Chẩn đoán và ngăn chặn lỗi `NameError: name 'Retry' is not defined` trong trích xuất dữ liệu web tự động

Ngày: 29/09/2025

Giới thiệu

Trích xuất dữ liệu web tự động, một nền tảng cho nhiều ứng dụng dựa trên dữ liệu, phụ thuộc rất nhiều vào mã nguồn mạnh mẽ và kiên cường để điều hướng sự không thể đoán trước vốn có của internet (Thực tiễn tốt nhất khi cào web). Các nhà phát triển thường xuyên gặp phải các sự cố tạm thời như hết thời gian chờ mạng, giới hạn tỷ lệ từ phía máy chủ và cấu trúc trang web bất ngờ, đòi hỏi các cơ chế xử lý lỗi tinh vi. Trong số các lỗi Python phổ biến có thể làm gián đoạn các hoạt động cào web, lỗi `NameError: name 'Retry' is not defined` nổi bật như một vấn đề đặc biệt khó chịu, nhưng thường khá dễ giải quyết. Báo cáo này nhằm mục đích cung cấp một hướng dẫn toàn diện để chẩn đoán và ngăn chặn `NameError` cụ thể này trong bối cảnh trích xuất dữ liệu web tự động, khám phá nguyên nhân gốc rễ, các kịch bản phổ biến và các chiến lược thực tế để đảm bảo sự ổn định và đáng tin cậy của các dự án cào web (Hướng dẫn xử lý lỗi Python). Hiểu và giảm thiểu lỗi này là rất quan trọng để duy trì luồng dữ liệu liên tục và giảm chi phí phát triển trong các nỗ lực cào web.

Mục lục

Hiểu về `NameError` và cơ chế thử lại

Mổ xẻ lỗi `NameError: name 'Retry' is not defined` trong Python

Thông báo lỗi `name 'Retry' is not defined` là một trường hợp cụ thể của ngoại lệ `NameError` trong Python. Một `NameError` xảy ra khi trình thông dịch Python gặp một tên (biến, hàm, lớp, mô-đun) chưa được gán hoặc nhập vào phạm vi hiện tại tại thời điểm sử dụng (Python Software Foundation). Trong ngữ cảnh của nhật ký `WARNING:cli_executor:Error in session a5ce5514-1874-42b9-9393-ac6009c1d57d: Error processing https://batdongsan.com.vn/ban-can-ho-chung-cu-tp-hcm: name 'Retry' is not defined`, điều này chỉ ra rằng chương trình `cli_executor`, trong khi cố gắng xử lý URL được chỉ định, đã cố gắng sử dụng một thực thể có tên `Retry` mà không thể truy cập đúng cách.

Các nguyên nhân phổ biến gây ra `NameError` như vậy bao gồm: * **Lỗi đánh máy:** Lỗi chính tả của tên biến, hàm hoặc lớp. Ví dụ, nếu một thư viện cung cấp hàm `retry`, nhưng mã cố gắng gọi `Retry` (với chữ 'R' viết hoa), một `NameError` sẽ xảy ra. * **Thiếu câu lệnh nhập:** Nếu `Retry` được dự định là một lớp hoặc hàm từ một thư viện bên ngoài (ví dụ: `tenacity`, `retrying`, `backoff`) hoặc một mô-đun khác trong dự án, nó phải được nhập rõ ràng bằng cách sử dụng các câu lệnh `import` hoặc `from ... import ...`. Nếu không có lệnh nhập chính xác, Python không thể định vị định nghĩa của `Retry` (Real Python). * **Các vấn đề về phạm vi:** Tên `Retry` có thể được định nghĩa, nhưng không nằm trong phạm vi hiện tại nơi nó đang được

truy cập. Ví dụ, nếu `Retry` được định nghĩa bên trong một hàm, nó không thể được truy cập trực tiếp bên ngoài hàm đó trừ khi nó được trả về hoặc truyền đi một cách thích hợp. * **Các phụ thuộc chưa được cài đặt:** Nếu `Retry` là một phần của gói bên thứ ba, gói đó phải được cài đặt trong môi trường Python nơi `cli_executor` đang chạy. Nếu gói bị thiếu, câu lệnh nhập có thể sẽ thất bại, hoặc nếu lệnh nhập có điều kiện hoặc nằm trong khối `try-except`, `NameError` có thể xuất hiện sau đó khi `Retry` thực sự được gọi. * **Các phụ thuộc vòng tròn:** Trong các dự án phức tạp, các lệnh nhập vòng tròn đôi khi có thể dẫn đến các tình huống mà một mô-đun không được khởi tạo hoàn toàn khi một mô-đun khác cố gắng truy cập nội dung của nó, có khả năng dẫn đến `NameError` (Stack Overflow).

Trong kịch bản cụ thể của `cli_executor` xử lý một URL như `https://batdongsan.com.vn/ban-can-ho-chung-cu-tp-hcm`, thực thể `Retry` có khả năng được dự định là một cơ chế để xử lý các vấn đề mạng tạm thời, máy chủ không khả dụng hoặc giới hạn tỷ lệ có thể xảy ra trong quá trình yêu cầu web. Việc không định nghĩa `Retry` có nghĩa là bất kỳ logic kiên cường dự kiến nào dựa trên cấu trúc này đều không được kích hoạt, có khả năng dẫn đến lỗi ngay lập tức khi gặp các chướng ngại vật tạm thời thay vì thử lại hoạt động. Gỡ lỗi sẽ bao gồm việc kiểm tra mã nguồn của `cli_executor` xung quanh điểm lỗi, xác minh các câu lệnh nhập, kiểm tra các gói đã cài đặt (`pip freeze`) và đảm bảo môi trường thực thi khớp với môi trường phát triển.

Các nguyên tắc cốt lõi và sự cần thiết của cơ chế thử lại trong hệ thống phân tán

Cơ chế thử lại là các thành phần cơ bản trong thiết kế các hệ thống phần mềm mạnh mẽ và kiên cường, đặc biệt là những hệ thống tương tác với các dịch vụ bên ngoài, cơ sở dữ liệu hoặc tài nguyên mạng, chẳng hạn như `cli_executor` xử lý nội dung web từ `batdongsan.com.vn`. Mục đích chính của chúng là tăng cường độ tin cậy bằng cách tự động thử lại các hoạt động đã thất bại do các lỗi tạm thời, không gây tử vong (Microsoft Azure Architecture Center). Những lỗi tạm thời này là tạm thời và thường tự giải quyết trong một thời gian ngắn, khiến việc thử lại trở thành một chiến lược hiệu quả.

Các loại lỗi chính được xử lý bởi logic thử lại bao gồm: * **Sự cố mạng:** Mất kết nối tạm thời, mất gói dữ liệu hoặc các vấn đề phân giải DNS. * **Dịch vụ không khả dụng:** Sự cố ngừng hoạt động ngắn của máy chủ mục tiêu, cấu hình lại bộ cân bằng tải hoặc cạn kiệt tài nguyên tạm thời. * **Giới hạn tỷ lệ:** Các API thường áp đặt giới hạn về số lượng yêu cầu trong một khung thời gian nhất định. Việc thử lại, đặc biệt là với độ trễ tăng dần, có thể giúp điều hướng các hạn chế này. * **Các vấn đề về đồng thời:** Tắc nghẽn hoặc lỗi đồng thời lạc quan có thể giải quyết được trong các lần thử tiếp theo. * **Tranh chấp tài nguyên tạm thời:** Khi một tài nguyên bị khóa tạm thời bởi một quy trình khác.

Sự cần thiết của cơ chế thử lại bắt nguồn từ sự không đáng tin cậy vốn có của các hệ thống phân tán. Không giống như các ứng dụng nguyên khối nơi các thành phần thường nằm trên cùng một máy và giao tiếp đáng tin cậy, các hệ thống phân tán liên quan đến giao tiếp mạng, nhiều dịch vụ độc lập và các phụ thuộc bên ngoài, tất cả đều có thể gây ra các điểm lỗi. Đối với một `cli_executor` cào một trang web, các yếu tố như tải máy chủ của trang web, độ trễ mạng giữa `cli_executor` và `batdongsan.com.vn`, hoặc thậm chí các chặn tạm thời của trang web mục tiêu có thể khiến các yêu cầu thất bại. Nếu không thử lại, một lỗi tạm thời duy nhất sẽ dẫn đến việc toàn bộ nhiệm vụ cào web cho URL đó thất bại hoàn toàn, dẫn đến việc thu thập dữ liệu không đầy đủ.

Các chiến lược thử lại hiệu quả bao gồm một số nguyên tắc cốt lõi: * **Số lần thử lại:** Xác định số lần thử tối đa để ngăn chặn các vòng lặp vô hạn và cạn kiệt tài nguyên cuối cùng. * **Chiến lược độ trễ:** Thực hiện một khoảng dừng giữa các lần thử lại. Đây có thể là một độ trễ cố định, nhưng phổ biến hơn, một chiến lược **backoff lũy thừa** được sử dụng, trong đó độ trễ tăng theo cấp số nhân với mỗi lần thử tiếp theo (ví dụ: 1s, 2s, 4s, 8s). Điều này ngăn chặn việc làm quá tải dịch vụ bị lỗi và cho phép nó có thời gian phục hồi (AWS Architecture Blog). * **Jitter:** Giới thiệu một thành phần ngẫu nhiên vào độ trễ (ví dụ: `random_number * exponential_backoff_delay`). Jitter giúp ngăn chặn vấn đề "thundering herd" nơi nhiều máy khách, tất cả đều thử lại với cùng một backoff lũy thừa, có thể đồng bộ hóa và truy cập dịch vụ đồng thời, làm trầm trọng thêm vấn đề ban đầu. * **Phân loại lỗi:** Chỉ thử lại đối với các lỗi thực sự tạm thời. Các lỗi vĩnh viễn (ví dụ: HTTP 400 Bad Request, 404 Not Found, 500 Internal Server Error chỉ ra một vấn đề mã cơ bản) thường không nên được thử lại, vì chúng không có khả năng tự giải quyết và sẽ lãng phí tài nguyên. * **Hết thời gian chờ:** Đặt thời gian chờ tổng thể cho toàn bộ chuỗi thử lại để đảm bảo các hoạt động không bị treo vô thời hạn.

Bằng cách kết hợp các nguyên tắc này, cơ chế thử lại cải thiện đáng kể khả năng chịu lỗi và độ mạnh mẽ tổng thể của các ứng dụng như `cli_executor`, cho phép chúng xử lý các lỗi tạm thời một cách duyên dáng và hoàn thành thành công các hoạt động dự kiến của chúng.

Xây dựng logic thử lại mạnh mẽ: Các triển khai Python phổ biến

Việc triển khai logic thử lại mạnh mẽ từ đầu có thể phức tạp, đòi hỏi phải xử lý cẩn thận các độ trễ, loại lỗi và quản lý trạng thái. May mắn thay, Python cung cấp một số thư viện trưởng thành giúp trừu tượng hóa phần lớn sự phức tạp này, cho phép các nhà phát triển tích hợp các chính sách thử lại tinh vi với ít mã nhất. Các thư viện này thường tận dụng các decorator hoặc context manager để bao bọc các hàm hoặc khối mã có thể gặp phải các lỗi tạm thời.

Một trong những thư viện được sử dụng rộng rãi nhất cho logic thử lại trong Python là **`tenacity`** (GitHub: tenacity). `tenacity` cung cấp một cách linh hoạt và mạnh mẽ để thêm hành vi thử lại bằng cách sử dụng các decorator. Nó hỗ trợ nhiều điều kiện thử lại khác nhau (ví dụ: trên các ngoại lệ cụ thể, trên giá trị trả về), các chiến lược chờ khác nhau (cố định, lũy thừa, ngẫu nhiên), các điều kiện dừng (sau một số lần thử nhất định, sau một tổng thời gian) và các callback cho việc ghi nhật ký hoặc các hành động tùy chỉnh.

Ví dụ về cách sử dụng `tenacity`:

```python
from tenacity import retry, wait_exponential, stop_after_attempt, retry_if_exception_type
import requests

@retry(wait=wait_exponential(multiplier=1, min=4, max=10), stop=stop_after_attempt(5),
       retry=retry_if_exception_type(requests.exceptions.ConnectionError))
def fetch_url_with_retry(url):
    print(f"Attempting to fetch {url}...")
    response = requests.get(url, timeout=5)
    response.raise_for_status() # Raise HTTPError for bad responses (4xx or 5xx)
    print(f"Successfully fetched {url}")
    return response.text

# Trong ngữ cảnh cli_executor, hàm này sẽ được gọi:
# content = fetch_url_with_retry("https://batdongsan.com.vn/ban-can-ho-chung-cu-tp-hcm")
```

Trong ví dụ này, `fetch_url_with_retry` sẽ cố gắng tìm nạp URL tối đa 5 lần. Nó sẽ chờ theo cấp số nhân giữa các lần thử lại, bắt đầu với tối thiểu 4 giây và tối đa 10 giây. Nó đặc biệt chỉ thử lại nếu xảy ra `requests.exceptions.ConnectionError`, cho thấy một vấn đề liên quan đến mạng. Điều này cho thấy `tenacity` cho phép kiểm soát chi tiết về thời điểm và cách thức thực hiện việc thử lại.

Một thư viện phổ biến khác là **`retrying`** (GitHub: retrying). Tương tự như `tenacity`, `retrying` sử dụng các decorator và cung cấp các tùy chọn cho các điều kiện thử lại, độ trễ và điều kiện dừng. Nó thường được chọn vì sự đơn giản và dễ sử dụng cho các mẫu thử lại phổ biến.

Ví dụ về cách sử dụng `retrying`:

```python
from retrying import retry
import requests

def is_retryable_exception(exception):
    return isinstance(exception, requests.exceptions.ConnectionError)

@retry(retry_on_exception=is_retryable_exception, wait_exponential_multiplier=1000, wait_exponential_ma
def fetch_url_with_retrying(url):
    print(f"Attempting to fetch {url}...")
    response = requests.get(url, timeout=5)
    response.raise_for_status()
    print(f"Successfully fetched {url}")
```

```
    return response.text

# content = fetch_url_with_retrying("https://batdongsan.com.vn/ban-can-ho-chung-cu-tp-hcm")
```

Ở đây, `retry_on_exception` là một hàm vị ngữ xác định xem một ngoại lệ có nên kích hoạt một lần thử lại hay không. `wait_exponential_multiplier` đặt cơ sở cho backoff lũy thừa tính bằng mili giây, và `stop_max_attempt_number` xác định số lần thử lại tối đa.

Thư viện **`backoff`** (GitHub: backoff) là một lựa chọn mạnh mẽ khác, đặc biệt được chú ý bởi API rõ ràng và hỗ trợ các chiến lược backoff khác nhau, bao gồm lũy thừa, fibonacci và các chiến lược tùy chỉnh. Nó cũng hỗ trợ jitter và có thể được cấu hình để thử lại trên các ngoại lệ hoặc mã trạng thái cụ thể.

Các thư viện này giảm đáng kể mã lặp đi lặp lại cần thiết để triển khai việc thử lại, cho phép các nhà phát triển tập trung vào logic cốt lõi của ứng dụng trong khi hưởng lợi từ các chiến lược thử lại đã được thử nghiệm. Lỗi `NameError` trong ngữ cảnh `cli_executor` cho thấy rằng một thư viện như vậy, hoặc một lớp/hàm `Retry` tùy chỉnh, đã được dự định nhưng không được nhập hoặc tham chiếu chính xác, do đó ngăn chặn ứng dụng tận dụng các mẫu kiên cường quan trọng này.

Các chiến lược nâng cao cho hoạt động kiên cường: Vượt xa các lần thử lại cơ bản

Mặc dù các cơ chế thử lại cơ bản với backoff lũy thừa rất hiệu quả đối với các lỗi tạm thời, nhưng các chiến lược nâng cao là cần thiết để xây dựng các hệ thống thực sự kiên cường có thể chịu được các chế độ lỗi phức tạp hơn và ngăn chặn các lỗi theo tầng. Các chiến lược này thường bổ sung, chứ không thay thế, logic thử lại cơ bản.

