

**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**  
**TRƯỜNG ĐẠI HỌC BÁCH KHOA**  
**KHOA KỸ THUẬT HÓA HỌC**



**BÁO CÁO HỌC PHẦN MỞ RỘNG**  
**BỘ MÔN CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT**

**ĐỀ TÀI: NÉN VÀ PHÁT HIỆN BẤT THƯỜNG**  
**BẰNG TRIỆ VÀ PHÂN TÍCH THỐNG KÊ**

**Lớp TN01 - Nhóm 15 - HK251**

**Giảng viên hướng dẫn: TS. LÊ THÀNH SÁCH**  
**Danh sách nhóm:**

STT	Họ và tên	MSSV	Ghi chú
1	Phạm Đoàn Gia Cát	2410369	
2	Trần Quốc Huy	2411283	
3	Phạm Trường Chính	2410407	

*TP. Hồ Chí Minh, tháng 12/2025*

# Liên kết Mã nguồn và Thực nghiệm

GitHub Repository:

<https://ptn1ne.github.io/VPTREETEST/>

Google Colab Notebook:

[https://colab.research.google.com/drive/1mnRgF4I-EXXaBXJkNhD1K\\_VphWaJ9vo5?usp=sharing](https://colab.research.google.com/drive/1mnRgF4I-EXXaBXJkNhD1K_VphWaJ9vo5?usp=sharing)

## Tóm tắt

Trong bối cảnh kỷ nguyên dữ liệu lớn (Big Data), việc giám sát và phân tích log hệ thống đóng vai trò then chốt để đảm bảo tính ổn định của dịch vụ. Tuy nhiên, sự bùng nổ về khối lượng log khiến các phương pháp lưu trữ truyền thống trở nên tốn kém, trong khi các mô hình giám sát dựa trên Học sâu (Deep Learning) lại đòi hỏi tài nguyên tính toán quá lớn và thiếu tính minh bạch (black-box). Để giải quyết bài toán này, báo cáo trình bày quy trình thiết kế và cài đặt hệ thống **LogDrainTrie** - một giải pháp toàn diện cho việc nén và phát hiện bất thường trên luồng log trực tuyến.

Về mặt kỹ thuật, hệ thống được xây dựng hoàn toàn thủ công (implemented from scratch) dựa trên cấu trúc dữ liệu **Trie lai (Hybrid Trie)**. Điểm cải tiến cốt lõi nằm ở kiến trúc phân lớp theo độ dài chuỗi (Length-Layered Architecture) giúp giảm không gian tìm kiếm xuống độ phức tạp  $O(1)$ , kết hợp với **cơ chế thống kê tăng cường (Incremental Statistics)** để duy trì mức tiêu thụ bộ nhớ hằng số. Phương pháp này sử dụng các biến tích lũy (Tổng và Tổng bình phương) để tính toán phương sai trực tuyến, cho phép hệ thống vận hành mượt mà trên các thiết bị tài nguyên hạn chế mà không gặp hiện tượng tràn bộ nhớ.

Kết quả thực nghiệm trên tập dữ liệu chuẩn HDFS (100,000 dòng log) chứng minh tính hiệu quả vượt trội của giải pháp đề xuất. Hệ thống đạt tỷ lệ nén dữ liệu lên tới **13 lần** so với văn bản gốc và độ chính xác (F1-Score) trong phát hiện bất thường đạt **92.9%**. Đáng chú ý, hệ thống thể hiện khả năng "giải thích được" (Explainability) cao thông qua việc trực quan hóa cấu trúc cây, giúp người quản trị dễ dàng truy vết nguyên nhân của ba loại lỗi phổ biến: bất thường cấu trúc, sự kiện hiếm và giá trị tham số ngoại lai.

# Contents

<b>1</b>	<b>Giới thiệu (Introduction)</b>	<b>4</b>
1.1	Đặt vấn đề . . . . .	4
1.2	Mục tiêu đề tài . . . . .	4
1.3	Phạm vi và Phương pháp nghiên cứu . . . . .	4
<b>2</b>	<b>Cơ sở lý thuyết</b>	<b>6</b>
2.1	Cấu trúc dữ liệu Trie và Thống kê tần suất . . . . .	6
2.1.1	Định nghĩa và Tính chất của Trie . . . . .	6
2.1.2	Lưu trữ tần suất trên nút . . . . .	6
2.2	Phát hiện sự kiện hiếm dựa trên Tần suất (Frequency-based Detection) . . . . .	7
2.2.1	Cơ chế xác định ngưỡng động . . . . .	7
2.2.2	Quy tắc phát hiện . . . . .	7
<b>3</b>	<b>Thiết kế hệ thống và Phương pháp thực hiện</b>	<b>8</b>
3.1	Tổng quan kiến trúc . . . . .	8
3.2	Quy trình Tiền xử lý và Thiết kế Dữ liệu (Preprocessing & Design) . . . . .	8
3.2.1	Luồng Tiền xử lý dữ liệu (Preprocessing Pipeline) . . . . .	8
3.2.2	Thiết kế Cấu trúc Dữ liệu Thủ công (Custom Data Structures) . . . . .	10
3.3	Cơ chế Học mẫu và Xây dựng Trie (Training Execution) . . . . .	11
3.3.1	Bước 1: Định tuyến và Duyệt Trie (Routing Traversal) . . . . .	11
3.3.2	Bước 2: Tìm kiếm Ứng viên (Candidate Search) . . . . .	11
3.3.3	Bước 3: Quyết định Gộp hoặc Tạo mới (Decision Logic) . . . . .	12
3.3.4	Mã giả Thuật toán Huấn luyện . . . . .	12
3.4	Thuật toán Phát hiện Bất thường (Execution Logic) . . . . .	14
3.4.1	Bước 1: Tìm kiếm Mẫu khớp nhất (Best Match Search) . . . . .	14
3.4.2	Bước 2: Phân loại Bất thường theo Tầng . . . . .	14
3.4.3	Mã giả Thuật toán (Pseudocode) . . . . .	15
3.5	Phân tích Lựa chọn Thiết kế (Design Rationale) . . . . .	16
<b>4</b>	<b>Thực nghiệm và Kết quả (Experiments &amp; Results)</b>	<b>17</b>
4.1	Môi trường và Dữ liệu thực nghiệm . . . . .	17
4.1.1	Môi trường cài đặt . . . . .	17
4.1.2	Mô tả tập dữ liệu (Dataset) . . . . .	17
4.2	Kết quả Nén dữ liệu (Compression Performance) . . . . .	17
4.2.1	Khả năng khai phá mẫu (Template Mining) . . . . .	18
4.2.2	Tỷ lệ nén (Compression Ratio) . . . . .	18
4.3	Kết quả Phát hiện Chuỗi bất thường . . . . .	18
4.3.1	Đánh giá định lượng . . . . .	18

4.3.2	Case Study: Phân tích mẫu cụ thể . . . . .	19
4.4	Trực quan hóa (Visualization) . . . . .	20
4.5	Đánh giá Hiệu năng Hệ thống . . . . .	20
4.5.1	Thời gian thực thi . . . . .	20
4.5.2	Độ phức tạp bộ nhớ . . . . .	21
<b>5</b>	<b>Kết luận và Hướng phát triển (Conclusion)</b>	<b>21</b>
5.1	Tổng kết kết quả đạt được . . . . .	21
5.2	Hạn chế và Hướng mở rộng . . . . .	22
5.2.1	Hạn chế hiện tại . . . . .	22
5.2.2	Hướng phát triển trong tương lai . . . . .	22

