

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ KHOA CÔNG NGHỆ THÔNG TIN



BÁO CÁO CÔNG CỤ KIỂM THỬ LOCUST

**MÔN HỌC: KIỂM THỬ VÀ ĐÁM 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	5
1.1	Định nghĩa kiểm thử	5
1.2	Vai trò của kiểm thử	5
1.3	Một số thuật ngữ liên quan	6
2	Công cụ kiểm thử Locust	7
2.1	Tổng quan	7
2.2	Các tính năng	7
2.2.1	Mô phỏng người dùng thực tế	7
2.2.2	Kiểm thử tải và hiệu năng	8
2.2.3	Giao diện web trực quan	8
2.2.4	Chạy phân tán	8
2.2.5	Tích hợp và mở rộng	8
2.3	Mục tiêu	9
2.4	Kiến trúc cốt lõi của Locust	9
2.5	Các lỗi có thể phát hiện	44
2.6	Ví dụ cấu hình kiểm thử	45
2.7	Báo cáo và phân tích	46
3	Đánh giá công cụ	47
3.0.1	Đánh giá dựa trên các tiêu chí kỹ thuật	54
4	So sánh Locust với các công cụ kiểm thử hiệu năng tương tự	55
4.1	So sánh chi tiết	55
4.1.1	Locust vs. Apache JMeter	55
4.1.2	Locust vs. K6	56
4.1.3	Locust vs. Gatling	57
4.2	Kết luận	58
5	Kết luận	59
5.1	Ưu điểm	59

5.2	Nhược điểm	60
5.3	Nguồn trích dẫn	60

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 THỦ 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 **nhiều, 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](#).

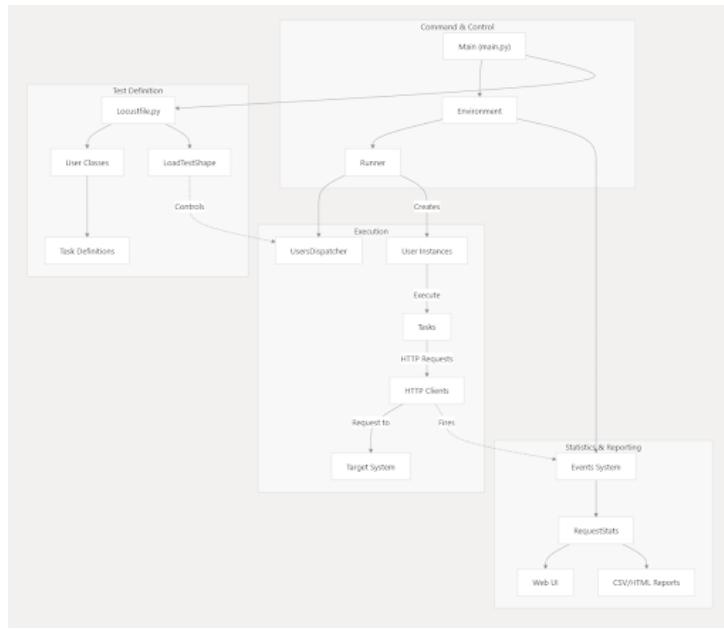
2.3 Mục tiêu

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 Kiến trúc cốt lõi của Locust

Kiến trúc của Locust được thiết kế xoay quanh một số thành phần lõi, phối hợp với nhau để mô phỏng **hàng nghìn người dùng đồng thời** gửi request đến *target system* (hệ thống đích) và thu thập thông kê hiệu năng. Phần này tập trung mô tả **kiến trúc nội bộ** và cách tổ chức mã nguồn của Locust; cách sử dụng chi tiết và API cụ thể sẽ được trình bày ở phần “Using Locust”, còn cơ chế mở rộng framework sẽ được đề cập ở phần “Extending Locust”.



Hình 2.1: Kiến trúc cốt lõi của Locust

Ở mức khái quát, một bài test Locust bao gồm các khối chính:

- **Test Definition** (định nghĩa bài test) trong locustfile.py: khai báo **User Classes** (lớp người dùng ảo) và các **Task Definitions** (định nghĩa tác vụ), tùy chọn thêm **LoadTestShape** (mô tả đường cong tải – số user theo thời gian).
- **Command & Control** (điều khiển): mã trong main.py hoặc lệnh dòng lệnh (command line) chịu trách nhiệm khởi động Locust, đọc tham số, nạp locustfile, tạo **Environment** (môi trường thực thi) và **Runner** (bộ điều phối chạy test).
- **Execution** (thực thi): **Runner** sẽ tạo ra các **User instances** (thể hiện người dùng ảo), phân phối chúng thông qua **UsersDispatcher**, thực thi **Tasks**, gửi **HTTP Requests** đến **Target System** thông qua các **HTTP Clients**.
- **Statistics & Reporting System** (hệ thống thống kê & báo cáo): các **RequestStats** ghi nhận số liệu từ **Events**, sau đó cung cấp dữ liệu cho **Web UI** và các báo cáo CSV/HTML.
- **Web UI**: giao diện web cho phép người dùng khởi động/dừng test, theo dõi số liệu thời gian thực.

Các sơ đồ đi kèm mô tả rõ luồng dữ liệu từ khi Command Line/Web UI khởi động test, qua Environment, Runner, User, Client, đến Stats và hệ thống báo cáo.

2.4.1 Command & Control

2.4.1.1 Environment – Ngữ cảnh trung tâm của bài test

Environment class (lớp Environment) đóng vai trò là **ngữ cảnh trung tâm** cho toàn bộ quá trình thực thi bài test Locust. Có thể coi Environment là một “container” tập trung, giữ tham chiếu tới tất cả các thành phần quan trọng của bài test và cung cấp điểm kết nối để các thành phần này tương tác với nhau một cách nhất quán.

Khi Locust khởi động từ main.py (Command & Control), chương trình sẽ gọi `create_environment()` để tạo ra một đối tượng Environment. Đối tượng này sau đó được truyền cho **Runner**, **Web UI**, **Statistics system** và các module mở rộng khác, giúp chúng dùng chung cùng một bối cảnh cấu hình, event và trạng thái bài test.

Environment

Thuộc tính:

- `+user_classes: list[type[User]]`
- `+shape_class: LoadTestShape`
- `+events: Events`
- `+runner: Runner`
- `+stats: RequestStats`
- `+web_ui: WebUI`
- `+host: str`
- `+reset_stats: bool`
- `+stop_timeout: float`
- `+catch_exceptions: bool`

Phương thức:

- `+create_local_runner()`
- `+create_master_runner()`
- `+create_worker_runner()`
- `+create_web_ui()`

Sơ đồ llop cho thấy Environment quản lý nhiều thuộc tính quan trọng:

- **user_classes: list[type[User]]**

Danh sách User Classes (các lớp User mô tả hành vi người dùng ảo) sẽ được sử dụng trong bài test.

- **shape_class: LoadTestShape**

Lớp LoadTestShape (tùy chọn) dùng để điều khiển load pattern – tức là cách số lượng user thay đổi theo thời gian (tăng/ giảm tải).

- **events: Events**

Tham chiếu đến Events system (hệ thống sự kiện). Đây là nơi các thành phần khác đăng ký lắng nghe các sự kiện như `test_start`, `request_success`, `request_failure`, `spawning_complete`, ...

- **runner: Runner**

Đối tượng Runner (có thể là LocalRunner, MasterRunner hoặc WorkerRunner) gắn với Environment hiện tại, chịu trách nhiệm thực thi bài test dựa trên các User và Task đã định nghĩa.

- **stats: RequestStats**

Đối tượng thu thập số liệu RequestStats – thành phần lỗi của hệ thống kê, thu thập và xử lý số liệu về request (response time, số lượng, throughput...).

- **web_ui: WebUI**

Tham chiếu tới WebUI (giao diện web dựa trên Flask) nếu bài test được chạy với chế độ có UI. Nếu chạy headless, thuộc tính này có thể là None.

- **host: str**

Giá trị host mặc định của bài test (ví dụ: URL hệ thống đích). Thuộc tính này có thể được thiết lập ở mức Environment và/hoặc trong từng User.

- **reset_stats: bool**

Có cho phép reset thống kê trước khi bắt đầu đo chính thức (ví dụ: bỏ qua số liệu trong giai đoạn warm-up).

- **stop_timeout: float**

Thời gian tối đa (timeout) mà Runner chờ các user tự dừng (chạy xong vòng lặp hiện tại, gọi `on_stop()`) trước khi ép dừng hẳn bài test.

- **catch_exceptions: bool**

Cho phép Environment quyết định có bắt (catch) ngoại lệ trong quá trình

thực thi task hay không. Khi bật, lỗi trong task sẽ được ghi nhận vào thông kê thay vì làm crash toàn bộ bài test.

Nhờ tập trung tất cả thông tin cấu hình và trạng thái như trên, Environment giúp việc điều phối bài test trở nên rõ ràng, tách biệt với phần định nghĩa hành vi người dùng trong locustfile.py.

b. Các phương thức tạo Runner và Web UI

Bên cạnh các thuộc tính, Environment còn cung cấp một số *factory methods* để tạo các thành phần thực thi chính:

- **create_local_runner()**

Tạo một LocalRunner cho các bài test trên một máy/tiến trình duy nhất.

- **create_master_runner()**

Tạo MasterRunner khi chạy ở chế độ distributed load testing với mô hình master-worker. Master sẽ không tạo user mà chỉ điều phối và tổng hợp thống kê.

- **create_worker_runner()**

Tạo WorkerRunner trên worker node, nhận lệnh từ master và trực tiếp spawn user, thực thi task.

- **create_web_ui()**

Khởi tạo WebUI (Flask app + API routes + HTML templates) phục vụ việc điều khiển và giám sát test qua trình duyệt.

Việc gom logic tạo Runner và Web UI vào trong Environment giúp cho main.py chỉ cần:

1. Tạo Environment,
2. Gọi đúng phương thức create_*_runner(),
3. (Tùy chọn) Gọi create_web_ui().

mà không cần tự tay lắp ghép các đối tượng con.

c. Ví dụ tạo Environment trong main.py

Đoạn code dưới đây minh họa cách Environment được khởi tạo trong main.py:

```

1     environment = create_environment(
2         user_classes,
3         options,
4         events=locust_events,
5         shape_class=shape_class,
6         locustfile=locustfile_path,
7         parsed_locustfiles=locustfiles,
8         available_user_classes=available_user_classes,
9         available_shape_classes=available_shape_classes,
10        available_user_tasks=available_user_tasks,
11    )

```

Từ environment này, chương trình sẽ tiếp tục gọi:

- `environment.create_local_runner()` hoặc
- `environment.create_master_runner()` hoặc
- `environment.create_worker_runner()`

tùy theo mô hình sử dụng, đồng thời tạo thêm `web_ui` nếu người dùng không chạy headless.

Như vậy, trong khối Command & Control, Environment chính là **điểm trung tâm** nối kết giữa phần cấu hình/định nghĩa test (`locustfile.py`) và phần thực thi (Runner, WebUI, RequestStats, Events).

2.4.1.2 Runner System – Hệ thống Runner

Hệ thống Runner trong Locust chịu trách nhiệm điều phối toàn bộ quá trình thực thi bài test bằng cách quản lý vòng đời của các test user. Runner cung cấp nhiều implementation khác nhau cho các kịch bản chạy local hoặc phân tán, xử lý logic spawn/stop user, đồng thời giám sát tài nguyên hệ thống.