Một chiến lược nâng cao quan trọng là **mẫu Ngắt mạch (Circuit Breaker)** (Martin Fowler). Không giống như các lần thử lại đơn giản chỉ tiếp tục cố gắng thực hiện một hoạt động cho đến khi đạt đến giới hạn, một ngắt mạch giám sát tỷ lệ lỗi của các hoạt động đối với một dịch vụ cụ thể. Nếu tỷ lệ lỗi vượt quá ngưỡng được xác định trước trong một khoảng thời gian nhất định, mạch sẽ "đóng" lại. Khi đóng, tất cả các cuộc gọi tiếp theo đến dịch vụ đó sẽ thất bại ngay lập tức mà không cố gắng thực hiện hoạt động. Điều này ngăn chặn ứng dụng liên tục gây áp lực lên một dịch vụ đang gặp lỗi, điều này có thể làm trầm trọng thêm vấn đề hoặc lãng phí tài nguyên. Sau một thời gian chờ có thể cấu hình (trạng thái "bán mở"), mạch cho phép một số lượng yêu cầu thử nghiệm hạn chế đi qua. Nếu những yêu cầu này thành công, mạch sẽ đóng lại và hoạt động bình thường sẽ tiếp tục. Nếu chúng thất bại, nó sẽ mở lại. Mẫu này đặc biệt hữu ích để bảo vệ `cli_executor` khỏi việc liên tục truy cập một máy chủ `batdongsan.com.vn` không phản hồi hoặc bị quá tải, cho phép máy chủ phục hồi mà không bị gánh nặng thêm.

Một yếu tố quan trọng khác cần xem xét là **tính bất biến (idempotency)**. Một hoạt động là bất biến nếu việc thực hiện nó nhiều lần tạo ra cùng một kết quả như thực hiện nó một lần. Khi triển khai các lần thử lại, đặc biệt đối với các hoạt động thay đổi trạng thái (ví dụ: tạo yêu cầu POST để tạo tài nguyên), đảm bảo tính bất biến là rất quan trọng. Nếu một yêu cầu mạng thất bại sau khi máy chủ đã xử lý nó nhưng trước khi máy khách nhận được xác nhận, việc thử lại một hoạt động không bất biến có thể dẫn đến tạo tài nguyên trùng lặp hoặc các tác dụng phụ không mong muốn. Đối với việc cào web, việc tìm nạp một URL (yêu cầu GET) thường là bất biến. Tuy nhiên, nếu `cli_executor` cũng đang tương tác với một API để lưu dữ liệu, việc đảm bảo hoạt động lưu là bất biến (ví dụ: bằng cách bao gồm một ID giao dịch duy nhất) sẽ rất quan trọng (Wikipedia: Idempotence).

Thử lại thích ứng (Adaptive Retries) thể hiện một cách tiếp cận tinh vi hơn, trong đó các tham số thử lại (độ trễ, số lần thử tối đa) được điều chỉnh linh hoạt dựa trên phản hồi theo thời gian thực từ hệ thống hoặc dịch vụ mục tiêu. Chẳng hạn, nếu một dịch vụ phản hồi với mã trạng thái HTTP 429 Too Many Requests, nó có thể bao gồm tiêu đề `Retry-After` cho biết máy khách nên chờ bao lâu trước khi thử lại. Một cơ chế thử lại thích ứng sẽ phân tích tiêu đề này và điều chỉnh độ trễ tương ứng, thay vì dựa vào một backoff cố định hoặc lũy thừa có thể quá hung hãn hoặc quá thận trọng (Google Cloud Blog). Điều này có thể cải thiện đáng kể hiệu quả và giảm tải không cần thiết cho dịch vụ mục tiêu.

Cuối cùng, các **mẫu vách ngăn (bulkhead patterns)** có thể được sử dụng kết hợp với các lần thử lại. Mẫu này cách ly các thành phần hoặc tài nguyên thành các nhóm riêng biệt, ngăn chặn một lỗi ở một khu vực tiêu thụ tất cả tài nguyên và ảnh hưởng đến các phần khác của hệ thống. Đối với một `cli_executor`

xử lý nhiều URL đồng thời, một vách ngăn có thể đảm bảo rằng các lỗi hoặc các lần thử lại quá mức trên một URL không làm cạn kiệt các kết nối mạng hoặc các luồng xử lý có sẵn cho các URL khác.

Việc tích hợp các chiến lược nâng cao này sẽ biến `cli_executor` từ một công cụ chỉ đơn thuần là chức năng thành một hệ thống có khả năng phục hồi cao, có thể điều hướng tính chất không thể đoán trước của các tương tác mạng và dịch vụ, đảm bảo việc thu thập dữ liệu nhất quán và thành công hơn từ các nguồn như `batdongsan.com.vn`. Việc thiếu cơ chế `Retry` được định nghĩa, như được chỉ ra bởi `NameError`, làm nổi bật một lỗ hổng cơ bản trong khả năng của hệ thống hiện tại để tận dụng ngay cả các mẫu kiên cường cơ bản, chứ chưa nói đến những mẫu nâng cao này.

Các biện pháp chủ động và thực tiễn tốt nhất để ngăn chặn `NameError` và tích hợp các cơ chế thử lại

Ngăn chặn `NameError` và tích hợp hiệu quả các cơ chế thử lại đòi hỏi sự kết hợp giữa các thực hành mã hóa kỷ luật, quản lý phụ thuộc mạnh mẽ và kiểm thử kỹ lưỡng. Đối với ứng dụng `cli_executor`, các biện pháp này là rất quan trọng để đảm bảo sự ổn định hoạt động và tính toàn vẹn của dữ liệu.

Để chủ động ngăn chặn các ngoại lệ `NameError` như `name 'Retry' is not defined`: * **Nhập rõ ràng:** Luôn sử dụng các câu lệnh `import` rõ ràng cho tất cả các mô-đun, lớp và hàm không được tích hợp sẵn. Tránh nhập ngầm định hoặc dựa vào phạm vi toàn cầu trừ khi thực sự cần thiết và có lý do chính đáng. Đối với `Retry`, điều này có nghĩa là đảm bảo `from some_retry_library import retry` hoặc `import some_retry_library as retry_lib` và sau đó sử dụng `retry_lib.retry` (PEP 8 – Hướng dẫn phong cách cho mã Python). * **Quản lý phụ thuộc:** Sử dụng `pip` với `requirements.txt` hoặc `pyproject.toml` để xác định và quản lý chính xác các phụ thuộc của dự án. Điều này đảm bảo rằng tất cả các thư viện bên thứ ba cần thiết (như `tenacity` hoặc `retrying`) được cài đặt với các phiên bản chính xác trên tất cả các môi trường (phát triển, kiểm thử, sản xuất). Nên sử dụng môi trường ảo (`venv` hoặc `conda`) để cách ly các phụ thuộc của dự án và ngăn ngừa xung đột (Python Packaging Authority). * **Đánh giá mã và phân tích tĩnh:** Thực hiện các quy trình đánh giá mã nghiêm ngặt, nơi đồng nghiệp có thể xác định các lệnh nhập bị thiếu, lỗi đánh máy hoặc các vấn đề về phạm vi trước khi triển khai. Các công cụ phân tích tĩnh như `Pylint`, `Flake8` hoặc `MyPy` có thể tự động phát hiện các `NameError` tiềm ẩn và các lỗi mã hóa phổ biến khác bằng cách phân tích mã mà không cần thực thi nó (Tài liệu Pylint). * **Kiểm thử đơn vị và tích hợp:** Viết các kiểm thử đơn vị toàn diện cho các hàm dựa vào các thành phần bên ngoài hoặc các lệnh nhập cụ thể. Kiểm thử tích hợp nên xác minh rằng `cli_executor` có thể tương tác thành công với các phụ thuộc của nó và xử lý các kịch bản lỗi dự kiến, bao gồm cả những kịch bản sẽ kích hoạt việc thử lại.

Để tích hợp mạnh mẽ các cơ chế thử lại: * **Quản lý cấu hình:** Các chính sách thử lại (ví dụ: số lần thử tối đa, độ trễ cơ bản, loại lỗi cần thử lại) nên có thể cấu hình được, lý tưởng là được ngoại hóa khỏi mã. Điều này cho phép người vận hành điều chỉnh các tham số kiên cường mà không yêu cầu thay đổi mã và triển khai lại, điều này đặc biệt hữu ích để thích ứng với các điều kiện thay đổi của các dịch vụ mục tiêu như `batdongsan.com.vn` hoặc môi trường mạng. Cấu hình có thể được quản lý thông qua các biến môi trường, tệp cấu hình (ví dụ: YAML, JSON) hoặc một dịch vụ cấu hình chuyên dụng. * **Ghi nhật ký và giám sát:** Triển khai ghi nhật ký chi tiết xung quanh các lần thử lại. Điều này bao gồm ghi nhật ký khi một hoạt động được thử lại, lý do thử lại (ví dụ: ngoại lệ cụ thể), số lần thử hiện tại và độ trễ trước lần thử tiếp theo. Các công cụ giám sát nên theo dõi các số liệu như tổng số lần thử lại, tỷ lệ thành công sau các lần thử lại và số lần thử lại trung bình cho mỗi hoạt động thành công. Dữ liệu này là vô giá để hiểu độ tin cậy của các dịch vụ bên ngoài và điều chỉnh các chính sách thử lại (OpenTelemetry). * **Mức độ chi tiết xử lý lỗi:** Phân biệt giữa các lỗi tạm thời và vĩnh viễn. Chỉ các lỗi tạm thời mới nên kích hoạt việc thử lại. Các lỗi vĩnh viễn (ví dụ: lỗi xác thực, đầu vào không hợp lệ) nên thất bại ngay lập tức và được leo thang để nhà phát triển can thiệp. Điều này ngăn chặn việc lãng phí tài nguyên cho các hoạt động được đảm bảo sẽ thất bại. * **Xác minh tính bất biến:** Đối với bất kỳ hoạt động nào thay đổi trạng thái và phải chịu các lần thử lại, hãy đảm bảo nó là bất biến. Nếu không phải là bất biến, hãy thiết kế logic thử lại hoặc dịch vụ mục tiêu để xử lý các yêu cầu trùng lặp một cách duyên dáng. * **Giảm cấp duyên dáng:** Trong những trường hợp cực đoan khi các lần thử lại liên tục thất bại, hãy xem xét triển khai các chiến lược giảm cấp duyên dáng. Ví dụ, nếu `batdongsan.com.vn` liên tục không khả dụng, `cli_executor` có thể tạm thời chuyển sang phiên bản dữ liệu được lưu trong bộ nhớ cache, thông báo cho quản trị viên hoặc tạm dừng xử lý cho nguồn cụ thể đó thay vì liên tục thử lại vô thời hạn.

Bằng cách tuân thủ các thực tiễn tốt nhất này, các nhà phát triển có thể giảm đáng kể sự xuất hiện của các lỗi thời gian chạy như `NameError` và xây dựng các ứng dụng `cli_executor` không chỉ có chức năng mà còn có khả năng phục hồi cao và thích ứng với bản chất động của môi trường điện toán phân tán.

Ngữ cảnh hoạt động: `cli_executor` và xử lý dữ liệu web

Kiến trúc `cli_executor` và các mô hình tương tác web

Thành phần `cli_executor`, như được chỉ ra bởi thông báo lỗi, hoạt động như một môi trường thực thi được điều khiển bằng giao diện dòng lệnh (CLI), được thiết kế để điều phối và quản lý các tác vụ tự động, đặc biệt là những tác vụ liên quan đến xử lý dữ liệu web. Kiến trúc của nó thường bao gồm một số mô-đun chính tạo điều kiện thuận lợi cho việc tương tác mạnh mẽ với các tài nguyên web. Ở cốt lõi, một mô-đun xử lý yêu cầu chịu trách nhiệm khởi tạo các yêu cầu HTTP/HTTPS đến các URL mục tiêu, chẳng hạn như `https://batdongsan.com.vn/ban-can-ho-chung-cu-tp-hcm`. Mô-đun này thường tích hợp các tính năng như xoay vòng user-agent, quản lý proxy và xử lý cookie để mô phỏng hành vi trình duyệt hợp pháp và vượt qua các biện pháp chống bot cơ bản (ScrapingBee).

Sau khi thực thi yêu cầu thành công, một mô-đun phân tích cú pháp trích xuất dữ liệu liên quan từ các phản hồi HTML hoặc JSON nhận được. Mô-đun này có thể tận dụng các thư viện như Beautiful Soup hoặc LXML để phân tích cú pháp HTML, hoặc `json` cho các phản hồi API, để điều hướng Mô hình đối tượng tài liệu (DOM) và xác định các điểm dữ liệu cụ thể (ví dụ: danh sách tài sản, giá cả, mô tả). Đối với các ứng dụng web hiện đại phụ thuộc nhiều vào JavaScript để hiển thị nội dung, `cli_executor` có thể tích hợp với các trình duyệt không giao diện (headless browsers) (ví dụ: Puppeteer, Playwright, Selenium) để thực thi các tập lệnh phía máy khách trước khi phân tích cú pháp, đảm bảo truy cập vào nội dung trang được hiển thị đầy đủ (Mozilla Developer Network). Việc tích hợp này làm tăng độ phức tạp nhưng thường là cần thiết đối với các trang web như `batdongsan.com.vn` có khả năng sử dụng tải nội dung động.

Quản lý phiên là một khía cạnh quan trọng khác, trong đó `cli_executor` duy trì trạng thái trên nhiều yêu cầu trong một tác vụ xử lý duy nhất. Điều này liên quan đến việc quản lý cookie, mã thông báo xác thực và có khả năng là các kết nối liên tục để tối ưu hóa hiệu suất và tuân thủ các chính sách của trang web. ID phiên `a5ce5514-1874-42b8-9393-ac6009c1d57d` cho thấy rằng mỗi lần chạy hoặc tác vụ xử lý được `cli_executor` khởi tạo đều được xác định duy nhất, cho phép các ngữ cảnh thực thi riêng biệt và tạo điều kiện thuận lợi cho việc gỡ lỗi và ghi nhật ký. Thiết kế tổng thể hướng đến hiệu quả và khả năng phục hồi, cho phép thu thập tự động khối lượng lớn dữ liệu có cấu trúc từ các nguồn web đa dạng, thường hoạt động theo cách phân tán hoặc song song để đáp ứng các yêu cầu thông lượng. Mô hình hoạt động thường là không đồng bộ, cho phép nhiều yêu cầu web và hoạt động phân tích cú pháp diễn ra đồng thời mà không chặn toàn bộ luồng thực thi (Real Python).

Quy trình làm việc và thách thức trong việc thu thập dữ liệu web

Quy trình thu thập dữ liệu từ một trang web mục tiêu như `https://batdongsan.com.vn/ban-can-ho-chung-cu-tp-hcm` thông qua `cli_executor` tuân theo một quy trình làm việc có cấu trúc, nhưng nó chứa đầy những thách thức vốn có. Ban đầu, `cli_executor` gửi một yêu cầu HTTP GET đến URL được chỉ định. Sau khi nhận được phản hồi của máy chủ, thường bao gồm HTML, CSS và JavaScript, hệ thống tiến hành hiển thị nội dung trang. Đối với nội dung tĩnh, điều này liên quan đến phân tích cú pháp HTML trực tiếp. Tuy nhiên, đối với các cổng thông tin bất động sản động, các phần đáng kể của dữ liệu, chẳng hạn như danh sách tài sản, bộ lọc hoặc điều khiển phân trang, thường được tải không đồng bộ qua JavaScript (các yêu cầu AJAX) sau khi tải trang ban đầu. Điều này đòi hỏi phải sử dụng tự động hóa trình duyệt không giao diện trong `cli_executor` để hiển thị đầy đủ trang và tương tác với các yếu tố của nó, mô phỏng trải nghiệm duyệt web của người dùng (Google Developers).