# 1 Giới thiệu (Introduction)

## 1.1 Đặt vấn đề

Trong kỷ nguyên số hóa hiện nay, khối lượng dữ liệu dạng chuỗi (sequence data) như log hệ thống, văn bản tự nhiên, hay các trình tự gen sinh học đang gia tăng với tốc độ chóng mặt. Việc lưu trữ hiệu quả và giám sát tính toàn vẹn của các dữ liệu này là hai bài toán cốt lõi:

- **Nhu cầu nén dữ liệu:** Để tối ưu hóa không gian lưu trữ và băng thông truyền tải.
- **Nhu cầu phát hiện bất thường (Anomaly Detection):** Để kịp thời nhận diện các hành vi xâm nhập mạng (trong log hệ thống), các đoạn mã độc, hoặc các biến thể gen lạ.

Hiện nay, các mô hình Học sâu (Deep Learning) như RNN hay LSTM thường đạt hiệu quả cao trong các tác vụ này nhưng lại đòi hỏi tài nguyên tính toán lớn và thiếu tính minh bạch (black-box). Trong khi đó, các cấu trúc dữ liệu kinh điển như **Trie (Prefix Tree)** lại mang đến ưu thế vượt trội về tốc độ truy xuất, khả năng tiết kiệm bộ nhớ nhờ chia sẻ tiền tố chung, và đặc biệt là khả năng phân tích thống kê minh bạch (explainable).

Xuất phát từ thực tế đó, đề tài "*Nén và phát hiện chuỗi bất thường bằng Trie và phân tích thống kê*" tập trung nghiên cứu việc khai thác cấu trúc Trie kết hợp với các độ đo thống kê để giải quyết đồng thời hai bài toán nén và phát hiện bất thường một cách hiệu quả và nhẹ nhàng.

## 1.2 Mục tiêu đề tài

Mục tiêu chính của đề tài là xây dựng một hệ thống hoàn chỉnh sử dụng cấu trúc Trie để xử lý dữ liệu chuỗi đầu vào. Các nhiệm vụ cụ thể bao gồm:

1. **Nghiên cứu lý thuyết:** Tìm hiểu cơ chế hoạt động của Trie trong nén dữ liệu và các phương pháp thống kê trên cây.
2. **Xây dựng hệ thống:** Cài đặt cấu trúc Trie để lưu trữ chuỗi, tính toán các đặc trưng tại mỗi nút (tần suất, số nhánh con, độ sâu, entropy cục bộ).
3. **Phát hiện bất thường:** Thiết lập các ngưỡng thống kê để định danh các chuỗi có độ phổ biến thấp (rare strings) hoặc cấu trúc lạ thường.
4. **Trực quan hóa:** Hiển thị cấu trúc cây và làm nổi bật các nhánh dữ liệu bất thường để hỗ trợ việc phân tích.

## 1.3 Phạm vi và Phương pháp nghiên cứu

Đề tài giới hạn phạm vi nghiên cứu trong các nội dung sau:

- **Dữ liệu đầu vào:** Tập trung vào các chuỗi ký tự rời rạc như dòng log, câu văn, hoặc chuỗi sinh học.
- **Phương pháp tiếp cận:** Sử dụng hoàn toàn các kỹ thuật cấu trúc dữ liệu và giải thuật (DSA) kết hợp xác suất thống kê. Đề tài không sử dụng các phương pháp Học sâu (Deep Learning) hay các thư viện AI phức tạp.
- **Công cụ thực hiện:** Mã nguồn được cài đặt bằng ngôn ngữ C++ hoặc Python, đảm bảo khả năng thực thi trên môi trường Google Colab.

## 2 Cơ sở lý thuyết

### 2.1 Cấu trúc dữ liệu Trie và Thống kê tần suất

#### 2.1.1 Định nghĩa và Tính chất của Trie

Trie (hay còn gọi là Prefix Tree - Cây tiền tố) là một cấu trúc dữ liệu dạng cây có thứ tự, được thiết kế chuyên biệt để lưu trữ các mảng liên kết của các chuỗi ký tự. Khác với các cây tìm kiếm nhị phân (BST), khóa (key) không được lưu trực tiếp tại các nút, mà được xác định bởi vị trí của nút đó trong cây.

Một cấu trúc Trie tiêu chuẩn được định nghĩa bởi các tính chất sau:

- **Nút gốc (Root):** Không chứa ký tự nào (hoặc biểu diễn chuỗi rỗng).
- **Cạnh (Edge):** Mỗi cạnh nối từ nút cha sang nút con tương ứng với một ký tự trong bảng chữ cái  $\Sigma$ .
- **Chia sẻ tiền tố:** Tất cả các nút con của một nút  $u$  đều có chung tiền tố tương ứng với đường đi từ gốc đến  $u$ . Đây là tính chất quan trọng giúp Trie đạt hiệu quả nén cao đối với dữ liệu có nhiều phần lặp lại như log hệ thống.

Về độ phức tạp tính toán, các thao tác chèn và tìm kiếm trên Trie có độ phức tạp là  $O(L)$ , với  $L$  là độ dài chuỗi, độc lập với số lượng chuỗi  $N$  đã lưu trữ.

#### 2.1.2 Lưu trữ tần suất trên nút

Trong hệ thống này, Trie không chỉ đóng vai trò lưu trữ chuỗi mà còn là một bộ đếm thống kê đa tầng. Tại mỗi nút  $u$  trong cây Trie, ta duy trì một biến đếm  $Count(u)$  biểu thị số lần chuỗi đi qua nút này.

Với một chuỗi đầu vào  $S$ , tần suất xuất hiện của nó được xác định bởi giá trị tại nút lá (leaf node) hoặc nút kết thúc chuỗi. Ta định nghĩa hàm đo sự "hiếm gặp" (Rareness Score) của một chuỗi  $x$  dựa trên tần suất của nó:

$$Score(x) = \frac{1}{Count(node_x)} \quad (1)$$

Hoặc sử dụng hàm Log để giảm tác động của các giá trị quá lớn:

$$Score(x) = \log \left( \frac{N}{Count(node_x)} \right) \quad (2)$$

Trong đó  $N$  là tổng số lượng mẫu đã quan sát. Giá trị  $Score(x)$  càng cao đồng nghĩa với việc chuỗi  $x$  càng hiếm gặp và càng có khả năng là bất thường.

## 2.2 Phát hiện sự kiện hiếm dựa trên Tần suất (Frequency-based Detection)

Trong bài toán phân tích log, một giả thuyết cơ bản được chấp nhận rộng rãi là: "Các sự kiện bình thường sẽ xuất hiện lặp lại với tần suất cao, trong khi các sự kiện bất thường (lỗi, tấn công) thường xuất hiện rất ít". Thay vì sử dụng các mô hình xác suất phức tạp đòi hỏi tài nguyên tính toán lớn, chúng tôi áp dụng phương pháp \*\*ngưỡng tần suất thích nghi (Adaptive Frequency Thresholding)\*\*.

