National College of Ireland

Methodology for Automated Forensic Web Scraping of Pricing Information

MSc Research Project

Cybersecurity

Peter Byrne

Student ID: X19164131

School of Computing

National College of Ireland

Supervisor: Raza Ul Mustafa

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	PETER BYRNE
Student ID:	X19164131
Programme:	MSc. Cybersecurity **Year:** 2023
Module:	MSc. Research Project
Supervisor:	Raza Ul Mustafa
Submission Due Date:	
Project Title:	Methodology for Automated Forensic Web Scraping of Pricing Information
Word Count:	10673 **Page Count** 34

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: PETER BYRNE

Date: 18/09/23

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Methodology for Automated Forensic Web Scraping of Pricing Information

Peter Byrne

X19164131

Abstract

This research examines current methods around web scraping. The research proposes a Python based solution to carry out automated forensic web scraping with the main objective of scraping e-commerce pricing data. The methodology describes the use of Selenium and Beautiful Soup 4 libraries with MD5 hashing and automation via use of bash script and *'cron'* scheduling in the Linux environment. Six existing Python libraries are extensively tested and compared with each other, using a sample number of websites across local virtual machine, Amazon Web Services (AWS), Microsoft Azure and Linode cloud platforms. The comparison experiment aims to answer the question as to whether there are significant differences in efficacy, amount of data-scraped and time taken, across the different test environments and libraries. The final proposed methodology incorporates two stages; a downloader and a parser, to acquire, store and extrapolate meaningful information from the website data. The methodology uses a supervised syntactic approach from a *JSON* configuration file.

1 Introduction

The objective of this research is to outline an automated forensic methodology for web scraping pricing data from websites using the Python language.

Web sites on the world-wide-web provide an unparalleled opportunity to collect a large amount of real-time data as well as collecting and storing data for historical analysis (Diouf *et al.*, 2019). Data such as pricing information can be collected for analysis by regulatory bodies to indicate activities such as cartel-type behaviour, price-fixing or false advertising among other breaches (Harrington, 2006). The process of collecting this data from web sites is known as *'scraping'* or *'web scraping'*.

The research will first compare and evaluate different approaches of scraping data from web sites. The research will compare existing methods, including a full comparison of six (6) different Python web scraping methods across five (5) measurements in four (4) different test environments using a sample size of fifty (50) web sites. Finally, it will outline a full methodology for carrying out automated web scraping of pricing information in a forensically sound manner based on these conclusions.

Web-scraping broadly speaking consists of a *two-software* program process; *collecting* the data and *analysing* the data to extrapolate meaningful information (Farley and Pierotte, 2017). This methodology will follow these two processes and outline them fully. The research will use three (3) Python scripts to achieve the objective. The scripts will be divided into a comparison script (*sitetest.py*) to evaluate Python methods, a downloader script (*scraper.py*) to collect data and a parser script (*parser.py*) to extrapolate meaningful information.

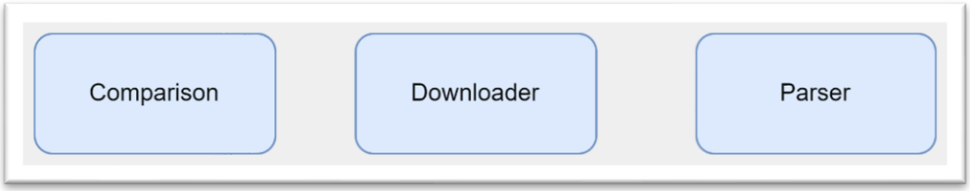

Figure 1

This paper proposes a forensically sound methodology to scrape pricing information from chosen website sources using an MD5 (*messaging digest 5*) hashing algorithm to generate a value when the source content of the page is acquired and stored. The MD5 hashing algorithm is a one-way deterministic mathematical algorithm producing a fixed length output, that is commonly used to verify integrity in computer science (Roussev, 2009).

Automated approaches to web scraping are divided into three approaches; *supervised*, *unsupervised* and *hybrid* (Uzun, 2020). This methodology will use a supervised approach, in that a manual JSON configuration file is required to parse the end results for each site. The reason for this approach is because website architecture can change periodically, and as such code changes may be necessary (Khder, 2021).

This research paper will begin by examining the state-of-the-art in the literature review, in *related work,* discussing the various methods currently in use. The *research methodology* will outline the parameters of the research experiment comparing the efficacy, time and amount of data scraped by six (6) existing Python methods across four (4) different test environments. The *design specification* will outline the process flows of the comparison script, downloader script and parser script. *Implementation* will show the outputs and operation of the final web scraper downloader and parser. The *evaluation* section will show the results of the comparison experiment and evaluate the results of the final web scraper. Finally, there will be a *conclusion* and suggestions for future work in the area.

2 Related Work

The importance of web scraping for data for the purpose of price analysis is widely acknowledged in academic papers (Upadhyay *et al.*, 2017).

There are many suggested tools and methods suggested to achieve the goals, but broadly speaking they are divided into the following methodologies: the use of *browser extensions*, the

use of *third-party tools or services* or the use of *Python scripting* (Khder, 2021). The methodologies can then comprise of '*web crawling'*, which involves '*spidering'* or accessing all or many of the web pages present on a web site, or a targeted acquisition using a DOM (document object model).

2.1 DOM Scraping

Targeted proposals for web scraping in the literature do not consider the forensic element of the acquisition and are concerned only with access to the data. Examples of this can be seen in proposals for DOM scraping (De S Sirisuriya, 2015). This refers to the Document Object Model, which is defined by Mozilla as *"... the data representation of the objects that comprise the structure and content of a document on the web"*[1]. Methods proposing use of DOM scraping usually propose that the only data collected are the required data fields from the website source. This means the processing and analysis of the data is done during the acquisition phase. This method is very efficient in terms of data storage, but there are some problems with the method. There is no forensic element inherent in the method, so any data obtained is unverifiable and must be taken at face value. Another problem with acquisition-stage DOM scraping is that if there is an issue with the acquisition, there is no option to reprocess. The data acquired during the scrape is final. The methodology proposed in this paper will keep a copy of the webpage HTML, and this data can be parsed or analysed at any stage and in different ways but primarily using a post-acquisition DOM method to parse. The Beautiful Soup library is a commonly recommended and supported Python library which performs DOM parsing (Zheng, He and Peng, 2015).

2.2 Forensic Acquisition

The lack of a forensic component during the acquisition phase is a serious consideration for pricing information from a regulatory or criminal competition angle. Potential competition offences such as price signalling or cartel behaviour can be shown with multi-variant analysis which includes pricing information (Harrington, 2006). In such a case, in order to satisfy the burden of proof, some evidence of the forensic acquisition of the pricing information may be required. There would be issues with presenting scraped data without a forensically acquired copy of the HTML code with an accompanying MD5 hash value (Roussev, 2009). Most papers describing web scraping fail to consider a forensic component to the acquisition, however there has been strong work done on forensic acquisition and analysis of web pages which are directly applicable to an automated large web scraping approach to analyse pricing information (Vidya, Saly and Balan, 2022).

[1] https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

2.3 Browser Extensions

Proposals for scraping web pages using *web browser extensions* are easy to implement, however they usually do not have robust automation or scalability capabilities, so are not as effective as the proposed methodology of this paper. The extension is maintained and developed by a third party, and over time some extensions are no longer supported. In a 2019 paper *'Web Scraping: State-of-the-Art and Areas of Application'* (Diouf *et al.*, 2019), a number of web browser extensions are listed:

Spider can no longer be found as an extension, so could not be evaluated. It is described in the paper as showing a list of elements for which data can be retrieved for. This would mean that it was a manual DOM scraper, and as such would be time-consuming and unsuitable for automation or forensic acquisition.

Data Scraper[2] is an extension that requires subscription or allows 500 pages to be scraped per month. The user interface is very straightforward and there are multiple different scripts and customisation options available. It is a manual process however and is a DOM scraper that saves the target to a CSV or Excel spreadsheet file.

Agenty[3] is a visual DOM selector for scraping element data from a web page, which is easy to use, but again is a DOM scraper which is geared towards targeted scraping. There is an option for downloading full web page code, however this option uses the *'curl'* method which does not execute JavaScript code on the page, and as a result may not acquire the full content[4]. The proposed methodology in this research uses a method that can process and interpret JavaScript.

Data Miner and *Cloump U-Scraper Plugin* for Chrome were no longer available. This demonstrates a weakness with using browser extensions which can lose support from developers, and which tend to utilise the DOM model for targeted acquisition of data.

2.4 Python Scripting

The Python scripting method can be broken down into the use of several different methodologies and libraries which are used when the approach is to develop a web scraping application (Milev, 2017). Some websites provide an application programming interface (API) which provides direct access to structured data (Lawson, 2015), however most websites do not provide an API, and scraping data via an API does not address the digital forensics aspect of acquiring forensically sound data. Similarly, methodologies which are focused on speed and efficiency target only the relevant or required data and eliminate or omit the extraneous data (Uzun, 2020). A number of libraries are available in Python which can acquire the complete

[2] https://dataminer.io/
[3] https://agenty.com/
[4] https://everything.curl.dev/http/post/javascript

web page source code. Some libraries such as the '*Requests'* library or the *'Urllib'* library will make *HTTP* (hyper-text transfer protocol) requests and obtain the *HTML* (hyper-text markup language) code of the page (Chapagain, 2019). This can be problematic when web page content can be generated by *JavaScript* running on the page. Having decided that an automated and forensically-sound web scraping methodology is currently best achieved using Python scripting, the next challenges to address are issues like dynamic content loaded via Javascript, pagination, cyber-security applications including heuristic based detection to prevent automated access and cookies (Upadhyay *et al.*, 2017). For this reason, if choosing Python scripting for the web scraping methodology a library that processes and interprets JavaScript will yield better results. Newer libraries, such as the *Selenium* library, which is chosen for the recommended methodology, utilise actual web browsers to access and scrape the data. This allows Selenium to use a 'webdriver' to interact with a web-browser in order to process and interpret JavaScript and return a better amount of data from the page (Nyamathulla, Ratnababu and Shaik, 2021).

There are third-party service companies which offer web scraping, however the issues arising from these are that lack of forensic process (DOM model again), chain of custody (Prayudi and Sn, 2015), potential loss of confidentiality and cost. In the case of regulatory or law enforcement actions in court, presenting evidence provided from a third-party source which may be located outside of the jurisdiction can be problematic and may be disputed without a physical witness from the third-party company to give evidence as to how the data was obtained.

To summarise, the state-of-the-art in relation to Web Scraping focusses heavily on DOM scraping of targeted data. This approach is a data-mining approach and does not lend itself to the forensic method which may be required for regulatory or law enforcement proofs. The proposed methodology of this paper outlines a method for forensic acquisition of entire web pages, along with screenshots and parsing of the acquired data using the DOM model.

3 Research Methodology

The research methodology was both *comparative* and *experimental*. In evaluating current methodologies for scraping website information using Python, a comparison script *sitetest.py* was developed which identified six (6) commonly used and recommended libraries for web scraping in Python. Each of these was written into a separate Python method and the comparison script carried out five (5) measurements of a sample of fifty (50) websites. The sample was comprised of the top 50 websites in Ireland, as listed by *'Similarweb'*, an industry standard website metric service. One of the entries in the sample set was no longer a valid URL and was therefore of no value to the test. This entry was replaced by a top Irish retailer. Ranking by web traffic was not a requirement for the research, as any random sample set of websites could have been used. The comparison script consisted of a python method for each of the following:

- Wget
- URLLib
- Requests
- Selenium
- Pyppeteer
- Python HTTPLib2

Each method was passed the same inputs and was run in an asynchronous mode to ensure that the parameters for each method were as equal as possible. Each method returned three metrics; the *efficacy* of the method measured by size of downloaded web page, the *time taken* to scrape the page, and the *size* of the returned content.

