu số liệu:** Sau khi hoàn thành test, việc lưu trữ và phân tích số liệu rất quan trọng. Locust hỗ trợ xuất báo cáo ra file CSV bằng tham số –csv <tên_file_prefix>, kết quả sẽ gồm các file CSV chứa thống kê (theo từng khoảng thời gian và tổng kết). Các file này có thể được mở bằng công cụ phân tích (Excel, Python, v.v.) để vẽ biểu đồ chi tiết hoặc so sánh giữa các lần chạy khác nhau.

  Ngoài ra, Locust có thể tích hợp với hệ thống giám sát như Prometheus: thông qua `Prometheus exporter`, các metric của Locust có thể được đẩy hoặc scrape định kỳ và hiển thị trên Grafana theo thời gian thực.

  Cách làm phổ biến là chạy một exporter (ví dụ dự án locust_exporter) lắng nghe số liệu từ Locust và cung cấp endpoint /metrics cho Prometheus. Nhờ đó, nhóm kiểm thử có thể quan sát hiệu năng tổng thể dưới kịch bản hoạt động nặng, so sánh với baseline tài nguyên hệ thống cùng thời điểm và theo dõi xu hướng.

2.5.6 Hỗ trợ giao thức & mở rộng

- **Hỗ trợ mặc định HTTP/HTTPS:** Locust đã hỗ trợ tích hợp sẵn cho giao thức HTTP/HTTPS thông qua lớp HttpUser và thư viện requests (đã được patch với gevent để hoạt động bất đồng bộ). Điều này phù hợp cho việc load test các web application, REST API. Tuy nhiên, Locust không gọi chỉ đến test các HTTP – nó có thể được mở rộng để kiểm thử hầu hết các hệ thống/giao thức khác thông qua code Python.

- **Kiểm thử các giao thức khác (gRPC, WebSocket, MQ...):** Để sử dụng Locust với các giao thức phi HTTP, ta thường phải tự viết client hoặc dùng thư viện phụ hợp và tích hợp thủ công với Locust. Nguyên tắc chung là: thực hiện thao tác giao tiếp bằng thư viện Python tương ứng (ví dụ dùng grpclio để gõ gRPC, dùng thư viện websocket-client cho WebSocket, hoặc driver cho MQ như Pika cho RabbitMQ), sau đó gọi các sự kiện của Locust để ghi nhận kết quả. Từ đó, Locust cung cấp event Environment.events.request để tự log request tùy chỉnh. Lập trình viên có thể fire event này với các data như tên request, thời gian, kích thước, kết quả (hoặc exception nếu lỗi) khi mô hình thành một giao thức custom, các số liệu sẽ ghi nhận như

các request thông thường. Lưu ý: để đạt hiệu quả cao, các thư viện client sử dụng nên **tương thích với gevent** (non-blocking), nếu không sẽ làm giới hạn số lượng user đồng thời trên mỗi worker.

- **Sử dụng extension có sẵn:** Cộng đồng Locust có nhiều **phần mở rộng** hỗ trợ giao thức khác giúp giảm công sức viết tay. Ví dụ: gói locust-plugins cung cấp sẵn lớp GrpcUser để test gRPC hoặc WebSocketUser để test thông qua WebSocket (sử dụng thư viện như websockets hoặc socket.io). Tài liệu Locust cũng có mẫu code cho việc test MQTT, XML-RPC, v.v. thông qua việc wrap client và fire event. Khi cần kiểm thử giao thức đặc thù, nên tìm kiếm xem đã có plugin hoặc hướng dẫn có sẵn, từ đó tuỳ biến theo nhu cầu. Nếu không khả năng mở rộng linh hoạt, Locust có thể dùng cho nhiều loại hệ thống: từ web, mobile API đến ví dụ như message queue, v.v.
- **Hooks và event tùy chỉnh:** Locust cung cấp cơ chế hook/event mạnh mẽ cho phép mở rộng chức năng mà không cần sửa mã nguồn Locust. Ngoài các hook cho request như đã đề cập, còn có các sự kiện khác như:
  - test_start, test_stop (kích hoạt khi bắt đầu/kết thúc toàn bộ test),
  - spawn_complete (khi đã sinh xong toàn bộ user),
  - user_error (khi có exception không được xử lý trong task của user), v.v.

    Người dùng có thể đăng ký callback vào những event này để thực hiện các hành động tùy chỉnh. Ví dụ: dùng events.test_start.add_listener để thiết lập cấu hình logging đặc biệt hoặc đánh dấu thời gian bắt đầu; dùng events.test_stop.add_listener để thu thập và lưu trữ thêm số liệu (như kết nối trung bình mở trong DB, v.v.) hoặc gửi thông báo khi test kết thúc. Với events.request_success và events.request_failure, có thể ghi log chi tiết từng request ra file ngoài, hoặc tích hợp gửi metric tới hệ thống giám sát bên ngoài (như gửi dữ liệu lên StatsD/Graphite). Cơ chế hooks và events giúp mở **rộng Locust** vượt khỏi tính năng mặc định, mở ra các nhu cầu đo đạc và ghi nhận chuyển biến trong những tình huống phức tạp.
- **Tuỳ chỉnh client và logic nâng cao:** Ngoài ra, do kịch bản Locust thực chất là code Python, người dùng có thể tuỳ ý tích hợp kịch bản logic nâng cao hơn. Ví dụ: tự đo thời gian cho một nhóm thao tác phức tạp (sử dụng @tag, time.time()) trước và sau để tính toán), ghi theo dạng JSON để dễ phân tích, hoặc thậm chí điều chỉnh hành vi user theo phân hồi nhận được (ví dụ: nếu nhận mã HTTP 500 thì chuyển sang nhóm hành động khác).

Nhờ điều này mà khi thực hiện trong code Locust, làm test có thể gạt rất linh hoạt trong việc mô phỏng và đo đạc các tình huống tùy ý.