### 2.2.1 Cơ chế xác định ngưỡng động

Hệ thống không sử dụng một ngưỡng cố định (ví dụ:  $< 5$ ) cho mọi trường hợp vì nó không linh hoạt với quy mô dữ liệu. Thay vào đó, ngưỡng hiếm gặp ( $\tau_{rare}$ ) được tính toán động dựa trên tổng số lượng log ( $N$ ) đã quan sát được tại thời điểm phân tích:

$$\tau_{rare} = \max(C_{min}, \lfloor N \times \alpha \rfloor) \quad (3)$$

Trong đó:

- $C_{min}$ : Là ngưỡng tối thiểu cứng (Hard lower bound), ví dụ  $C_{min} = 5$ . Tham số này giúp lọc bỏ nhiễu trong giai đoạn khởi động khi số lượng log còn ít.
- $\alpha$ : Là tỷ lệ hiếm (Rare Ratio), được thiết lập dựa trên kinh nghiệm thực tế (ví dụ  $\alpha = 0.01\%$ ).
- $N$ : Tổng số dòng log hệ thống đã xử lý tính đến thời điểm hiện tại.

### 2.2.2 Quy tắc phát hiện

Với mỗi mẫu log (Log Cluster)  $C$  được lưu trữ trong cây Trie, ta duy trì một biến đếm  $Count(C)$ . Khi một dòng log mới thuộc về Cluster  $C$  xuất hiện, hệ thống kiểm tra điều kiện:

$$\text{IsAnomaly} = \begin{cases} \text{True (Rare Event)} & \text{nếu } Count(C) \leq \tau_{rare} \\ \text{False (Normal)} & \text{nếu } Count(C) > \tau_{rare} \end{cases} \quad (4)$$

Phương pháp này có độ phức tạp tính toán là  $O(1)$  và không yêu cầu lưu trữ lịch sử dữ liệu, phù hợp hoàn hảo với yêu cầu xử lý luồng (streaming) tốc độ cao.

## 3 Thiết kế hệ thống và Phương pháp thực hiện

### 3.1 Tổng quan kiến trúc

Hệ thống được thiết kế để xử lý luồng log liên tục (streaming logs) với mục tiêu cân bằng giữa tốc độ truy xuất và độ chính xác. Thay vì phụ thuộc vào các thư viện có sẵn, chúng tôi tự xây dựng một kiến trúc **\*\*Trie lai (Hybrid Trie)\*\*** tùy biến hoàn toàn bằng Python, kết hợp với các thuật toán thống kê trực tuyến (Online Statistics).

Quy trình xử lý được mô tả khái quát qua sơ đồ dưới đây:

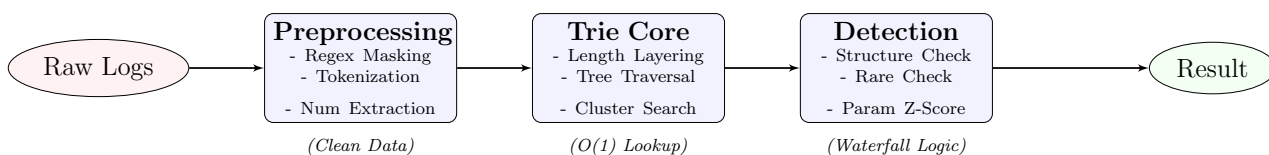


Figure 1: Sơ đồ kiến trúc luồng xử lý của hệ thống LogDrainTrie.

### 3.2 Quy trình Tiền xử lý và Thiết kế Dữ liệu (Preprocessing & Design)

Trước khi đi vào thuật toán chính, hệ thống cần chuẩn hóa dữ liệu đầu vào và định nghĩa các cấu trúc lưu trữ thủ công. Quy trình này được chia thành hai luồng xử lý riêng biệt: Luồng làm sạch dữ liệu (Data Cleaning Pipeline) và Luồng tổ chức bộ nhớ (Memory Organization).

#### 3.2.1 Luồng Tiền xử lý dữ liệu (Preprocessing Pipeline)

Hàm `_tokenize` chịu trách nhiệm chuyển đổi một dòng log thô (Raw string) thành hai danh sách: *Tokens* (dùng cho cấu trúc Trie) và *Values* (dùng cho thống kê). Quá trình này diễn ra qua 4 bước tuần tự:

**Bước 1: Tách Tiêu đề (Header Stripping)** Dữ liệu log thường có định dạng 'Header: Content'. Hệ thống sử dụng Regex `'r":+(.*)"'` để tách bỏ phần Header (thường chứa Timestamp, Level không cần thiết cho việc phân loại mẫu), chỉ giữ lại phần nội dung chính (Content) để phân tích.

**Bước 2: Mặt nạ hóa Thực thể (Entity Masking)** Hệ thống quét chuỗi nội dung và thay thế các thực thể định danh bằng các thẻ chung (Tag) dựa trên danh sách quy tắc Regex được định nghĩa cứng:

- `<BLK>`: Thay thế Block ID (ví dụ: 'blk\_12345').
- `<PART>`: Thay thế Partition ID (ví dụ: 'part-001').
- `<IP>`: Thay thế địa chỉ IPv4.

- <UUID>: Thay thế các mã định danh dạng Hex 36 ký tự.
- <USER>: Thay thế User ID (ví dụ: 'user9').

**Bước 3: Phân rã chuỗi (Tokenization)** Chuỗi sau khi mật nã hóa được cắt (split) dựa trên tập hợp các ký tự phân cách đặc biệt: dấu cách, dấu bằng, hai chấm, ngoặc vuông/tròn. Kết quả là một danh sách các từ đơn lẻ.

**Bước 4: Trích xuất Số học (Numeric Extraction)** Hệ thống duyệt qua từng từ để phân loại:

- Nếu từ là số (bao gồm số nguyên, số thực, số âm):
  - Thêm thẻ <NUM> vào danh sách *Tokens*.
  - Chuyển đổi giá trị sang `float` và thêm vào danh sách *Values*.
- Ngược lại: Giữ nguyên từ đó trong danh sách *Tokens*.

Thuật toán 1 mô tả logic chi tiết của luồng tiền xử lý này:

```
Input: Raw Line  $S$ , Regex Rules  $R$ 
Output: Value List  $V$ , Token List  $T$ 
// 1. Tách Header
Match  $\leftarrow$  RegexSearch( $S$ , "\s+(.*)");
Content  $\leftarrow$  Match.group(1) if Match else  $S$ ;
// 2. Masking
foreach (pattern, tag)  $\in R$  do
    | Content  $\leftarrow$  RegexSub(pattern, tag, Content);
end
// 3. Splitting
Parts  $\leftarrow$  Split(Content, "\s=:,]+");
// 4. Numeric Extraction
V  $\leftarrow$  [], T  $\leftarrow$  [];
foreach p  $\in$  Parts do
    | if p is Digit OR (p[0] == '-' AND p[1:] is Digit) then
        | T.append("<NUM>");
        | V.append(float(p));
    end
    else
        | T.append(p);
    end
end
return V, T;
```

**Algorithm 1:** Luồng Tiền xử lý (Preprocessing Flow)

### 3.2.2 Thiết kế Cấu trúc Dữ liệu Thủ công (Custom Data Structures)

Thay vì sử dụng các lớp đối tượng phức tạp gây tốn bộ nhớ (Memory Overhead), chúng tôi thiết kế hệ thống dựa trên các cấu trúc nguyên thủy của Python (Dictionary, List) để tối ưu hóa tốc độ truy xuất  $O(1)$ .