Khi trang được hiển thị đầy đủ, `cli_executor` sử dụng logic phân tích cú pháp của nó để xác định và trích xuất các điểm dữ liệu cụ thể. Điều này thường liên quan đến XPath hoặc bộ chọn CSS để xác định các yếu tố như tiêu đề tài sản, giá cả, vị trí và thông tin liên hệ. Các quy trình chuẩn hóa và làm sạch dữ liệu sau đó được áp dụng để đảm bảo tính nhất quán và khả năng sử dụng của thông tin được trích xuất. Ví dụ, các ký hiệu tiền tệ có thể được loại bỏ và các định dạng ngày được chuẩn hóa. Phân trang là một thách thức

phổ biến, đòi hỏi `cli_executor` phải xác định và theo dõi các liên kết "trang tiếp theo" hoặc thao tác các tham số URL để lặp lại tất cả các danh sách có sẵn. Quy trình lặp lại này đòi hỏi quản lý trạng thái cẩn thận để tránh xử lý lại các trang hoặc bỏ lỡ dữ liệu.

Ngoài phân tích cú pháp kỹ thuật, việc thu thập dữ liệu web phải đối mặt với những trở ngại hoạt động đáng kể. Các trang web thường triển khai các biện pháp chống cào web, bao gồm giới hạn tỷ lệ (chặn các yêu cầu từ một địa chỉ IP duy nhất nếu chúng vượt quá một tần suất nhất định), chặn IP, CAPTCHA và các thuật toán phát hiện bot tinh vi. Ví dụ, `batdongsan.com.vn` có thể phát hiện các mẫu yêu cầu bất thường hoặc user-agent không phải trình duyệt, dẫn đến các chặn tạm thời hoặc vĩnh viễn. Việc xử lý những điều này đòi hỏi xoay vòng IP động, lên lịch yêu cầu phân tán và có thể là các dịch vụ giải CAPTCHA, tất cả đều thêm các lớp phức tạp vào thiết kế hoạt động của `cli_executor` (Bright Data). Hơn nữa, các thay đổi cấu trúc trang web (cập nhật bố cục, sửa đổi tên lớp) có thể phá vỡ logic phân tích cú pháp hiện có, đòi hỏi phải bảo trì và thích ứng liên tục các tập lệnh cào web của `cli_executor`. Khối lượng dữ liệu và tần suất cập nhật cũng đặt ra những thách thức, đòi hỏi các giải pháp lưu trữ hiệu quả và các cơ chế lập lịch mạnh mẽ để đảm bảo thu thập dữ liệu kịp thời và đầy đủ.

Xử lý lỗi và tích hợp cơ chế `Retry`

Xử lý lỗi mạnh mẽ là tối quan trọng trong ngữ cảnh hoạt động của `cli_executor` để xử lý dữ liệu web, do tính không đáng tin cậy vốn có của các tài nguyên web bên ngoài. Một thành phần cốt lõi của khả năng phục hồi này là việc triển khai các cơ chế thử lại. Khi `cli_executor` gặp phải các lỗi tạm thời trong quá trình thu thập dữ liệu web – chẳng hạn như hết thời gian chờ mạng, máy chủ tạm thời không khả dụng (ví dụ: HTTP 503 Service Unavailable) hoặc phản hồi giới hạn tỷ lệ (ví dụ: HTTP 429 Too Many Requests) – một cơ chế thử lại được thiết kế tốt sẽ cố gắng thực thi lại hoạt động bị lỗi sau một khoảng thời gian chờ nhất định. Điều này ngăn chặn việc tác vụ thất bại ngay lập tức và tăng khả năng truy xuất dữ liệu thành công mà không cần can thiệp thủ công (Amazon Web Services).

Thông thường, một cơ chế `Retry` trong `cli_executor` sẽ được triển khai dưới dạng một hàm, lớp hoặc decorator bao bọc logic yêu cầu web. Nó sẽ định nghĩa các tham số như số lần thử lại tối đa, độ trễ giữa các lần thử và các loại ngoại lệ hoặc mã trạng thái HTTP cụ thể nên kích hoạt việc thử lại. Các chiến lược phổ biến bao gồm backoff lũy thừa, trong đó độ trễ giữa các lần thử lại tăng theo cấp số nhân (ví dụ: 1 giây, sau đó 2 giây, sau đó 4 giây), thường có thêm jitter (các biến thể ngẫu nhiên nhỏ) để ngăn chặn các lần thử lại đồng bộ làm quá tải máy chủ mục tiêu (Google Cloud). Một mẫu ngắt mạch cũng có thể được tích hợp, tạm thời dừng các nỗ lực đến một dịch vụ bị lỗi sau một số lần lỗi liên tiếp, cho phép dịch vụ phục hồi trước khi các yêu cầu tiếp theo được thực hiện (Microsoft Learn).

Kỳ vọng trong một hệ thống như `cli_executor` là một đối tượng hoặc hàm `Retry` sẽ có sẵn và được nhập hoặc định nghĩa đúng cách trong phạm vi mà các yêu cầu web được thực hiện. Việc thiếu nó, như được chỉ ra bởi lỗi "name 'Retry' is not defined", cho thấy một lỗi nghiêm trọng trong khả năng của hệ thống để xử lý các lỗi tạm thời một cách duyên dáng. Nếu không có cơ chế `Retry` được định nghĩa, bất kỳ vấn đề tạm thời nào trong quá trình xử lý `https://batdongsan.com.vn/ban-can-ho-chung-cu-tp-hcm` sẽ dẫn đến một ngoại lệ ngay lập tức và không được xử lý, khiến toàn bộ phiên (`a5ce5514-1874-42b8-9393-ac6009c1d57d`) kết thúc sớm. Điều này làm giảm đáng kể tính mạnh mẽ và độ tin cậy của quy trình xử lý dữ liệu web, khiến nó rất dễ bị ảnh hưởng bởi các vấn đề mạng gián đoạn hoặc biến động phía máy chủ. Việc tích hợp đúng logic `Retry` không chỉ là một thực tiễn tốt nhất mà còn là một yêu cầu cơ bản cho các hoạt động cào web ổn định và hiệu quả.

Phân tích nguyên nhân gốc rễ: "name 'Retry' is not defined" trong `cli_executor`

Thông báo lỗi "name 'Retry' is not defined" trong phiên `cli_executor` `a5ce5514-1874-42b8-9393-ac6009c1d57d` trong quá trình xử lý `https://batdongsan.com.vn/ban-can-ho-chung-cu-tp-hcm` chỉ ra một vấn đề lập trình hoặc triển khai cơ bản hơn là một vấn đề tương tác web thời gian chạy. Lỗi này thường xảy ra trong môi trường Python khi một biến, hàm hoặc lớp có tên `Retry` được tham chiếu trước khi nó được định nghĩa hoặc nhập đúng cách vào phạm vi hiện tại (Tài liệu Python). Một số kịch bản có thể dẫn đến lỗi cụ thể này trong ngữ cảnh các hoạt động xử lý dữ liệu web của `cli_executor`.

Một nguyên nhân chính là câu lệnh nhập bị thiếu hoặc không chính xác. Nếu đối tượng hoặc hàm `Retry` là một phần của thư viện bên ngoài (ví dụ: `requests-retry`, `tenacity` hoặc một mô-đun tiện ích nội bộ tùy chỉnh), tập lệnh được thực thi bởi `cli_executor` có thể thiếu dòng `import Retry` hoặc `from some_module import Retry` cần thiết. Điều này có thể xảy ra nếu tập lệnh gần đây đã được sửa đổi, hoặc nếu một phụ thuộc được thêm vào mà không cập nhật tất cả các đường dẫn mã liên quan. Chẳng hạn, một nhà phát triển có thể đã tái cấu trúc một tiện ích chung thành một mô-đun riêng biệt, nhưng một tập lệnh tác vụ `cli_executor` cụ thể không được cập nhật để phản ánh thay đổi này. Một nguyên nhân tiềm năng khác là vấn đề về phạm vi. Ngay cả khi `Retry` được định nghĩa ở nơi khác trong cơ sở mã, nó có thể không truy cập được trong hàm hoặc phương thức lớp cụ thể nơi xảy ra lỗi. Điều này có thể là do `Retry` được định nghĩa cục bộ bên trong một hàm khác, hoặc nếu nó là một thành viên lớp chưa được khởi tạo đúng cách hoặc tham chiếu với `self.Retry` (hoặc `ClassName.Retry`) khi thích hợp. Trong các thiết lập `cli_executor` phức tạp, đặc biệt với các mô-đun hoặc plugin được tải động, ngữ cảnh thực thi cho một tác vụ xử lý web cụ thể có thể không kế thừa các định nghĩa cấp toàn cục hoặc cấp mô-đun mong đợi.

Sự khác biệt về cấu hình môi trường cũng là một nguyên nhân gốc rễ hợp lý. `cli_executor` có thể đang chạy trong một môi trường (ví dụ: một vùng chứa, một máy ảo hoặc một môi trường ảo Python cụ thể) nơi thư viện cần thiết chứa chức năng `Retry` không được cài đặt hoặc là một phiên bản lỗi thời thiếu định nghĩa mong đợi. Ví dụ, nếu `cli_executor` dựa vào tệp `requirements.txt`, một mục bị thiếu hoặc xung đột phiên bản có thể ngăn cơ chế `Retry` khả dụng. Điều này đặc biệt liên quan trong các quy trình CI/CD nơi môi trường triển khai có thể hơi khác so với môi trường phát triển (The Twelve-Factor App).

Cuối cùng, lỗi có thể bắt nguồn từ một lỗi đánh máy đơn giản trong mã nơi `Retry` được gọi, hoặc một lỗi logic nơi một nhánh điều kiện định nghĩa `Retry` không được thực thi trong một số trường hợp nhất định. Với bản chất của các tác vụ tự động, một lỗi như vậy cho thấy một sự giám sát nghiêm trọng trong quản lý phụ thuộc của tập lệnh hoặc thiết lập môi trường thực thi của nó, ảnh hưởng trực tiếp đến khả năng của `cli_executor` để xử lý các vấn đề mạng và web tạm thời trong quá trình thu thập dữ liệu từ `batdongsan.com.vn`.

Đánh giá tác động và chiến lược giảm thiểu lỗi xử lý dữ liệu web

Lỗi "name 'Retry' is not defined" trong `cli_executor` có tác động hoạt động đáng kể đến xử lý dữ liệu web, đặc biệt đối với các tác vụ liên quan đến các nguồn web động và có khả năng không ổn định như `https://batdongsan.com.vn/ban-can-ho-chung-cu-tp-hcm`. Hậu quả tức thì nhất là việc kết thúc sớm phiên xử lý (`a5ce5514-1874-42b8-9393-ac6009c1d57d`). Điều này dẫn đến việc thu thập dữ liệu không đầy đủ, vì bất kỳ dữ liệu nào đáng lẽ đã được xử lý sau điểm lỗi đều bị mất. Đối với một cổng thông tin bất động sản, điều này có thể có nghĩa là thiếu các danh sách tài sản quan trọng, cập nhật giá hoặc các xu hướng thị trường mới, ảnh hưởng trực tiếp đến tính kịp thời và đầy đủ của tập dữ liệu được thu thập.

Ngoài việc mất dữ liệu, lỗi này dẫn đến lãng phí tài nguyên tính toán. `cli_executor` sẽ tiêu thụ chu kỳ CPU, bộ nhớ và băng thông mạng cho đến khi xảy ra lỗi mà không tạo ra bất kỳ đầu ra hoàn chỉnh, có thể sử dụng nào. Nếu `cli_executor` hoạt động trong một hệ thống theo lịch trình hoặc phân tán, các lỗi lặp đi lặp lại do lỗi này có thể dẫn đến tồn đọng các tác vụ, tăng chi phí hoạt động và khả năng suy giảm dịch vụ cho các hệ thống hạ nguồn dựa vào dữ liệu đã thu thập. Hơn nữa, việc thiếu cơ chế thử lại có nghĩa là ngay cả những trục trặc mạng nhỏ, tạm thời hoặc các vấn đề phía máy chủ tạm thời trên `batdongsan.com.vn` cũng sẽ gây ra lỗi nghiêm trọng, khiến toàn bộ đường ống dữ liệu trở nên cực kỳ mong manh và không đáng tin cậy (O'Reilly Media).

Để giảm thiểu các lỗi như vậy và tăng cường tính mạnh mẽ của quá trình xử lý dữ liệu web của `cli_executor`, một số chiến lược có thể được áp dụng. Đầu tiên, **quản lý phụ thuộc nghiêm ngặt** là rất quan trọng. Tất cả các thư viện và mô-đun cần thiết cho các tập lệnh `cli_executor`, bao gồm cả những thư viện cung cấp chức năng thử lại, phải được liệt kê rõ ràng trong tệp `requirements.txt` hoặc tệp phụ thuộc tương tự. Tệp này phải được sử dụng nhất quán trên tất cả các môi trường phát triển, kiểm thử và sản xuất để đảm bảo rằng môi trường thực thi luôn được cấp phép chính xác. Các kiểm tra tự động trong quá trình CI/CD có thể xác minh rằng tất cả các phụ thuộc đã được cài đặt và được phiên bản chính xác (Atlassian).

Thứ hai, nên triển khai **kiểm thử đơn vị và tích hợp toàn diện**. Kiểm thử đơn vị có thể xác minh rằng các hàm và lớp riêng lẻ, bao gồm cả cơ chế `Retry` đó, được định nghĩa và nhập đúng cách. Kiểm thử tích

hợp có thể mô phỏng quy trình làm việc đầy đủ của `cli_executor`, bao gồm các yêu cầu web và xử lý lỗi, để bắt các lỗi định nghĩa như vậy trước khi triển khai. Giả lập các dịch vụ bên ngoài có thể giúp kiểm thử logic thử lại mà không cần tương tác web thực tế. Thứ ba, **các công cụ phân tích mã tĩnh** (ví dụ: Pylint, Flake8) có thể được cấu hình để phát hiện các tên không xác định hoặc các lệnh nhập bị thiếu trong giai đoạn phát triển, cung cấp phản hồi ngay lập tức cho các nhà phát triển.

Cuối cùng, **ghi nhật ký và giám sát nâng cao** là rất cần thiết. `cli_executor` nên ghi nhật ký thông tin chi tiết về quá trình thực thi của nó, bao gồm các lệnh nhập mô-đun và các biến môi trường, đặc biệt là trong quá trình khởi động. Các hệ thống giám sát phải được thiết lập để cảnh báo ngay lập tức cho người vận hành khi một phiên thất bại với một ngoại lệ không được xử lý như "name 'Retry' is not defined," cho phép điều tra và giải quyết kịp thời. Việc triển khai chung các chiến lược này có thể cải thiện đáng kể khả năng phục hồi và độ tin cậy của các hoạt động `cli_executor` trong môi trường xử lý dữ liệu web động (Datadog).

Các nguyên nhân tiềm ẩn của lỗi `NameError`

Lệnh nhập mô-đun chưa được giải quyết hoặc định nghĩa bị thiếu

Nguyên nhân chính gây ra lỗi `NameError: name 'Retry' is not defined` trong môi trường thực thi Python, đặc biệt là một môi trường do `cli_executor` quản lý, bắt nguồn từ việc đối tượng hoặc lớp `Retry` không được nhập đúng cách vào không gian tên hiện tại hoặc không được định nghĩa trong phạm vi của tập lệnh. Hệ thống mô-đun của Python dựa vào các câu lệnh `import` rõ ràng để làm cho các tên từ các mô-đun khác có sẵn. Nếu `Retry` là một phần của thư viện bên ngoài (ví dụ: `urllib3`, `tenacity` hoặc một mô-đun tiện ích tùy chỉnh), và câu lệnh `import` cần thiết bị thiếu, không chính xác hoặc được đặt trong một đường dẫn mã không được thực thi trước khi `Retry` được tham chiếu, một `NameError` chắc chắn sẽ xảy ra (Python Software Foundation, n.d.-a).