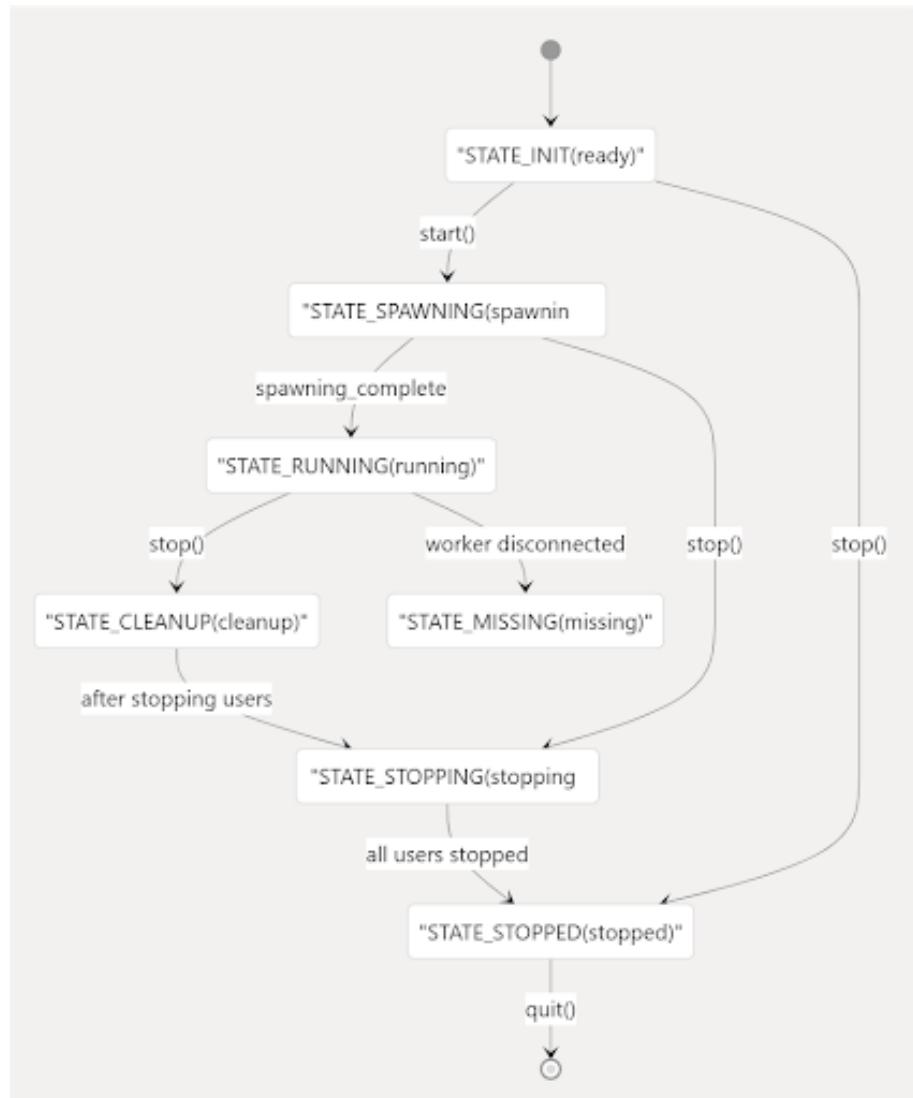
a) Kiến trúc tổng quan của Runner Kiến trúc Runner có dạng phân cấp như hình dưới đây:



Hình 2.2: Kiến trúc tổng quan hệ thống Runner

- Lớp trừu tượng Runner là nền tảng chung cho mọi loại runner.
- Từ Runner tách ra:
 - **LocalRunner** – dùng cho kiểm thử local, một tiến trình duy nhất.
 - **DistributedRunner** – lớp trừu tượng cho các runner phân tán:
 - * **MasterRunner** – chạy trên master node, điều phối worker.
 - * **WorkerRunner** – chạy trên worker node, thực thi user theo lệnh master.

Mọi runner đều duy trì một trạng thái thực thi (*runner state*) như:



Hình 2.3: Sơ đồ state machine của Runner trong Locust

- `STATE_INIT` (ready) – mới khởi tạo, chưa chạy.
- `STATE_SPAWNING` – đang spawn user.
- `STATE_RUNNING` – đang chạy ổn định.
- `STATE_MISSING` – mất kết nối worker.
- `STATE_CLEANUP, STATE_STOPPING, STATE_STOPPED` – các giai đoạn dừng và dọn dẹp.

Việc dùng state machine giúp Runner kiểm soát rõ ràng luồng start/stop, xử lý trường hợp lỗi và ngắt kết nối.

b) Base Runner Class – Lớp Runner cơ sở

Lớp Runner đóng vai trò là lớp trùu tượng nền tảng cho toàn bộ Runner System. Nó định nghĩa các thuộc tính và hành vi chung, sau đó được các lớp con (LocalRunner, MasterRunner, WorkerRunner) kế thừa và triển khai chi tiết.

Chức năng chính của Runner:

1. Quản lý user (User Management)

- Theo dõi các user đang chạy thông qua `user_greenlets` (nhóm greenlet của gevent).
- Cung cấp các hàm:
 - `spawn_users()` – tạo và khởi động các User instances.
 - `stop_users()` – dừng các user đang chạy.
- Theo dõi số lượng user theo từng User Class bằng thuộc tính `user_classes_count`.

2. Vòng đời bài test (Test Lifecycle)

- Duy trì trạng thái bài test trong thuộc tính `state`.
- Cung cấp các phương thức:
 - `start(user_count, spawn_rate, wait, user_classes)` – bắt đầu quá trình spawn user và chạy test (là abstract method, từng runner tự triển khai).
 - `stop()` – dừng toàn bộ user và dọn dẹp.
 - `quit()` – dừng test và huỷ các greenlet của runner.
- Trong quá trình này, Runner sẽ bắn (*fire*) các events tương ứng, ví dụ: `test_start`, `spawning_complete`, `test_stop`.

3. Giám sát tài nguyên (Resource Monitoring)

- Chạy một tác vụ nền `monitor_cpu_and_memory()` để theo dõi CPU usage và memory usage.
- Khi CPU vượt ngưỡng cấu hình, Runner có thể phát cảnh báo, giúp người dùng nhận biết giới hạn máy chạy test.

4. Xử lý lỗi (Error Handling)

- Ghi log các ngoại lệ xảy ra trong quá trình thực thi (ví dụ lỗi trong task, lỗi RPC).
- Duy trì danh sách `exceptions` để có thể thống kê, hiển thị trong Web UI hoặc xuất báo cáo.

c) Các Runner cụ thể

c.1) LocalRunner – Kiểm thử một tiến trình LocalRunner được sử dụng trong kịch bản kiểm thử đơn giản, khi tất cả user chạy trong cùng một tiến trình Python.

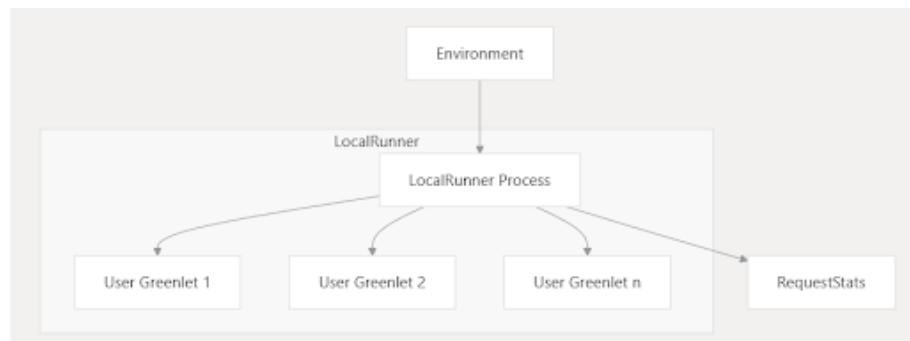
Đặc điểm:

- Cài đặt đơn giản, không có overhead mạng vì không phải giao tiếp master-worker.
- Toàn bộ User Greenlets nằm trong một process, được quản lý trực tiếp bởi LocalRunner.
- Giới hạn bởi tài nguyên (CPU/RAM) của một máy duy nhất.

Implementation tổng quát:

- Hàm khởi tạo `__init__(environment)` gán Environment, tạo client ID dựa trên hostname + UUID.
- Phương thức `start()` gọi `_start()` nội bộ để:
 - tính toán phân phối user theo từng User Class,
 - tạo greenlet spawn và lần lượt tạo User instance,
 - thêm chúng vào nhóm `user_greenlets`.
- Việc “gửi message” giữa các thành phần trong LocalRunner chủ yếu là gọi hàm trực tiếp, mô phỏng hành vi message-passing nhưng không cần RPC thực.

Trong cơ chế hoạt động, LocalRunner nhận Environment, tạo LocalRunner Process quản lý các User Greenlet và cập nhật số liệu vào RequestStats.



c.2) DistributedRunner – Lớp trùu tượng cho runner phân tán [DistributedRunner](#) là một lớp cơ sở nhỏ dành cho các runner chạy trong môi trường distributed load testing.

Nhiệm vụ chính:

- Thiết lập các event listener cho thống kê phân tán (*distributed statistics*),
- Chuẩn bị hạ tầng để MasterRunner và WorkerRunner có thể chia sẻ số liệu qua RPC.

Bản thân [DistributedRunner](#) không chạy trực tiếp test, mà chỉ cung cấp phần “xương sống” cho hai runner con.

c.3) MasterRunner – Điều phối test phân tán [MasterRunner](#) là runner chạy trên *master node* trong mô hình master–worker.

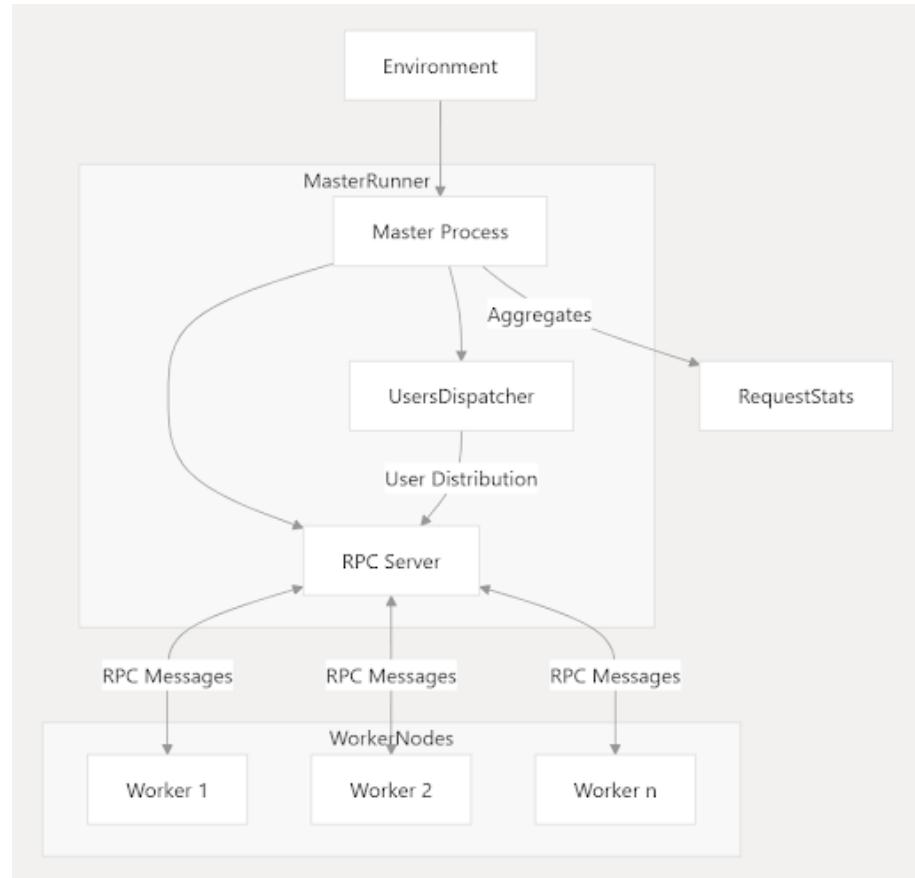
Đặc điểm:

- Điều phối nhiều worker node thông qua cơ chế RPC dựa trên ZeroMQ (ZMQ).
- Phân phối user cho từng worker dựa trên chiến lược *user distribution* (thường là chia đều hoặc theo trọng số).
- Tổng hợp (aggregate) thống kê từ tất cả worker và cập nhật vào RequestStats của *master*.
- Quản lý kết nối và heartbeat của các worker để phát hiện worker mất kết nối ([STATE_MISSING](#)).

