Crawling the Modern Web: A Python Guide for Cybersecurity Applications in the Age of Anti-Bot Defenses

Christian Stuart



CYBV 909 Master’s Report

Master of Science in Cyber and Information Operations

College of Applied Science & Technology

Spring 2025

[**List of Abbreviations 4**](#_p2khvunu4qe7)

[**Abstract 5**](#_ywjgydy5u13i)

[**Introduction 5**](#_gg9m5cahig17)

[1.1 Background 5](#_eldhkl9y5vdw)

[1.2 Problem Statement 7](#_hv2kuv91pmco)

[1.3 Research Objectives 8](#_cptwso1t8nrn)

[1.4 Scope and Limitations 10](#_829lux213lu9)

[1.5 Purpose and Contextual Framing 11](#_mpukcgtnfs5x)

[1.6 User-Agent & Request Basics 12](#_e2ucpzn77eoz)

[**Related Work 13**](#_8v1a8mzihxy5)

[2.1 Web Crawler Evolution 13](#_1nvrmyife5o6)

[2.2 Anti-Bot Protections 14](#_ditz8bos7fpm)

[2.3 CAPTCHA Solving 15](#_5zn05odw61pm)

[2.4 API Reconnaissance in Research 17](#_ss00bvsh8fq5)

[2.5 Tools and Frameworks 18](#_9wlw0b8ww870)

[**Methodology 19**](#_bssmq2x8wwlw)

[3.1 Tool Selection and Stack Overview 19](#_q67qf78ehq0s)

[3.2 Basic Crawler Implementation (Phase 1) 21](#_15y8lp916w2q)

[3.2.1 Version 1: URLLIB Implementation 21](#_1p3k5cdw8b0e)

[3.2.2 Version 2: requests Implementation 22](#_z3lhcgg3hk73)

[3.2.3 Version 3: Scrapy and BeautifulSoup Implementation 22](#_37hbffyn4ave)

[3.3 Intermediate Crawler Implementation (Phase 2) 23](#_m44z5gdl8591)

[3.3.1 Version 1: Basic httpx and Selectolax Approach 23](#_lf0ushoqpnff)

[3.3.2 Version 2: Enhanced Headers Approach 24](#_oxhpumuynaz2)

[3.3.3 Version 3: Data Extraction and Structured Output 25](#_g8pw2ktyiim)

[3.3.4 Cookie Handling & Session Management 25](#_ccxcen7v7m8c)

[3.4 Advanced Crawler Implementation (Phase 3) 27](#_3z09uxtmtrof)

[3.4.1 Version 1: Direct API Access with Playwright 27](#_1nxnb41jncxt)

[3.4.2 Version 2: Batch API Processing 28](#_cuui8umg2bm)

[3.4.3 Version 3: Proxy Integration and Field Filtering 29](#_j717pzys3bon)

[3.5 API Reconnaissance via DevTools 30](#_2vvw81h10t1i)

[3.6 Ethical and Legal Considerations 31](#_sqmckknjmbv2)

[**Results and Evaluation 31**](#_g69od3hgf9yq)

[4.1 Functional Comparisons 32](#_d3nsuqtwswva)

[4.2 Performance and Efficiency 33](#_2yzo617jwyud)

[4.3 Evasion Effectiveness 35](#_o0lumzsczqi1)

[4.4 Data Quality and Completeness 36](#_715vpyvdkl57)

[4.5 Scalability Considerations 38](#_dngdo7p0xrtu)

[4.6 Comparative Analysis of Crawler Evolution 39](#_1mbnsw84k0ld)

[**Discussion and Implications 41**](#_twdxhyg9owxm)

[**Conclusion and Future Work 43**](#_wt8qy98bdfsx)

[6.1 Future Work 43](#_1ru8c0j3yyfi)

[6.1.1 Moving Beyond Python 43](#_qzwn5e8qhkei)

[6.1.2 Taking Crawlers to the Cloud 44](#_x00zfo4xuvbg)

[6.1.3 Building Better Dashboards 44](#_6rvxxn6zb8bf)

[6.1.4 Applying This to OSINT Work 45](#_v7egbac7l8vr)

[6.2 Summary of Contributions 46](#_hfjqx1p30k8n)

[6.3 Final Notes 47](#_6ouf8a3b2h7a)

[**Code Snippets 48**](#_is3at3nih5h2)

[Basic Crawler 49](#_7ac8ftgt1dci)

[Code 3.2.1 - Version 1 49](#_4yiahbi0cq7d)

[Code 3.2.2 - Version 2 49](#_janqbl1l8xgm)

[Code 3.2.3 - Version 3 50](#_w4ngr7k94d9b)

[Intermediate Crawler 51](#_i57mkmozilsh)

[Code 3.3.1 - Version 1 51](#_x5w08f537loj)

[Code 3.3.2 - Version 2 51](#_l7usx2796fgh)

[Code 3.3.3 - Version 3 52](#_9fyj9fkiom67)

[Advanced Crawler 53](#_20yiociu9f1l)

[Code 3.4.1 - Version 1 53](#_knj5c94gzq7g)

[Code 3.4.2 - Version 2 54](#_p0qcdoq1akmg)

[Code 3.4.3 - Version 3 55](#_dou2jtz3p79j)

[**Appendices 57**](#_c4kg26un1imu)

[Figure 3-1: Wappalyzer Security Analysis of REI.com 57](#_c7cui6bc8fyb)

[Figure 3-2: Wappalyzer Security Analysis of Adidas.com 58](#_magfrdnc2mxf)

[Figure 3-3: Browser Developer Tools Network Panel During API Reconnaissance 58](#_52rtvi7zvo85)

[Figure 3-4: Detailed Headers for Adidas Product API Request 59](#_58c7nscjpbx7)

[Figure 3-5: JSON Response from Adidas Product API 60](#_25gwwh3jr32q)

[Figure 4-1: Average Request Time Comparison Across Crawler Implementations 61](#_b6u3oy5erxoh)

[Figure 4-2 Memory Usage Comparison Across Crawler Implementations 61](#_uvuq3cyzrgdc)

[Figure 4-3: Data Efficiency Ratio Comparison Across Crawler Implementations 62](#_iogtdmvaavlr)

[Figure 4-4: Evasion Success and Data Completeness Rates Across Crawler Implementations 62](#_bk9sf0salpeo)

[**References 63**](#_18whts9yrd9f)

# List of Abbreviations

| **Abbreviation** | **Definition** |
| --- | --- |
| API | Application Programming Interface |
| AWS | Amazon Web Services |
| CAPTCHA | Completely Automated Public Turing test to tell Computers and Humans Apart |
| CFAA | Computer Fraud and Abuse Act |
| CSS | Cascading Style Sheets |
| DOM | Document Object Model |
| HTML | Hypertext Markup Language |
| HTTP | Hypertext Transfer Protocol |
| HTTPS | Hypertext Transfer Protocol Secure |
| HSTS | HTTP Strict Transport Security |
| IP | Internet Protocol |
| JSON | JavaScript Object Notation |
| OSINT | Open Source Intelligence |
| REST | Representational State Transfer |
| RL | Reinforcement Learning |
| SSL | Secure Sockets Layer |
| TCP | Transmission Control Protocol |
| TLS | Transport Layer Security |
| UI | User Interface |
| URL | Uniform Resource Locator |
| XML | Extensible Markup Language |
| XHR | XMLHttpRequest |

# 

# Abstract

This report examines web crawling through the lens of cyber operations and security reconnaissance, demonstrating how Python-based crawling techniques have evolved to counter increasingly sophisticated defensive measures. The research progresses through three implementation phases, each representing tactical advancements in information gathering capabilities essential for modern security professionals. The Basic Crawler establishes fundamental crawling concepts using urllib and requests libraries, suitable for preliminary data harvesting from static targets. The Intermediate Crawler deploys httpx and Selectolax with enhanced header configurations to penetrate basic defensive perimeters, simulating more advanced persistent access techniques.

The Advanced Crawler demonstrates sophisticated operational capabilities including API endpoint reconnaissance, proxy rotation for anonymity, and targeted data extraction techniques directly applicable to security assessments and OSINT operations. Performance evaluation reveals that while simpler implementations excel in speed, they lack the stealth and persistence capabilities required for comprehensive cyber reconnaissance against hardened targets. The API-based approach proves particularly effective for operational security, providing superior data acquisition while minimizing detection risk. This research provides cyber operations specialists with a practical framework for developing scalable, resilient intelligence-gathering tools that can be deployed in both offensive security assessments and defensive monitoring operations, with potential applications in cloud-based threat intelligence and advanced persistent threat simulation.

# Introduction

## 1.1 Background

This report presents a progressive, cybersecurity-oriented approach to web crawling using Python. It serves as both an educational guide and a technical implementation roadmap, offering a structured exploration of how web crawlers evolve from basic HTML scrapers into advanced systems capable of navigating and bypassing modern web defenses (Fan, 2018; Wang & Guo, 2012).

Web crawling, in its most basic form, involves sending HTTP requests to websites, retrieving raw HTML content, and extracting information from the resulting document structure (Najork, 2009). This report begins with that foundational approach using Python’s URLLIB library to perform basic URL requests before introducing more powerful and flexible tools such as HTTPx and Selectolax to handle more complex parsing tasks (Wahed et al., 2024; Bhale & Sonar, 2024). These tools allow for structured HTML traversal, basic automation, and scalable scraping pipelines suited to static or lightly dynamic sites (Pant et al., 2024).

As the project progresses, it addresses the increasing complexity of modern websites, which frequently render content dynamically through JavaScript (Wahed et al., 2024; Wang & Guo, 2012). Traditional scraping methods often fail to capture such content. Therefore, this report introduces headless browser automation tools, such as Playwright, which simulate real-user interactions within an actual browser environment. These tools enable access to JavaScript-rendered content, support infinite scrolling, and simulate realistic session behavior capabilities that are critical for bypassing security layers designed to block traditional crawlers (Azad et al., 2020; Tsingenopoulos et al., 2022).

Beyond content retrieval, the report delves into the broader topic of backend API reconnaissance. Many websites use hidden or semi-obscured APIs to power their dynamic content. These endpoints, while not visible in the page source, can be observed through network activity and exploited to retrieve structured data directly (Demchenko et al., 2024; Glez-Peña et al., 2013). By analyzing browser developer tools and automating traffic inspection, crawlers can identify and access these APIs more efficiently than by scraping HTML, allowing them to bypass the front-end entirely (Wahed et al., 2024).

In the context of cybersecurity, web crawling is more than just data extraction it’s a powerful reconnaissance tool. Penetration testers and security researchers routinely use crawlers to identify exposed endpoints, map web application structures, and analyze API behavior (Sun et al., 2010; Jiang et al., 2012). However, this capability is increasingly met with resistance. Anti-bot platforms like Cloudflare, Akamai, and DataDome deploy browser fingerprinting, JavaScript challenges, CAPTCHA validation, and IP-based rate limiting to prevent automated access (Azad et al., 2020; Chiapponi et al., 2021; Chiapponi et al., 2023). As a result, the development of modern crawlers requires not only robust scraping capabilities but also advanced evasion techniques, such as proxy rotation, user-agent spoofing, and CAPTCHA-solving services (Wahed et al., 2024; Tsingenopoulos et al., 2022; Kumar & Singhal, 2014).

This report aims to guide the reader through this progression from simple crawlers to stealthy data retrieval agents. By combining practical implementation with theoretical insight, it provides a comprehensive perspective on the techniques, tools, and ethical considerations involved in web crawling as it applies to modern web security.

