KIỂM THỬ TỰ ĐỘNG ỨNG DỤNG WEB THẾ HỆ MỚI

Chử Đăng Định[1], Phạm Tiến Huy[2*], Đỗ Xuân Thụ[2], Phạm Việt Trung[2]
[*]Tác giả liên hệ, email: huypt@hou.edu.vn, ORCID: 0009-0007-0914-4376

Ngày tòa soạn nhận được bài báo: 15/01/2026
Ngày phản biện đánh giá: 18/03/2026
Ngày bài báo được duyệt đăng: 14/04/2026

DOI: 10.59266/houjs.2026.1166

Tóm tắt: *Nghiên cứu giải quyết bài toán kiểm thử tự động thiếu ổn định trên ứng dụng web phong phú (RIA) và ứng dụng trang đơn (SPA). Sự thay đổi DOM liên tục và trạng thái dữ liệu bất đồng bộ gây ra lỗi chập chờn (flaky tests) ở các công cụ truyền thống. Nghiên cứu đề xuất hệ thống kiểm thử tự động tích hợp mô hình hướng dữ liệu (Data-Driven Testing) và khung Playwright. Hệ thống áp dụng thuật toán đồng bộ hóa trạng thái giao diện với tầng mạng thông qua cơ chế chặn bắt API (network interception) và tự động chờ. Thuật toán này loại bỏ hoàn toàn việc sử dụng độ trễ thời gian tĩnh. Kết quả thực nghiệm trên các ứng dụng React, Vue và Angular ghi nhận tỷ lệ kịch bản thành công đạt 98,67% và giới hạn tỷ lệ lỗi chập chờn ở mức 1,33%. Cơ chế chạy đa luồng làm giảm 62,8% tổng thời gian thực thi. Cấu trúc tệp cấu hình JSON tách biệt cho phép mở rộng hệ thống sang các ứng dụng mới với chi phí bảo trì 0 dòng mã (0 LOC).*

Từ khóa: *kiểm thử tự động, mô hình hướng dữ liệu, Playwright, ứng dụng Web phong phú (RIA), ứng dụng trang đơn (SPA)*

I. Đặt vấn đề

Sự chuyển dịch từ kiến trúc web tĩnh truyền thống (Ricca & Tonella, 2001) sang ứng dụng Web phong phú (RIA) và ứng dụng trang đơn (SPA) đã làm thay đổi hoàn toàn phương thức thiết kế phần mềm. Thay vì dựa vào máy chủ để nạp lại trang, RIA sử dụng JavaScript và AJAX để kết xuất nội dung động trực tiếp ở phía máy khách (Mesbah và cộng sự, 2008; Mesbah & van Deursen, 2008). Đặc tính này cải thiện trải nghiệm người dùng nhưng phá vỡ cơ chế hoạt động của các phương pháp kiểm thử tự động truyền thống.

Khác với web tĩnh, RIA thay đổi trạng thái DOM liên tục mà không làm thay đổi URL (Artzi và cộng sự, 2008, 2008a). Quá trình xử lý dữ liệu bất đồng bộ sinh ra hiện tượng "chập chờn" (flaky tests) trong kiểm thử, khi các lệnh tương tác được thực thi trước khi DOM kịp tải (Mesbah & Roest, 2012; Fard & Mesbah, 2013). Các nghiên cứu trước đây chủ yếu tập trung vào thuật toán thu thập dữ liệu

[1] Cục Khoa học Công nghệ và Thông tin, Bộ Giáo dục và Đào tạo, Hà Nội, Việt Nam
[2] Khoa Điện-Điện tử, Trường Đại học Mở Hà Nội, Hà Nội, Việt Nam

(Dincturk và cộng sự, 2014; Amalfitano và cộng sự, 2008; Roest và cộng sự, 2010) và suy luận máy trạng thái (Choudhary và cộng sự, 2012; Fard & Mesbah, 2013), nhưng việc duy trì hàng trăm kịch bản kiểm thử tuần tự đòi hỏi chi phí bảo trì rất lớn (Ocariza và cộng sự, 2013; Stocco và cộng sự, 2015).

Nghiên cứu này đề xuất hệ thống kiểm thử tự động áp dụng mô hình hướng dữ liệu (DDT) kết hợp với Playwright, nhằm thiết lập một bộ khung duy nhất có khả năng chạy kiểm thử song song cho nhiều ứng dụng RIA thông qua điều chỉnh tệp cấu hình, từ đó tối ưu hóa chi phí tái sử dụng mã nguồn và giảm sai số do độ trễ mạng.