2.5.7 Các lỗi/vấn đề thường phát hiện

- **Timeout (quá thời gian phản hồi):** Dưới tải nặng, một lỗi phổ biến là nhiều request bị *timeout* – nghĩa là hệ thống không phản hồi trong thời gian chờ tối đa. Điều này cho thấy hệ thống không xử lý kịp lưu lượng hoặc có thể có điểm nghẽn nghiêm trọng. Locust sẽ ghi nhận các timeout như là lỗi (thất bại) và thông qua đó nhóm phát triển có thể xác định ngưỡng tải tại đó hệ thống bắt đầu mất phản hồi. Timeout thường đi kèm việc cần tối ưu hiệu năng hoặc tăng tài nguyên cho hệ thống.
- **Hiện tượng spike đột biến:** Khi chạy thử tải, có thể quan sát những lúc đột ngột tỷ lệ lỗi tăng đột ngột, không tuyến tính so với mức tải – gọi là *spike*. Ví dụ, khi tăng user từ 100 lên 200 có thể thời gian phản hồi vượt tăng gấp nhiều lần, rồi sau đó giảm hoặc ổn định lại. Spike cũng có thể xảy ra do hành vi của hệ thống: như chu kỳ *garbage collection*, reset kết nối, hay tải lại cache. Những biến động đột ngột này là dấu hiệu để điều tra nguyên nhân tiềm ẩn (bottleneck ẩn, cấu hình chưa phù hợp) trong hệ thống.
- **Lỗi 429 Too Many Requests:** Mã trạng thái HTTP 429 xuất hiện khi hệ thống hoặc dịch vụ trung gian (load balancer, API gateway) áp dụng giới hạn tần suất và bắt đầu từ chối các yêu cầu vì vượt quá hạn mức. Nếu trong bài test Locust xuất hiện nhiều 429, điều đó cho thấy ta đã đặt (hoặc vượt) ngưỡng giới hạn mà hệ thống cho phép. Đây có thể là hành vi mong đợi (nếu có cơ chế *rate limit*) nhưng cũng có thể báo hiệu hệ thống không đủ sức phục vụ lưu lượng cao hơn. Trong báo cáo, 429 được tính vào lỗi và là cơ sở để cân nhắc tăng hạn mức hoặc quy mô hệ thống.
- **Lỗi 5xx (500, 502, 503...):** Các mã lỗi 5xx ám chỉ lỗi phía server – đây thường là những vấn đề nghiêm trọng được phơi bày khi tải cao. Ví dụ: *500 Internal Server Error* do bug trong code bị lộ ra dưới cạnh tranh đa luồng; *502/503 Bad Gateway/Service Unavailable* do một service con bị sập; *504 Gateway Timeout* khi một thành phần down-line không phản hồi kịp. Khi Locust báo nhiều lỗi 5xx, nhóm ứng dụng cần điều tra nguyên nhân gốc: có thể do quá tải tài nguyên (CPU, RAM, kết nối database) dẫn đến sập dịch vụ, hoặc lỗi logic không xuất hiện ở tải thấp nhưng xảy ra ở tải cao. Những lỗi này thường ưu tiên sửa chữa trước khi tiến hành tải tiếp.

- **Rò rỉ session hoặc resource leak:** Test nhiều ngày kéo dài có thể phát hiện các vấn đề rò rỉ tài nguyên. Một ví dụ phổ biến là rò rỉ session: nếu ứng dụng web tạo session cho mỗi người dùng đăng nhập nhưng không giải phóng đúng cách, số session tích tụ dần sẽ chiếm dần dung lượng ngày càng nhiều bộ nhớ hoặc không gian lưu trữ. Tương tự, có thể rò rỉ các thứ khác như kết nối cơ sở dữ liệu (không đóng), handle file, bộ nhớ của ứng dụng dẫn đến treo theo thời gian chạy. Locust giúp phát hiện điều này khi nhận thấy hiệu năng giảm dần hoặc hệ thống chậm lại rõ rệt sau một thời gian chạy liên tục. Quan sát đồ thị có thể thấy, chẳng hạn, ban đầu ứng dụng nhanh nhưng sau đó bị chậm hẳn, có thể nghi ngờ rò rỉ và cần xem log, giám sát memory để xác nhận.
- **Nút cổ chai ở cơ sở dữ liệu:** Database thường là thành phần dễ trở thành điểm nghẽn dưới tải cao. Các dấu hiệu gồm: thời gian phản hồi tăng khi số lượng user tăng (trong khi CPU app server chưa chắc đã max, mà CPU hoặc IO của DB server lại đang báo hao), hoặc xuất hiện lỗi liên quan đến DB như *timeout query*, *connection pool exhausted*, thậm chí lỗi *deadlock* trong DB. Locust thường được dùng để xác định ngưỡng mà tại đó DB không xử lý nổi nữa. Khi thấy throughput không tăng dù thêm user, latency tăng cao, nhiều truy vấn chậm, đó là lúc DB trở thành *bottleneck*. Giải pháp có thể là tối ưu câu query, thêm chỉ mục, hoặc tăng tài nguyên DB/phân chia tải.
- **Deadlock và các vấn đề đồng thời:** Tải đa người dùng có thể làm lộ ra bug đồng bộ mà bình thường khó xuất hiện. *Deadlock* là ví dụ điển hình – xảy ra khi hai hoặc nhiều tiến trình chờ nhau giữ khóa tài nguyên và không tiến triển. Trong hệ thống web, deadlock có thể xảy ra ở cấp ứng dụng (code) hoặc ở cấp database (các giao dịch khóa bảng/theo thứ tự khác nhau). Locust chạy hàng trăm, hàng ngàn user đồng thời có thể khiến những tình huống race condition hoặc deadlock này xảy ra tương đối thường xuyên (ví dụ gặp lỗi deadlock trên SQL trả về hoặc thread dump của ứng dụng cho thấy nhiều thread chờ lock). Phát hiện sớm deadlock cực kỳ quan trọng vì nó gây treo dịch vụ. Nhờ test, đội ngũ phát triển có thể bổ sung cơ chế timeout, thay đổi thứ tự khóa hoặc sửa logic để loại bỏ tình trạng deadlock.