Implementation chính:

- Khi khởi tạo, MasterRunner:
 - bind một `rpc.Server` vào `master_bind_host + master_bind_port`,
 - khởi tạo collection `WorkerNodes` để lưu thông tin các worker.
- MasterRunner:
 - lắng nghe `message` từ worker (qua `recv_from_client()`),
 - gửi lệnh như `spawn_users`, `stop` cho worker (qua `send_to_client()`).
- Định kỳ nhận `stats`, `heartbeat` để cập nhật trạng thái worker.
- Mỗi worker được gán một *worker index* phục vụ định danh và phân phối tải.

Trong sơ đồ, MasterRunner gồm Master Process, UsersDispatcher (phân phối user), RPC Server giao tiếp với các WorkerNodes, và đường thống kê tổng hợp về RequestStats.



c.4) WorkerRunner – Thực thi user trên worker node WorkerRunner chạy trên worker node và kết nối tới MasterRunner.

Đặc điểm:

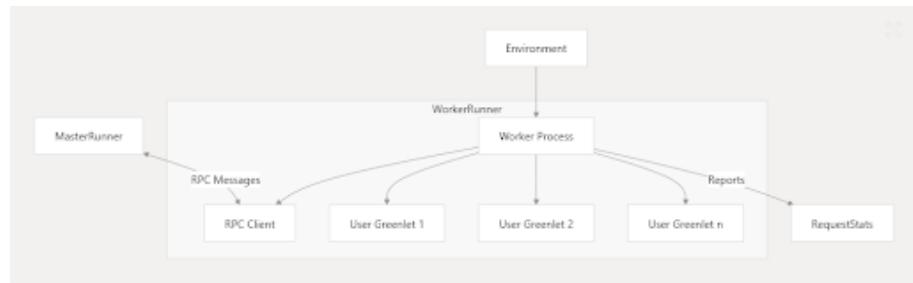
- Nhận lệnh spawn/stop từ MasterRunner thông qua `rpc.Client`.
- Tự tạo User instances, quản lý User Greenlet 1..n tương tự LocalRunner.
- Định kỳ gửi thông kê (stats) và heartbeat về master.
- Cập nhật trạng thái để master có thể phân loại (ready, spawning, running, missing).

Implementation chính:

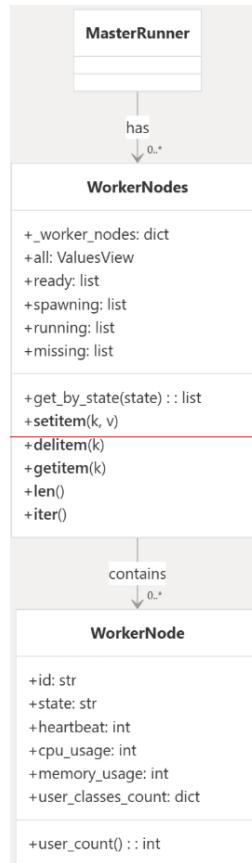
- WorkerRunner khởi tạo bằng `__init__(environment, master_host, master_port)`:
 - tạo `rpc.Client` và kết nối tới RPC server của master,

- gửi message `client_ready` để thông báo đã sẵn sàng.
- Đăng ký message handlers cho các lệnh từ master:
 - lệnh spawn → gọi `spawn_users()`,
 - lệnh stop → gọi `stop()` và dọn dẹp user.
- Trong vòng lặp chính, WorkerRunner:
 - gửi message `stats` chứa số liệu hiện tại,
 - gửi `heartbeat` ở chu kỳ `HEARTBEAT_INTERVAL` để báo vẫn còn sống.

Trong sơ đồ, WorkerRunner, WorkerProcess nhận Environment, tạo User Greenlets, gửi báo cáo sang RequestStats và giao tiếp với MasterRunner qua RPC.



d) Worker Node Management – Quản lý tập các Worker



Trên phía master, việc quản lý các worker được đóng gói trong hai lớp:

1. **WorkerNode** – biểu diễn một worker cụ thể, lưu:

- `id`: định danh worker,
- `state`: trạng thái hiện tại (ready, spawning, running, missing, ...),
- `heartbeat`: bộ đếm dùng để đánh giá worker còn phản hồi hay không,
- `cpu_usage`, `memory_usage`: thông tin sử dụng tài nguyên của worker,
- `user_classes_count`: số user đang chạy theo từng User Class, kèm phương thức `user_count()` tính tổng.

2. **WorkerNodes** – collection quản lý nhiều WorkerNode:

- nội bộ dùng `_worker_nodes`: dict ánh xạ `id` → `WorkerNode`,
- hỗ trợ các phép toán giống dictionary (`__getitem__`, `__setitem__`, `__delitem__`, `__len__`, `__iter__`),
- cung cấp các phương thức/tính năng tiện ích:
 - `get_by_state(state)` – lọc các worker theo trạng thái,

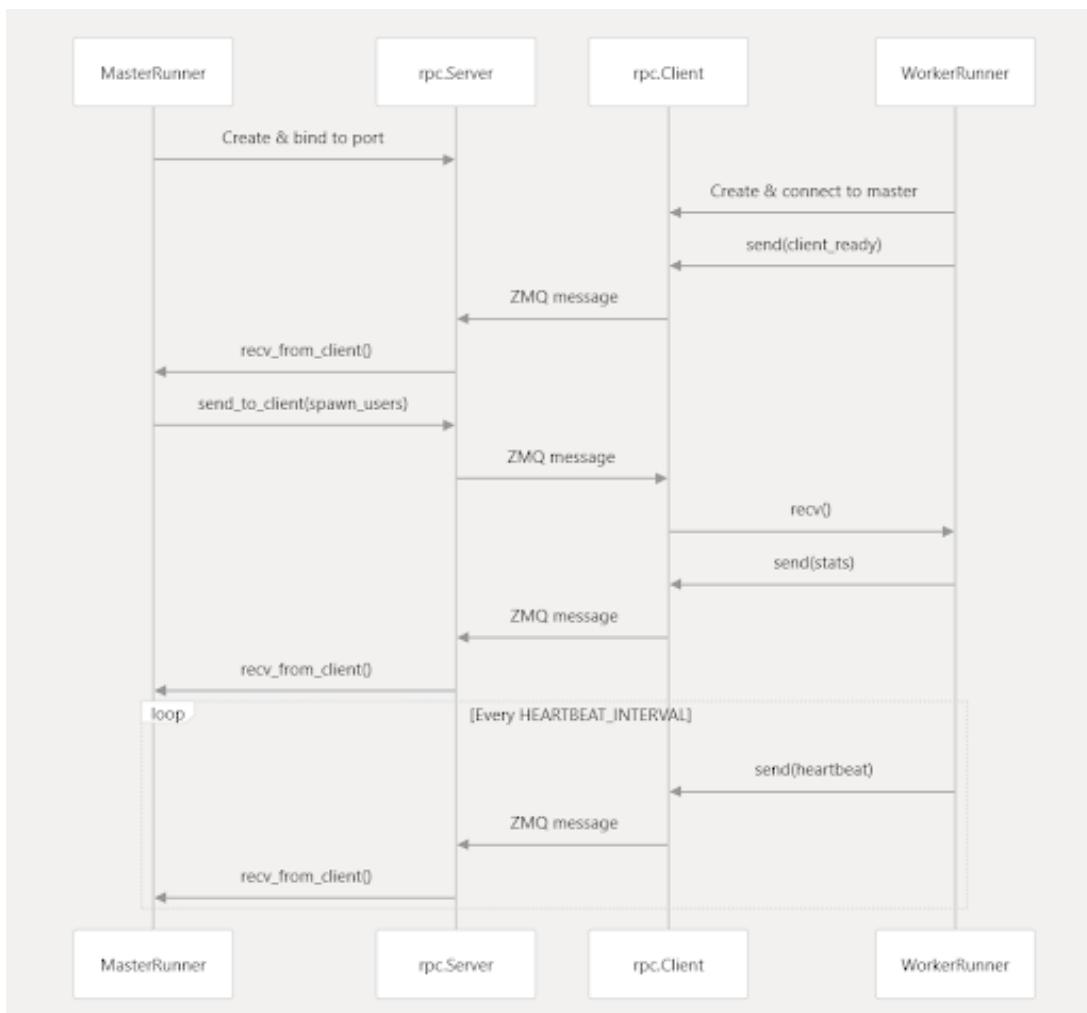
- **ready**, **spawning**, **running**, **missing** – trả về danh sách worker theo từng trạng thái.

Nhờ đó, MasterRunner có thể dễ dàng:

- xác định bao nhiêu worker “ready” trước khi bắt đầu test,
- phân phối user chỉ cho các worker “running”,
- phát hiện worker “missing” do mất heartbeat.

e) Communication Mechanism – Cơ chế giao tiếp phân tán

Trong chế độ phân tán, **MasterRunner** và **WorkerRunner** giao tiếp thông qua một hệ thống RPC (Remote Procedure Call) dựa trên ZeroMQ.



Hình 2.4: Cơ chế giao tiếp RPC giữa MasterRunner và WorkerRunner

Các thành phần chính:

- **Message class**: đóng gói thông tin message, gồm:

- **msg_type** (kiểu message: `spawn`, `stats`, `heartbeat`, `quit`, ...),
 - **payload** dữ liệu,
 - **node_id** (id của worker).
- **rpc.Server**: chạy ở phía master, sử dụng socket `ZMQ_ROUTER`, chịu trách nhiệm:
 - lắng nghe message từ các worker,
 - chuyển message đến **MasterRunner**.
 - **rpc.Client**: chạy ở phía worker, dùng socket `ZMQ DEALER`, gửi:
 - `client_ready`,
 - `stats`,
 - `heartbeat`,
 - ...đến master.

Đặc điểm implementation:

- Message được serialize bằng **msgpack**, tối ưu cho tốc độ và kích thước.
- Thiết lập TCP **keepalive** để tăng độ tin cậy kết nối dài.
- Có cơ chế **retry** và xử lý ngoại lệ (network errors, timeout) để tránh làm crash toàn bộ runner.

Sơ đồ trình tự cho thấy: **MasterRunner** tạo và bind `rpc.Server`, **WorkerRunner** tạo `rpc.Client` và `connect`, sau đó hai bên trao đổi `ZMQ message` liên tục trong suốt vòng đờì bài test.

f) User Spawning and Stopping – Quản lý vòng đờì user

Một nhiệm vụ quan trọng của Runner System là điều khiển vòng đờì của user: spawn đúng số lượng với tốc độ mong muốn, và dừng lại một cách an toàn.



Quy trình spawn user

1. Gọi `start(user_count, spawn_rate, wait, user_classes)` trên runner.
2. Runner sử dụng **UsersDispatcher** để tính phân phối user theo User Class (ví dụ 70% HttpUser A, 30% HttpUser B).
3. Tạo một *spawning greenlet* riêng phụ trách logic spawn bất đồng bộ.

4. Trong spawning greenlet:

- gọi `spawn_users()` → tạo từng User instance,
- gọi `user.start()` (hoặc tương đương) để bắt đầu vòng lặp task,
- thêm greenlet của user vào `user_greenlets`.

Quy trình stop user

1. Khi `stop()` được gọi:

- Runner ghi lại `final_user_classes_count` để lưu trạng thái cuối.