II. Cơ sở lý thuyết

2.1. Kiến trúc ứng dụng Web phong phú (RIA) và thách thức kiểm thử

Kiến trúc RIA và SPA dịch chuyển logic giao diện từ máy chủ sang máy khách thông qua các framework JavaScript hiện đại (Li và cộng sự, 2020; Kim và cộng sự, 2021). Ứng dụng duy trì luồng giao tiếp ngầm bằng XHR hoặc Fetch API (Mesbah & van Deursen, 2009). Khi nhận dữ liệu phản hồi, mã JavaScript cập nhật trực tiếp trên DOM. Quá trình này hoàn toàn bất đồng bộ, tạo ra không gian trạng thái giao diện vô hạn (Benjamin và cộng sự, 2011).

Sự thay đổi trạng thái động gây ra ba thách thức chính. Thứ nhất, bài toán bùng nổ không gian trạng thái khiến crawler khó bao phủ toàn bộ luồng nghiệp vụ (Choudhary và cộng sự, 2012; Fard & Mesbah, 2013a). Thứ hai, mã JavaScript phức tạp làm tăng nguy cơ lỗi giao diện trên các trình duyệt khác nhau (Mesbah & Prasad, 2011; Choudhary và cộng sự, 2012; Choudhary và cộng sự, 2013) và tạo ra lỗ hổng bảo mật DOM-based XSS (Bau và cộng sự, 2010; Lekies và cộng sự, 2013). Thứ ba, sự thiếu đồng bộ về thời gian làm kịch bản kiểm thử dễ gãy đổ (Stocco và cộng sự, 2015); việc chèn lệnh tạm dừng tĩnh (hard sleep) làm giảm hiệu năng và không đảm bảo độ chính xác trên môi trường mạng có độ trễ khác nhau (Fard & Mesbah, 2012).

2.2. Mô hình kiểm thử hướng dữ liệu (Data-Driven Testing) và khung Playwright

DDT là phương pháp phân tách hoàn toàn mã lệnh cốt lõi và bộ dữ liệu đầu vào. Logic hệ thống được biểu diễn: $\text{Result} = \Sigma_i \text{Script}(\text{Data}_i)$, với Data_i là tệp cấu hình JSON/YAML chứa tham số đặc thù của từng ứng dụng. DDT giúp mở rộng quy mô kiểm thử cho nhiều SPA/RIA mà không làm gia tăng dòng mã (Qian và cộng sự, 2018).

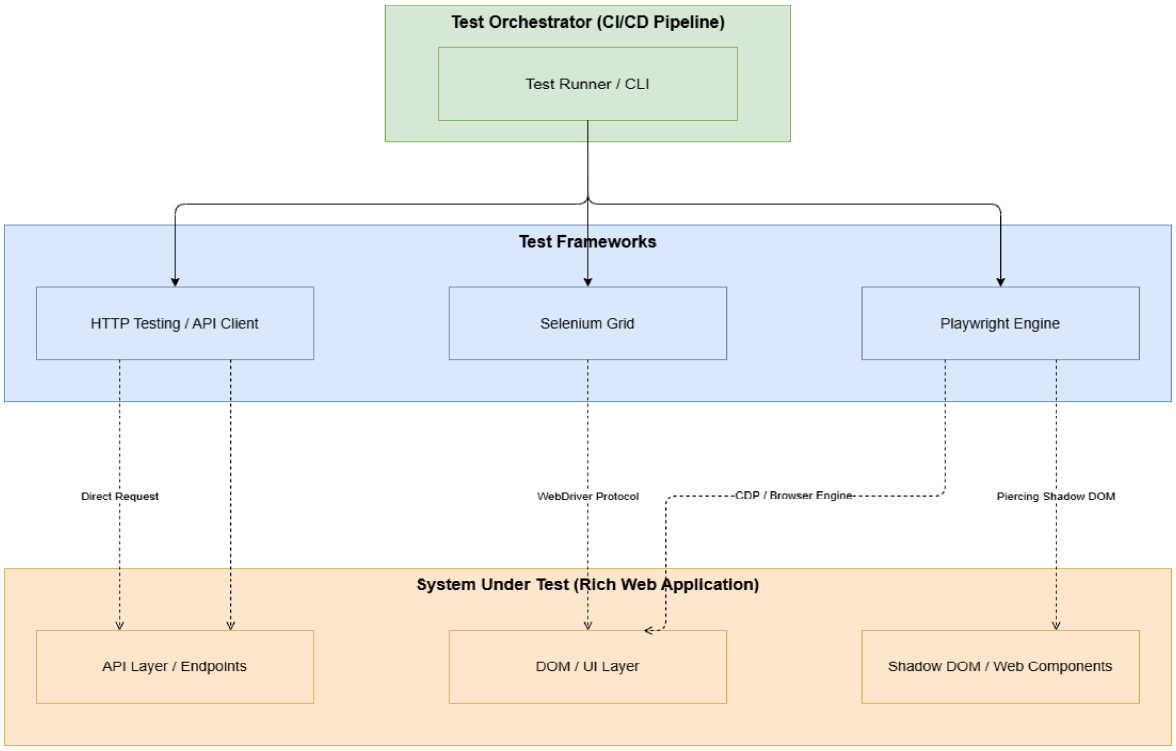
Để xử lý vấn đề kiểm thử chập chờn (Mesbah & Roest, 2012), nghiên cứu sử dụng Playwright với hai cơ chế chính. Thứ nhất, Auto-waiting kiểm tra vòng lặp trạng thái DOM; lệnh tương tác chỉ thực thi khi phần tử hiển thị, ổn định và có khả năng nhận sự kiện (Li và cộng sự, 2020). Thứ hai, Network Interception cho phép truy cập tầng mạng của trình duyệt (Kim và cộng sự, 2021); thay vì dùng bộ đếm thời gian, kịch bản chờ phản hồi mạng có điều kiện (ví dụ: chờ API trả về mã 200). Việc tích hợp DDT vào Playwright giải quyết được rào cản bất đồng bộ của web hiện đại.