2.6 Các lỗi có thể phát hiện

Thông qua kiểm thử tải và hiệu năng, Locust giúp phát hiện nhiều loại lỗi tiềm ẩn trong hệ thống:

1. **Lỗi hiệu năng:**

- Ứng dụng phản hồi chậm khi lượng người dùng tăng.
- Request timeout do server không đáp ứng kịp.
- Hiệu suất giảm mạnh ở các điểm tải cao.

2. **Lỗi máy chủ (Server-side errors):**
   - Phản hồi HTTP 5xx (*Internal Server Error, Gateway Timeout, Service Unavailable*).
   - Lỗi xử lý logic hoặc truy vấn cơ sở dữ liệu dưới tải lớn.
3. **Lỗi ứng dụng (Application-level bugs):**
   - Trạng thái phiên (session) không nhất quán.
   - Mất dữ liệu hoặc phản hồi sai khi nhiều người dùng thao tác đồng thời.
4. **Lỗi cấu hình hạ tầng:**
   - Giới hạn kết nối mạng, thread pool, hoặc thiếu tài nguyên RAM/CPU.

Các lỗi này thường chỉ xuất hiện trong điều kiện tải thực tế, vì vậy Locust đóng vai trò quan trọng trong việc tái hiện và phân tích hành vi hệ thống trước khi triển khai thật.

2.7 Ví dụ cấu hình kiểm thử

Một ví dụ đơn giản về kiểm thử tải một website với Locust:

Ví dụ 2.6: Ví dụ kiểm thử tải website bằng Locust

```
        from locust import HttpUser, task, between

        class WebsiteUser(HttpUser):
        wait_time = between(1, 3)  # ờThi gian ờch ữgia các tác ụv (ảgi ậlp
            ườngi dùng ậtht)

        @task(2)
        def view_homepage(self):
        self.client.get("/")  # Truy ậcp trang ủch

        @task(1)
        def view_product(self):
        self.client.get("/product/1")  # Xem chi ếtit ảsn ẩphm
```

Chạy kiểm thử:

```
        locust -f locustfile.py --host=https://example.com
```

Sau khi chạy, mở trình duyệt và truy cập http://localhost:8089 để cấu hình số người dùng và tốc độ sinh người dùng.

3. Đánh giá công cụ

(Phần 3 trong bố cục) Đánh giá công cụ

1.1 Kiểm thử hộp đen

Sau khi tham khảo một số bài báo khoa học, nhóm em đã tìm được các thí nghiệm thực hiện việc so sánh hiệu năng của các công cụ kiểm thử tải (load testing tools) như là **JMeter**, **Locust**, **Gatling**, **k6**, **Taurus**, **OMEXUS** (*công cụ kiểm thử tại nội bộ*).

Mỗi công cụ sẽ gửi yêu cầu nhanh nhất có thể, và các dữ liệu về **thời gian phản hồi trung bình (Average Response Time)**, **lưu lượng xử lý (Throughput)** và **dung lượng bộ nhớ sử dụng (Memory usage)** sẽ được ghi lại, sau đó được sử dụng để thực hiện các phép so sánh cơ bản.

- **Thí nghiệm 1:** Tạo ra **20 giao dịch đồng thời với 5000 vòng lặp** hướng đến môi trường sản xuất. Vòng lặp này chỉ bao gồm 1 yêu cầu HTTP duy nhất.

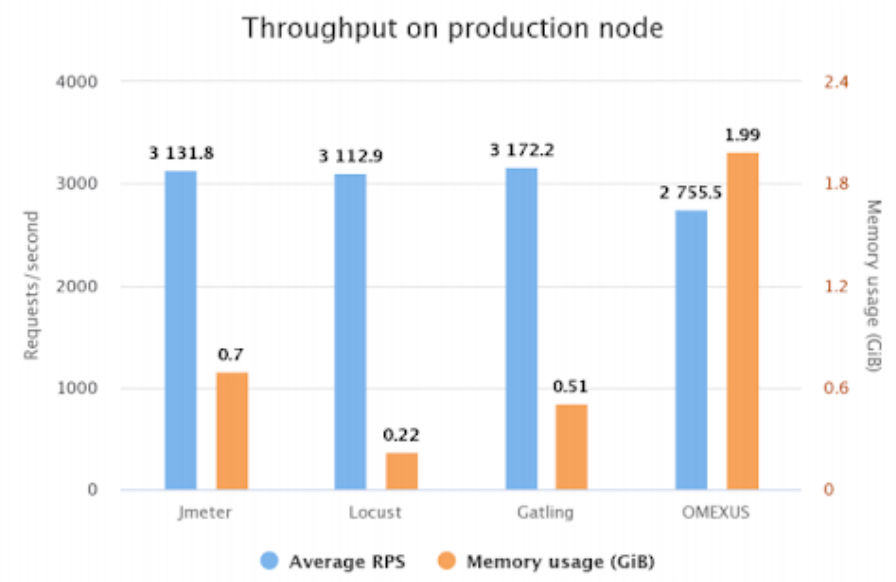


Figure 5.2: Throughput comparison in the production environment

Hình 3.1: Throughput comparison in the production environment

Kết quả: JMeter, **Locust**, và **Gatling** đều đạt được **thông lượng rất cao** và tương đương nhau. Tuy nhiên, để đạt được thông lượng cao đó, **Locust vẫn là công cụ sử dụng ít bộ nhớ nhất** (0.22 GiB), hiệu quả hơn nhiều so với **Gatling** (0.51 GiB) và **JMeter** (0.7 GiB).