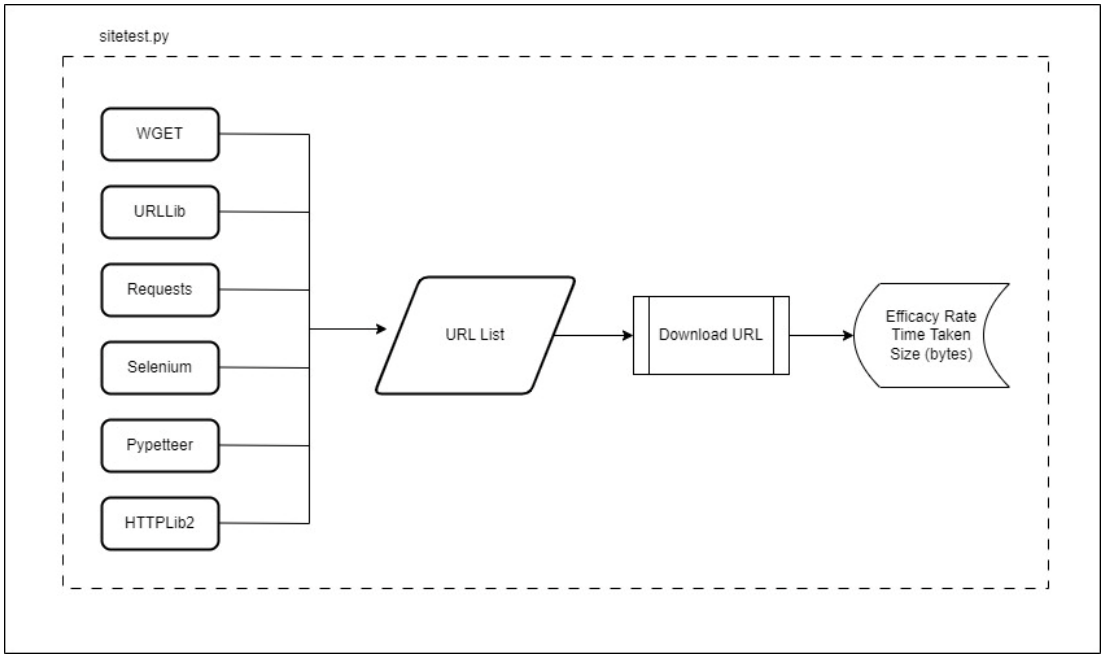

Figure 2

In the comparison script, each method starts a timer when called, and has the three (3) dictionaries initialised to store the metrics for each web site. The websites in the list are then iterated through sequentially and downloaded where possible. This ensures that testing is as equal as possible. The output of the script then shows the efficacy rate for each method, the time taken for each method, the individual URLs with a *PASS* or *FAIL*, along with the *size* of the retrieved website source code. The comparison script was run across four (4) identically configured test environments on a schedule at the same time over a period of five (5) days. The results of each day were averaged to account for any anomalies caused by extrinsic factors, and to provide a more accurate result than a single sample. Increasing the sample size from fifty (50) and the number of measurements from five (5) could be considered to provide a more reliable result, however with the time constraints of this paper it was felt that the sample size and number of measurements was sufficient to give a good overview in order to make the decision on which method to use for the proposed web scraping methodology.

3.1 Control Group

A failure of efficacy on a given sample or samples would be expected to lower the *size* and *time* metrics. As the efficacy rate was a variable which could influence other variables, a control group of twenty (20) websites was established. This group showed 100% efficacy across all test environments and methods, in other words the variance of efficacy was removed. This allowed direct comparison between the methods and between the test environments for average amount of data downloaded and time taken. The control group results were presented alongside the main sample group for cleaner results and to compare against the 'real-world' scenario tested with the larger sample size.

3.2 Measuring Efficacy

The hypertext transfer protocol (HTTP) is an application layer protocol which web browsers use to request and render web pages under the TCP/IP networking protocol. Requests are made through HTTP and response codes can be returned based on the results. A successful request for a web page normally returns a HTTP code of 200 (Fielding *et al.,* 1999).

Initially the HTTP response code 200 rate of test sites was used as a metric to measure efficacy of each method, but this was found to be unreliable. In certain circumstances, a third-party web service such as *CloudFlare* would return a HTTP response code of 200 but return a small *CloudFlare* holding page instead of the actual web page. The purpose of this is to mitigate against automated attacks, such as *DDoS* (distributed denial of service attacks) on the web site. Measuring efficacy in terms of HTTP 200 in such a case would result in a false positive. Another issue encountered with using the response code as a metric was that the Selenium method currently does not have a method to obtain a HTTP response code. As a result, to measure the rate of efficacy of each method, the sizes of the webpages obtained were used. The threshold for this test was 1000 bytes of size for content. There are limitations with using this as an indicator of success, and the different methods can return different sized code for the website, depending on methodology. Some methods return only the bare XML/HTML code while others execute scripts on the web page and return additional page source code from these scripts.

As the methods are never exactly like for like in terms of operation, the evaluation only included a fast way to screen out non-HTTP 200 returns and CloudFlare pages. More analysis could be done to ensure a more accurate efficacy return by analysing the downloaded pages to ensure they are sufficient. A way of doing this could be to get the size of the page source code manually and compare to the returned results. In order to mitigate against making a false determination based on the efficacy, I also measured the sizes retrieved from each page. The sizes were averaged for each method under each test environment, and this allows us to make a determination based on which method obtains the most data, with the method returning the most data on average being more effective than the method which returned less data.

The timeout supplied to each of the testing algorithms had an impact on the efficacy rate, with a timeout of 30s supplied on each method. This allowed for the script to open and render the pages as well as downloading the data. Initially a lower timeout was used on all algorithms with lower efficacy noted. In the final web scraper proposal, a 30 second timeout will be recommended and used as a result of this testing.

3.3 Measuring Time Taken

When each method is called in the evaluation script, a timer is started which completes at the end of the iteration of samples for that method. This will provide an accurate record of the time taken for each method. It should be noted that there can be cases where the efficacy can be lower, and this can skew the timers positively or negatively depending on the type of failure preventing the page from being downloaded. For example, a HTTP 404 error or a HTTP 200 CloudFlare type small page return could result in a lower recorded time for that page. Conversely, a HTTP 302 redirect error may result in a longer recorded time. As such the timers can only be directly compared in cases where both methods have the same efficacy rate. In cases where the efficacy rate is different, for the aforementioned reasons, the timers can be used as a guide only.

A more scientific evaluation of time taken is carried out and presented using the control group. The purpose of this control group was to remove the effect of variability of efficacy on the results. The average amount of data downloaded for this sample would also have to be considered along with the average time taken measurement as the average size of data result shows a strong positive correlation (+0.68) with the average time taken. As such both these metrics will be presented together.

A potential improvement to the measurements could be to run each method separately and concurrently. This ensures that the variability of time and changing environment are mitigated even further but would mean that six (6) different methods would have to be run at pre-determined initialisation times across each test environment for the number of measurements required (5). As it stands, the tests are run *sequentially*, so the variable of time for each method will affect the start time of the next method. This essentially means that method 2 to method *n* will be evaluated on each test environment at slightly different times depending on the amount of time each method takes to complete on each test environment. I would expect the differences caused by this to be very negligible, however the effect should be noted due to the internet being a test environment in flux.

3.4 Measuring Amount of Data Scraped

The comparison script logged the size of each web page that was scraped in the experiment. The different methods used different methodologies to download the web page source, some making simple HTTP requests for HTML code, and others rendering the page and processing

and interpreting JavaScript into the downloaded page source. To account for this behaviour the acquired webpage size was obtained and used in conjunction with rate of efficacy. The size of all of the pages obtained by method and in each test environment was averaged over each measurement. The interpretation of the average amount of data scraped by method was that more data was better than less data, as it meant a more complete amount of the page data was scraped. This was also run across the group of twenty (20) 100% efficacy websites, so that the data amount was not influenced by the rate of efficacy. Only web page source code was downloaded and measured, and did not include any graphical elements.

3.5 Test Environment

The test environment used was a Ubuntu Server 22.04-2 LTS environment, configured with 1 processor and 2gb of RAM. All other test environments were a type-1 hypervisor on the cloud platforms and Proxmox Virtualisation Server (v8.03).

An issue presented itself when evaluating the experiment results from the Local Virtual Machine, in that there were anomalous failures found in efficacy across the control group where 100% efficacy should be expected. This issue seems to have been caused by running the Local VM as a type-2 hypervisor, and when the experiment was re-done, using a type-1 hypervisor the data was as expected. This demonstrates the importance of using Type-1 hypervisors for virtualisation experiments.

The next stage of the evaluation was to measure the speed and efficacy using the same script across three different cloud service providers. The three (3) providers used for testing were Amazon AWS, Linode and Microsoft Azure. All three systems were configured in the same way as the local virtual machine test environment, using the same operating system, software versions and allocated resources. The purpose of the cloud test environments was to evaluate whether there were significant changes to efficacy or time taken using the methods, since current literature often recommends use of cloud environments due to ease of use, high availability and scalability of resources.

The *comparison* script, *downloader* script and *parser* scripts were programmed using Python 3.6 and designed to run in a *Ubuntu 22.04-2 Server* environment. The web scraper can be converted to run in a Windows environment; however, Ubuntu Linux was chosen as part of the recommended methodology because it is an open-source operating system with proven reliability and security. The environment also includes supporting BASH scripts to run the web scraping python script and to carry out status checks. In built Linux *'cron jobs'* are used to schedule the web scraping, which was configured to run at the same time each day across each platform.

During testing, the most important consideration for the effective use of the scraper was RAM. This is because the Selenium library uses a *webdriver* to essentially run a browser and open the target web page, which can require more memory. The web scraper makes use of

parallelisation and benefits from multiple processors or cores, but again this can result in requiring more RAM due to opening multiple pages simultaneously. The web pages are opened only while they are being acquired, so the requirements are still quite low. A single scraping project can be done using 2gb of RAM, however 4gb would be recommended with a quad core processor. In the event that useable RAM is fully utilised then the scraper environment can be configured to use a swap file which would result in slower performance. This was an extra variable affecting results, and as such swap files were not used and the testing never utilised all of the 2gb of RAM provided.

The internet is not a sterile lab environment and at various stages in testing, slightly different results could be observed at different times. This can be caused by website maintenance or infrastructural issues. For this reason an average for all values over the five (5) measurements was calculated and used. An average of ICMP ping was recorded and calculated to give an indication of network latency in the test environments when the measurements were taken.

The ping tests measuring network latency relate to network infrastructure provisioning and bandwidth fluctuations which were extrinsic and outside of the control of the experiment.