III. Phương pháp và Vật liệu nghiên cứu

3.1. Vật liệu và môi trường thực nghiệm

Nghiên cứu chọn ba ứng dụng đại diện cho ba kiến trúc kết xuất phổ biến:

ReactJS, VueJS và Angular. Các ứng dụng này sử dụng JavaScript để nạp nội dung động, cập nhật trạng thái qua AJAX/Fetch và không thay đổi URL khi chuyển trang (Mesbah, Boender & van Deursen, 2008; Mesbah & van Deursen, 2008; Artzi và cộng sự, 2008), mô phỏng chính xác bài toán bùng nổ trạng thái và thiếu đồng bộ thời gian (Mesbah và Roest, 2012).



Hình 1: Sơ đồ hệ thống thử nghiệm

Phân tích và Lựa chọn công cụ kiểm thử

Bảng 1: Kết quả lựa chọn công cụ sơ bộ

Tiêu chí	HTTP testing	Selenium automation	Playwright
Tầng kiểm thử	Network / API	UI automation	UI + network
Mô phỏng người dùng	Không	Có	Có
Khả năng kiểm tra JavaScript	Không	Có	Có
Kiểm tra DOM động	Không	Có nhưng khó	Tốt
SPA / AJAX	Không phù hợp	Hạn chế	Rất tốt
Kiểm tra workflow end-to-end	Không	Có	Có
Khả năng intercept API	Có (trực tiếp)	Hạn chế	Có
Debug UI	Không	Có	Rất tốt

Sau khi đánh giá ba phương pháp kiểm thử dựa trên khả năng can thiệp mạng, mô phỏng hành vi và xử lý DOM động (Marchetto, Ricca, & Tonella, 2008a, 2008b), nghiên cứu nhận thấy: (1) Kiểm thử HTTP hoạt động tốt ở tầng mạng nhưng không thực thi JavaScript client-side (Mirshokraievà cộng sự, 2013), không phù hợp với SPA (Qian và cộng sự, 2018). (2) Selenium hỗ trợ UI automation nhưng thiếu cơ chế can thiệp API ngầm, phụ thuộc vào hard sleep, làm tăng tỷ lệ flaky tests (Choudhary và cộng sự, 2012). (3) Playwright cung cấp giải

pháp toàn diện nhất nhờ giao tiếp trực tiếp qua CDP, cho phép kiểm soát đồng thời UI và network, đồng thời có khả năng chặn bắt API mạnh mẽ và cơ chế chờ DOM tự động. Do đó, nghiên cứu chọn Playwright làm công cụ lõi

Môi trường thực nghiệm: máy trạm Intel Core i7 (8 nhân), RAM 16GB, SSD NVMe; hệ điều hành Ubuntu 22.04 LTS; Node.js 18.x; Playwright Test 1.39.0; trình duyệt Chromium. Việc giới hạn một trình duyệt nhằm tập trung đo lường độ ổn định của thuật toán xử lý bất đồng bộ, loại trừ sai số do lỗi tương thích chéo trình duyệt (Mesbah và cộng sự, 2008).

