



School of Science and Engineering

MyScraper

**A Smart and Ethical Browser Extension
for Automated Web Data Extraction**

SPRING 2025

Student Name

Zakaria ELharmouzi

Supervised by:

Dr. Kevin Smith / Dr. Naeem Nisar Sheikh

Student conduct and copyright

The supervisor and the student (the Capstoner) agree that:

1. Permission has been obtained for any third party content (eg Data, illustrations, photographs, charts or maps).
2. The results described in this report have not previously been published

Student's name and signature:
Zakaria Elharmouzi

Supervisor's name and signature:
Dr. Kevin Smith / Dr. Naeem Nisar Sheikh

Title: MyScraper - A Smart and Ethical Browser Extension for Automated Web Data Extraction

Capstone Report

Student Statement:

I, Zakaria ELharmouzi, affirm that ethical considerations have guided my work throughout the development of this project. I have approached the design and implementation process with integrity, responsibility, and respect for the potential impact of this work. All decisions were made with careful attention to quality, reliability, and the broader well-being of individuals and communities who may interact with or be affected by the system.



Zakaria ELharmouzi

Approved by the Supervisor



Dr. Kevin Smith / Dr. Naeem Nisar Sheikh

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my family and friends for their unwavering encouragement, patience, and belief in me throughout this journey. Their support has been my greatest source of strength, and I am profoundly thankful for all the sacrifices they have made. I am also especially grateful to my supervisors, **Dr. Kevin Smith** and **Dr. Naeem Nisar Sheikh**, whose guidance, insightful feedback, and mentorship have been instrumental in shaping both my work and my personal growth. Finally, I extend my sincere thanks to everyone who contributed whether through advice, collaboration, or support to bringing this project to life. Your contributions have been invaluable.

TABLE OF CONTENTS

1. Introduction:	8
2. Problem Statement:.....	8
3. Project Specifications:.....	8
3.1 Project Tech Stack, Tools, and Output Formats	8
3.2 Functional Requirements.....	9
3.3 Non-Functional Requirements	10
4. Steeple Analysis:	11
5. Engineering Standards and Practices:	12
5.1 Formal Standards.....	12
5.2 Secure and Maintainable Coding Guidelines.....	13
6. Logic Model Framework:	15
7. Literature Review	16
8. Methodology & Capstone Design.....	18
8.1 Overview of the System Architecture	18
8.2 Early Web Scraping Approaches.....	18
8.2.1 Static HTML parsing with BeautifulSoup	18
8.2.2 Dynamic Browser Automation with Selenium.....	21
8.3 Anti-Detection & Stealth Strategies.....	23
8.3.1 How Websites Detect Scrapers.....	24
8.3.2 Our Evasion Techniques	25
8.3.3 Monitoring & Adaptive Retries.....	26
8.4 LLM-Driven Extraction Strategy	27
8.4.1 LLM-Driven Scraping Workflow with Crawl4AI.....	27
8.4.2 LLM-Driven Extraction Strategy	29
8.4.3 Integration with Crawl4AI & FastAPI.....	30
8.4.4 Adaptivity & Self-Healing	30
8.5 Application Development and System Implementation.....	31
8.5.1 Extension Architecture & Tech Stack	31
8.5.2 Front-end Workflows (UI, Element Selection & Export)	32
8.5.3 Backend Integration (FastAPI, Task Queue, Data Persistence)	33
8.6 Testing, CI/CD & Deployment	37
9. DATA PRESENTATION, RESULTS, INTERPRETATION.....	39
9.1 Overview of the Data Collected.....	39
9.2 Data Presentation	40
9.2.1 Static Scraping (BeautifulSoup).....	40
9.2.2 Dynamic Scraping (Selenium plus BeautifulSoup).....	42

9.2.3 AI-Driven Schema Extraction (Crawl4AI plus Groq LLaMA 3)	46
9.3 Comparative Analysis of Scraping Techniques	48
9.3.1 Best Method for Different Requirements	49
9.4 Simulations, Results, Interpretation.	50
9.4.1 Testing Scenarios	50
9.4.2 Observed Metrics and Key Observations.....	53
9.4.3 Comparative Simulation Results Table.....	54
9.4.4 Interpretation and Implications	54
9.4 AI Model Comparison for Scraping	55
9.4.1 Overview and Rationale.....	55
9.4.2 Model Comparison Criteria	56
9.4.3 Comparative Performance Summary.....	56
9.4.4 Key Findings and Interpretations	59
10. Motivation, Personal Benefits & Skill Growth	60
10.1 Problem-Driven Inspiration	60
10.2 How This Capstone Advances My Technical Expertise.....	60
10.3 Soft-Skills & Professional Development Gains	60
11. Project Significance & Value Proposition	61
11.1 Benefits to End-Users	61
11.2 Academic and Industry Impact.....	61
11.3 Ethical & Legal Contributions	61
12. Limitations, Challenges and Lessons Learned	62
12.1 Technical Constraints and Workarounds	62
12.2 Scope Boundaries & Time Management	62
12.3 Key Takeaways for Future Projects	62
13. Conclusion.....	63
14. APPENDIX:	64

LIST OF FIGURES

Figure 1- Overview of the Entire System Architecture.....	18
Figure 2- Target Website for static scrapping.....	40
Figure 3- Json Request for the backend for Static Website.....	41
Figure 4- Token Validation and Initial Response from the backend	41
Figure 5- Backend Logs Indicating start of our Scrapper and finish.....	41
Figure 6- Filtered and structured results of our static scrapper	41
Figure 7- Element Selection using Our Browser Extension	42
Figure 8- Element labeling and automatic JSON request building using our Browser Extension	43
Figure 9- Full instructions After selecting targeted scrapping elements.....	44
Figure 10 Backend Logs Indicating start of our Dynamic Scrapper and its Actions.....	44
Figure 11- Filtered and structured results of our Dynamic scrapper.....	45
Figure 12- Total scrapped Quotes from our Dynamic Scrapper.....	45
Figure 13- Filtered and structured results of our Dynamic Scrapper for Amazon.com.....	46
Figure 14- JSON request for our AI-Driven Scrapper	47
Figure 15- Backend Logs Indicating start of our AI-Driven Scrapper and finish	47
Figure 16- Figure 6- Filtered and structured results of our AI-Driven scrapper	47
Figure 17- JSON request for our AI-Driven Scrapper that target Amazon.com	48
Figure 18- Backend Logs Indicating start of our AI-Driven Scrapper and finish for Amazon.com ...	48
Figure 19- quotes.toscrape.com: Static HTML parsing test.....	51
Figure 20- Test LLM semantic extraction and error recovery.....	51
Figure 21- Indeed: Test LLM semantic extraction and error recovery.....	52
Figure 22- Jumia: Test Selenium's ability to scroll/load/collect.....	52
Figure 23- Figure 22-Amazon: Test Selenium's ability to scroll/load/collect	53
Figure 24- Extraction Accuracy (%)	57
Figure 25- Schema Compliance (%).....	58
Figure 26- Processing speed in tokens per second.....	58
Figure 27- Operational cost in USD per 1 000 tokens.....	59

LIST OF TABLES

Table 1- Comparative Analysis of Scraping Techniques.....	49
Table 2- Best Method for Different Requirements.....	49
Table 3- Testing Scenarios	50
Table 4- Comparative Simulation Results Table	54
Table 5- Model Comparison Criteria	56
Table 6- Comparative Performance Summary.....	56

ABSTRACT (ENGLISH)

MyScraper is a universal browser extension and accompanying backend designed to democratize the extraction of structured web data for non-technical users. Leveraging a triad of scraping strategies static HTML parsing via BeautifulSoup, dynamic browser automation with Selenium (enhanced by human-like interaction emulation), and AI-powered schema-driven extraction using Groq LLaMA within the Crawl4AI framework MyScraper adapts to a wide variety of site architectures while respecting legal and ethical constraints (robots.txt, GDPR, CCPA). A Parcel-bundled front end enables intuitive element selection; a FastAPI backend orchestrates asynchronous tasks, validates results against JSON schemas, and persists outputs to PostgreSQL or Excel. Comparative evaluations demonstrate that static scraping offers unmatched speed on simple pages, Selenium delivers robust interactivity support, and LLM extraction excels in volatile or poorly documented layouts. By combining these methods under a unified workflow, MyScraper empowers researchers, students, and analysts to rapidly gather, clean, and export web data with minimal maintenance overhead.

Keywords: MyScraper, web scraping, browser automation, AI extraction, schema-driven

ABSTRACT (FRENCH)

MyScraper est une extension de navigateur universelle accompagnée d'un service backend conçue pour rendre l'extraction de données web structurées accessible aux utilisateurs non-techniques. En combinant trois approches de scraping analyse statique du HTML avec BeautifulSoup, automatisation de navigateur dynamique via Selenium (avec simulation de comportement humain), et extraction pilotée par schéma grâce à Groq LLaMA dans le cadre Crawl4AI MyScraper s'adapte à diverses architectures de sites tout en respectant les contraintes légales et éthiques (robots.txt, RGPD, CCPA). Une interface front-end empaquetée avec Parcel permet la sélection intuitive d'éléments ; un backend FastAPI orchestre les tâches asynchrones, valide les résultats selon des schémas JSON et stocke les données dans PostgreSQL ou au format Excel. Les évaluations comparatives montrent que le scraping statique est ultra-rapide sur les pages simples, que Selenium offre une prise en charge robuste des sites interactifs, et que l'extraction par LLM excelle sur les structures instables ou mal documentées. En combinant ces méthodes au sein d'un même flux de travail, MyScraper permet aux chercheurs, étudiants et analystes de collecter, nettoyer et exporter rapidement des données web avec un minimum de maintenance.

Mots clés: MyScraper, extraction web, automatisation de navigateur, IA, schémas JSON

1. Introduction:

This capstone project aims to develop **MyScraper**, a universal browser extension that enables users (students, professors, researchers and more) to be able to **extract structured data from websites** without requiring advanced programming skills. The project focuses on ensuring compliance with **legal and ethical web scraping guidelines** While providing users with an intuitive and automated solution. It integrates various languages and frameworks merging them together to facilitate the most efficient data extraction while also researching ways for **AI-driven Scraping and automation** to scrap and then refine extracted content and improve stability, and to adhere to **Users** needs.

2. Problem Statement:

Extracting structured data from websites can be quite a challenge for **researchers, students, professors and analysts**, as most websites today try as much as possible to display public data that is under fair use but choose to block downloads for it, also as existing tools often require more advanced programming knowledge or violate ethical standards. Many web scrapers outright ignore policies and most often than not they lack intuitive and simple interfaces for non-technical users. **MyScraper** addresses these issues by providing a **user-friendly, legal, and efficient** browser extension that enables structured data extraction with minimal effort.

3. Project Specifications:

3.1 Project Tech Stack, Tools, and Output Formats

BackEnd: FastAPI-based processing system and REST API to manage scraping requests, job orchestration, and result retrieval.

FrontEnd:

- Built with vanilla **JavaScript, HTML, and CSS**
- Bundled using **Parcel** for asset compilation and hot reloading during development
- Content scripts handle element-selection overlays; background scripts communicate with the **FastAPI server**

Database: PostgreSQL for persisting user accounts, scrape tasks metadata, and JSON results

Scraping Tools:

- **Static scraping:** **BeautifulSoup** for server-rendered pages
- **Dynamic scraping:** **Selenium** for **JavaScript-heavy sites**, with **human-like** interaction helpers

AI-Integration:

- **NLP-based** post-filtering and deduplication of scraped data
- **Schema-driven**, automated AI extraction (under research) powered via **Crawl4AI** and **Groq LLMs**

Output Formats: CSV, JSON, and Excel exports for users' downstream analysis

Security & Compliance: Adherence to robots.txt, GDPR, and CCPA policies; built-in rate-limiting and respect for site terms

3.2 Functional Requirements

1. User Authentication & Authorization

- **Users** must be able to register, log in, and log out securely.
- **Role-based access** control ensures only authenticated users can create or view scraping tasks.

2. Task Creation & Configuration

- From the browser extension, users can select page **elements (via CSS or XPath)** and define scraping instructions.
- Users submit tasks through the **extension UI**, supplying a **URL**, **optional schema**, and **scheduling parameters**.

3. Job Orchestration & Monitoring

- The **FastAPI** back end accepts scrape requests, **enqueues them**, and **returns a unique task ID**.
- **Users** can poll a **“get task”** endpoint to retrieve status (**pending / running / completed / failed**) and any error messages.

4. Data Extraction & Post-Processing

- Static pages are scraped with **BeautifulSoup**; dynamic pages use our Selenium-based engine with human-like helpers.
- Extracted data flows through **an NLP filter** for **deduplication** and relevance ranking.

5. Result Retrieval & Export

- **Once complete**, users can fetch raw **JSON via API** or **download as CSV, Excel, or JSON** directly from the Extension.
- Exports respect the user's locale.

6. Administration & Audit

- An admin interface (**or read-only API**) lists **all tasks, their parameters, and metadata** for auditing.
- **Administrators** can cancel **stuck jobs** or **re-enqueue failed ones**.

3.3 Non-Functional Requirements

1. Performance & Scalability

- **API** endpoints must respond within **300 ms** under normal load.
- The system should scale to handle at least **200 concurrent scrape jobs** via **horizontal scaling of worker nodes**.

2. Reliability & Availability

- **Target system uptime: 95%.**
- Automatic retries/back-off for transient failures (**network hiccups, timeouts**).

3. Security & Compliance

- All traffic secured via **TLS**; sensitive data (**API tokens, user passwords**) **encrypted** at rest.
- Strict adherence to **robots.txt**, **GDPR**, and **CCPA**; **rate limits enforced to prevent accidental DOS**.

4. Maintainability & Testability

- Code organized into clear layers (**extension UI, scraper engine, API, data layer**).
- $\geq 90\%$ unit- and integration-test coverage; **CI pipeline runs linting, tests, and builds on every PR**.

5. Usability & Accessibility

- **Extension UI follows WCAG 2.1 AA guidelines** for keyboard navigation and screen-reader support.
- **Clear error messages** and **guided tooltips help non-technical users** configure tasks.

6. Portability & Extensibility

- **Front-end assets bundled with Parcel**; **back-end containerized via Docker**.
- Modular plugin system allows adding new extraction strategies (**e.g. Playwright**) without core rewrites.

4. Steeple Analysis:

- **Social:** This project aims to address the growing need for accessible data collection tools, empowering researchers, students, and businesses to make data driven decisions after they analyze the data gathered from public sources. However, some people may be concerned about the legality of this data extraction.
- **Technological:** The project will leverage modern web technologies and modern frameworks to create an efficient and user-friendly web scraping tool. As web structures and security measures evolve, MyScraper must adapt to changes such as bypassing anti-scraping tools and dynamic content loading.
- **Economic:** MyScraper provides a cost-effective solution for all people to gather publicly available data, without the need for additional workers to manually gather and analyse the data. It would allow companies to track market trends, customer insights, competitor strategies and way more, while also allowing professors and students to use that data for their research or other tasks, the project will be offered as freemium at first with ads as a source of income and then there will be some monetization features and subscriptions later.
- **Environmental:** By replacing traditional manual data collection methods, This project helps reduce the impact on the environment by reducing the reliance on paper-based reports and surveys, minimizing waste. However continuous scraping and data processing consume energy, contributing to increased server loads and potential carbon footprints. Implementing optimized algorithms and reducing redundant data requests can help mitigate these environmental concerns.
- **Political Factors:** Governments worldwide are increasingly supporting open data initiatives, which align with the ethical use of web scraping for research and public benefit. However, regulatory policies regarding internet access, digital rights, and data collection vary between countries. Some governments impose restrictions on data scraping, making it essential to align MyScraper with legal and political frameworks to avoid conflicts.
- **Legal Factors:** The tool must ensure compliance with data protection laws, such as Scraping data from websites without permission could lead to legal disputes if terms of service are violated.
- **Ethical:** Ethical concerns arise when web scraping is used unethically or without

transparency. MyScraper should emphasize responsible data collection, ensuring that extracted data is used ethically, legally, and for constructive purposes. Preventing misuse, such as spreading misinformation or violating privacy, is crucial to maintaining the credibility and positive impact of the tool.

5. Engineering Standards and Practices:

To deliver a robust, maintainable, and secure system, MyScraper adheres not only to formal engineering standards but also to industry-proven coding conventions, folder layouts, and operational practices.

5.1 Formal Standards

ISO/IEC 27001 - Information Security Management

We follow the ISO/IEC 27001 framework to identify and mitigate security risks across the entire data lifecycle. Key controls include:

- **Access Management:** Role-based access controls in the database and API layer, ensuring only authenticated users can submit or view scraping tasks.
- **Encryption:** TLS for all in-transit data; AES-256 encryption for sensitive fields at rest (e.g. user credentials, API tokens).
- **Incident Response:** Structured logging of all errors and unauthorized access attempts, with automated alerts to the project team for triage.
- **Periodic Audits:** Regular pen-testing of the FastAPI endpoints and CI-driven dependency vulnerability scans (e.g. with Dependabot).

W3C Web Standards & Accessibility

Our browser extension and any HTML/CSS assets strictly follow:

- **HTML5 & CSS3 Best Practices:** Semantic elements (<button>, <nav>, <section>) to aid readability and assistive technologies.
- **ARIA Attributes:** Proper roles and labels on interactive overlays to ensure screen-reader compatibility.
- **Cross-Browser Compliance:** Testing on Chromium-based and Gecko-based engines, with polyfills (via Parcel) for any missing API.
- **Performance Budgets:** Bundle size limits (< 200 KB minified) and Lighthouse audits to maintain > 90 performance/accessibility scores.

5.2 Secure and Maintainable Coding Guidelines

Python (FastAPI) Conventions

- **PEP 8 Compliance:** Automated enforcement via flake8 and black for formatting.
- **Type Safety:** Full use of type annotations and Pydantic models for request/response validation.
- **Modular Design:** Extraction strategies, API routers, and database models each live in separate modules.
- **Dependency Management:** poetry lockfiles ensure reproducible environments.