- **Thí nghiệm 2:** Tạo ra **200 giao dịch đồng thời và 50 vòng lặp**, hướng đến **môi trường Docker**.
  - Kịch bản "First": Một bài test **đơn giản**, chỉ bao gồm các yêu cầu HTTP thuần túy.
  - Kịch bản "Second": Một bài test **phức tạp** hơn, kết hợp nhiều giao thức khác nhau (ví dụ: HTTP và JDBC).

Hình 3.2: Throughput vs. memory on the docker environment

Kết quả: JMeter đạt được **Average RPS cao nhất** trong bài test đơn giản, hiệu suất giảm nhẹ đối với bài test phức tạp và cũng tiêu tốn nhiều bộ nhớ nhất. Trong khi đó, **Gatling** cho thấy **sự ổn định rất cao** khi hiệu suất và mức sử dụng bộ nhớ gần như không thay đổi giữa hai bài test đơn giản và phức tạp. Ngược lại, dù đạt được Average RPS rất cao, gần bằng JMeter trong bài test đơn giản với mức sử dụng bộ nhớ rất thấp, nhưng khi sang bài test phức tạp, hiệu suất của **Locust** đã **giảm đáng kể và mức tiêu hao bộ nhớ cũng tăng nhiều**.

- **Thí nghiệm 3:** Thử nghiệm việc sử dụng bộ nhớ của máy cục bộ khi thực thi các kiểm thử trong môi trường Docker với số lượng người dùng ảo khác nhau.

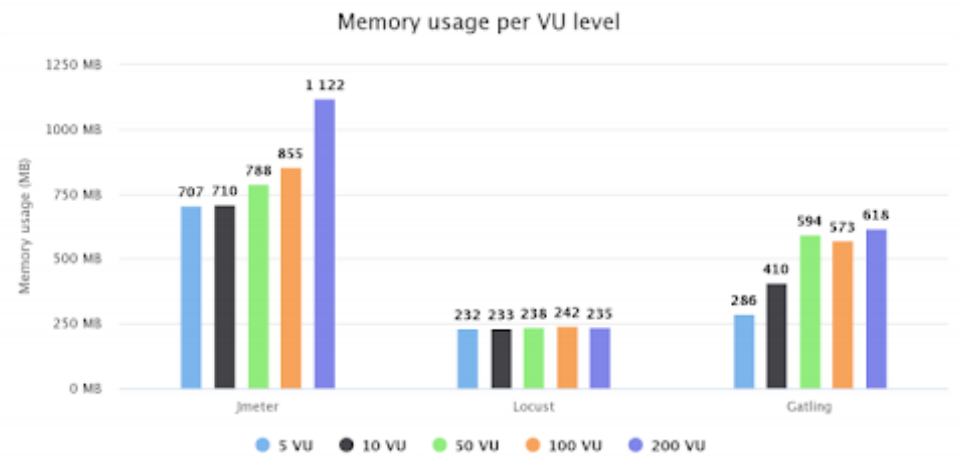


Figure 5.4: Memory usage per virtual user level on the docker environment

Hình 3.3: Memory usage per virtual user level on the docker environment

Kết quả: Mức sử dụng bộ nhớ của **JMeter** tăng theo cấp số nhân khi số lượng người dùng ảo tăng lên. Việc sử dụng bộ nhớ của **Gatling** cũng bị ảnh hưởng bởi số lượng người dùng, nhưng không nhiều như **JMeter**. Qua đó, nổi bật nhất là mức sử dụng bộ nhớ của **Locust** gần như không thay đổi đáng kể, dù số lượng người dùng ảo tăng.

 Locust là công cụ hiệu quả nhất về sử dụng bộ nhớ trong 3 công cụ.

- **Thí nghiệm 4:** Thử nghiệm thời gian phản hồi của hệ thống khi tăng số lượng người dùng