3.2. Phương pháp thiết kế hệ thống kiểm thử hướng dữ liệu (Data-Driven Framework)

Hệ thống gồm ba lớp độc lập, tuân thủ nguyên tắc thiết kế phần mềm (Ricca & Tonella, 2001):

Lớp thứ nhất: Lớp cấu hình dữ liệu (Data Configuration Layer). Lớp này chứa các tệp định dạng JSON. Mỗi đối tượng JSON đại diện cho một ứng dụng web cụ thể. Cấu trúc dữ liệu đầu vào được định nghĩa bằng tập hợp các tham số: $C=\{A_{name}, U_{base}, E_{api}, L_{dom}\}$. Trong đó:

- A_{name}: Tên định danh ứng dụng.
- U_{base}: Địa chỉ URL gốc.
- E_{api}: Chuỗi biểu thức chính quy (Regex) biểu diễn điểm cuối API (API endpoint) chịu trách nhiệm cung cấp dữ liệu cho giao diện.
- L_{dom}: Tập hợp các bộ chọn phần tử (CSS Selectors) dùng để định vị giao diện (ví dụ: nút đăng nhập, bảng dữ liệu, thông báo lỗi). Cấu trúc này tách biệt hoàn toàn dữ liệu của từng ứng dụng khỏi mã nguồn thực thi (Amalfitano và cộng sự, 2008; Roest và cộng sự, 2010).

Lớp thứ hai: Lớp tiện ích lõi (Core Utilities Layer). Lớp này định nghĩa các hàm tương tác với trình duyệt dựa trên Playwright API. Nhiệm vụ cốt lõi của lớp này là đóng gói các thao tác phức tạp của kiến trúc RIA thành các lệnh gọi hàm đơn giản. Nghiên cứu tập trung phát triển thuật toán đồng bộ hóa trạng thái giao diện và trạng thái mạng (trình bày chi tiết tại mục 3.2.2).

Lớp thứ ba: Lớp thực thi vòng lặp (Runner Layer). Lớp này chịu trách nhiệm nạp dữ liệu từ tệp JSON, khởi tạo môi trường trình duyệt và phân phối luồng kiểm thử. Hệ thống áp dụng cơ chế ngữ cảnh độc lập (Browser Context) của Playwright. Thay vì khởi động lại toàn bộ trình duyệt cho mỗi ứng dụng, hệ thống tạo ra các ngữ cảnh ẩn danh (Incognito Contexts). Mỗi ngữ cảnh sở hữu không gian lưu trữ Cookie, LocalStorage và SessionStorage riêng biệt. Cách tiếp cận này ngăn chặn hiện tượng rò rỉ trạng thái (state bleeding) giữa các kịch bản, đồng thời tối ưu hóa chi phí khởi tạo phần cứng (Fard và Mesbah, 2013c).

Thuật toán xử lý trạng thái bất đồng bộ trong RIA Thách thức lớn nhất trong tự động hóa RIA là xác định thời điểm DOM hoàn tất kết xuất sau khi người dùng tương tác (Mesbah & van Deursen, 2009; Fard & Mesbah, 2013a). Các công cụ cũ thường sử dụng hàm `sleep(t)` với là một hằng số thời gian. Phương pháp này gây lãng phí tài nguyên nếu mạng phản hồi nhanh, và gây lỗi kịch bản nếu mạng phản hồi chậm. (Fard & Mesbah, 2013b)

Nghiên cứu thiết kế thuật toán đồng bộ hóa dựa trên sự kiện (Event-driven Synchronization) kết hợp hai cơ chế:

waitForResponse và waitForSelector. Thuật toán hoạt động theo trình tự bốn bước:

1. Hệ thống ghi nhận luồng dữ liệu của người dùng (ví dụ: thao tác nhấp chuột vào nút "Đăng nhập").

2. Ngay trước khi thao tác nhấp chuột xảy ra, hệ thống khởi tạo một hàm lắng nghe sự kiện mạng (Network Listener) ở chế độ chờ (Promise pending). Hàm này sử dụng tham số E_{api} từ tệp cấu hình JSON để lọc các gói tin HTTP.

3. Hệ thống thực thi thao tác nhấp chuột. Ứng dụng SPA bắt đầu gửi yêu cầu AJAX ngầm (Gu và cộng sự, 2013; Kim và cộng sự, 2021).