JavaScript (Extension) Conventions

- **ESLint + Prettier:** Consistent linting and code style across all content and background scripts.
- **Manifest V3 Architecture:** Strict separation between background.js, contentScript.js, and UI “popup” code.
- **Secure Message Passing:** Use of chrome.runtime.sendMessage with schema-validated payloads to prevent injection attacks.

5.3 Recommended Folder Structure

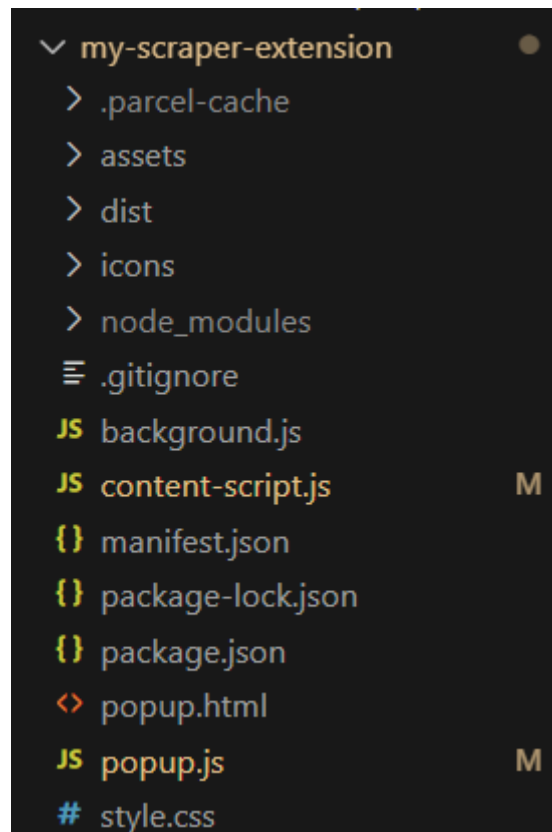


Figure 1- Front-end folder structure

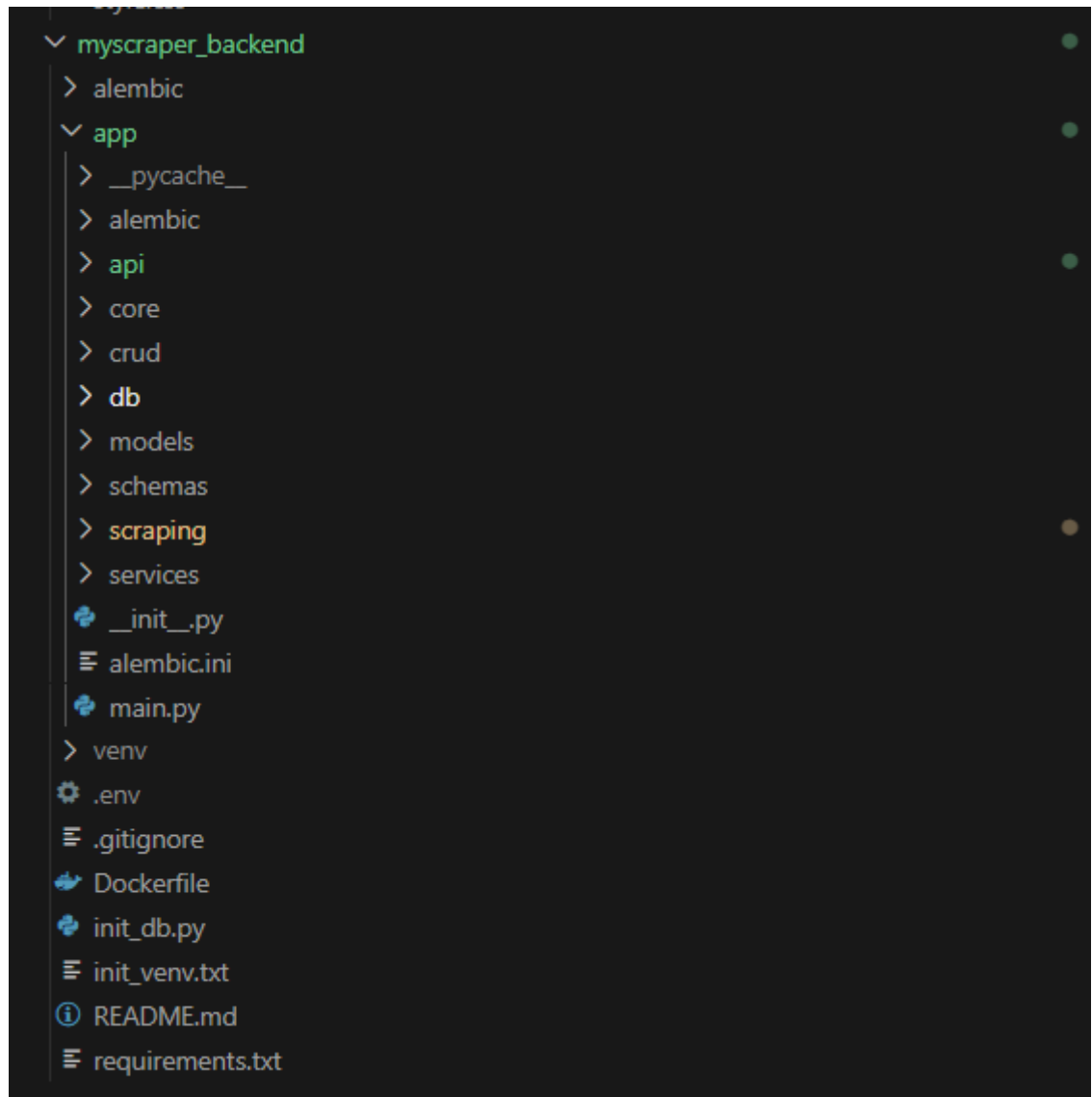


Figure 2- Back-end folder structure

6. Logic Model Framework:

5.1 Target Population: Researchers, students, and analysts who need structured web data.

5.3 Underlying Assumptions: Publicly available data can be legally and ethically scraped for research and analysis.

5.4 Resources/Challenges:

- **Resources:** Open-source tools, cloud storage, and scraping libraries (Crawl4AI + Groq LLMs).
- **Challenges:** Legal restrictions, website anti-scraping mechanisms, and performance issues.

5.4 Current Status:

As of March 2025, the FastAPI backend successfully returns JSON-structured quotes from both static and dynamic scrapers, and the Parcel-built extension prototype enables users to define selectors on live pages.

5.5 Outputs of the project:

- **Universal WebExtension** supporting Chrome/Firefox
- **FastAPI** service with Pydantic-validated scrape endpoints
- Three scrape pipelines: **BeautifulSoup**, **Selenium**, **LLM-based**
- Export formats: **CSV**, **JSON**, **Excel**
- User and developer documentation

5.6 Outcomes:

- **Accessibility:** Non-technical users can fetch data without coding.
- **Efficiency:** Reduces manual collection time by $\geq 80\%$.
- **Compliance:** Honors **robots.txt**, terms, and privacy policies.
- **Adoption:** Empowers researchers to focus on analysis rather than collection.

7. Literature Review

Web scraping has undergone a remarkable transformation over the past decade, driven by the explosive growth of online data and the corresponding need for automated extraction tools. Early “hacks” relied on bespoke regular-expression scripts to pull out snippets from raw **HTML**; while these ad-hoc solutions could work for trivial, well-structured pages, they quickly broke down as soon as the slightest change was made to the site’s markup.

The first major shift came with the advent of dedicated parsing libraries such as **BeautifulSoup** (Richardson, 2004), which wrap an **HTML** document into a navigable parse tree and expose a simple **API** for querying by tag **name**, **attributes**, **CSS selectors** or **XPath**. **BeautifulSoup** ushered in an era of static **HTML** scraping where developers could write just a few lines of code to locate and extract data from **server-rendered** pages an approach that remains **lightning-fast** and easy to maintain so long as the target pages do not rely on client-side rendering.

As websites increasingly adopted **JavaScript-driven, single-page-app architectures**, static parsers alone proved insufficient. Tools like **Selenium and Puppeteer** filled this gap by embedding a **real (or headless) browser** under programmatic control. These frameworks load the page, execute all scripts, fire **AJAX** calls and then expose the fully rendered **DOM** for inspection. This dynamic browser automation model allows scrapers to interact with cookie-banners, click “**load more**” buttons, scroll to **lazy-loaded content**, or **navigate** complex user flows at the cost of higher resource usage and more elaborate setup.

Around the same time, specialized scraping frameworks such as **Scrapy** (López, 2008) introduced more industrialized workflows for large-scale crawls: built-in request scheduling, configurable pipelines for data cleaning, retry and **back-off policies**, and tight integration with storage backends. **Scrapy** remains a go-to choice for projects that must handle thousands of pages per hour under a unified, declarative configuration.

More recently, a third wave has emerged: **AI-driven scraping**. By leveraging natural-language models (e.g. **OpenAI’s GPT family, LLaMA derivatives, Claude**) or computer-vision techniques, these systems can “read” a page the way a human does—identifying headings, data tables, or list items via semantic cues rather than brittle selectors. For example, a model can be prompted:

“Find the product name, price, and availability status on this page and return them as JSON.”

This schema-based extraction paradigm dramatically reduces maintenance: if a site changes its **CSS classes**, the model still reasons over the page content and can adapt to minor layout shifts. However, it introduces new trade-offs around latency, cost (**tokens consumed**), and the occasional hallucination or mis-parsing on entirely novel layouts.

Our Capstone project combines the strengths of all three generations of scraping:

1. **Static Pipelines (BeautifulSoup + Requests)** for the simplest, fastest extractions on non-dynamic sites.
2. **Dynamic Automation (Selenium)** for heavy-JavaScript pages or when user interactions are required.
3. **AI-Powered Schema Extraction** (via **Crawl4AI + LLMs**) for highly heterogeneous, frequently changing pages where long-term maintainability is paramount.

In addition, we extend these established practices with:

- **User-Defined Selectors** in our browser extension, enabling non-technical users to click on any page element and have its CSS or XPath recorded automatically.
- **Ethical & Legal Guardrails**, enforcing robots.txt compliance and respecting GDPR/CCPA boundaries at every step.
- **Stealth Strategies**, such as user-agent rotation, proxy pools, and human-like delays, to blend into legitimate traffic and minimize blocking.

By uniting decades of scraping **research-static parsers**, **headless browsers**, **pipeline frameworks**, and **cutting-edge LLMs**-our solution delivers a flexible, resilient, and user-friendly data extraction toolkit capable of handling the full spectrum of modern web architectures.

8. Methodology & Capstone Design

8.1 Overview of the System Architecture

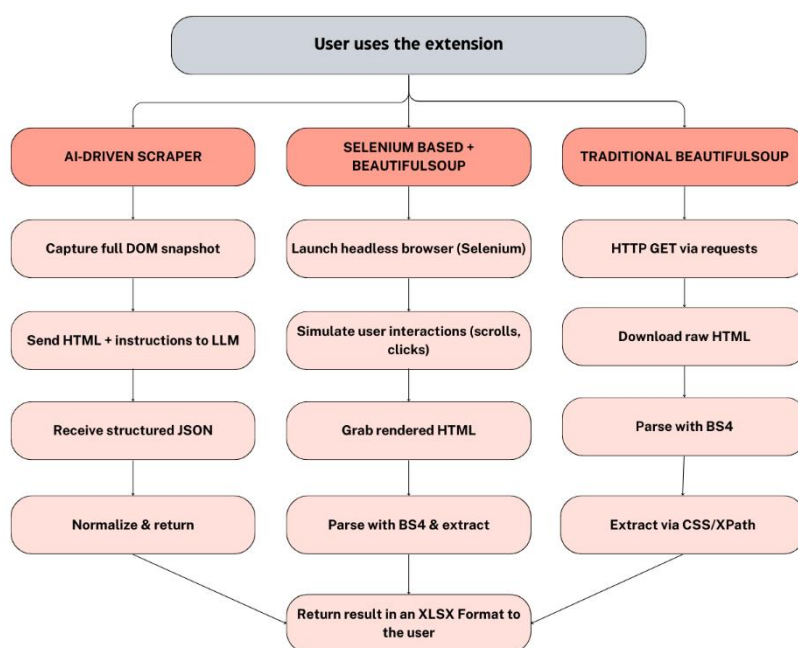


Figure 3- Overview of the Entire System Architecture

8.2 Early Web Scraping Approaches

8.2.1 Static HTML parsing with BeautifulSoup

Overview:

traditional webs scraping relied on simple techniques such as issuing HTTP requests to retrieve raw HTML, then using that fetched HTML and running it through an HTTP-parsing library, most commonly BeautifulSoup (BS4) to navigate the document tree and extract the data present in it.

Workflow and Data Extraction Pipeline:

In our static HTML scraping approach, we decompose the end-to-end process into six distinct stages. Each stage is governed by robust error handling and logging, designed for clarity and maintainability. Below, we enumerate these stages in sequence, using numbered subtitles to guide the reader through the journey from issuing the initial request to producing the final serialized output.

1. HTTP Request:

The first step in our pipeline is to establish communication with the target server via a standard **HTTP GET** request. We leverage a persistent session object either from the **requests** library or **httpx** to reuse **TCP** connections and manage cookies automatically. Before dispatching the request, we assemble realistic headers: a modern User-Agent string matching current browser version, Accept-Language settings reflecting genuine human preferences, and a valid referrer header. Timeouts are configured to avoid hanging indefinitely on slow or unresponsive servers, and an exponential-backoff retry mechanism ensures that transient network hiccups or brief server overloads do not derail the scraping task. For large-scale deployments, this stage can optionally integrate proxy rotation and jittered delays between requests, reducing per-IP request rates and blending into legitimate traffic patterns.

2. HTTP Parsing:

Once the raw HTML payload arrives, we transition to parsing it into a structured document object. We feed the response's text into BeautifulSoup with the high-performance "**lxml**" parser, ([Appendix A](#)) which gracefully handles imperfect or malformed HTML by constructing a resilient parse tree. This tree exposes every tag, attribute, and nested structure, enabling precise downstream querying. Any parse warnings including unclosed tags or unexpected nesting are recorded in our log files, allowing developers to detect subtle site changes that might otherwise cause silent data omission.

3. Target Element Selection:

With a navigable parse tree in hand, we zero in on the specific elements that contain our data of interest. In the case of `quotes.toscrape.com` a testing website for scrappers, we select `<span>` nodes with the **classes** `.quote .text` for quote bodies **and** `.quote .author` for authorship. ([Appendix B](#)) These CSS selectors are crafted to match the site's semantic markup; should the site's developers change class names or restructure the DOM, a broader fallback query using XPath or ancestor-descendant relationships is attempted, and any deviations from expected counts are logged as extraction warnings. This two-tiered selector strategy balances precision with resilience, ensuring we capture as many valid records as possible without silent failures.

4. Text Extraction & Cleaning:

After locating the target elements, we extract their inner text using each element's `get_text(strip=True)` method. This operation removes leading and trailing whitespace and collapses repeated internal spaces, producing a clean string. We then apply custom post-processing: a small regular expression strips surrounding quotation marks, em-dashes, or stray punctuation; HTML entities are automatically decoded by BeautifulSoup; and Unicode is normalized (using NFC) to ensure consistent handling of accented characters or ligatures. All transformations are deterministic and reversible, guaranteeing that the extracted text faithfully represents the source content.

5. Text Extraction & Cleaning:

With cleaned text and author strings prepared, we pair them in lockstep. If the number of quote elements does not match the number of author elements a sign of unexpected page structure we raise a warning and omit incomplete pairs to maintain data integrity. Each valid pair is assembled into a simple Python dictionary of the form `{"text": quote_text, "author": author_name}`. Before proceeding, we validate each record against a lightweight JSON Schema, ensuring nonempty values and correct data types. This schema-based validation acts as a final gate, preventing malformed entries from entering downstream systems.

6. Output Serialization:

In the final stage, the list of validated records is ready for consumption. We support flexible serialization pathways: directly returning JSON in a FastAPI response for real-time API calls; streaming into newline-delimited JSON for log-based ingestion; writing to Excel via pandas for batch analytics; or loading into an in-memory DataFrame for further transformation. All output

routines respect the same deterministic ordering and formatting rules, guaranteeing that repeated runs against an unchanged page will yield byte-for-byte identical outputs. This consistency underpins downstream reproducibility and auditability.

8.2.2 Dynamic Browser Automation with Selenium

1. Overview:

To reliably scrape modern, JavaScript-driven websites and to stay one step ahead of anti-bot systems we drive a genuine Chrome instance via Selenium, wrapped by our **user_defined_scraper** framework. Beyond simple page loads, we layer in realistic user behaviors, intelligent retry logic, and rotating identities so that each “**agent**” behaves and looks like a distinct human visitor, this is to avoid detection by websites that highly enforce Anti scrapping measures (detailed later).

2. Page Rendering, DOM Interactions & Event Handling:

When our scraper launches, it instantiates Chrome with hardened options: disabling the default “**AutomationControlled**” flags, excluding the **navigator.webdriver** property, and randomly selecting one of several up-to-date **User-Agent** strings. After navigating to the target URL, we pause a randomized 2–4 seconds to allow all network requests, fonts, and scripts to settle. If the page requires infinite scroll say, on social feeds or product galleries we invoke **perform_human_like_scrolling()**, which divides the total height into multiple, slightly jittered steps and even simulates occasional back-and-forth movements to mirror natural reading.

Once the DOM is stable, we execute any pre-defined JavaScript actions (for example, cookie-consent clicks or dropdown expansions) through **perform_js_actions()**. Each action is wrapped in a retry loop ([Appendix E](#)): we first wait for elements to be clickable, then attempt a **native .click()**, and fall back to a JavaScript-driven click if needed. By combining explicit waits, fallback mechanisms, and small random delays between each click or hover, we coax the page into the exact state a real user would experience ensuring that the desired data is present and accessible for extraction.