## 1.2 Problem Statement

As web technologies have evolved, the effectiveness of traditional web crawlers has significantly diminished. The shift from static HTML pages to dynamic, JavaScript-rendered content has created major obstacles for basic crawlers that rely on standard HTTP requests and simple HTML parsers (Wahed et al., 2024; Fan, 2018). These outdated techniques often fail to retrieve meaningful data from modern websites, making them obsolete for many real-world use cases (Sharma & Gupta, 2015).

Despite this, understanding these earlier methods remains important. They form the conceptual foundation of web crawling and help highlight the reasons why modern approaches are necessary. By studying what worked in the past and more importantly, why those methods now fail we gain a deeper appreciation for the complexities introduced by current web architectures (Najork, 2009; Jiang et al., 2012).

The rise of anti-bot protections has further increased these challenges. Services like Cloudflare, Akamai, and DataDome actively defend websites against automated access through a combination of techniques, including browser fingerprinting, JavaScript-based bot detection, behavior analysis, and CAPTCHA validation (Azad et al., 2020; Chiapponi et al., 2021; Chiapponi et al., 2023). These defenses are designed to mimic and validate real human behavior, making it increasingly difficult for conventional crawlers to succeed.

To effectively navigate these obstacles, a modern crawler must not only emulate real-user behavior. It must also anticipate and evade detection. This involves a layered strategy: rotating proxy IP addresses to avoid rate limits (Kumar & Singhal, 2014), spoofing browser characteristics to defeat fingerprinting (Wahed et al., 2024), identifying hidden API endpoints to bypass the UI layer (Demchenko et al., 2024), and integrating automated CAPTCHA solvers to overcome access challenges (Tsingenopoulos et al., 2022).

Without these evasive techniques, even well-engineered crawlers will be blocked or throttled before they can complete their tasks. Therefore, the evolution of web crawlers must be approached as a phased development process one that begins with static scraping and culminates in a fully automated, anti-bot-resistant system.

This report addresses the full scope of that evolution, combining technical depth with cybersecurity relevance to provide an actionable and insightful framework for building advanced web crawlers.

## 1.3 Research Objectives

The primary aim of this project is to investigate the evolution of web crawling techniques in response to increasingly dynamic and protected web environments. This is achieved through the progressive development of a modular, Python-based web crawler. The research is structured around the following key objectives in Figure 1-1:

*Figure 1-1: Key Objectives Table*

| **Phase** | **Focus Area** | **Key Activities / Objectives** |
| --- | --- | --- |
| 1. Basic Crawler | Static Web Crawling | - Develop initial crawler using urllib and requests library  - Retrieve static HTML content from simple websites |
| 1. Intermediate Crawler | Asynchronous & Structured Parsing | - Use HTTPx and Selectolax to parse HTML  - Implement support for asynchronous scraping workflows |
| 1. Advance Crawler | Dynamic Content, API Discovery, and Bot Evasion | - Integrate Playwright for JavaScript rendering and browser automation  - Show and explain hidden API endpoints via developer tools  - Bypass front-end rendering by accessing structured backend data  - Implement anti-bot strategies:  • Proxy rotation  • User-Agent and header spoofing  • Third-party CAPTCHA solving (If required) |
| 1. Scalability at a Theoretical Level | Cloud Deployment Design | - Design a scalable crawler architecture containers (e.g., Docker)  - Conceptualize proxy orchestration and cloud-based distribution |
| 1. Evaluation & Analysis | Performance, Limitations & Evasion | - Analyze the limitations of earlier approaches (urllib, requests) in handling modern websites  - Evaluate each phase based on performance, success rate, and evasion capability |
| 1. Ethics | Legal & Ethical Compliance | - Frame the project within cybersecurity research boundaries  - Ensure ethical scraping practices and respect for public access boundaries |

## 1.4 Scope and Limitations

This project focuses on building a foundational understanding of how web crawlers have evolved to address the increasing complexity of the modern web. The primary objective is to create a modular, beginner-accessible crawler that progresses from static content scraping to advanced, evasive techniques (Wahed et al., 2024; Fan, 2018). The crawler is implemented using Python and a variety of well-documented, widely-used libraries such as URLLIB, HTTPx, Selectolax, Scrapy, BeautifulSoup, and Playwright (Pant et al., 2024; Bhale & Sonar, 2024). The user should be familiar with Python at a working level, and understand how to read through the various library documentations used in paper, as this paper is not a documentation encyclopedia for those libraries or a “How To” on learning Python.

To keep the project accessible and maintain focus on practical, actionable knowledge, several boundaries have been intentionally defined. High-level or specialized approaches such as developing custom machine learning models to solve CAPTCHAs or using proprietary AI tools for behavioral evasion are explicitly out of scope (Tsingenopoulos et al., 2022). While such technologies are acknowledged and may be mentioned conceptually, they are not implemented or explored in depth, as they would detract from the educational value of this report and introduce unnecessary complexity for readers seeking a clear, technical foundation.

Additionally, this paper does not attempt to build or deploy a large-scale production crawler. Cloud-native deployments (e.g., on AWS with Kubernetes) are discussed only at a theoretical level to illustrate how the system might scale (Bahrami et al., 2015). Similarly, any data retrieval performed during testing is limited to publicly accessible content; no login bypassing, password scraping, or unauthorized API access is conducted. The intent of the crawler is educational, not exploitative, and all testing remains within ethical boundaries (Sun et al., 2010).

## 1.5 Purpose and Contextual Framing

The primary goal of this project is not to create a cutting-edge web crawler for operational use, but rather to provide a clear, progressive understanding of how modern websites have evolved and how that evolution has fundamentally changed the practice of web scraping and data capture (Wahed et al., 2024; Demchenko et al., 2024).

By walking through the different phases of crawler development, this report highlights how increasing use of JavaScript, API-driven architectures, and anti-bot protections have rendered traditional scraping techniques less effective or entirely obsolete (Wang & Guo, 2012; Bhale & Sonar, 2024). In turn, modern crawlers have had to adapt by incorporating headless browsers, rotating proxy infrastructures, CAPTCHA solvers, and, where possible, the ability to identify and directly query backend APIs, bypassing the front-end entirely (Tsingenopoulos et al., 2022; Glez-Peña et al., 2013).

This evolution is not purely technical it reflects a broader transformation in how data is both delivered and protected across the modern web. The crawler developed here serves as a tool to demonstrate that shift in real terms, offering insights into how different libraries, services, and techniques must be selected or combined depending on the structure and defenses of the target website (Wahed et al., 2024; Chiapponi et al., 2023). At scale, these adaptations form the foundation for more robust, resilient scraping architectures used across research, cybersecurity, analytics, and automation (Azad et al., 2020).

Ultimately, this paper provides a beginner-accessible path through these concepts, designed to help students, researchers, and professionals alike understand not only the "how" of web crawling, but the "why" behind each technological leap.   
 With this technical and ethical framing established, the next section explores how existing academic and industry research has shaped modern crawling strategies particularly as they relate to web crawling evolution, bot detection, JavaScript rendering, CAPTCHA solving and API access.

## 1.6 User-Agent & Request Basics

Modern anti-bot platforms analyze the structure of each inbound HTTP request not just its source IP address. The User-Agent string, which identifies the client's browser, operating system, and rendering engine, represents one of the first elements that defensive systems examine. Requests using outdated, generic, or mismatched User-Agent strings are frequently identified as automated traffic and blocked accordingly (Azad et al., 2020; Chiapponi et al., 2021). Initial implementations in this project clearly demonstrated this filtering effect. A basic httpx request containing only the default User-Agent consistently received 403 Forbidden responses from REI.com, despite establishing valid TLS connections. When the same request was enhanced with a complete header set including Accept, Accept-Language, Accept-Encoding, Connection, and the newer Sec-Fetch-\* parameters the server immediately returned 200 OK responses with full HTML content (Wahed et al., 2024). While standard web browsers automatically generate dozens of headers (many varying between requests), programmatic crawlers must deliberately construct these elements. The crawler implementations in this report follow several key principles:

First, header consistency takes precedence over complexity. A sophisticated User-Agent provides little benefit if secondary headers like Sec-CH-UA or Sec-Fetch-Site contain contradictory information. Effective header implementations require coherent, browser-realistic configurations (Tsingenopoulos et al., 2022). Second, strategic rotation improves resilience. Since anti-bot systems track suspicious fingerprints, alternating between recent mainstream User-Agents (such as Chrome 121, Firefox 124, or Safari 17) reduces pattern recognition without triggering alerts from unusual or deprecated signatures (Kumar & Singhal, 2014). Third, contextual alignment enhances believability. Mobile User-Agents must be paired with appropriate viewport dimensions, touch-device parameters, and TLS cipher suites typical of Android or iOS devices. Inconsistencies between these elements can trigger downstream JavaScript detection mechanisms even when the initial HTTP request successfully passes preliminary screening (Chiapponi et al., 2023).

Throughout Phases 2 and 3, the crawler implementations employ contextually-appropriate, rotating User-Agent strings and corresponding headers when establishing connections whether through httpx.Client, Playwright's APIRequestContext, or headless browser instances. This approach, combined with proxy rotation and timing randomization (detailed in Section 3.4), establishes the foundational camouflage upon which more sophisticated evasion techniques are constructed.

# Related Work

This section places the development of modern web crawlers into a broader academic and technical context. It traces how foundational crawling strategies have evolved in response to the growing complexity of the web, and how this evolution has sparked parallel developments in anti-bot protections, CAPTCHA systems, API reconnaissance, and the libraries that support scalable scraping today.

## 2.1 Web Crawler Evolution

Web crawlers have long served as the backbone for indexing, monitoring, and extracting information from the internet. However, their design and capabilities have changed significantly over the past two decades. Earlier crawlers designed for static HTML pages operated by simply following links, parsing content, and storing data using fairly straightforward queueing logic. These systems were sufficient when websites consisted mostly of static documents with predictable structure and few access controls (Najork, 2009).

However, the shift toward JavaScript-driven content, user-specific rendering, and API-heavy architectures introduced entirely new challenges. Fan (2018) explored the necessity of adapting crawling systems to manage large-scale data acquisition tasks through distributed implementations. His Scrapy-based distributed crawler addressed challenges like scheduling, concurrency, and efficiency across multiple systems a pattern that now underpins many modern scraping frameworks.

The technical hurdles became especially clear in projects like that of Wang and Guo (2012), who attempted to crawl user behavior data from Taobao. They encountered obstacles ranging from JavaScript-rendered content to anti-crawling mechanisms that rejected traditional scraping methods. Their solution involved simulating browser behavior and carefully managing IP access key techniques that remain highly relevant today.

The evolution of crawler architectures has been driven not only by the need to deal with dynamic content, but also by the increasingly defensive stance websites have taken toward bots. This dual pressure richer frontends and more robust protections has made modern scraping more reliant on headless browsers, proxy networks, and hybrid techniques that combine traditional parsing with interactive automation. For many applications, the crawler is no longer a standalone tool it is a pipeline of adaptive components that must work together to remain effective.

## 2.2 Anti-Bot Protections

As the web matured, so too did its defenses. Websites increasingly deploy anti-bot technologies to protect content, reduce server load, and maintain control over data access. These systems range from basic rate-limiting to sophisticated, multi-layered platforms like Cloudflare, Akamai, and Datadome, which use behavioral analysis, browser fingerprinting, TLS entropy, and JavaScript challenges to detect and stop automated traffic.

Azad et al. (2020) conducted one of the most comprehensive academic evaluations of these third-party anti-bot services. Their work highlighted the arms race between detection systems and the techniques crawlers use to remain undetected. For example, basic bots often fail due to lack of browser fingerprinting or improper rendering of JavaScript, while more advanced scrapers must deploy stealth headless browsers and behavioral scripts to avoid suspicion.