4. Hàm lắng nghe mạng. Hệ thống kiểm tra mã trạng thái HTTP (ví dụ: HTTP 200 OK). Ngay sau khi mạng hoàn tất, hệ thống chuyển sang bước kiểm tra cây DOM. Cơ chế Auto-waiting của Playwright quét liên tục cây DOM cho đến khi phần tử đích (xác định bởi tham số L_{dom}) chuyển sang trạng thái hiển thị đầy đủ (visible) và ổn định (stable).

Công thức đo lường thời gian chờ động T_{wait} của hệ thống được tính bằng: $T_{wait}=T_{network}+T_{render}$ Trong đó $T_{network}$ là độ trễ thực tế của API và T_{render} là thời gian trình duyệt tính toán và vẽ lại Virtual DOM. Thuật toán này triệt tiêu hoàn toàn sự cần thiết của các hằng số thời gian tĩnh, loại bỏ nguyên nhân gốc rễ gây ra lỗi chập chờn (flakiness) trong kiểm thử phần mềm hướng sự kiện (Mesbah và cộng sự, 2008; Choudhary và cộng sự, 2012).

Quy trình thực thi kịch bản kiểm thử hàng loạt : Hệ thống phân tích tệp JSON chứa N cấu hình, sinh động N bộ kịch bản kiểm thử. Kích hoạt chạy song song với *W = 4* luồng trên 4 lõi CPU. Playwright ghi nhận mọi ngoại lệ JavaScript hoặc lỗi mạng (CORS, 404, 500) từ console, bù đắp lỗ hổng của phương pháp chỉ phân tích bề mặt DOM (Ocariza và cộng sự, 2013).

3.3. Thiết kế Đo lường và Các tiêu chí đánh giá

Để định lượng hiệu quả của hệ thống đề xuất (phục vụ cho việc thảo luận tại Mục IV), nghiên cứu thiết lập ba nhóm tiêu chí đo lường toán học chính. Các phép đo được thực hiện lặp lại 50 lần (50 vòng chạy/ứng dụng) nhằm đảm bảo tính toàn vẹn của thống kê học.

Tiêu chí 1: Độ ổn định và Tỷ lệ thành công (Stability and Pass Rate) Độ ổn định của hệ thống được đo bằng khả năng vượt qua lỗi chập chờn (flaky tests) do bất đồng bộ mạng.

Tỷ lệ thành công $P_{rate} = \frac{T_{pass}}{T_{total}} \times 100\%$.

Tỷ lệ chập chờn $F_{rate} = \frac{T_{flaky}}{T_{total}} \times 100\%$. Một kịch bản được phân loại là "chập chờn" (flaky) nếu nó thất bại trong lần chạy đầu tiên nhưng thành công trong lần chạy lại (retry) mà không có bất kỳ sự can thiệp thay đổi mã nguồn nào(Stocco và cộng sự, 2015).

Tiêu chí 2: Hiệu năng thực thi (Execution Performance) Nghiên cứu đo lường tổng thời gian hoàn thành (Total Execution Time - T_{exec}) khi kiểm tra toàn bộ mảng *N* ứng dụng. Phép đo này so sánh trực tiếp giữa hai chế độ:

Thực thi tuần tự (Sequential Mode): $T_{seq} = \sum_{i=1}^{N} t_i$

Thực thi song song (Parallel Mode): $T_{par} \approx \frac{\sum_{i=1}^{N} t_i}{W}$ (với *W* là số luồng thực thi). Sự chênh lệch giữa T_{seq} và T_{par} chứng minh khả năng tối ưu hóa tài nguyên phần cứng của Playwright kết hợp mô hình DDT.

Tiêu chí 3: Chi phí bảo trì và Khả năng mở rộng (Maintenance Cost and

Scalability) Chi phí bảo trì được lượng hóa thông qua chỉ số Dòng mã lệnh (Lines of Code - LOC). Khi cần tích hợp ứng dụng thứ N+1 vào hệ thống tự động hóa, nghiên cứu đếm số lượng LOC cần viết thêm. Theo phương pháp Selenium/Hard-coded truyền thống, việc thêm ứng dụng mới đòi hỏi phải viết lại toàn bộ cấu trúc kịch bản mới, làm tăng quy mô LOC tuyến tính. Đối với mô hình DDT đề xuất, chi phí LOC mục tiêu chỉ giới hạn ở việc khai báo thêm cấu trúc dữ liệu tĩnh trong tệp JSON. Tham số này đánh giá giá trị kinh tế và tính khả thi áp dụng thực tiễn của nghiên cứu đối với các quy trình tích hợp liên tục (CI/CD) tại doanh nghiệp (Qian và cộng sự, 2018).