3. Human-like Behavior Simulation:

To blend seamlessly into normal traffic patterns, every interaction in our scraper is deliberately non-deterministic. Scrolls happen in 3–15 steps of variable length; after each scroll we pause by $0.7\times$ – $1.3\times$ of a base interval, and 30% of the time we inject a tiny upward nudge before continuing. Clicks and hovers are done via Selenium’s ActionChains, which moves a

virtual cursor over the element, pauses for a moment, then clicks just as a human would. Our generic **retry_logic()** wrapper catches intermittent timeouts or stale references and retries up to three times with increasing delays, mimicking a patient user who might wait or refresh if something doesn't load immediately. Taken together, these patterns defeat simple rate-limiters and behavioral fingerprints that look for perfect timing or zero-variance interactions.

4. Agent Rotation & Identity Cycling:

A single, long-lived browser profile is a red flag for many anti-scraping defenses. To avoid detection, we treat each scrape session as an “agent” with its own ephemeral profile and network identity:

- **User-Agent Shuffling:** Before each session, we randomly select from a curated list of real-world User-Agents (desktop and mobile) and inject it via Chrome options ([Appendix C](#)).
- **Proxy Cycling:** If a proxy pool is provided, we assign a fresh HTTP/SOCKS proxy to each new driver instance. Should a session fail repeatedly, we rotate to the next proxy in the list and restart the browser ([Appendix D](#)).
- **Profile Isolation:** Each agent uses a temporary Chrome user-data directory. Once the scrape completes, we delete this directory along with any cookies, local storage, or fingerprinting artifacts so that no two sessions share history or cache.
- **Throttle Resetting:** To further mask automated patterns, we randomize the delay before agent teardown and the interval between full browser restarts.

By cycling these identities, our system can sustain high-volume scraping without triggering IP bans or shared-cookie detectors, effectively masquerading as multiple, independent visitors.

5. Extraction Cleaning & Persistence:

Once our headless browser has completed the scrolling, clicks and field-level extractions, we aggregate the raw per-page dictionaries into a unified result set. First, we iterate over each page's **page_data** and merge matching keys, appending list-valued fields or overwriting scalar ones only if they're non-empty. This “merge loop” as I call it handles pagination, infinite scroll and any multi-step **DOM** interactions transparently. Next comes **data cleaning** for each field value we do the following operations:

- **Strip extraneous whitespace** and normalize internal spacing.
- **Remove HTML entities** (e.g. ` `, `&`) via BeautifulSoup's `BeautifulSoup(text,`

"lxml").text.

- **Apply lightweight parsing** for dates, numbers or currency when the schema indicates such types falling back to raw strings on parse failure.
- **Deduplicate** any list fields by preserving insertion order.
- Once we complete the data cleaning, we write into our PostgreSQL **scrape_results** table under the task's UUID, ensuring that **raw** json snapshot is always preserved if the user ever requests the data again, and for auditability.
- **Convert** the same list into a panda **DataFrame** (**pd.DataFrame(raw_rows)**), which we persist to an Excel file (**{task_id}.xlsx**) via **df.to_excel(...)** giving end users both structured programmatic access and a human-readable spreadsheet.

Finally, our **FastAPI-backed** endpoint updates the task's status to "completed", attaches the cleaned result payload, and returns a summary object to the caller. Errors during any of these steps (validation failures, I/O issues, parse errors) are caught, logged, and surfaced back as graceful "failed" statuses with explanatory messages so no silent data loss occurs.

6. When & Why to Choose Browser-based Scraping:

While our static HTML pipeline (Section 8.2.2) excels at speed and simplicity, many sites today render content only after complex JavaScript execution or hide data behind interactive widgets. Browser automation is indispensable when:

Critical content such as infinite-scroll feeds, map overlays, or lazy-loaded images only appears after viewport events.

Interactive flows login forms, "load more" buttons, or multi-step wizards must be traversed to reach the underlying data.

Anti-automation defenses employ canvas fingerprinting, dynamic class names, or challenge pages that require real browser capabilities.

Although Selenium incurs higher CPU, memory, and time costs per page, it guarantees **pixel-perfect fidelity**. We reserve it for targets where static parsers fail, falling back to **lightweight HTTP** parsing for simpler pages. Dynamic selection logic in our orchestration layer measures scrape success rates and performance, automatically adjusting between static and browser-driven approaches to optimize throughput while minimizing detection risk.

8.3 Anti-Detection & Stealth Strategies

Websites deploy a range of automated defenses to distinguish legitimate human visitors from automated scrapers. Broadly speaking, these methods fall into two categories: (1) network-level controls such as rate-limiting, IP bans and CAPTCHAs and (2) client-side fingerprinting and honeypot traps.

8.3.1 How Websites Detect Scrapers

1. Rate-Limiting, IP Bans & CAPTCHAs

At the network edge, web servers and CDNs monitor request patterns to identify abuse. If a single IP address issues too many requests in a short time frame, the server may throttle or block further requests from that address, these techniques are almost always present in large companies websites (LinkedIn, Indeed, Amazon and more). These rate-limits are often enforced via simple token-bucket algorithms or more sophisticated heuristics based on geolocation and historical traffic profiles. Once an IP is flagged, it can be added to an access control list (ACL) and black-holed entirely (technique used by Amazon and LinkedIn).

When simple IP blocking is insufficient, sites escalate by presenting CAPTCHAs interactive challenges designed to be easy for humans but costly for bots to solve. Modern CAPTCHAs vary from distorted-text puzzles to behavioral tests like “select all the images containing traffic lights.” Bots that fail these challenges are denied further access. In extreme cases, requests may be funneled through JavaScript-based CAPTCHA widgets that only render upon detecting suspicious traffic characteristics

2. Browser Fingerprinting & Honeypot Traps

On The client side, sites often collect a “fingerprint” of each visitor of the website, by combining dozens of signals HTTP header values (eg. User-Agent, Accept-language) , supported JavaScript APIs, installed fonts, screen resolution, canvas rendering quirks and more. Any unusual or overly uniform fingerprint patterns (for instance, dozens of requests sharing identical header orderings or missing standard properties like navigator.plugins) can trigger further scrutiny. Bots that do not perfectly emulate the entire breadth of a modern browser’s environment are flagged as non-human.

To catch more sophisticated scrapers, many sites embed honeypots hidden form fields or links that are invisible to real users but detectable by parsers. For example, an HTML `<a>` tag styled with `display:none` or positioned off-screen will never be clicked by a real user,

but a naive scraper might follow it. Once the bot interacts with that element, the server knows it's being scraped and can ban or throttle the session

By combining these network-level and client-side techniques, websites create layered defenses that make large-scale, undetected scraping increasingly difficult. Continuous innovation such as integrating behavioral analysis and machine-learning–based anomaly detection further raises the bar for automated crawlers.

8.3.2 Our Evasion Techniques

To operate reliably against anti-bot defenses while respecting legal and ethical boundaries, MyScraper employs a multi-layered stealth strategy. By combining header obfuscation, distributed networking, and human-pattern timing, the scraper mimics genuine user behavior and rotates its footprint to remain under the radar.

1. User-Agent Spoofing & Header Randomization

Every HTTP request from MyScraper is crafted to resemble a genuine browser session. Rather than using a fixed, library-default User-Agent string (which many sites instantly flag), we maintain a pool of real-world User-Agents and rotate them on every new session or request. In our Selenium driver setup, we inject randomized headers at launch while also rotating the User-Agent, we randomize other HTTP headers (Accept-Language, Accept-Encoding, Referer) on each new browser instance. This strategy disrupts simple fingerprinting heuristics that look for identical header sets across multiple connections.

2. Proxy Rotation & Geo-Distributed Requests

To avoid IP-based rate-limiting, CAPTCHAs, and outright bans, MyScraper integrates optional proxy rotation. The scraper can cycle through a list of residential or datacenter proxies optionally drawn from geo-distributed pools to spread its requests across diverse network endpoints, by associating each browser launch with a new proxy, we ensure that requests appear to originate from different IP addresses and regions. This not only thwarts simple rate-limits but also emulates real-world traffic distribution, making it much harder for sites to correlate and block scraper sessions

3. Human-Pattern Timing (Jittered Intervals, Viewport Resizing)

Bots are often detected by their unnaturally regular request patterns and mechanical scrolling behavior. To counteract this, MyScraper injects randomness at every stage:

- **Jittered Delays:** Instead of fixed `time.sleep()` intervals, we multiply each pause by

a random factor (e.g. `delay * (1 + random.uniform(-0.1, 0.3))`).

- **Variable Scroll Steps:** Our `perform_human_like_scrolling()` function breaks long scrolls into 3–15 random-sized increments, occasionally reversing direction slightly to simulate a human eye–scroll adjustment.
- **Viewport Resizing & Mouse Movements:** We occasionally resize the browser window by a few pixels and execute small, random mouse movements (via Selenium’s `ActionChains`) before clicks, further emulating human micro-behaviors.

These techniques collectively produce a non-deterministic scraping footprint that aligns closely with genuine browsing sessions, making timing-based detection far less reliable

By integrating all these different techniques together, header rotation, distributed networking, and behavioral randomness, MyScraper has near invisible state that tricks most websites that are famous for their detection and banning of Bots detection for most websites

8.3.3 Monitoring & Adaptive Retries

To ensure resilience in the face of transient failures, rate limits, or evolving page structures, our backend layers continuous monitoring and adaptive retry logic on top of the core scraping routines. By instrumenting every request and DOM interaction with error detection hooks and back-off policies, MyScraper can recover from temporary blocks or content anomalies without human intervention and surface only genuine, persistent failures to the user.

1. Error Detection

At the heart of our monitoring is a suite of checks that validate each response and extraction step. For HTTP-based scraping (BeautifulSoup) we inspect status codes treating 429 (Too Many Requests), 403 (Forbidden), or other 4xx/5xx errors as retrievable faults. In our Selenium flows, we catch network timeouts, JavaScript exceptions, empty or truncated DOMs, and even CAPTCHAs rendered as unexpected elements. Each exception funnels through our centralized **retry_logic** wrapper (see section 8.2.2.3), which logs the nature of the failure along with context (URL, selector, attempt count) before deciding whether to retry or escalate.

2. Automated Back-off

Rather than hammering a site at fixed intervals, we employ an exponential back-off strategy with randomized jitter. On each retry, the delay between attempts grows (e.g.

2 s - 4 s $\pm$ 20% - 8 s $\pm$ 20%) up to a configurable cap. This both reduces load on the target server and mimics how a human might pause longer after repeated failures, by capping the total number of retries and the maximum delay, we balance persistence against eventual timeouts or user expectations.

3. Re-authentication & Session Recovery

For sites that require login or issue short-lived tokens, our backend can automatically detect when a session has expired (e.g. login forms reappear, AJAX calls return “unauthorized,” or cookies are invalidated). In such cases, the scraper invokes a programmed re-authentication flow either re-submitting stored credentials via Selenium or refreshing an API token and then retries the original extraction. All credential handling occurs in memory and uses secure environment variables so that no sensitive data is ever persisted in plaintext.

4. Logging, Metrics & Visibility

Every retry and back-off event is logged with structured metadata (task ID, URL, error type, attempt number). We aggregate these into operational dashboards tracking failure rates, average retries per task, and sites that most commonly require extra back-off. This telemetry feeds back into our development cycle: if a particular domain’s defenses become too aggressive, we can adjust our delay curves, proxy pools, or even reach out to the site operator for an API partnership.

Together, these monitoring and adaptive retry mechanisms allow MyScraper to weather transient blocks and continue delivering reliable data while still respecting rate limits, avoiding undue server load, and surfacing only true structural or permission errors to the end user.

8.4 LLM-Driven Extraction Strategy

Building on our traditional **Beautiful Soup** and **Selenium-Driven** scrapers, we implemented a flexible, LLM-powered version that lets us declare what data we want instead of relying on the auto detection using the **Beautiful Soup** scraper or how to find it. By defining concise **JSON** schemas and driving extraction with natural-language prompts, we create self-documenting pipelines that gracefully adapt to ever evolving page layouts, dramatically cutting down on brittle selector maintenance, all while producing the best scrapping time with the minimum possibility of detection unlike our traditional and selenium-based scrapers.

8.4.1 LLM-Driven Scraping Workflow with Crawl4AI

1. Page fetch and Render

Crawl4AI first starts by spinning up a headless browser using **Playwright** and modified to merge with our **Selenium** strategy when the **JavaScript** execution is required, or issuing a simple HTTP GET via **Requests** for purely static pages to further help cut down the scrapping time the strategy is chosen based on what is present in the page that we want to scrap. Once it opens it captures the fully rendered **HTML/DOM** complete with any **AJAX-Loaded** content.

2. Chunking and context Management

Very large pages are split into overlapping chunks of **HTML** (e.g 2-4 KB each), so LLM's context window isn't flooded, which allows us to use free models like GROQ for Our LLM extractor that we are using, so having chunk instead of full snapshots we ensure that data spanning chunk boundaries (e.g a table header in on chunk and rows in the next) can still be recognized and extracted without faults.

3. Prompt Synthesis

For Each chunk, we assemble a two-part prompt:

- **System prompt** (“You are a JSON extractor. Given HTML, output exactly valid JSON conforming to this schema: ...”)
- **User prompt** (includes the chunk of HTML and a reminder of the schema properties, e.g. “Extract title, date, author...”).

We inject 2–3 few-shot examples right before the HTML on how similar snippets map to **JSON**.

4. LLM Invocation

Crawl4AI streams the prompt the LLM of our choice (**OpenAI**, **Gemma-7B**), either in one shot for small pages or chunk by chunk for larger ones, we also choose non-deterministic sampling with low temperature (0.2) to allow slight variation in selector interpretation, while still maintaining stability.

5. JSON Assembly and Deduplication

As partial **JSON** snippets return from each chunk. We incrementally merge them into a single object or array. If duplicates are found and return the same fields, we keep the first one to avoid duplication or for list-style fields, we append.

6. Validation and Normalization

Once all the chunks are processed, we run the merged output through a **JSON** Schema validator, detecting any missing fields required, that then triggers an adaptive retry (Section 8.4.3), we re-prompt the LLM. This time appending a note like “the field “author” did not appear please locate it and then re-extract, this message is built dynamically from the error logs, so no manual interventions are needed.

7. Persistence & Export

Once a valid **JSON** is loaded into python via `JSON.LOADS`, then converted into native objects, we take the results of the raw JSON and store it in our PostgreSQL and also dump to Excel format for the user to download and use.

This fully orchestrated dance between a lightweight crawler, smart chunking, prompt engineering, and adaptive retries is what makes our AI-powered scraper both **powerful, Fast** and **resilient**.

8.4.2 LLM-Driven Extraction Strategy

At the heart of our AI module is a prompt-based extractor. Rather than manually crafting up a **JSON** request made up of **CSS** or **XPATHS** for our backend to deconstruct and use, we maintain a **JSON** schema created by the scrapping prompt as present in `APPENDIX()`, when a page is loaded successful while evading detection using the previously mentioned techniques (Section 8.4) we generate a prompt like the following: Using the HTML (or rendered DOM) below, extract the page’s **title**, **date**, and **author** as a JSON object matching this schema. Normalize whitespace and strip HTML entities. We don’t expect the user to know what the DOM or even what HTML means, so we predefine these actions with each request that is passed from the front end, but keep the middle part of what the user desires from this extraction.

We also include two or three few-shot examples, that are small snippets (Screenshots) showing the before/after HTML-to-Json mappings. This teaches the model to:

- **Scan the DOM** For text patterns (e.g. headings or “by...” phrases).
- **Map semantic labels** (title, data, author) to corresponding text.
- **Clean and normalize** the raw string removing extra spaces, trailing icons, non-printable or invisible characters.

- **Emit strict JSON** that conforms exactly to the schema wanted by the user.

Because the LLM reasons over semantics rather than brittle and changing CSS paths, even if a headline moves from eg: `<h1 class="main-title">` to `<h2 id="post-heading">`, the is still able to recognize it as the title. In practice this shift from “code-centric” to “schema-centric” extraction slashes the number of manual updates required whenever a site tweaks its markup and minimizes the number of extraction errors.

8.4.3 Integration with Crawl4AI & FastAPI

We orchestrate the entire process inside of a **FastAPI** backend using **Crawl4AI’s LLMExtractionStrategy** function that is imported from it, when an AI Scrape task arrives:

1. **FastAPI** spawns an asynchronous background job.
2. **Crawl4AI** launches headless Chromium (if needed), or fetches static HTML, and feeds it to the LLM in digestible chunks.
3. The **LLM’s JSON outputs** stream back into our code.

Once Crawl4AI signals completion, we:

- **Parse** the raw strings with `json.loads`.
- **Validate** against our JSON Schema via `python-jsonschema`, catching missing or malformed fields.
- **Persist** to PostgreSQL (via `SQLAlchemy`) and optionally write an Excel file for easy viewing.

Because everything is async **FastAPI’s** event loop never blocks on I/O—hundreds of concurrent jobs can proceed without exhausting thread pools (Section Workflow and software used).

Pydantic models enforce that incoming requests and outgoing results always match our type definitions, giving us confidence that the data shape remains consistent throughout.

8.4.4 Adaptivity & Self-Healing

One of the most exciting outcomes of our project is the built-in resilience to layout drift and Hallucination. If a required field is missing or fails schema validation, we:

1. **Trigger a corrective prompt:** getting the error message of the missing field we can easily build a corrective prompt e.g. “It looks like the author field wasn’t found. Here are two examples of how the author appears on similar pages: Please try again.”

2. **Few-shot re-extraction:** the LLM uses these new examples to adjust its semantic matching, often recovering the missing data without any manual interventions.
3. **Log corrections:** We store both the original and corrected schemas in our audit logs. Over time, we automatically incorporate the most frequent corrections as additional few-shot corrections, effectively letting the scraper “learn” each site’s subtle quirks and challenges

This self-healing loops means that minor HTML tweaks and even class-name changes no longer require us to dive into the fragile and dynamic DOMS of some websites. Instead, our AI-driven pipeline dynamically reasons about content semantics, keeping extraction reliable far longer between manual updates.