3.6 Browser User Agent Used

Web browsers send a *'user agent'* to the webpage as part of the HTTP request. This user agent is basically version information of the browser making the request. The user agent used for both the method evaluation and the final web scraper was :

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/114.0.0.0 Safari/537.36
```

This was the most popular user agent at the time of the test, with the Chrome web browser being the most popular web browser in use globally[5]. The webdriver used by the Selenium method was Chrome version 114.0.5735.90.

3.7 Web Scraper Downloader

In order to select a web site for testing of the web scraper, I checked the *'robots.txt'* for an Irish retailer. The *robots.txt* is a configuration file used by web sites to prevent automated *'bots'* from indexing the web site using *web crawling* or *spidering* techniques (Jha *et al.*, 2014). The proposed methodology does not use a web crawling technique, as it simply visits a single page and takes a copy of the source code along with a screenshot. As the site would be scraped for a prolonged period, I checked the robots.txt and selected one that allowed indexing. I used a browser extension to check the robots.txt, although the comparison script also contains a method which was coded to check the robots.txt of sites and display them.

[5] https://gs.statcounter.com/browser-market-share

The web scraper downloader was used to test functionality on this site using a test environment of Ubuntu Server 22.04-2 with 4gb of RAM and 1 x 4 Core Processor. Outputs from the downloader were generated by running the parser across the downloaded pages and outputting to a TSV (tab-separated value) file.

4 Design Specification

The research evaluated six different methods to download web pages. Based on the findings of this research a web scraper solution was developed. All scripts were written in Python 3.6, and there were supporting BASH scripts developed to run and maintain the web scraper.

The web scraper solution is comprised of two main components, a downloader and a parser. The downloader requires an input list of URLS, and the parser uses data structures contained in a JSON configuration file to parse the data. The solution is modular in design as much as possible to allow for easier development.

4.1 Comparison Script Design

The comparison script is called *sitetest.py* and requires an input text file containing a single URL without qualifier per line. When the script is called, it will check the length of the arguments passed to it, and if they are not correct the usage of the script is displayed. The next step is to check that the input file is present, and then the file is read into a list and sorted alphabetically. The next step is to set the web browser user agent which is used when making requests for web pages. The initialisation phase of the script is :

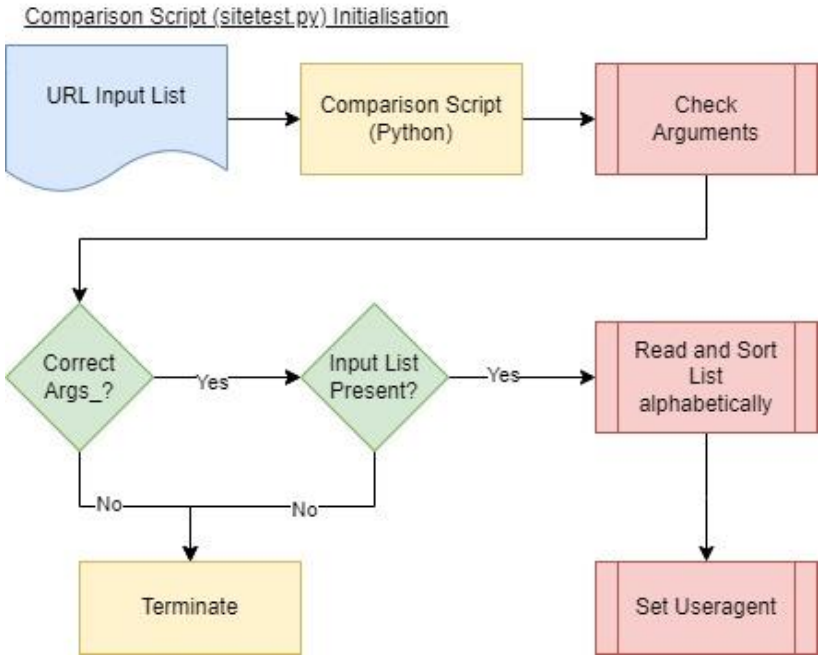

Figure 3

Once initialised, each of the six (6) methods for downloading the page is called sequentially. The parameters passed to each method and the outputs taken from each method are the same. Each method is passed the input list of URLs and the user agent. Each method then returns a results dictionary which contains the *response* (efficacy) measured as a PASS or FAIL, the *content* itself, the *size* of the content for each URL in the list. It also returns the time taken for the method to complete.

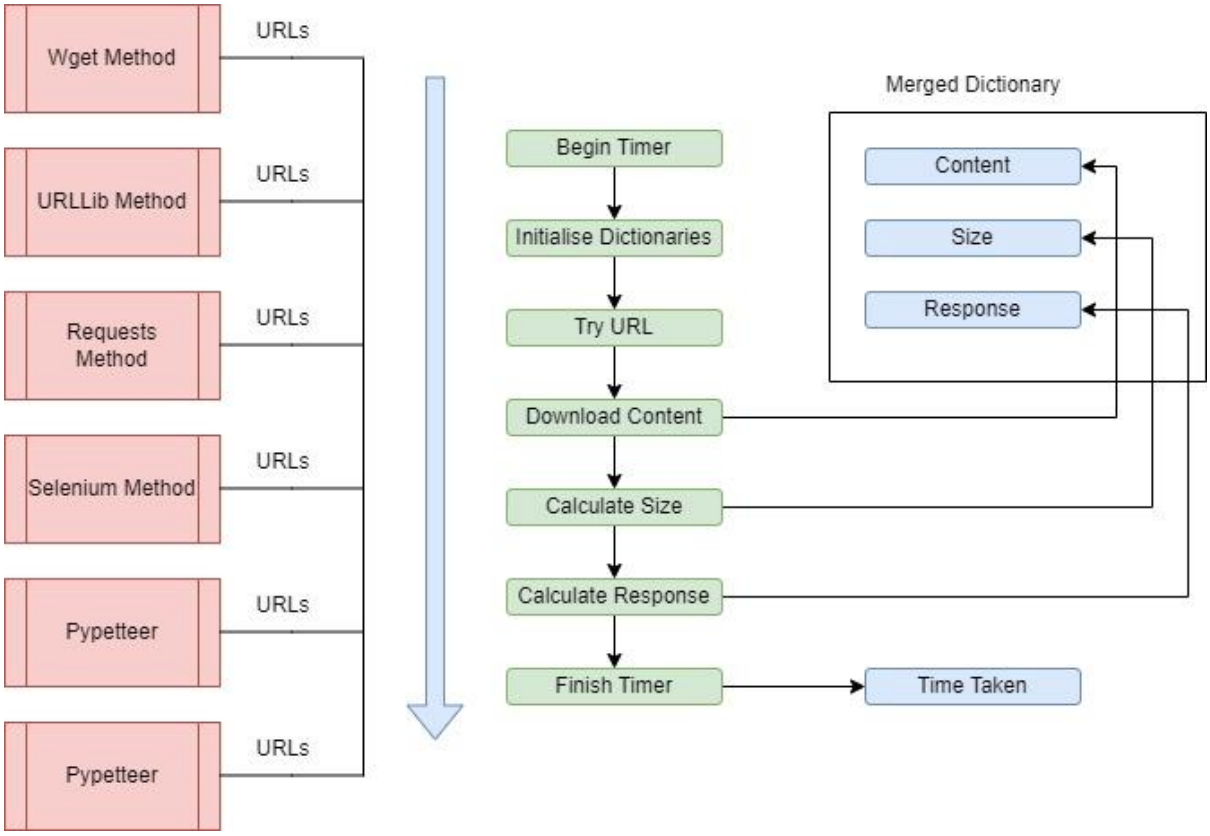

Figure 4

4.2 Comparison Script Libraries

A full list of versions used for Python and dependent libraries is generated in a directory containing the comparison python script. The methodology to generate the list of required dependencies was to use the following pip3 command:

```
pipreqs --force
```

This command generates a file named *'requirements.txt'* which contains the following list of libraries and versions required for the comparison script (sitetest.py) to function:

```
httplib2==0.20.2
pyppeteer==1.0.2
pyppeteer_stealth==2.7.4
```

```
Requests==2.31.0
requests_html==0.10.0
selenium==4.10.0
wget==3.2
```

These libraries can be installed in the test environment by running:

```
pip3 install -r requirements.txt
```

To automate setting up a test environment for the comparison script, a small installation bash script was run. This bash script contained the following:

```
export PATH=$PATH:~/.local/bin

sudo apt install python3-pip
pip3 install pipreqs
pipreqs --force
pip3 install -r requirements.txt
sudo apt-get install -y libgbm-dev
sudo apt install -y gconf-service libasound2 libatk1.0-0 libc6
libcairo2 libcups2 libdbus-1-3 libexpat1 libfontconfig1
libgcc1 libgconf-2-4 libgdk-pixbuf2.0-0 libglib2.0-0 libgtk-3-
0 libnspr4 libpango-1.0-0 libpangocairo-1.0-0 libstdc++6
libx11-6 libx11-xcb1 libxcb1 libxcomposite1 libxcursor1
libxdamage1 libxext6 libxfixes3 libxi6 libxrandr2 libxrender1
libxss1 libxtst6 ca-certificates fonts-liberation
libappindicator1 libnss3 lsb-release xdg-utils wget

wget https://dl.google.com/linux/direct/google-chrome-
stable_current_amd64.deb
sudo dpkg -i google-chrome-stable_current_amd64.deb
sudo apt -f install
```

This script basically installs required python libraries, operating system libraries and downloaded and installed the latest Google Chrome browser.

4.3 Qualitative Considerations

There are other considerations in relation to choosing a web scraping methodology apart from the quantitative metrics uncovered in the research. The features of the libraries and support available also have a qualitative weight.

The qualitative considerations of the web scraping methodologies can be seen in two of the available methods; namely the Selenium and the Pypetteer methods. Both methods are more

difficult to detect as originating from an artificial or automated environment. This can be observed from the efficacy rates and is a result of the methodology of using real web browsers in order to make the HTTP request and access the web page. This is the most future proof method for ensuring high efficacy of web scraping in the face of increasing cybersecurity and anti-scraping technologies being deployed to prevent automated scanning and attacks on websites. These methods also allow passing of extra testing parameters and functionality such as capturing screenshots of websites and interacting with user inputs or buttons on the site.

The ability to screenshot a web page can be useful to lend weight to the forensic methodology of hashing the website source code. Having the ability to run and scrape JavaScript code on the page is essential to obtaining the relevant data. Failure to obtain the JavaScript code could result in failure to obtain data such as pricing information on the page. Cross browser compatibility refers to the ability of the method to use different browsers such as Firefox. Browser automation is another important feature and refers to the ability of the method to interact with the page by activating and interacting with controls on the page.

The following is a brief list of qualitative considerations for each method:

Table 1

Method	Screenshot	Javascript Code	Cross Browser Compatibility	Browser Automation
Wget	No	No	No	No
URLlib	No	No	No	No
Requests	No	No	No	No
Selenium	Yes	Yes	Yes	Yes
Pyppeteer	Yes	Yes	No	Yes
HTTPlib2	No	No	No	No

The newer generation methods of Selenium and Pyppeteer have clear advantages over the older HTTP request methods. The cost of the added functionality is more memory and processor usage and longer time to scrape each page. Selenium, however, has more support and a broader user base, due to the power of its features and the fact that Pyppeteer is an unofficial Python port of a JavaScript browser automation tool called *'Puppeteer'*.

4.4 Hashing Algorithm

Messaging Digest-5 (MD5) is a 128-bit hashing algorithm that is commonly used in computing and digital forensics to verify integrity of a file. A hashing algorithm is a one-way deterministic mathematical algorithm with a fixed length output. A hashing algorithm incorporates an *'avalanche-effect'* whereby any change to the input will result in a vastly different fixed-length output. In this way they can prove that no binary-digit (bit) in a digital file has been changed. The MD5 hashing algorithm is used to ensure integrity and prove forensic acquisition for this

web scraping methodology. The *hashlib* library for Python was used in the web scraper, and the resulting hashes were cross checked and verified with *md5sum* in Linux.

4.5 Downloader Libraries

The web scraper downloader for the proposed solution uses the following libraries:

```
pyppeteer==1.0.2
pyppeteer_stealth==2.7.4
Requests==2.31.0
requests_html==0.10.0
selenium==4.10.0
undetected_chromedriver==3.4.7
webdriver_manager==4.0.0
```

4.6 Downloader Design

The downloader takes the input URL and creates a unique MD5 hash value to refer to it. The downloader stores the table of URLs and MD5 values in a file called *cipher.csv* which it can use to reverse the MD5 to the original URL. The original URL is not used as a reference as it can be too long in length and can also contain special characters. The parser file can use the *cipher.csv* file to reverse the URLs back to original if required. Here is an example of the structure of the cipher.csv. It consists of only two (2) tab separated columns, the first containing the MD5 hash value and the second is the original URL.

1cd2eb121c22e79120d6443b682c484c	http://www.theguardian.com
1a38b3c506aeaf5b339446931cb47837	http://www.thejournal.ie
9c0bebca0d64c1a1402b4b70837a1597	http://www.thesun.ie
dc208907834bbe9deb5fec6636865202	http://www.three.ie
b02030b4551d928416b039ee1b0e22cd	http://www.tiktok.com
a4a5e06d210c89df6d45e81caed26d4f	http://www.twitch.tv
54872245d63c76f4707418790fc8ea71	http://www.twitter.com

Figure 5

Here is a process flow for the initialisation of the downloader script (*scraper.py*).

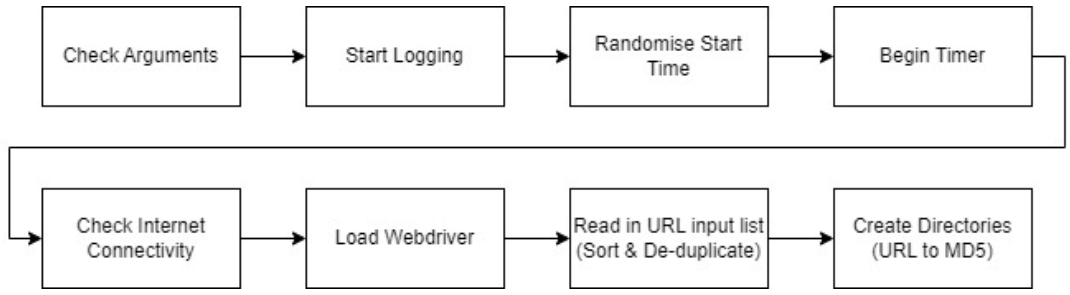

Figure 6

First the script will check the arguments are correct, the script takes an input list of URLs, in the same way as the comparison script. The downloader will then start logging of all activity, which is essential to check that the scrape has been successful and create alerts for unsuccessful attempts. The script uses a start time randomiser so that the scrape is not carried out at the same time every day. This can be customised to be random within a range of values. Then a timer is started to keep track of how long the scraping should take.

Next the downloader will check internet connectivity and log or alert if there is no connection. The webdriver version is checked or updated and then loaded to enable the Selenium library to carry out activity on the target web site. This is an important step, as the user agent and other parameters are passed to the webdriver. The following shows the options added to the web driver to maximise the chances of successful web scraping.

```
chrome_options = Options()
chrome_options.binary_location = '/usr/bin/google-chrome'
chrome_options.add_argument("--headless")
chrome_options.add_argument("--no-sandbox")
chrome_options.add_argument("--incognito")
chrome_options.add_argument("--disable-setuid-sandbox")

chrome_options.add_argument("--remote-debugging-port=9222")  # this

chrome_options.add_argument("--disable-dev-shm-usage")
chrome_options.add_argument("--disable-software-rasterizer")
chrome_options.add_argument("--disable-web-security")  # Disable web security
chrome_options.add_argument("--disable-xss-auditor")  # Disable XSS auditor

chrome_options.add_argument("--disable-extensions")
chrome_options.add_argument("--disable-gpu")
chrome_options.add_argument("--start-maximized")
chrome_options.add_argument("--disable-infobars")
chrome_options.add_argument("--profile-directory=Default")
chrome_options.add_argument("--user-data-dir=/home/peter/.config/google-chrome")

user_agent = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.0.0 Safari/537.36"

chrome_options.add_argument(f"user-agent={user_agent}")  # Set the user agent
```

Figure 7

The parameters that are passed to the browser, via the use of *chrome_options.add_argument()* commands can be easily tailored and customised for use with Selenium. This is a very powerful feature as it allows the browser to be changed and even browser profiles to be created to ensure that the target web site does not block the scrape. An explanation of each of the features shown above is beyond the scope of this section, but they are designed to make the browser appear more natural and human. A scrape of the site *'bot.sannysoft.com'* will return a list of browser properties passed to a website in order to profile the browser through browser fingerprinting or identify bots. The list of arguments added in the above figure are used as during the experimental stage of development they were found to have high efficacy.

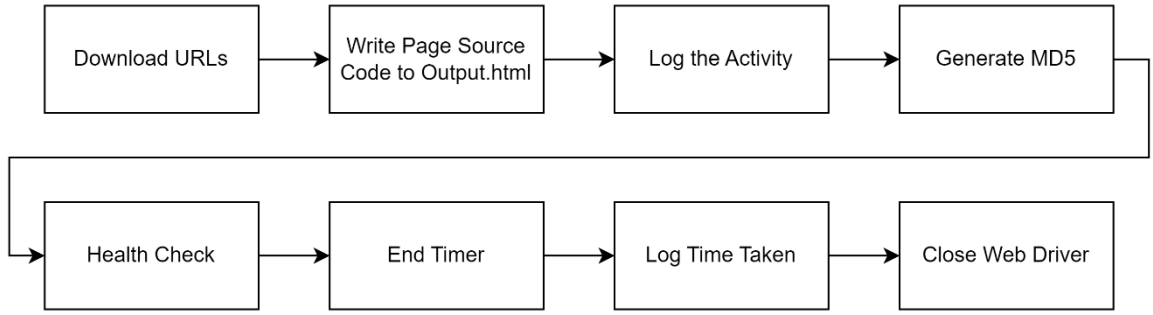

Figure 8

Once the downloader has finished initialising, the second phase commences. The script first checks if there is a downloaded file called *'output.html'* present for the target URL. If the file is not present, then it will attempt to download the target. The reason for this step is that it allows efficient use of more than one iteration of the script. It is recommended to run the scraping script three (3) times each day, by scheduling a cron-job, to mitigate any failures that may occur such as the site being offline or the internet connection having a problem at the time. The second and third iterations of the script will only look to acquire those URLs that have not already been acquired, which is more efficient and less intrusive on the web sites.

The downloader uses the Selenium library and the webdriver to download the source code of the target web site. The page is rendered using the web driver for Chrome and the content is written to *output.html* in the correct folder and subfolder. A graphical screenshot of the page is also taken at the time and then all cookies are cleared. The activity is entered into the log files. Next an MD5 value is generated for the *output.html* file and saved as *'hash.txt'* in the same folder as the output.html file. Time of acquisition is not added as this will be present in the file creation date and time metadata of the *output.html* file.

Having attempted to acquire the content of the target URL, a health check is done to check recursively that an *output.html* file exists for each URL in the list. If a file exists then it is logged as a PASS, if the file is not present it is logged as a FAIL. A failure could occur when a web page is no longer online or when the access is blocked. The contents of the health check are logged, this is essential for determining success of the operation each day.

4.7 Parser Libraries

The parsing script for the proposed solution requires only the Beautiful Soup 4 library:

```
beautifulsoup4==4.11.2
```

4.8 Parser Design

The parser design process flow is as follows:

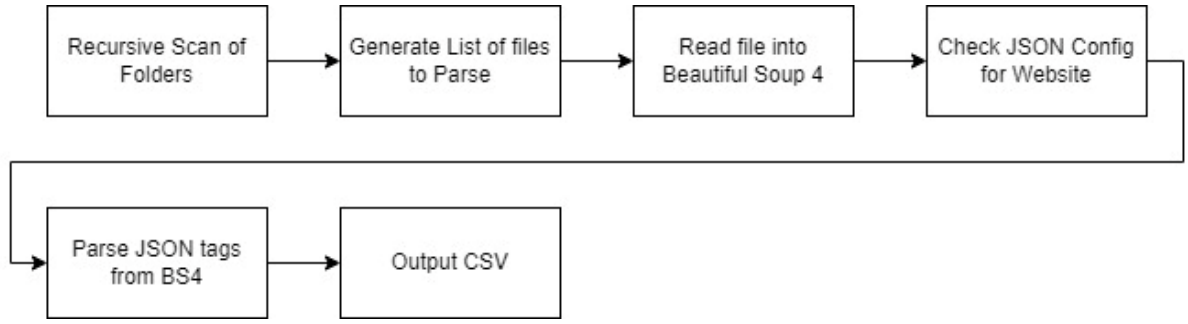

Figure 9

The parser first carries out a recursive scan of the folders and subfolders, looking for the presence of the *output.html* files to parse. Once an *output.html* file is found it is added to a list. The parser then reads the *output.html* files from the list and uses the Beautiful Soup 4 library to create a dictionary of the tags and data found in the *output.html* file. Next the parser reads the JSON configuration file containing the tag entries for each root URL address. If the root URL is found in the JSON then the parser parses the data from tags in the JSON record and outputs the information onto the standard output. Outputting to the standard output (STDOUT) allows for greater flexibility in Linux for redirecting to a log file or carrying out user operations on the data before redirecting (eg. Grep, Sed, Awk commands). The output to STDOUT is by default a TSV file separated by tabs. The design of the TSV by tabs allows easy importing into data programs like Excel and R Studio.

5 Implementation

A BASH script can be used to install all the required dependencies onto the Linux virtual machine, this includes installing the python libraries required.

Evaluating the different methods was carried out by means of using a *'cron'* scheduler built into the Linux operating system to run an executable bash script containing the following commands:

```
#!/bin/bash

# Get the current date
date=$(date +%Y-%m-%d)

# Run the Python script and redirect output to a logfile
cd ~
```

```
python3 sitetest.py top50.txt > "${date}_logfile.txt"
python3 sitetest.py top20.txt > "${date}_top20logfile.txt"
ping -c 100 8.8.8.8 > "${date}_ping.txt"

rm -r *wget*
```

This bash script would get the current date and use it to name the logfiles needed. Metrics are obtained via log files output from the Python 3.6 comparison script (sitetest.py). There are three (3) logfiles obtained, a logfile of URLs with a sample size of 50, a logfile of a sample size of 20 used for the control group and a logfile with ping results. Design and use of the script in this form would make it easy to configure for a larger number of measurements or platforms. The final line of the bash script is for housekeeping and removing of the files saved by the Wget method.

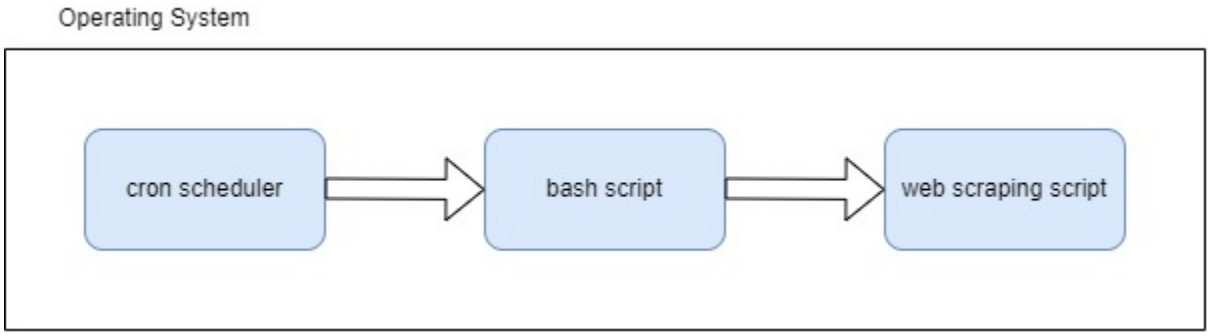

Figure 10

5.1 Comparison Script Log Files

The output of the log files is plain text, tab separated values for easy use with spreadsheet software or other data analytics (R Studio etc.). The following is an example of the log files obtained by the *sitetest.py* script.