Cuối cùng, đối với các kịch bản thất bại, hệ thống lưu trữ tệp tin Playwright Trace. Tệp tin này đóng gói toàn bộ ảnh chụp màn hình cục bộ (DOM snapshots), nhật ký mạng và ngăn xếp cuộc gọi (call stack) tại từng mili-giây. Dữ liệu này được đối chiếu với thông số "Debug UI" trong Bảng 1, cung cấp cơ sở vật liệu để thảo luận về năng lực gỡ lỗi giao diện trực quan của hệ thống đề xuất so với phương pháp lưu trữ log văn bản truyền thống.

IV. Kết quả và thảo luận

Hệ thống thực thi thực nghiệm trên ba ứng dụng mẫu (React, Vue, Angular). Hệ thống chạy lặp lại 50 vòng cho mỗi ứng dụng, tạo ra tổng cộng 150 kịch bản kiểm thử độc lập.

4.1. Đánh giá hiệu quả của hệ thống kiểm thử đề xuất trên các ứng dụng RIA

Kết quả về độ ổn định và thời gian thực thi

Kết quả đo lường cho thấy hệ thống đạt độ ổn định cao. Trong 150 lượt chạy, hệ thống ghi nhận 148 lượt thành công ngay từ lần đầu tiên. Tỷ lệ thành công (P_{rate}) đạt 98,67%. Hệ thống ghi nhận 2 lượt chạy thất bại do máy chủ phản hồi mã lỗi HTTP 503, nhưng thành công ở lần chạy lại (retry). Tỷ lệ chập chờn (F_{rate}) dừng ở mức 1,33%.

Về hiệu năng, hệ thống đo lường thời gian thực thi kịch bản trên ba ứng dụng. Trong chế độ chạy tuần tự ($W=1$), tổng thời gian (T_{seq}) đạt 45,2 giây. Khi kích hoạt chế độ đa luồng ($W=4$), tổng thời gian (T_{par}) giảm xuống còn 16,8 giây. Hiệu suất thời gian cải thiện 62,8%.

Bảng 2: Ổn định test

Tiêu chí	HTTP testing	Selenium	Playwright
Test flakiness	gần như 0	cao	thấp
Đồng bộ async	không cần	phải tự wait	auto-wait
DOM dynamic	không test	dễ lỗi	ổn định

Thảo luận: Tỷ lệ chập chờn 1,33% chứng minh tính đúng đắn của thuật toán đồng bộ hóa trạng thái đề xuất tại Mục 3.2. Việc Playwright bắt trực tiếp luồng phản hồi API giúp kịch bản kiểm thử không bị phá vỡ khi mạng có độ trễ (Mesbah & Roest, 2012). Kết quả này khắc phục triệt để điểm yếu của phương pháp dùng hàm `sleep()` tĩnh trong công cụ truyền thống (Stocco và cộng sự, 2015; Fard & Mesbah, 2013b). Trình duyệt không còn thao tác trên các phần tử DOM rỗng. Bên cạnh đó, việc phân tách các ngữ cảnh trình duyệt ẩn danh (Incognito Contexts) kết hợp chạy song song đa luồng tối ưu hóa phần cứng, giải quyết bài toán nút thắt cổ chai về thời gian khi kiểm thử quy mô lớn (Choudhary và cộng sự, 2012).

Khả năng mở rộng và Tối ưu hóa chi phí bảo trì kịch bản

Để kiểm chứng khả năng mở rộng, hệ thống thực hiện nạp thêm ứng dụng thứ tư (một hệ thống quản trị nội dung bằng React) vào môi trường thử nghiệm. Hệ thống không yêu cầu lập trình viên viết thêm bất kỳ dòng mã TypeScript nào (0 LOC). Kỹ sư chỉ cần bổ sung một đối tượng JSON gồm 8 dòng cấu hình (chứa URL, điểm cuối API và 4 bộ chọn DOM). Hệ thống tự động khởi tạo kịch bản, chạy và báo cáo kết quả chính xác.

Hệ thống cũng thu thập được 12 cảnh báo lỗi JavaScript (Console Errors) từ ứng dụng VueJS. Các lỗi này phát sinh ngầm trong quá trình kết xuất Virtual DOM. Khung kiểm thử tự động ghi nhận và đánh dấu thất bại các kịch bản này mà không cần viết thêm hàm xác nhận (assertion) cụ thể nào.