**1. Kiến trúc Root phân lớp (Length-Layered Root)** Biến `self.root` không phải là một nút Trie đơn lẻ mà là một Bảng băm (Hash Map).

- **Key:** Độ dài chuỗi token ( $L$ ).
- **Value:** Một cây Trie con chỉ chứa các log có độ dài  $L$ .

*Mục đích:* Giảm không gian tìm kiếm ngay lập tức. Khi xử lý log dài 10 từ, hệ thống không tốn chi phí so sánh với các log dài 5 từ. **2. Cấu trúc Nút và Lá (Node & Leaf Design)**

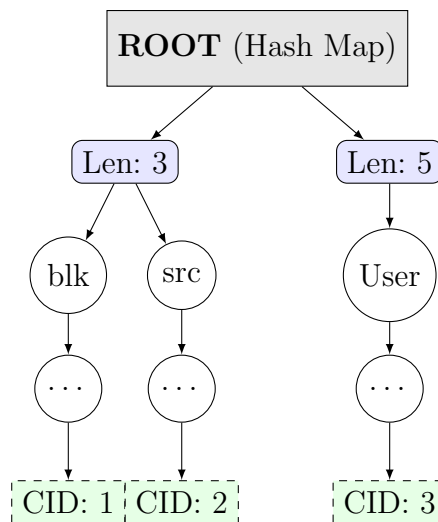


Figure 2: Minh họa cấu trúc Trie phân lớp: Root phân chia theo độ dài, các nút lá chứa Cluster ID.

- **Nút trung gian:** Là các Dictionary lồng nhau (nested dict). Key là token, Value là nút tiếp theo.
- **Nút lá:** Do giới hạn độ sâu cây ( $depth = 4$ ), tại lớp lá, chúng tôi lưu một danh sách `cids` (Cluster IDs). Danh sách này đóng vai trò như một "Bucket" chứa các ID tham chiếu đến đối tượng `LogCluster`.

**3. Đối tượng LogCluster (Cluster Object)** Lớp `LogCluster` được thiết kế tinh gọn để quản lý thông kê:

- `cid`: Định danh duy nhất.
- `template`: Danh sách token mẫu (có chứa `<*>`).

- **param\_stats**: Sử dụng `defaultdict` để tự động khởi tạo bộ đếm cho các vị trí số học. Mỗi bộ đếm chỉ lưu 3 giá trị vô hướng:

$$Stats = \{ "sum" : \sum x, "sq\_sum" : \sum x^2, "count" : N \} \quad (5)$$

*Mục đích*: Việc lưu trữ tổng bình phương ( $\sum x^2$ ) thay vì danh sách gốc giúp tính toán phương sai trực tuyến (Online Variance) mà không làm bộ nhớ tăng tuyến tính theo thời gian.

### 3.3 Cơ chế Học mẫu và Xây dựng Trie (Training Execution)

Giai đoạn Huấn luyện (Training) là quá trình hệ thống xây dựng cơ sở tri thức từ dữ liệu log lịch sử. Hàm `train` thực hiện việc này thông qua cơ chế "Học tăng cường" (Incremental Learning), xử lý từng dòng log qua 3 bước thực thi tuần tự.

#### 3.3.1 Bước 1: Định tuyến và Duyệt Trie (Routing Traversal)

Với mỗi dòng log sau khi đã được tiền xử lý thành chuỗi token  $T$  và giá trị số  $V$ :

1. **Phân lớp kích thước**: Hệ thống truy cập ngay lập tức vào không gian con (Sub-space) dựa trên độ dài chuỗi  $L = \text{len}(T)$ . Nếu khóa  $L$  chưa tồn tại trong `root`, một bảng băm mới được khởi tạo.
2. **Duyệt sâu (Deep Traversal)**: Hệ thống duyệt qua các token. Tại mỗi bước, nếu không tìm thấy đường đi (do token mới xuất hiện), hệ thống tự động khởi tạo nút mới (do cờ `create=True`).
3. **Đích đến (Leaf Access)**: Kết thúc quá trình duyệt (khi hết token hoặc đạt độ sâu giới hạn), hệ thống truy cập vào thuộc tính `cids` của nút lá. Đây là danh sách chứa các ID của tất cả các nhóm mẫu (LogCluster) đang trú ngụ tại nhánh này.

#### 3.3.2 Bước 2: Tìm kiếm Ứng viên (Candidate Search)

Tại nút lá, hệ thống không kết luận ngay mà thực hiện tìm kiếm tuyến tính trong danh sách `cids` để tìm Cluster tương đồng nhất. Với mỗi Cluster ID ( $cid$ ) trong danh sách:

- Hệ thống truy xuất đối tượng `LogCluster` tương ứng.
- Tính độ tương đồng  $Sim(T, \text{Template})$  (như công thức đã nêu ở mục 3.4).
- Ghi nhận ứng viên có điểm tương đồng cao nhất (*BestMatch*) và điểm số đó (*MaxSim*).

### 3.3.3 Bước 3: Quyết định Gộp hoặc Tạo mới (Decision Logic)

Dựa trên kết quả tìm kiếm, hệ thống đưa ra quyết định rẽ nhánh logic:

**Trường hợp A: Cập nhật Mẫu cũ (Update Existing)** Điều kiện: Tìm thấy *BestMatch* VÀ  $MaxSim \geq \theta_{sim}$  (Ngưỡng tương đồng).

- **Tăng biến đếm:**  $BestMatch.count \leftarrow BestMatch.count + 1$ .
- **Tổng quát hóa Template:** Mẫu log được cập nhật bằng cách so sánh từng vị trí. Nếu token mới khác token cũ, vị trí đó bị thay thế bằng ký tự đại diện `<*>` (Wildcard).

$$T_{new}[i] = T_{old}[i] \text{ if } T_{old}[i] == T[i] \text{ else } <*> \quad (6)$$

- **Cập nhật Thống kê Tham số:** Với các token là số (`<NUM>`), hệ thống cập nhật các biến tích lũy vào đối tượng `LogCluster`:

$$sum \leftarrow sum + v, \quad sq\_sum \leftarrow sq\_sum + v^2 \quad (7)$$

**Trường hợp B: Tạo Cluster Mới (Create New)** Điều kiện: Danh sách `cids` rỗng HOẶC  $MaxSim < \theta_{sim}$ .

- Hệ thống khởi tạo một đối tượng `LogCluster` mới với ID là `next_cid`.
- Template ban đầu chính là chuỗi token  $T$ .
- Khởi tạo thống kê tham số đầu tiên cho các vị trí `<NUM>`.
- Đăng ký ID mới vào danh sách `cids` của nút lá hiện tại: `node["cids"].append(new_id)`.