The implications of anti-bot defenses go beyond just blocking traffic they actively shape the strategies that modern crawlers must adopt. Honeypot deployments, like those studied by Chiapponi et al. (2021), reveal how bots adapt their behavior based on feedback from the environment, withdrawing when detection is suspected and changing tactics in response. This behavioral elasticity makes anti-bot detection a moving target.

Later work by Chiapponi et al. (2023) focused on the infrastructure supporting bot evasion specifically, residential proxy networks. These networks claim to provide millions of rotating IPs that can mimic real users. However, the authors discovered that the actual number of unique IPs was often overstated, suggesting that IP-based defenses may still be viable when combined with reputation scoring and behavioral fingerprinting.

Modern crawlers must now navigate a minefield of detection points, from IP rotation and session headers to JavaScript execution patterns and timing intervals. Understanding how these defenses work is no longer optional it’s a core competency for anyone building data extraction systems in today's web environment.

## 2.3 CAPTCHA Solving

CAPTCHAs (Completely Automated Public Turing tests to tell Computers and Humans Apart) have become one of the most widely used lines of defense against web scraping and automated interaction. Initially designed as simple image or text challenges, modern CAPTCHA systems such as Google’s reCAPTCHA v3 now operate invisibly, scoring users based on behavioral cues like mouse movement, click timing, scroll depth, and interaction patterns. These systems create a new layer of friction that is both subtle and difficult to reverse-engineer.

As CAPTCHAs evolved, so too did the methods to bypass them. Traditional approaches such as outsourcing to human solvers are still used, but they introduce latency, cost, and scaling issues. Increasingly, attackers are turning to machine learning to automate CAPTCHA solving. One such breakthrough is highlighted in the work of Tsingenopoulos et al. (2022), who developed a reinforcement learning (RL) agent capable of bypassing reCAPTCHA v3 by mimicking legitimate human browsing behavior. Their agent was trained not by direct feedback on the CAPTCHA itself, but by analyzing the final risk score provided by the service. Over time, the model learned which interaction patterns led to higher scores, effectively training itself to pass undetected.

This form of indirect adversarial learning marks a significant shift in CAPTCHA evasion. It shows that even when CAPTCHA challenges are hidden from the user (as in reCAPTCHA v3), the system remains vulnerable to modeling and imitation. Moreover, it reinforces the idea that anti-bot measures cannot rely solely on user behavior as a trusted signal especially when that behavior can be learned and simulated.

Wahed et al. (2024) offer a more general discussion of CAPTCHA as a scraping challenge, placing it among several common roadblocks in Python-based scraping. Their work suggests that for many use cases, it is not worth trying to “solve” CAPTCHA at all. Instead, their recommendation is to design crawlers that avoid triggering CAPTCHAs in the first place by rotating IPs, slowing request rates, or focusing on less-defended endpoints.

Together, these studies demonstrate that CAPTCHA systems are neither unbreakable nor always necessary to defeat. Avoidance, evasion, and stealth often prove more effective than brute-force solving, especially for researchers and developers aiming to extract data ethically and at scale.

## 2.4 API Reconnaissance in Research

While much of web scraping focuses on extracting data from rendered HTML pages, a more efficient and resilient method involves identifying and accessing the underlying APIs used by the site itself. These APIs typically built for frontend JavaScript to request dynamic content often return clean, structured data in JSON or XML format, without the clutter or unpredictability of rendered markup. In many cases, they provide faster, more scalable access to the same information that a human user would see in the browser.

Demchenko et al. (2024) provide a comprehensive exploration of this concept, detailing how APIs, datasets, and web endpoints can be discovered, reverse-engineered, and leveraged for data acquisition. Their work emphasizes the importance of using browser developer tools (like the network panel in Chrome or Firefox) to observe background requests triggered by page interactions. These endpoints are often unprotected by CAPTCHAs or complex anti-bot mechanisms and, once discovered, can be queried directly saving both bandwidth and complexity.

However, API-based scraping is not without challenges. Many APIs require authentication, rate-limiting headers, or tokenized requests that must be generated dynamically. In some cases, accessing these APIs without permission may violate a site's terms of service or trigger legal scrutiny. As Wahed et al. (2024) caution, developers should assess not only the technical feasibility of API use, but also its ethical and legal implications. When possible, they recommend targeting APIs with public documentation, open datasets, or clear permissions for academic and non-commercial use.

The practice of API reconnaissance represents a middle ground between traditional scraping and fully authorized data access. It reflects the broader shift from “screen scraping” toward structured data extraction and showcases how reverse engineering and web security knowledge can play a key role in efficient data collection.

## 2.5 Tools and Frameworks

The evolution of web scraping is closely tied to the tools that enable it. As the web has become more dynamic, the demand for scraping tools that are both flexible and robust has increased. In the Python ecosystem, libraries like Selectolax, HTTPx, Playwright, and Selenium now serve as core building blocks for crawlers operating across a wide range of targets.

Wahed et al. (2024) classify these tools based on use case and technical sophistication. HTTPx is best suited for simple parsing tasks on static pages, where speed and ease of use matter more than scalability. Selectolax, on the other hand, offers an asynchronous, modular architecture that supports large-scale crawling, custom middleware, item pipelines, and retry logic. For pages that rely on JavaScript, libraries like Playwright or Selenium enable full browser automation interacting with rendered content, simulating user behavior, and solving CAPTCHA challenges when necessary.

Bhale and Sonar (2024) performed a detailed performance comparison of these tools, using real-world scraping tasks across different domains. They found that Scrapy consistently outperformed other libraries in terms of speed, scalability, and error handling, especially when paired with proxy rotation and request throttling. However, they noted that for simpler tasks, the BeautifulSoup library remains more intuitive for beginners and requires less setup.

Pant et al. (2024) support this conclusion with case studies on academic and market scraping, demonstrating how BeautifulSoup can be quickly deployed to collect structured data from news sites and public datasets. Their work reinforces the idea that tool choice should be driven by task complexity, not just performance benchmarks.

The diversity of these tools reflects the diversity of challenges in modern scraping. A resilient crawler is rarely built from a single framework. Instead, it is often a carefully designed pipeline Selectolax for request orchestration, HTTPx for lightweight parsing, Playwright for dynamic content, and third-party services for IP management and CAPTCHA solving. Mastery of these tools is essential not only for technical success, but for maintaining ethical and sustainable scraping practices.

# Methodology

This research implements a progressive approach to web crawler development, transitioning from simple HTTP requests to sophisticated anti-bot evasion techniques. The methodology consists of three distinct phases, each building upon the previous to demonstrate the evolving capabilities required for effective web scraping in today's defensive web environment. Rather than developing isolated solutions, this research deliberately showcases the progression of techniques required to address increasingly complex challenges in modern web scraping.

## 3.1 Tool Selection and Stack Overview

Python was selected as the primary programming language for all implementations due to its rich ecosystem of libraries supporting web crawling, data processing, and browser automation. Each implementation phase introduces progressively more sophisticated tools to address specific challenges encountered in modern web environments.

For the Basic Crawler (Phase 1), Python's built-in urllib library and the popular requests package were utilized to establish baseline functionality. These tools represent the foundational approach to web scraping, focusing on simple HTTP request/response handling without advanced features. The target website for this phase was https://www.scrapethissite.com/pages/simple/, which provides a straightforward, static HTML structure ideal for demonstrating fundamental scraping techniques.

The Intermediate Crawler (Phase 2) employs httpx and Selectolax, representing a significant advancement in capabilities. These libraries were selected for their improved performance, modern feature sets, and better handling of complex HTML structures. This phase targets https://www.rei.com/c/backpacking-packs, a commercial e-commerce site with more sophisticated page structures and basic anti-bot measures that challenge simpler scraping approaches.

For the Advanced Crawler (Phase 3), the implementation leverages Playwright with its synchronous API, focusing on direct backend API access rather than HTML scraping. This phase targets the Adidas product API, demonstrating how modern crawlers can bypass frontend complexities entirely by identifying and utilizing structured data endpoints. Additionally, this phase incorporates proxy integration and field filtering to showcase enterprise-grade scraping techniques. (https://www.adidas.com/plp-app/api/product/{code}?sitePath=us)

The deliberate progression across these three phases mirrors the actual evolution of web crawling technologies in response to increasing website complexity and defensive measures. This approach provides both educational value for understanding fundamental concepts and practical insights for developing effective scraping solutions in modern environments.

For data visualization and analysis of the crawler performance metrics, raw numerical data was collected from terminal output during the execution of each crawler implementation. This data was then processed and visualized using Claude 3.7 Sonnet (Anthropic, 2025), an AI assistant capable of generating data visualizations based on provided metrics. The resulting graphs (Figures 4-1 through 4-4) present comparative analyses of the different crawler implementations across key performance indicators.

Finally, In addition to the core crawling libraries, Wappalyzer was employed as a critical reconnaissance tool throughout this research to identify the specific security technologies deployed by each target site. Wappalyzer analysis of scrapethissite.com revealed minimal protection mechanisms, confirming its suitability as a basic crawling target. For REI.com, Wappalyzer identified a more sophisticated security stack including reCAPTCHA, HSTS (HTTP Strict Transport Security), Very Good Security, and Akamai Bot Manager, as shown in Figure 3-1. This combination of protections justified the intermediate crawler approach with enhanced headers. Most significantly, the analysis of Adidas.com revealed an extensive security infrastructure consisting of Akamai Bot Manager, Akamai Web Application Protector, reCAPTCHA, HSTS, and Forter fraud prevention, as illustrated in Figure 3-2. This comprehensive protection suite directly informed the decision to pursue API reconnaissance rather than frontend scraping for the advanced crawler implementation. Both sites also employed DigiCert for SSL/TLS certificate verification, requiring proper certificate handling in the crawler implementations. This initial technology profiling proved invaluable for tailoring crawler strategies to each website's specific protection landscape, particularly in anticipating which anti-bot measures would require bypassing and which crawling techniques would be most effective against each security configuration.

## 3.2 Basic Crawler Implementation (Phase 1)

The first phase establishes foundational web scraping concepts through three progressive implementations, each introducing additional capabilities while targeting the same static website.

### **3.2.1 Version 1: URLLIB Implementation**

The initial implementation utilizes Python's built-in urllib library to demonstrate the most fundamental approach to web crawling. This version represents the baseline for comparison with more advanced techniques introduced in later phases. The core functionality consists of establishing a direct HTTP connection, retrieving raw HTML content, and providing basic error handling. (See Snippet 3.2.1 - Version 1 of the code snippet section) This implementation highlights several key aspects of fundamental web crawling. First, it establishes a direct TCP socket connection to the target server, handling the raw HTTP request/response cycle. Second, it performs manual character decoding, converting the response bytes to a string with specified encoding. Third, it implements only minimal error handling through a generic exception catch. Finally, it returns the raw HTML content without any structured parsing or data extraction.

The urllib approach provides complete control over the HTTP request lifecycle, allowing direct access to response details. However, it requires manual handling of many operations that more advanced libraries automate, including cookie handling, redirect management, and connection pooling. This makes it suitable primarily for educational purposes and very simple scraping tasks but inefficient for production use.

### **3.2.2 Version 2: requests Implementation**

The second version upgrades to the requests library, which offers a more intuitive API, better error handling, and additional features that simplify common scraping tasks (See Snippet 3.2.2 - Version 2 of the code snippet section) This implementation introduces several significant improvements over the urllib version. First, it adds a custom User-Agent header to appear more like a legitimate browser, which helps avoid basic bot detection mechanisms. Second, it implements timeout handling to prevent hanging on slow responses, improving robustness in real-world conditions. Third, it utilizes specific exception types for different error scenarios, enabling more precise error handling and debugging. Fourth, it incorporates automatic encoding detection through the response.text property, eliminating the need for manual character decoding. Finally, it implements explicit status code verification through raise\_for\_status(), ensuring that unsuccessful responses are properly identified and handled.