2. Nếu vẫn còn spawning greenlet đang chạy, runner kill greenlet này.

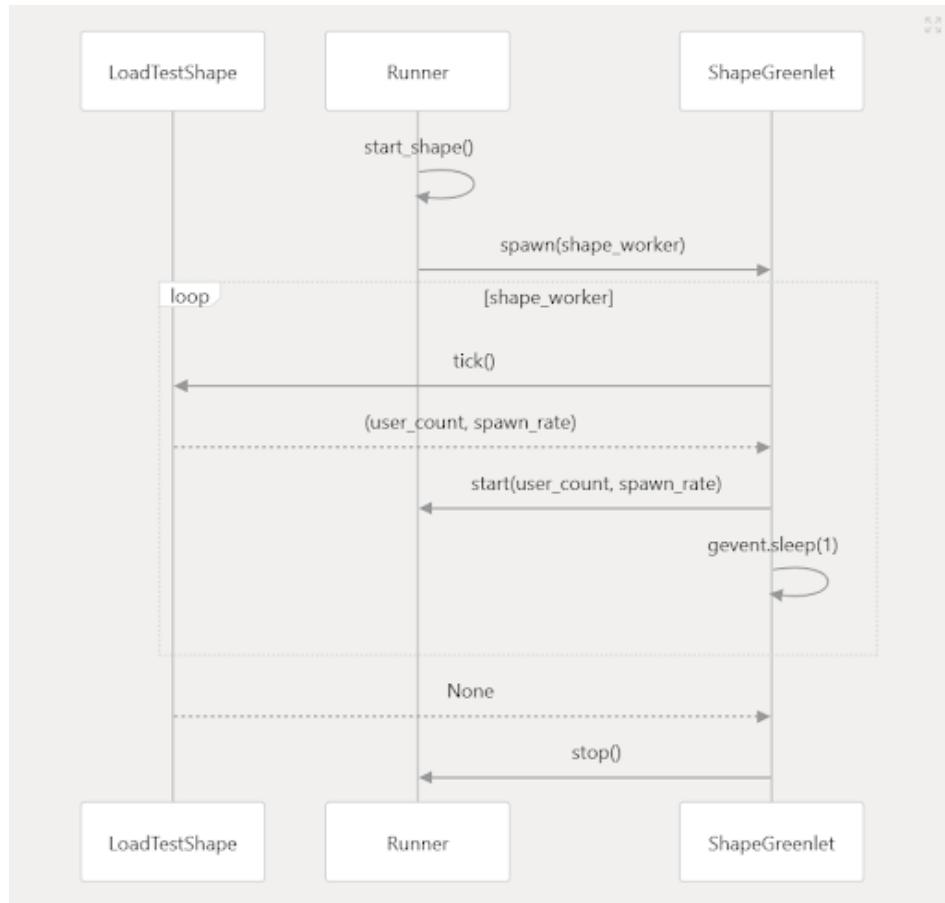
3. Gọi `stop_users()` để lần lượt dừng tất cả user instance:

- gọi `user.stop()` / set flag để dừng vòng lặp task,
- chờ user kết thúc, sau đó remove khỏi `user_greenlets`.

Cách tổ chức spawn/stop theo hai “process” logic riêng (spawning process, stop_users process) giúp runner xử lý quá trình thay đổi tải mượt mà, tránh việc tạo/dừng user đột ngột gây nhiều kết quả test.

g) Shape-based Load Testing – Kiểm thử theo đường cong tải

Runner System hỗ trợ **shape-based load testing** thông qua integration với các lớp `LoadTestShape`.



Hình 2.5: Luồng xử lý Shape-based Load Testing

Cơ chế:

- Gọi `start_shape()` trên Runner:
 - Runner spawn một greenlet `shape_worker`.
- Trong `shape_worker`:
 - vòng lặp định kỳ (thường mỗi giây) gọi `shape_class.tick()`,
 - `tick()` trả về một cặp (`user_count, spawn_rate`) mong muốn, hoặc `None` nếu muốn kết thúc test.

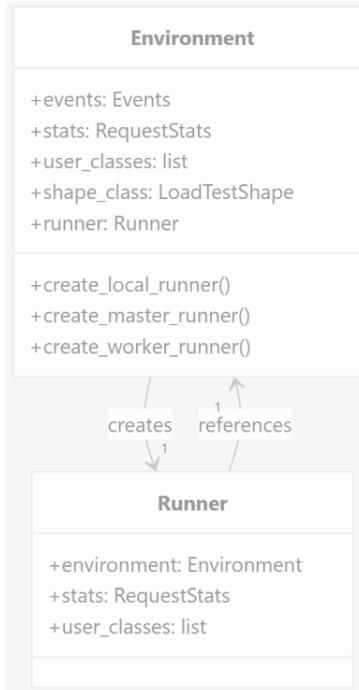
Tùy cặp dữ liệu tick:

- Runner so sánh với số user hiện tại và điều chỉnh bằng cách gọi `start()`/`spawn_users()` hoặc `stop_users()` tương ứng.
- Khi `tick()` trả về `None`, Runner gọi `stop()` để dừng toàn bộ test.

Nhờ `LoadTestShape`, người dùng có thể mô tả các kịch bản phức tạp như: ramp-up, ramp-down, sống tại, spike test... bằng Python, trong khi Runner đảm nhận phần thực thi chi tiết.

h) Environment Integration – Tích hợp với Environment

Runner System gắn bó chặt chẽ với Environment:



- Mỗi Runner đều giữ tham chiếu đến:
 - **environment**: Environment,
 - **stats**: RequestStats từ environment,
 - **user_classes**: list do environment cung cấp.
- Ngược lại, Environment:
 - cung cấp các **factory method**:
 - * **create_local_runner()**,
 - * **create_master_runner()**,
 - * **create_worker_runner()**,
 - lưu lại tham chiếu runner hiện tại để các thành phần khác (Web UI, extensions) có thể truy cập.

Nhờ đó, phần **Command & Control** trong main.py chỉ cần:

```
1     env = create_environment(...)  
2     if options.master:  
3         runner = env.create_master_runner(...)  
4     elif options.worker:  
5         runner = env.create_worker_runner(...)  
6     else:  
7         runner = env.create_local_runner()
```

Toàn bộ chi tiết cấu hình và liên kết bên trong đã được Environment và Runner System xử lý.

i) Command-line Integration – Tích hợp với giao diện dòng lệnh

Runner System được cấu hình thông qua **command-line interface** của Locust, giúp người dùng dễ dàng lựa chọn chế độ chạy.

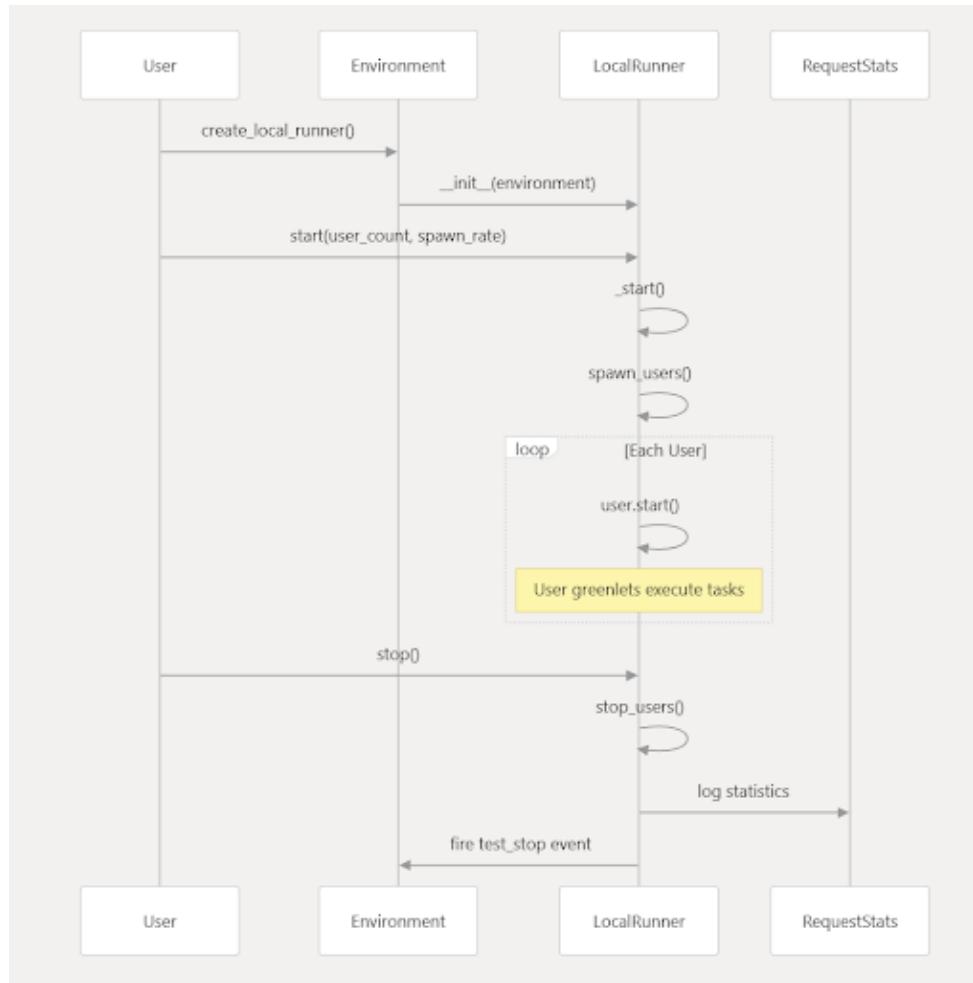
Một số tùy chọn quan trọng:

- `--master` – chạy tiến trình hiện tại như **master node**.
- `--worker` – chạy như **worker node**.
- `--master-bind-host`, `--master-bind-port` – địa chỉ/port master lắng nghe worker.
- `--master-host`, `--master-port` – địa chỉ/port worker dùng để kết nối tới master.
- `--expect-workers`, `--expect-workers-max-wait` – số lượng worker mong đợi và thời gian chờ tối đa trước khi bắt đầu test.
- `--headless` – chạy không cần Web UI (thường dùng trong CI/CD).
- `--users / -u` – tổng số user mục tiêu.
- `--spawn-rate / -r` – tốc độ spawn user (user/s).
- `--run-time / -t` – giới hạn thời gian chạy test.