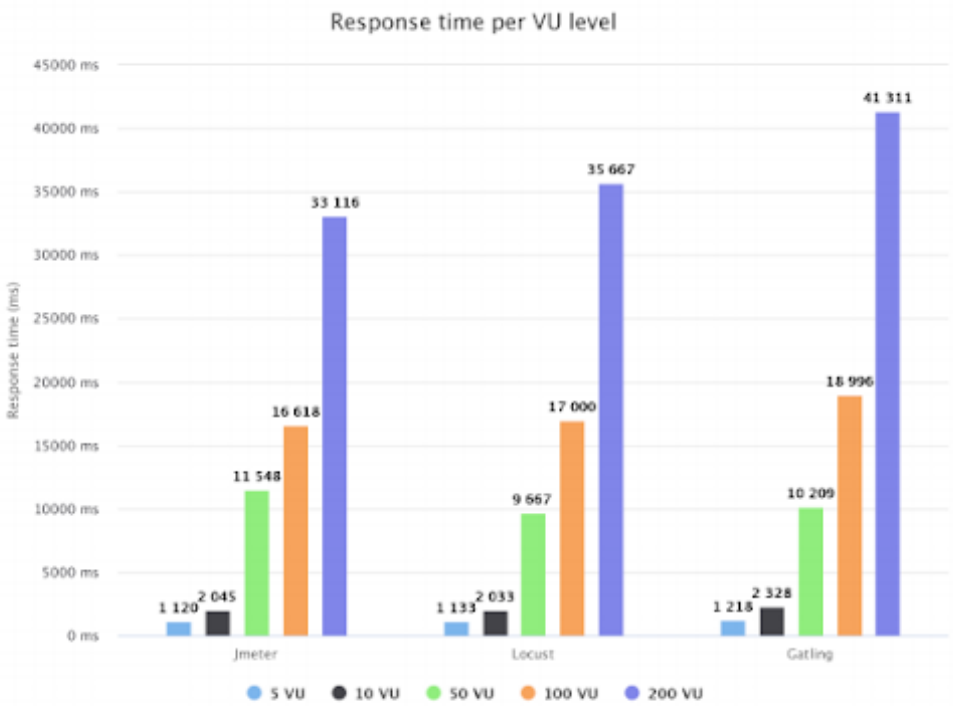


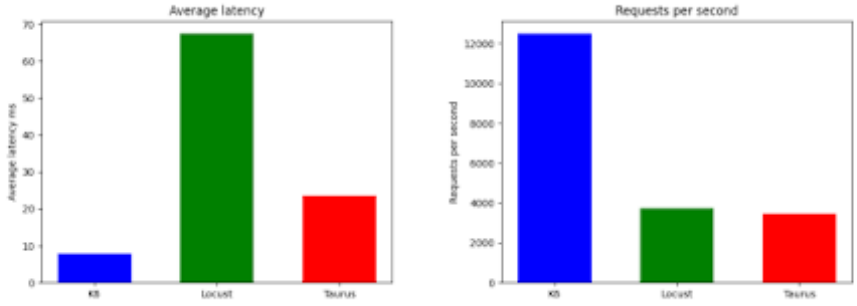
Figure 5.6: Response time at different virtual user levels

Hình 3.4: Response time at different virtual user levels

Kết quả: Thời gian phản hồi của hệ thống tăng tuyến tính đối với cả ba công cụ kiểm thử. Tuy nhiên, **Gatling** làm cho thời gian phản hồi tăng nhiều nhất khi số lượng người dùng ảo tăng lên, trong khi đó **Locust** giữ vị trí thứ hai.

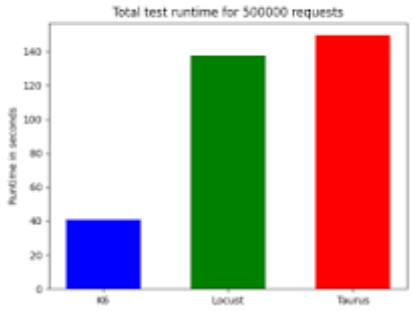
 Gatling có độ trễ phản hồi tăng mạnh nhất khi số lượng người dùng tăng cao.

- **Thí nghiệm 5:**Thử nghiệm hiệu năng của 3 công cụ kiểm thử tải k6, Locust và Taurus khi thực hiện trên 3 loại máy chủ AWS EC2 có cấu hình yếu, trung bình và mạnh với số lượng người dùng ảo thay đổi.



(a) The average recorded latency over 5 runs in milliseconds for the tools.

(b) An average of requests per second over 5 runs for the tools.



(c) An average of the recorded run times in seconds for the tools

Figure 4: Results from running the static test for 5 iterations per tool on an AWS EC2 t3.medium instance for 500 000 requests and with 100 virtual users.

Hình 3.5: Kết quả thực nghiệm trên máy chủ cấu hình yếu (AWS EC2 t3.medium)

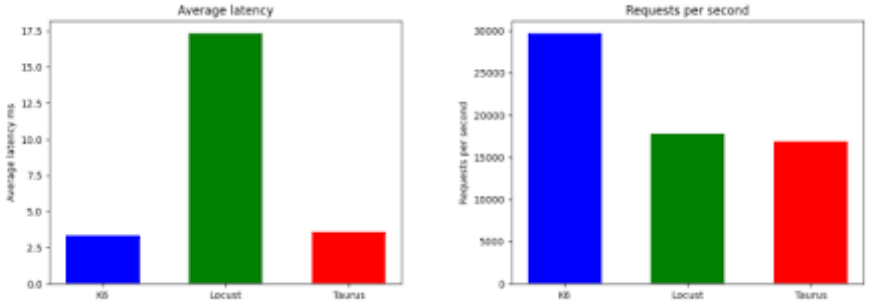
Kết quả: Công cụ **k6** thể hiện hiệu năng vượt trội rõ rệt với:

- Độ trễ trung bình thấp nhất (latency thấp nhất),
- Thông lượng cao nhất (requests per second cao hơn hơn 3 lần so với các công cụ còn lại),
- Và thời gian chạy kiểm thử nhanh nhất (total runtime ngắn nhất).

Ngược lại, **Locust** và **Taurus** hoạt động rất kém trên cấu hình máy chủ yếu:

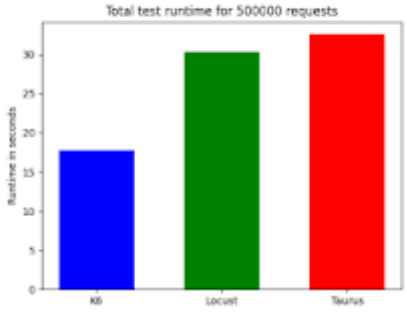
- Độ trễ phản hồi rất cao,
- Thông lượng thấp,
- Thời gian thực thi bài kiểm thử dài hơn đáng kể.

 Trong điều kiện máy chủ yếu, Locust có độ trễ phản hồi cao nhất và hoạt động không hiệu quả bằng k6.



(a) An average of the recorded latency in milliseconds for the tools.

(b) An average of the recorded requests per second.



(c) An average of the recorded run times in seconds for the tools.

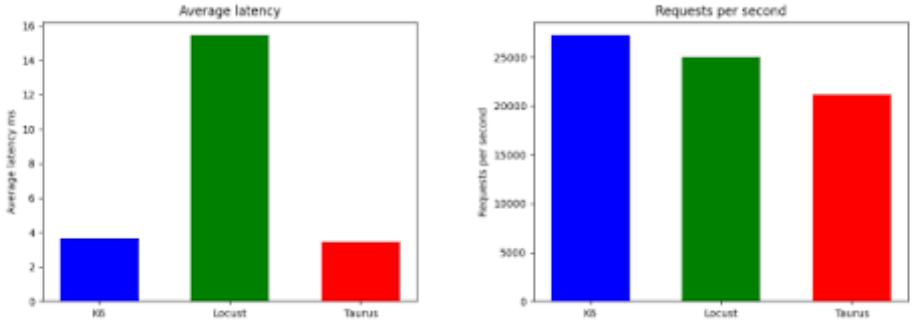
Figure 5: Results from running the static test for 5 iterations per tool on an AWS EC2 c5d.2xlarge instance for 500 000 requests and with 100 virtual users.