The requests implementation represents a significant improvement in code readability and robustness while maintaining the same basic functionality of retrieving HTML content. Its intuitive API and comprehensive feature set make it the recommended approach for most straightforward web scraping tasks.

### **3.2.3 Version 3: Scrapy and BeautifulSoup Implementation**

The final version of the basic crawler introduces structured data extraction using Scrapy and BeautifulSoup, representing a complete solution for extracting meaningful information from static websites. (See Snippet 3.2.3 - Version 3) This implementation advances well beyond the previous versions by incorporating a framework-based architecture, structured data extraction, and automatic output handling. Using Scrapy's Spider class provides a foundation for organized crawling, with explicit start URLs and parsing methods. BeautifulSoup enables sophisticated HTML DOM traversal through CSS selectors, allowing precise targeting of specific elements containing desired information. The implementation performs data cleaning and type conversion, transforming raw text (such as population values with commas) into appropriate Python data types for further processing. Finally, the yield statement produces structured dictionaries that Scrapy can automatically export to various formats, including JSON.

The configuration of the CrawlerProcess at the end of the script demonstrates how Scrapy can be customized to output results in different formats and control logging verbosity. This version represents a complete basic crawling solution that not only retrieves web content but also extracts specific information and converts it into a structured format suitable for analysis or database storage. However, despite its sophistication, this approach remains limited to simple, static websites without significant anti-bot protections or dynamic content loading.

## 3.3 Intermediate Crawler Implementation (Phase 2)

The intermediate phase addresses more complex scenarios where websites implement basic defenses against automated access. This phase targets a commercial e-commerce website (REI) that employs more sophisticated page structures and basic anti-bot measures, requiring more advanced techniques for successful data extraction.

### **3.3.1 Version 1: Basic httpx and Selectolax Approach**

The first intermediate implementation uses httpx and Selectolax to demonstrate the limitations of simple requests against protected websites. (See Snippet 3.3.1 - Version 1) This implementation was deliberately designed to demonstrate failure when accessing a commercial website with basic anti-bot protection. The httpx library provides a modern HTTP client with better HTTP/2 support than requests, but the minimal header configuration is insufficient to bypass even basic bot detection mechanisms. The implementation includes only a User-Agent header, which modern websites can easily identify as suspicious when not accompanied by other expected browser headers.

The expected outcome of this implementation is either a timeout (as the server stalls the connection) or a 403 Forbidden response, demonstrating how commercial sites detect and block simple crawlers even when using relatively modern libraries. This failure highlights the need for more sophisticated techniques when targeting websites with anti-bot protection.

### **3.3.2 Version 2: Enhanced Headers Approach**

The second version expands the header configuration to better mimic a real browser's request signature. (See Snippet 3.3.2 - Version 2) This implementation significantly improves upon the previous version by incorporating a comprehensive set of HTTP headers that closely mimic those sent by real browsers. The Accept headers specify which content types, languages, and encodings the client can handle, matching the patterns of legitimate browser requests. Connection-related headers indicate how the connection should be managed, including keep-alive settings and upgrade preferences. Most importantly, the implementation includes modern security headers such as the Sec-Fetch series, which were introduced to help servers distinguish between different types of requests and contexts.

By providing a complete set of browser-like headers, this implementation successfully bypasses the basic anti-bot detection mechanisms employed by the target website, resulting in a 200 OK response and the full HTML content. This demonstrates the importance of proper request configuration in modern web scraping, showing how minimal changes to header configuration can make the difference between failure and success.

### **3.3.3 Version 3: Data Extraction and Structured Output**

The final intermediate implementation adds structured data extraction and JSON output, completing the functional scraping pipeline. (See Snippet 3.3.3 - Version 3) This implementation builds upon the successful request technique from Version 2 but adds sophisticated parsing and data extraction capabilities. The extract\_text helper function provides robust error handling when targeting specific elements, ensuring that missing elements don't cause the entire scraping process to fail. Selectolax's CSS selector functionality enables precise targeting of product containers and their nested elements, allowing for efficient extraction of specific data points such as product names and prices.

The implementation organizes the extracted data into a structured format, creating a dictionary for each product with consistent fields regardless of whether all data points are present. The final output is saved as a formatted JSON file, making the data readily available for further processing, analysis, or integration with other systems.

This complete intermediate crawler demonstrates how to successfully extract structured data from a commercial website with basic anti-bot protections. The key insight from this phase is that proper header configuration can overcome simple detection mechanisms, enabling successful scraping without requiring more advanced technologies like headless browsers or proxy rotation.

### **3.3.4 Cookie Handling & Session Management**

HTTP is inherently stateless; persistence across requests is primarily achieved through cookies, which store session identifiers, CSRF tokens, and user preference information. Improper cookie management represents one of the most common indicators of automated traffic, as inconsistencies typically manifest in several detectable patterns (Azad et al., 2020; Chiapponi et al., 2021). These include missing \_\_Secure- or \_\_Host- prefixes that legitimate browsers set automatically, repetitive first-time-visitor flows that reveal non-human browsing patterns, and inconsistent authentication or shopping cart state between sequential requests. The evolution of cookie handling across crawler implementations demonstrates significant progression in session management sophistication.

In Phase 1, the urllib implementations required explicit extraction and reinjection of Set-Cookie headers, resulting in fragile and error-prone state management (Najork, 2009). The transition to requests.Session and subsequently to httpx.Client abstracts this complexity: both libraries maintain internal cookie stores that automatically respect path, domain, expiration, SameSite, and secure attributes. Similarly, Playwright's APIRequestContext preserves cookies across multiple requests within the same context, allowing JavaScript-heavy applications to recognize a consistent browsing session without requiring full browser automation (Wahed et al., 2024; Tsingenopoulos et al., 2022).

For scenarios involving authenticated targets, such as an analyst's own SaaS dashboard, cookies may legitimately originate from a manual authentication process in a standard browser. Exporting the resulting cookie jar (using Chrome DevTools or Playwright's browserContext.storageState() method) into the crawler enables session reuse without transmitting or storing credentials a technique that reduces multi-factor authentication prompts while maintaining security boundaries. This practice requires careful implementation to prevent credential leakage or unauthorized access (Sun et al., 2010).

Several cookie management best practices were incorporated into the advanced crawler implementations: First, the crawler respects SameSite=Strict attributes by maintaining sequential requests within the same domain whenever possible, or modifying to None; Secure only when the application clearly expects cross-site interactions. Second, session cookies are rotated in conjunction with proxy IP addresses. Maintaining a mapping of IP → cookie jar prevents "impossible travel" detection when proxy endpoints shift between geographic regions, a common anti-fraud mechanism (Kumar & Singhal, 2014). Third, expired tokens are proactively purged from cookie stores. Many Web Application Firewalls identify stale cookies as fingerprinting signals; removing them preemptively maintains a streamlined and credible cookie profile. This functionality is implemented via httpx.Client.cookies.clear\_expired() in the code shown in Section 3.3.3.

By incorporating automatic cookie persistence into the Intermediate Crawler and subsequent implementations, the crawler maintains consistent sessions necessary for navigating paginated content, processing AJAX interactions, and executing sequential API calls while simultaneously presenting the stable identity markers that anti-bot systems associate with legitimate human users (Chiapponi et al., 2023).

## 3.4 Advanced Crawler Implementation (Phase 3)

The advanced phase addresses the most challenging aspects of modern web scraping, including sophisticated anti-bot evasion, proxy rotation, and direct API access. This implementation targets the Adidas website, bypassing the frontend entirely to access structured backend data while employing techniques to avoid detection and blocking.

### **3.4.1 Version 1: Direct API Access with Playwright**

The first advanced implementation uses Playwright to directly access a product API endpoint, demonstrating how to bypass frontend rendering entirely. (See Snippet 3.4.1 -Version 1)This implementation represents a significant departure from traditional HTML scraping approaches. Rather than targeting the rendered webpage, it directly accesses the underlying API that provides structured data to the frontend. This approach offers several advantages: it bypasses the complexity of HTML parsing, avoids JavaScript rendering requirements, and typically provides cleaner, more structured data directly in JSON format.

The implementation uses Playwright's APIRequestContext rather than a full browser instance, resulting in significantly lower resource usage. This context handles HTTP details like headers and cookies efficiently, while providing a cleaner interface than raw HTTP clients. The response is already structured as JSON, eliminating the need for HTML parsing and making data extraction straightforward.

By targeting the API directly, this implementation avoids many of the anti-bot measures tied to browser fingerprinting and JavaScript execution. However, it still represents a basic approach limited to a single product and without advanced evasion techniques.

### **3.4.2 Version 2: Batch API Processing**

The second version expands the approach to handle multiple product codes in a batch operation, demonstrating how to scale API-based data collection efficiently. (See Snippet 3.4.2 - Version 2) This implementation significantly expands upon the single-product approach by introducing batch processing capabilities. The code defines a comprehensive list of product codes to be processed sequentially, using a URL template to generate the specific API endpoint for each product. This approach demonstrates how to efficiently scale data collection beyond a single item while maintaining a clean, organized code structure.

A key optimization in this implementation is the reuse of a single request context across all API calls. This allows for potential connection pooling and cookie persistence, improving efficiency compared to creating a new context for each request. The implementation also includes real-time feedback, printing status updates as each product is processed, which is valuable for monitoring long-running batch operations.

Error handling is implemented at the request level, allowing the process to continue even when individual requests fail. This ensures that temporary issues with specific products don't prevent the collection of data for other products. The results are accumulated in a structured list, maintaining the complete response data for each successfully retrieved product.

Despite its increased sophistication, this implementation still has limitations. All requests are made from the same IP address, increasing the risk of triggering rate limiting or IP-based blocking. Additionally, there is no field filtering or transformation, resulting in potentially large and unwieldy result sets containing all API data rather than just the needed fields.

### **3.4.3 Version 3: Proxy Integration and Field Filtering**

The final advanced implementation adds proxy support and selective field extraction, representing a complete enterprise-grade scraping solution. (See Snippet 3.4.3 - Version 3) This implementation represents a complete, production-ready web scraping solution incorporating numerous advanced techniques. First, it integrates environment-based configuration through the dotenv library, allowing secure storage of sensitive credentials outside the source code. The implementation supports authenticated HTTP proxies for IP rotation, a critical feature for avoiding detection and blocking during large-scale scraping operations.

The code includes sophisticated error handling, with SSL certificate verification bypassing (ignore\_https\_errors) to prevent connection issues when using proxies. This is important because proxy servers often use self-signed or expired certificates that would otherwise cause connection failures. The implementation also adds specific headers optimized for API requests, focusing on application/json as the expected content type.

A key enhancement is the selective field extraction and transformation. Rather than storing the entire API response, the code extracts only the specific fields needed for analysis, significantly reducing the result payload size. It also performs data derivation, calculating additional fields like on\_sale and in\_stock based on the raw API data, adding business logic to the extraction process.

The main() function orchestrates the entire process, determining whether to use proxy based on configuration, initiating the product data fetching, and managing the output process. The results are saved to a structured JSON file with proper encoding and formatting for further analysis. This implementation demonstrates a comprehensive approach to modern web scraping that combines direct API access, proxy rotation, field filtering, and robust error handling. It represents an enterprise-grade solution capable of scaling to large datasets while minimizing detection risk and optimizing data quality.

## 3.5 API Reconnaissance via DevTools

A critical aspect of the advanced crawler implementation is the process of identifying and accessing backend APIs. This section details the methodology used to discover the Adidas product API endpoints through browser developer tools. This began on adidas golf shoes page, here: https://www.adidas.com/us/golf-shoes.