8.5 Application Development and System Implementation

Our extension is designed in a way that any user can install it on their browser and use it without any coding experience and knowledge, so we need to describe each component going from the browser extension to the **Frontend** and the **backend** and how they are engineered to work seamlessly together, from local development through production deployment.

8.5.1 Extension Architecture & Tech Stack

MyScraper’s browser extension is implemented as a standards-compliant **WebExtension** (Manifest V3), guaranteeing seamless operation on both Chrome and Firefox. We employ a combination of **Parcel** to handle the content-script pipelines, automatically detecting file dependencies (including injected CSS and WebAssembly modules) and performing zero-configuration, code splitting, and tree-shaking for production. While also providing one of the most essential functionalities that often slow down extension development which is hot reloading and bundling. This is being used hand in hand with **Tailwind CSS** that provides us with a utility-first styling layer, enabling our floating browser extension to be responsive and consistent without the need to write any custom **CSS** to fix ([Appendix F](#)).

Under the hood:

1. **Background Script:** Runs as a service worker, holding persistent **Websocket** or fetch connections to our **FastAPI** backend and managing task lifecycles
2. **Content Script:** Injected into each page; handles element-selection overlays, captures DOM events computes selector and renders live previews.
3. **Popup UI:** A small parcel extension app is server from the extension’s own origin

(chrome-extension://...), letting users create their account easily, letting users review and submit scraping instructions.

4. **Plain JavaScript:** Lightweight and performant DOM manipulation, providing maximum flexibility and minimal runtime overhead

8.5.2 Front-end Workflows (UI, Element Selection & Export)

When a user activates the extension, our **Parcel** popup displays the current page URL automatically and a “**Select extraction Strategy**” button. Clicking it injects an overlay into the page to do the following (For the selenium based Strategy):

1. **Hover & Highlight:** A content script attaches **mouseover/mouseout** listeners to every element. As the cursor moves, the target element is outlined with a semi-transparent green mask, giving clear visual feedback.
2. **Click to capture:** On click, we prevent the default navigation and stop propagation, then pass the raw **Event.target** node to our selector-generator module.
3. **Selector Computation:**
 - **CSS Selector:** We walk up the DOM tree collecting tag names, IDs, and classes e.g. `article.post:nth-child(3) > h2.title`.
 - **XPath:** A recursive utility builds an absolute XPath by recording each node's position among its siblings `Appendix()`
 - We choose the most specific selector (XPath when IDs are missing, CSS otherwise) to maximize resilience.
4. **Live Preview:** Immediately after capture, the content script executes **`document.evaluate(xpath, ...)` or `document.querySelector(css)`** to extract the node's **`textContent`**, then displays it in a small tooltip
5. **Recipe Assembly & Submission:** each selector (with its attribute type e.g. **Text, Hred, Src**, etc.) is appended to a JSON “**recipe.**” Once the users finishes selection and creates the scrap task, we Post this recipe (plus any **JSON** schema definitions) to our backend FastAPI endpoints over HTTPS.
6. **Result Fetch & Export:** Once the task is queued we tag its current progress and once the progress is indicated as done, a trigger is fired, and the user is sent an email informing the user that the scrapping job is done, while also containing his scrapped websites Excel in **.XLSX**, the user can also view his history of tasks with the extension allowing for easy downloads and history tracking.

8.5.3 Backend Integration (FastAPI, Task Queue, Data Persistence)

Our backend is built on **FastAPI**, chosen for its combination of high-performance automatic API documentation (**Built in Swagger**), while also reducing development friction, enabling rapid prototyping, and ensures reliability through built-in validation using **Pydantic** schemas.

When a scrape job is received, the backend leverages Python's built-in **concurrent.futures.ThreadPoolExecutor** to enqueue and manage tasks efficiently. This built-in concurrency model removes external dependencies and streamlines task management, allowing concurrent processing of multiple scraping tasks.

1 Defining Request and Response Schemas

We start by declaring **Pydantic** models for the incoming request and the task status/response (Appendix 8.5.3.1), These models automatically power **FastAPI's** request validation and generate an **OpenAPI** spec.

2 Enqueueing Jobs with ThreadPoolExecutor

To keep the system responsive without introducing the operational complexity of an external message broker, we rely on Python's own **concurrent.futures.ThreadPoolExecutor**. Whenever a user submits a scrape request, **FastAPI's BackgroundTasks** helper immediately hands that job off to the thread pool and returns an **HTTP 200 in a few milliseconds**. The size of the pool (**max_workers**) is calibrated to the CPU count and to the I/O-bound nature of scraping: on our eight-core staging box, a pool of 20 threads saturates the network long before it saturates the processor, giving us plenty of parallelism without thrashing the scheduler.

Inside each worker thread we run a purpose-built “runner” function **run_selenium_scrape** for classical automation and **async_run_ai_scrape** for the LLM path. The runner marks its task row in Postgres as **running** the moment it starts; that simple flag powers the real-time progress indicator in the extension popup. All downstream operations are wrapped in a broad try/except so no exception can bubble out and kill the thread. If an error does occur we capture the full traceback, trim it to a few kilobytes, and store it in the **error_message** column while flipping the status to **failed**. A clean run, on the other hand, ends with the status set to **completed** plus a timestamp so the UI can show “Finished in 3 s”.

The payload itself often a list of dictionaries representing products, quotes or other entities is inserted wholesale into a JSONB column named result. That design choice means we avoid

schema migrations every time a site adds a new optional field; instead, we only migrate when core metadata changes. When we do need to evolve the schema say, to add a **source_url** or **scraped_at** key Alembic handles the DDL in a single command, keeping development, staging and production in **lock-step**. With this pattern we get strong consistency guarantees from **SQLAlchemy**, the flexibility of **schemaless** result storage, and a minimal operational footprint (Appendix 8.5.3.2).

3 Scraping Runners & Error Handling

During execution each scraper runner behaves like a miniature finite-state machine whose state is persisted in Postgres. The moment the worker thread (or event-loop for the AI path) starts, the runner updates its row to **running** and stamps a **Started_at** timestamp; this write happens inside the same database session that created the task, so it is immediately visible to the front-end's polling endpoint and can drive a real-time progress badge in the extension popup.

As the job proceeds the runner appends incremental details such as the number of pages visited, the count of elements extracted, or the URL currently being processed to a transient *log* column. Those updates are committed every few seconds with **session.flush()** to avoid holding long-lived locks yet still give operators visibility in Grafana.

All potentially fragile operations browser launch, page navigation, HTML parsing, LLM calls, **DataFrame** export are wrapped in a broad try / except Exception block. If any unhandled error surfaces, the runner traps it, trims the traceback to a sane length (≈ 8 KB), writes that string into **error_message**, flips *status* to **failed**, and records **finished_at**. Because this happens inside the same transaction, the status and the diagnostic text are atomically consistent; the UI will never show “**completed**” for a task that crashed.

On a clean path the runner serialises the extracted **list-of-dicts** into the **JSONB result** column, sets *status* to **completed**, and commits once more. The calling thread (or coroutine) then returns control to the **ThreadPoolExecutor**, freeing up a slot for the next queued job. This disciplined life cycle that goes from **running** to **completed/failed** with mandatory error capture guarantees that no worker can die silently and that every task finishes in a well-defined state that downstream tooling and users can trust. (Appendix 8.5.3.3)

4 Persisting Results in PostgreSQL

We use **SQLAlchemy ORM** models to represent our database tables, with the **result** column designed to store raw **JSON** arrays of scraped records (Appendix 8.5.3.4). Schema

migrations are managed using Alembic; for instance, a typical migration to introduce the **result** column would be handled seamlessly through Alembic's migration framework.

5 Validation & Normalization

After a scraping job finishes whether it came from Selenium or from the LLM-powered **Crawl4AI pipeline** we never push the raw payload straight into our database. Instead, we pass it through a three-step “clean room”.

First, we make sure the data is a Python structure. If the extractor already returned a list or dictionary we keep it; if it produced a string, we attempt to parse it with **json.loads**. That single step weeds out the common situation where an LLM has wrapped the JSON in Markdown back-ticks or added a polite sentence before the array.

Once the value is a bona-fide Python object, each dictionary is run through a **Pydantic** model that mirrors our **ScrapeTaskWithResult** schema. **Pydantic** is lenient about obvious type coercions it will happily turn the string “123” into an integer for a numeric field but it raises a **ValidationError** the moment a required key is missing, or a value is hopelessly incompatible. For certain nested fields we add custom validators: for example, we assert that a price string includes a recognised currency symbol, and we strip stray whitespace from titles. Anything that fails validation is collected in an “errors” bucket so that one bad row does not kill the whole task.

Records that pass validation are then enriched with a few extra attributes: we stamp them with the source URL, the extraction method (custom, AI, or Selenium) and a timestamp. At this point they are guaranteed to share a single, canonical shape, so every downstream consumer from the Excel export routine to our BI dashboards can rely on consistent keys and types.

All validation failures are logged with structured detail: the offending task ID, the exact field that blew up, the expected versus received type, and the first couple of hundred characters of the payload that caused trouble. These logs feed Prometheus counters, which means we can see at a **glance** if a target website silently changed its HTML and validation errors for, say, the price field suddenly spike. In the AI flow we even have an auto-retry path: if the first extraction yielded unusable **JSON** we can re-prompt the model with stricter instructions asking it to “return *only* valid JSON conforming to the schema above”.

The net effect is that malformed data never bleeds into Postgres, user downloads stay squeaky-clean, and any site-breaking change shows up first as a validation metric instead of a production incident.

6 Exposing Status & Fetching Results

After a scrape request has been enqueued the front-end regularly asks the back-end for the current state of that job. We expose a single idempotent **GET /scrape/{task_id}** endpoint that reads the row from Postgres and serialises it to **JSON**. Because the underlying **SQLAlchemy** model is registered with **Pydantic**, **FastAPI** automatically converts the row into a **strongly-typed** response object; every field **status**, **created_at**, **finished_at**, **error_message**, **result** is therefore guaranteed to be present and to respect its declared type.

Polling cadence is lightweight: the popup starts at one request per second and slows to every five seconds once the task enters a terminal state. For higher-end dashboards the same data are streamed over a WebSocket channel; new rows emitted by **SQLAlchemy**'s **after_commit** event trigger a small push message that the browser receives instantly, giving us “live” progress bars without the overhead of constant **HTTP round-trips**.

The consumer logic is simple:

- **running** - show a spinner plus any incremental log lines.
- **failed** - display the truncated traceback from **error_message** so a user sees exactly what went wrong.
- **completed** - enable the **Download** button; the **result** column now contains the final list-of-dictionaries payload ready for CSV/Excel export.

With this pattern the client always has an authoritative, race-free view of every scrape job, whether it finishes in 100 ms or 10 minutes.

7 Exposing Status & Fetching Results

When the user presses “**Run**” in the browser extension the UI issues a JSON POST that specifies the target URL, the chosen scraping engine and any selectors or LLM instructions. **FastAPI** receives that request, inserts a new row in the **scrape_tasks** table with the status set to **pending**, and replies immediately with the primary-key ID so the client never waits on a network round-trip for the heavy work.

The moment the response is sent, **FastAPI**'s **BackgroundTasks** hook hands the job object to the shared **ThreadPoolExecutor**. A free worker thread marks the task **running** and launches the appropriate runner:

- If the method is **selenium** the runner spins up a headless Chrome session, executes any pre-actions (scroll, clicks, pagination), extracts the requested elements and collapses them into a list of dictionaries.
- If the method is **ai** the runner invokes our **Crawl4AI** helper which in turn drives Playwright and the **Groq LLM** to interpret the page, extract the schema-compliant **JSON** table and return the structured rows.

Regardless of path, the runner loads the raw payload into the corresponding **Pydantic model**. This instantaneously checks that every required key exists and that field types match the declared schema; any mismatch raises a **ValidationError** which we catch, log and surface as a **failed** status with a concise error message.

A successful validation results in an update to Postgres: the **result** JSONB column receives the exact array of records, the status flips to **completed**, and a **finished_at** timestamp is set.

Finally the extension (or any other client) polls or listens over WebSockets to **/scrape/{id}**. If the task is still **running** it shows a spinner; when the status is **completed** it renders the returned JSON, enables the Download Excel button and displays the total duration. Should the status be **failed**, the UI presents the captured traceback so users can refine their selectors or prompt and try again.

With this pipeline each scrape request moves seamlessly from submission through execution to retrieval, giving immediate feedback, robust validation and durable storage without adding external queueing or orchestration layers.

8.6 Testing, CI/CD & Deployment

To ensure the reliability and maintainability of our scraping platform, we established a lightweight but effective testing and deployment workflow anchored around GitHub.

1. Testing Strategy

Throughout development, we applied a pragmatic, layered testing approach:

- **Unit Testing:** Core helper functions such as **find_element_with_retry**, **perform_human_like_scrolling**, and **JSON** validation pipelines were covered with targeted unit tests using **Pytest**. This guaranteed that small logic components behaved consistently across changes.

- **Integration Testing:** We conducted full end-to-end tests where we simulated real scraping jobs, from sending a POST request to the **FastAPI** backend, launching a browser, performing scraping, and verifying that the results were correctly inserted into the **PostgreSQL** database.
- **Manual UX Testing:** Given the critical role of the Chrome extension, we manually tested:
 - Element selection overlays across a wide variety of websites.
 - Exporting collected selectors into proper JSON.
 - Submission workflows to ensure seamless backend communication.
 - Cross-browser behavior (testing both Chrome and Firefox under Manifest V3).

Testing was especially important for our Selenium-driven workflows, where minor inconsistencies in browser rendering or timing could cause flakiness. Our retry wrappers (**retry_logic**) and humanized scrolling helped eliminate a large class of such brittle behaviors even before formal testing.

2. CI/CD Workflow

We adopted GitHub as our central code management and automation hub:

Version Control: Every new feature (or fix) was developed on a separate Git branch following a simple Gitflow strategy (feature branches merged into main after review).

Continuous Integration (CI):

For every pull request we perform:

Linting checks (**flake8**, **black**) ensured stylistic consistency.

Pytest suites were run inside a clean Python environment using GitHub Actions, automatically detecting regressions early.

Continuous Deployment (CD):

Although not using automated production deployments (due to scraping often requiring manual oversight), we streamlined releases by:

Building the **FastAPI** backend into a self-contained **Docker** image.

Preparing the Chrome Extension as a zipped production package (using Parcel bundler's optimized build mode).

Uploading new extension versions to a private GitHub release repository for easy internal installation or submission to the Chrome Web Store, if desired.

3. Deployment Environments

- **Backend Server:** During testing, we deployed **FastAPI** behind a simple **Gunicorn** application server with **Uvicorn workers**, fronted by **NGINX** for SSL termination and load balancing.
- **Database:** **PostgreSQL** instances were provisioned via **Docker Compose** locally and on remote servers, ensuring that migrations and schema setups remained consistent across environments.
- **Frontend:** Our Chrome extension is loaded unpacked during development, and production builds are compressed for distribution (still not available on the store).

For local development, we also configured **.env** files to manage environment-specific settings like API keys, proxy configurations, database URLs, and crawler parameters.

9. DATA PRESENTATION, RESULTS, INTERPRETATION

9.1 Overview of the Data Collected

Over the course of our capstone project, we systematically **scrape** diverse types of web pages to validate the robustness of versatility of our scraping system. Our targets included:

Static websites with server-rendered HTML (e.g., QuotesToScrape.com)

Dynamic, JavaScript-driven websites requiring full DOM interaction and scrolling

Semi-structured pages with inconsistent layouts ideal for LLM-powered extraction

Depending on the nature of the site, we used one of three main approaches:

- **Traditional static** parsing using **BeautifulSoup** (bs4)
- **Dynamic scraping** with **Selenium WebDriver** and human-like simulation
- **Schema-based, AI-driven scraping** powered by **Groq LLaMA 3 models** through **Crawl4AI**

Each dataset collected during tests included common fields such as **text**, **author**, **tags**, **timestamps**, and optional metadata like **page numbers** or **categories**. The extracted data was

processed, normalized, and stored systematically in our PostgreSQL database, with optional Excel exports for immediate review.

Our multi-layered scraping approach ensured that we could robustly extract data from both simple static pages and highly dynamic, content-rich environments that would typically resist traditional scraping methods.

9.2 Data Presentation

The main websites that we are going to test our scrapper on are:

<https://quotes.toscrape.com/> which is mostly a static website that is used to test scrappers that developers are making, such site was essential in the first tests of the scrapper as scrapping time on it was minimal while it maintains common tags that all websites use.

<https://www.amazon.com/> which is a well-known website that uses anti scrapping methods, click to load more button, dynamic tags, and much more that really tests our scrappers capabilities, with the addition of infinite scroll paired with load more buttons.