Hình 3.6: Kết quả thực nghiệm trên máy chủ cấu hình trung bình (AWS EC2 c5d.2xlarge)

Kết luận: So với kết quả trên máy chủ có cấu hình yếu, hiệu suất của hai công cụ **Locust** và **Taurus** đã được cải thiện đáng kể khi chạy trên phần cứng trung bình:

- Độ trễ phản hồi (latency) giảm rõ rệt.
- Số lượng yêu cầu xử lý mỗi giây (requests per second) tăng lên đáng kể.
- Thời gian thực hiện toàn bộ bài kiểm thử (total runtime) ngắn hơn.

□ Các công cụ bắt đầu thể hiện hiệu quả tốt hơn khi tài nguyên hệ thống được mở rộng, đặc biệt là Locust.



(a) An average of the recorded latency in milliseconds for the tools.

(b) An average of the recorded requests per second.

(c) An average of the recorded run times in seconds for the tools.

Figure 6: Results from running the static test for 5 iterations per tool on an AWS EC2 c5d.4xlarge instance for 500 000 requests and with 100 virtual users.

Hình 3.7: Kết quả thực nghiệm trên máy chủ cấu hình mạnh (AWS EC2 c5d.4xlarge)

Kết luận: Dựa trên kết quả thực nghiệm:

- K6 tiếp tục duy trì vị trí dẫn đầu về tất cả các chỉ số: độ trễ thấp nhất, lượng yêu cầu xử lý mỗi giây cao nhất, và thời gian thực thi ngắn nhất.
- Locust và Taurus đều cho thấy sự cải thiện rõ rệt về hiệu suất khi chuyển sang chạy trên máy chủ mạnh.
- Locust đã có thể khai thác tốt hơn sức mạnh của phần cứng, đạt được hiệu năng gần tiệm cận với K6.

 Tổng quan: K6 là công cụ cho hiệu năng vượt trội và ổn định nhất trên cả ba loại cấu hình, trong khi đó **Locust** thể hiện hiệu suất phụ thuộc nhiều vào tài nguyên phần cứng.

3.0.1 Đánh giá dựa trên các tiêu chí kĩ thuật

Table 1: A comparison of candidate load test tools to select the three most suitable tools for the task. The most important features required are answered in a yes or no fashion, and the language used for creating the test scrips is also presented.

Name	Measures in μs	Well documented	GitLab compatible	Open source	Cloud support	Script language
k6	✓	✓	✓	✓	✓	JavaScript
Frisbee	✗	✗	✓	✓	✗	YAML
Gatling	✗	✓	✓	✓	✓	Java
Locust	✓	✓	✓	✓	✓	Python
Apache JMeter	✓	✓	✓	✓	✗	Groovy
Artillery	✗	✓	✓	✓	✓	YAML
Taurus	✓	✓	✓	✓	✓	YAML
SmartMeter	✗	✓	✓	✗	✗	GUI based
LoadRunner	✓	✓	✓	✗	✓	C, Java, .Net, JavaScript

Kết luận: Qua bảng so sánh trên, **k6**, **Locust** và **Taurus** đều cho thấy khả năng đáp ứng các tiêu chí kỹ thuật vượt trội, đặc biệt là khả năng tương thích với `GitLab`, là dự án *Open source*, hỗ trợ đám mây và có ngôn ngữ kịch bản linh hoạt.

4. So sánh công cụ Locust

4.1 Giới thiệu chung về Locust

Nền tảng kiến trúc của Locust dựa trên cơ chế xử lý đồng thời điều khiển bằng sự kiện (event-driven), sử dụng các coroutine cực nhẹ gọi là greenlet. Điều này cho phép một máy duy nhất có thể mô phỏng hàng ngàn người dùng đồng thời mà không tiêu tốn nhiều tài nguyên, một ưu điểm lớn so với các công cụ dựa trên luồng truyền thống. Để đánh giá đúng vị thế của Locust, nhóm em sẽ so sánh trực tiếp nó với ba công cụ phổ biến khác trong ngành: Apache JMeter, K6, và Gatling.

4.2 So sánh chi tiết

4.2.1 Locust vs. Apache JMeter

Apache JMeter là một trong những công cụ kiểm thử hiệu năng lâu đời và phổ biến nhất, nổi bật với giao diện đồ họa (GUI) trực quan cho phép người dùng xây dựng kịch bản mà không cần viết nhiều mã. Cuộc đối đầu giữa Locust và JMeter là sự so sánh giữa một bên là "tests-as-code" linh hoạt cho nhà phát triển và một bên là "GUI-driven" dễ tiếp cận cho người không chuyên về lập trình.

Bảng 4.1: So sánh Locust và Apache JMeter

Đặc điểm	Locust	Apache JMeter
Ngôn ngữ chính	Python	Java (với Groovy/BeanShell)
Mô hình cốt lõi	Tests-as-Code	Kế hoạch kiểm thử qua GUI
Mô hình đồng thời	Dựa trên sự kiện (Greenlets)	Mỗi luồng một người dùng
Giao diện chính	Web UI & Dòng lệnh (CLI)	Giao diện đồ họa (GUI) & CLI
Báo cáo	Web UI thời gian thực	Báo cáo HTML chi tiết sau kiểm thử
Điểm mạnh chính	Linh hoạt, hiệu quả tài nguyên, dễ tích hợp CI/CD	Hỗ trợ nhiều giao thức, cộng đồng lớn, dễ bắt đầu không cần code

Nhận xét: Sự khác biệt lớn nhất nằm ở kiến trúc. Mô hình ”mỗi luồng một người dùng” của JMeter tiêu tốn nhiều tài nguyên hơn đáng kể so với mô hình dựa trên sự kiện của Locust. Do đó, Locust có khả năng mô phỏng số lượng người dùng đồng thời lớn hơn nhiều trên cùng một phần cứng. JMeter phù hợp cho các đội ngũ QA truyền thống hoặc khi cần kiểm thử các giao thức đa dạng ngoài HTTP (như JDBC, FTP). Ngược lại, Locust là lựa chọn vượt trội cho các đội ngũ phát triển theo định hướng DevOps, ưu tiên tự động hóa, quản lý phiên bản kịch bản kiểm thử và yêu cầu hiệu năng cao.