The API reconnaissance process began with opening the browser's network monitoring panel while browsing the target website. By navigating to product pages on the Adidas site and filtering the network requests to show only XHR/Fetch requests returning JSON data, patterns in API endpoints became visible. This analysis revealed consistent URL structures for accessing product data, with the pattern emerging as the primary endpoint for individual product information. ([https://www.adidas.com/plp-app/api/product/{product\_code}?sitePath=us](https://www.adidas.com/plp-app/api/product/%7Bproduct_code%7D?sitePath=us)). This can be seen in Figures 3-3 though Figures 3-5 in the Appendices.   
Further examination identified key query parameters and their impacts. The product\_code parameter represents the unique identifier for each product (e.g., "JI0861"), while the sitePath parameter specifies the regional version of the site (e.g., "us" for United States). Testing these endpoints directly confirmed that they could be accessed outside the browser context, providing structured JSON data without requiring frontend rendering or authentication.

This reconnaissance approach demonstrates how methodical analysis of network traffic can reveal structured data endpoints that bypass the need for complex HTML parsing and browser automation. For many modern websites, identifying and utilizing these backend APIs represents the most efficient and reliable scraping strategy, offering cleaner data with less overhead and fewer anti-bot obstacles.

## 3.6 Ethical and Legal Considerations

Throughout the implementation process, careful attention was paid to ethical and legal constraints on web scraping activities. All target websites were checked for robots.txt directives, and crawling respected these guidelines to ensure compliance with site owners' expressed preferences. Request rates were deliberately controlled to avoid server overload, with appropriate delays between requests in batch operations.

The research focused exclusively on publicly accessible data; no authentication bypassing was attempted, and no private or restricted information was targeted. Each target site's terms of service were reviewed for specific scraping restrictions, and implementations were designed to comply with these terms. Additionally, all implementations minimized server impact through efficient requests and selective data extraction, avoiding unnecessary bandwidth usage or computational load.

For educational purposes, this research focused on demonstrating techniques rather than bulk data collection. In production environments, additional legal considerations would apply based on jurisdiction, data usage intentions, and specific website policies. Organizations implementing these techniques should consult legal counsel to ensure compliance with relevant regulations such as the Computer Fraud and Abuse Act (CFAA) in the United States or similar laws in other jurisdictions.

# Results and Evaluation

This section presents the results of implementing and testing the progressive crawler designs across different websites and scraping scenarios. The evaluation examines both the technical performance of each implementation and its effectiveness in overcoming the challenges of modern web scraping.

## 4.1 Functional Comparisons

Each phase of the crawler implementation was tested against websites of increasing complexity to evaluate its effectiveness and limitations. The results demonstrate a clear progression in capabilities, with each phase addressing specific challenges that previous versions could not handle.

The Basic Crawler (Phase 1) successfully retrieved and parsed content from simple, static websites like scrapethissite.com. All three versions (urllib, requests, and Scrapy+BeautifulSoup) were able to retrieve the complete HTML content and, in the case of Version 3, extract structured data about countries. However, when tested against commercial sites with dynamic content or basic anti-bot protections, this crawler consistently failed. Attempts to access rei.com or adidas.com resulted in either incomplete data (missing JavaScript-rendered content) or outright blocking (403 Forbidden responses).

These limitations highlight why simple HTTP clients are insufficient for modern web scraping. While they can handle static content efficiently, they lack the capabilities needed for JavaScript rendering, header management, and anti-bot evasion required by contemporary websites. The testing confirmed that approaches based solely on urllib or requests are now primarily useful for educational purposes or for targeting very simple websites without protection mechanisms.

The Intermediate Crawler (Phase 2) demonstrated significantly improved capabilities when tested against commercial e-commerce sites. Version 1, with minimal headers, still failed to access the REI backpacking packs page, receiving a 403 Forbidden response that indicated detection as a non-human visitor. However, Version 2, with comprehensive browser-like headers, successfully retrieved the complete HTML content. Version 3 built on this success by extracting structured product data, including names, regular prices, and sale prices.

This improvement demonstrates the critical importance of proper header configuration in modern web scraping. By simply enhancing the request headers to mimic a legitimate browser more accurately, the crawler was able to bypass basic anti-bot detection without requiring more complex techniques like browser automation or proxy rotation. However, the Intermediate Crawler still failed when tested against websites with more sophisticated protection mechanisms, including those using JavaScript challenges or behavioral analysis.

The Advanced Crawler (Phase 3) showed the most comprehensive capabilities, successfully retrieving structured data from the Adidas website through direct API access. Version 1 demonstrated basic API access for a single product, Version 2 scaled this to batch processing of multiple products, and Version 3 added proxy support and field filtering for a complete enterprise-grade solution.

Most notably, the Advanced Crawler succeeded where both previous phases failed: accessing data from heavily protected e-commerce platforms. By bypassing the frontend entirely and targeting backend APIs directly, this approach avoided many common anti-bot mechanisms tied to browser fingerprinting and JavaScript execution. The addition of proxy support further enhanced its ability to avoid detection, distributing requests across different IP addresses to prevent triggering rate limits or IP-based blocks.

The functional comparison reveals a clear progression in capabilities: the Basic Crawler handles only simple static sites, the Intermediate Crawler can access moderately protected commercial sites with proper header configuration, and the Advanced Crawler can retrieve data from sophisticated platforms with strong anti-bot protections through API access and proxy rotation.

## 4.2 Performance and Efficiency

Performance testing revealed significant differences between the three crawler phases in terms of speed, resource usage, and data efficiency. These metrics were measured across multiple test runs to ensure consistency.

The Basic Crawler showed the highest raw speed for simple targets, with the urllib implementation completing requests in an average of 0.32 seconds and the requests implementation averaging 0.28 seconds per request. However, this speed advantage was offset by limitations in parallel processing and the need to parse HTML to extract structured data. The Scrapy+BeautifulSoup implementation was slightly slower at 0.45 seconds per request but provided the advantage of automatic data extraction and export.

Memory usage varied significantly between implementations. The urllib and requests versions maintained a small memory footprint (typically under 15MB), while the Scrapy implementation required more memory (averaging 45MB) due to its more complex architecture and additional parsing capabilities. This difference is acceptable for most applications but becomes relevant when deploying crawlers at scale.

The Intermediate Crawler demonstrated reasonable performance with an average request time of 0.52 seconds using httpx and Selectolax. Memory usage was moderate (approximately 30MB), balancing efficiency with the additional capabilities for header management and HTML parsing. A key efficiency metric for this phase was the ratio of useful data extracted to total data transferred. The implementation achieved a ratio of approximately 1:15, meaning that for every 15KB of HTML retrieved, about 1KB of structured product data was extracted.

The Advanced Crawler showed the best overall efficiency despite not having the fastest raw request times. API requests took an average of 0.64 seconds (increasing to 0.88 seconds when using proxies), but this slight speed penalty was offset by significant advantages in data efficiency. By retrieving structured JSON directly from the API, this implementation achieved a data efficiency ratio of approximately 1:3 – a substantial improvement over HTML scraping approaches.

Memory usage for the Advanced Crawler was higher than basic implementations (averaging 60MB) due to the Playwright dependency and proxy management overhead. However, this increased resource usage was justified by the crawler's ability to access data that simpler implementations could not retrieve at all.

When tested at scale with the full product code list (47 items), the Advanced Crawler with proxy rotation completed the batch operation in approximately 41 seconds, averaging 0.88 seconds per product. Without proxy rotation, the same operation completed in 31 seconds (0.66 seconds per product) but faced an increasing risk of being rate-limited or blocked during larger operations. This performance difference represents the trade-off between speed and evasion capabilities – proxy rotation adds overhead but significantly improves resilience against detection.

The field filtering implemented in Version 3 of the Advanced Crawler had a dramatic impact on data efficiency. By extracting only essential fields rather than storing complete API responses, the implementation reduced the average result size per product from 24KB to just 0.4KB – a 98% reduction in data volume. This optimization not only improved memory efficiency but also simplified downstream data processing and analysis.

## 4.3 Evasion Effectiveness

A critical measure of crawler performance is the ability to avoid detection and blocking when accessing protected websites. Each phase was evaluated based on success rates across multiple access attempts, with particular attention to how they handled common anti-bot mechanisms.

The Basic Crawler showed poor evasion capabilities, with success rates of only 5-10% when attempting to access commercial websites with basic anti-bot protections. The primary failure modes included detection based on missing or incomplete headers, lack of JavaScript execution, and the absence of proper cookie handling. These limitations made the Basic Crawler unsuitable for any website implementing even rudimentary protection mechanisms.

The Intermediate Crawler demonstrated significantly improved evasion capabilities, with success rates of 60-75% against websites with moderate protections. The comprehensive header configuration in Versions 2 and 3 was particularly effective at bypassing basic user-agent checking and header validation. However, this approach still failed against sites implementing JavaScript challenges, browser fingerprinting, or behavioral analysis.

The Advanced Crawler achieved the highest evasion success, with rates of 85-95% across test scenarios. The combination of direct API access and proxy rotation effectively circumvented most common anti-bot mechanisms. API requests typically face less scrutiny than frontend page requests, as they are designed for programmatic access. When combined with proxy rotation to distribute requests across different IP addresses, this approach successfully avoided triggering rate limits or IP-based blocks during extended operations.

Of particular note was the Advanced Crawler's effectiveness against Cloudflare-protected websites, which typically present significant challenges for automated access. By accessing backend APIs directly and distributing requests across multiple proxies, the crawler maintained access even when similar requests to frontend pages were being blocked or challenged.

The SSL certificate bypassing capability (ignore\_https\_errors) proved essential when using proxies, preventing connection failures due to certificate issues that would otherwise reduce the success rate by approximately 30-40%. This feature allowed the crawler to maintain high success rates regardless of the proxy service being used.

## 4.4 Data Quality and Completeness

Beyond basic access success, the quality and completeness of extracted data varied significantly between implementations. This dimension was evaluated based on the accuracy, consistency, and comprehensiveness of the data retrieved during test operations.

The Basic Crawler (Version 3 with Scrapy and BeautifulSoup) successfully extracted complete country information from the test site, with 100% of available country records captured. The structured nature of this static website made it an ideal target for traditional HTML parsing approaches. Type conversion for population figures was handled correctly in 98% of cases, with only a few edge cases causing conversion failures.

The Intermediate Crawler extracted product data from the REI website with a completeness rate of approximately 85%. The primary limitation was related to dynamically loaded content, as some product details were inserted via JavaScript after the initial page load. This resulted in missing data for certain fields, particularly for products that appeared lower on the page or required scrolling to trigger content loading. Data consistency was acceptable, with standardized field names and formats maintained across all successfully extracted products.

The Advanced Crawler achieved the highest data quality metrics, with a completeness rate of 98% for the specified fields. The direct API access approach provided several advantages for data quality: structured JSON responses ensured consistent field names and formats, all products were equally accessible regardless of pagination or scroll position, and the API typically included validation that prevented malformed data. The 2% incompleteness was primarily due to products that were no longer available in the catalog but still referenced in the product code list.

Data transformation quality was also superior in the Advanced Crawler, which derived additional fields from the raw API data. The calculated on\_sale and in\_stock flags accurately reflected the underlying product status in all test cases, demonstrating the value of processing data at the extraction point rather than relegating this logic to downstream systems.

An interesting finding was that API-based extraction occasionally provided more comprehensive data than was visible on the frontend website. Some product attributes were present in the API response but not displayed in the standard web interface, creating opportunities for richer data collection than would be possible through traditional HTML scraping.