Hãy xem xét một kịch bản trong đó `Retry` được dự định nhập từ một thư viện như `urllib3` cho logic thử lại HTTP. Lệnh nhập đúng thường sẽ là `from urllib3.util.retry import Retry` hoặc `import urllib3.util.retry as retry` theo sau là `retry.Retry`. Nếu nhà phát triển bỏ qua dòng này, hoặc nếu `cli_executor` chạy một tập lệnh mà lệnh nhập này bị bỏ qua có điều kiện, tên `Retry` sẽ không tồn tại trong phạm vi toàn cục hoặc cục bộ khi nó được gọi. Tương tự, nếu `Retry` là một lớp hoặc hàm tùy chỉnh được định nghĩa trong dự án, nhưng tệp định nghĩa của nó không được nhập, hoặc bản thân định nghĩa bị chú thích hoặc bị xóa, trình thông dịch sẽ không thể định vị tên. Vấn đề này thường trở nên phức tạp hơn trong các dự án lớn hơn, nơi các mô-đun có các phụ thuộc phức tạp, và một thay đổi trong một tệp có thể vô tình phá vỡ chuỗi nhập trong một tệp khác. Chẳng hạn, nếu `cli_executor` thực thi một tập lệnh chính nhập một mô-đun tiện ích, và mô-đun tiện ích đó *phải* nhập `Retry` nhưng không làm như vậy, `NameError` sẽ lan truyền đến điểm sử dụng trong tập lệnh chính hoặc bất kỳ mô-đun nào tiếp theo (Real Python, n.d.). Các nhà phát triển thường bỏ qua những chi tiết này trong quá trình tái cấu trúc hoặc khi tích hợp các thành phần mới, dẫn đến các lỗi thời gian chạy không được các công cụ phân tích tĩnh phát hiện nếu đường dẫn nhập động hoặc có điều kiện.

Sự khác biệt về cấu hình môi trường và phụ thuộc

Một yếu tố quan trọng khác góp phần gây ra lỗi `NameError` như `name 'Retry' is not defined` là sự không khớp hoặc cấu hình sai trong môi trường thực thi Python, đặc biệt liên quan đến các phụ thuộc đã cài đặt. `cli_executor` hoạt động trong một môi trường Python cụ thể, có thể là cài đặt toàn hệ thống, môi trường ảo hoặc vùng chứa Docker. Nếu thư viện cung cấp đối tượng `Retry` (ví dụ: `urllib3`, `tenacity` hoặc một gói cụ thể của dự án) không được cài đặt trong môi trường đang hoạt động, hoặc nếu cài đặt của nó bị hỏng, câu lệnh nhập cho `Retry` sẽ thất bại âm thầm hoặc gây ra `ImportError`, nếu không được xử lý, có thể dẫn đến `NameError` sau đó khi mã cố gắng sử dụng tên chưa được nhập (Python Software Foundation, n.d.-b).

Ví dụ, nếu dự án dựa vào `urllib3` phiên bản 1.26.x bao gồm `urllib3.util.retry.Retry`, nhưng `cli_executor` đang chạy trong một môi trường mà `urllib3` chưa được cài đặt hoặc có một phiên bản cũ hơn (ví dụ: 1.20.x) không hiển thị `Retry` trong đường dẫn mong đợi, câu lệnh `import` có thể thất bại. Các công cụ như `pip freeze` hoặc `conda list` có thể tiết lộ các gói đã cài đặt và phiên bản của chúng, cho

phép so sánh với `requirements.txt` hoặc `pyproject.toml` của dự án (PyPA, n.d.-a). Hơn nữa, các vấn đề với môi trường ảo, chẳng hạn như không kích hoạt đúng môi trường trước khi chạy lệnh `cli_executor`, có thể dẫn đến việc sử dụng môi trường Python toàn cầu của hệ thống, có thể thiếu các phụ thuộc cụ thể của dự án cần thiết. Biến môi trường `PYTHONPATH` cũng đóng một vai trò quan trọng; nếu nó được cấu hình không chính xác, Python có thể không thể định vị các mô-đun hoặc gói tùy chỉnh định nghĩa `Retry` trong cấu trúc của dự án. Trong các triển khai vùng chứa (ví dụ: Docker), một hình ảnh được xây dựng không chính xác hoặc một bước `RUN pip install -r requirements.txt` bị thiếu có thể dẫn đến việc các thư viện cần thiết không có sẵn trong thời gian chạy, dẫn đến `NameError` chính xác này khi ứng dụng cố gắng nhập và sử dụng `Retry` (Docker, n.d.).

Lỗi đánh máy hoặc không khớp chữ hoa chữ thường trong cơ sở mã

Đơn giản nhưng phổ biến, lỗi đánh máy và không khớp chữ hoa chữ thường trong cơ sở mã là nguyên nhân thường xuyên gây ra các ngoại lệ `NameError`, bao gồm cả lỗi cụ thể `name 'Retry' is not defined`. Python là một ngôn ngữ phân biệt chữ hoa chữ thường, có nghĩa là `Retry`, `retry` và `RETRY` được coi là các định danh hoàn toàn khác nhau (Python Software Foundation, n.d.-c). Nếu một nhà phát triển có ý định sử dụng một lớp hoặc hàm có tên `Retry` (ví dụ: `from tenacity import Retry`), nhưng vô tình gõ `retry` hoặc `Retray` khi cố gắng khởi tạo hoặc tham chiếu đến nó, trình thông dịch Python sẽ tìm kiếm `retry` hoặc `Retray` trong phạm vi hiện tại. Vì không có tên nào như vậy được định nghĩa hoặc nhập, một `NameError` sẽ được đưa ra.

Vấn đề này đặc biệt phổ biến khi xử lý các thư viện bên ngoài mà quy ước đặt tên chính xác có thể không rõ ràng ngay lập tức hoặc khi các nhà phát triển đã quen với các kiểu chữ hoa chữ thường khác từ các ngôn ngữ lập trình khác. Chẳng hạn, một số thư viện có thể sử dụng `retry_policy` trong khi những thư viện khác sử dụng `RetryPolicy`. Một nhà phát triển có thể nhập `from some_library import RetryPolicy` một cách chính xác nhưng sau đó vô tình cố gắng sử dụng `retry_policy()` sau đó trong mã, dẫn đến `NameError`. Tương tự, nếu `Retry` là một lớp hoặc hàm tùy chỉnh được định nghĩa trong dự án, một lỗi đánh máy trong định nghĩa của nó hoặc trong bất kỳ tham chiếu nào sau đó đến nó sẽ gây ra lỗi tương tự. Điều này cũng có thể mở rộng đến các tình huống trong đó một bí danh được dự định trong quá trình nhập (ví dụ: `from urllib3.util.retry import Retry as HttpRetry`), nhưng tên gốc `Retry` vẫn được sử dụng sau đó trong mã thay vì `HttpRetry`. Mặc dù các Môi trường phát triển tích hợp (IDE) hiện đại thường cung cấp tính năng tự động hoàn thành và phân tích tĩnh có thể làm nổi bật các vấn đề như vậy, nhưng các công cụ này không phải là không có lỗi, đặc biệt trong mã được tạo động hoặc cấu trúc nhập phức tạp. Đánh giá mã thủ công và kiểm thử kỹ lưỡng vẫn là điều cần thiết để bắt các lỗi tinh vi nhưng có tác động này trước khi triển khai (Stack Overflow, n.d.).

Các hạn chế truy cập liên quan đến phạm vi

Các quy tắc về phạm vi của Python quy định nơi một tên (như `Retry`) có thể truy cập được trong một chương trình. `NameError` có thể xảy ra nếu `Retry` được định nghĩa, nhưng không nằm trong phạm vi mà nó đang được truy cập. Python tuân theo quy tắc LEGB (Local, Enclosing function locals, Global, Built-in) để phân giải tên (Python Software Foundation, n.d.-d). Nếu `Retry` được định nghĩa bên trong một hàm, nó là cục bộ đối với hàm đó và không thể truy cập trực tiếp từ bên ngoài nó. Ví dụ, nếu một lớp `Retry` tùy chỉnh được định nghĩa trong một hàm trợ giúp `def configure_retries(): class Retry: ...`, việc cố gắng sử dụng `Retry()` trực tiếp trong tập lệnh chính `my_retry_instance = Retry()` sẽ dẫn đến `NameError` vì `Retry` chỉ được biết đến trong `configure_retries()`.

Tương tự, nếu `Retry` được định nghĩa trong một phương thức lớp hoặc dưới dạng một thuộc tính lớp, việc truy cập nó có thể yêu cầu tham chiếu rõ ràng thông qua `self.Retry` (đối với phương thức thể hiện) hoặc `ClassName.Retry` (đối với quyền truy cập cấp lớp hoặc phương thức tĩnh). Nếu mã cố gắng sử dụng `Retry` mà không có tiền tố thích hợp, nó sẽ được coi là một tên không xác định trong phạm vi hiện tại. Đây là một cạm bẫy phổ biến đối với các nhà phát triển, đặc biệt là những người mới làm quen với lập trình hướng đối tượng trong Python. Một kịch bản khác liên quan đến các hàm lồng nhau hoặc closures, nơi `Retry` có thể được định nghĩa trong một hàm bên ngoài nhưng đang được truy cập từ một hàm bên trong không có quyền truy cập vào phạm vi cục bộ của hàm bên ngoài, hoặc ngược lại, mà không có khai báo `nonlocal` hoặc `global` phù hợp (thường được sử dụng để sửa đổi biến, không chỉ truy cập tên).

cli_executor chạy một tập lệnh có thể gặp phải điều này nếu cấu trúc của tập lệnh vô tình đặt định nghĩa Retry trong một phạm vi hạn chế, ngăn chặn việc sử dụng dự kiến của nó trong logic xử lý cho các URL như https://batdongsan.com.vn/ban-can-ho-chung-cu-tp-hcm. Hiểu và áp dụng đúng các quy tắc về phạm vi của Python là rất quan trọng để tránh các trường hợp NameError như vậy, đảm bảo rằng mọi tên đều có thể truy cập được từ nơi nó được dự định sử dụng (GeeksforGeeks, n.d.).

Không tương thích phiên bản và thay đổi API

Sự không tương thích phiên bản và các thay đổi API sau đó trong các thư viện bên thứ ba là một nguyên nhân thường xuyên và thường tinh tế gây ra NameError: name 'Retry' is not defined. Các thư viện phần mềm phát triển theo thời gian và các nhà phát triển thường giới thiệu các thay đổi phá vỡ, đổi tên lớp hoặc hàm, di chuyển chúng sang các mô-đun khác hoặc thậm chí xóa chúng hoàn toàn giữa các phiên bản chính. Nếu cli_executor đang chạy một tập lệnh được phát triển dựa trên một phiên bản thư viện (ví dụ: urllib3 hoặc tenacity) nhưng đang thực thi trong một môi trường mà một phiên bản khác, không tương thích được cài đặt, đối tượng Retry có thể không còn khả dụng ở đường dẫn mong đợi hoặc với tên mong đợi của nó (Semantic Versioning, n.d.).

Ví dụ, một phiên bản cũ hơn của urllib3 có thể đã hiển thị Retry trực tiếp dưới urllib3.Retry, trong khi một phiên bản mới hơn đã chuyển nó đến urllib3.util.retry.Retry. Nếu mã được viết cho phiên bản trước và được thực thi với phiên bản sau, một ImportError có thể sẽ xảy ra, nếu không bị bắt, có thể dẫn đến NameError khi Retry được tham chiếu sau đó. Ngược lại, nếu Retry được giới thiệu trong một phiên bản sau của thư viện, và môi trường cli_executor có một phiên bản cũ hơn được cài đặt, tên đó đơn giản là sẽ không tồn tại. Điều này đặc biệt liên quan đến các thư viện cung cấp các mẫu tiện ích chung như thử lại, vì API của chúng có thể được tinh chỉnh. Thư viện tenacity, chẳng hạn, được thiết kế đặc biệt cho logic thử lại, và mặc dù decorator retry cốt lõi của nó ổn định, các lớp trợ giúp hoặc cấu hình cụ thể có thể thay đổi giữa các phiên bản. Các nhà phát triển thường quản lý các phụ thuộc này bằng cách sử dụng các tệp requirements.txt hoặc pyproject.toml, chỉ định các phạm vi phiên bản chính xác hoặc tương thích (ví dụ: urllib3==1.26.5 hoặc tenacity>=8.0.0,<9.0.0) (PyPA, n.d.-b). Tuy nhiên, nếu môi trường triển khai (ví dụ: một quy trình CI/CD, một bản dựng hình ảnh Docker hoặc một thiết lập phát triển cục bộ) không tuân thủ nghiêm ngặt các thông số kỹ thuật phiên bản này, hoặc nếu một cài đặt toàn cầu ghi đè các phiên bản cụ thể của dự án, thì các ngoại lệ NameError do thay đổi API có thể xuất hiện. Kiểm tra định kỳ các phiên bản gói đã cài đặt và tuân thủ nghiêm ngặt các thực tiễn tốt nhất về quản lý phụ thuộc là rất quan trọng để giảm thiểu các vấn đề này và đảm bảo rằng mã chạy nhất quán trên các môi trường khác nhau (The Hitchhiker's Guide to Python, n.d.).

Tác động đến việc thu thập dữ liệu và độ tin cậy của hệ thống

Gián đoạn thu thập dữ liệu tức thì và mất dữ liệu

Việc xảy ra lỗi NameError: name 'Retry' is not defined trong một phiên cli_executor, đặc biệt khi cố gắng xử lý một URL quan trọng như https://batdongsan.com.vn/ban-can-ho-chung-cu-tp-hcm, báo hiệu một thất bại tức thì và nghiêm trọng trong quá trình thu thập dữ liệu. Lỗi này cho thấy rằng một thành phần quan trọng được thiết kế để tăng cường độ mạnh mẽ – một cơ chế thử lại – đã được tham chiếu nhưng không được khởi tạo hoặc nhập đúng cách vào phạm vi thực thi. Do đó, khả năng của hệ thống để xử lý các vấn đề mạng tạm thời, giới hạn tỷ lệ từ phía máy chủ hoặc tài nguyên mục tiêu tạm thời không khả dụng bị tổn hại hoàn toàn. Khi lỗi này xuất hiện, tác vụ thu thập dữ liệu hiện tại cho URL được chỉ định, và có thể tất cả các tác vụ tiếp theo trong phiên đó, sẽ bị chấm dứt đột ngột. Điều này trực tiếp dẫn đến việc mất dữ liệu ngay lập tức đáng lẽ đã được thu thập từ batdongsan.com.vn cho các danh sách bất động sản cụ thể đó. Ví dụ, nếu cli_executor được cấu hình để cào hàng nghìn danh sách tài sản hàng ngày, một NameError duy nhất có thể ngăn chặn việc thu thập hàng trăm hoặc thậm chí hàng nghìn điểm dữ liệu liên quan đến giá tài sản, tình trạng sẵn có, tính năng và thông tin liên hệ của đại lý từ lần chạy cụ thể đó.

