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

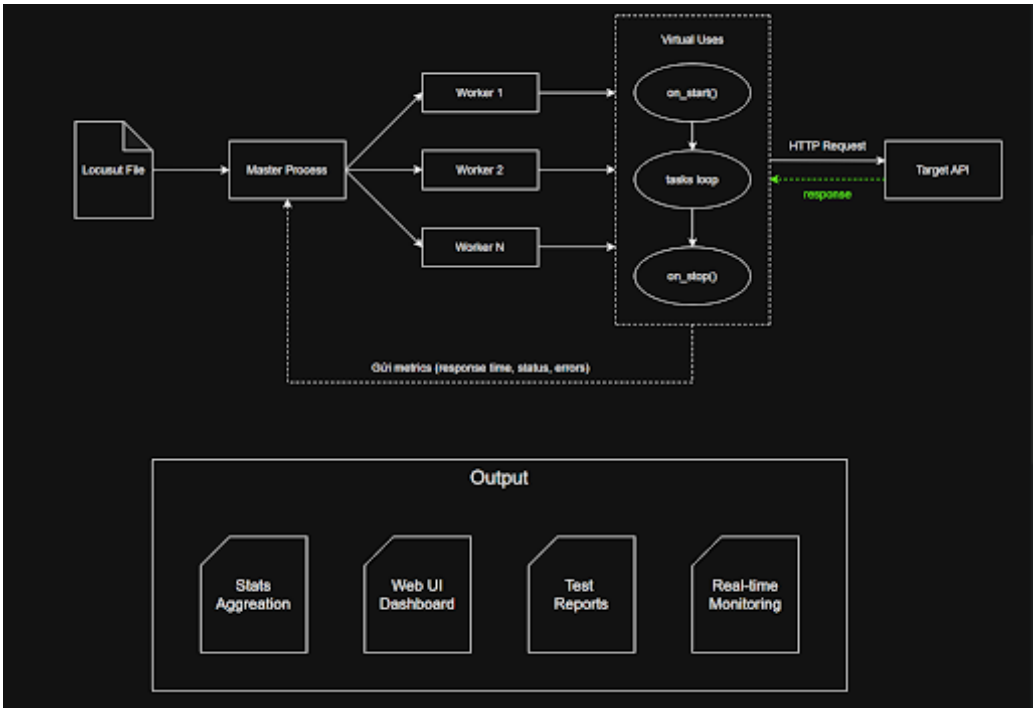
- Dễ dàng tích hợp vào pipeline CI/CD (Jenkins, GitHub Actions, GitLab CI).
- Có thể ghi log, xuất kết quả sang CSV hoặc gửi đến hệ thống giám sát như Grafana, Prometheus, InfluxDB.
- Hỗ trợ kiểm thử không chỉ HTTP mà còn WebSocket, MQTT, gRPC, và các giao thức tuỳ chỉnh thông qua lớp User.

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")
```

3. Đánh giá công cụ

4. So sánh công cụ Locust

5. Thực nghiệm

6. Tổng kết