Automated Asset Extraction and PDF Generation: A Playwright Implementation

Independent Project

Yash Sharma

I. Introduction

1.1 Overview

The core objective of this project was to completely eliminate the severe bottleneck of manual asset extraction and document compilation; specifically, the tedious process of saving hundreds of individual images and formatting them into PDFs issue by issue. Automating this workflow should be straightforward, but standard Python libraries immediately fail against platforms utilizing dynamic JavaScript and aggressive anti-bot security. Because standard headless Chromium instances are instantly fingerprinted and blocked, the primary engineering roadblock was gaining initial access. To bypass this, I independently developed a custom web-to-PDF Playwright script. By injecting stealth evasion techniques and stripping default "automation-controlled" flags, I engineered a persistent, human-mimicking browser session capable of operating under the radar.

Once the stealth environment was stable, the script physically manipulated the viewport to trigger lazy-loaded elements. To maintain chronological data integrity, I implemented algorithmic natural sorting to resolve the file sequence corruption caused by standard OS alphabetical sorting (where "10.jpg" incorrectly precedes "2.jpg"). Ultimately, this framework guarantees accurate data sequencing from extraction to final PDF compilation, reducing a massive manual ordeal into a single command-line execution.

II. MODEL OVERVIEW

2.1 Session Initialization

To successfully operate within a hostile web environment, the workflow begins with configuring a persistent Chromium context via Playwright. By routing the browser session through a localized user_data_dir and injecting playwright-stealth evasion scripts, the system effectively mimics a returning human user. This approach strips default automation flags, bypassing initial bot-detection mechanisms.

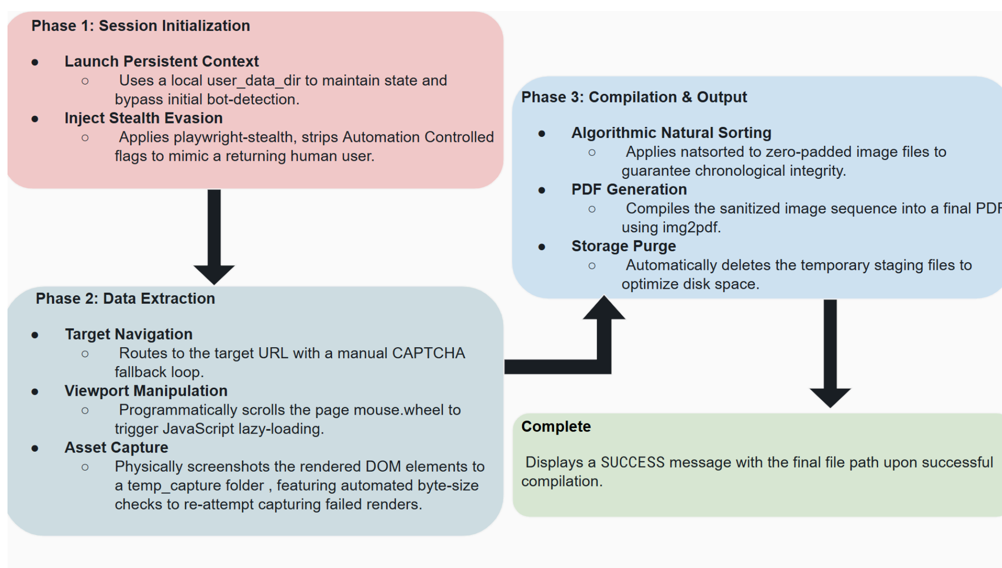

Figure 1 illustrates the overall automated batch extraction process implemented in the Web-to-PDF pipeline.

2.2 Data Extraction & Active Error Handling

Once initial security barriers are cleared, the script routes to the target URL. Crucially, it features an interactive fallback loop (check_for_captcha) to detect Cloudflare security challenges, pausing execution for manual resolution without dropping the browser context. To extract the dynamic payload, the script programmatically simulates human viewport scrolling to trigger lazy-loaded elements. It then physically screenshots the rendered DOM elements to a temporary folder, utilizing an automated byte-size check (verifying payloads are > 1000 bytes) to instantly detect and re-capture incomplete or failed renders.

2.3 Compilation & Storage Optimization

Before compilation, dynamic Regex sanitization is applied to the output filenames, automatically stripping invalid characters to prevent OS-level pathing errors during the final save state. To prevent the chronological sequence corruption inherent to standard OS alphabetical sorting, the script applies algorithmic natural sorting (natsort) across the staged, zero-padded image files. Once the sequence is validated, it compiles the sanitized assets into a final PDF using img2pdf. Following generation, the script automatically deletes the temporary staging files to optimize local disk space.

2.4 Output Validation & Completion

The execution lifecycle concludes with a final state validation check. If valid images are staged, the system compiles the PDF and outputs a SUCCESS message with the destination path. If the extraction fails, it catches the empty state and logs a FAILED error. In both scenarios, the script cleanly terminates the browser context to ensure a graceful exit and release system resources.