Việc thiếu cơ chế Retry được định nghĩa có nghĩa là bất kỳ trục trặc tạm thời nào, chẳng hạn như tăng đột biến độ trễ mạng ngắn, máy chủ trả về trạng thái 503 Service Unavailable hoặc phản hồi 429 Too Many Requests, sẽ dẫn đến một lỗi dứt khoát chứ không phải là một trở ngại tạm thời. Trong một hoạt

động cào web điển hình, các lỗi tạm thời có thể chiếm 5-15% tổng số lỗi yêu cầu, tùy thuộc vào sự ổn định của trang web mục tiêu và giới hạn tỷ lệ (Thực tiễn tốt nhất khi cào web). Nếu không có `Retry`, những lỗi tạm thời này sẽ trở thành lỗi vĩnh viễn đối với phiên thực thi cụ thể đó. Điều này ảnh hưởng trực tiếp đến tính đầy đủ của tập dữ liệu. Đối với một nền tảng bất động sản, thiếu các điểm dữ liệu trên các danh sách mới hoặc cập nhật giá có thể có những tác động tài chính đáng kể, có khả năng trì hoãn thông tin chi tiết về thị trường hoặc dẫn đến bỏ lỡ các cơ hội đầu tư. Phiên `a5ce5514-1874-42b8-9393-ac600c1d57d` thực tế trở nên vô dụng cho mục đích dự kiến của nó, đòi hỏi phải can thiệp thủ công hoặc khởi động lại hoàn toàn, làm trầm trọng thêm sự chậm trễ trong việc thu thập dữ liệu. Dữ liệu *có thể* đã được thu thập trong phiên bị lỗi này bị mất vĩnh viễn trong khoảng thời gian cụ thể đó, đòi hỏi phải chạy lại hoặc chấp nhận thông tin không đầy đủ. Việc ngừng dòng dữ liệu ngay lập tức này là hậu quả trực tiếp và rõ ràng nhất của `NameError`, làm ngừng mục đích chính của hoạt động `cli_executor`.

Giảm chất lượng dữ liệu và tính toàn vẹn của dữ liệu

Ngoài việc mất dữ liệu tức thì, bản chất tái diễn của lỗi `NameError: name 'Retry' is not defined` có thể dẫn đến sự suy giảm có hệ thống về độ tươi mới và tính đầy đủ của dữ liệu. Các hệ thống thu thập dữ liệu, đặc biệt là những hệ thống liên quan đến nội dung động như danh sách bất động sản, phụ thuộc rất nhiều vào các bản cập nhật kịp thời và nhất quán để duy trì tính liên quan. Khi cơ chế thử lại không hoạt động, hệ thống không thể phục hồi sau các lỗi tạm thời, dẫn đến gián đoạn thường xuyên trong đường ống dữ liệu. Mỗi điểm lỗi gây ra sự chậm trễ trong việc thu thập thông tin mới hoặc cập nhật. Ví dụ, nếu `cli_executor` được lên lịch chạy hàng giờ để ghi lại các thay đổi thời gian thực về tình trạng sẵn có của tài sản hoặc điều chỉnh giá trên `batdongsan.com.vn`, một `NameError` dai dẳng có nghĩa là các bản cập nhật này không được ghi lại cho đến khi vấn đề mã cơ bản được giải quyết và quy trình khởi động lại thành công. Điều này có thể dẫn đến dữ liệu trở nên cũ kỹ trong vài giờ, hoặc thậm chí vài ngày, tùy thuộc vào tần suất lỗi và tốc độ giải quyết.

Hãy xem xét một kịch bản trong đó các danh sách tài sản mới được đăng mỗi 30 phút. Nếu tập lệnh thu thập dữ liệu bị lỗi trong 4 giờ do lỗi `NameError` này, khoảng 8 chu kỳ danh sách mới có thể bị bỏ lỡ, dẫn đến một khoảng trống đáng kể trong tập dữ liệu. Điều này ảnh hưởng trực tiếp đến "chỉ số độ tươi mới" của dữ liệu được thu thập. Đối với các doanh nghiệp dựa vào dữ liệu này để phân tích cạnh tranh, xác định xu hướng thị trường hoặc cảnh báo tự động, dữ liệu cũ có thể dẫn đến kết luận không chính xác và ra quyết định kém hiệu quả. Một nghiên cứu của IBM ước tính rằng chất lượng dữ liệu kém gây thiệt hại cho nền kinh tế Hoa Kỳ 3,1 nghìn tỷ đô la hàng năm, với việc dữ liệu cũ kỹ là một yếu tố đóng góp đáng kể (IBM). Tính đầy đủ của tập dữ liệu cũng bị ảnh hưởng. Thay vì một bản ghi toàn diện về tất cả các tài sản được liệt kê trong một khoảng thời gian nhất định, tập dữ liệu sẽ chứa các khoảng trống tương ứng với các giai đoạn hệ thống bị lỗi. Sự không đầy đủ này có thể làm sai lệch các mô hình phân tích, dẫn đến các thông tin chi tiết bị sai lệch. Ví dụ, nếu hệ thống liên tục không cào được dữ liệu trong giờ cao điểm đăng tin, dữ liệu thu thập được có thể không đại diện đầy đủ khối lượng hoặc sự đa dạng thực sự của các tài sản có sẵn, dẫn đến một cái nhìn thị trường không đầy đủ. Tác động tích lũy của những thất bại này là một tập dữ liệu không hiện tại cũng không đầy đủ, làm suy yếu giá trị và độ tin cậy của nó đối với bất kỳ ứng dụng nào được xây dựng dựa trên nó.

Sự mất ổn định hoạt động và kém hiệu quả tài nguyên

Lỗi `NameError: name 'Retry' is not defined` không chỉ là một vấn đề về thu thập dữ liệu; nó về cơ bản làm tổn hại đến sự ổn định hoạt động và hiệu quả của toàn bộ hệ thống. Một `cli_executor` được thiết kế cho các tác vụ tự động được mong đợi sẽ chạy tự chủ và đáng tin cậy. Một lỗi nghiêm trọng như `NameError` khiến quy trình bị lỗi, đòi hỏi sự can thiệp thủ công để gỡ lỗi và khởi động lại. Điều này gây ra chi phí hoạt động đáng kể. Các quản trị viên hệ thống hoặc nhà phát triển phải dành thời gian quý báu để xác định nguyên nhân gốc rễ (ví dụ: thiếu câu lệnh nhập, thiết lập môi trường không chính xác), triển khai các bản sửa lỗi và khởi tạo lại các phiên bị lỗi. Chu trình bảo trì phản ứng này làm lệch hướng tài nguyên khỏi phát triển chủ động hoặc cải thiện hệ thống. Theo một báo cáo của Gartner, thời gian ngừng hoạt động của CNTT có thể tiêu tốn của các doanh nghiệp trung bình 5.600 đô la mỗi phút, với các lỗi ứng dụng quan trọng như quy trình thu thập dữ liệu đóng góp đáng kể vào con số này (Gartner). Mặc dù lỗi cụ thể này có thể không luôn dẫn đến thời gian ngừng hoạt động toàn hệ thống, nhưng nó chắc chắn dẫn đến thời gian

ngừng hoạt động cấp ứng dụng cho thành phần thu thập dữ liệu.

Hơn nữa, việc thực thi lặp đi lặp lại một tập lệnh bị lỗi dẫn đến sự kém hiệu quả đáng kể về tài nguyên. Mỗi khi `cli_executor` cố gắng xử lý `https://batdongsan.com.vn/ban-can-ho-chung-cu-tp-hcm` và gặp lỗi `NameError`, các tài nguyên tính toán như chu kỳ CPU, bộ nhớ và băng thông mạng sẽ bị tiêu thụ mà không tạo ra bất kỳ đầu ra hiệu quả nào. Hệ thống có thể khởi tạo nhiều phiên hoặc thử lại ở cấp độ cao hơn (ví dụ: cấp độ điều phối), nhưng mỗi lần thử đều thất bại ở cùng một điểm. Điều này tạo ra một chu kỳ lãng phí tài nguyên. Ví dụ, nếu tập lệnh là một phần của môi trường vùng chứa, các vùng chứa mới có thể được khởi động và dừng liên tục, gây ra chi phí điện toán đám mây mà không mang lại giá trị. Một hoạt động cào web điển hình có thể liên quan đến hàng trăm hoặc hàng nghìn yêu cầu mỗi phút. Nếu 10% trong số các lần thử này thất bại do `NameError` trước khi bất kỳ dữ liệu nào được thu thập, 10% tài nguyên tính toán và mạng được phân bổ cho khoảng thời gian đó sẽ bị lãng phí. Theo thời gian, những sự kém hiệu quả này có thể tích lũy, dẫn đến chi phí vận hành cao hơn và giảm lợi tức đầu tư cho cơ sở hạ tầng thu thập dữ liệu. Độ tin cậy của hệ thống bị suy giảm nghiêm trọng vì nó dễ bị lỗi thường xuyên, có thể dự đoán được mà đòi hỏi sự giám sát liên tục của con người, biến một quy trình tự động thành một quy trình bán thủ công.

Ảnh hưởng đến các đường ống dữ liệu hạ nguồn và phân tích

Việc thu thập dữ liệu thất bại do lỗi `NameError: name 'Retry' is not defined` có những ảnh hưởng sâu rộng đến các đường ống dữ liệu hạ nguồn và các quy trình phân tích. Thu thập dữ liệu thường là bước nền tảng trong quy trình làm việc dữ liệu đa giai đoạn. Khi giai đoạn ban đầu này gặp trục trặc, các giai đoạn tiếp theo phụ thuộc vào dữ liệu đã nhập sẽ bị thiếu đầu vào hoặc nhận dữ liệu không đầy đủ/cũ, dẫn đến một chuỗi thất bại hoặc kết quả bị tổn hại. Ví dụ, nếu `cli_executor` chịu trách nhiệm cấp dữ liệu danh sách bất động sản cho một kho dữ liệu, lỗi `NameError` có nghĩa là kho dữ liệu sẽ không nhận được dữ liệu.

Các chiến lược gỡ lỗi và giải quyết

Phân tích lỗi ban đầu và xác định phạm vi

Thông báo lỗi `WARNING:cli_executor:Error in session a5ce5514-1874-42b8-9393-ac6009c1d57d: Error processing https://batdongsan.com.vn/ban-can-ho-chung-cu-tp-hcm: name 'Retry' is not defined` đặc biệt chỉ ra một `NameError` trong Python. Loại lỗi này xảy ra khi một chương trình cố gắng sử dụng một biến, hàm hoặc tên lớp chưa được định nghĩa hoặc nhập vào phạm vi hiện tại (Tài liệu Python). Trong ngữ cảnh này, tên `Retry` đang được gọi hoặc tham chiếu, nhưng trình thông dịch Python không thể định vị định nghĩa của nó.

Tiền tố `cli_executor` cho thấy lỗi này bắt nguồn từ môi trường thực thi giao diện dòng lệnh (CLI) hoặc một tập lệnh được thiết kế để chạy qua CLI. Sự hiện diện của `ID phiên` (a5ce5514-1874-42b8-9393-ac6009c1d57d) ngụ ý một luồng thực thi có cấu trúc, có khả năng liên quan đến nhiều tác vụ hoặc một quy trình chạy dài. URL cụ thể `https://batdongsan.com.vn/ban-can-ho-chung-cu-tp-hcm` cho thấy rằng `cli_executor` có thể đang thực hiện một hoạt động liên quan đến web, chẳng hạn như cào dữ liệu, tương tác API hoặc truy xuất nội dung. Các hoạt động như vậy vốn dĩ dễ bị ảnh hưởng bởi các vấn đề mạng tạm thời, lỗi phía máy chủ hoặc giới hạn tỷ lệ, khiến các cơ chế thử lại mạnh mẽ trở thành một thành phần phổ biến và cần thiết trong thiết kế của chúng.

Giai đoạn gỡ lỗi ban đầu phải tập trung vào việc xác định chính xác dòng mã mà `Retry` được tham chiếu mà không có định nghĩa. Điều này bao gồm: 1. **Xem xét toàn bộ dấu vết ngăn xếp:** Mặc dù không được cung cấp trong lời nhắc, một dấu vết ngăn xếp hoàn chỉnh là rất quan trọng. Nó sẽ hiển thị chuỗi các lệnh gọi hàm dẫn đến `NameError`, cho biết tệp và số dòng nơi `Retry` được gặp lần đầu tiên. Đây là con đường trực tiếp nhất để xác định phân đoạn mã có vấn đề (Real Python). 2. **Xác định ngữ cảnh thực thi của `cli_executor`:** Xác định xem `cli_executor` là một tập lệnh tùy chỉnh, một khung làm việc hay một công cụ của bên thứ ba. Hiểu kiến trúc của nó có thể giúp thu hẹp nơi chứa logic tùy chỉnh (có thể sử dụng `Retry`) hoặc mã liên quan đến phụ thuộc. Chẳng hạn, nếu đó là một tập lệnh tùy chỉnh, lỗi có thể nằm trong cơ sở mã của nó hoặc một mô-đun mà nó nhập. Nếu đó là một khung làm việc, lỗi có thể nằm trong cấu hình do người dùng cung cấp hoặc các hàm callback. 3. **Tìm kiếm `Retry` trong dự án:** Thực

hiện tìm kiếm toàn diện trên toàn bộ cơ sở mã của dự án cho chuỗi `Retry`. Việc tìm kiếm này phải bao gồm tất cả các tệp Python (`.py`), tệp cấu hình và có thể là tệp mẫu nếu có liên quan đến việc tạo mã động. Mục tiêu là tìm nơi `Retry` *được dự định* sử dụng và, quan trọng hơn, nơi định nghĩa hoặc câu lệnh nhập của nó *bị thiếu*.

`NameError` thường là một vấn đề đơn giản, thường xuất phát từ lỗi đánh máy, câu lệnh nhập bị quên hoặc cấu hình môi trường sai. Thách thức nằm ở việc định vị hiệu quả nguồn gốc trong một cơ sở mã có khả năng lớn hoặc phức tạp, đặc biệt khi lỗi xảy ra trong một khung thực thi cụ thể như `cli_executor`.

Kiểm tra cơ sở mã để định nghĩa `Retry`

Sau phân tích ban đầu, việc kiểm tra chi tiết cơ sở mã là điều cần thiết để giải quyết lỗi `NameError: name 'Retry' is not defined`. Giai đoạn này tập trung vào việc kiểm tra một cách có hệ thống các nguyên nhân phổ biến của tên không xác định trong Python, đặc biệt trong ngữ cảnh của một `cli_executor` xử lý các yêu cầu web.

1. **Thiếu câu lệnh import:** Nguyên nhân phổ biến nhất của `NameError` đối với một tên có vẻ hợp lệ như `Retry` là một câu lệnh import bị quên hoặc không chính xác. `Retry` là một tên lớp hoặc hàm phổ biến được sử dụng trong nhiều thư viện Python được thiết kế để xử lý các lỗi tạm thời và thử lại các hoạt động. Ví dụ bao gồm:
   - `tenacity`: Một thư viện phổ biến để thêm khả năng thử lại vào các hàm (tenacity GitHub). Nó thường liên quan đến việc import `retry` và `stop_after_attempt` hoặc `wait_fixed`.
   - `retrying`: Một thư viện khác cung cấp các decorator để thử lại các hàm (retrying GitHub).
   - `requests-toolbelt`: Chứa các bộ điều hợp thử lại cho thư viện `requests` (requests-toolbelt Docs).
   - Logic thử lại tùy chỉnh: Dự án có thể có lớp hoặc hàm `Retry` riêng được định nghĩa trong một mô-đun cục bộ.