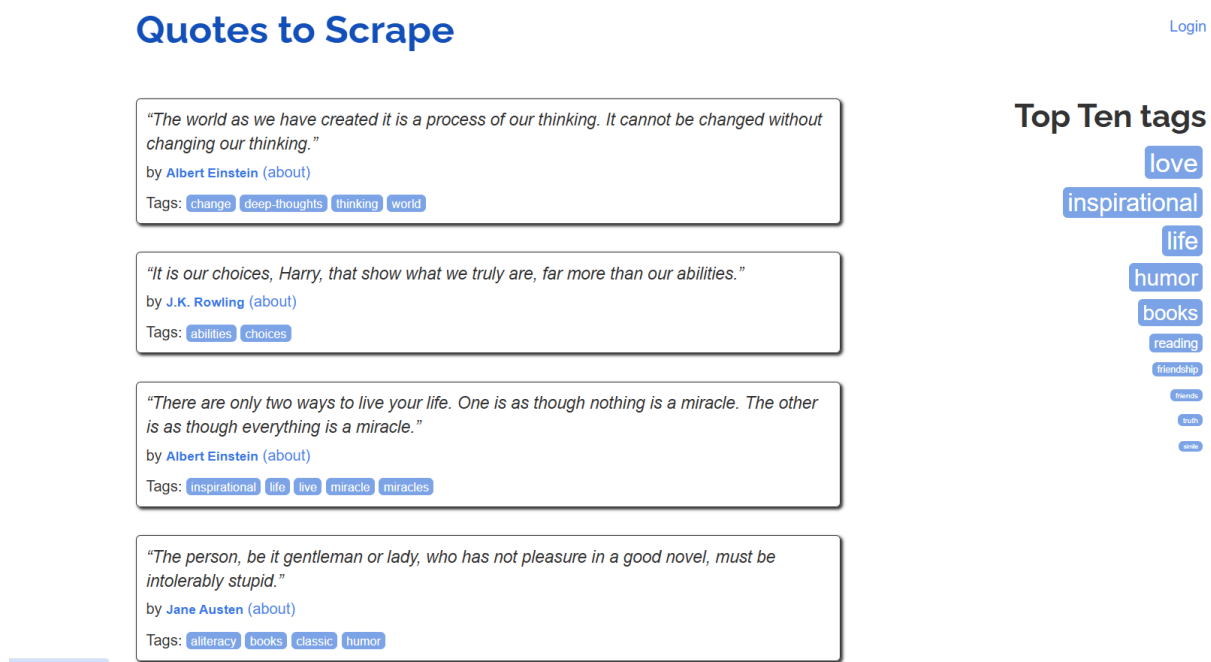


Figure 4- Target Website for static scraping

9.2.1 Static Scraping (BeautifulSoup)

In the first phase of testing, we targeted static websites where all content is fully embedded in the server's HTML response. Using BeautifulSoup (bs4), we parsed the page DOM, located quote texts and authors via CSS selectors, and extracted clean strings.

Figure 5- Json Request for the backend for Static Website

```
curl -X 'POST' \
  'http://127.0.0.1:8000/scrape/' \
  -H 'accept: application/json' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJCT1VEQUQ1L0JlIiwiaWF0IjE3NDU2Mjc0OTN9.s18oZY2J0Vei1sbXr26i03ff_pljJE7Qixd1YSgQyapM' \
  -H 'Content-Type: application/json' \
  -d '{
    "url": "https://quotes.toscrape.com/",
    "method": "selenium",
    "parameters": ""
  }'
```

Figure 6- Token Validation and Initial Response from the backend

```
INFO: 127.0.0.1:50365 - "POST /scrape/ HTTP/1.1" 200 OK
2025-04-26 00:56:43,689 [INFO] app.scraping.selenium_scraper: Starting (BeautifulSoup) static scrapping for URL: https://quotes.toscrape.com/
2025-04-26 00:56:43,689 [INFO] app.scraping.selenium_scraper: Starting (BeautifulSoup) static scrapping for URL: https://quotes.toscrape.com/
2025-04-26 00:56:43,689 [DEBUG] app.scraping.selenium_scraper: Chrome options set.
2025-04-26 00:56:43,689 [DEBUG] app.scraping.selenium_scraper: Chrome options set.
2025-04-26 00:56:43,690 [INFO] WDM: ===== WebDriver manager =====
```

Figure 7- Backend Logs Indicating start of our Scraper and finish

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	text	quote	author	quote href	tags	tag href	tag	tag href (2)	tag (2)	tag href (3)	tag (3)	tag href (4)	tag (4)
2	â€œThe world as we have created it is a process by Albert Einstein (about)		Albert Eins	https://q	Tags: chan	https://q	change	https://q	deep-thou	https://q	thinking	https://q	world
3	â€œIt is our choices, Harry, that show what we t by J.K. Rowling (about)		J.K. Rowlin	https://q	Tags: abilit	https://q	abilities	https://q	choices				
4	â€œThere are only two ways to live your life. One by Albert Einstein (about)		Albert Eins	https://q	Tags: inspl	https://q	inspiration	https://q	life	https://q	live	https://q	miracle
5	â€œThe person, be it gentleman or lady, who ha: by Jane Austen (about)		Jane Auste	https://q	Tags: aliter	https://q	alteracy	https://q	books	https://q	classic	https://q	humor
6	â€œImperfection is beauty, madness is genius a by Marilyn Monroe (about)		Marilyn Mo	https://q	Tags: be-y	https://q	be-yoursel	https://q	inspirational				
7	â€œTry not to become a man of success. Rather by Albert Einstein (about)		Albert Eins	https://q	Tags: adult	https://q	adulthood	https://q	success	https://q	value		
8	â€œIt is better to be hated for what you are than by Andr�� Gide (about)		Andr�� Gi	https://q	Tags: life lc	https://q	life	https://q	love				
9	â€œI have not failed. I've just found 10,000 ways by Thomas A. Edison (about)		Thomas A.	https://q	Tags: edisc	https://q	edison	https://q	failure	https://q	inspiration	https://q	paraphrased
10	â€œA woman is like a tea bag; you never know h by Eleanor Roosevelt (about)		Eleanor Ro	https://q	Tags: misa	https://q	misattributed-eleanor-roosevelt						
11	â€œA day without sunshine is like, you know, nig by Steve Martin (about)		Steve Mart	https://q	Tags: humi	https://q	humor	https://q	obvious	https://q	simile		
12													

Figure 8- Filtered and structured results of our static scraper

Extraction was instantaneous (under 1 second), with near-perfect accuracy assuming no major changes in the HTML structure, but it falls short in terms of clicking elements to go to the next page, scrolling to load more elements, and in terms of detection avoidance.

9.2.2 Dynamic Scraping (Selenium plus BeautifulSoup)

Using our dynamic scrapper requires us to choose the elements that we want to scrap and the actions that we want our scrapper to take depending on the website.

For <https://quotes.toscrape.com/> we use apply the following instructions:

Choosing author and quotes as the following:

The image displays a browser extension interface for selecting elements on a website. On the left, four quote cards are shown. The first card is highlighted with a green dashed border. The right panel shows the configuration for the selected element. It includes a 'Max Pages' input set to 10, an 'Infinite Scroll' section with a 'x scroll' input set to 0.5 and a 's pause' button, a green '+ Create Task' button, a 'Stop Selecting' button, a 'Label for Selected Element' input field containing 'quote' with an 'Add' button, an 'Action Type' dropdown menu set to 'Click', a 'Selected Elements' section with a list icon and the text 'CSS selectors from the page', and a 'Payload Preview' section showing a JSON object: { "url": "https://quotes.toscrape.com/", ... }.

Figure 9- Element Selection using Our Browser Extension

And then choosing the pagination button as follows:

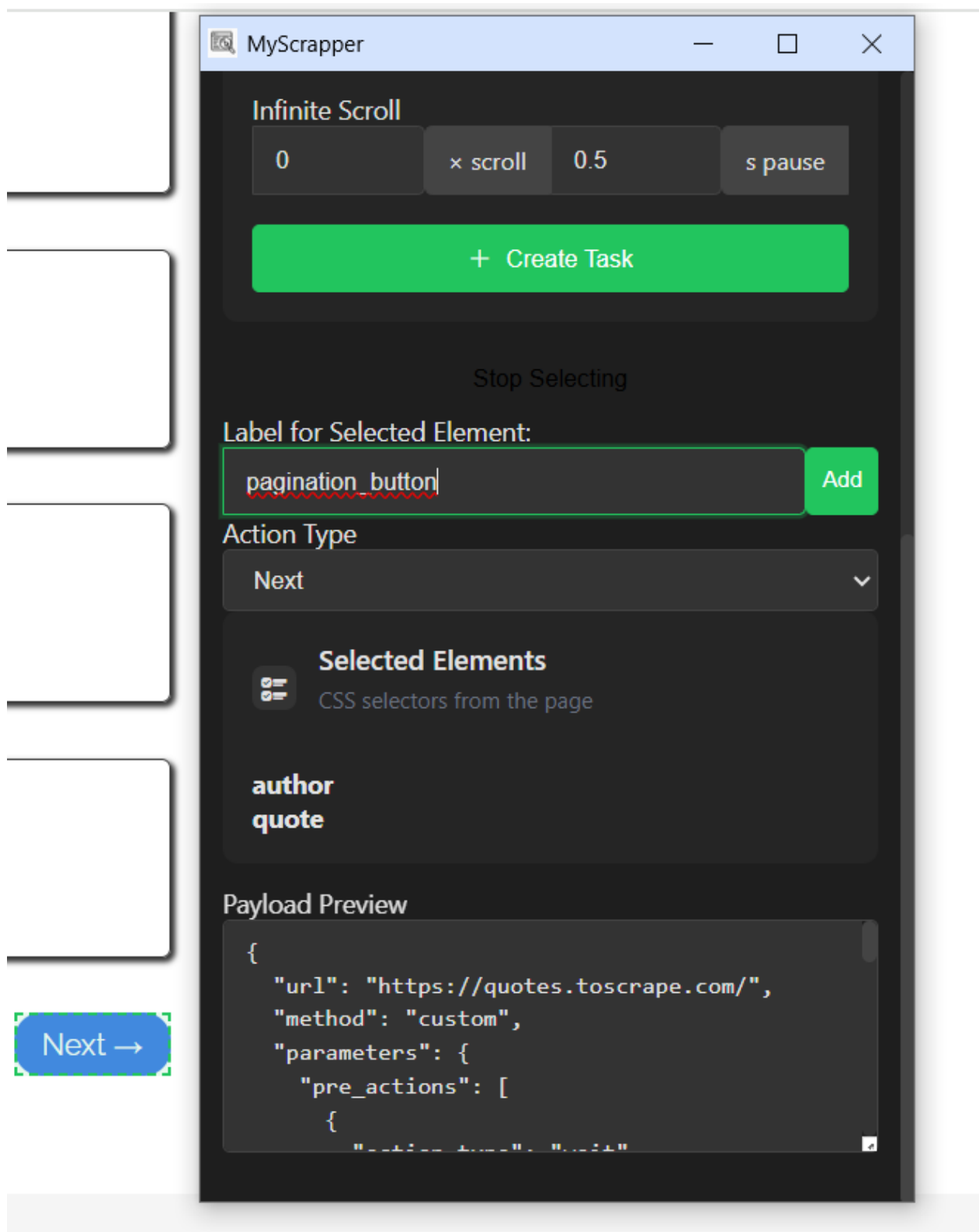


Figure 10- Element labeling and automatic JSON request building using our Browser Extension

Yielding us the following instructions:

No parameters

Request body required application/json

```
{
  "url": "https://quotes.toscrape.com/",
  "method": "custom",
  "parameters": {
    "pre_actions": [
      {
        "action_type": "wait",
        "delay": 3
      }
    ],
    "max_pages": 10,
    "next_button": {
      "selector": "pagination_button",
      "selector_type": "css"
    },
    "author": {
      "selector": "body > div.container > div.row > div.col-md-8 > div.quote > span:nth-of-type(2) > small.author",
      "selector_type": "css",
      "attribute": "text",
      "multiple": true,
      "limit": 10
    },
    "quote": {
      "selector": "body > div.container > div.row > div.col-md-8 > div.quote > span.text",
      "selector_type": "css",
      "attribute": "text",
      "multiple": true,
      "limit": 10
    }
  }
}
```

Execute Clear

Figure 11- Full instructions After selecting targeted scrapping elements

```
INFO: 127.0.0.1:50756 - "POST /scrape/ HTTP/1.1" 200 OK
2025-04-26 01:11:00,297 [INFO] WDM: ===== WebDriver manager =====
2025-04-26 01:11:03,512 [INFO] WDM: Get LATEST chromedriver version for google-chrome
2025-04-26 01:11:03,513 [DEBUG] urllib3.connectionpool: Starting new HTTPS connection (1): googlechromelabs.github.io:443
2025-04-26 01:11:03,655 [DEBUG] urllib3.connectionpool: https://googlechromelabs.github.io:443 "GET /chrome-for-testing/latest-patch-versions-per-build.json HTTP/1.1" 200 10888
2025-04-26 01:11:03,668 [INFO] WDM: Get LATEST chromedriver version for google-chrome
2025-04-26 01:11:03,669 [DEBUG] urllib3.connectionpool: Starting new HTTPS connection (1): googlechromelabs.github.io:443
2025-04-26 01:11:03,811 [DEBUG] urllib3.connectionpool: https://googlechromelabs.github.io:443 "GET /chrome-for-testing/latest-patch-versions-per-build.json HTTP/1.1" 200 10888
2025-04-26 01:11:03,813 [INFO] WDM: Driver [C:\Users\zakiie\wdm\drivers\chromedriver\win64\135.0.7049.114\chromedriver-win32\chromedriver.exe] found in cache
2025-04-26 01:11:03,813 [DEBUG] selenium.webdriver.common.driver_finder: Skipping Selenium Manager; path to chrome driver specified in Service class: C:\Users\zakiie\wdm\drivers\chromedriver\win64\135.0.7049.114\chromedriver-win32\chromedriver.exe
2025-04-26 01:11:03,910 [DEBUG] selenium.webdriver.common.service: Started executable: `C:\Users\zakiie\wdm\drivers\chromedriver\win64\135.0.7049.114\chromedriver-win32\chromedriver.exe` in a child process with pid: 21032 using 0 to output -3
2025-04-26 01:11:04,412 [DEBUG] selenium.webdriver.remote.remote_connection: POST http://localhost:50763/session {'capabilities': {'firstMatch': [{}], 'alwaysMatch': {'browserName': 'chrome', 'pageLoadStrategy': <PageLoadStrategy.normal: 'normal'>, 'goog:chromeOptions': {'excludeSwitches': ['enable-automation'], 'useAutomati
```

Figure 12 Backend Logs Indicating start of our Dynamic Scraper and its Actions

Returning the following results to us:

	A	B	C	D	E	F	G	H
1	quote	author	quote href					
2	The world as we have created it is a process of our thinking. It c	by Albert Einstein	https://quotes.toscrape.com/author/Albert-Einstein					
3	It is our choices, Harry, that show what we truly are, far more tl	by J.K. Rowling	https://quotes.toscrape.com/author/J-K-Rowling					
4	There are only two ways to live your life. One is as though nothir	by Albert Einstein	https://quotes.toscrape.com/author/Albert-Einstein					
5	The person, be it gentleman or lady, who has not pleasure in a g	by Jane Austen	https://quotes.toscrape.com/author/Jane-Austen					
6	Imperfection is beauty, madness is genius and it's better to be i	by Marilyn Monroe	https://quotes.toscrape.com/author/Marilyn-Monroe					
7	Try not to become a man of success. Rather become a man of i	by Albert Einstein	https://quotes.toscrape.com/author/Albert-Einstein					
8	It is better to be hated for what you are than to be loved for wha	by Andr�� Gide	https://quotes.toscrape.com/author/Andre-Gide					
9	I have not failed. I've just found 10,000 ways that won't work.��	by Thomas A. Edison	https://quotes.toscrape.com/author/Thomas-A-Edison					
10	A woman is like a tea bag; you never know how strong it is until	by Eleanor Roosevelt	https://quotes.toscrape.com/author/Eleanor-Roosevelt					
11	A day without sunshine is like, you know, night.��	by Steve Martin	https://quotes.toscrape.com/author/Steve-Martin					
12	This life is what you make it. No matter what, you're going to me	by Marilyn Monroe	https://quotes.toscrape.com/author/Marilyn-Monroe					
13	It takes a great deal of bravery to stand up to our enemies, but j	by J.K. Rowling	https://quotes.toscrape.com/author/J-K-Rowling					
14	If you can't explain it to a six year old, you don't understand it y	by Albert Einstein	https://quotes.toscrape.com/author/Albert-Einstein					
15	You may not be her first, her last, or her only. She loved before	by Bob Marley	https://quotes.toscrape.com/author/Bob-Marley					
16	I like nonsense, it wakes up the brain cells. Fantasy is a necess	by Dr. Seuss	https://quotes.toscrape.com/author/Dr-Seuss					
17	I may not have gone where I intended to go, but I think I have en	by Douglas Adams	https://quotes.toscrape.com/author/Douglas-Adams					
18	The opposite of love is not hate, it's indifference. The opposit	by Elie Wiesel	https://quotes.toscrape.com/author/Elie-Wiesel					
19	It is not a lack of love, but a lack of friendship that makes unhap	by Friedrich Nietzsche	https://quotes.toscrape.com/author/Friedrich-Nietzsche					
20	Good friends, good books, and a sleepy conscience: this is the	by Mark Twain	https://quotes.toscrape.com/author/Mark-Twain					
21	Life is what happens to us while we are making other plans.��	by Allen Saunders	https://quotes.toscrape.com/author/Allen-Saunders					
22	I love you without knowing how, or when, or from where. I love y	by Pablo Neruda	https://quotes.toscrape.com/author/Pablo-Neruda					
23	For every minute you are angry you lose sixty seconds of happir	by Ralph Waldo Emerson	https://quotes.toscrape.com/author/Ralph-Waldo-Emerson					
24	If you judge people, you have no time to love them.��	by Mother Teresa	https://quotes.toscrape.com/author/Mother-Teresa					
25	Anyone who thinks sitting in church can make you a Christian n	by Garrison Keillor	https://quotes.toscrape.com/author/Garrison-Keillor					
26	Beauty is in the eye of the beholder and it may be necessary fro	by Jim Henson	https://quotes.toscrape.com/author/Jim-Henson					
27	Today you are You, that is truer than true. There is no one alive	by Dr. Seuss	https://quotes.toscrape.com/author/Dr-Seuss					
28	If you want your children to be intelligent, read them fairy tales.	by Albert Einstein	https://quotes.toscrape.com/author/Albert-Einstein					

Figure 13- Filtered and structured results of our Dynamic scrapper

From the results we can see that the scrapper completed the whole 10 pages that <https://quotes.toscrape.com/> has to offer.