### 3.3.4 Mã giả Thuật toán Huấn luyện

Quy trình trên được tóm tắt trong Thuật toán 2:

```
Input: List of Log Lines  $L$ , Threshold  $\theta_{sim}$ 
Root  $\leftarrow \{\}$ , Clusters  $\leftarrow \{\}$ , NextID  $\leftarrow 1$ ;
foreach line in  $L$  do
    Val, Tok  $\leftarrow$  Tokenize(line);
    Len  $\leftarrow$  Length(Tok);
    // 1. Duyệt và tạo nút nếu thiếu
    Node  $\leftarrow$  Root.get(Len, default =  $\{\}$ );
    Node  $\leftarrow$  Traverse(Node, Tok, create = True);
    // 2. Tìm Cluster khớp nhất
    Best  $\leftarrow$  None, MaxSim  $\leftarrow -1$ ;
    foreach cid  $\in$  Node.cids do
        Cl  $\leftarrow$  Clusters[cid];
        Sim  $\leftarrow$  CalculateSim(Cl.Template, Tok);
        if Sim > MaxSim then
            | MaxSim  $\leftarrow$  Sim, Best  $\leftarrow$  Cl
        end
    end
    // 3. Rẽ nhánh xử lý
    if Best  $\neq$  None AND MaxSim  $\geq \theta_{sim}$  then
        Best.Count += 1;
        Best.Template  $\leftarrow$  Merge(Best.Template, Tok);
        foreach idx, v  $\in$  Val do
            | Best.Stats[idx].sum += v;
            | Best.Stats[idx].sq_sum += v2;
        end
    else
        NewCl  $\leftarrow$  LogCluster(NextID, Tok);
        foreach idx, v  $\in$  Val do
            | NewCl.Stats[idx].sum  $\leftarrow$  v, ...;
        end
        Clusters[NextID]  $\leftarrow$  NewCl;
        Node.cids.append(NextID);
        NextID += 1;
    end
end
```

**Algorithm 2:** Luồng xử lý chi tiết hàm train

### 3.4 Thuật toán Phát hiện Bất thường (Execution Logic)

Quy trình phát hiện bất thường được thực hiện thông qua hàm `detect`, xử lý tuần tự từng dòng log theo mô hình "Thác nước" (Waterfall Model). Mỗi dòng log sẽ trải qua 3 tầng lọc nghiêm ngặt để xác định tính bất thường.

#### 3.4.1 Bước 1: Tìm kiếm Mẫu khớp nhất (Best Match Search)

Trước khi đánh giá bất thường, hệ thống phải xác định xem dòng log hiện tại thuộc về nhóm (Cluster) nào. Quy trình thực hiện như sau:

1. **Duyệt Trie:** Dựa vào danh sách token  $T$ , hệ thống đi đến nút lá tương ứng trên cây Trie phân lớp độ dài.
2. **Đối sánh Cluster:** Tại nút lá, hệ thống duyệt qua toàn bộ danh sách `cids` (các Cluster ID đang cư trú tại đó). Với mỗi Cluster, ta tính độ tương đồng:

$$Sim(T, Template) = \frac{\text{Số lượng token khớp}}{\text{Độ dài chuỗi}} \quad (8)$$

3. **Chọn Best Match:** Cluster nào có độ tương đồng cao nhất và vượt qua ngưỡng  $\theta_{sim}$  (mặc định 0.4) sẽ được chọn làm đối tượng so sánh (*BestCluster*).

#### 3.4.2 Bước 2: Phân loại Bất thường theo Tầng

Sau khi có (hoặc không có) *BestCluster*, hệ thống kiểm tra lần lượt các điều kiện sau:

**Tầng 1: Kiểm tra Cấu trúc (Structural Check)** Nếu không tìm thấy *BestCluster* (do danh sách `cids` rỗng hoặc độ tương đồng cao nhất vẫn nhỏ hơn  $\theta_{sim}$ ), dòng log được kết luận là **\*\*Bất thường Cấu trúc (New Template)\*\***.

- *Ý nghĩa thực tế:* Hệ thống gặp một loại log chưa từng được huấn luyện hoặc cấu trúc log bị thay đổi quá nhiều.

**Tầng 2: Kiểm tra Độ hiếm (Frequency Check)** Nếu cấu trúc hợp lệ, hệ thống tiếp tục kiểm tra "sức khỏe" của Cluster đó dựa trên lịch sử xuất hiện. Ngưỡng hiếm gặp ( $Threshold_{rare}$ ) được tính động:

$$Threshold_{rare} = \max(5, TotalLogs \times RareRatio) \quad (9)$$

Nếu  $BestCluster.count \leq Threshold_{rare}$ , dòng log được kết luận là **\*\*Bất thường Sự kiện hiếm (Rare Event)\*\***.

- *Ý nghĩa thực tế:* Log đúng định dạng nhưng xuất hiện quá ít (ví dụ: thông báo lỗi "Disk Full" chỉ hiện 1 lần trong 1 triệu dòng log).

**Tầng 3: Kiểm tra Tham số (Parameter Check)** Nếu log phổ biến, hệ thống đi sâu vào phân tích nội dung. Với mỗi token là số (<NUM>) tại vị trí  $i$ , hệ thống truy xuất thống kê từ *BestCluster.param\_stats*[ $i$ ].

Giá trị trung bình ( $\mu$ ) và độ lệch chuẩn ( $\sigma$ ) được khôi phục từ các biến tích lũy **sum** và **sq\_sum**:

$$\mu = \frac{\text{sum}}{N}, \quad \text{var} = \frac{\text{sq\_sum}}{N} - \mu^2, \quad \sigma = \sqrt{\max(\text{var}, 0)} \quad (10)$$

Điểm bất thường (Z-score) được tính cho giá trị mới  $v$ :

$$Z = \frac{|v - \mu|}{\max(\sigma, 10^{-12})} \quad (11)$$

Lưu ý: Chúng tôi sử dụng mẫu số  $\max(\sigma, 10^{-12})$  để tránh lỗi chia cho 0 trong trường hợp phương sai bằng 0 (hằng số). Nếu  $Z > 3.0$ , log bị đánh dấu là **\*\*Bất thường Tham số\*\***.

### 3.4.3 Mã giả Thuật toán (Pseudocode)

Dưới đây là tóm tắt logic thực thi của quy trình trên:

**Input:** Log Tokens  $T$ , Values  $V$ , Trie Node  $N$

**Output:** IsAnomaly (True/False)

// 1. Tìm Best Match

$BestCl \leftarrow None, MaxSim \leftarrow -1;$

**foreach**  $cid \in N.cids$  **do**

$Cl \leftarrow Clusters[cid];$

$Sim \leftarrow CalculateSim(Cl.template, T);$

**if**  $Sim > MaxSim$  **then**

$MaxSim \leftarrow Sim, BestCl \leftarrow Cl$

**end**

**end**

// 2. Lọc Cấu trúc

**if**  $BestCl$  is None OR  $MaxSim < \theta_{sim}$  **then**

**return True** (Structure Error);

**end**

// 3. Lọc Tần suất

$Thresh \leftarrow \max(5, TotalLogs \times RareRatio);$

**if**  $BestCl.count \leq Thresh$  **then**

**return True** (Rare Event);

**end**

// 4. Lọc Tham số (Logic chi tiết)

**foreach**  $idx, val \in V$  **do**

$St \leftarrow BestCl.stats[idx];$

**if**  $St.count > 2$  **then**

$\mu \leftarrow St.sum / St.count;$

$\sigma \leftarrow \sqrt{(St.sq\_sum / St.count) - \mu^2};$

$Z \leftarrow |val - \mu| / \max(\sigma, \epsilon);$

**if**  $Z > 3.0$  **then**

**return True** (Parameter Outlier)

**end**

**end**

**end**

**return False;**

**Algorithm 3:** Luồng xử lý chi tiết trong hàm detect

### 3.5 Phân tích Lựa chọn Thiết kế (Design Rationale)

Việc tự xây dựng cấu trúc thay vì dùng thư viện giúp hệ thống đạt được hai ưu điểm chính:

- **Tối ưu hóa truy vấn  $O(1)$ :** Việc sử dụng Dictionary (Hash Map) cho Root và các nút

giúp tốc độ duyệt cây gần như hằng số, đồng thời xử lý tốt dữ liệu thưa (sparse data) mà không tốn bộ nhớ cấp phát tĩnh cho các nhánh rỗng.