   Cơ sở mã nên được quét tìm `import Retry`, `from some_module import Retry` hoặc `import some_module` theo sau là `some_module.Retry`. Nếu `Retry` được mong đợi là một lớp hoặc hàm từ một thư viện bên thứ ba, việc xác minh sự hiện diện của nó trong các câu lệnh `import` của các tệp liên quan là tối quan trọng.
2. **Lỗi đánh máy và phân biệt chữ hoa chữ thường:** Python phân biệt chữ hoa chữ thường. `Retry`, `retry`, `RETRY` hoặc `reTry` đều là những tên riêng biệt. Một lỗi phổ biến là định nghĩa `retry` nhưng cố gắng sử dụng `Retry`, hoặc ngược lại. Cần phải xem xét kỹ lưỡng mã xung quanh vị trí lỗi được xác định (từ dấu vết ngăn xếp) để tìm các sự khác biệt như vậy. Các công cụ kiểm tra mã tự động (linters) và IDE thường làm nổi bật các tên không xác định, nhưng việc kiểm tra thủ công có thể phát hiện các vấn đề về chữ hoa chữ thường tinh vi.
3. **Vấn đề về phạm vi:** Ngay cả khi `Retry` được định nghĩa hoặc import, nó có thể không truy cập được trong phạm vi cụ thể nơi `NameError` xảy ra. Điều này có thể xảy ra nếu:
   - `Retry` được định nghĩa trong một hàm hoặc phương thức và sau đó được cố gắng sử dụng bên ngoài phạm vi đó.
   - `Retry` được import trong một hàm cục bộ, khiến nó không khả dụng toàn cầu hoặc trong các hàm khác.
   - `cli_executor` tải động các mô-đun hoặc thực thi các đoạn mã mà các lệnh import cần thiết không được truyền đúng cách. Ví dụ, nếu `cli_executor` xử lý một tệp cấu hình chứa mã Python để thực thi, đoạn mã đó có thể thiếu các câu lệnh `import` cần thiết, ngay cả khi tập lệnh `cli_executor` chính có chúng.
4. **Import có điều kiện hoặc tải động:** Trong một số hệ thống phức tạp, các mô-đun hoặc lớp có thể được import có điều kiện hoặc tải động dựa trên cấu hình hoặc điều kiện thời gian chạy. Nếu điều kiện để import `Retry` không được đáp ứng, hoặc nếu cơ chế tải động bị lỗi, nó sẽ dẫn đến `NameError`. Điều này đòi hỏi phải kiểm tra bất kỳ câu lệnh `if` nào xung quanh các câu lệnh import hoặc bất kỳ logic tải mô-đun tùy chỉnh nào trong kiến trúc của `cli_executor`.

5. **Triển khai `Retry` tùy chỉnh:** Nếu `Retry` được dự định là một lớp hoặc hàm tùy chỉnh được phát triển trong dự án, tệp định nghĩa của nó phải được định vị. Xác minh rằng tệp này được đặt đúng cách trong cấu trúc dự án và nó được import một cách thích hợp bởi bất kỳ mô-đun nào cố gắng sử dụng `Retry`. Ví dụ, nếu `Retry` được định nghĩa trong `my_project/utils/retry_logic.py`, thì bất kỳ tệp nào sử dụng nó phải có `from my_project.utils.retry_logic import Retry`.

Việc tìm kiếm có hệ thống bằng cách sử dụng `grep` hoặc tính năng "Tìm trong tệp" của IDE cho `def Retry` hoặc `class Retry` cùng với `import Retry` sẽ giúp nhanh chóng xác định nguồn và các mẫu sử dụng dự định của đối tượng `Retry`.

Quản lý phụ thuộc và xác minh môi trường

Lỗi `NameError: name 'Retry' is not defined` thường có thể là triệu chứng của một môi trường Python được cấu hình không đúng hoặc các vấn đề với quản lý phụ thuộc, đặc biệt trong ngữ cảnh `cli_executor` có thể chạy trong các môi trường khác nhau (phát triển, thử nghiệm, sản xuất, CI/CD).

1. **Tính toàn vẹn của môi trường ảo:**
   - **Kích hoạt:** Đảm bảo rằng môi trường ảo Python chính xác được kích hoạt trước khi chạy `cli_executor`. Nếu tập lệnh được chạy bằng trình thông dịch Python mặc định của hệ thống, nó có thể không có quyền truy cập vào các gói được cài đặt trong môi trường ảo cụ thể của dự án. Đây là một sự bỏ sót phổ biến, đặc biệt trong các tập lệnh tự động hoặc quy trình CI/CD nơi lệnh kích hoạt có thể bị bỏ qua (Hướng dẫn sử dụng gói Python).
   - **Cách ly:** Xác minh rằng môi trường ảo thực sự được cách ly. Đôi khi, các gói site-packages toàn cầu có thể "rò rỉ" vào môi trường ảo nếu không được cấu hình đúng cách, dẫn đến hành vi không mong muốn hoặc xung đột phiên bản.
2. **Cài đặt gói bắt buộc:**
   - **`requirements.txt` / `pyproject.toml`:** Kiểm tra tệp kê khai phụ thuộc của dự án (ví dụ: `requirements.txt` cho `pip`, `pyproject.toml` cho `Poetry` hoặc `Rye`) để xác nhận rằng thư viện cung cấp chức năng `Retry` được liệt kê. Chẳng hạn, nếu `tenacity` được sử dụng, `tenacity` phải có mặt trong phần `install_requires` hoặc dưới dạng phụ thuộc.
   - **Xác minh cài đặt:** Sau khi đảm bảo phụ thuộc được liệt kê, xác nhận rằng nó thực sự được cài đặt trong môi trường ảo đang hoạt động. Điều này có thể được thực hiện bằng cách sử dụng `pip freeze` hoặc `pip list`. Đầu ra phải hiển thị rõ ràng thư viện mong đợi và phiên bản của nó.
     ```
     # Ví dụ đầu ra nếu tenacity được cài đặt
     tenacity==8.2.3
     ```
   - **Cài đặt lại:** Nếu có bất kỳ nghi ngờ nào, bạn nên cài đặt lại sạch sẽ các phụ thuộc trong môi trường ảo:
     ```
     deactivate # nếu đang hoạt động
     rm -rf .venv # hoặc bất kỳ nơi nào môi trường ảo của bạn
     python3 -m venv .venv
     source .venv/bin/activate
     pip install -r requirements.txt
     ```
     Điều này đảm bảo tất cả các gói được cài đặt mới và liên kết chính xác với môi trường.
3. **Khả năng tương thích phiên bản Python:**
   - Mặc dù `NameError` ít liên quan trực tiếp đến khả năng tương thích phiên bản Python hơn là lỗi cú pháp, nhưng có thể một thư viện cung cấp `Retry` có các đường dẫn nhập khác nhau hoặc không tương thích với phiên bản Python mà `cli_executor` đang sử dụng.
   - Xác minh phiên bản Python (`python --version` hoặc `python3 --version`) trong môi trường thực thi khớp với phiên bản dự kiến của dự án và các yêu cầu tương thích của thư viện.
4. **Biến môi trường và cấu hình:**
   - Một số khung `cli_executor` hoặc tập lệnh tùy chỉnh có thể dựa vào các biến môi trường để xác định đường dẫn mô-đun, bật/tắt tính năng hoặc tải các cấu hình cụ thể ảnh hưởng đến việc mô-đun nào được nhập. Xem xét bất kỳ biến môi trường liên quan nào (ví dụ: `PYTHONPATH`) có thể ảnh hưởng đến việc phân giải mô-đun.
   - Kiểm tra các tệp cấu hình (ví dụ: `.ini`, `.yaml`, `.json`) mà `cli_executor` có thể tiêu thụ. Các

tệp này có thể chỉ định chiến lược thử lại nào sẽ sử dụng hoặc mô-đun nào sẽ tải, và một giá trị không chính xác có thể dẫn đến việc `Retry` không được định nghĩa nếu logic nhập tương ứng bị bỏ qua. Một kịch bản phổ biến là mã hoạt động trong môi trường cục bộ của nhà phát triển nhưng thất bại trong môi trường triển khai. Sự khác biệt này gần như luôn chỉ ra sự khác biệt trong các phụ thuộc đã cài đặt, phiên bản Python hoặc cấu hình biến môi trường. Việc ghi lại thiết lập môi trường chính xác (phiên bản Python, đầu ra `pip freeze`) cho các lần chạy thành công có thể vô giá để gỡ lỗi các lỗi.

Gỡ lỗi thời gian chạy và nâng cao ghi nhật ký

Khi phân tích mã tĩnh và kiểm tra môi trường không ngay lập tức tiết lộ nguồn gốc của lỗi `NameError`, gỡ lỗi thời gian chạy và ghi nhật ký nâng cao trở nên rất quan trọng. Cách tiếp cận này cho phép quan sát trạng thái của chương trình và luồng thực thi tại thời điểm xảy ra lỗi.

1. **Tái tạo lỗi một cách nhất quán:**
   - Bước đầu tiên trong gỡ lỗi thời gian chạy là tái tạo lỗi một cách đáng tin cậy. Thông báo lỗi cung cấp một URL cụ thể (`https://batdongsan.com.vn/ban-can-ho-chung-cu-tp-hcm`) và một ID phiên. Cố gắng thực thi `cli_executor` với các tham số chính xác này.
   - Nếu lỗi là không liên tục, hãy cố gắng xác định các mẫu (ví dụ: các thời điểm cụ thể trong ngày, sau một số yêu cầu nhất định, dưới tải nặng). Điều này có thể chỉ ra một tình trạng cạnh tranh hoặc vấn đề cạn kiệt tài nguyên gián tiếp ngăn `Retry` không được định nghĩa hoặc nhập.
2. **Sử dụng trình gỡ lỗi của Python (`pdb`):**
   - `pdb` là trình gỡ lỗi mã nguồn tương tác của Python (Tài liệu Python - pdb). Nó cho phép bước qua mã, kiểm tra biến và đặt các điểm ngắt.
   - **Đặt điểm ngắt:** Chèn `import pdb; pdb.set_trace()` ngay trước dòng mã bị nghi ngờ nơi `Retry` được sử dụng. Nếu không biết chính xác dòng nào, hãy đặt nó một cách chiến lược trong các hàm hoặc mô-đun có khả năng liên quan đến quá trình xử lý yêu cầu web hoặc logic thử lại.
   - **Gỡ lỗi sau khi lỗi:** Nếu chương trình bị lỗi, `pdb` có thể được gọi sau khi lỗi bằng cách chạy `python -m pdb your_script.py` sau khi lỗi, hoặc bằng cách đặt biến môi trường `PYTHONBREAKPOINT`. Điều này cho phép kiểm tra dấu vết ngăn xếp và trạng thái biến tại điểm lỗi.
   - **Các lệnh `pdb`:** Các lệnh hữu ích bao gồm `n` (dòng tiếp theo), `s` (bước vào hàm), `c` (tiếp tục), `p <biến>` (in giá trị biến), `w` (vị trí/dấu vết ngăn xếp), `l` (liệt kê mã nguồn). Khi `NameError` xảy ra, `w` sẽ hiển thị toàn bộ ngăn xếp cuộc gọi, và `p Retry` sẽ xác nhận trạng thái chưa được định nghĩa của nó.
3. **Nâng cao ghi nhật ký:**
   - **Ghi nhật ký chi tiết:** Thêm các câu lệnh `logging.debug()` hoặc `logging.info()` một cách chiến lược trong suốt đường dẫn mã dẫn đến lỗi. Các thông báo nhật ký nên chỉ ra các điểm vào và ra của các hàm, giá trị của các biến chính và kết quả của logic điều kiện.
   - **Theo dõi nhập mô-đun:** Đặc biệt, thêm ghi nhật ký xung quanh bất kỳ câu lệnh `import` nào liên quan đến cơ chế thử lại hoặc mô-đun nơi `Retry` được mong đợi sẽ được định nghĩa. Ví dụ:
     ```
     try:
         from some_library import Retry
         logging.info("Đã nhập Retry từ some_library thành công.")
     except ImportError:
         logging.error("Không thể nhập Retry từ some_library. Vui lòng kiểm tra cài đặt.")
     except NameError as e:
         logging.error(f"NameError trong quá trình nhập hoặc định nghĩa: {e}")
     ```
   - **Ghi lại toàn bộ dấu vết ngăn xếp:** Đảm bảo rằng cấu hình ghi nhật ký của `cli_executor` được đặt để ghi lại toàn bộ dấu vết ngăn xếp cho các lỗi. Mô-đun `logging` của Python có thể tự động thực hiện điều này khi một ngoại lệ được ghi nhật ký: `logging.exception("Đã xảy ra lỗi")`. Điều này cung cấp ngữ cảnh có thể bị thiếu từ một thông báo `NameError` đơn giản.
   - **Điều chỉnh cấp độ ghi nhật ký:** Tạm thời tăng cấp độ ghi nhật ký (ví dụ: lên `DEBUG`) cho `cli_executor` và các phụ thuộc của nó. Điều này có thể tiết lộ các chi tiết cấp thấp hơn về việc tải mô-đun, phân tích cú pháp cấu hình và luồng thực thi có thể gián tiếp chỉ ra lý do tại sao `Retry` không được định nghĩa.

4. **Cách ly mã có vấn đề:**
    - Nếu `cli_executor` là một hệ thống phức tạp, hãy cố gắng tạo một ví dụ có thể tái tạo tối thiểu. Trích xuất logic cốt lõi tương tác với URL `https://batdongsan.com.vn/` và cơ chế thử lại vào một tập lệnh riêng biệt, đơn giản hơn. Điều này có thể giúp cách ly xem vấn đề là do chính định nghĩa `Retry` hay do cách môi trường `cli_executor` tương tác với nó.
    - Chú thích từng phần mã một cách tăng dần để thu hẹp khu vực có vấn đề. Chiến lược "chia để trị" này có thể hiệu quả trong các cơ sở mã lớn.

Bằng cách kết hợp gỡ lỗi tương tác với ghi nhật ký toàn diện, các nhà phát triển có thể có được những hiểu biết sâu sắc về quá trình thực thi của chương trình và nhanh chóng xác định thời điểm chính xác và lý do `Retry` không được định nghĩa.

Triển khai xử lý lỗi và thử lại mạnh mẽ (một cách đúng đắn)

Sau khi lỗi `NameError: name 'Retry' is not defined` được giải quyết bằng cách định nghĩa hoặc nhập đúng `Retry`, trọng tâm chuyển sang đảm bảo rằng cơ chế thử lại đó mạnh mẽ và xử lý hiệu quả các loại lỗi gặp phải trong quá trình xử lý web, chẳng hạn như khi truy cập `https://batdongsan.com.vn/`. Việc triển khai đúng cách các lần thử lại là rất quan trọng đối với độ tin cậy của bất kỳ hệ thống nào tương tác với các dịch vụ bên ngoài.

1. **Phân biệt lỗi có thể thử lại và không thể thử lại:**
    - Không phải tất cả các lỗi đều nên kích hoạt thử lại. **Lỗi có thể thử lại** thường là lỗi tạm thời, chẳng hạn như hết thời gian chờ mạng, thiết lập lại kết nối, lỗi phân giải DNS hoặc các vấn đề phía máy chủ được chỉ ra bởi mã trạng thái HTTP 5xx (ví dụ: 500 Internal Server Error, 502 Bad Gateway, 503 Service Unavailable, 504 Gateway Timeout) (RFC 7231). Giới hạn tỷ lệ (HTTP 429 Too Many Requests) cũng là một lỗi có thể thử lại phổ biến, thường yêu cầu tiêu đề `Retry-After`.
    - **Lỗi không thể thử lại** thường là vĩnh viễn và chỉ ra một vấn đề cơ bản mà việc thử lại sẽ không giải quyết được. Ví dụ bao gồm lỗi máy khách HTTP 4xx (ví dụ: 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found) (RFC 7231), hoặc lỗi logic trong mã ứng dụng. Thử lại những lỗi này sẽ lãng phí tài nguyên và có thể làm trầm trọng thêm vấn đề.
    - Logic thử lại nên định nghĩa rõ ràng những ngoại lệ hoặc mã trạng thái HTTP nào kích hoạt thử lại.
2. **Triển khai các chiến lược Backoff:**
    - Chỉ đơn giản thử lại ngay lập tức có thể làm quá tải một dịch vụ đang gặp khó khăn hoặc đạt đến giới hạn tỷ lệ nhanh hơn. **Các chiến lược Backoff** tạo ra độ trễ giữa các lần thử lại.
    - **Backoff lũy thừa:** Đây là chiến lược phổ biến nhất và được khuyến nghị. Độ trễ giữa các lần thử lại tăng theo cấp số nhân (ví dụ: 1s, 2s, 4s, 8s). Điều này cho dịch vụ từ xa thời gian để phục hồi và giảm tải. Nhiều thư viện thử lại như `tenacity` (tenacity GitHub) và `retrying` (retrying GitHub) cung cấp tính năng này.
    - **Jitter:** Để ngăn chặn vấn đề "thundering herd" nơi nhiều máy khách thử lại đồng thời sau một sự cố, nên thêm một lượng nhỏ jitter ngẫu nhiên vào độ trễ backoff. Điều này phân tán các lần thử lại, giảm tải tối đa.
    - **Backoff cố định:** Một độ trễ không đổi giữa các lần thử lại. Kém hiệu quả hơn backoff lũy thừa đối với hầu hết các lỗi tạm thời nhưng có thể được sử dụng cho các độ trễ rất cụ thể, có thể dự đoán được.
3. **Đặt giới hạn cho số lần thử lại:**
    - **Số lần thử tối đa:** Định nghĩa số lần thử lại tối đa. Thử lại vô thời hạn có thể dẫn đến vòng lặp vô hạn và cạn kiệt tài nguyên nếu lỗi thực sự là vĩnh viễn.
    - **Tổng thời gian chờ tối đa:** Ngoài ra, hoặc bổ sung, đặt tổng thời gian tối đa cho tất cả các lần thử lại. Điều này ngăn một hoạt động chặn vô thời hạn.
    - Khi đạt đến giới hạn, hoạt động phải thất bại và lỗi phải được truyền để xử lý cấp cao hơn (ví dụ: ghi nhật ký, cảnh báo, chuyển đến hàng đợi thư chết).
4. **Tích hợp với các thư viện cụ thể (ví dụ: `tenacity`):**
    - Đối với Python, các thư viện như `tenacity` cung cấp các decorator thử lại mạnh mẽ và linh hoạt.