```
******** Python requests method.
Time Taken = 25.565601348876953 seconds
Percentage of web pages downloaded successfully: 98.00%
www.aib.ie           PASS    256770
www.amazon.co.uk     PASS    7102
www.bbc.co.uk        PASS    654339
www.bbc.com          PASS    497190
www.bet365.com       PASS    3726
www.booking.com      PASS    1174160
www.daft.ie          PASS    631032
www.dailymail.co.uk PASS    3346002
www.donedeal.ie      PASS    766576
```

The output shows the time taken for each method, along with the efficacy rate, then for each URL in the input list the script records the URL, the efficacy and the size of the downloaded web page source code.

The software component of the downloader is a Python script called *webscraper.sh* that accepts an input text file consisting of a series of web addresses (URLs). The following is a list of processes which are followed in sequential order by the downloader:

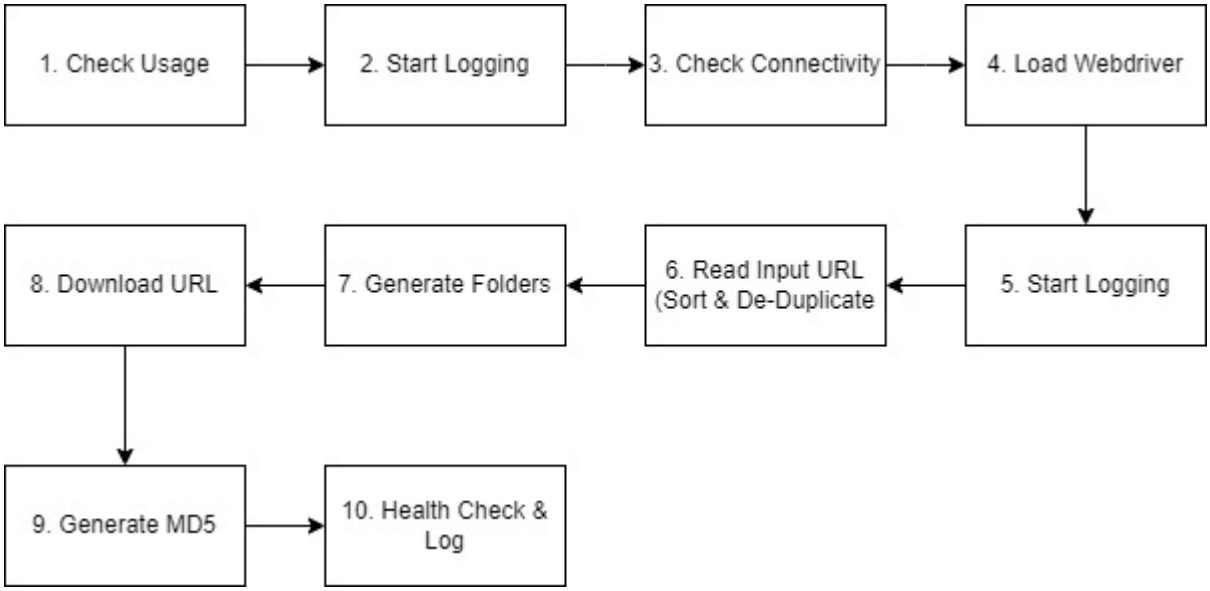

Figure 11

The first task performed by the web scraper is to check the input parameters are correct and that the input file is present. If the correct number of arguments are not passed to the script, it will print the usage to the standard output and terminate.

The web scraper will next start logging all activity of the scraper, starting with the timestamp of commencement, and the status of the network connectivity. The log file is located in a subfolder under the main project folder and is stored in plain text. Log files are essential to track the success of the web scraper and identify any issues that arise. The log file could be e-mailed on a schedule to aid in monitoring.

Network connectivity is checked by accessing a Google server. If network connectivity is missing then the web scraper will terminate, and an alert can be sent.

The webdriver is an API used to automate functions in a web browser. The purpose of the API is to test website functionality and web applications. The Selenium library used for the web scraper to acquire the web site data utilises a webdriver, and as such, the webdriver is loaded to memory in order to be used later.

The web scraper creates a directory structure such as the following:

/Scraping_Project/<root domain>/<date>/<MD5 hash of URL>/

The content of the web page is downloaded and stored as a file called *'output.html'*. During the digital forensic part of the process, a log file named *hash.txt* is stored alongside the content. A graphical screenshot of the page is stored as *'screenshot.png'*. The output of the downloader is as follows:

```
2023-05-25 12:38:17.322882 Logging Started.
2023-05-25 12:38:17.323387 2023-05-25
2023-05-25 12:38:17.432258 Internet Connection OK
2023-05-25 12:44:32.955006 Logging Started.
2023-05-25 12:44:32.955348 2023-05-25
2023-05-25 12:44:33.035632 Internet Connection OK
2023-05-25 12:44:37.202286 Successfully Downloaded: /home/peter/Dropbox/Python/Webscraper_Scripts/test_e
 2023-05-25/2ae6ed1021481b579bd5abbf04c02d4a/
2023-05-25 12:44:37.202422 Downloaded URL:
2023-05-25 12:44:38.626383 Successfully Downloaded: /home/peter/Dropbox/Python/Webscraper_Scripts/test_e
 2023-05-25/229d9da056e0e985a519048f7816952b/
2023-05-25 12:44:38.626473 Downloaded URL:
2023-05-25 12:46:25.474836 Logging Started.
2023-05-25 12:46:25.475037 2023-05-25
2023-05-25 12:46:25.521886 Internet Connection OK
2023-05-25 12:46:35.827850 Successfully Downloaded: /home/peter/Dropbox/Python/Webscraper_Scripts/test_e
 2023-05-25/51cd1c07986a834a01299546a8878ef8/
2023-05-25 12:46:35.828076 Downloaded URL:
2023-05-25 12:46:37.478593 Successfully Downloaded: /home/peter/Dropbox/Python/Webscraper_Scripts/test_e
 2023-05-25/033f80956ef93ba5c1c378fcd99d6d32/
2023-05-25 12:46:37.478764 Downloaded URL:
2023-05-25 12:46:37.479450 health_check Results
2023-05-25 12:46:37.479529 ************************************************************
2023-05-25 12:46:37.480352 /home/peter/Dropbox/Python/Webscraper_Scripts/test_env/Scraping_Project/www.c
 2023-05-25/2ae6ed1021481b579bd5abbf04c02d4a/output.html PASS
2023-05-25 12:46:37.480938 /home/peter/Dropbox/Python/Webscraper_Scripts/test_env/Scraping_Project/www.g
 2023-05-25/229d9da056e0e985a519048f7816952b/output.html PASS
2023-05-25 12:46:37.481620 /home/peter/Dropbox/Python/Webscraper_Scripts/test_env/Scraping_Project/www.h
 2023-05-25/51cd1c07986a834a01299546a8878ef8/output.html PASS
2023-05-25 12:46:37.487803 /home/peter/Dropbox/Python/Webscraper_Scripts/test_env/Scraping_Project/www.s
 2023-05-25/033f80956ef93ba5c1c378fcd99d6d32/output.html PASS
2023-05-25 12:46:37.488148 ************************************************************



2023-05-25 12:46:37.488513 4 Websites Downloaded with 0 errors.
2023-05-25 12:46:38.500230 Completed in : 13.02455735206604 seconds.
```

Figure 12

5.2 MD5 Hash

As discussed in Design Specification, the web scraper uses the MD5 hashing algorithm to ensure forensic acquisition of the web page. The method used to generate the MD5 hash value for the downloaded web code, and the URL to generate a unique folder, is the Python library *hashlib*.

5.3 Parsing the Data

The methodology for parsing is to use a *supervised syntactic* approach. The supervised approach means that the elements from the web page containing the desired data to be scraped must be specified. In order to store the elements a JSON (JavaScript Object Notation) configuration file is used to contain a list of root URLs and a list of the element names to parse for the data. Once the entry is found in the JSON file the web scraper will recursively scan

through all folders and *output.html* files found therein. The structure of the JSON configuration file is as follows:

```
{
    "webElements": [
        {
            "name": "www.currys.ie",
            "elContainer": ["div", "class", "product-tile"],
            "elBrand":["h2", "class", "pdp-grid-product-name"],
            "elModel":["h2", "class", "pdp-grid-product-name"],
            "elPrice":["span", "class", "sales"],
            "elPastPrice":["div", "class", "price-date"]
        },
```

Figure 13

The JSON container is named *webElements* and consists or a data structure containing the name of the root website DNS as the primary key. The data structure then consists of five (5) lists containing three (3) items each. The first list is the *elContainer* (element container) and is used to identify *'containers'* in the web site source code. Containers are used when there are multiple items using the same structure on a page. An example of this would be a page that shows a list of fifty products. Each one of the products would usually be in a container that uses the same structure and repeats fifty-times. The parser will first recursively scan through each *output.html* file contained in the subfolders of the project. When it finds a valid *output.html*, it uses the Beautiful Soup 4 (BS4) library in Python to create a Python dictionary of the tags from the web page source code. We can then use beautiful soup to search the dictionary for these tags and display the value contained under the tags in the web page source code.

The parser outputs by date, hash of URL, folder location and elements 1 through 4 as defined by the JSON. By default for price scraping purposes these are set to Brand, Model, Price, Previous Price.

Here is an example output from the parser:

2023-04-24	ae6e59a43412eb9676d659a125dcc513	/home/peter/Dropbc	www.appliancesdelivered.ie	LG	LG, OLED77C26LD, C2 77 Inch 4K Smart OLED TV, Black	€3,499.95	€3,799.95
2023-04-25	ae6e59a43412eb9676d659a125dcc513	/home/peter/Dropbc	www.appliancesdelivered.ie	LG	LG, OLED77C26LD, C2 77 Inch 4K Smart OLED TV, Black	€3,499.95	€3,799.95
2023-04-26	ae6e59a43412eb9676d659a125dcc513	/home/peter/Dropbc	www.appliancesdelivered.ie	LG	LG, OLED77C26LD, C2 77 Inch 4K Smart OLED TV, Black	€3,499.95	€3,799.95
2023-04-27	ae6e59a43412eb9676d659a125dcc513	/home/peter/Dropbc	www.appliancesdelivered.ie	LG	LG, OLED77C26LD, C2 77 Inch 4K Smart OLED TV, Black	€3,499.95	€3,799.95
2023-04-28	ae6e59a43412eb9676d659a125dcc513	/home/peter/Dropbc	www.appliancesdelivered.ie	LG	LG, OLED77C26LD, C2 77 Inch 4K Smart OLED TV, Black	€3,499.95	€3,799.95
2023-04-29	ae6e59a43412eb9676d659a125dcc513	/home/peter/Dropbc	www.appliancesdelivered.ie	LG	LG, OLED77C26LD, C2 77 Inch 4K Smart OLED TV, Black	€3,399.95	€3,799.95
2023-04-30	ae6e59a43412eb9676d659a125dcc513	/home/peter/Dropbc	www.appliancesdelivered.ie	LG	LG, OLED77C26LD, C2 77 Inch 4K Smart OLED TV, Black	€3,399.95	€3,799.95
2023-05-01	ae6e59a43412eb9676d659a125dcc513	/home/peter/Dropbc	www.appliancesdelivered.ie	LG	LG, OLED77C26LD, C2 77 Inch 4K Smart OLED TV, Black	€3,399.95	€3,799.95
2023-05-02	ae6e59a43412eb9676d659a125dcc513	/home/peter/Dropbc	www.appliancesdelivered.ie	LG	LG, OLED77C26LD, C2 77 Inch 4K Smart OLED TV, Black	€3,399.95	€3,799.95
2023-05-03	ae6e59a43412eb9676d659a125dcc513	/home/peter/Dropbc	www.appliancesdelivered.ie	LG	LG, OLED77C26LD, C2 77 Inch 4K Smart OLED TV, Black	€3,399.95	€3,799.95
2023-05-04	ae6e59a43412eb9676d659a125dcc513	/home/peter/Dropbc	www.appliancesdelivered.ie	LG	LG, OLED77C26LD, C2 77 Inch 4K Smart OLED TV, Black	€3,399.95	€3,799.95
2023-05-05	ae6e59a43412eb9676d659a125dcc513	/home/peter/Dropbc	www.appliancesdelivered.ie	LG	LG, OLED77C26LD, C2 77 Inch 4K Smart OLED TV, Black	€3,399.95	€3,799.95
2023-05-06	ae6e59a43412eb9676d659a125dcc513	/home/peter/Dropbc	www.appliancesdelivered.ie	LG	LG, OLED77C26LD, C2 77 Inch 4K Smart OLED TV, Black	€3,399.95	€3,799.95
2023-05-07	ae6e59a43412eb9676d659a125dcc513	/home/peter/Dropbc	www.appliancesdelivered.ie	LG	LG, OLED77C26LD, C2 77 Inch 4K Smart OLED TV, Black	€3,399.95	€3,799.95
2023-05-08	ae6e59a43412eb9676d659a125dcc513	/home/peter/Dropbc	www.appliancesdelivered.ie	LG	LG, OLED77C26LD, C2 77 Inch 4K Smart OLED TV, Black	€3,399.95	€3,799.95
2023-05-09	ae6e59a43412eb9676d659a125dcc513	/home/peter/Dropbc	www.appliancesdelivered.ie	LG	LG, OLED77C26LD, C2 77 Inch 4K Smart OLED TV, Black	€3,399.95	€3,799.95
2023-05-10	ae6e59a43412eb9676d659a125dcc513	/home/peter/Dropbc	www.appliancesdelivered.ie	LG	LG, OLED77C26LD, C2 77 Inch 4K Smart OLED TV, Black	€3,399.95	€3,799.95

Figure 14

The first column is the date of the scrape and is the primary key for which the output is sorted. The second column shows an MD5 hash representing the URL. Column three shows the location of the files on the filesystem, column four shows the brand, column five shows the model, column six shows the current price, and column seven shows the previous price. All of this data was obtained from scraping open-source websites and extracted from the page source code.

6 Evaluation

The data found during the five (5) measurement comparison of the six (6) methods on a sample size of fifty (50) across the four (4) test environments will be presented. The totals will be shown first to give an overview and the results will then be broken down by each test environment. Control group results on the sample size of twenty (20) removing efficacy as a variable will also be presented and form part of the dataset.

6.1 Total Efficacy Rates

The following efficacy rates were found across the different environments. The table is presented first, followed by graphs for visualisation.

Table 2

		AWS	Linode	Azure	Local	Avg
Wget		70.40%	69.20%	69.60%	70.80%	**70.00%**
URLLib		82.40%	80.40%	82.40%	82.80%	**82.00%**
Requests		98.00%	98.00%	98.00%	96.80%	**97.70%**
Selenium		99.60%	99.20%	99.60%	100.00%	**99.60%**
Pypetteer		97.60%	98.00%	98.00%	97.20%	**97.70%**
HTTPLib2		93.60%	91.60%	93.60%	93.60%	**93.10%**

The x-axis shows the methods, with each bar showing a different test environment, and the y-axis shows the efficacy rate in terms of percent.

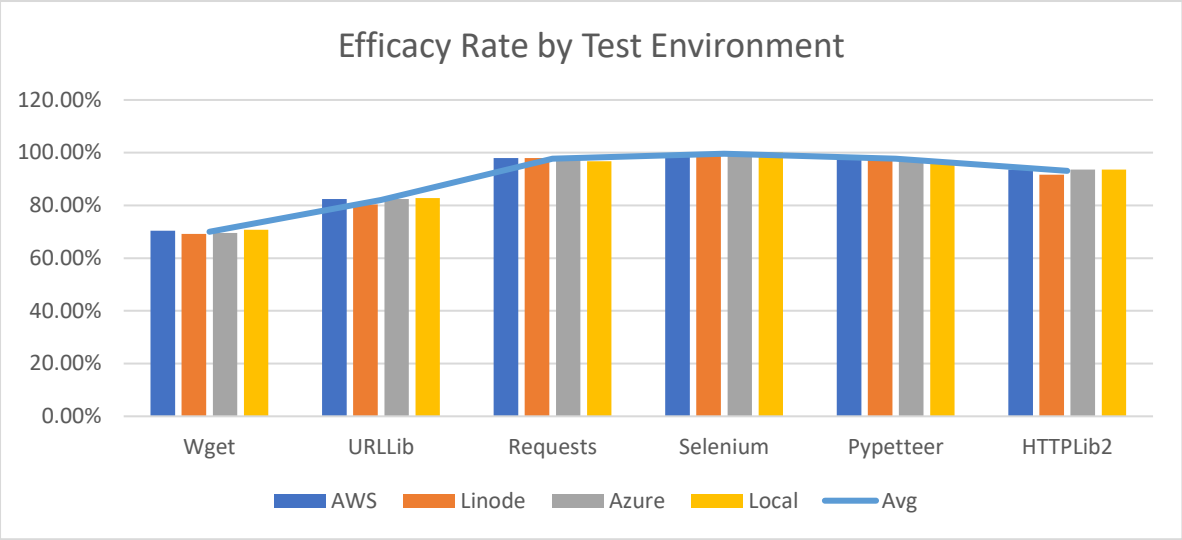

Figure 15

We can see when measuring efficacy that the Selenium method has the highest success rate at 99.60%. In fact, the Selenium method scored above 99% consistently across the three cloud test environments over the five measurements. The Wget method showed the lowest efficacy rate of the methods tested at 70%.

6.2 Total Average Data Scraped

The average data obtained over the course of the test in bytes. Selenium scraped the most data across the test environments, with the Azure Cloud Platform scraping the most data.

Table 3

		AWS	Linode	Azure	Local	Avg
Wget		686666	692695	689466	692454	**690,320**
URLLib		395411	400706	393274	398333	**396,931**
Requests		817680	820099	836345	844986	**829,777**
Selenium		1241046	1156664	1297043	1268101	**1,240,714**
Pypetteer		987543	928059	1017541	974151	**976,823**
HTTPLib2		390049	402317	395092	400780	**397,059**

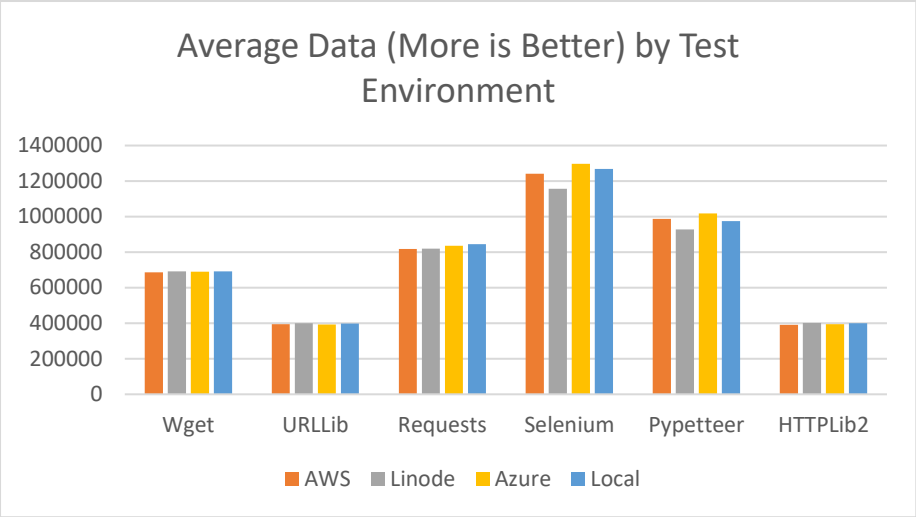

Figure 16

The average data downloaded test carried out on the control group sample size of twenty (20) of 100% efficacy items revealed the following amount of data in bytes:

Table 4

		AWS	Linode	Azure	Local	Avg
Wget		1,064,858	1,057,605	1,039,545	1,036,242	**1,049,562**
URLLib		517,471	521,664	523,658	517,591	**520,096**
Requests		1,015,049	1,017,713	1,029,811	1,022,979	**1,021,388**
Selenium		1,221,486	1,225,528	1,319,228	1,216,861	**1,245,776**
Pypetteer		1,163,007	1,143,877	1,219,537	1,133,833	**1,165,064**
HTTPLib2		517,446	521,304	523,783	517,500	**520,008**

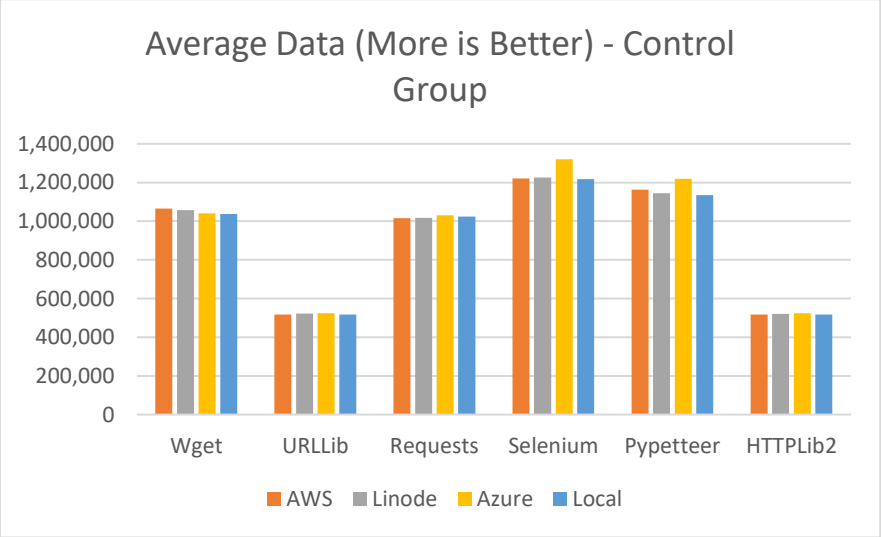

Figure 17

6.3 Total Average Time Taken

The following table lists the time taken to complete the scraping of the sample URLs across the different environments. The data was accumulated over a period of five (5) days, all at the same time, and the average rate is presented in seconds.

Table 5

		AWS	Linode	Azure	Local	Avg
Wget		16.30	16.31	21.47	21.07	**18.79**
URLLib		16.30	19.73	19.22	20.27	**18.88**
Requests		22.98	24.46	21.12	31.55	**25.03**
Selenium		158.74	240.77	128.92	230.55	**189.74**
Pypetteer		148.23	230.72	114.37	219.67	**178.25**
HTTPLib2		20.70	21.23	23.22	28.80	**23.49**

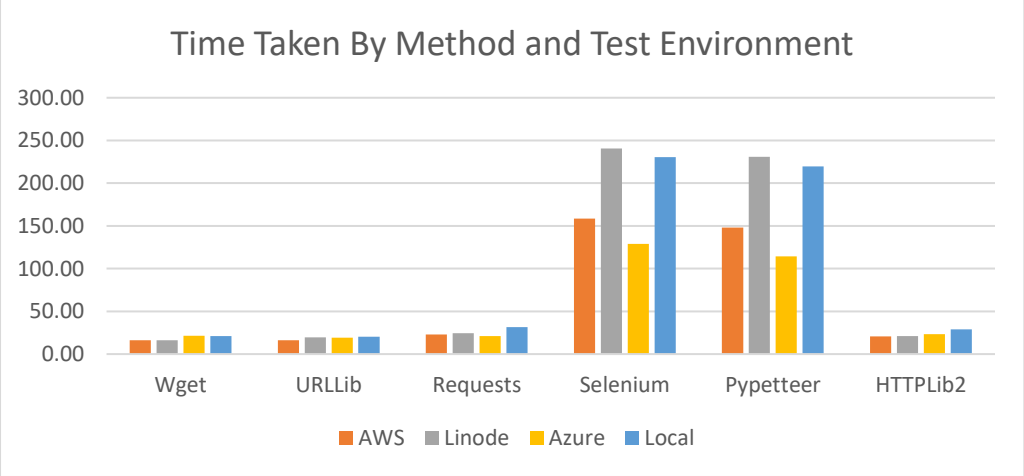

Figure 18

The timing tests carried out on the control group with a sample size of twenty (20) with 100% efficacy items revealed the following results measured in seconds:

Table 6

		AWS	Linode	Azure	Local	Avg
Wget		6.07	5.78	10.89	11.11	**8.46**
URLLib		7.40	6.34	7.86	9.35	**7.74**
Requests		10.26	7.50	9.01	12.70	**9.87**
Selenium		55.82	83.33	54.97	83.32	**69.36**
Pypetteer		61.10	97.42	53.14	99.46	**77.78**
HTTPLib2		8.83	10.80	11.98	11.42	**10.76**

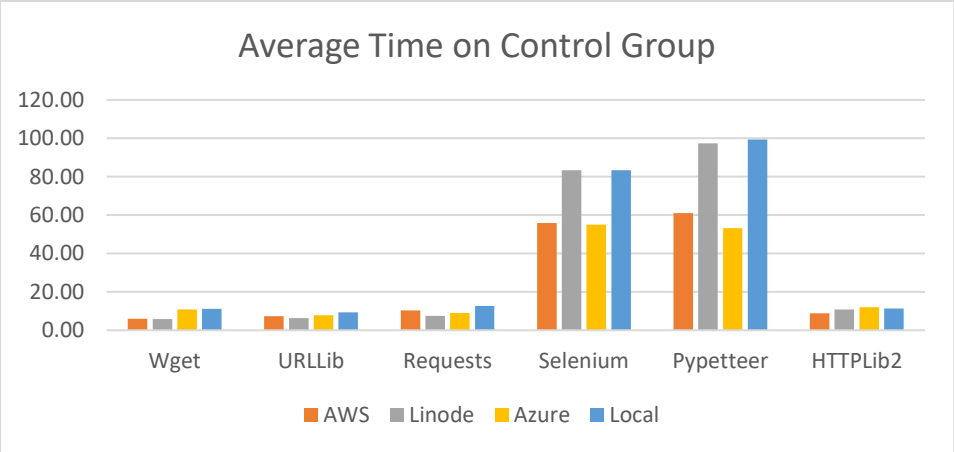

Figure 19

The average time per method, measured in seconds, was calculated by averaging across all test environments and is as follows:

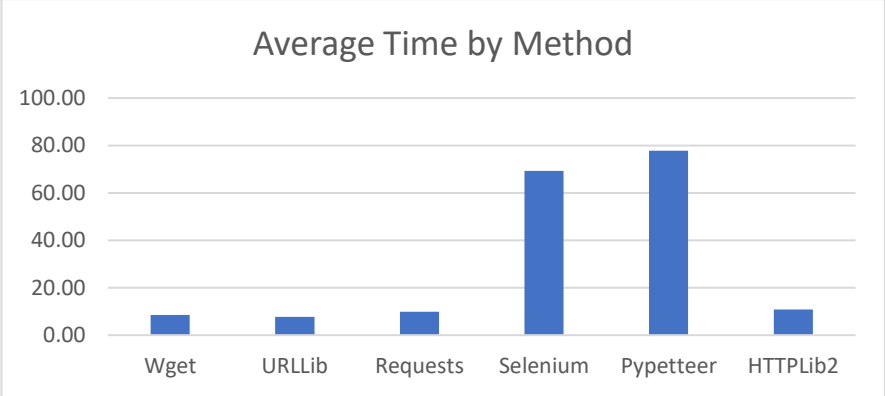

Figure 20

6.4 Network Latency

The following table shows the average ping values recorded for each of the test environments over the measurements. This data can be used in conjunction with the time taken data to establish is network conditions may have influenced the different methods.