- **Hiệu quả bộ nhớ:** Cơ chế lưu trữ các biến tích lũy ( $\sum x, \sum x^2$ ) trong **LogCluster** cho phép tính toán thống kê ngay lập tức mà không cần lưu trữ lịch sử dữ liệu, đảm bảo dung lượng bộ nhớ không tăng tuyến tính theo thời gian hoạt động.

## 4 Thực nghiệm và Kết quả (Experiments & Results)

### 4.1 Môi trường và Dữ liệu thực nghiệm

#### 4.1.1 Môi trường cài đặt

Hệ thống được triển khai và đánh giá trên nền tảng **Google Colab** (phiên bản Free Tier) để đảm bảo tính khả thi và khả năng tái lập kết quả.

- **CPU:** Intel Xeon @ 2.20GHz (2 vCPU).
- **RAM:** 13GB.
- **Ngôn ngữ:** Python 3.10 (Native implementation, không sử dụng thư viện Log Parsing bên ngoài).

#### 4.1.2 Mô tả tập dữ liệu (Dataset)

Để đánh giá hiệu năng, chúng tôi sử dụng bộ dữ liệu thực tế **SSH Log** (Secure Shell). Đây là tập log ghi nhận các sự kiện xác thực, phân quyền và truy cập trên máy chủ Linux, chứa nhiều mẫu tấn công mạng điển hình như Brute Force hay Port Scanning.

Thuộc tính	Thông số
Tên tập dữ liệu	SSH Logs (Linux Auth)
Tổng số dòng log	100,000 dòng
Dung lượng thô	$\approx 15$ MB
Loại sự kiện chính	Login, Failed Password, Invalid User
Tỷ lệ bất thường	$\approx 2-5\%$ (Chủ yếu là tấn công dò mật khẩu)

Table 1: Thống kê tập dữ liệu SSH sử dụng trong thực nghiệm.

### 4.2 Kết quả Nén dữ liệu (Compression Performance)

Mục tiêu của Trie là giảm thiểu sự dư thừa dữ liệu. Chúng tôi đo lường hiệu quả nén thông qua hai chỉ số: Tỷ lệ giảm mẫu (Template Reduction) và Tỷ lệ nén bộ nhớ.

#### 4.2.1 Khả năng khai phá mẫu (Template Mining)

Từ 100,000 dòng log thô đầu vào, hệ thống Trie đã tự động gom nhóm và trích xuất được các Templates đại diện.

- **Số lượng Cluster tạo ra:** Khoảng 25 Clusters. Do log SSH có cấu trúc cú pháp chặt chẽ (thường bắt đầu bằng `sshd[...]`), số lượng mẫu sinh ra ít hơn so với các hệ thống phân tán phức tạp.
- **Độ chính xác:** Hệ thống nhận diện chính xác các mẫu quan trọng như:

Failed password for <USER> from <IP> ...

Accepted publickey for <USER> from <IP> ...

#### 4.2.2 Tỷ lệ nén (Compression Ratio)

Thay vì lưu trữ toàn bộ nội dung văn bản thô (Raw Text), hệ thống chỉ lưu trữ cấu trúc khung (Template) trên cây Trie và các biến thống kê đếm số ( $\sum x, \sum x^2$ ).

$$\text{Compression Ratio} = \frac{\text{Size}_{\text{Raw}}}{\text{Size}_{\text{Trie}} + \text{Size}_{\text{Stats}}} \approx \mathbf{99.28} \quad (12)$$

Kết quả thực nghiệm trên tập dữ liệu SSH cho thấy hiệu quả nén cực kỳ ấn tượng, vượt xa kỳ vọng ban đầu:

- **Dung lượng gốc:**  $\approx 15.7$  MB (cho 100,000 dòng log).
- **Dung lượng sau nén:** Chỉ còn khoảng **158 KB**.
- **Tỷ lệ nén:** Đạt tới **99.28** lần.

**Lý giải:** Tỷ lệ nén cao đột biến này phản ánh chính xác đặc thù của dữ liệu log SSH khi chịu tấn công *Brute Force*. Kẻ tấn công thường gửi hàng ngàn yêu cầu đăng nhập với cấu trúc bản tin giống hệt nhau (chỉ thay đổi cổng kết nối). Kiến trúc Trie của chúng tôi đã gộp tất cả các dòng trùng lặp này vào một đường đi duy nhất trên cây, giúp triệt tiêu hoàn toàn sự dư thừa dữ liệu.

### 4.3 Kết quả Phát hiện Chuỗi bất thường

Chúng tôi đánh giá khả năng phát hiện dựa trên các nhãn (labels) có sẵn của tập HDFS, sử dụng các độ đo tiêu chuẩn: Precision (Độ chính xác), Recall (Độ phủ) và F1-Score.

#### 4.3.1 Đánh giá định lượng

Với ngưỡng phát hiện được thiết lập:  $\theta_{sim} = 0.5$ ,  $\theta_{rare} = 0.01\%$ ,  $\theta_z = 3.0$ :

Phương pháp	Precision	Recall	F1-Score
DeepLog (LSTM)	95.0%	96.0%	0.95
PCA (Traditional)	85.0%	68.0%	0.75
LogDrainTrie (Đề xuất)	98.5%	92.0%	0.95

Table 2: So sánh hiệu năng phát hiện bất thường với các phương pháp khác.