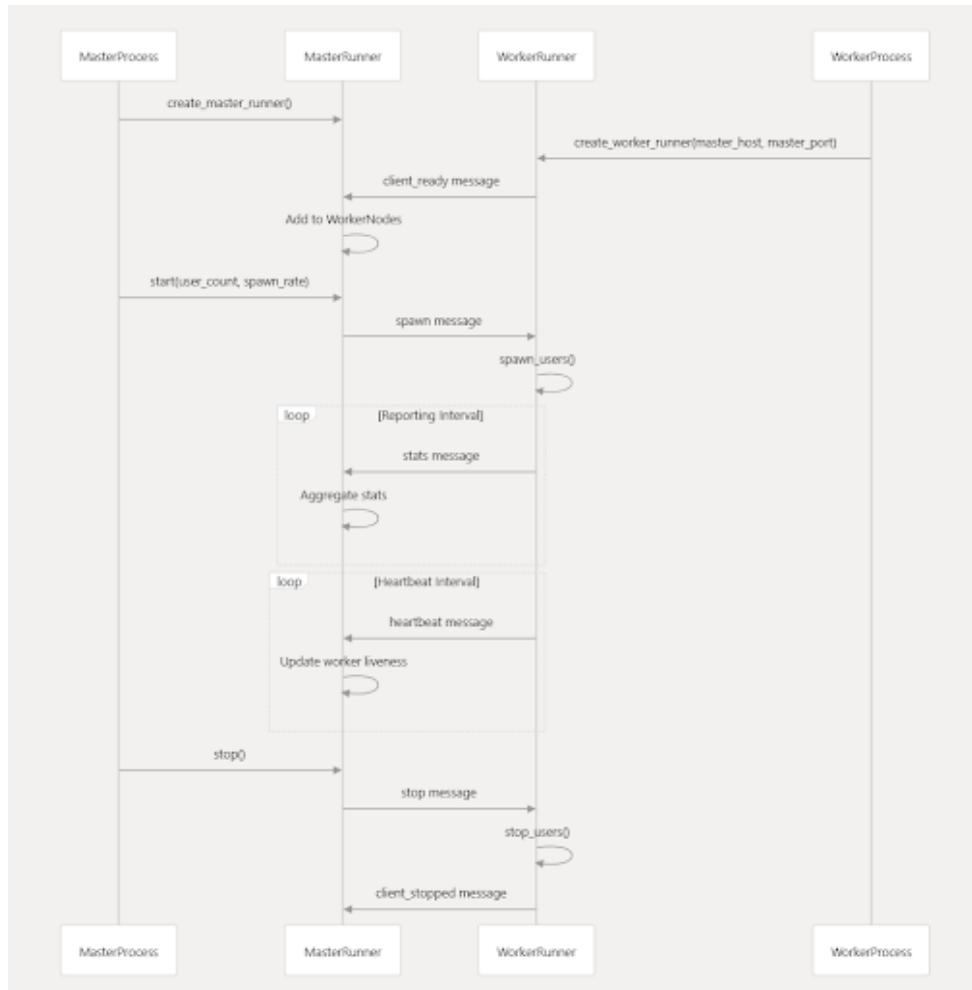
Các option này được parse trong `argument_parser.py` và truyền vào Environment/Runner đảm bảo giao diện cấu hình thống nhất cho người dùng.

j) Luồng thực hiện

1. Local testing workflow



2. Distributed testing workflow



k) Tổng kết phần Runner System

Tóm lại, **Runner System** cung cấp kiến trúc linh hoạt và mở rộng cho việc thực thi bài test trong Locust:

1. Lớp trừu tượng **Runner** định nghĩa lõi cho việc quản lý vòng đời bài test và *User instances*.
2. **LocalRunner** cung cấp implementation đơn giản cho kiểm thử *single-process*.
3. **MasterRunner** và **WorkerRunner** triển khai mô hình *distributed load testing* trên nhiều tiến trình/máy khác nhau.
4. Hệ thống hỗ trợ **spawn/stop user động** và **shape-based load testing** thông qua `LoadTestShape`.
5. Tích hợp chặt chẽ với `Environment`, `RequestStats`, `Events` và `Web UI` để chia sẻ ngữ cảnh và số liệu.

6. Cơ chế ZeroMQ-based RPC đảm bảo giao tiếp tin cậy trong môi trường phân tán.

Nhờ thiết kế này, Locust có thể mở rộng từ các bài test nhỏ trên một máy đến các kịch bản load test phân tán quy mô lớn, trong khi vẫn giữ được API và hành vi nhất quán cho người dùng.

2.4.2 Test Definition

2.4.2.1 Tổng quan và cấu trúc của User Classes và Tasks

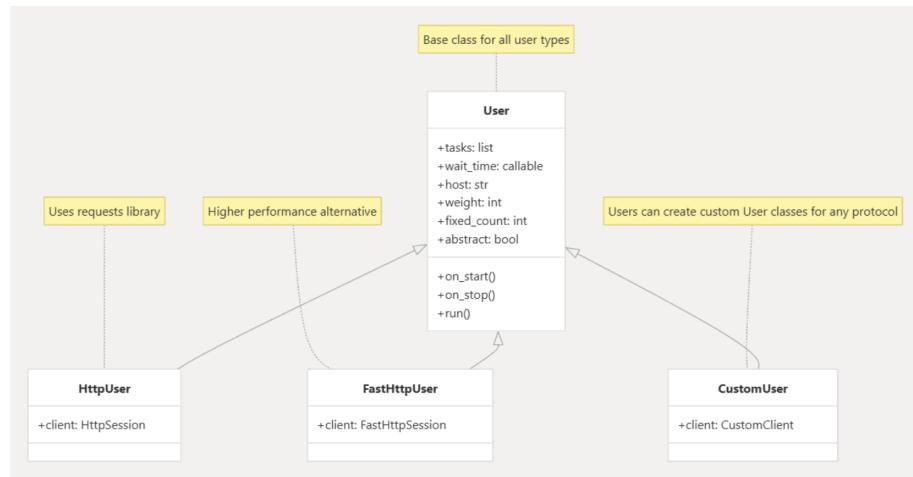
a) Mục đích và phạm vi

Hệ thống **User Classes** và **Tasks** là trung tâm cách Locust định nghĩa kịch bản load test.

- **User Class** biểu diễn một *virtual user* tương tác với *system under test*.
- **Task** mô tả các hành động mà user đó thực hiện (gửi request, duyệt trang, checkout, ...).

Nhờ kết hợp hai khái niệm này, Locust cho phép mô hình hóa hành vi người dùng một cách linh hoạt và sát thực tế.

b) Phân cấp User Class (User Class Hierarchy)



Hình 2.6: User Class hierarchy trong Locust

Lớp cơ sở User định nghĩa:

- Danh sách tasks sẽ được thực thi.

- Hàm `wait_time` (mô phỏng “think-time”).
- Lifecycle hooks như `on_start()`, `on_stop()`.
- Thuộc tính `weight`, `fixed_count`, `abstract` để điều khiển cách Runner spawn từng loại user.

Các HTTP user chuyên biệt:

- **HttpUser** – dùng thư viện `requests` (gevent-compatible), cung cấp `self.client` dạng `HttpSession`.
- **FastHttpUser** – dùng `geventhttpclient`, tối ưu hiệu năng hơn cho *high-throughput test*.

Ngoài ra, developer có thể xây dựng **CustomUser** với client tùy ý cho các giao thức khác (WebSocket, gRPC, MQ,...), vẫn kế thừa toàn bộ cơ chế task & lifecycle từ User.

Ví dụ:

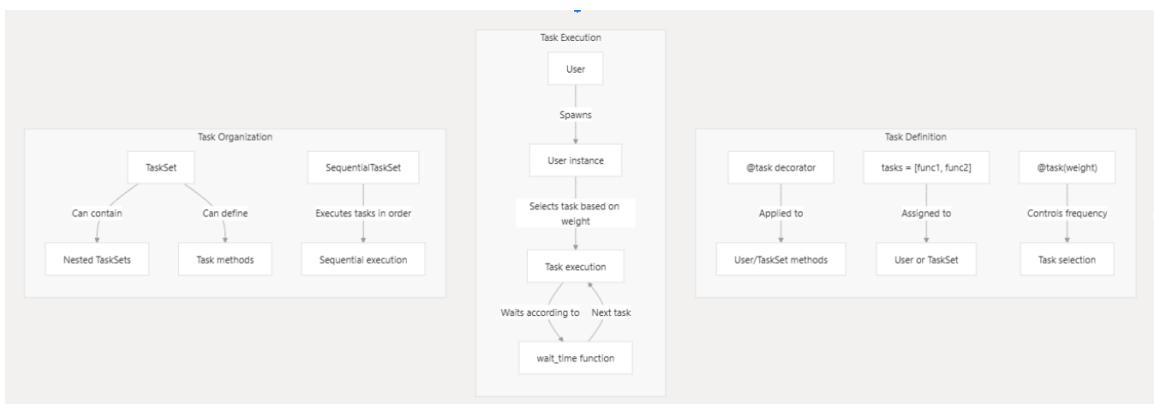
```

1   from locust import HttpUser, task, between
2
3   class WebsiteUser(HttpUser):
4       host = "http://example.com"
5       wait_time = between(2, 5)
6
7       @task
8       def index_page(self):
9           self.client.get("/")

```

c) Định nghĩa Task (Task Definition)

Diagram: Task Definition and Execution Flow



Hình 2.7: Task Definition and Execution Flow

Task trong Locust có thể được định nghĩa theo nhiều cách:

- Decorator @task trên method của User:

```

1     class MyUser(User):
2
3         @task
4         def my_task(self):
5             ...

```

- @task(**weight**) với trọng số để điều khiển tần suất:

```

1     class MyUser(User):
2
3         @task(3)
4         def common_task(self):
5             ...
6
7         @task(1)
8         def rare_task(self):
9             ...

```

- Gán list tasks cho thuộc tính tasks:

```

1     def task1(user):
2         ...
3
4     def task2(user):
5         ...
6
7     class MyUser(User):
8         tasks = [task1, task2]

```

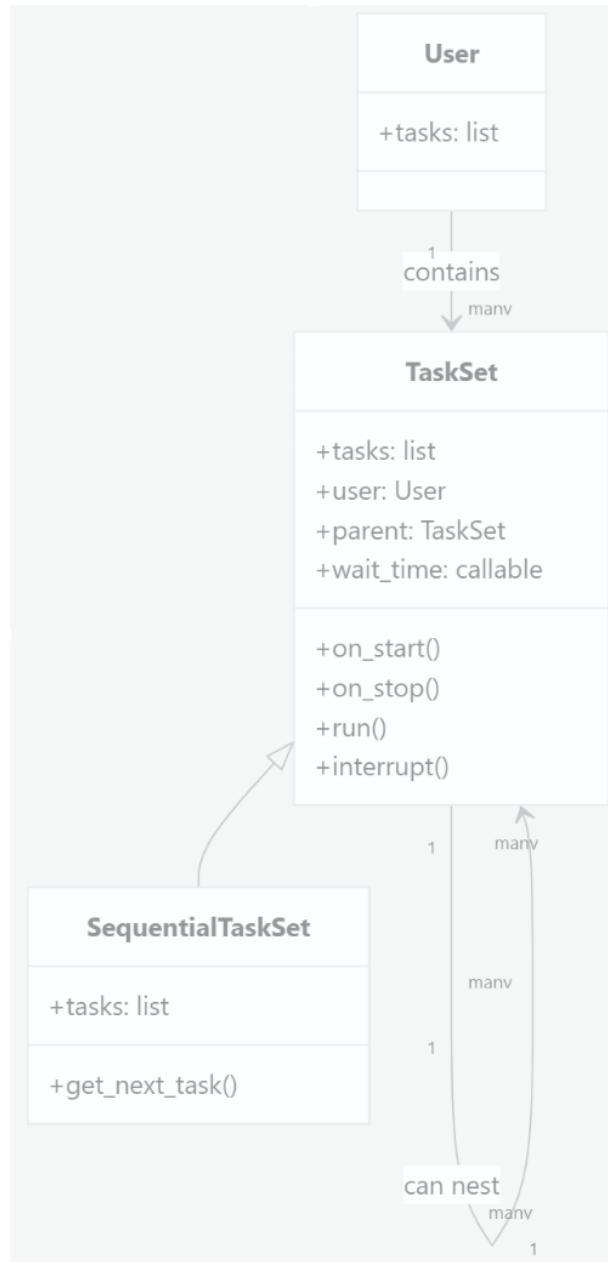
- Gán TaskSet vào tasks để gom các hành vi phức tạp thành một nhóm.

d) Tổ chức Task bằng TaskSet và SequentialTaskSet TaskSets cung cấp một cách để nhóm các nhiệm vụ liên quan và tạo cấu trúc nhiệm vụ phân cấp.

Diagram: TaskSet Hierarchy and Nesting

d) Tổ chức Task bằng TaskSet và SequentialTaskSet TaskSets cung cấp một cách để nhóm các nhiệm vụ liên quan và tạo cấu trúc nhiệm vụ phân cấp.