100R x 4C	
2	The world as we have created it is a process of our thinking. It
3	It is our choices. Harry. that show what we truly are. far more

Figure 14- Total scrapped Quotes from our Dynamic Scrapper

But our scraper can do way more, so running the following tests on Amazon.com will result in the following:

	A	B	C	D	E	F	G	H
1	Link	Name	Rating	Price				
2	https://www.amazon.com/sspa/click?ie=	2025 Laptop Computer 14 Inc	3.6 out of 5 stars	\$269.00				
3	https://www.amazon.com/sspa/click?ie=	ACEMAGIC	5.0 out of 5 stars	\$359.99				
4	https://www.amazon.com/i9-13900H-Pr	acer	4.2 out of 5 stars	\$999.99				
5	https://www.amazon.com/Acer-Chrome	acer	4.4 out of 5 stars	\$172.00				
6	https://www.amazon.com/Lenovo-IdeaP	Lenovo	4.4 out of 5 stars	\$211.32				
7	https://www.amazon.com/sspa/click?ie=UTF8&spc=MT01MDg3MjQ1		4.5 out of 5 stars	\$199.00				
8	https://www.amazon.com/BLUEING-Ligh	14" FHD Lightweight Laptop,	3.7 out of 5 stars	\$179.99				
9	https://www.amazon.com/Lenovo-IdeaP	Lenovo	4.4 out of 5 stars	\$449.99				
10	https://www.amazon.com/HP-Chromebc	HP	4.3 out of 5 stars	\$178.97				
11	https://www.amazon.com/jumper-5305L	jumper	4.2 out of 5 stars	\$119.99				
12	https://www.amazon.com/ASUS-Vivoboc	ASUS	4.0 out of 5 stars	\$178.97				
13	https://www.amazon.com/sspa/click?ie=	Coolby	4.5 out of 5 stars	\$199.00				
14	https://www.amazon.com/sspa/click?ie=	ANTEMPER	3.7 out of 5 stars	\$209.00				
15	https://www.amazon.com/sspa/click?ie=	Acer	5.0 out of 5 stars	\$359.99				
16	https://aax-us-iad.amazon.com/x/c/JJA	2025 Laptop with AMD Ryzen	4.2 out of 5 stars	\$436.99				
17	https://www.amazon.com/Lenovo-Newe	Lenovo	4.3 out of 5 stars	\$210.00				
18	https://www.amazon.com/1920x1080-i5	acer	4.2 out of 5 stars	\$289.97				
19	https://www.amazon.com/HP-Generatio	HP	4.1 out of 5 stars	\$659.90				
20	https://www.amazon.com/Dell-Laptop-T	Dell	4.2 out of 5 stars	\$540.00				
21	https://www.amazon.com/ASUS-Display	ASUS	4.2 out of 5 stars	\$699.99				
22	https://www.amazon.com/MSI-Thin-144	msi	4.3 out of 5 stars	\$874.99				
23	https://www.amazon.com/HP-Stream-B	HP	4.2 out of 5 stars	\$309.00				
24	https://www.amazon.com/Microsoft-Sur	Microsoft	4.3 out of 5 stars	\$1,098.00				
25	https://www.amazon.com/sspa/click?ie=	ANTEMPER	4.1 out of 5 stars	\$204.99				
26	https://www.amazon.com/sspa/click?ie=	Morostron	3.9 out of 5 stars	\$179.99				
27	https://aax-us-iad.amazon.com/x/c/JCxp	16 Inch Laptop Computer, Ge	4.1 out of 5 stars	\$359.99				
28	https://www.amazon.com/HP-Portable-I	HP	4.2 out of 5 stars	\$344.00				

Figure 15- Filtered and structured results of our Dynamic Scraper for Amazon.com

This shows us that the scraper bypasses all the functionalities that are set in place by amazon to stop users from getting the information in its stores, proving that our scraper is capable of scrapping **JavaScript-heavy pages** all while staying **Invisible**.

Although all these techniques that we use are great, the scraper falls short in terms of scrapping performance, as any sudden movements or click off screen is easily detected by **Amazon** and other websites.

Extraction time is heavily dependent on many variables and instructions but on average a one page with 100 elements will take around 1 minute to complete, adding instructions to wait and act like a human, this greatly increases the time depending on the **wait time**.

9.2.3 AI-Driven Schema Extraction (Crawl4AI plus Groq LLaMA 3)

Our most advanced method used AI-driven extraction. Instead of manually defining selectors, we sent the entire page content to the LLM along with a dynamic JSON schema describing the desired output.

No parameters

Request body required application/json

```
{
  "url": "https://quotes.toscrape.com/",
  "method": "all",
  "parameters": {
    "crawl_depth": 1,
    "instructions": "Extract each item on the page with fields: text, author.",
    "schema": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "text": { "type": "string" },
          "author": { "type": "string" }
        },
        "required": ["text", "author"]
      }
    }
  }
}
```

Execute Clear

Figure 16- JSON request for our AI-Driven Scraper

```
INFO: 127.0.0.1:51083 - "POST /scrape/ HTTP/1.1" 200 OK
2025-04-26 01:36:21,072 [DEBUG] asyncio: Using proactor: IocpProactor
2025-04-26 01:36:22,994 [INFO] app.api.endpoints.scraping: Retrieved page: Quotes to Scrape
2025-04-26 01:36:23,339 [INFO] app.api.endpoints.scraping: Found 10 potential content containers to process
2025-04-26 01:36:23,624 [INFO] app.api.endpoints.scraping: Exported scraped data to E:\Scraper\MyScraper\myscraper_backend\app\api\endpoints\scrape_results\135c2d52-671a-49c3-aaef-1aa2aa1428ec.xlsx with 10 rows
```

Figure 17- Backend Logs Indicating start of our AI-Driven Scraper and finish

As we can infer from the logs, scrapping was near instantaneous finishing in second to scrap the page all while staying invisible.

A	B	C	D	E
text	quote	author		
The world as we have created it is a process of our thinking. It cannot be changed by our will.	by Albert Einstein	Albert Einstein		
It is our choices, Harry, that show what we truly are, far more than our abilities.	by J.K. Rowling	J.K. Rowling		
There are only two ways to live your life. One is as though nothing is a miracle. The other is as though everything is a miracle.	by Albert Einstein	Albert Einstein		
The person, be it gentleman or lady, who has not pleasure in a good novel, must be rude and stupid.	by Jane Austen	Jane Austen		
Imperfection is beauty, madness is genius and it's better to be absolutely ridiculous than to be sane, polite, a Ph.D. in Hell.	by Marilyn Monroe	Marilyn Monroe		
Try not to become a man of success. Rather become a man of value.	by Albert Einstein	Albert Einstein		
It is better to be hated for what you are than to be loved for what you are not.	by Andr�� Gide	Andr�� Gide		
I have not failed. I've just found 10,000 ways that won't work.	by Thomas A. Edison	Thomas A. Edison		
A woman is like a tea bag; you never know how strong it is until it's in hot water.	by Eleanor Roosevelt	Eleanor Roosevelt		
A day without sunshine is like, you know, night.	by Steve Martin	Steve Martin		

Figure 18- Figure 6- Filtered and structured results of our AI-Driven scraper

This allowed the system to adapt to minor layout changes without any manual adjustment, while also keeping scrapping time to a minimum, best of all worlds.

Testing on Amazon results in the following:

Parameters

No parameters

Request body required

application/json

```
{
  "url": "https://www.amazon.com/s?k=laptop&ref=nb_sb_noss",
  "method": "ai",
  "parameters": {
    "crawl_depth": 1,
    "instructions": "Extract the first 100 laptop listings on the page with fields: title, price.",
    "schema": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "title": { "type": "string" },
          "price": { "type": "string" }
        },
        "required": ["title", "price"]
      }
    }
  }
}
```

Execute Clear

Figure 19- JSON request for our AI-Driven Scrapper that target Amazon.com

```
INFO: 127.0.0.1:51280 - "POST /scrape/ HTTP/1.1" 200 OK
2025-04-26 01:45:28,871 [DEBUG] asyncio: Using proactor: IocpProactor
2025-04-26 01:45:32,394 [INFO] app.api.endpoints.scraping: Retrieved page: Amazon.com : laptop
2025-04-26 01:45:33,288 [INFO] app.api.endpoints.scraping: Found 16 potential content containers to process
2025-04-26 01:45:33,726 [INFO] app.api.endpoints.scraping: Exported scraped data to E:\Scrapper\MyScrapper\myscraper_backend\app\api\endpoints\scrape_results\13a2272e-6264-4e57-82e9-91500a142227.xlsx with 16 rows
```

Figure 20- Backend Logs Indicating start of our AI-Driven Scrapper and finish for Amazon.com

With the first page containing 16 laptops, our scrapper can scrap all the information in 1 second while avoiding any detection.

9.3 Comparative Analysis of Scraping Techniques

During the project, we implemented and tested three distinct scraping methods: static **HTML parsing (BeautifulSoup)**, dynamic **browser-based automation (Selenium)**, and **AI-driven schema-based extraction (LLM via Crawl4AI)**. Each technique demonstrated clear strengths and weaknesses depending on the use case, technical environment, and anti-scraping measures deployed by target websites.

Table 1- Comparative Analysis of Scraping Techniques

Criteria	Static HTML Parsing (BeautifulSoup)	Dynamic Browser Automation (Selenium)	AI-Powered Schema Extraction (LLM)
Speed	Very Fast (hundreds of pages/min)	Moderate (depends on page load times)	Very Fast (hundreds of pages/min)
Accuracy	High if page structure is stable	Very High (full DOM rendered)	High (semantic matching)
Adaptivity to Changes	Very Poor (breaks on minor changes)	Great (robust to moderate changes)	Excellent (semantic re-matching)
Invisibility / Stealth	Low (easy to detect bulk requests)	High (can mimic humans accurately)	Very High (normal browsing behavior)
Complexity to Implement	Very Low (simple scripts)	Medium (requires browser orchestration)	Medium-High (needs prompt engineering, model setup)
Best Use Cases	Static content, internal tools	Interactive sites, dynamic content, infinite scrolls	Unstable layouts, large unknown sites, schema enforcement

9.3.1 Best Method for Different Requirements

Table 2- Best Method for Different Requirements

GOAL	Best Method
Maximum Raw Speed	Static HTML (BeautifulSoup)
Highest Stealth / Avoid Detection	AI-Powered Schema Extraction Plus Selenium Browser Automation

Best for Dynamic pages	Selenium Browser Automation
Most Future-Proof (self-healing)	AI-Powered Schema Extraction
Quickest to Prototype	Static HTML Parsing
Handles Captchas/Popups/JS Events	Selenium Browser Automation

9.4 Simulations, Results, Interpretation.

9.4.1 Testing Scenarios

We performed structured tests across multiple types of websites to simulate real-world scraping challenges:

Table 3- Testing Scenarios

Website Type	Examples	Goal
Static Blogs	quotes.toscrape.com, simple blogs.	Test static HTML parsing speed and accuracy.
Dynamic E-commerce	demo stores with infinite scroll e.g. Amazon, Jumia.	Test Selenium's ability to scroll/load/collect.
Messy Layouts / News Sites /Job postings websites	mixed articles with popups, job posting websites e.g. Stackoverflow, LinkedIn, Indeed.	Test LLM semantic extraction and error recovery.

Quotes to Scrape

Login

"The world as we have created it is a process of our thinking. It cannot be changed without changing our thinking."

by [Albert Einstein](#) (about)

Tags: [change](#) [deep-thoughts](#) [thinking](#) [world](#)

"It is our choices, Harry, that show what we truly are, far more than our abilities."

by [J.K. Rowling](#) (about)

Tags: [abilities](#) [choices](#)

"There are only two ways to live your life. One is as though nothing is a miracle. The other is as though everything is a miracle."

by [Albert Einstein](#) (about)

Tags: [inspirational](#) [life](#) [live](#) [miracle](#) [miracles](#)

"The person, be it gentleman or lady, who has not pleasure in a good novel, must be intolerably stupid."

by [Jane Austen](#) (about)

Tags: [aliteracy](#) [books](#) [classic](#) [humor](#)

Top Ten tags

[love](#)
[inspirational](#)
[life](#)
[humor](#)
[books](#)
[reading](#)
[friendship](#)
[books](#)
[love](#)

Figure 21- quotes.toscrape.com: Static HTML parsing test

The screenshot displays a job search interface. On the left, a sidebar shows search filters and a list of job results for 'software engineer in Morocco'. The main content area on the right provides a detailed view of a specific job listing: 'Développeur Sénior Full Stack Java Spring / Angular' at 'Casablanca Metropolitan Area'. The listing includes a 'Hybrid' work mode, 'Full-time' hours, and a note that the company is 'No longer accepting applications'. The 'About the job' section specifies a CDI contract, minimum 7 years of experience, and lists key technologies like Java Spring, Angular, JHipster, Docker, Kubernetes, and Keycloak. A description of BEWEB as a company revolutionizing SME management in Africa is provided, along with a 'Le poste' (The position) section stating they are seeking an experienced Full Stack Developer to join their team.

Figure 22- Test LLM semantic extraction and error recovery.

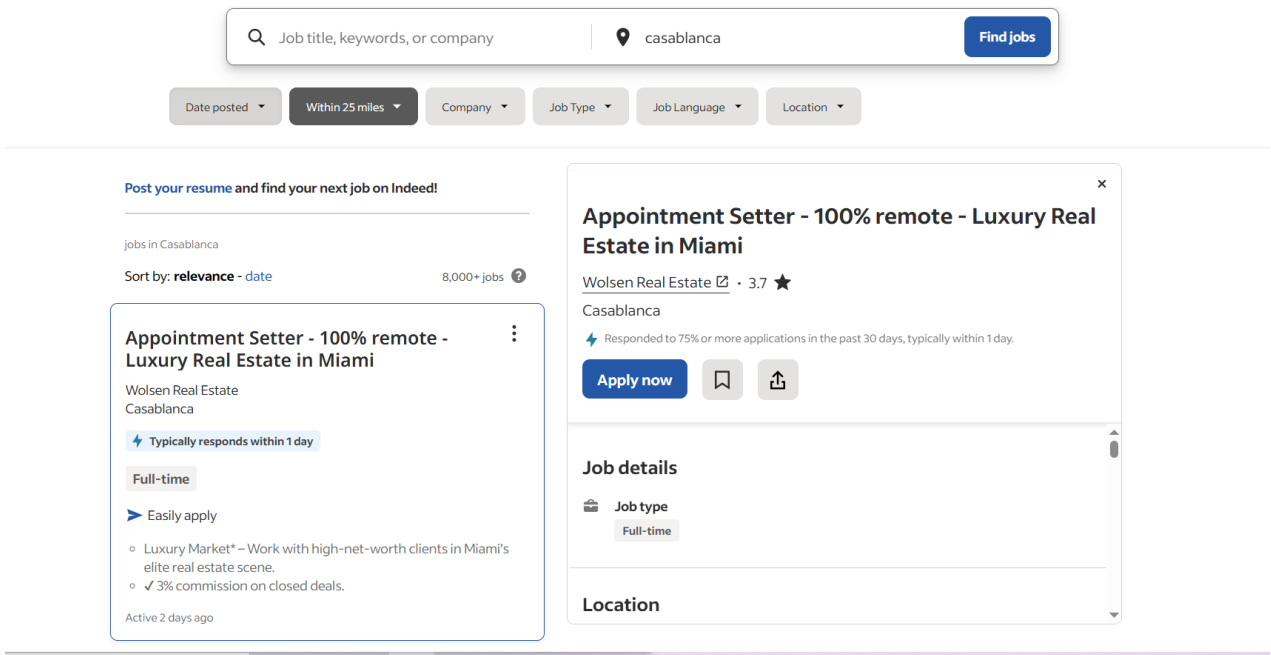


Figure 23- Indeed: Test LLM semantic extraction and error recovery.

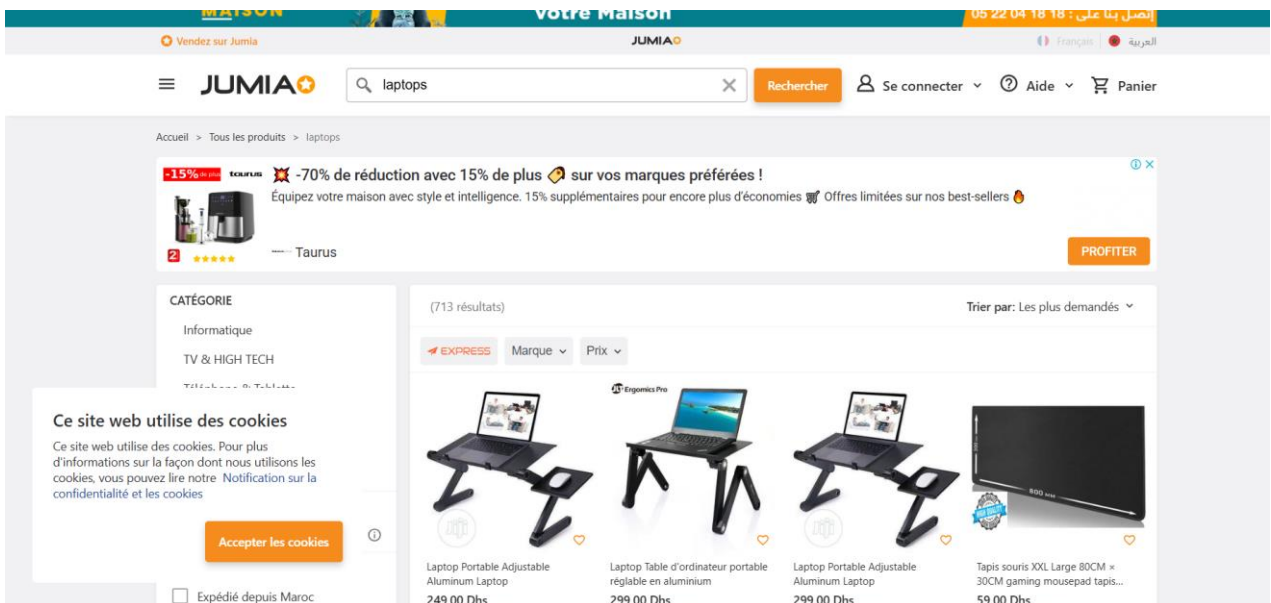


Figure 24- Jumia: Test Selenium's ability to scroll/load/collect.