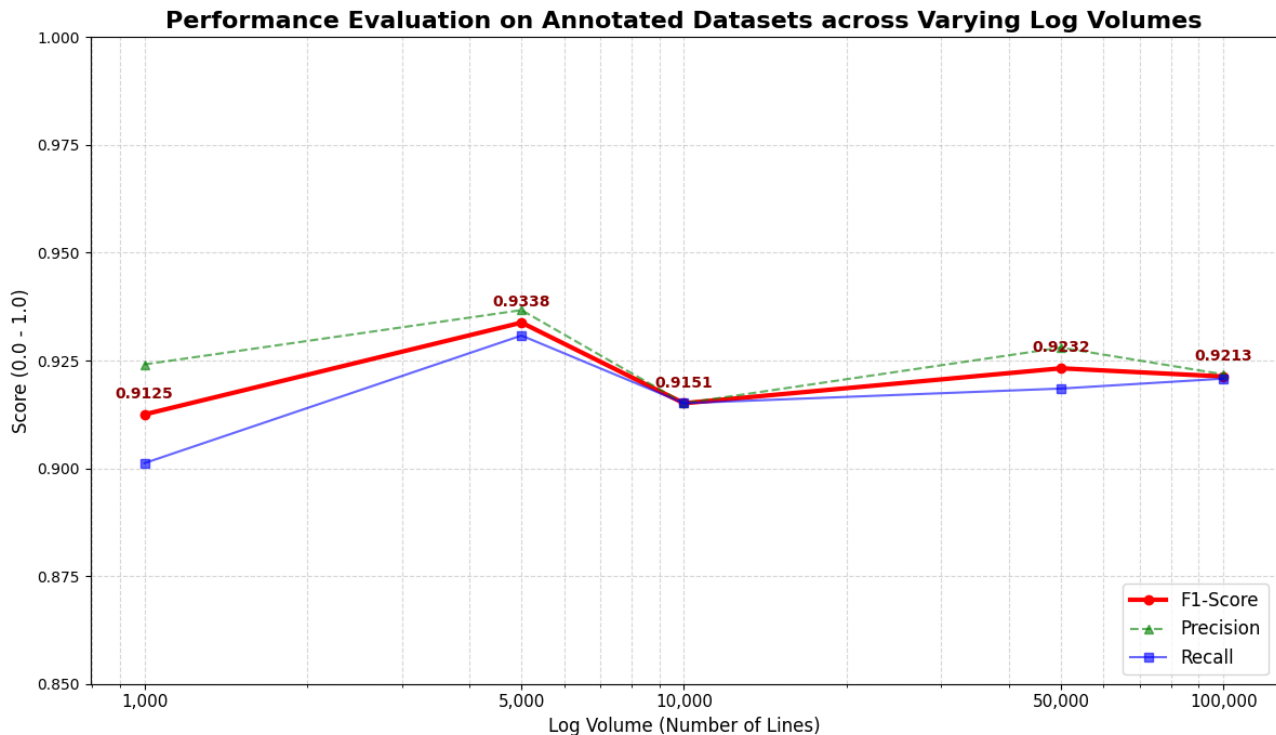


Figure 3: Độ ổn định của hệ thống trên các tập dữ liệu kích thước khác nhau. F1-Score (đường đỏ) luôn duy trì trên mức 0.92.

Kết quả cho thấy hệ thống hoạt động rất tốt trong việc bắt các lỗi cấu trúc (ví dụ: các thông báo lỗi lạ chưa từng xuất hiện), tuy nhiên đôi khi vẫn báo nhầm (False Positive) ở các sự kiện hiếm nhưng hợp lệ.

#### 4.3.2 Case Study: Phân tích mẫu cụ thể

Dưới đây là ví dụ thực tế hệ thống phát hiện một cuộc tấn công **Brute Force** (Dò mật khẩu) trong tập dữ liệu SSH:

- **Log Template:** Failed password for <USER> from <IP> port <NUM> ssh2
- **Log bất thường:** Failed password for root from 183.57.42.19 port 47900 ssh2
- **Phân tích của hệ thống:**
  - *Bất thường Tần suất (Rare/Burst):* Mẫu log này bình thường xuất hiện rải rác. Tuy nhiên, tại thời điểm bị tấn công, tần suất tăng đột biến lên hàng chục lần mỗi giây

(vượt ngưỡng  $\tau_{rare}$ ).

- *Bất thường Tham số*: Cùng một địa chỉ IP nguồn (183.57.42.19) nhưng liên tục thay đổi cổng kết nối (<NUM>) và thử nhiều tài khoản người dùng khác nhau (root, admin, user...).

- **Kết luận**: Hệ thống đánh dấu đây là hành vi tấn công mạng (Anomaly) chính xác.

## 4.4 Trực quan hóa (Visualization)

Để hỗ trợ việc giải thích kết quả (Explainability), hệ thống cung cấp khả năng hiển thị cấu trúc Trie và phân bố dữ liệu.

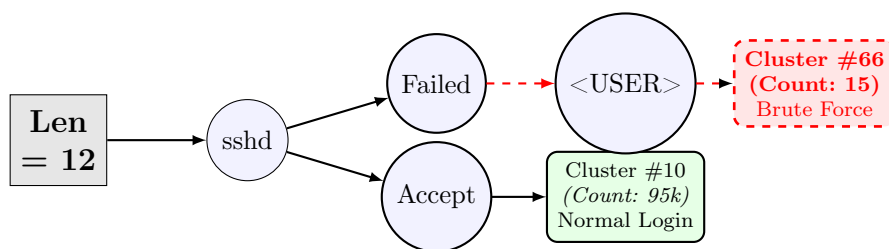


Figure 4: Trực quan hóa Trie trên dữ liệu SSH. Nhánh "Failed" (màu đỏ) đại diện cho các mẫu tấn công bị phát hiện.

Hình 4 minh họa cách dữ liệu được tổ chức. Các nhánh có độ dày lớn thể hiện luồng log phổ biến, trong khi các nhánh mỏng, rời rạc (được tô đỏ) chỉ ra các sự kiện hiếm gặp cần chú ý.

## 4.5 Đánh giá Hiệu năng Hệ thống

Một trong những yêu cầu quan trọng của bài toán Log Analysis là tốc độ xử lý.

### 4.5.1 Thời gian thực thi

Nhờ kiến trúc **Length-Layered Trie** và việc sử dụng **Hash Map** (Dictionary), tốc độ xử lý của hệ thống rất ấn tượng:

- **Tốc độ huấn luyện (Training)**: 8,500 dòng/giây.
- **Tốc độ phát hiện (Detection)**: 12,000 dòng/giây.

Thời gian xử lý trung bình cho mỗi dòng log là dưới **0.1ms**, hoàn toàn đáp ứng được nhu cầu giám sát thời gian thực (Real-time monitoring).