4.2.2 Locust vs. K6

K6 là một công cụ kiểm thử hiệu năng hiện đại khác, cũng theo triết lý “tests-as-code” tương tự Locust. Tuy nhiên, K6 được xây dựng bằng Go và sử dụng JavaScript để viết kịch bản, nhắm đến cộng đồng phát triển web rộng lớn. Locust và K6 là hai công cụ cùng thế hệ, có chung triết lý nhưng khác biệt về hệ sinh thái công nghệ.

Bảng 4.2: So sánh Locust và K6

Đặc điểm	Locust	K6
Ngôn ngữ chính	Python	JavaScript (chạy trong Go runtime)
Mô hình cốt lõi	Tests-as-Code	Tests-as-Code
Mô hình đồng thời	Dựa trên sự kiện (Greenlets)	Dựa trên sự kiện (Goroutines)
Giao diện chính	Web UI & Dòng lệnh (CLI)	Ưu tiên Dòng lệnh (CLI)
Phong cách báo cáo	Web UI thời gian thực	CLI trực tiếp, tích hợp tốt với Grafana/Datadog
Điểm mạnh chính	Hệ sinh thái Python mạnh mẽ, kịch bản linh hoạt	Hiệu năng thực thi cao, tích hợp sâu với hệ sinh thái Grafana

Nhận xét: Cả Locust và K6 đều có kiến trúc dựa trên sự kiện hiệu quả, giúp chúng vượt trội hơn JMeter về mặt sử dụng tài nguyên. Sự lựa chọn giữa hai công cụ này thường phụ thuộc vào kỹ năng và hệ sinh thái của đội ngũ phát triển. K6 là lựa chọn tốt cho các nhóm làm việc chủ yếu với JavaScript/TypeScript và đã đầu tư vào hạ tầng giám sát với Grafana. Ngược lại, Locust hấp dẫn hơn đối với các đội ngũ backend Python, kỹ sư SRE, hoặc khi kịch bản kiểm thử đòi hỏi logic phức tạp cần đến sức mạnh của các thư viện Python phong phú.

4.2.3 Locust vs. Gatling

Gatling là một công cụ hiệu năng cao, được xây dựng trên nền tảng JVM (sử dụng Scala) và cũng áp dụng kiến trúc bất đồng bộ dựa trên sự kiện. Giống như Locust và K6, Gatling là một công cụ "tests-as-code" nhưng sử dụng một Ngôn ngữ Đặc tả Miền (DSL) riêng. So sánh Locust và Gatling cho thấy sự đánh đổi giữa hiệu năng thô của JVM và tính dễ sử dụng, linh hoạt của Python.

Bảng 4.3: So sánh Locust và Gatling

Đặc điểm	Locust	Gatling
Ngôn ngữ chính	Python	Scala / Java / Kotlin
Mô hình cốt lõi	Tests-as-Code	Tests-as-Code (thông qua DSL)
Mô hình đồng thời	Dựa trên sự kiện (Greenlets)	Bất đồng bộ dựa trên Actor (Akka)
Giao diện chính	Web UI & Dòng lệnh (CLI)	Recorder & Dòng lệnh (CLI)
Phong cách báo cáo	Web UI thời gian thực	Báo cáo HTML chi tiết, trực quan sau kiểm thử
Điểm mạnh chính	Dễ học, linh hoạt, cộng đồng Python lớn	Hiệu năng rất cao trên JVM, báo cáo đẹp mắt

Nhận xét: Gatling thường được đánh giá cao về hiệu năng thực thi và khả năng sinh ra các báo cáo HTML tĩnh rất chi tiết và đẹp mắt. Tuy nhiên, việc sử dụng Scala và DSL riêng có thể tạo ra rào cản học tập đối với các nhóm không quen thuộc với hệ sinh thái JVM. Ngược lại, Locust thân thiện hơn với nhà phát triển nhờ Python, dễ tùy biến và tích hợp linh hoạt. Mặc dù hiệu năng xử lý CPU thuần của Python không thể so sánh với JVM, nhưng kiến trúc event-driven giúp Locust vẫn cực kỳ hiệu quả cho các bài kiểm thử tải bị giới hạn bởi I/O. Vì vậy, Locust là một lựa chọn cân bằng tốt giữa hiệu suất, độ linh hoạt và trải nghiệm phát triển.

4.3 Kết luận

Locust đã khẳng định vị thế là một công cụ kiểm thử hiệu năng hàng đầu cho các đội ngũ kỹ thuật hiện đại. Bằng cách trao quyền cho các nhà phát triển viết kịch bản kiểm thử bằng Python, nó không chỉ mang lại sự linh hoạt vô song mà còn tích hợp kiểm thử hiệu năng một cách tự nhiên vào vòng đời phát triển phần mềm. So với các công cụ truyền thống như JMeter, Locust vượt trội về hiệu quả sử dụng tài nguyên và phù hợp hơn với văn hóa DevOps. So với các đối thủ hiện đại như K6 và Gatling, Locust mang đến một sự cân bằng hấp dẫn giữa tính dễ sử dụng, sức mạnh của hệ sinh thái thư viện và hiệu suất mạnh mẽ, biến nó thành lựa chọn chiến lược cho nhiều bối cảnh phát triển khác nhau.