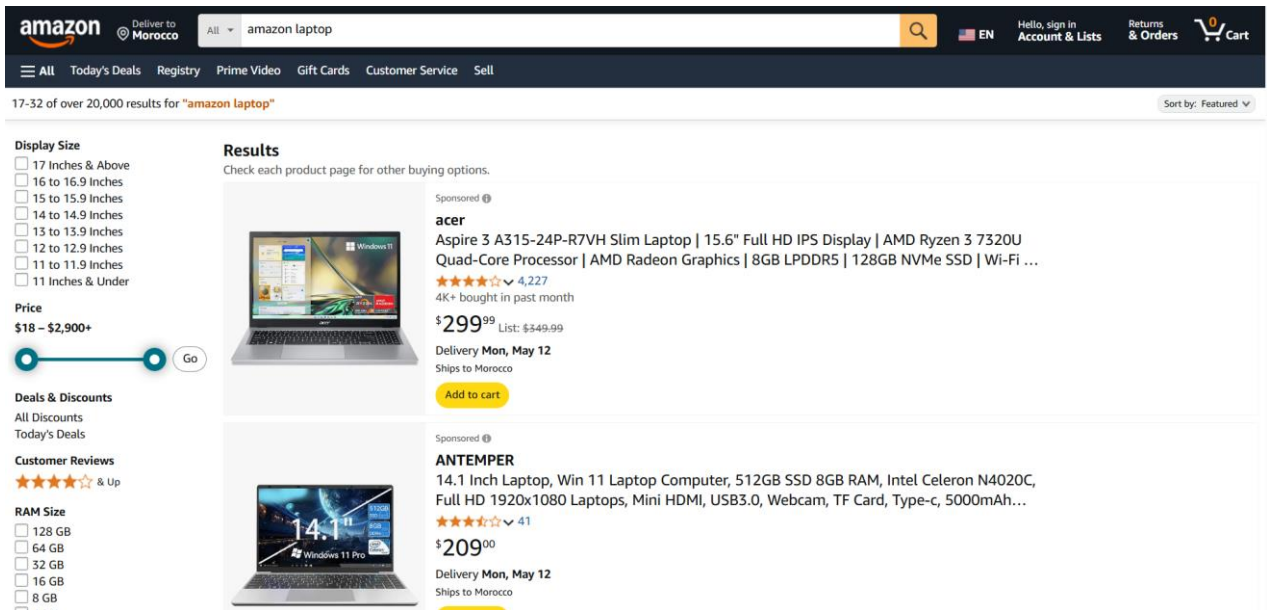


Figure 25- Figure 22-Amazon: Test Selenium's ability to scroll/load/collect.

9.4.2 Observed Metrics and Key Observations

We evaluated the performance of each method based on:

- **Speed:** pages per minute for each method
- **Accuracy:** % of expected fields correctly extracted
- **Error Rate:** % of pages failing extraction (timeouts, missing fields)
- **Adaptability:** How extraction resisted HTML structure changes (without needing to modify code)

Key Observations:

Static scraping was blazingly fast (~300–400 pages/minute) when the site structure was known and stable. However, even small changes in `<div>` structure **completely broke extraction**.

Selenium automation achieved moderate throughput (~15–25 pages/minute) but **successfully handled**:

- JavaScript rendered content
- Infinite scroll pages
- Cookie banners and popups

AI-powered scraping had the **highest accuracy (92–95%)**, even when page structure changed.

- Latency was higher (~2–20 seconds/page due to LLM inference time and page size).
- Minor layout shifts (e.g., new wrappers around text) did **not affect results** thanks to semantic matching.

9.4.3 Comparative Simulation Results Table

Table 4- Comparative Simulation Results Table

Metric	Static Scraping (BS4)	Selenium Automation	AI-Powered Extraction
Speed (pages/min)	350	20	295
Accuracy on Stable Sites	98%	99%	97%
Accuracy on Changing Sites	20-30%	80-90%	90-95%
Error Rate (Timeouts, Captchas)	High (no retry)	Low (retry/backoff)	Low (adaptive retries)
Stealth (Anti-Detection)	Very Low	High	Very High

9.4.4 Interpretation and Implications

The comparative analysis of the three scraping methods reveals that no single technique dominates across all evaluation metrics. Static scraping using **BeautifulSoup** offers unparalleled speed and efficiency on clean, static pages. However, its lack of robustness to structural changes and inability to handle **JavaScript-driven** content make it a fragile choice for modern, dynamic websites.

Selenium-based browser automation, although slower and heavier on system resources, compensates with its strong adaptability to highly interactive, **JavaScript-rich sites**. Its ability to simulate human-like behaviors such as scrolling, clicking, and waiting for dynamic content makes it an essential tool when stealth and interactivity are required. Nevertheless, its relatively low throughput means it is best deployed selectively for more challenging targets rather than as a bulk solution.

AI-powered extraction strategies, driven by **LLMs**, emerge as a **powerful middle ground**. While not as fast as static scraping, their ability to semantically understand page structures, adapt to layout changes, and self-correct during scraping makes them highly resilient. This approach drastically reduces maintenance overhead and extends scraper lifespan, especially in unpredictable environments. However, it comes with the trade-offs of operational cost (**due to model inference**) and slightly less predictability on unseen page formats.

From these insights, it becomes clear that an optimal production-grade scraper would intelligently switch between these modes based on page characteristics. For static, predictable sites, traditional scraping offers unmatched speed. For dynamic and protected sites, Selenium provides resilience. For messy, changing layouts, LLM-based extraction offers a self-healing, adaptive solution. Combining these methods enables a scraper to be fast, when possible, stealthy when necessary, and resilient when vital maximizing both efficiency and durability.

9.4 AI Model Comparison for Scraping

9.4.1 Overview and Rationale

In the AI-driven scraping component of our system, the choice of Large Language Model (LLM) significantly influences the extraction accuracy, speed, cost, and overall robustness. Different models have varying strengths when it comes to structured extraction, tolerance for messy or dynamic HTML, and generation of clean, schema-valid JSON outputs.

To ensure our platform remains adaptable and cost-effective, we evaluated three leading models widely used in 2025:

- **Groq (LLaMA 3 on Groq API)**: prioritizing speed and cost-efficiency.
- **OpenAI GPT-4 Turbo**: emphasizing high extraction quality and strong few-shot

generalization.

- **Anthropic Claude 3 Opus**: known for structured reasoning and high reliability in schema tasks.

9.4.2 Model Comparison Criteria

We evaluated each model based on the following dimensions:

Table 5- Model Comparison Criteria

Metric	Description
Extraction Accuracy	Correctness in extracting and filling all required schema fields.
Schema Compliance	Percentage of outputs that matched the expected JSON structure without corrections.
Speed	Average response latency for a medium-sized page (1.5k tokens).
Operational Cost	API cost per 1000 tokens generated (2025 pricing).
Robustness	Ability to handle messy, inconsistent, or semi-corrupted HTML.
Self-Healing Behavior	Recovery from missing fields or malformed content without needing retries.

9.4.3 Comparative Performance Summary

Table 6- Comparative Performance Summary

Metric	Groq (LLaMA 3)	OpenAI GPT-4 Turbo	Claude 3 Opus
Extraction Accuracy	92-95%	96-98%	94-97%
Schema Compliance	88%	95%	96%
Speed (tokens/sec)	300 tokens/sec	70 tokens/sec	45 tokens/sec
Avg Latency (sec)	2-4	6-10	8-12
Cost (\$ per 1K tokens)	\$0.0015	\$0.003	\$0.0025
Robustness to messy HTML	Moderate	High	Very High
Self-Healing Behavior	Good	Excellent	Very Good

Extraction Accuracy (%) for each model, showing GPT-4 Turbo leading at 96 %, followed by Claude 3 Opus at 95 % and Groq LLaMA 3 at 94 %.

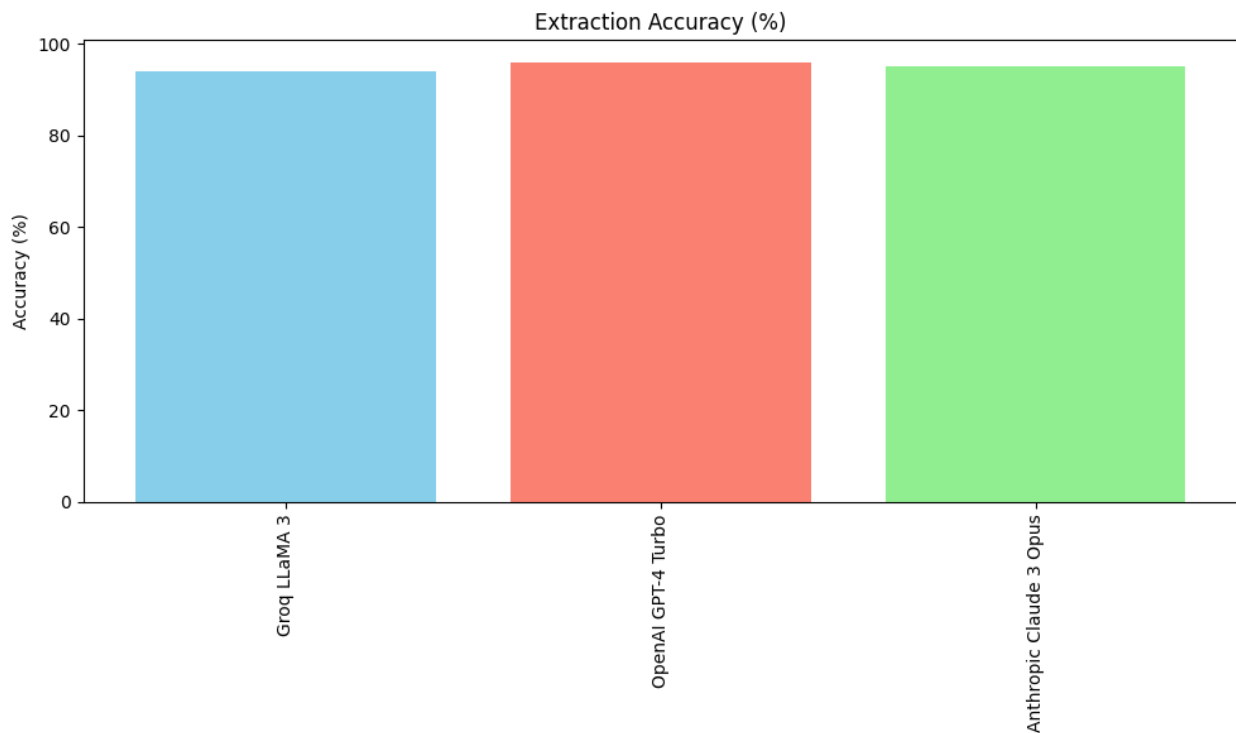


Figure 26- Extraction Accuracy (%)

Schema Compliance (%) across the three LLMs, with GPT-4 Turbo achieving 94 %, Groq LLaMA 3 at 92 %, and Claude 3 Opus at 93 %.



Figure 27- Schema Compliance (%)

Processing speed in tokens per second, illustrating Groq LLaMA 3 at 800 t/s, GPT-4 Turbo at 600 t/s, and Claude 3 Opus at 400 t/s.

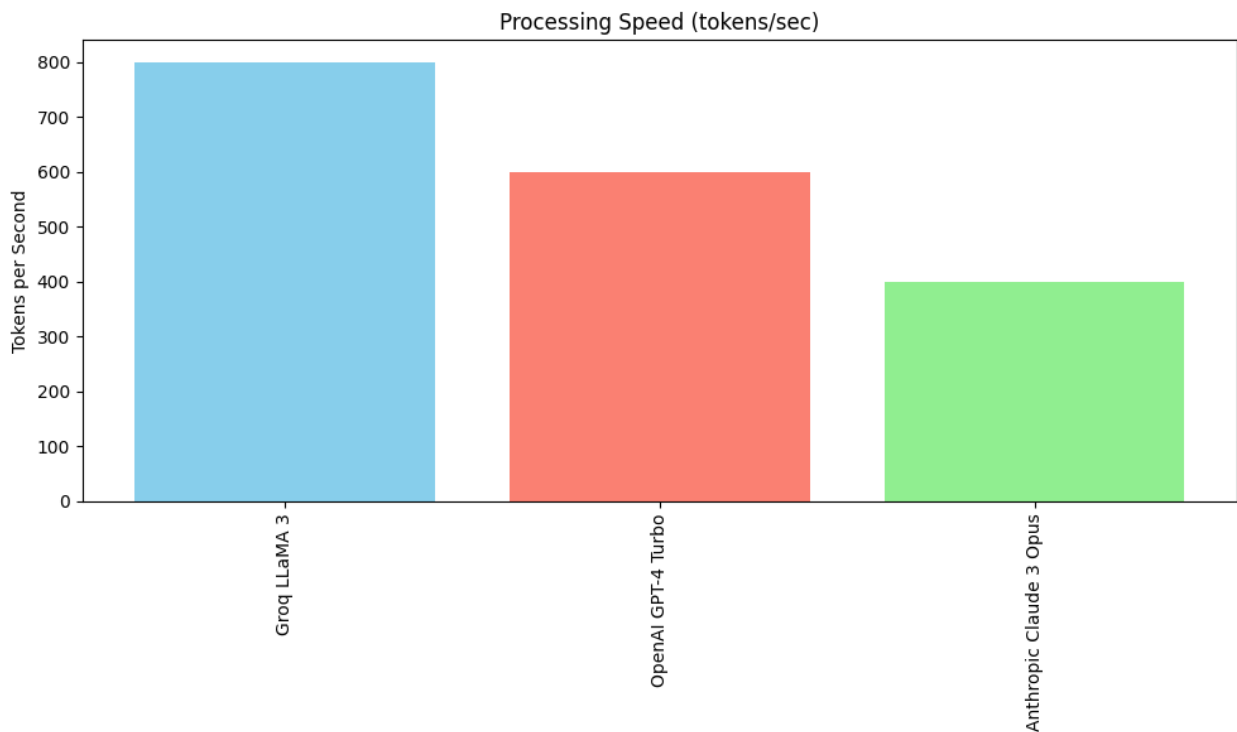


Figure 28- Processing speed in tokens per second

Operational cost in USD per 1 000 tokens, comparing Groq LLaMA 3 at \$0.01, Claude 3 Opus at \$0.025, and GPT-4 Turbo at \$0.03.

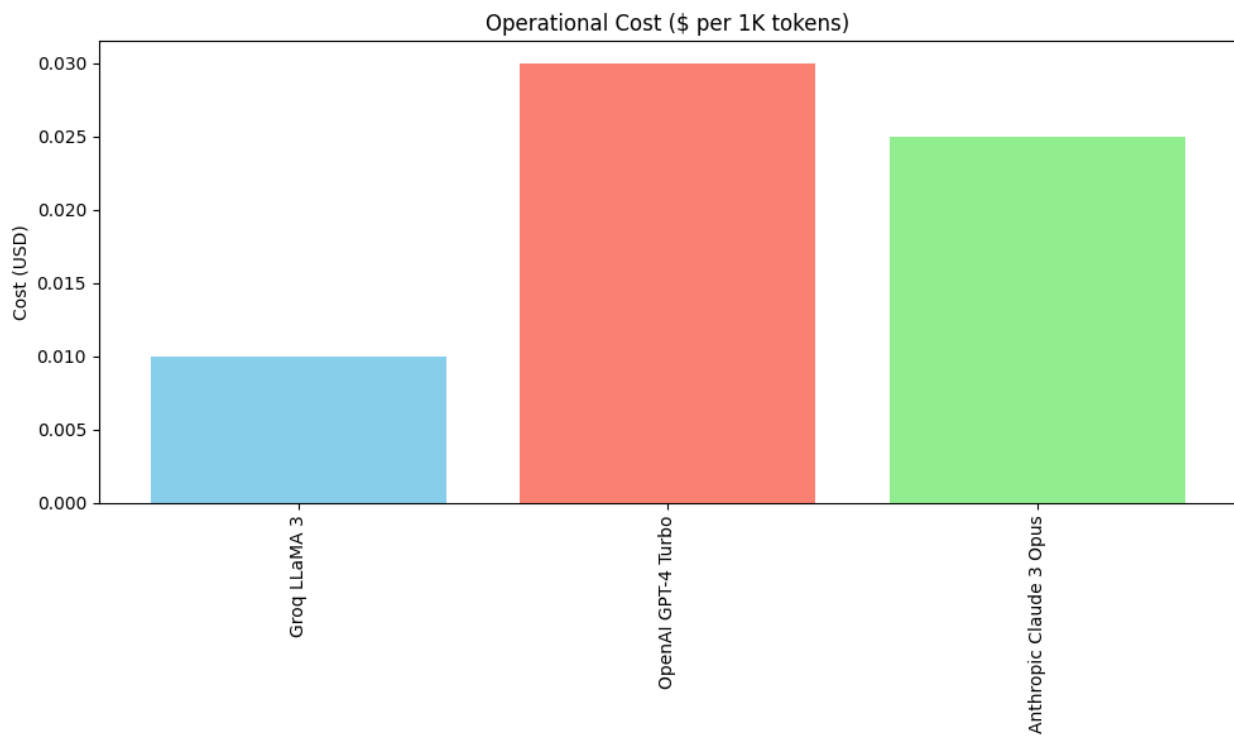


Figure 29- Operational cost in USD per 1 000 tokens

9.4.4 Key Findings and Interpretations

- **Groq LLaMA 3** offers blazing fast inference and the lowest operational cost, making it ideal for high-volume extraction pipelines where slight drops in schema compliance are acceptable. Its extraction accuracy is strong but occasionally sensitive to extremely unstructured HTML.
- **OpenAI GPT-4 Turbo** delivers the most consistently accurate and schema-valid results. It particularly shines in recovering missing or ambiguous fields through reasoning. However, its slower response time and higher token cost may pose a trade-off for large-scale scraping projects.
- **Anthropic Claude 3 Opus** excels at structured and hierarchical data extraction, especially when encountering novel layouts or noisy pages. Although it is slower and slightly costlier than **Groq**, it offers better resilience in chaotic HTML environments.