III. TECHNICAL ROADBLOCKS AND OPTIMIZATION

3.1 Environment & Network Initialization

Getting the automation off the ground was unexpectedly difficult right from the start. Simply installing the required libraries wasn't enough; the script immediately crashed upon execution because it lacked a dedicated browser engine. I realized the environment required explicit terminal commands to fetch and install isolated browser binaries before the local setup could function. Once the browser was running, I hit another massive snag: the script would freeze indefinitely on the target site. It turned out the site was streaming an endless loop of background trackers and ad payloads. My script was waiting for the network traffic to completely quiet down before proceeding, which was never going to happen. I had to optimize the execution trigger to ignore the background network noise and start scraping the exact moment the visual page structure loaded.

3.2 Payload Routing & Asset Validation

Extracting the actual images required bypassing strict platform constraints. By default, the site forced users to click through pages one by one, which completely broke my automated scrolling logic. To bypass this, I reverse-engineered how the site handled server requests and found I could inject a specific parameter directly into the URL. This forced the server to bypass pagination and dump every image into a single, vertically scrollable layout. From there, I ran into a data validation issue. To catch broken image renders, I built a safeguard that checked file sizes and deleted anything that was too small. However, my original threshold was way too high, meaning the script was aggressively throwing away

perfectly valid, highly compressed pages; which then caused the PDF compiler to crash due to missing files. Carefully lowering that threshold allowed me to catch the genuine render failures without losing optimized data.

3.3 Data Sanitization & State Management

The final major hurdles happened right at the safe state, primarily dealing with local operating system constraints. If a title had characters like a colon or quotation marks, the OS would reject the file path and crash the script right at the finish line. My initial basic find-and-replace wasn't catching everything, so I refactored the naming logic to use a strict regular expression that automatically scrubs any OS-forbidden characters from the title before saving. Finally, I noticed a frustrating bug where if a run was manually interrupted, it would leave orphaned images in the temporary folder. The next time the script ran, it would blindly stitch those old pages into the new PDF, creating cross-contaminated files. I fixed this by fortifying the storage logic to aggressively wipe the temporary directory clean on a strict per-issue basis, guaranteeing a completely sterile workspace for every loop.

IV. RESULTS AND CONCLUSION

4.1 Execution

As captured in the execution log below, the process begins by navigating to the localized working directory using the cd command, followed by initializing the engine via Python Comic_B.py. The script immediately prompts the user to "Paste the series link:" and select a deployment mode. By selecting mode '3', the system is instructed to execute a full batch extraction. While the console displays a clean "Fetching series list” followed by "Ready to process 4 files.", the script is actively establishing a stealth browser context. For each file, it dynamically creates a temporary staging folder, physically extracts and validates the rendered image assets, compiles them into a perfectly sorted PDF, and finally wipes that temporary cache clean before outputting the final "Done, Enjoy reading!" confirmation.

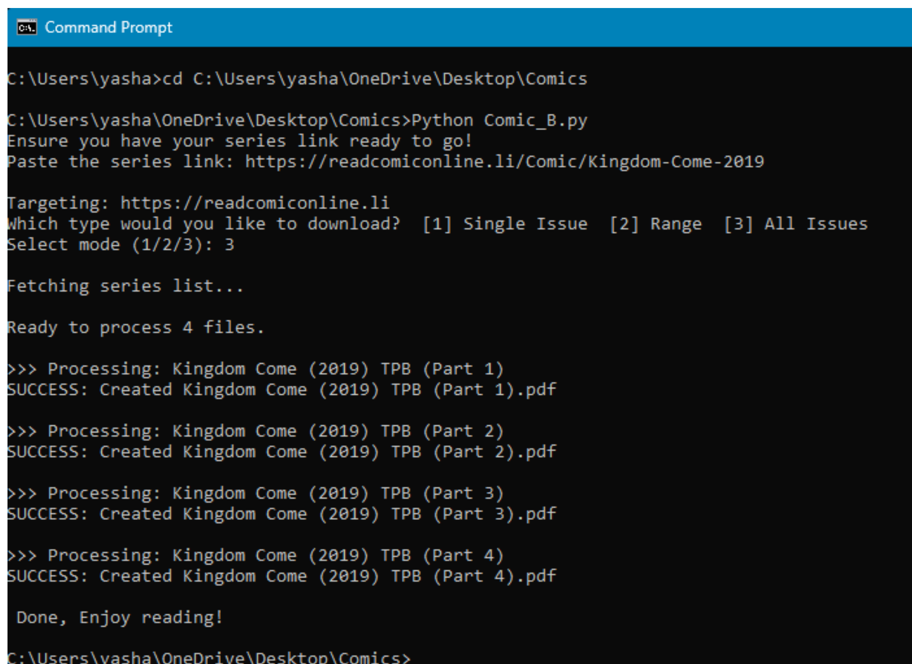

Figure 1 illustrates the overall execution flow structure implemented in the automated extraction engine.

4.2 Conclusion

The core objective of this project was to completely replace a fragile, manual extraction process with a fully autonomous ETL pipeline. By leveraging Playwright for stealth browser automation, img2pdf for document assembly, and natsort for file sequencing, the resulting framework successfully manages the entire extraction lifecycle. Transitioning this from a basic scraper to a production-ready tool required solving concrete environmental constraints, such as bypassing bot detection, sanitizing OS-forbidden characters, and programmatically catching broken image renders. Furthermore, hardcoding strict local storage management to aggressively wipe temporary directories eliminated cache contamination during large batch runs. Ultimately, this project demonstrates that a robust automated tool must explicitly account for real-world edge cases, proactively manage compute resources, and maintain strict data integrity within highly volatile web environments.