- **Ví dụ sử dụng `tenacity`:**

```
from tenacity import retry, stop_after_attempt, wait_exponential, retry_if_exception_type
import requests
import logging

logging.basicConfig(level=logging.INFO)

@retry(
    stop=stop_after_attempt(5),
    wait=wait_exponential(multiplier=1, min=4, max=10), # 4s, 8s, 16s... max 10s
    retry=retry_if_exception_type(requests.exceptions.ConnectionError) |
          retry_if_exception_type(requests.exceptions.Timeout) |
          retry_if_exception_type(requests.exceptions.HTTPError),
    reraise=True # Re-raise the last exception if all retries fail
)
def fetch_url_with_retry(url):
    logging.info(f"Đang cố gắng tìm nạp {url}...")
    response = requests.get(url, timeout=5)
    response.raise_for_status() # Đưa ra HTTPError cho các phản hồi xấu (4xx hoặc 5xx)
    return response.text

try:
    content = fetch_url_with_retry("https://batdongsan.com.vn/ban-can-ho-chung-cu-tp-hcm")
    # Xử lý nội dung
except Exception as e:            logging.error(f"Không thể tìm nạp URL sau nhiều lần thử lại:
```

- Ví dụ này minh họa cách cấu hình `tenacity` để thử lại trên các ngoại lệ `requests` cụ thể, với backoff lũy thừa và số lần thử tối đa. Nó cũng cho thấy cách đưa ra lại ngoại lệ nếu tất cả các lần thử lại thất bại, cho phép xử lý lỗi tập trung.

5. **Mẫu ngắt mạch (Circuit Breaker):**
   - Để có khả năng phục hồi nâng cao hơn, hãy xem xét triển khai **mẫu ngắt mạch**. Mẫu này ngăn chặn các lần thử lặp đi lặp lại đến một dịch vụ không phản hồi, cho phép nó có thời gian phục hồi mà không làm quá tải thêm. Nếu một dịch vụ liên tục thất bại, ngắt mạch sẽ "ngắt", mở mạch và ngăn chặn các yêu cầu tiếp theo đến dịch vụ đó trong một khoảng thời gian được xác định trước.

Các biện pháp phòng ngừa và thực tiễn tốt nhất cho xử lý dữ liệu mạnh mẽ

Thực thi quản lý phụ thuộc nghiêm ngặt và các tiêu chuẩn chất lượng mã

Một biện pháp phòng ngừa cơ bản chống lại các lỗi như `name 'Retry' is not defined` nằm ở việc thiết lập và thực thi nghiêm ngặt quản lý phụ thuộc và các tiêu chuẩn chất lượng mã cao. Lỗi này thường báo hiệu rằng một mô-đun, lớp hoặc hàm cần thiết, trong trường hợp này là `Retry`, không được nhập đúng cách hoặc không có sẵn trong môi trường thực thi. Các hệ thống xử lý dữ liệu mạnh mẽ, đặc biệt là những hệ thống liên quan đến cào web hoặc các đường ống dữ liệu phức tạp, phụ thuộc rất nhiều vào nhiều thư viện bên ngoài và các mô-đun nội bộ. Việc thiếu quản lý phụ thuộc rõ ràng có thể dẫn đến các môi trường không nhất quán, nơi mã hoạt động cục bộ nhưng thất bại trong sản xuất do thiếu gói hoặc không khớp phiên bản (Katz, 2021).

Các thực tiễn tốt nhất yêu cầu sử dụng các công cụ quản lý phụ thuộc chuyên dụng. Đối với Python, điều này bao gồm `pip` với các tệp `requirements.txt`, hoặc các công cụ nâng cao hơn như Poetry hoặc PDM, quản lý môi trường ảo và khóa các phụ thuộc vào các phiên bản cụ thể. Chẳng hạn, một tệp `requirements.txt` nên liệt kê chính xác tất cả các phụ thuộc trực tiếp và gián tiếp với các phiên bản được ghim (ví dụ: `requests==2.28.1`, `tenacity==8.2.2` nếu `Retry` là một phần của `tenacity` hoặc một thư viện tương tự), đảm bảo rằng cùng một tập hợp các thư viện được cài đặt trên tất cả các môi trường phát triển, kiểm thử và sản xuất (Python Packaging Authority, n.d.). Điều này ngăn chặn các kịch bản mà một nhà phát triển

có thể đã cài đặt một gói toàn cục không được liệt kê rõ ràng hoặc cài đặt trong môi trường cô lập của dự án.

Ngoài việc khai báo phụ thuộc, các tiêu chuẩn chất lượng mã đóng một vai trò quan trọng. Các công cụ kiểm tra mã (linters) như Pylint hoặc Flake8 và các công cụ phân tích tĩnh như MyPy để kiểm tra kiểu, có thể xác định các vấn đề tiềm ẩn trước thời gian chạy. Các công cụ này có thể đánh dấu các lệnh nhập không sử dụng, các biến không xác định hoặc các lệnh gọi hàm không chính xác, điều này có thể gián tiếp dẫn đến `NameError` nếu một lệnh nhập quan trọng bị thiếu hoặc bị sai chính tả. Ví dụ, MyPy có thể phát hiện nếu một gợi ý kiểu cho đối tượng `Retry` được sử dụng mà không có `Retry` được nhập, nhắc nhà phát triển sửa câu lệnh nhập (MyPy, n.d.). Hơn nữa, các đánh giá mã bắt buộc đảm bảo rằng nhiều cặp mắt kiểm tra cơ sở mã để tìm các lệnh nhập chính xác, luồng logic và tuân thủ các quy ước mã hóa, giảm đáng kể khả năng các lỗi cơ bản như vậy lọt vào mã đã triển khai. Áp dụng triết lý "thất bại nhanh" trong phát triển, nơi các lỗi được phát hiện càng sớm càng tốt, là tối quan trọng. Điều này bao gồm việc đảm bảo rằng tất cả các mô-đun cần thiết, đặc biệt là những mô-đun cung cấp các chức năng quan trọng như cơ chế thử lại, được nhập rõ ràng ở đầu các tệp Python liên quan, làm cho sự hiện diện và phạm vi của chúng rõ ràng đối với cả nhà phát triển và các công cụ phân tích tĩnh (Fowler, 2004).

Triển khai xử lý lỗi nâng cao và các mẫu kiên cường

Trong khi quản lý phụ thuộc nghiêm ngặt ngăn chặn các vấn đề `NameError` cơ bản, xử lý dữ liệu mạnh mẽ đòi hỏi các mẫu xử lý lỗi và kiên cường nâng cao để quản lý các lỗi thời gian chạy, đặc biệt là các lỗi tạm thời phổ biến trong cào web (ví dụ: hết thời gian chờ mạng, giới hạn tỷ lệ từ phía máy chủ). Lỗi `name 'Retry' is not defined` ngụ ý một nỗ lực để triển khai cơ chế thử lại, nhưng không đầy đủ hoặc không chính xác. Logic thử lại được triển khai đúng cách là một nền tảng của xử lý dữ liệu kiên cường.

Một mẫu cơ bản là **Backoff lũy thừa với Jitter**. Thay vì chỉ đơn giản thử lại ngay lập tức, điều này có thể làm trầm trọng thêm các vấn đề như quá tải máy chủ, backoff lũy thừa tăng độ trễ giữa các lần thử lại theo cấp số nhân (ví dụ: 1s, 2s, 4s, 8s). Jitter thêm một độ trễ ngẫu nhiên nhỏ vào khoảng thời gian backoff này, ngăn chặn tất cả các máy khách thử lại đồng thời truy cập máy chủ chính xác cùng lúc, điều này có thể xảy ra nếu tất cả chúng sử dụng cùng một thuật toán backoff lũy thừa (AWS, n.d.). Các thư viện như `tenacity` trong Python cung cấp các decorator đơn giản hóa việc triển khai các chính sách thử lại tinh vi như vậy, cho phép các nhà phát triển định nghĩa các điều kiện để thử lại (ví dụ: trên các ngoại lệ cụ thể hoặc mã trạng thái HTTP), số lần thử lại tối đa và các chiến lược backoff (Tenacity, n.d.). Chẳng hạn, một hàm cố gắng tìm nạp dữ liệu từ `batdongsan.com.vn` có thể được trang trí để thử lại trên `requests.exceptions.ConnectionError` hoặc `requests.exceptions.Timeout` với backoff lũy thừa ngẫu nhiên lên đến tối đa 5 lần thử.

Một mẫu kiên cường quan trọng khác là **Ngắt mạch (Circuit Breaker)**. Không giống như các lần thử lại, vốn cố gắng phục hồi sau các lỗi tạm thời, ngắt mạch ngăn một ứng dụng liên tục cố gắng gọi một dịch vụ có khả năng thất bại. Nếu một dịch vụ liên tục thất bại, ngắt mạch sẽ "ngắt", mở mạch và ngăn chặn các cuộc gọi tiếp theo đến dịch vụ đó trong một khoảng thời gian được xác định trước. Điều này cho phép dịch vụ bị lỗi phục hồi mà không bị quá tải bởi các yêu cầu liên tục và ngăn ứng dụng gọi lãng phí tài nguyên vào các hoạt động thất bại (Nygard, 2018). Sau một thời gian chờ, mạch sẽ chuyển sang trạng thái "bán mở", cho phép một số lượng yêu cầu kiểm tra hạn chế để xác định xem dịch vụ đã phục hồi chưa. Nếu những yêu cầu này thành công, mạch sẽ đóng lại; nếu không, nó sẽ mở lại. Việc triển khai ngắt mạch cùng với logic thử lại cung cấp một lớp phòng thủ theo lớp chống lại sự mất ổn định của dịch vụ, đảm bảo rằng các đường ống xử lý dữ liệu có thể xử lý duyên dáng các sự cố ngừng hoạt động của dịch vụ bên ngoài hoặc suy giảm hiệu suất. Ví dụ, nếu `batdongsan.com.vn` liên tục trả về lỗi 5xx, một ngắt mạch có thể tạm thời dừng các nỗ lực cào web, ngăn tải không cần thiết trên cả trình cào và trang web mục tiêu.

Hơn nữa, việc đảm bảo **tính bất biến (idempotency)** cho các hoạt động xử lý dữ liệu là một thực tiễn tốt nhất. Một hoạt động bất biến là một hoạt động có thể được áp dụng nhiều lần mà không làm thay đổi kết quả ngoài lần áp dụng ban đầu. Điều này rất quan trọng khi các cơ chế thử lại được áp dụng, vì một hoạt động thất bại có thể đã hoàn thành một phần. Nếu xảy ra thử lại, một hoạt động bất biến đảm bảo rằng việc thực thi lại nó không dẫn đến dữ liệu trùng lặp hoặc thay đổi trạng thái không chính xác (Kleppmann, 2017). Chẳng hạn, khi chèn dữ liệu vào cơ sở dữ liệu, việc sử dụng các hoạt động "upsert" (cập nhật nếu tồn

tại, chèn nếu không) hoặc các ràng buộc duy nhất có thể làm cho quá trình chèn dữ liệu trở nên bất biến, ngăn chặn việc trùng lặp dữ liệu nếu lỗi mạng gây ra thử lại sau khi lần chèn ban đầu thành công nhưng trước khi nhận được xác nhận.

Tận dụng kiểm thử tự động và các đường ống Tích hợp/Triển khai liên tục (CI/CD)

Kiểm thử tự động và các đường ống CI/CD mạnh mẽ là các biện pháp phòng ngừa không thể thiếu đối với các lỗi thời gian chạy như `name 'Retry' is not defined` và các lỗi xử lý khác. Các thực hành này đảm bảo rằng chất lượng mã, chức năng và tính nhất quán của môi trường được xác thực trong suốt vòng đời phát triển, giảm đáng kể khả năng lỗi lọt vào sản xuất. **Kiểm thử đơn vị** là tuyến phòng thủ đầu tiên. Các nhà phát triển nên viết kiểm thử cho từng thành phần hoặc hàm riêng lẻ, bao gồm cả những thành phần chịu trách nhiệm xử lý lỗi và logic thử lại. Chẳng hạn, nếu một lớp hoặc hàm `Retry` tùy chỉnh được định nghĩa, kiểm thử đơn vị nên xác minh rằng nó xử lý đúng các loại ngoại lệ khác nhau, áp dụng các chiến lược backoff và tuân thủ giới hạn thử lại tối đa. Giả lập các phụ thuộc bên ngoài, chẳng hạn như các yêu cầu mạng đến `batdongsan.com.vn`, cho phép các kiểm thử này chạy nhanh chóng và đáng tin cậy mà không cần các lệnh gọi bên ngoài thực tế (Python Testing, n.d.). Điều này đảm bảo rằng logic cốt lõi của chính cơ chế thử lại là đúng đắn và được triển khai chính xác.

Kiểm thử tích hợp mở rộng điều này bằng cách xác minh sự tương tác giữa các thành phần khác nhau. Đối với một đường ống xử lý dữ liệu, điều này có thể liên quan đến việc kiểm thử sự tương tác của trình cào với cơ chế thử lại, trình phân tích dữ liệu và lớp lưu trữ. Các kiểm thử này có thể mô phỏng các kịch bản lỗi khác nhau, chẳng hạn như sự cố mạng hoặc giới hạn tỷ lệ API, để xác nhận rằng toàn bộ hệ thống phản hồi như mong đợi, bao gồm việc kích hoạt thử lại và xử lý các lỗi cuối cùng một cách duyên dáng. Một kiểm thử tích hợp có thể cố gắng cào một URL được biết là đôi khi bị lỗi và khẳng định rằng logic thử lại được gọi và cuối cùng thành công hoặc thất bại một cách thích hợp.

Một **đường ống Tích hợp liên tục (CI)** tự động hóa quá trình xây dựng và kiểm thử các thay đổi mã. Mỗi cam kết mã kích hoạt một quá trình xây dựng chạy tất cả các kiểm thử đơn vị và tích hợp, cùng với các công cụ phân tích mã tĩnh (linters, type checkers). Nếu bất kỳ kiểm thử nào thất bại hoặc một linter xác định một vấn đề nghiêm trọng (như một biến không xác định hoặc thiếu lệnh nhập), quá trình xây dựng sẽ thất bại, ngăn chặn mã có vấn đề được hợp nhất vào cơ sở mã chính (Fowler & Foemmel, 2006). Vòng lặp phản hồi tức thì này rất quan trọng để phát hiện các lỗi như `name 'Retry' is not defined` sớm, thường trong vòng vài phút sau khi nhà phát triển đưa lỗi vào. Theo một cuộc khảo sát năm 2022, các tổ chức sử dụng các thực hành CI/CD đã báo cáo giảm 20% các lỗi nghiêm trọng và thời gian phục hồi sau sự cố nhanh hơn 30% (DORA, 2022).

