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.

3. Đánh giá công cụ

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.