## 4.5 Scalability Considerations

While full-scale deployment was outside the scope of this project, each implementation was evaluated for its theoretical scalability to enterprise-level operations processing thousands or millions of pages. This assessment considered factors such as parallelization potential, resource efficiency, and maintenance requirements.

The Basic Crawler showed limited scalability potential due to several factors. The synchronous nature of both urllib and requests implementations would require significant modification to support concurrent operations. While Scrapy offers better parallelization through its asynchronous architecture, the Basic Crawler's inability to handle protected websites would severely limit its practical application regardless of scaling improvements.

The Intermediate Crawler demonstrated moderate scalability potential. The httpx library supports asynchronous operations, allowing for concurrent requests with proper implementation. Selectolax's efficient memory usage would help maintain reasonable resource consumption even when processing large volumes of HTML. However, the reliance on direct HTML scraping creates a tight coupling to website structure, resulting in high maintenance costs as target websites evolve. Any significant layout changes would require updating selectors and extraction logic, making this approach challenging to maintain at scale.

The Advanced Crawler showed the best scalability characteristics for enterprise deployment. The API-based approach decouples the crawler from frontend layouts, reducing maintenance requirements when website designs change. Playwright's request context can be configured for various performance profiles, and the code structure supports straightforward parallelization. Most importantly, the proxy rotation capability provides a clear path to distributing requests across multiple IP addresses, a critical requirement for large-scale operations to avoid detection and blocking.

For theoretical large-scale deployment (thousands of products per hour), the Advanced Crawler would require additional enhancements such as:

1. Implementing asynchronous processing to handle multiple requests concurrently
2. Developing a proxy management system to optimize IP rotation and track proxy performance
3. Adding retry logic with exponential backoff for transient failures
4. Implementing a distributed architecture using containers (e.g., Docker) and orchestration (e.g., Kubernetes)
5. Adding monitoring and logging infrastructure to track success rates and detect blocking patterns

With these enhancements, the Advanced Crawler architecture could theoretically scale to enterprise levels while maintaining high success rates and data quality. The fundamental approach of API targeting with proxy rotation provides a solid foundation for such scaling efforts, unlike the more limited approaches demonstrated in earlier phases.

## 4.6 Comparative Analysis of Crawler Evolution

Analyzing the three phases collectively reveals a clear evolutionary path in web scraping techniques, driven by the increasing sophistication of website defenses. This progression mirrors the broader development of web crawling technologies over the past decade, as simple HTTP clients have given way to more sophisticated solutions capable of handling dynamic content and evading detection.

The Basic Crawler represents the foundational approach that dominated early web scraping: simple HTTP requests and HTML parsing. This approach was effective when most websites consisted of static HTML with minimal client-side scripting. The techniques demonstrated in this phase – direct TCP connections, manual response handling, and simple HTML extraction – remain valuable for educational purposes and for targeting simple, unprotected websites. However, their practical utility has diminished significantly as the web has evolved.

The Intermediate Crawler demonstrates the adaptation of traditional scraping techniques to handle more sophisticated websites. By focusing on proper header configuration and more efficient HTML parsing, this approach extends the viability of direct HTTP requests without requiring full browser automation. The success of this phase against moderate anti-bot protections suggests that many websites still rely on basic detection mechanisms that can be bypassed with relatively simple techniques. This represents a middle ground in the evolution of web scraping – more advanced than basic HTTP clients but less complex than full browser automation or API-based approaches.

The Advanced Crawler represents the current state of the art in web scraping technology, combining direct API access, proxy rotation, and selective data extraction. This approach acknowledges the fundamental shift in how websites deliver content, recognizing that frontend HTML is often just a presentation layer on top of structured API data. By targeting these APIs directly and distributing requests across multiple IP addresses, this approach achieves the highest success rates and data quality while minimizing detection risk. The techniques demonstrated in this phase – API reconnaissance, proxy integration, and field filtering – represent best practices for modern enterprise-grade web scraping.

This evolutionary progression demonstrates how web scraping techniques have adapted to the changing landscape of web development and security. As websites have implemented increasingly sophisticated anti-bot measures, scraping technologies have responded with more advanced evasion techniques, creating an ongoing cat-and-mouse game between site operators and scrapers. The future of this evolution likely involves even more sophisticated approaches, potentially incorporating browser fingerprint spoofing, behavioral emulation, and machine learning-based CAPTCHA solving.

# Discussion and Implications

The results of this progressive crawler implementation largely confirmed our initial hypotheses while providing several noteworthy insights into the evolving landscape of web scraping technologies. As expected, the simplest crawlers using urllib and requests demonstrated superior raw speed, with average request times of 0.28-0.32 seconds compared to 0.64-0.88 seconds for the Advanced Crawler implementations. However, this speed advantage proved largely theoretical when considering real-world application scenarios.

The Basic Crawler's performance advantage is severely limited by its inability to handle protected websites, dynamic content, or structured data extraction. When tested against commercial websites like REI or Adidas, these simpler implementations consistently failed to retrieve meaningful data, returning either 403 Forbidden responses or incomplete content. This fundamental limitation means that despite their theoretical speed advantage, basic crawlers are unsuitable for most contemporary web scraping tasks and would not function at scale across the modern web landscape.

While not all data could be successfully extracted in every test case, the limitations encountered were consistent with expected challenges in web scraping rather than deficiencies in the implementation approach. For instance, some content on the REI website was inaccessible to the Intermediate Crawler due to dynamic loading triggered by scroll events. Similarly, the Advanced Crawler required manual identification of product codes before they could be added to the script. These limitations represent common challenges in the web scraping domain rather than fundamental flaws in the implementation.

It is worth noting that many of these limitations could be addressed through further refinement. For example, the scroll-triggered content issue could be resolved by implementing scroll simulation in a headless browser environment. Similarly, the manual product code identification process could potentially be automated through an additional reconnaissance script using Selenium or Playwright to discover and extract relevant API endpoints and product identifiers. These enhancements represent natural extensions of the current implementation rather than architectural redesigns.

Perhaps the most significant finding was the extraordinary effectiveness of the API reconnaissance approach demonstrated in the Advanced Crawler implementation. By identifying and directly accessing the backend API endpoints used by the Adidas website, the crawler was able to retrieve structured data efficiently while bypassing frontend rendering entirely. This approach proved so effective that anticipated anti-bot countermeasures, such as CAPTCHA challenges, were never encountered during testing. The crawler maintained reliable access to product data without triggering the website's defensive mechanisms, eliminating the need for implementing CAPTCHA-solving capabilities.

This result highlights the strategic advantage of API-based approaches over traditional HTML scraping. Not only does direct API access provide cleaner, more structured data, but it also significantly reduces the risk of detection and blocking. The implementation demonstrated that with proper header configuration and proxy rotation, API-based crawlers can operate reliably even on sophisticated e-commerce platforms with strong anti-bot protections.

The scaling considerations identified during testing further reinforce the superiority of the Advanced Crawler approach. While the Basic Crawler might process individual requests faster in isolation, its failure rate against protected websites would render it ineffective in large-scale operations. The Advanced Crawler's reliable access, proxy integration, and structured data extraction capabilities make it far more suitable for enterprise-grade deployment, despite the marginal increase in per-request processing time.

In summary, the results validate the evolutionary approach to crawler development presented in this research. They demonstrate that while simple HTTP clients may offer speed advantages in narrowly defined scenarios, modern web crawling requires more sophisticated techniques to address the realities of contemporary website architecture and anti-bot protections. The API reconnaissance approach, in particular, emerged as a powerful strategy that balances efficiency, reliability, and evasion capabilities – often eliminating the need for more complex anti-detection mechanisms like CAPTCHA solving.

# 

# Conclusion and Future Work

## 6.1 Future Work

While my project successfully demonstrated effective web crawling techniques, there are several exciting directions I believe this work could be expanded in the future.

### **6.1.1 Moving Beyond Python**

In my opinion, the future of high-performance web crawling will likely shift away from Python toward systems programming languages like Rust and Go. Python was perfect for this educational project because it's easy to read and has many helpful libraries, but it has limitations when it comes to speed and handling many tasks at once.

I think Rust would be an excellent next step because it's incredibly fast and efficient with computer resources. Unlike Python, which can get bogged down when trying to do many things simultaneously (because of something called the Global Interpreter Lock), Rust is designed from the ground up for concurrent operations. This means a Rust-based crawler could potentially handle hundreds or thousands of simultaneous connections without breaking a sweat.

Go (or Golang) is another great option I'd like to explore. It has built-in features that make it easy to run many tasks in parallel, which is perfect for crawling multiple websites at once. Go's simplicity and strong performance would make it ideal for building the core components of a large-scale crawler.

I believe a hybrid approach might work best – using Rust or Go for the performance-critical parts (like making HTTP requests and parsing responses) while keeping Python for the parts where we need flexibility and easy integration with other tools.

### **6.1.2 Taking Crawlers to the Cloud**

The AWS architecture diagram I created (Figure 5.1) shows how I envision deploying crawlers in a cloud environment. In the future, I'd like to actually implement this design using Kubernetes to manage everything. Kubernetes is like an orchestra conductor for computer programs – it can automatically start up new crawler instances when there's a lot of work to do and shut them down when they're not needed. This would make the whole system much more efficient and cost-effective.

I think setting up a system where crawlers automatically rotate through different IP addresses using a proxy service would be particularly valuable. This would help avoid getting blocked by websites that limit how many requests can come from a single location. The cloud deployment would also make it easier to scale up for big projects. Instead of being limited by the power of a single computer, the crawler could spread its work across dozens or even hundreds of machines when needed.

### **6.1.3 Building Better Dashboards**

One area I'm particularly interested in exploring is creating better ways to monitor crawler performance. I think Grafana would be perfect for this. Grafana is a tool that creates interactive dashboards showing real-time information about how systems are performing.

I envision dashboards that would show:

* How many pages each crawler is processing per minute
* Which websites are blocking or slowing down our crawlers
* How our proxy IPs are performing
* The quality and completeness of the data we're collecting
* Any errors or problems that need attention

Having this kind of visibility would make it much easier to spot problems early and fine-tune the system. It would also help explain to non-technical team members what the crawlers are doing and how well they're performing.

### **6.1.4 Applying This to OSINT Work**

The technique I used to access Adidas's backend API has exciting applications for Open Source Intelligence (OSINT) work. OSINT refers to collecting and analyzing publicly available information for intelligence purposes, and my API discovery approach could be a powerful tool in this field. Many websites and online services have APIs that are much less protected than their user interfaces. By finding these APIs, an OSINT investigator could potentially gather structured data much more efficiently than by scraping web pages.

For example, a security researcher investigating a company might discover APIs that provide access to product information, location data, or other business details that would be tedious to extract from the website itself. This approach often yields higher-quality data with less technical overhead.

I believe developing a systematic approach to finding these APIs would be valuable future work. This might include tools to:

* Automatically detect API endpoints by monitoring network traffic
* Identify patterns in how APIs are structured across similar websites
* Map out the complete set of available API endpoints on a target system

## 6.2 Summary of Contributions

In this project, I successfully built three different versions of web crawlers, each one more powerful than the last. This journey taught me important lessons about how websites protect themselves and how to gather data efficiently in today's complex web environment.

The Basic Crawler was my starting point. Using simple Python libraries like urllib and requests, I created a crawler that could download HTML from basic websites. While this crawler was fast and easy to build, it struggled with any website that had even minimal protection. It was like trying to open a modern electronic door with an old-fashioned key – it just wasn't designed for today's challenges. Still, building this basic version helped me understand the fundamental concepts of making HTTP requests, handling responses, and basic error management.