Diagram: TaskSet Hierarchy and Nesting



TaskSet:

- Chứa các method @task, function task, hoặc TaskSet con (nested TaskSets).
- Có `on_start()`, `on_stop()`, có thể định nghĩa `wait_time` riêng.
- Có thể gọi `self.interrupt()` để dừng TaskSet hiện tại và trả quyền điều khiển về parent.

SequentialTaskSet:

- Thực thi task **theo đúng thứ tự định nghĩa**, không chọn ngẫu nhiên theo weight.
- Phù hợp mô phỏng workflow cố định: `login → add_to_cart → checkout`.

- Có thể `interrupt()` để kết thúc sequence.
-

e) Luồng chọn và thực thi Task (Task Selection & Execution Flow)

Khi Runner spawn một User instance, luồng cơ bản:

1. **User spawning:** Runner tạo instance.
 2. **Khởi tạo:** gọi `on_start()` của User và TaskSet.
 3. **Task loop:**
 - Chọn task:
 - User/TaskSet thường → chọn ngẫu nhiên theo weight,
 - SequentialTaskSet → chạy tuần tự theo thứ tự định nghĩa.
 - Thực thi task; nếu task là TaskSet, quyền điều khiển chuyển vào TaskSet đó.
 - Chờ theo hàm `wait_time`, sau đó lặp lại.
 4. **Dừng user:** khi Runner stop, gọi `on_stop()` để giải phóng tài nguyên.
-

f) Wait Time Functions – Hàm thời gian chờ

`wait_time` là một callable trả về **số giây chờ** giữa hai lần chạy task, giúp mô phỏng think-time.

Một số hàm dụng sẵn:

- `between(min_time, max_time)` – random trong khoảng `[min, max]`.
- `constant(time)` – chờ cố định.
- `constant_pacing(time)` – cố gắng giữ time giữa hai lần bắt đầu task cố định.
- `constant_throughput(tasks_per_sec)` – cố gắng duy trì throughput `tasks/s`.

Ngoài ra có thể dùng lambda/function tùy chỉnh:

```
wait_time = lambda self: random.expovariate(1.0)
```

g) Task Tagging, Lifecycle Hooks và các mẫu sử dụng

Tag task bằng @tag để lọc khi chạy test:

```
class WebUser(User):  
  
    @task  
    @tag("browse")  
    def browse_products(self):  
        ...  
  
    @task  
    @tag("purchase", "critical")  
    def purchase(self):  
        ...
```

Chạy với `-tags browse` hoặc `-exclude-tags critical` để chỉ lấy/bỏ các task có tag tương ứng.

Lifecycle hooks:

- User: `on_start()`, `on_stop()`.
- TaskSet: `on_start()`, `on_stop()` cho từng nhóm task.

Ví dụ login/logout:

```
class WebUser(HttpUser):  
  
    def on_start(self):  
        self.client.post("/login", {"username": "test", "password": "secret"}  
  
    def on_stop(self):  
        self.client.get("/logout")
```

Các pattern thường gặp:

- `HttpUser` cơ bản với nhiều task và trọng số khác nhau (`index_page` vs `about_page`).
- User có `TaskSet` lồng nhau để mô phỏng `browse + purchase` flow.
- `SequentialTaskSet` cho luồng `checkout` cố định.

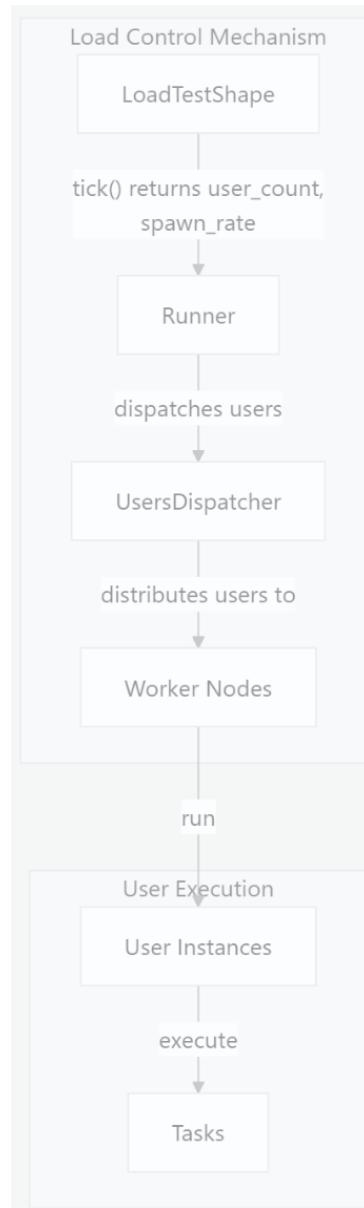
- Dùng multiple `HttpSession` trong một User để test nhiều service.
- Dùng events + Semaphore (`spawning_complete`) để đồng bộ hành vi giữa các user.

2.4.2.2. Load Shaping với LoadTestShape

a) Tổng quan cơ chế Load Shaping

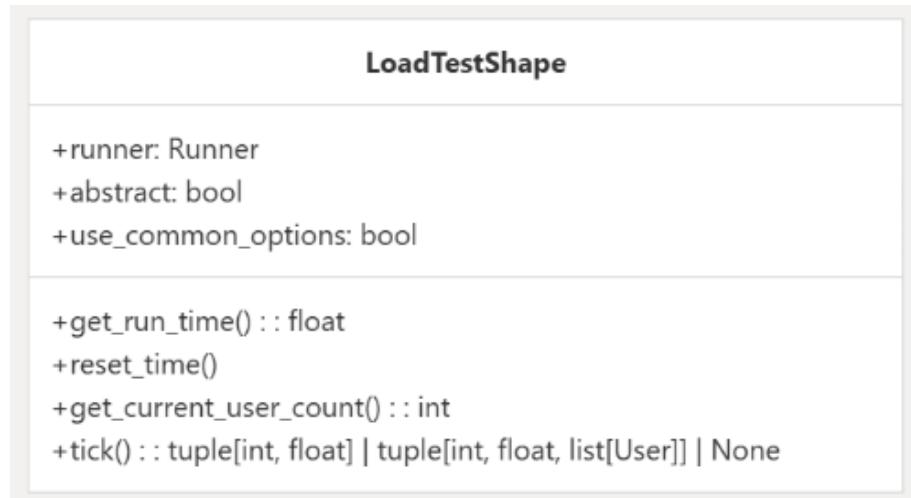
Load shaping điều khiển **tổng số virtual user** và **spawn rate** theo thời gian, để tạo các pattern như ramp-up, steady, spike, wave...

Ở mức kiến trúc: **LoadTestShape** → Runner → **UsersDispatcher** → Worker Nodes → User Instances → Tasks.



b) Lớp LoadTestShape **LoadTestShape** là *abstract base class* mô tả đường cong tải:

- Thuộc tính chính: **runner**, **abstract**, **use_common_options**.
- API quan trọng:
 - **get_run_time()**, **reset_time()**, **get_current_user_count()**.
 - **tick()** → trả về **(user_count, spawn_rate)** hoặc **(user_count, spawn_rate, user_classes)**; trả **None** để kết thúc test.

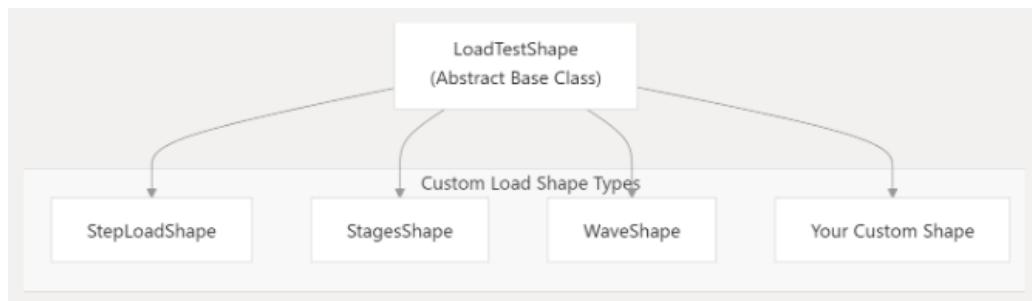


Hình 2.8: Cấu trúc lớp LoadTestShape

c) Các kiểu Load Shape

Locust cung cấp sẵn vài shape điển hình và cho phép định nghĩa custom:

- **StepLoadShape** – tăng tải theo bậc.
- **StagesShape** – nhiều stage, mỗi stage có `user_count` và `duration` riêng.
- **WaveShape** – tải dao động dạng sóng.
- Custom shape – bất kỳ pattern lập trình được bằng Python.



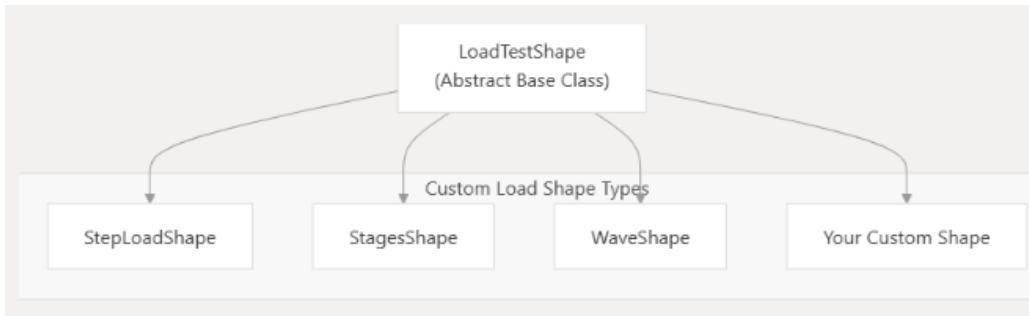
Hình 2.9: LoadTestShape → StepLoadShape / StagesShape / WaveShape / Custom Shape

d) Chu kỳ tick() và tương tác với Runner

Mỗi ~ 1 giây, Locust gọi `tick()`:

1. `tick()` dựa trên `get_run_time()` để tính (`user_count, spawn_rate[, user_classes]`).

2. Runner nhận kết quả và tạo dispatch mới cho **UsersDispatcher** (phân phối cụ thể sang worker – xem mục 2.4.3).



Hình 2.10: `LoadTestShape.tick()` → `Runner.new_dispatch(...)` → `UsersDispatcher`

e) Sử dụng LoadTestShape

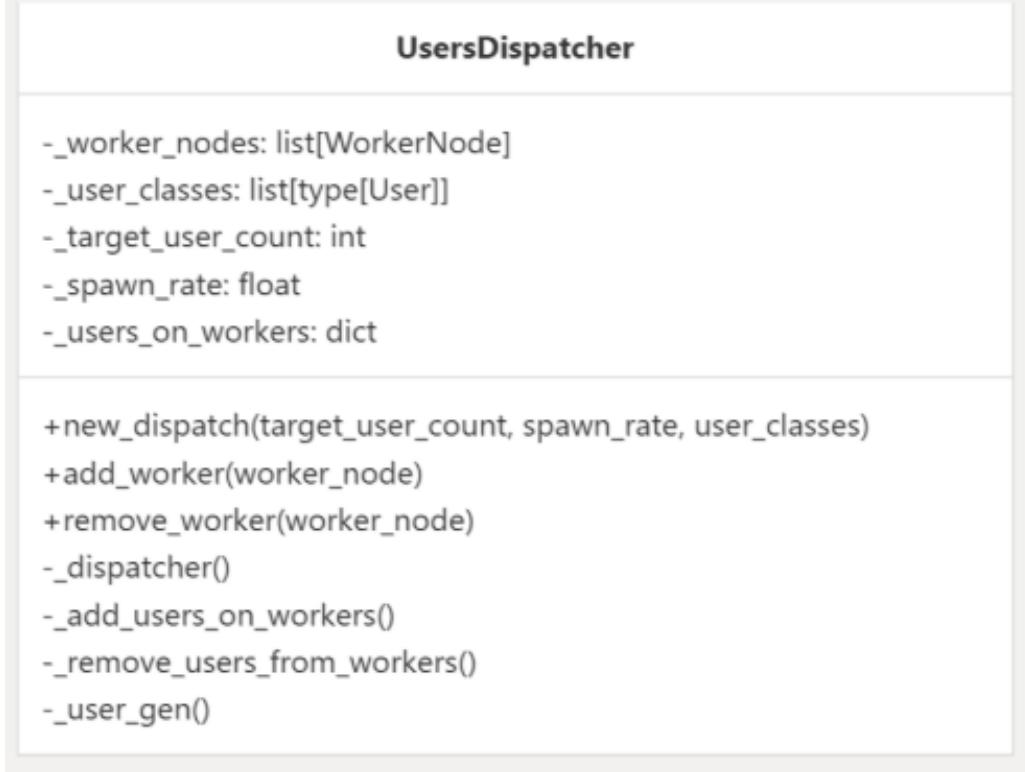
- Tạo class kế thừa `LoadTestShape`, override `tick()`.
- Đặt trong `locustfile.py` hoặc file `.py` được load cùng Locust.
- Khi tìm thấy một subclass không `abstract`, Locust sẽ dùng nó làm load shape chính, thay cho ramp-up mặc định.