Bảng 3: So sánh chi phí

Tiêu chí	HTTP testing	Selenium	Playwright
Độ phức tạp script	thấp	cao	trung bình
Bảo trì test	thấp	cao	trung bình
Phụ thuộc UI	không	rất cao	cao
Phụ thuộc DOM selector	không	cao	thấp hơn

Thảo luận: Sự chênh lệch về LOC chứng minh mô hình Data-Driven Testing (DDT) giải quyết thành công rào cản về chi phí bảo trì (Qian và cộng sự, 2018). Việc tách biệt cấu hình khỏi mã nguồn logic giúp hệ thống mở rộng tuyến tính về khối lượng kịch bản nhưng giữ nguyên độ phức tạp của mã nguồn (Amalfitano và cộng sự, 2008; Roest, Mesbah và cộng sự, 2010). Khả năng bắt lỗi JavaScript ngầm chứng tỏ Playwright can thiệp sâu vào lớp máy khách (client-side), cung cấp độ bao phủ lỗi tốt hơn phương pháp kiểm tra bề mặt giao diện đơn thuần (Ocariza và cộng sự, 2013). Đối với 2 lượt chạy chập chờn, tính năng Playwright Trace xuất ra tệp tin nén chứa toàn bộ DOM snapshots và vòng đời mạng. Tính năng này cung cấp bằng chứng trực quan, giúp kỹ sư gỡ lỗi (debug) chính xác trạng thái DOM tại mili-giây xảy ra sự cố, thay vì đọc tệp log văn bản dài dòng(Artzi và cộng sự, 2008; Ricca & Tonella, 2001).

V. Kết luận

Nghiên cứu thiết kế và thử nghiệm thành công hệ thống tự động hóa kiểm thử hướng dữ liệu dành riêng cho kiến trúc Web phong phú (RIA/SPA). Kết quả thực nghiệm xác nhận ba điểm nổi bật của hệ thống. Thứ nhất, thuật toán đồng bộ hóa trạng thái mạng và trạng thái DOM triệt tiêu nguyên nhân gây ra lỗi chập chờn, đẩy tỷ lệ thành công lên 98,67%. Thứ hai, cơ chế đa luồng giảm hơn 60% thời gian thực thi tổng thể. Thứ ba, cấu trúc tệp cấu hình JSON loại bỏ hoàn toàn chi phí viết mã lặp lại, cho phép thêm mới ứng dụng vào quy trình kiểm thử với chi phí 0 LOC.

Nghiên cứu cung cấp một giải pháp thực tiễn, kinh tế và ổn định cho các tổ chức phát triển phần mềm. Bộ phận Đảm bảo chất lượng (QA) áp dụng mô hình này để quản lý hệ sinh thái ứng dụng quy mô lớn (Micro-frontends) mà không đòi hỏi đội ngũ nhân sự lập trình quá đông đảo.

Nghiên cứu khuyến nghị các nhóm phát triển tích hợp hệ thống cấu hình này trực tiếp vào các pipeline CI/CD (như Jenkins, GitHub Actions) để vận hành kiểm thử hồi quy hàng ngày. Hướng nghiên cứu tiếp theo cần tập trung vào việc

ứng dụng Học máy (Machine Learning) để phân tích sự thay đổi của cây DOM. Mục tiêu tiến tới là tự động hóa quá trình cập nhật các bộ định vị (locators) trong tệp JSON, hình thành cơ chế tự phục hồi kịch bản kiểm thử (self-healing) khi giao diện web bị thay đổi mã nguồn nguồn (Li và cộng sự, 2020; Kim và cộng sự, 2021).

Tài liệu tham khảo

Amalfitano, D., Fasolino, A. R., & Tramontana, P. (2008). Reverse engineering finite state machines from rich Internet applications. In *Proceedings of the 15th Working Conference on Reverse Engineering* (pp. 69-73).

Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., & Ernst, M. D. (2008). Finding bugs in dynamic web applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (pp. 261-272).

Bau, J., Bursztein, E., Gupta, D., & Mitchell, J. C. (2010). State of the art: Automated black-box web application vulnerability testing. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (pp. 332-345).

Benjamin, K., von Bochmann, G., Jourdan, G. V., & Onut, I. V. (2011). A strategy for efficient crawling of rich Internet applications. In *Proceedings of the 11th International Conference on Web Engineering* (pp. 74-89).