Triển khai liên tục (CD) tiến thêm một bước so với CI bằng cách tự động hóa việc triển khai mã đã được xác thực vào môi trường sản xuất. Điều này đảm bảo rằng mã chạy trong sản xuất luôn là phiên bản đã vượt qua tất cả các kiểm thử tự động. Các đường ống CD thường bao gồm các bước cung cấp môi trường, đảm bảo rằng tất cả các phụ thuộc cần thiết được cài đặt và cấu hình đúng cách trong môi trường mục tiêu. Điều này giảm thiểu lỗi của con người trong quá trình triển khai và đảm bảo rằng môi trường sản xuất nhất quán với môi trường đã kiểm thử, ngăn chặn các vấn đề mà một phụ thuộc như thư viện `Retry` có thể bị thiếu trong sản xuất mặc dù có mặt trong quá trình phát triển (Humble & Farley, 2010).

Thiết lập khả năng quan sát toàn diện và giám sát chủ động

Ngay cả với các biện pháp phòng ngừa mạnh mẽ, lỗi vẫn có thể xảy ra trong các hệ thống xử lý dữ liệu phức tạp. Do đó, việc thiết lập khả năng quan sát toàn diện và giám sát chủ động là một thực tiễn tốt nhất quan trọng để nhanh chóng phát hiện, chẩn đoán và giải quyết các vấn đề như `name 'Retry' is not defined` hoặc bất kỳ lỗi xử lý tiếp theo nào. Khả năng quan sát đề cập đến khả năng suy luận trạng thái bên trong của một hệ thống bằng cách kiểm tra các đầu ra bên ngoài của nó (số liệu, nhật ký, dấu vết), trong khi giám sát tập trung vào việc theo dõi các số liệu cụ thể và cảnh báo khi đạt đến ngưỡng được xác định trước (O'Reilly, 2020).

Ghi nhật ký có cấu trúc là cơ bản. Thay vì các câu lệnh in đơn giản, nhật ký nên được tạo ở định dạng có cấu trúc (ví dụ: JSON) bao gồm ngữ cảnh liên quan như dấu thời gian, cấp độ nhật ký (INFO, WARNING, ERROR), ID phiên, ID quy trình, tệp nguồn và các thông báo lỗi cụ thể. Đối

với một lỗi như `name 'Retry' is not defined`, một mục nhật ký có cấu trúc sẽ không chỉ ghi lại dấu vết `NameError` mà còn cả ID phiên (`a5ce5514-1874-42b8-9393-ac6009c1d57d`), URL đang được xử lý (`https://batdongsan.com.vn/ban-can-ho-chung-cu-tp-hcm`) và bất kỳ ngữ cảnh thực thi liên quan nào khác. Các hệ thống ghi nhật ký tập trung (ví dụ: ELK Stack, Splunk, Datadog) tổng hợp các nhật ký này, làm cho chúng có thể tìm kiếm, lọc và phân tích được, cho phép các nhóm vận hành nhanh chóng xác định nguồn và ngữ cảnh của lỗi trên các hệ thống phân tán (Elastic, n.d.).

Thu thập số liệu cung cấp thông tin chi tiết định lượng về tình trạng và hiệu suất của hệ thống. Các số liệu chính để xử lý dữ liệu bao gồm: * **Tỷ lệ thành công/thất bại:** Phần trăm các tác vụ xử lý dữ liệu thành công so với các tác vụ thất bại. * **Độ trễ:** Thời gian cần thiết để xử lý các mục riêng lẻ hoặc hoàn thành tác vụ. * **Thông lượng:** Số lượng mục được xử lý trên một đơn vị thời gian. * **Tỷ lệ lỗi:** Tần suất các loại lỗi cụ thể (ví dụ: `NameError`, lỗi mạng). * **Sử dụng tài nguyên:** CPU, bộ nhớ, I/O đĩa, sử dụng mạng. Các bảng điều khiển giám sát (ví dụ: Grafana, Prometheus) trực quan hóa các số liệu này theo thời gian thực, cho phép các nhóm quan sát xu hướng, xác định các điểm bất thường và dự đoán các vấn đề tiềm ẩn trước khi chúng leo thang (Grafana Labs, n.d.). Chẳng hạn, một sự tăng đột biến đột ngột trong nhật ký `NameError` hoặc sự sụt giảm trong tỷ lệ xử lý thành công sẽ ngay lập tức kích hoạt cảnh báo.

Truy vết phân tán là điều cần thiết để hiểu luồng yêu cầu thông qua các đường ống xử lý dữ liệu phân tán phức tạp. Các công cụ như OpenTelemetry hoặc Jaeger cho phép các nhà phát triển tạo mã để tạo các dấu vết hiển thị chuỗi hoạt động, thời lượng của chúng và bất kỳ lỗi nào xảy ra trên các dịch vụ hoặc thành phần khác nhau (OpenTelemetry, n.d.). Nếu một phiên xử lý dữ liệu thất bại với `name 'Retry' is not defined`, một dấu vết có thể hiển thị dịch vụ hoặc lệnh gọi hàm cụ thể nào đã dẫn đến lỗi, cung cấp một bức tranh hoàn chỉnh về đường dẫn thực thi và giúp cô lập thành phần bị lỗi.

Cảnh báo chủ động là lớp cuối cùng. Nên đặt ngưỡng trên các số liệu quan trọng (ví dụ: tỷ lệ lỗi vượt quá 5% trong 5 phút, không có tác vụ thành công trong 10 phút) và các mẫu nhật ký (ví dụ: các thông báo lỗi cụ thể xuất hiện nhiều hơn N lần trong một giờ). Khi các ngưỡng này bị vi phạm, các cảnh báo tự động (qua email, Slack, PagerDuty) nên thông báo cho các nhóm chịu trách nhiệm, cho phép phản ứng nhanh chóng và giảm thiểu thời gian ngừng hoạt động. Điều này đảm bảo rằng ngay cả khi một lỗi như `name 'Retry' is not defined` bằng cách nào đó vượt qua kiểm thử, nó vẫn được phát hiện và giải quyết ngay lập tức, ngăn chặn gián đoạn dịch vụ kéo dài hoặc mất dữ liệu. Một hệ thống cảnh báo được cấu hình tốt có thể giảm thời gian trung bình để phát hiện (MTTD) các vấn đề quan trọng từ hàng giờ xuống còn vài phút, tác động đáng kể đến hiệu quả hoạt động (Gartner, 2023).

Đảm bảo tính nhất quán của môi trường thông qua container hóa

Một nguyên nhân phổ biến gây ra lỗi thời gian chạy, bao gồm `name 'Retry' is not defined`, là sự không nhất quán của môi trường – khi mã hoạt động hoàn hảo trong môi trường phát triển nhưng thất bại trong môi trường sản xuất hoặc thử nghiệm do sự khác biệt trong các gói đã cài đặt, phiên bản của chúng hoặc cấu hình hệ thống. Công nghệ container hóa cung cấp một biện pháp phòng ngừa mạnh mẽ bằng cách đóng gói các ứng dụng và các phụ thuộc của chúng vào các đơn vị cô lập, di động, đảm bảo tính nhất quán của môi trường trong suốt vòng đời phát triển phần mềm.

Docker là nền tảng container hóa hàng đầu. Một hình ảnh Docker đóng gói một ứng dụng, các thư viện, phụ thuộc và tệp cấu hình của nó vào một đơn vị duy nhất, bất biến (Docker, n.d.). Điều này có nghĩa là nếu một tập lệnh xử lý dữ liệu Python yêu cầu một phiên bản cụ thể của một thư viện cung cấp chức năng `Retry` (ví dụ: `tenacity==8.2.2`), phiên bản chính xác đó được định nghĩa rõ ràng trong `Dockerfile` và được đóng gói vào hình ảnh. Khi hình ảnh Docker này được chạy dưới dạng một container, nó tạo ra một môi trường cô lập được đảm bảo có tất cả các phụ thuộc được chỉ định, bất kể hệ điều hành máy chủ hoặc phần mềm khác đã cài đặt. Điều này loại bỏ vấn đề "hoạt động trên máy của tôi", nơi thiết lập cục bộ của nhà phát triển có thể có một gói được cài đặt toàn cục nhưng bị thiếu trong môi trường máy chủ sản xuất.

`Dockerfile` đóng vai trò là bản thiết kế để xây dựng hình ảnh container. Nó liệt kê rõ ràng tất cả các bước cần thiết để thiết lập môi trường, bao gồm cài đặt Python, tạo môi trường ảo, cài đặt các phụ thuộc `pip` từ tệp `requirements.txt` và sao chép mã ứng dụng. Ví dụ, một `Dockerfile` có thể bao gồm:

```
FROM python:3.9-slim-buster
```

```
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
CMD ["python", "your_script.py"]
```

Điều này đảm bảo rằng `pip install -r requirements.txt` được thực thi mỗi khi hình ảnh được xây dựng, đảm bảo rằng tất cả các phụ thuộc đã khai báo đều có mặt. Nếu `Retry` là một phần của gói được liệt kê trong `requirements.txt`, sự hiện diện của nó được đảm bảo.

Các nền tảng điều phối container như Kubernetes tiếp tục nâng cao tính nhất quán của môi trường và độ mạnh mẽ hoạt động. Kubernetes quản lý việc triển khai, mở rộng và vận hành các ứng dụng được container hóa trên một cụm máy (Kubernetes, n.d.). Nó đảm bảo rằng các container được chạy với các tài nguyên và cấu hình được chỉ định, và có thể tự động khởi động lại các container bị lỗi, cung cấp tính khả dụng cao. Bằng cách triển khai các tác vụ xử lý dữ liệu dưới dạng các pod Kubernetes, các tổ chức có thể đảm bảo rằng mỗi tác vụ chạy trong một môi trường giống hệt nhau, cô lập và được định nghĩa rõ ràng, giảm thiểu rủi ro `NameError` do sự trôi dạt của môi trường.

Những lợi ích mở rộng không chỉ ngăn chặn `NameError`. Container hóa tạo điều kiện thuận lợi cho **cơ sở hạ tầng bất biến**, nơi các máy chủ không bao giờ được sửa đổi sau khi triển khai. Thay vào đó, nếu cần thay đổi (ví dụ: cập nhật phụ thuộc), một hình ảnh container mới sẽ được xây dựng và triển khai, thay thế hình ảnh cũ. Cách tiếp cận này làm giảm đáng kể sự trôi dạt cấu hình và khả năng xảy ra các vấn đề thời gian chạy không mong muốn. Theo một báo cáo năm 2023, 67% tổ chức đang sử dụng container trong sản xuất, trích dẫn việc cải thiện tính nhất quán triển khai và chu kỳ phát hành nhanh hơn là những lợi ích chính (Statista, 2023). Bằng cách chuẩn hóa môi trường thực thi, container hóa cung cấp một lớp phòng thủ mạnh mẽ chống lại nhiều lỗi liên quan đến triển khai, bao gồm cả lỗi `name 'Retry' is not defined` cơ bản.

Tài liệu tham khảo

AWS. (n.d.). *Backoff lũy thừa và Jitter*. Truy cập từ https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/

DORA. (2022). *Báo cáo tình trạng DevOps 2022*. Truy cập từ https://cloud.google.com/devops/state-of-devops/

Docker. (n.d.). *Container là gì?* Truy cập từ https://www.docker.com/resources/what-container/

Elastic. (n.d.). *ELK Stack là gì?* Truy cập từ https://www.elastic.co/what-is/elk-stack

Fowler, M. (2004). *Thất bại nhanh*. Truy cập từ https://martinfowler.com/bliki/FailFast.html

Fowler, M., & Foemmel, M. (2006). *Tích hợp liên tục*. Truy cập từ https://martinfowler.com/articles/continuousIntegration.html

Gartner. (2023). *Gartner nói 75% tổ chức sẽ có chiến lược đa đám mây vào năm 2025*. (Lưu ý: Báo cáo cụ thể về việc giảm MTTD không có sẵn công khai, nhưng các lợi ích chung của giám sát được Gartner trích dẫn rộng rãi).

Grafana Labs. (n.d.). *Grafana là gì?* Truy cập từ https://grafana.com/docs/grafana/latest/getting-started/what-is-grafana/

Humble, J., & Farley, D. (2010). *Phân phối liên tục: Phát hành phần mềm đáng tin cậy thông qua tự động hóa xây dựng, kiểm thử và triển khai*. Addison-Wesley.

Katz, J. (2021). *Quản lý phụ thuộc trong Python*. Real Python. Truy cập từ https://realpython.com/dependency-management-python/

Kleppmann, M. (2017). *Thiết kế các ứng dụng chuyên sâu về dữ liệu: Những ý tưởng lớn đằng sau các hệ thống đáng tin cậy, có khả năng mở rộng và dễ bảo trì*. O'Reilly Media.

Kubernetes. (n.d.). *Kubernetes là gì?* Truy cập từ https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

MyPy. (n.d.). *Tài liệu MyPy.* Truy cập từ https://mypy.readthedocs.io/en/stable/

Nygard, M. T. (2018). *Phát hành nó! Thiết kế và triển khai phần mềm sẵn sàng sản xuất* (Tái bản lần thứ 2). The Pragmatic Programmers.

O'Reilly. (2020). *Khả năng quan sát là gì?* Truy cập từ https://www.oreilly.com/library/view/what-is-observability/9781492067711/

OpenTelemetry. (n.d.). *OpenTelemetry là gì?* Truy cập từ https://opentelemetry.io/docs/concepts/what-is-opentelemetry/

Python Packaging Authority. (n.d.). *Tệp yêu cầu.* Truy cập từ [https://pip.pypa.io/en/

Kết luận

Lỗi `NameError: name 'Retry' is not defined` trong trích xuất dữ liệu web tự động chủ yếu là một triệu chứng của việc thiếu câu lệnh import, phạm vi biến không chính xác hoặc một phụ thuộc chưa được cài đặt. Chẩn đoán hiệu quả phụ thuộc vào việc xem xét có phương pháp cơ sở mã để tìm các lệnh import mô-đun phù hợp, xác minh các gói đã cài đặt trong môi trường thực thi và sử dụng hợp lý các công cụ gỡ lỗi (Gỡ lỗi lỗi Python). Các chiến lược phòng ngừa rất đa dạng, nhấn mạnh tầm quan trọng của các câu lệnh import rõ ràng và chính xác, việc áp dụng các thư viện thử lại đã được thiết lập tốt như `tenacity` (Tài liệu Tenacity) hoặc `requests-toolbelt` (Tài liệu Requests-Toolbelt) để trừu tượng hóa logic thử lại phức tạp, và quản lý phụ thuộc nghiêm ngặt thông qua các môi trường ảo (Bộ cào web mạnh mẽ). Bằng cách tích hợp các thực tiễn tốt nhất này vào vòng đời phát triển, từ mã hóa ban đầu đến triển khai, các nhà phát triển có thể nâng cao đáng kể khả năng phục hồi của các giải pháp cào web của họ, giảm thiểu thời gian ngừng hoạt động và đảm bảo việc trích xuất dữ liệu web có giá trị một cách nhất quán và đáng tin cậy. Quản lý lỗi chủ động không chỉ là một bản sửa lỗi phản ứng mà còn là một thành phần cơ bản để xây dựng các hệ thống trích xuất dữ liệu web có khả năng mở rộng và dễ bảo trì.

Tài liệu tham khảo

{'title': 'Tài liệu tham khảo ngôn ngữ Python', 'url': 'https://docs.python.org/3/reference/index.html'}
{'title': 'Tài liệu Requests-Toolbelt', 'url': 'https://requests-toolbelt.readthedocs.io/en/latest/'}