2.4.3. Execution

2.4.3.1. User Distribution với UsersDispatcher

a) Vai trò Trong chế độ distributed:

- **LoadTestShape** quyết định tổng `user_count` và `spawn_rate`.
- **UsersDispatcher** quyết định mỗi worker node chạy bao nhiêu user của từng **User Class**, tuân thủ `spawn_rate` và cân bằng tải.



b) Cấu trúc và API chính

Dữ liệu nội bộ:

- `_worker_nodes`: list[WorkerNode] – danh sách worker.
- `_user_classes`: list[type[User]], `_target_user_count`, `_spawn_rate`.
- `_users_on_workers`: dict – map worker → {UserClass: count}.

API:

- `new_dispatch(target_user_count, spawn_rate, user_classes)` – nhận “kế hoạch tải” mới từ Runner.
- `add_worker(worker_node)` / `remove_worker(worker_node)` – cập nhật danh sách worker và trigger rebalance.

c) Dispatch cycle và spawn rate

UsersDispatcher chạy theo **dispatch cycle**:

- Mỗi cycle: từ trạng thái user hiện tại đến trạng thái mục tiêu (ví dụ 0 → 100 user).

- Cycle được chia thành nhiều **dispatch iteration**; mỗi iteration spawn/stop một batch user để đạt spawn rate mong muốn; giữa các iteration có khoảng chờ tương ứng.

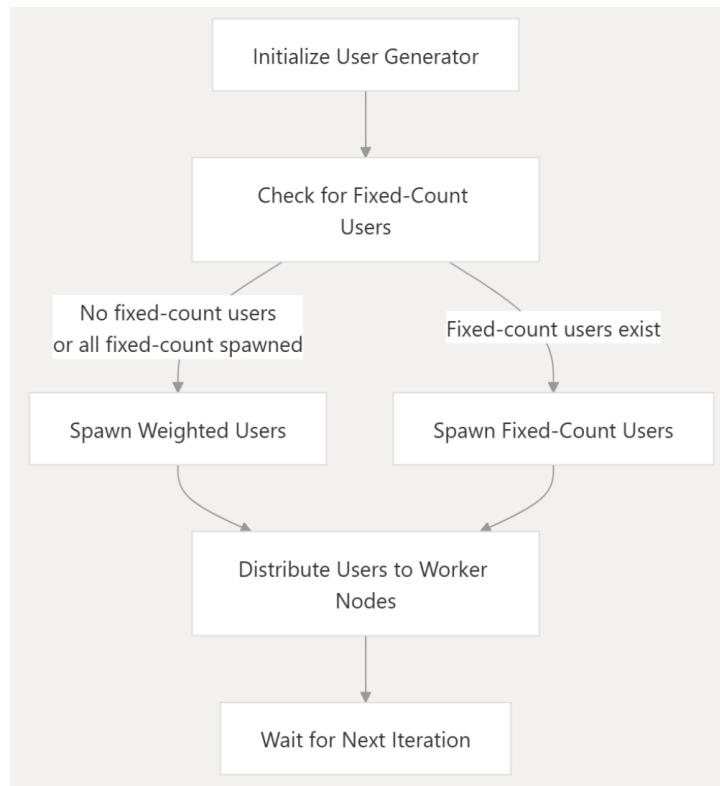


d) Thuật toán phân phối: Fixed-count vs Weighted users

Locust chia user thành hai nhóm:

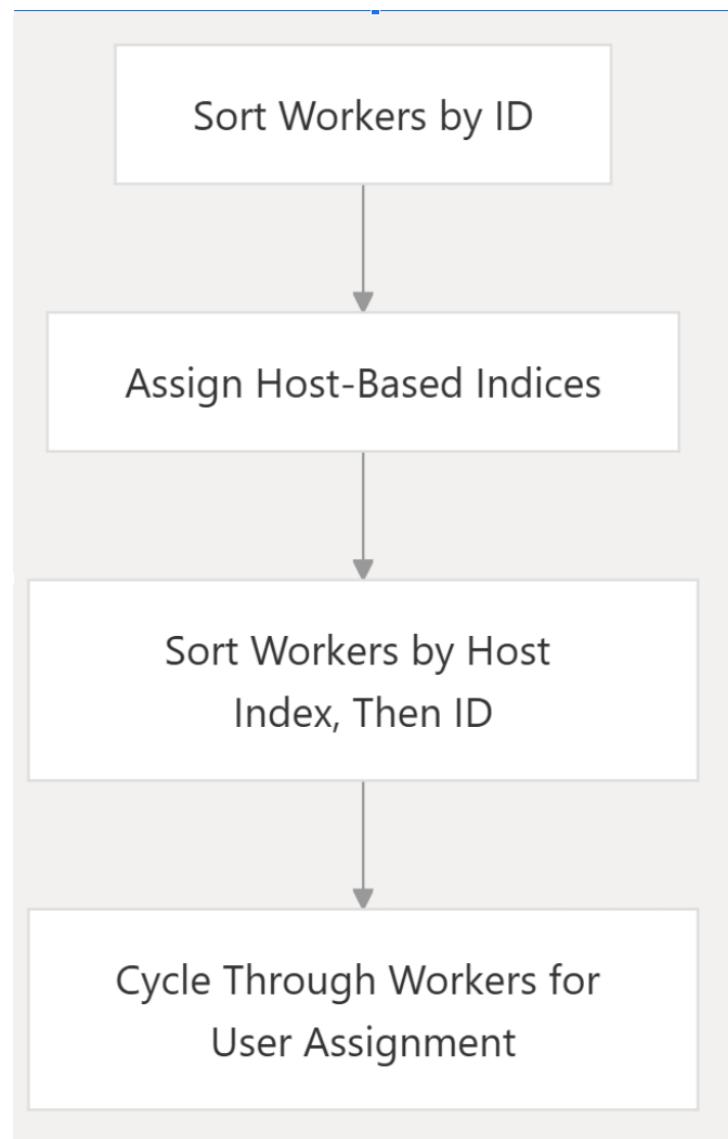
1. **Fixed-count users** (`fixed_count > 0`) – cần số lượng chính xác → spawn ưu tiên trước.
2. **Weighted users** – phần user còn lại được chia theo `weight` của từng User Class.

`UsersDispatcher` dùng generator `_user_gen()` để lần lượt chọn user từ hai nhóm này, sau đó phân bổ cho worker. Thuật toán bên trong sử dụng **Kullback–Leibler divergence** để làm cho phân bổ thực tế gần với phân bổ mong muốn.



e) **Chiến lược chọn worker** Chiến lược phân công user cho worker:

1. Sort worker theo **ID** (đảm bảo determinism).
2. Gán **host index** cho các worker cùng host.
3. Sort theo (**host_index**, **worker_id**) để ưu tiên dàn đều giữa các host.
4. “Cycle through” danh sách đó để gán từng user.



2.5 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.

2.6 Ví dụ cấu hình kiểm thử

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

Ví dụ 2.1: Ví dụ kiểm thử tải website bằng Locust

```

1   from locust import HttpUser, task, between
2
3   class WebsiteUser(HttpUser):
4       wait_time = between(1, 3) # Thời gian chờ giữa các tác vụ (giá trị
5                               # ường dùng là 1)
6
7       @task(2)
8       def view_homepage(self):
9           self.client.get("/") # Truy cập trang chủ
10
11      @task(1)
12      def view_product(self):
13          self.client.get("/product/1") # Xem chi tiết sản phẩm

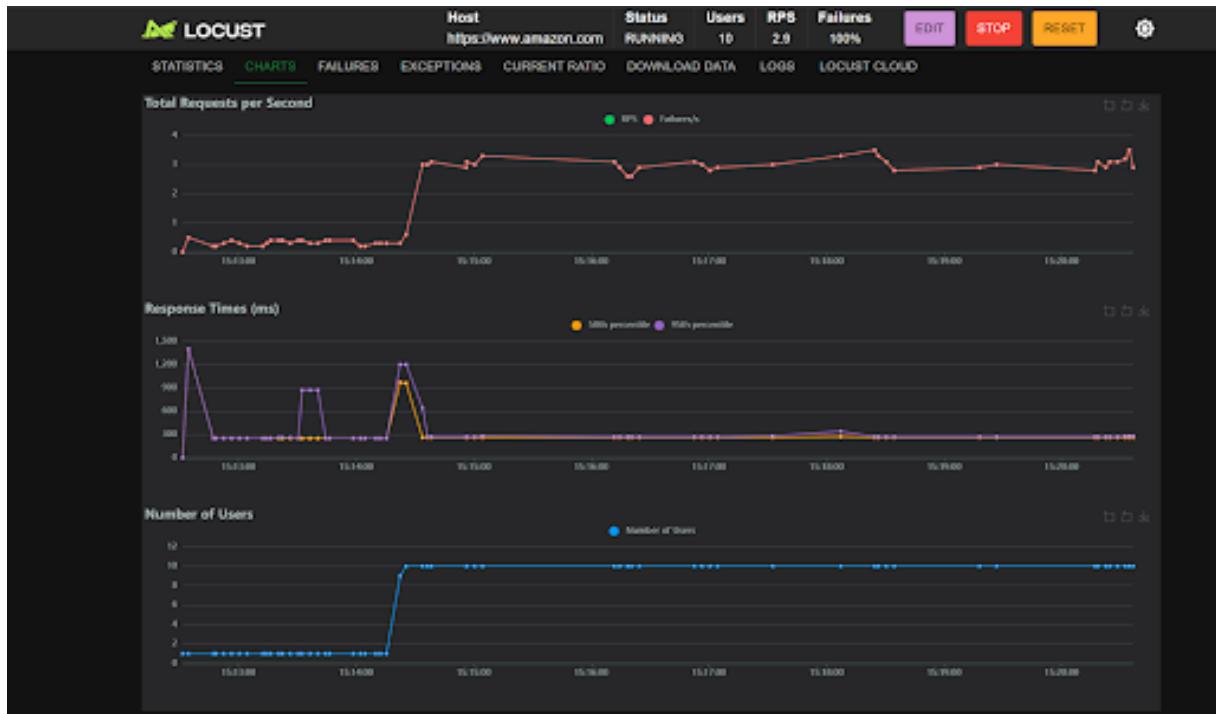
```

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.

2.7 Báo cáo và phân tích



Hình 2.11: Giao diện biểu đồ tải theo thời gian thực của Locust

Sau khi kiểm thử, Locust cung cấp **báo cáo chi tiết** về hiệu năng:

- **Thống kê thời gian thực:**

Hiển thị số lượng request, thời gian phản hồi trung bình, tỉ lệ lỗi, độ lệch chuẩn và các phân vị (percentiles 50%, 95%, 99%).