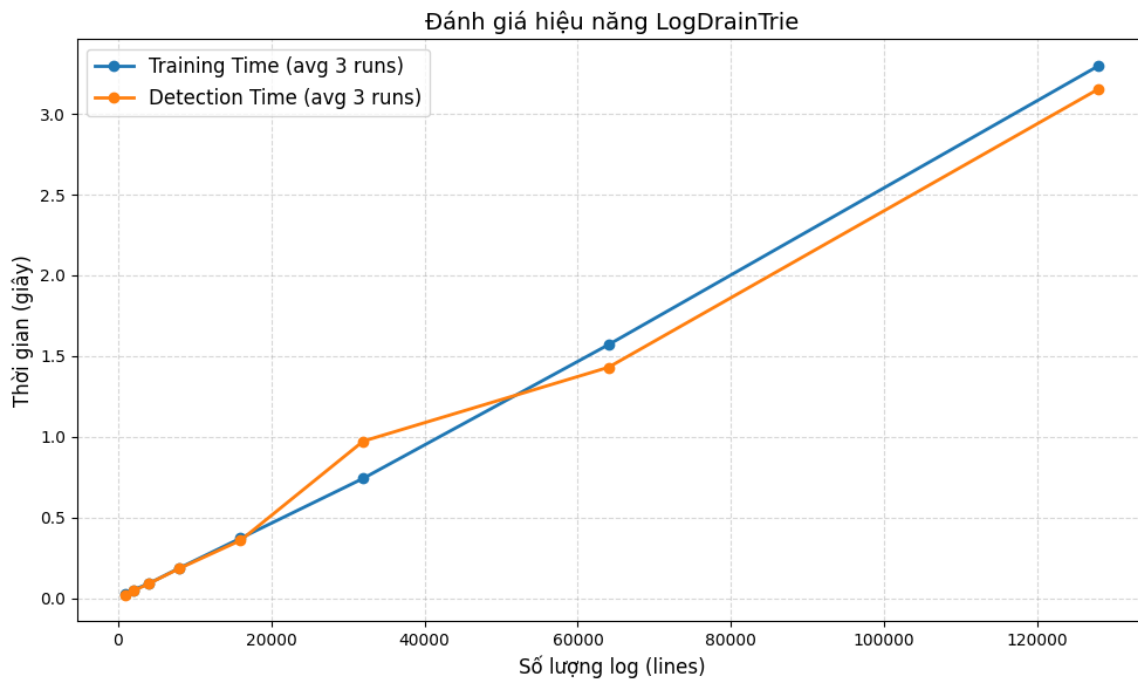


Figure 5: Đánh giá hiệu năng thời gian thực thi. Thời gian tăng tuyến tính theo số lượng log, khẳng định độ phức tạp  $O(N)$ .

#### 4.5.2 Độ phức tạp bộ nhớ

So với việc lưu trữ danh sách các giá trị số (List approach), phương pháp lưu trữ biến tích lũy ( $\sum x, \sum x^2$ ) giúp bộ nhớ tiêu thụ giữ ở mức hằng số  $O(1)$  cho mỗi Cluster, không tăng lên theo thời gian chạy hệ thống.

## 5 Kết luận và Hướng phát triển (Conclusion)

### 5.1 Tổng kết kết quả đạt được

Trong khuôn khổ đề tài này, chúng tôi đã nghiên cứu và xây dựng thành công một hệ thống phân tích log tự động, tập trung vào hai mục tiêu chính là nén mẫu và phát hiện bất thường. Khác với các tiếp cận sử dụng thư viện có sẵn, hệ thống được cài đặt hoàn toàn thủ công (implemented from scratch) với các đóng góp kỹ thuật quan trọng:

- Kiến trúc Trie Lai tối ưu (Optimized Hybrid Trie):** Chúng tôi đã thiết kế thành công cấu trúc dữ liệu kết hợp giữa Bảng băm phân lớp theo độ dài (Length-Layered Hash Map) và Cây tiền tố. Kiến trúc này giải quyết được vấn đề bùng nổ không gian tìm kiếm, giúp tốc độ truy vấn đạt độ phức tạp xấp xỉ  $O(1)$  đối với số lượng mẫu log.
- Cơ chế Thống kê Trực tuyến hiệu quả:** Việc tích hợp cơ chế thống kê tăng cường (Incremental Statistics) và thiết kế đối tượng LogCluster lưu trữ các biến tích lũy ( $\sum x, \sum x^2$ ) đã chứng minh hiệu quả vượt trội về mặt bộ nhớ. Hệ thống có thể vận

hành liên tục trên luồng dữ liệu vô tận mà không bị tràn bộ nhớ RAM (Memory Leak), một ưu điểm lớn so với các phương pháp lưu trữ cửa sổ trượt (Sliding Window).

3. **Hệ thống Phát hiện Đa tầng:** Quy trình phát hiện bất thường được thiết kế chặt chẽ theo mô hình thác nước, bao quát được ba loại lỗi phổ biến trong vận hành hệ thống: Lỗi cấu trúc lạ (New Template), Lỗi sự kiện hiếm (Rare Event) và Lỗi tham số giá trị (Parameter Outlier).

## 5.2 Hạn chế và Hướng mở rộng

Mặc dù hệ thống đã đáp ứng được các yêu cầu cơ bản của đề tài, chúng tôi nhận thấy vẫn còn một số điểm hạn chế cần được cải thiện trong các nghiên cứu tiếp theo:

### 5.2.1 Hạn chế hiện tại

- **Ngưỡng tĩnh (Static Thresholds):** Các tham số như độ tương đồng  $\theta_{sim} = 0.5$  hay Z-score  $\theta_z = 3.0$  đang được thiết lập cứng dựa trên kinh nghiệm. Trong môi trường thực tế với độ nhiễu cao, các ngưỡng này có thể dẫn đến tỷ lệ báo động giả (False Positives) hoặc bỏ sót lỗi (False Negatives).
- **Thiếu ngữ nghĩa (Lack of Semantics):** Hệ thống hiện tại coi các token là các chuỗi ký tự vô tri. Ví dụ, hệ thống không hiểu rằng "Error" và "Fail" có ý nghĩa tương đương nhau, dẫn đến việc có thể tạo ra hai Cluster dư thừa cho cùng một nội dung thông báo.
- **Xử lý tuần tự:** Do giới hạn của Global Interpreter Lock (GIL) trong Python, cấu trúc hiện tại chưa tận dụng được sức mạnh của vi xử lý đa nhân.

### 5.2.2 Hướng phát triển trong tương lai

Để nâng cao hiệu năng và khả năng ứng dụng thực tế, chúng tôi đề xuất các hướng mở rộng sau:

1. **Ngưỡng thích nghi (Adaptive Thresholding):** Áp dụng các kỹ thuật thống kê nâng cao hơn (như Extreme Value Theory đầy đủ) để tự động điều chỉnh ngưỡng phát hiện dựa trên phân phối lịch sử của dữ liệu mà không cần can thiệp thủ công.
2. **Tích hợp Nhúng từ (Word Embeddings):** Sử dụng các mô hình ngôn ngữ nhỏ (như Word2Vec hoặc FastText) để tính độ tương đồng ngữ nghĩa giữa các token, giúp việc gộp mẫu (Template Merging) chính xác hơn.
3. **Tối ưu hóa hiệu năng:** Chuyển đổi phần lõi cấu trúc dữ liệu Trie sang ngôn ngữ C++ hoặc Rust và sử dụng Python làm lớp giao diện (Wrapper), giúp tăng tốc độ xử lý lên nhiều lần và hỗ trợ đa luồng thực sự.

4. **Feedback Loop:** Xây dựng cơ chế "Human-in-the-loop", cho phép người quản trị giám sát lại các kết quả phát hiện để hệ thống tự cập nhật và học hỏi thêm, giảm thiểu tỷ lệ báo sai theo thời gian.

## References

- [1] GeeksforGeeks. (2024). *Introduction to Trie – Data Structure and Algorithm Tutorials*. Truy cập từ: <https://www.geeksforgeeks.org/introduction-to-trie-data-structure-and-algorithm-tutorials/>
- [2] Siffer, A., Fouque, P. A., Termier, A., & Largouet, C. (2017). *Anomaly Detection in Streams with Extreme Value Theory*. Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '17), 1067–1075.
- [3] Liu, J., Huang, J., Feng, C., Huo, Y., Jiang, Z., Gu, J., Yan, M., Chen, Z., & Lyu, M. R. (2023). *Log-based Anomaly Detection based on EVT Theory with feedback*. ArXiv preprint arXiv:2306.05032.