# System Design: Confluence Page Verifier

This script complements the find\_public\_pages.py script by automating the audit process. It reads the generated CSV, samples the links, and attempts to access them anonymously to confirm they are public.

## 1. Overview & Architecture

This script will read the public\_pages.csv file, select a random 10% sample of the URLs, and then use a concurrent process (a thread pool) to check each URL.

To meet your requirements, the architecture will be built on:

* **Concurrency:** A ThreadPoolExecutor will be used to make many HTTP requests in parallel, addressing the "relatively quick" requirement.
* **Rate Limiting:** A Semaphore will be used to limit the number of *simultaneously* active requests (e.g., 5-10 at a time). This directly addresses the "not sending large per-second volumes" requirement to avoid being blocked.

## 2. Core Technology Stack

This script will also be a simple Python 3 file, adding libraries for concurrency.

* **Language:** Python 3.x
* **Libraries:**
  + csv: (Built-in) To read the public\_pages.csv file.
  + random: (Built-in) To select the 10% sample.
  + requests: To make HTTP HEAD requests. A HEAD request is much lighter than a GET as it only fetches headers (including the status code) and not the page body.
  + concurrent.futures: (Built-in) To use the ThreadPoolExecutor for managing concurrent tasks.
  + threading: (Built-in) To use a Semaphore for rate limiting.
  + sys: (Built-in) To print progress and a final report.

## 3. Component Design (Script Structure)

### 3.1. Configuration (Global)

Similar to the first script, config variables will be at the top.

Python

# --- CONFIGURATION ---  
# The file to read from (should match OUTPUT\_FILENAME from the first script)  
INPUT\_FILENAME = "public\_pages.csv"  
  
# What percentage of links to sample for verification.  
# 0.1 = 10%, 0.25 = 25%, etc.  
SAMPLE\_PERCENT = 0.1  
  
# How many requests to run at the same time.  
# Keeps us from getting rate-limited or blocked.  
MAX\_CONCURRENT\_REQUESTS = 5  
# --- END CONFIGURATION ---

### 3.2. check\_link Worker Function

This is the core function that will be run in a thread for each sampled URL.

check\_link(url, title, semaphore)

1. **Acquire Semaphore:** The function will first wait to acquire the semaphore (with semaphore:). This blocks the thread until a "slot" is free, enforcing the rate limit.
2. **Make Request:**
   * It will make an anonymous requests.head(url, allow\_redirects=True, timeout=10) call.
   * It's wrapped in a try...except block.
3. **Check Status:**
   * **On Success:** If response.status\_code is 200 (OK), it's a pass.
   * **On Failure:** Any requests.exceptions.RequestException (like a timeout, connection error) or a non-200 status code (like 404 Not Found, 403 Forbidden, 500 Server Error) is a failure.
4. **Return Result:** The function returns a simple tuple, e.g., ("PASS", title, url) or ("FAIL", title, url, "Error: 403 Forbidden").
5. **Release Semaphore:** The with statement automatically releases the semaphore when the function (or an error) completes, allowing another waiting thread to start.

### 3.3. main Orchestrator Function

The main function will:

1. Read all pages from the INPUT\_FILENAME.
2. Calculate the sample size and use random.sample() to get the list of pages to test.
3. Initialize the ThreadPoolExecutor and the Semaphore.
4. Submit all check\_link tasks to the executor.
5. As tasks complete, collect the results into passed\_links and failed\_links lists.
6. Once all tasks are done, print a final, clear summary report to the console, listing any failures in detail.

This design directly meets the change request by being fast (concurrent), safe (rate-limited), and focused on verification (10% sampling).