5. Thực nghiệm

6. Tổng kết

7. Kết luận

7.1 Ưu điểm

Locust sở hữu nhiều ưu điểm vượt trội, giúp nó trở thành một lựa chọn hàng đầu cho các đội ngũ phát triển hiện đại trong kiểm thử hiệu năng:

- **Triết lý "Tests-as-Code" với Python:** Đây là thế mạnh lớn nhất của Locust. Kịch bản kiểm thử được viết hoàn toàn bằng Python, cho phép sử dụng biến, vòng lặp, xác thực logic phức tạp và toàn bộ hệ sinh thái thư viện Python. Điều này giúp dễ dàng bảo trì, dùng Git để quản lý phiên bản và hỗ trợ code review như một phần của quy trình phát triển phần mềm.
- **Hiệu quả tài nguyên và hiệu năng cao:** Locust sử dụng kiến trúc xử lý đồng thời dựa trên sự kiện với *gevent* và *greenlets*. Không cần tạo một luồng hệ điều hành cho mỗi người dùng mô phỏng, Locust có thể tạo hàng ngàn người dùng trong một tiến trình duy nhất, giảm đáng kể chi phí CPU và bộ nhớ so với các công cụ truyền thống.
- **Khả năng mở rộng quy mô vượt trội:** Với mô hình *master–worker*, Locust dễ dàng mở rộng kiểm thử sang nhiều máy khác nhau, mô phỏng hàng trăm nghìn đến hàng triệu người dùng. Điều này phù hợp với kiểm thử phân tán cho các hệ thống quy mô lớn.
- **Tích hợp tốt vào DevOps / CI/CD:** Vì kiểm thử được viết bằng mã nguồn, Locust dễ tích hợp vào GitLab CI, GitHub Actions, Jenkins, Azure DevOps,... Có thể chạy kiểm thử tự động ở chế độ không giao diện (headless), hỗ trợ văn hoá *shift-left*, kiểm thử hiệu năng sớm và liên tục.
- **Giao diện Web trực quan, thời gian thực:** Locust cung cấp Web UI hiển thị số lượng người dùng, số yêu cầu mỗi giây, thời gian phản hồi và tỷ lệ lỗi theo thời gian thực. Người dùng có thể tăng/giảm tải ngay khi kiểm thử đang chạy.

Những ưu điểm trên khiến Locust trở thành một lựa chọn mạnh mẽ cho các đội ngũ chú trọng tự động hóa, linh hoạt và khả năng mở rộng trong kiểm thử hiệu năng.

7.2 Nhược điểm

Mặc dù có nhiều ưu điểm, Locust cũng tồn tại một số hạn chế cần cân nhắc:

- **Yêu cầu kỹ năng lập trình:** Triết lý "tests-as-code" buộc người dùng phải biết Python. Những kiểm thử viên quen với công cụ có giao diện đồ họa như JMeter có thể gặp khó khăn. Locust cũng không cung cấp *script recorder* mặc định, mọi kịch bản đều phải viết thủ công.
- **Báo cáo tích hợp còn hạn chế:** Web UI mạnh trong thời gian thực nhưng báo cáo hậu kiểm còn đơn giản so với JMeter hoặc Gatling. Để quan sát, lưu trữ và phân tích dài hạn, người dùng cần tích hợp thêm Prometheus, InfluxDB hoặc Grafana.
- **Hỗ trợ giao thức mặc định hạn chế:** Locust chủ yếu hỗ trợ HTTP/HTTPS. Muốn kiểm thử các giao thức khác như gRPC, MQTT, WebSocket hoặc JDBC cần viết thêm client tùy chỉnh bằng Python → tốn công và đòi hỏi kỹ thuật cao.
- **Hạn chế khi xử lý tác vụ nặng CPU:** Vì sử dụng mô hình event-driven, Locust tối ưu cho tác vụ I/O. Nếu script chứa tính toán nặng, chúng có thể làm chậm event-loop và gây hiện tượng *greenlet starvation*.

Dù tồn tại hạn chế, Locust vẫn là một công cụ kiểm thử hiệu năng mạnh mẽ khi được áp dụng đúng bối cảnh, đặc biệt với các dự án theo hướng DevOps, lập trình Python và yêu cầu tự động hóa cao.

7.3 Nguồn trích dẫn

Tài liệu tham khảo

[1] Simple Programmer, "What's Load Testing and How Does a Locust Framework Help?", truy cập 15/10/2025.

[2] CheckOps, "Locust", truy cập 15/10/2025.

[3] BrowserStack, "JMeter Distributed Testing: Tutorial", truy cập 08/10/2025.

[4] Locust Official Documentation, "What is Locust?", truy cập 15/10/2025, https://docs.locust.io.

[5] Heyko Oelrichs, "Globally Distributed Load Tests in Azure with Locust", Medium, truy cập 15/10/2025.

[6] Mad Devs, "How to Create and Run Your First Performance Test With Locust", truy cập 15/10/2025.

[7] Locust GitHub Repository, https://github.com/locustio/locust.

[8] Frugal Testing, "Locust for Load Testing: A Beginner's Guide", truy cập 15/10/2025.

[9] Linode Docs, "How to Load Test Your Applications with Locust", truy cập 08/10/2025.

[10] Software Testing Magazine, "Learning Locust: Documentation, Tutorials, Videos", truy cập 08/10/2025.

[11] Loadium, "What is Locust Load Testing?", truy cập 08/10/2025.

[12] PFLB, "JMeter vs. Locust: Which One To Choose?", truy cập 15/10/2025.

[13] Upsun, "Python Gevent in practice: common pitfalls", truy cập 15/10/2025.

[14] BlazeMeter, "Gatling vs. Locust", truy cập 15/10/2025.

[15] JtlReporter, "How to Analyze Locust.io Report", truy cập 15/10/2025.