Table 7

		M1	M2	M3	M4	M5	Avg
AWS		1.424	1.505	1.47	1.414	1.494	1.4614
Linode		0.831	0.863	0.844	0.872	0.85	0.852
Azure		2.424	2.412	2.387	1.647	1.501	2.0742
Local		2.145	2.136	2.253	2.264	2.224	2.2044

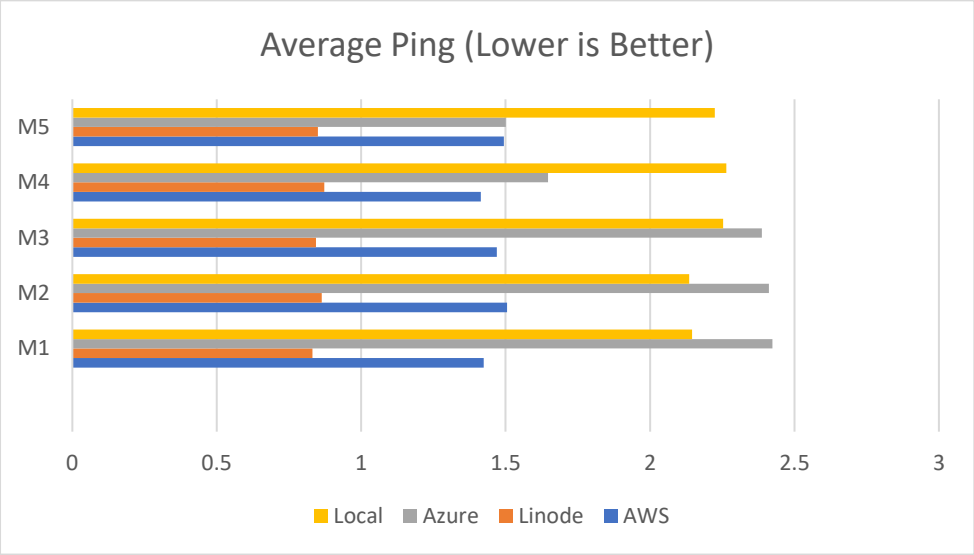

Figure 21

We can see that each test environment has quite low latency, with Linode coming in with the lowest ping latency. This metric is an overview, and better relevance to the *'time taken'* test would be gained by recording the latency to each of the target URLs in the sample to evaluate any correlation between network latency and time taken to complete. As it stands based on the data of the experiment, we can say that the low ping latency measured against the Google DNS servers has no direct correlation to low time taken for completion of each of the methods. The correlation co-efficient for time-taken by test environment and latency by test environment is : -0.083966527.

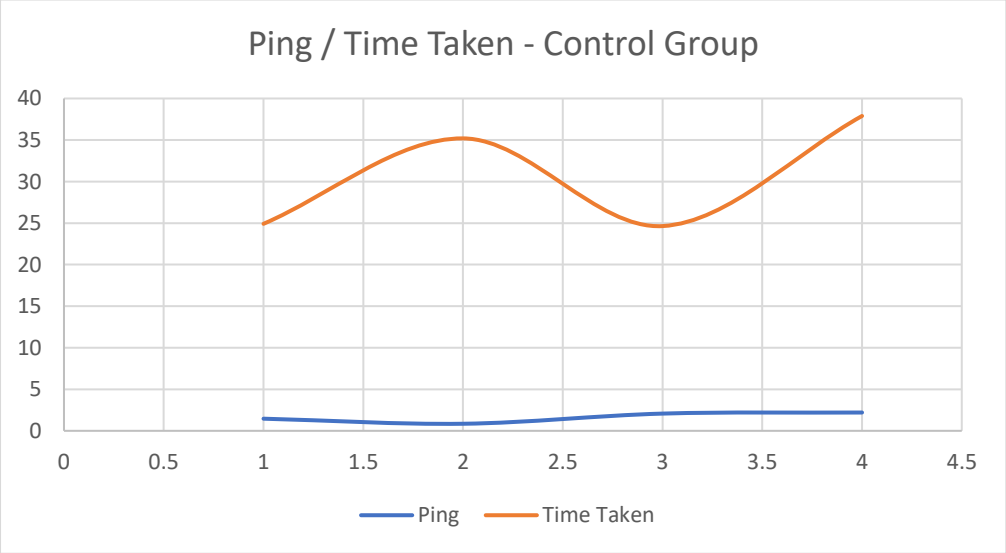

Figure 22

6.5 Web Scraper Methodology Evaluation

In evaluating the proposed forensic web scraper, the web scraper was run over a number of different online vendors over a period of time with successful results and a large amount of parsed pricing data. Using the test vendor for this research, the site was scraped 88 times, with two URLs each time. The data was parsed and produced 3556 lines of pricing information.

2023-07-03	cf523562a965049d2bf7be44e813f3b6	/home/peter/Test-MSc/Scraping_Project/www.did.ie/2023-07-03/cf523562a965049d2bf7be44e813f3b6/output.html	www.did.ie	Apple	Apple iPhone 12 6.1" 64GB Smartphone - Black | MGJ53ZD/A	€829.99	NONE
2023-04-01	cf523562a965049d2bf7be44e813f3b6	/home/peter/Test-MSc/Scraping_Project/www.did.ie/2023-04-01/cf523562a965049d2bf7be44e813f3b6/output.html	www.did.ie	Apple	Apple iPhone 13 128GB Smartphone - Blue | MLPK3B/A	€869.99	NONE
2023-04-02	cf523562a965049d2bf7be44e813f3b6	/home/peter/Test-MSc/Scraping_Project/www.did.ie/2023-04-02/cf523562a965049d2bf7be44e813f3b6/output.html	www.did.ie	Apple	Apple iPhone 13 128GB Smartphone - Blue | MLPK3B/A	€869.99	NONE
2023-04-03	cf523562a965049d2bf7be44e813f3b6	/home/peter/Test-MSc/Scraping_Project/www.did.ie/2023-04-03/cf523562a965049d2bf7be44e813f3b6/output.html	www.did.ie	Apple	Apple iPhone 13 128GB Smartphone - Blue | MLPK3B/A	€869.99	NONE
2023-04-04	cf523562a965049d2bf7be44e813f3b6	/home/peter/Test-MSc/Scraping_Project/www.did.ie/2023-04-04/cf523562a965049d2bf7be44e813f3b6/output.html	www.did.ie	Apple	Apple iPhone 13 128GB Smartphone - Blue | MLPK3B/A	€869.99	NONE
2023-04-05	cf523562a965049d2bf7be44e813f3b6	/home/peter/Test-MSc/Scraping_Project/www.did.ie/2023-04-05/cf523562a965049d2bf7be44e813f3b6/output.html	www.did.ie	Apple	Apple iPhone 13 128GB Smartphone - Blue | MLPK3B/A	€869.99	NONE
2023-04-06	cf523562a965049d2bf7be44e813f3b6	/home/peter/Test-MSc/Scraping_Project/www.did.ie/2023-04-06/cf523562a965049d2bf7be44e813f3b6/output.html	www.did.ie	Apple	Apple iPhone 13 128GB Smartphone - Blue | MLPK3B/A	€869.99	NONE
2023-04-07	cf523562a965049d2bf7be44e813f3b6	/home/peter/Test-MSc/Scraping_Project/www.did.ie/2023-04-07/cf523562a965049d2bf7be44e813f3b6/output.html	www.did.ie	Apple	Apple iPhone 13 128GB Smartphone - Blue | MLPK3B/A	€869.99	NONE
2023-04-08	cf523562a965049d2bf7be44e813f3b6	/home/peter/Test-MSc/Scraping_Project/www.did.ie/2023-04-08/cf523562a965049d2bf7be44e813f3b6/output.html	www.did.ie	Apple	Apple iPhone 13 128GB Smartphone - Blue | MLPK3B/A	€869.99	NONE
2023-04-09	cf523562a965049d2bf7be44e813f3b6	/home/peter/Test-MSc/Scraping_Project/www.did.ie/2023-04-09/cf523562a965049d2bf7be44e813f3b6/output.html	www.did.ie	Apple	Apple iPhone 13 128GB Smartphone - Blue | MLPK3B/A	€869.99	NONE
2023-04-10	cf523562a965049d2bf7be44e813f3b6	/home/peter/Test-MSc/Scraping_Project/www.did.ie/2023-04-10/cf523562a965049d2bf7be44e813f3b6/output.html	www.did.ie	Apple	Apple iPhone 13 128GB Smartphone - Blue | MLPK3B/A	€869.99	NONE
2023-04-11	cf523562a965049d2bf7be44e813f3b6	/home/peter/Test-MSc/Scraping_Project/www.did.ie/2023-04-11/cf523562a965049d2bf7be44e813f3b6/output.html	www.did.ie	Apple	Apple iPhone 13 128GB Smartphone - Blue | MLPK3B/A	€869.99	NONE
2023-04-01	cf523562a965049d2bf7be44e813f3b6	/home/peter/Test-MSc/Scraping_Project/www.did.ie/2023-04-01/cf523562a965049d2bf7be44e813f3b6/output.html	www.did.ie	Apple	Apple iPhone 13 128GB Smartphone - Midnight | MLPF3B/A	€869.99	NONE
2023-04-02	cf523562a965049d2bf7be44e813f3b6	/home/peter/Test-MSc/Scraping_Project/www.did.ie/2023-04-02/cf523562a965049d2bf7be44e813f3b6/output.html	www.did.ie	Apple	Apple iPhone 13 128GB Smartphone - Midnight | MLPF3B/A	€869.99	NONE
2023-04-03	cf523562a965049d2bf7be44e813f3b6	/home/peter/Test-MSc/Scraping_Project/www.did.ie/2023-04-03/cf523562a965049d2bf7be44e813f3b6/output.html	www.did.ie	Apple	Apple iPhone 13 128GB Smartphone - Midnight | MLPF3B/A	€869.99	NONE
2023-04-04	cf523562a965049d2bf7be44e813f3b6	/home/peter/Test-MSc/Scraping_Project/www.did.ie/2023-04-04/cf523562a965049d2bf7be44e813f3b6/output.html	www.did.ie	Apple	Apple iPhone 13 128GB Smartphone - Midnight | MLPF3B/A	€869.99	NONE
2023-04-05	cf523562a965049d2bf7be44e813f3b6	/home/peter/Test-MSc/Scraping_Project/www.did.ie/2023-04-05/cf523562a965049d2bf7be44e813f3b6/output.html	www.did.ie	Apple	Apple iPhone 13 128GB Smartphone - Midnight | MLPF3B/A	€869.99	NONE
2023-04-06	cf523562a965049d2bf7be44e813f3b6	/home/peter/Test-MSc/Scraping_Project/www.did.ie/2023-04-06/cf523562a965049d2bf7be44e813f3b6/output.html	www.did.ie	Apple	Apple iPhone 13 128GB Smartphone - Midnight | MLPF3B/A	€869.99	NONE

Figure 23

6.6 Discussion

The efficacy rate across the four platforms shows that the Wget method is the least effective at 70% when compared with the most effective method which is Selenium with an average efficacy of 99.6%. Using efficacy as a primary metric, the Wget method should not be used or recommended as a web scraping solution. Other methods such as

The data shows that the Selenium method returns the most data, with an average of *1,240,714* bytes. The method which returned the least data on average was the URLLib method with an average of *396,931* bytes, which was very similar to the HTTPLib2 method.

With the HTTPLib2 and URLLib methods we can see that the methods make a HTTP request for HTML code, which in a successful case is returned. This HTML web page code does not include any data or code which is generated by JavaScript scripts in the code. The method does not call these scripts or obtain the results of them. Depending on the website architecture, this may be adequate to achieve successful price scraping, however with more complex web page coding and architecture, the pricing information can often be retrieved from back-end or server-side databases (often linked to stock control and other systems) and rendered to the web page. In evaluating these two methods, we see a difference in efficacy between the two, with HTTPLib2 being the more effective. The requests method is a decent performer in terms of efficacy, average data downloaded, and time taken.

However, when the high efficacy of the Selenium method is taking into consideration with the higher amount of data, we can see that Selenium is the preferred method for web scraping among any of the Python methods shown. The Pyppeteer method comes close, and utilises a similar methodology for rendering JavaScript content, but obtains less average data including

in the control group, has slightly lower efficacy and does not perform significantly faster. The Pyppeteer library is also a Java port and not as well supported as Selenium.

The price for the increased efficacy and data acquisition due to the use of the webdriver is that it takes longer and uses more memory during the process. The fastest method shown was Wget with an average of 18.79 seconds. The other HTML scrapers; URLLib, Requests and HTTPLib2 were also similar. The control group shows that the URLLib method was faster than the Wget method, indicating that efficacy was an influencing variable in the real-world test. The correlation co-efficient with ping average times measured against time taken on the control group is -0.084.

The time taken for the methods in the real-world and control group testing show that Selenium and Pypetteer are much slower than the other methods, and this can be accounted for due to use of web drivers and real web browsers to render the pages, however when considering an automated and forensic web scraping methodology the most important metric would logically be the efficacy and completeness of the data along with the forensic component. More work could be done measuring any correlation between network latency and the time taken for each method to complete. As it stands, evaluating the data with latency measured by pinging Google servers at the time of the scraping, there doesn't appear to be a correlation between latency and time taken in the control group using the sample size. There is a strong correlation (+0.68) in the control group between the average data downloaded and the amount of time taken, which is what we would expect.