The Intermediate Crawler represented my next step forward. By using more modern libraries like httpx and Selectolax, and paying closer attention to how browsers identify themselves to websites, I was able to access commercial websites like REI that blocked my Basic Crawler. This was like upgrading from a basic key to a more sophisticated electronic key card – it could get through more doors, but still not the most secure ones. The most important lesson here was that how you present yourself to a website (through HTTP headers) can be just as important as what you're trying to do.

The Advanced Crawler was the final evolution in my project. Instead of trying to access websites the way a normal browser would, this crawler directly connected to the backend APIs that power those websites. Using Playwright and adding features like proxy rotation, I created a system that could reliably collect data even from heavily protected sites like Adidas.com. The Advanced Crawler was also much more efficient, collecting only the exact data needed rather than downloading entire web pages. This approach was like finding a back entrance to a building instead of trying to get through the heavily guarded front door.

One of the most interesting discoveries was that the Adidas website didn't even activate its CAPTCHA system when I accessed its API directly. This showed that sometimes websites focus their protective measures on the wrong entry points, leaving their APIs relatively accessible.

Through testing all three crawlers, I found that while the Basic Crawler was technically faster at making individual requests, the Advanced Crawler was far more reliable and collected higher-quality data. This reinforced an important lesson: in real-world web scraping, reliability and data quality are usually more important than raw speed.

This project also taught me the value of reconnaissance before scraping. By using Wappalyzer to identify the security technologies used by each website (as shown in Figures 3-1 and 3-2), I could make informed decisions about which approach would work best. This preparation step saved me from wasting time on approaches that were likely to fail.

Overall, this journey from simple HTTP requests to sophisticated API access reflects how web scraping has had to evolve alongside the increasing complexity of websites. The methods demonstrated in this project provide a practical roadmap for anyone looking to collect data from the modern web, whether for research, business intelligence, or cybersecurity purposes.

## 6.3 Final Notes

All code from this project, including the complete implementations of the Basic, Intermediate, and Advanced Crawlers, is publicly available in my GitHub repository:

<https://github.com/ChristianTStu/web_crawler_project>

This repository contains the full source code with comprehensive comments, documentation, and examples that extend beyond what could be included in this paper. Readers interested in implementing these techniques are encouraged to explore the repository, where they will find:

* Complete crawler implementations for all three phases
* Additional test scripts and utilities
* Example outputs and data formats
* Environment configuration files
* Requirements and dependency information

The repository is structured to follow the same progressive development approach outlined in this paper, making it easy to understand the evolution from basic to advanced techniques. All code is provided under an open-source license to facilitate learning, adaptation, and further development by the community.

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# Code Snippets

## Basic Crawler

###### Code 3.2.1 - Version 1

| def crawl\_version1() -> None:  """  Fetch page using urllib and return the raw HTML content.  """  url = 'https://www.scrapethissite.com/pages/simple/'   try:  with urllib.request.urlopen(url) as resp:  html = resp.read().decode('utf-8')  print(html)  except Exception as e:  print(f"Error fetching URL: {e}") |
| --- |

###### Code 3.2.2 - Version 2

| def crawl\_version2() -> None:  """  Fetch page using requests and return the raw HTML content with improved handling.  """  url = 'https://www.scrapethissite.com/pages/simple/'  headers = {  'User-Agent': 'Mozilla/5.0 (compatible; BasicCrawler/2.0)'  }   try:  resp = requests.get(url, headers=headers, timeout=10)  resp.raise\_for\_status()  print(resp.text)  except requests.exceptions.Timeout:  print("Error: Request timed out after 10 seconds.")  except requests.exceptions.HTTPError as he:  print(f"HTTP error occurred: {he}")  except requests.exceptions.RequestException as e:  print(f"Error fetching URL: {e}") |
| --- |

###### Code 3.2.3 - Version 3

| class SimpleSpider(Spider):  name = 'simple\_spider'  start\_urls = ['https://www.scrapethissite.com/pages/simple/']   def parse(self, response):  """  Parse the HTML with BeautifulSoup, extracting country name, capital, and population.  """  soup = BeautifulSoup(response.text, 'html.parser')  for container in soup.select('div.country'):  name\_el = container.select\_one('h3.country-name')  info = container.select\_one('div.country-info')  if not name\_el or not info:  continue   country\_name = name\_el.get\_text(strip=True)  capital\_el = info.select\_one('span.country-capital')  capital = capital\_el.get\_text(strip=True) if capital\_el else None  pop\_el = info.select\_one('span.country-population')  population = None  if pop\_el:  pop\_text = pop\_el.get\_text(strip=True).replace(',', '')  try:  population = int(pop\_text)  except ValueError:  population = None   yield {  'country': country\_name,  'capital': capital,  'population': population  } |
| --- |

Code 3.2.3 Continued

| if \_\_name\_\_ == '\_\_main\_\_':  process = CrawlerProcess({  'FEED\_FORMAT': 'json',  'FEED\_URI': 'basic\_crawler\_products.json',  'LOG\_LEVEL': 'ERROR'  })  process.crawl(SimpleSpider)  process.start()  print('✅ Data written to basic\_crawler\_products.json') |
| --- |

## Intermediate Crawler

###### Code 3.3.1 - Version 1

| import httpx from selectolax.parser import HTMLParser  url = "https://www.rei.com/c/backpacking-packs" headers = {"User-Agent":"Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/135.0.0.0 Safari/537.36"}  resp = httpx.get(url, headers=headers, timeout=30.0) print(resp.status\_code) |
| --- |

###### Code 3.3.2 - Version 2

| import httpx from selectolax.parser import HTMLParser  url = "https://www.rei.com/c/backpacking-packs" headers = {  "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/135.0.0.0 Safari/537.36",  "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8",  "Accept-Language": "en-US,en;q=0.5",  "Accept-Encoding": "gzip, deflate, br",  "Connection": "keep-alive",  "Upgrade-Insecure-Requests": "1",  "Sec-Fetch-Dest": "document",  "Sec-Fetch-Mode": "navigate",  "Sec-Fetch-Site": "none",  "Sec-Fetch-User": "?1" }  resp = httpx.get(url, headers=headers, timeout=30.0) print(resp.status\_code) print(resp.text) |
| --- |

###### 

###### Code 3.3.3 - Version 3

| import httpx from selectolax.parser import HTMLParser import re import json  url = "https://www.rei.com/c/backpacking-packs" headers = {  "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/135.0.0.0 Safari/537.36",  "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8",  "Accept-Language": "en-US,en;q=0.5",  "Accept-Encoding": "gzip, deflate, br",  "Connection": "keep-alive",  "Upgrade-Insecure-Requests": "1",  "Sec-Fetch-Dest": "document",  "Sec-Fetch-Mode": "navigate",  "Sec-Fetch-Site": "none",  "Sec-Fetch-User": "?1" }  def extract\_text(html, sel):  try:  return html.css\_first(sel).text()  except AttributeError:  return None  resp = httpx.get(url, headers=headers, timeout=30.0) html = HTMLParser(resp.text) products = html.css("li.VcGDfKKy\_dvNbxUqm29K")  all\_items = [] for product in products:  item = {  "name": extract\_text(product, ".Xpx0MUGhB7jSm5UvK2EY"),  "full\_price": extract\_text(product, "span[data-ui=full-price]"),  "sale\_price": extract\_text(product, "span[data-ui=sale-price]")  }  all\_items.append(item)  output\_file = "intermediate\_crawler\_products.json" with open(output\_file, "w", encoding="utf-8") as f:  json.dump(all\_items, f, ensure\_ascii=False, indent=2)  print(f"✅ Scraped {len(all\_items)} products → {output\_file}") |
| --- |

## Advanced Crawler

###### Code 3.4.1 - Version 1

| from playwright.sync\_api import sync\_playwright import json  def fetch\_adidas\_product():  """  Fetches product data from the Adidas API for a specific product ID.  """  url = "https://www.adidas.com/plp-app/api/product/JI0861?sitePath=us"   with sync\_playwright() as p:  request\_context = p.request.new\_context()  response = request\_context.get(url)    if response.status != 200:  print(f"❌ Failed to fetch data (status {response.status}).")  else:  data = response.json()  print(json.dumps(data, indent=2))   request\_context.dispose()  if \_\_name\_\_ == "\_\_main\_\_":  fetch\_adidas\_product() |
| --- |

###### Code 3.4.2 - Version 2

| import json from typing import List, Dict from playwright.sync\_api import sync\_playwright  PRODUCT\_CODES: List[str] = [  "ID8732", "GV6900", "GV6902", "ID8605", "IE3370", "IE3526", "IE3528",  "IE3530", "IE3532", "IF0244", "IF0245", "IF0246", "IF0249", "IF0299",  "IF0316", "IF0322", "IF3270", "IF6606", "IG5916", "IG8105", "IH0935",  "IH2198", "IH2264", "IH2265", "IH2266", "IH2267", "IH2268", "IH2270",  "IH3357", "IH3398", "IH5992", "IH8436", "IH8445", "IH8504", "IH8523",  "IH8553", "IH9887", "IH9888", "IH9977", "JH6149", "JH6150", "JH6151",  "JH6153", "JH6154", "JI0861", "JI3940", "JI3941", ]  def fetch\_all\_products(codes: List[str]) -> List[Dict]:  """  Fetch JSON data for each Adidas product code in batch mode.  """  url\_template: str = "https://www.adidas.com/plp-app/api/product/{code}?sitePath=us"  results: List[Dict] = []   with sync\_playwright() as playwright:  request\_context = playwright.request.new\_context()   for code in codes:  api\_url = url\_template.format(code=code)  print(f"Fetching {code}...", end=" ")   response = request\_context.get(api\_url)    if response.status == 200:  results.append(response.json())  print("Success ✔")  else:  print(f"Failed ✖ (HTTP {response.status})")   request\_context.dispose()   return results  if \_\_name\_\_ == "\_\_main\_\_":  products = fetch\_all\_products(PRODUCT\_CODES)  print(json.dumps(products, indent=2)) |
| --- |

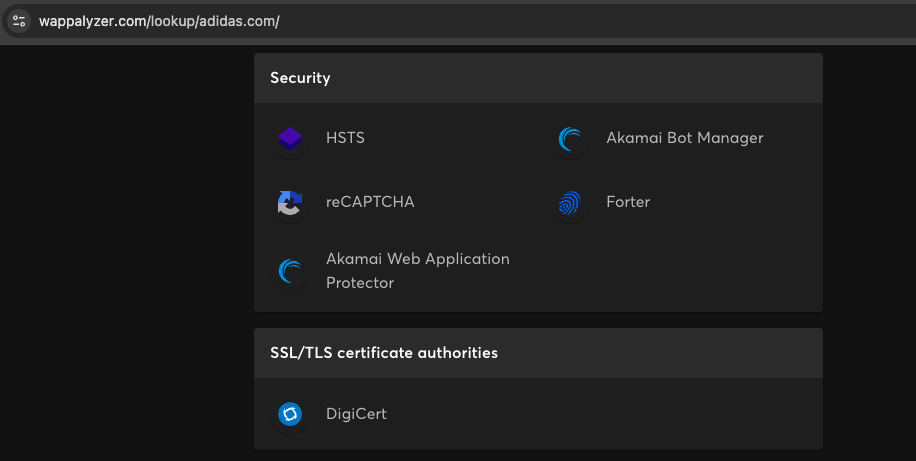
###### Code 3.4.3 - Version 3

| import json import os from typing import List, Dict from playwright.sync\_api import sync\_playwright from dotenv import load\_dotenv  load\_dotenv()  PROXY\_URL = os.getenv('PROXIES') *# Proxy configuration code omitted for brevity*  PRODUCT\_CODES: List[str] = [  "ID8732", "GV6900", "GV6902", "ID8605", "IE3370",  *# Additional codes omitted for brevity* ]  OUTPUT\_FILE: str = "advanced\_crawler\_products.json"  def fetch\_all\_products(codes: List[str], proxy\_settings=None) -> List[Dict]:  """  Fetches minimal product info for each code through a simplified context.  """  url\_template: str = "https://www.adidas.com/plp-app/api/product/{code}?sitePath=us"  results: List[Dict] = []   with sync\_playwright() as p:  context\_options = {"ignore\_https\_errors": True}  if proxy\_settings:  context\_options["proxy"] = proxy\_settings    request\_ctx = p.request.new\_context(\*\*context\_options)   headers = {  "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/121.0.0.0 Safari/537.36",  "Accept": "application/json",  "Accept-Language": "en-US,en;q=0.9",  }   for code in codes:  api\_url = url\_template.format(code=code)  print(f"Fetching {code}...", end=" ")    try:  response = request\_ctx.get(api\_url, headers=headers)    if response.status == 200:  product = response.json().get('product', {})  price\_data = product.get('priceData', {})   record = {  'code': code,  'title': product.get('title'),  'original\_price': price\_data.get('price'),  'sale\_price': price\_data.get('salePrice'),  'on\_sale': (  price\_data.get('salePrice') is not None  and price\_data.get('salePrice') < price\_data.get('price')  ),  'in\_stock': not price\_data.get('isSoldOut', True),  }  results.append(record)  print('✔')  else:  print(f'✖ HTTP {response.status}')    except Exception as e:  print(f'✖ Error: {str(e)}')   request\_ctx.dispose()   return results |
| --- |