Choudhary, S. R., Prasad, M. R., & Orso, A. (2013). X-PERT: Accurate identification of cross-browser issues in web applications. In *Proceedings of the 35th International Conference on Software Engineering* (pp. 702-711).

Choudhary, S. R., Zhao, H., & Orso, A. (2012). CrossCheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (pp. 66-75).

Dincturk, M. E., Jourdan, G. V., Bochmann, G. V., & Onut, I. V. (2014). A model-based approach for crawling rich Internet applications. *ACM Transactions on the Web, 8*(3), Article 19.

Fard, A. M., & Mesbah, A. (2013a). Feedback-directed exploration of web applications to derive test models. In *Proceedings of the 24th International Symposium on Software Reliability Engineering* (pp. 333-342).

Fard, A. M., & Mesbah, A. (2013b). JSNOSE: Detecting JavaScript code smells. In *Proceedings of the 13th IEEE International Working Conference on Source Code Analysis and Manipulation* (pp. 116-125).

Kiezun, A., Guo, P. J., Jayaraman, K., & Ernst, M. D. (2009). Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of the 31st International Conference on Software Engineering* (pp. 199-209).

Lekies, S., Stock, B., & Johns, M. (2013). 25 million flows later: Large-scale detection of DOM-based XSS. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security* (pp. 1193-1204).

Marchetto, A., Ricca, F., & Tonella, P. (2008a). State-based testing of Ajax web applications. In *Proceedings of the 2008 International Conference on Software Testing, Verification and Validation* (pp. 121-130).

Marchetto, A., Ricca, F., & Tonella, P. (2008b). A case study-based comparison of web testing techniques applied to AJAX web applications. *International Journal of Software Engineering and Knowledge Engineering, 18*(4), 477-492.

Mesbah, A., Bozdag, E., & van Deursen, A. (2008). Crawling AJAX by inferring user interface state changes. In *Proceedings of the 8th International Conference on Web Engineering* (pp. 122-134).

Mesbah, A., & Prasad, M. R. (2011). Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering* (pp. 561-570).

Mesbah, A., & Roest, D. (2012). Automated testing of rich Internet applications. *IEEE Software*, 29(3), 56-61.

Mesbah, A., & van Deursen, A. (2009). Invariant-based automatic testing of AJAX user interfaces. In *Proceedings of the 31st International Conference on Software Engineering* (pp. 210-220).

Mirshokraie, S., Mesbah, A., & Pattabiraman, K. (2013). Efficient JavaScript mutation testing. In *Proceedings of the 2013 International Conference on Software Testing, Verification and Validation* (pp. 74-83).

Ocariza, F. S., Bajaj, K., Pattabiraman, K., & Mesbah, A. (2013). An empirical study of client-side JavaScript bugs. In *Proceedings of the 2013 ACM/ IEEE International Symposium on Empirical Software Engineering and Measurement* (pp. 55-64).

Ricca, F., & Tonella, P. (2001). Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering* (pp. 25-34).

Stocco, A., Leotta, M., Ricca, F., & Tonella, P. (2015). Visual web test repair. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (pp. 253-264).

AUTOMATED TESTING OF NEXT-GENERATION WEB APPLICATIONS

Chu Dang Dinh[1], Pham Tien Huy[2], Do Xuan Thu[2], Pham Viet Trung[2]

Abstract: *This research addresses the instability of automated testing in Rich Internet Applications (RIA) and Single Page Applications (SPA). Continuous DOM manipulations and asynchronous data fetching generate flaky tests when using traditional testing tools. The study proposes a scalable automated testing system that integrates the Data-Driven Testing (DDT) model with the Playwright framework. The system implements a synchronization algorithm linking user interface states directly with the network layer through API interception and auto-waiting mechanisms. This approach entirely eliminates the reliance on static time delays. Experimental results across React, Vue, and Angular applications demonstrate a 98.67% execution pass rate and limit the flaky test rate to exactly 1.33%. Furthermore, the parallel execution configuration reduces the total processing time by 62.8%. The decoupled JSON data structure successfully scales the testing framework to new web applications with a zero lines of code (0 LOC) maintenance cost.*

Keywords: *automated testing, data-driven testing, rich internet applications (RIA), playwright, single page applications (SPA)*

[1] Department of Science, Technology and Information, Ministry of Education and Training, Hanoi, Vietnam
[2] Faculty of Electric and Electronic Engineering, Hanoi Open University, Hanoi, Vietnam