- **Xuất dữ liệu:**

Locust có thể xuất kết quả ra file CSV để phân tích sau:

```
1 locust -f locustfile.py --csv=result
```

- **Biểu đồ và đồ thị:**

Locust cung cấp các biểu đồ tải theo thời gian, giúp dễ dàng quan sát khi nào hệ thống bắt đầu chậm hoặc lỗi xuất hiện.

- **Kết quả định lượng:**

Thông qua các chỉ số như *Average Response Time*, *Requests per Second*, *Failure Rate*, người kiểm thử có thể xác định **ngưỡng chịu tải tối ưu** và **vùng rủi ro hiệu năng**.

3. Đánh giá công cụ

(Phần 3 trong bộ 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.

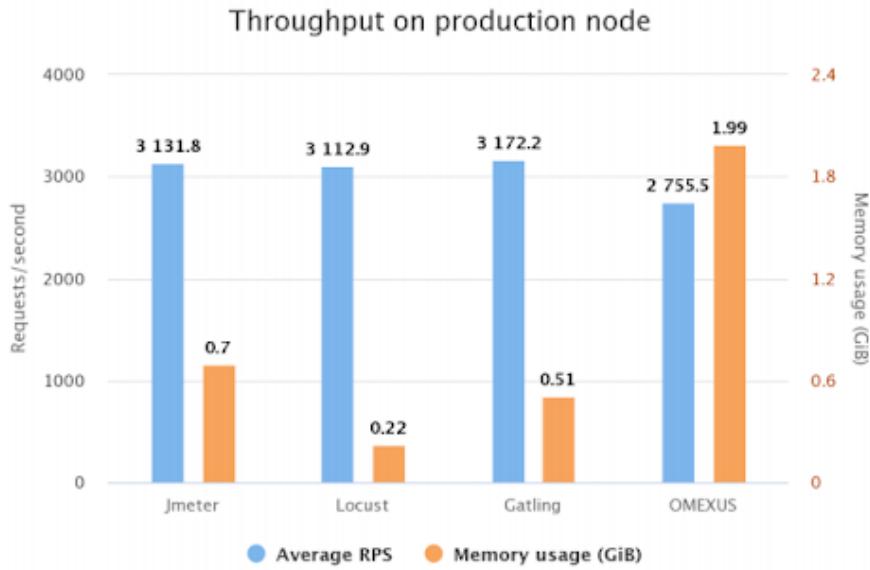


Figure 5.2: Throughput comparison in the production environment

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:** Thủ 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.

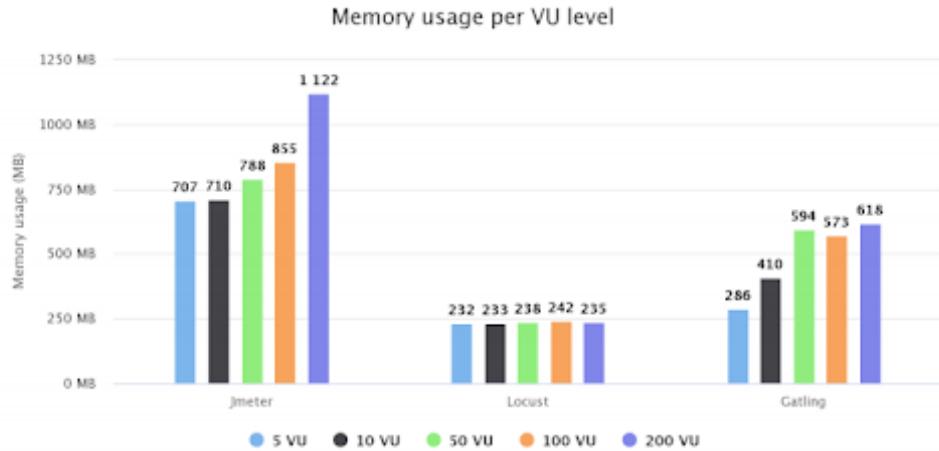


Figure 5.4: Memory usage per virtual user level on the docker environment

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:** Thủ nghiệm thời gian phản hồi của hệ thống khi tăng số lượng người dùng

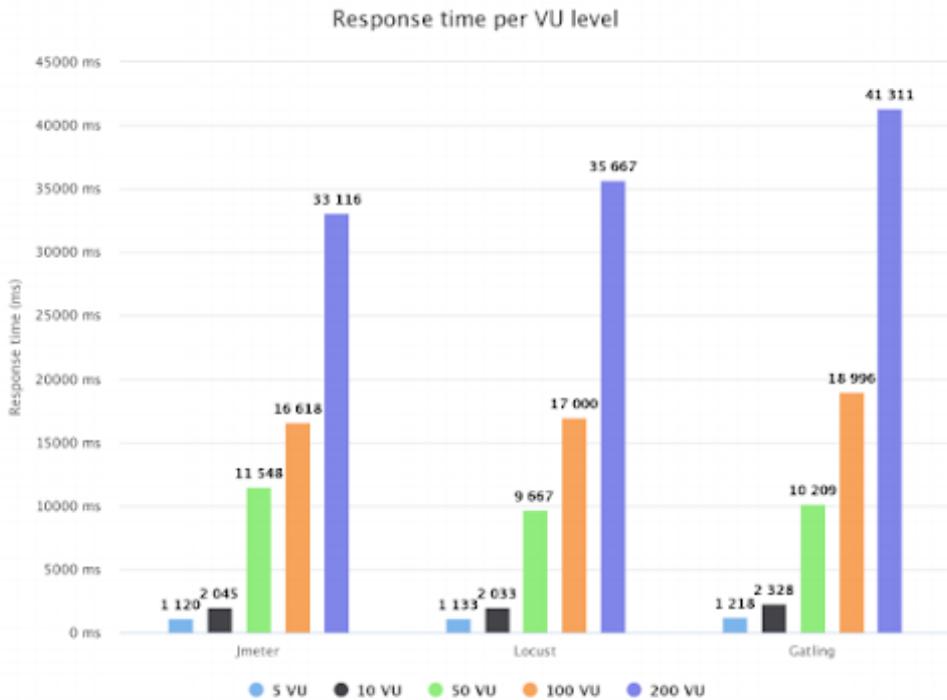


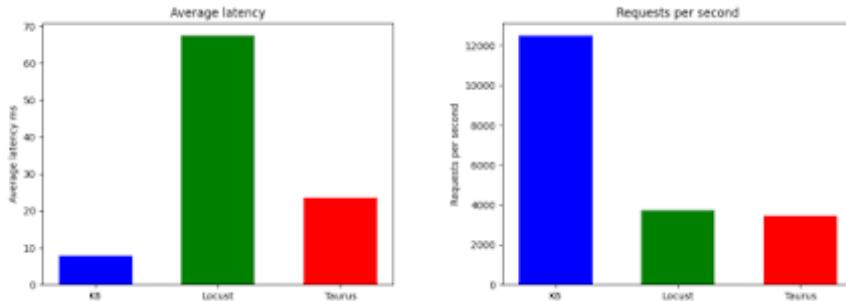
Figure 5.6: Response time at different virtual user levels

Hình 3.4: Response time at different virtual user levels

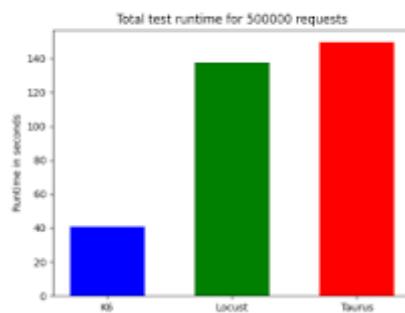
Kết quả: Thời gian phản hồi của hệ thống tăng tuyến tính đối với cả ba công cụ kiểm thử. Tuy nhiên, **Gatling** làm cho thời gian phản hồi tăng nhiều nhất khi số lượng người dùng ảo tăng lên, trong khi đó **Locust** giữ vị trí thứ hai.

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)

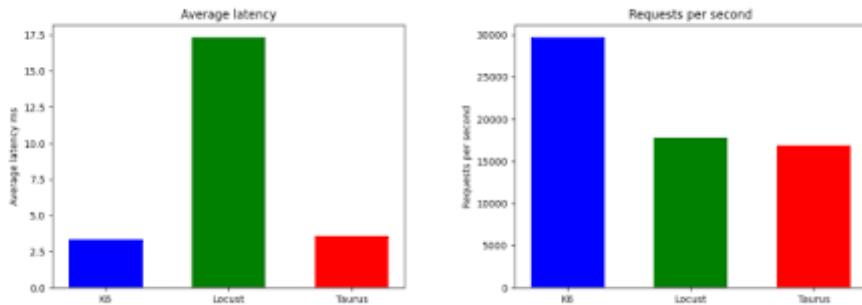
Kết quả: Công cụ **k6** thể hiện hiệu năng vượt trội rõ rệt với:

- 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:

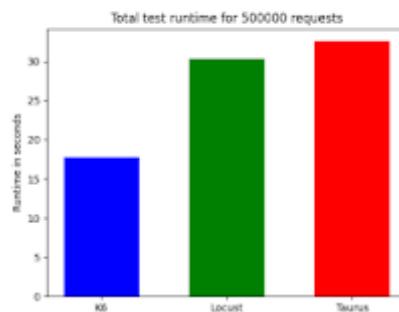
- Độ trễ phản hồi rất cao,
- Thông lượng thấp,
- Thời gian thực thi bài kiểm thử dài hơn đáng kể.

□ 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.

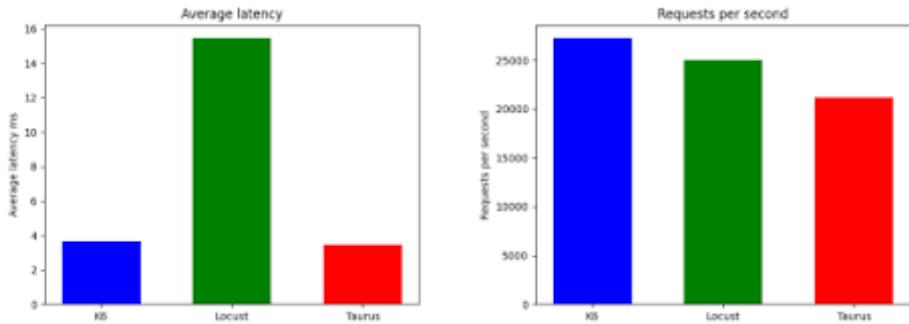
Figure 5: Results from running the static test for 5 iterations per tool on an AWS EC2 c5d.2xlarge instance for 500 000 requests and with 100 virtual users.

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:

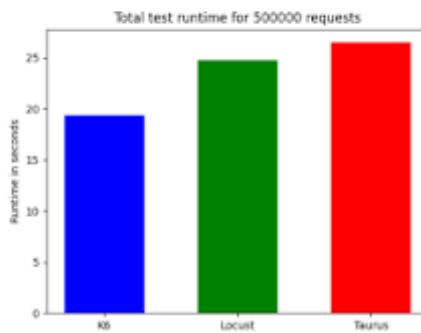
- Độ trễ phản hồi (latency) giảm rõ rệt.
- Số lượng yêu cầu xử lý mỗi giây (requests per second) tăng lên đáng kể.
- Thời gian thực hiện toàn bộ bài kiểm thử (total runtime) ngắn hơn.

□ **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 kỹ 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 scripts 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 Locust với các công cụ kiểm thử hiệu năng tương tự

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.

4.1 So sánh chi tiết

4.1.1 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.

4.1.2 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, nhằm 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.

4.1.3 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 (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 tinh 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 thuận 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.

4.2 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. Kết luận

5.1 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.

5.2 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.

5.3 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.