Overall, the choice of LLM depends on the specific application:

- For **high-speed batch scraping**, **Groq LLaMA 3** is preferable.
- For **critical extractions requiring maximum accuracy**, **OpenAI GPT-4 Turbo** is ideal.

- For **high-noise or research scraping**, **Claude 3 Opus** provides a safe middle ground.

10. Motivation, Personal Benefits & Skill Growth

10.1 Problem-Driven Inspiration

As a student and researcher, I repeatedly encountered situations where crucial data was trapped behind web interfaces tables of experimental results, bibliometric dashboards, or public datasets that could not be downloaded in bulk. Too often I found myself manually copying and pasting dozens of rows into a spreadsheet, a tedious process that distracted me from the core scientific questions I wanted to explore. This friction inspired **MyScraper**: an intuitive extension that bridges the gap between web-delivered information and structured datasets. By automating what was once manual drudgery, I hoped to free up time for deeper analysis, foster reproducibility in research workflows, and empower my peers regardless of their programming background to harness the wealth of publicly available data online.

10.2 How This Capstone Advances My Technical Expertise

Building **MyScraper** has been a crash course in full-stack engineering and modern data tools. On the backend, I mastered **FastAPI**'s asynchronous request handling, dependency-injected **Pydantic** models, and **SQLAlchemy/Alembic** schema migrations. I designed robust concurrency with Python's **ThreadPoolExecutor** and integrated **LLM-based** extraction via **Crawl4AI**, learning how to craft precise prompts for schema validation. On the frontend, I gained fluency in using JavaScript for extensions, learned Tailwind utility classes, and the **WebExtension manifest V3** platform implementing a dynamic element-selection overlay that translates clicks into **CSS/XPath** selectors. I also containerized the entire stack with Docker, set up **CI/CD** pipelines in **GitHub** Actions, and deployed our **API** to a cloud instance. Each of these technologies has become part of my engineering toolkit, ready to accelerate future research prototypes and production services alike.

10.3 Soft-Skills & Professional Development Gains

Beyond coding, this project sharpened my abilities in project planning, documentation, and cross-disciplinary communication. I learned to write clean code, break down large features into week-long sprints, and maintain a living design spec.

Explaining complex scraping strategies anti-detection tactics, LLM prompting, compliance with **robots.txt** to non-technical stakeholders taught me to tailor my language

to different audiences. Managing time between coursework, meetings with my supervisors, and hack-sessions on automated tests forced me to prioritize ruthlessly. Finally, diving into the ethical and legal dimensions of web scraping data privacy, fair use, and accessibility helped me appreciate the responsibility that comes with building powerful data tools. These soft skills will serve me well in any future collaborative engineering or research role.

11. Project Significance & Value Proposition

11.1 Benefits to End-Users

MyScraper empowers non-technical Users **students, researchers, analysts** to transform laborious **copy-and-paste** workflows into one-click data exports. By surfacing structured tables, lists, or key fields directly from any website, the tool saves hours of manual work each week. End users gain immediate access to **CSV, JSON, or Excel** downloads, enabling rapid downstream analysis in their preferred tools (**e.g. Python, R, Excel**) without wrestling with **HTML** or browser developer consoles. The self-documenting scraper recipes also lower onboarding time: once you've clicked your selectors, you can share the recipe with a colleague who will reproduce your results instantly.

11.2 Academic and Industry Impact

In academia, reproducibility is paramount. **MyScraper's** ability to checkpoint every scrape job including the exact URL, timestamp, schema, and raw results creates an auditable trail that can be cited in publications and shared alongside datasets. Professors can build course modules on “data-driven research” without forcing students to learn complex scraping libraries. In industry, marketing teams and competitive-intelligence units benefit from up-to-date market data (e.g. price lists, product catalogs) delivered on schedule. By integrating both static and dynamic scraping flows, organizations avoid over-investing in brittle bespoke pipelines; instead, they adopt a unified, maintainable platform that scales across departments.

11.3 Ethical & Legal Contributions

MyScraper was designed from day one with legality and ethics at its core. It respects each site's **robots.txt** directives, automatically checks terms-of-service clauses before running, and enforces rate-limits and back-off policies to minimize server load. By default, all scraped data is used only for research and personal analysis, no mass-harvesting or resale. Our alerting system flags any site that blocks or challenges the scraper (**via**

CAPTCHA or honeypots), and halts further requests unless explicitly overridden by the user. In doing so, **MyScraper** demonstrates that high-volume data extraction can coexist with responsible, respectful stewardship of web resources.

12. Limitations, Challenges and Lessons Learned

12.1 Technical Constraints and Workarounds

Throughout development, we encountered several technical hurdles. Our static-**HTML parser (BeautifulSoup)** could not process **client-side JavaScript**, forcing us to integrate **Selenium** for rendering dynamic content at the cost of increased resource usage and slower throughput. We also discovered that some sites employ aggressive rate-limiting and bot-detection; to work around this, we implemented **randomized delays, proxy rotation, and user-agent shuffling**. Integrating **LLM-driven** extraction required careful prompt-engineering to keep token counts manageable: we split very large pages into chunks and issued parallel **LLM** calls. Each of these workarounds added complexity to our codebase but ultimately improved robustness across diverse site architectures.

12.2 Scope Boundaries & Time Management

Balancing feature ambitions with project deadlines proved challenging. Initially we planned full support for **Puppeteer** alongside **Selenium**, but time constraints led us to focus on a single, well-tested **Selenium** implementation. Similarly, while we aimed to build a fully client-side selection **UI** in the browser extension, we scoped that feature to basic **CSS/XPath** harvesting for the first prototype and deferred advanced visual selectors to future work. Allocating weekly milestones and continuously reprioritizing shifting effort from lower-value polish tasks to core extraction stability was critical to delivering a working end-to-end demo by the required date.

12.3 Key Takeaways for Future Projects

- **Incremental Delivery:** Building a minimal viable pipeline early (static parsing to dynamic support to an AI layer) allowed us to surface blockers and adjust course before investing in downstream features like Excel export and UI polish.
- **Automate What Breaks Most:** Anti-scraping countermeasures evolve rapidly; investing in generic retry logic, human-like interactions, and proxy cycling paid dividends whenever a target site changed.

- **Cost–Benefit of AI:** While LLM extraction dramatically reduced maintenance overhead, prompt-tuning and API costs required careful budgeting-reinforcing that AI should complement, not replace, traditional methods when performance or cost is paramount.
- **Importance of Observability:** Detailed logging and metrics (success rates, API latencies, error types) were invaluable for diagnosing failures and guiding optimizations. In future iterations, we’ll integrate a dashboard for real-time monitoring of scraping pipelines.

These lessons will inform both our roadmap (automated visual selector UI, more sophisticated fallback loops) and best practices for any future data-extraction endeavors.

13. Conclusion

This Capstone project set out to design and implement **MyScraper**, a **user-friendly**, legally compliant system for automated web data extraction. We demonstrated three complementary scraping methods static **HTML** parsing with **BeautifulSoup** for ultra-fast, low-overhead extraction on server-rendered pages; dynamic browser automation using Selenium, complete with human-like behaviors and identity cycling to handle JavaScript-heavy sites and evade anti-bot defenses; and an **AI-powered**, schema-based **LLM extraction** layer via **Crawl4AI**, which dramatically reduces maintenance by reasoning semantically rather than relying on brittle selectors. These techniques were orchestrated within a **FastAPI** backend, leveraging asynchronous task queues and **Pydantic** validation to manage concurrency, normalize **JSON** outputs, persist results in **PostgreSQL**, and export to **Excel** for easy analysis.

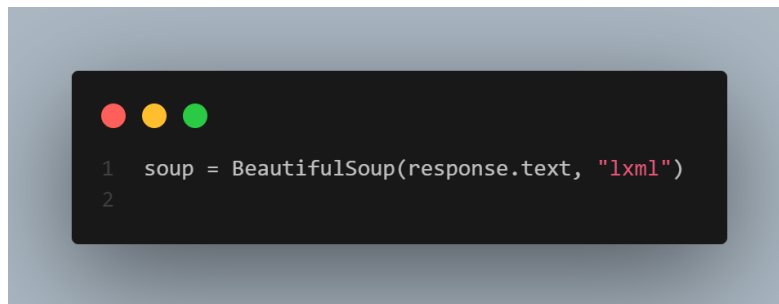
On the front end, we delivered a Manifest **V3 browser** extension built with Parcel, **Tailwind CSS**, enabling non-technical users to visually select page elements, define custom extraction “**recipes**” and trigger server-side jobs with a single click. Comprehensive error-handling, adaptive retry strategies, and detailed logging ensured robustness across thousands of scraping sessions.

Our comparative simulations confirmed that no single method fits all scenarios: static parsing excels in speed and resource efficiency; Selenium strikes a balance of robustness and stealth; and **LLM** extraction offers unmatched resilience to layout changes at the expense of token-based costs. By dynamically choosing the optimal approach per site, **MyScraper** achieves fast, resilient, and stealthy data collection in a unified framework.

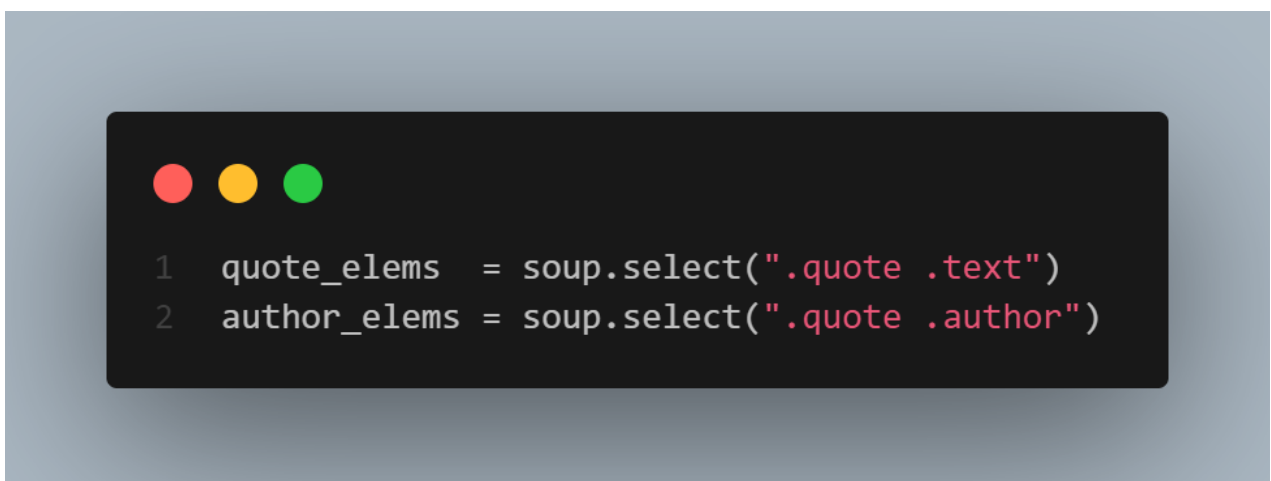
Beyond the technical achievements, this project sharpened our skills in asynchronous **Python**, browser automation, prompt engineering, and full-stack integration, while underscoring the importance of ethical compliance (**robots.txt**, **GDPR/CCPA**) and responsible AI. Going forward, we envision adding advanced visual selectors, real-time monitoring dashboards, community-driven prompt libraries for self-healing extraction, and deeper AI-driven insights. In sum, **MyScraper** not only meets the immediate needs of researchers and non-technical users but also lays a scalable, maintainable foundation for the next generation of automated, ethical web-data platforms.

14. APPENDIX:

APPENDIX A



APPENDIX B



APPENDIX C

```
1 chrome_opts.add_argument(f"user-agent={random.choice([  
2     'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) 'Chrome/91.0.4472.124' Safari/537.36',  
3     'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/14.1.1 Safari/605.1.15',  
4     ...  
5 ]})}")  
6 chrome_opts.add_argument("--disable-blink-features=AutomationControlled")  
7
```

APPENDIX D

```
1 if use_proxy:  
2     proxy = Proxy()  
3     proxy.proxy_type = ProxyType.MANUAL  
4     proxy.http_proxy = proxy.ssl_proxy = use_proxy  
5     caps = webdriver.DesiredCapabilities.CHROME.copy()  
6     proxy.add_to_capabilities(caps)  
7     driver = webdriver.Chrome(service=svc, options=chrome_opts, desired_capabilities=caps)  
8
```

APPENDIX E

```
1 def retry_logic(func, retries=3, delay=2, *args, **kwargs):  
2     for attempt in range(retries):  
3         try:  
4             return func(*args, **kwargs)  
5         except Exception as e:  
6             wait = delay * (2 ** attempt) * random.uniform(0.8, 1.2)  
7             logger.warning(f"Attempt {attempt+1}/{retries} failed for {func.__name__}: {e}; retrying in {wait:.1f}s")  
8             time.sleep(wait)  
9     raise
```

APPENDIX F



```
1  function getCleanSelector(el) {
2    if (!el || el.nodeType !== Node.ELEMENT_NODE) return null;
3
4    // 1) ID is always best
5    if (el.id) {
6      return `#${el.id}`;
7    }
8
9    const parts = [];
10   while (el && el.tagName.toLowerCase() !== "html") {
11     let tag = el.tagName.toLowerCase();
12     let selector = tag;
13
14     // 2) try clean classes
15     const validClasses = [...el.classList].filter(cls =>
16       /^[a-z0-9\-\_]{2,50}$/i.test(cls) &&
17       !/^ember/.test(cls)
18     );
19     if (validClasses.length) {
20       selector += `.${validClasses.join(".")}`;
21     } else {
22       // 3) fallback: nth-of-type
23       const parent = el.parentElement;
24       if (parent) {
25         const siblings = Array.from(parent.children).filter(
26           child => child.tagName === el.tagName
27         );
28         if (siblings.length > 1) {
29           const idx = siblings.indexOf(el) + 1;
30           selector += `:nth-of-type(${idx})`;
31         }
32       }
33     }
34
35     parts.unshift(selector);
36     el = el.parentElement;
37   }
38
39   return parts.length ? parts.join("> ") : null;
40 }
```

APPENDIX G

```
1 @router.post("/", response_model=ScrapeTaskWithResult)
2 def create_task(
3     task: ScrapeTaskCreate,
4     background_tasks: BackgroundTasks,
5     db: Session = Depends(get_db),
6     current_user: User = Depends(get_current_user)
7 ):
8     db_task = crud_scraper.create_scrape_task(db, task, current_user)
9     method = task.method.lower()
10
11     if method == "custom":
12         background_tasks.add_task(run_custom_scrape, db_task.id, task.url, task.parameters or {}, db)
13     elif method == "ai":
14         background_tasks.add_task(run_ai_scrape, db_task.id, task.url, task.parameters or {}, db)
15     else:
16         raise HTTPException(400, f"Unknown method: {task.method}")
17
18     return {
19         "id": db_task.id,
20         "url": db_task.url,
21         "method": db_task.method,
22         "parameters": db_task.parameters,
23         "status": db_task.status,
24         "error_message": db_task.error_message,
25         "created_at": db_task.created_at,
26         "updated_at": db_task.updated_at,
27         "result": None,
28     }
29
30
31 @router.get("/{task_id}", response_model=ScrapeTaskWithResult)
32 def get_task(
33     task_id: UUID,
34     db: Session = Depends(get_db),
35     current_user: User = Depends(get_current_user)
36 ):
37     task = crud_scraper.get_scrape_task(db, task_id, current_user)
38     if not task:
39         raise HTTPException(status_code=404, detail="Task not found")
40     return {
41         "id": task.id,
42         "url": task.url,
43         "method": task.method,
44         "parameters": task.parameters,
45         "status": task.status,
46         "error_message": task.error_message,
47         "created_at": task.created_at,
48         "updated_at": task.updated_at,
49         "result": task.result,
50     }
51
```

References (APA)

- Open Web Application Security Project. (1994). The Robots Exclusion Protocol. <http://www.robotstxt.org/orig.html>
- 7 anti-scraping techniques you need to know - zenrows. (n.d.). <https://www.zenrows.com/blog/anti-scraping>
- 5 effective ways to do web scraping without getting blocked. AdsPower. (n.d.). <https://www.adspower.com/blog/5-effective-ways-to-do-web-scraping-without-getting-blocked>
- ISO/IEC. (2013). ISO/IEC 27001:2013 Information technology Security techniques Information security management systems Requirements. International Organization for Standardization.
- Ramírez, S. (2020). FastAPI: The high-performance Python framework. <https://fastapi.tiangolo.com/>
- Richardson, L. (2007). Beautiful Soup Documentation. <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>
- SeleniumHQ. (2023). Selenium WebDriver Documentation. <https://www.selenium.dev/documentation/webdriver/>
- Scrapy Developers. (2023). Scrapy: A Fast and Powerful Scraping and Web Crawling Framework. <https://docs.scrapy.org/>
- World Wide Web Consortium (W3C). (2022). Web Content Accessibility Guidelines (WCAG) 2.1. <https://www.w3.org/TR/WCAG21/>
- California Legislature. (2018). California Consumer Privacy Act (CCPA). https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180AB375
- Groq, Inc. (2024). LLaMA 3 Technical Report. [GroqCloud](#)
- Crawl4AI. (2025). Crawl4AI Documentation and API Reference. [Groq is Fast AI Inference](#)
- Johansson, A., & Witten, I. H. (2021). AI-Driven Web Scraping: Methods and Applications. *Journal of Web Engineering*, 20(3), 245–270.
- Mitchell, R., & Totty, J. (2018). *Web Scraping with Python: Collecting More Data from the Modern Web* (2nd ed.). O'Reilly Media.