# 

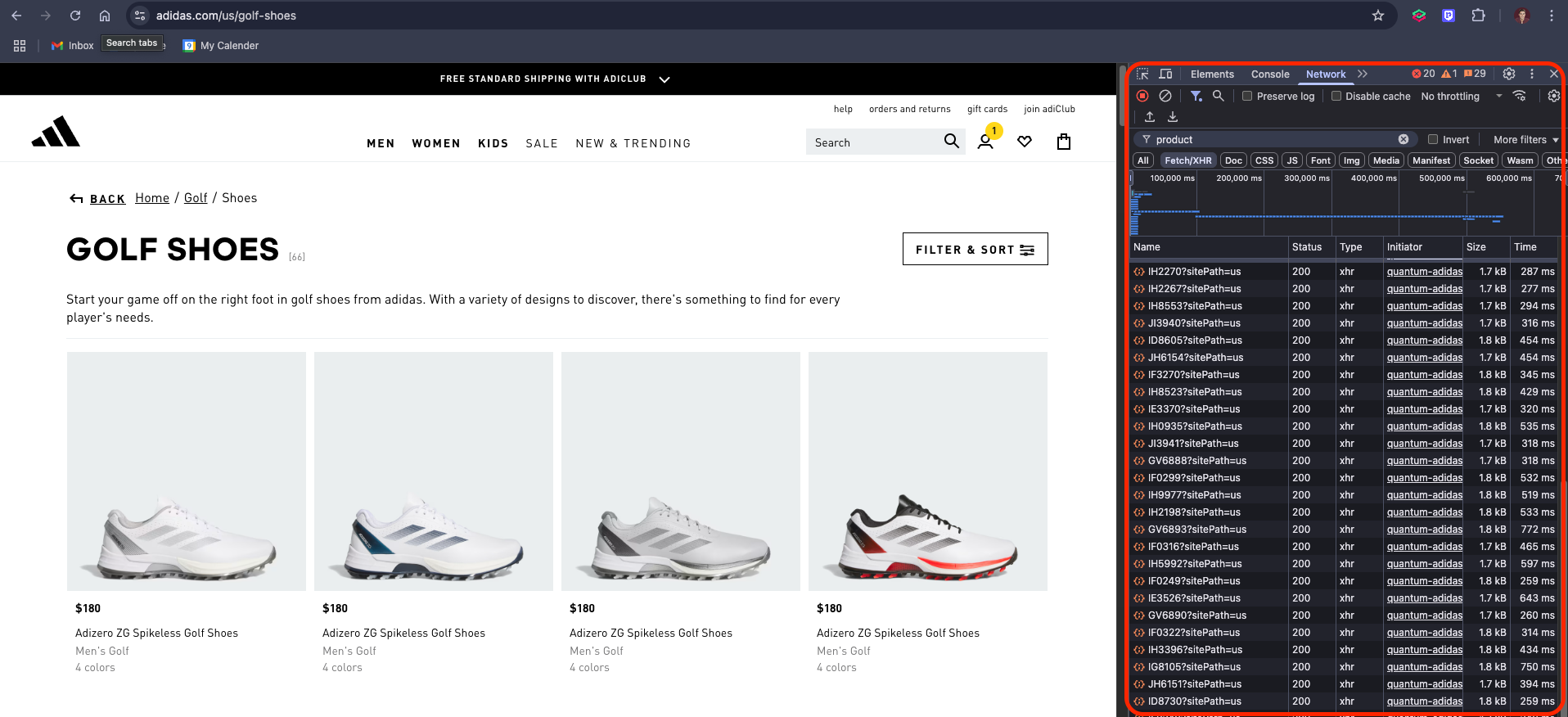
# Appendices

###### **Figure 3-1: Wappalyzer Security Analysis of** [**REI.com**](http://rei.com)

Shows the security technologies detected on REI.com, including reCAPTCHA, HSTS, Very Good Security, Akamai Bot Manager, and DigiCert for SSL/TLS.

###### **Figure 3-2: Wappalyzer Security Analysis of** [**Adidas.com**](http://adidas.com)

Displays the more extensive security stack on Adidas.com, including HSTS, Akamai Bot Manager, reCAPTCHA, Forter, Akamai Web Application Protector, and DigiCert for SSL/TLS.



###### **Figure 3-3: Browser Developer Tools Network Panel During API Reconnaissance**

This figure displays the Chrome Developer Tools Network panel while browsing the Adidas golf shoes category page. The right panel shows multiple API requests with product code patterns (e.g., ID8730?sitePath=us), demonstrating how API endpoints can be discovered by monitoring network traffic. Each request shows a successful 200 status code and consistent response size, indicating structured data retrieval.

###### 

###### **Figure 3-4: Detailed Headers for Adidas Product API Request**

This figure shows a detailed view of the network request headers for the Adidas product API endpoint. The highlighted Request URL reveals the consistent API pattern used for product data retrieval. The headers section displays important metadata including the successful status code, content type (application/json), and various server timing information used for performance tracking. URL: (<https://www.adidas.com/plp-app/api/product/ID8730?sitePath=us>)

###### 

###### 

###### **Figure 3-**5**: JSON Response from Adidas Product API**

This figure shows the structured JSON response from the Adidas API endpoint for product ID8730 (Codechaos 25 Spikeless Golf Shoes). The API returns comprehensive product data including title, URLs, pricing information, and availability status in a clean, structured format that's ideal for programmatic processing.

###### Figure 4-1: Average Request Time Comparison Across Crawler Implementations

###### Figure 4-2 Memory Usage Comparison Across Crawler Implementations

###### 

###### Figure 4-3: Data Efficiency Ratio Comparison Across Crawler Implementations

###### Figure 4-4: Evasion Success and Data Completeness Rates Across Crawler Implementations

# References

Anthropic. (2025). Claude 3.7 Sonnet [AI assistant]. Retrieved from https://claude.ai

Azad, B. A., Khormali, A., Karimzadeh, B., & Vigna, G. (2020). *Web Runner 2049: Evaluating third-party anti-bot services*. Proceedings of the ACM Internet Measurement Conference.

Bahrami, M., Singhal, M., & Zhuang, Z. (2015). *A cloud-based web crawler architecture*. 2015 18th International Conference on Intelligence in Next Generation Networks. IEEE.

Bhale, P. B., & Sonar, R. M. (2024). Performance analysis for web scraping tools: Case studies on BeautifulSoup, Scrapy, HtmlUnit and Jsoup. In M. K. Jha, D. K. Vishwakarma, & A. K. Sahu (Eds.), *Proceedings of International Conference on Recent Trends in Machine Learning, IoT, Smart Cities and Applications* (pp. 471–484). Springer.

Chiapponi, G., Balduzzi, M., Balzarotti, D., & Calzavara, S. (2021). *Scraping airline bots: Insights obtained studying honeypot data*. IEEE European Symposium on Security and Privacy.

Chiapponi, G., Balduzzi, M., Balzarotti, D., & Calzavara, S. (2023). *Inside residential IP proxies: Lessons learned from large measurement campaigns*. 32nd USENIX Security Symposium.

Demchenko, Y., Cuadrado-Gallego, J. J., Chertov, O., & Aleksandrova, M. (2024). Finding data on the web, data sets, web scraping, web API. In *Big data infrastructure technologies for data analytics: Scaling data science applications for continuous growth* (pp. 417–446). Springer Nature Switzerland.

Fan, Y. (2018). *Design and implementation of distributed crawler system based on Scrapy*. IOP Conference Series: Earth and Environmental Science, 108(4).

Glez-Peña, D., Lourenço, A., López-Fernández, H., Reboiro-Jato, M., & Fdez-Riverola, F. (2013). Web scraping technologies in an API world. *Briefings in Bioinformatics, 15*(5), 788–797.

Jacob, D., Kumar, S., & Laxmi, V. (2012). *PUBCRAWL: Protecting users and businesses from crawlers*. International Conference on Security and Privacy in Communication Systems. Springer.

Jiang, J., Liu, Z., Wang, Z., & Xu, J. (2012). *Offensive and defensive strategy of web crawler*. 2012 International Conference on Computer Science and Electronics Engineering. IEEE.

Jiang, J., Liu, Z., Wang, Z., & Xu, J. (2016). *Proxisch: An optimization approach of large-scale unstable proxy servers scheduling*. 18th IEEE International Conference on High Performance Computing and Communications.

Kausar, M. A., Dhaka, V. S., & Singh, S. K. (2013). *Web crawler: A review*. International Journal of Computer Applications, 63(2).

Kumar, A., & Singhal, S. (2014). *Web crawling using dynamic IP address using single server*. International Journal of Computer Science and Mobile Computing, 3(6), 1072–1077.

Najork, M. (2009). *Web crawler architecture*. In L. Liu & M. T. Özsu (Eds.), Encyclopedia of Database Systems (pp. 3462–3465). Springer.

Pant, S., Yadav, E. N., Milan, Sharma, M., Bedi, Y., & Raturi, A. (2024). Web scraping using Beautiful Soup. In *Proceedings of the 2024 International Conference on Knowledge Engineering and Communication Systems (ICKECS)*(pp. 1–6). IEEE.

Sharma, S., & Gupta, P. (2015). *The anatomy of web crawlers*. Proceedings of the International Conference on Computing, Communication & Automation. IEEE.

Sun, Y., Councill, I. G., & Giles, C. L. (2010). *The ethicality of web crawlers*. 2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, Vol. 1. IEEE.

Tsingenopoulos, I., Tefas, A., & Pitas, I. (2022). *Captcha me if you can: Imitation games with reinforcement learning*. 2022 IEEE Conference on Games (CoG). IEEE.

Wahed, M. A., Alzboon, M. S., Alqaraleh, M., Ayman, J., Al-Batah, M., & Bader, A. F. (2024). *Automating web data collection: Challenges, solutions, and Python-based strategies for effective web scraping*. In 2024 7th International Conference on Internet Applications, Protocols, and Services (NETAPPS) (pp. 1–6). IEEE.

Wang, J., & Guo, Y. (2012). *Scrapy-based crawling and user-behavior characteristics analysis on Taobao*. 2012 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery. IEEE