There were no significant differences in efficacy or average data downloaded across the test environments, however when looking at the time taken, particularly in the control group results, we can see some differences, with Azure and AWS test environments showing a significant lower time taken than the Linode or local VM environments when evaluating the Selenium and Pyppeteer methods. In evaluating whether the use of cloud platforms is advisable we can see very slight differences in efficacy across them. The use of cloud platforms for IaaS (infrastructure as a service) will by definition mean that the network infrastructure configuration is managed by the CSP and may have a slight bearing on returns. There is also the possibility that cloud IP address ranges may be blacklisted by website cybersecurity services.

To summarise, the findings of the comparison experiment show that the Selenium method is the best for efficacy and average data downloaded. There is a strong correlation between the average data downloaded and the amount of time taken, but no correlation between ping times and amount of time taken, and that cloud environments are as viable as local virtual machines or servers.

7 Conclusion and Future Work

This research has proposed a methodology for automated forensic web scraping of pricing information. This objective was successfully achieved by use of a *comparison* script, a *downloader* script and a *parser* script.

7.1 Comparison Experiment

A successful experiment using controlled conditions and the scientific method was used to carry out a comparison of different Python methods.

The experiment to test the efficacy and efficiency of the six methods across cloud and virtualised local environments shows that there are notable differences in the results. The methods were evaluated only insofar as having a scientific basis to recommend the proposed methodology. The Selenium library with its use of webdriver technology to control a real web browser is shown to be the best methodology to scrape data from web sites. It has the highest efficacy and the highest amount of data scraped per site, compared to older traditional HTTP requests for code.

The experiment also showed that the cloud environments are absolutely a viable platform for running these scripts, particularly for running the downloader. The Amazon Web Services (AWS) platform seemed to be the slightly better choice in terms of efficacy. The Local VM proved to be a problem and had sporadic fails in the efficacy for unknown reasons. The layers of complexity and variables introduced in using a VMWare virtual machine on a Windows 11 host should have been avoided and a Type-1 hypervisor used instead, either a Microsoft Hyper-V hypervisor or a Proxmox hypervisor would produce more stable results in line with the cloud test environments. The final web scraper downloader was tested on a Type-1 hypervisor (Proxmox) and did not show any of the efficacy or timing issues of the Local VM.

7.2 Downloader

A successful methodology and working script for the forensic acquisition of web page data was developed and shown to work. The web pages were captured for an extended period without any issues and MD5 hash values for each download were retained. These hash values were checked and found to be correct. This would enable any digital forensic specialist to stand over the downloaded page and the subsequent output from the parser.

More work could be done on the proposed methodology for the downloader to account for pagination and lists of products. This is possible to integrate with the Selenium method used, but due to time constraints on this research it was not implemented.

More work can be done on the scalability of the proposed methodology as the webdriver solution can use server RAM depending on the size of the pages. This memory usage is mitigated by closing pages in the webdriver once they are acquired, and in testing this did not appear to be an issue once the script reached the stage of closing the pages. Under anomalous conditions, however, the script may not complete properly, which is why the bash script contains housekeeping such as killing rogue chrome processes or python processes after running each day.

There is scope for future research into the content returned by each of the methods, as the results showed differences in the webpage content size depending on the method used to scrape.

7.3 Parser

The parser script successfully read the JSON configuration file and produced a useful output in TSV format. The JSON configuration file is a solution that can enable new web sites and DOM models to be easily added to the parser, based on the root address of the web site.

The running of the parser script was not tested on a cloud platform, as it would be recommended to run the parser on a machine with more computational resources and RAM than that needed for the downloader. Cloud platforms could be used but traditionally could be expensive for computation when compared to a local computer or virtual machine. No timing tests were carried out on the parser, and there would likely be room for optimisation and testing there. The time constraints prevented this avenue from being fully explored.

7.4 Future

The future of web scraping will likely utilise artificial intelligence (AI). When examining the methodology, the most obvious area for AI to operate will be on the JSON configuration for the parser. This is currently a manual process, whereby a human is required to examine the web page source code and input the DOM tags into a JSON configuration file. Changes to the website architecture and DOM can result in previous configurations no longer parsing the correct data, which could be more effectively dealt with by way of an AI component or application programming interface (API) integration with this methodology.

Other future challenges to web scraping may present themselves with Web 3.0 when the architecture of the web changes, and the methodology for efficient and effective web scraping of online data may change. However, as it stands the outlined methodology in this research is a robust and modular approach that can present a good amount of agility since the core methodology involves downloading and retaining the source material. This will provide good resilience to different parsing rules.

References

Chapagain, A. (2019) Hands-On Web Scraping with Python: Perform advanced scraping operations using various Python libraries and tools such as Selenium, Regex, and others. Packt Publishing Ltd.

De S Sirisuriya, S.C.M. (2015) 'A Comparative Study on Web Scraping'. Available at: http://ir.kdu.ac.lk/handle/345/1051 (Accessed: 21 July 2023).

Diouf, R. et al. (2019) 'Web Scraping: State-of-the-Art and Areas of Application', in 2019 IEEE International Conference on Big Data (Big Data). 2019 IEEE International Conference on Big Data (Big Data), pp. 6040–6042. Available at: https://doi.org/10.1109/BigData47090.2019.9005594.

Farley, E.J. and Pierotte, L. (2017) 'Web Scraping'.

Fielding, R. et al. (1999) 'RFC2616: Hypertext Transfer Protocol–HTTP/1.1'. RFC Editor.

Harrington, J.E. (2006) 'Behavioral screening and the detection of cartels', European competition law annual, pp. 51–68.

Jha, P. et al. (2014) 'Robots exclusion protocol', Internation journal of emerging science and engineering, 2(5).

Khder, M. (2021) 'Web Scraping or Web Crawling: State of Art, Techniques, Approaches and Application', International Journal of Advances in Soft Computing and its Applications, 13(3), pp. 145–168. Available at: https://doi.org/10.15849/IJASCA.211128.11.

Lawson, R. (2015) Web Scraping with Python. Packt Publishing Ltd.

Milev, P. (2017) 'Conceptual approach for development of web scraping application for tracking information', Economic Alternatives, 3, pp. 475–485.

Nyamathulla, S., Ratnababu, P. and Shaik, N.S. (2021) 'A Review on Selenium Web Driver with Python', Annals of the Romanian Society for Cell Biology, pp. 16760–16768.

Prayudi, Y. and Sn, A. (2015) 'Digital chain of custody: State of the art', International Journal of Computer Applications, 114(5).

Roussev, V. (2009) 'Hashing and data fingerprinting in digital forensics', IEEE Security & Privacy, 7(2), pp. 49–55.

Upadhyay, S. et al. (2017) 'Articulating the construction of a web scraper for massive data extraction', in 2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT). 2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT), pp. 1–4. Available at: https://doi.org/10.1109/ICECCT.2017.8117827.

Uzun, E. (2020) ‘A Novel Web Scraping Approach Using the Additional Information Obtained From Web Pages’, IEEE Access, 8, pp. 61726–61740. Available at: https://doi.org/10.1109/ACCESS.2020.2984503.

Vidya, V., Saly, K. and Balan, C. (2022) ‘Forensic Acquisition and Analysis of Webpage’, in 2022 2nd International Conference on Intelligent Technologies (CONIT). 2022 2nd International Conference on Intelligent Technologies (CONIT), pp. 1–6. Available at: https://doi.org/10.1109/CONIT55038.2022.9848303.

Zheng, C., He, G. and Peng, Z. (2015) ‘A Study of Web Information Extraction Technology Based on Beautiful Soup.’, J. Comput., 10(6), pp. 381–387.