Agentic Self-Correcting Social Media Data Extraction System

by

Alvin Tsang

June 1, 2026

A project write-up submitted to the

faculty of the Graduate School of

the University at Buffalo, The State University of New York

in partial fulfillment of the requirements for the

degree of

Master of Science
Department of Computer Science and Engineering

This writeup is dedicated to my time at the University at Buffalo, from 2018-2026, and all the people I met along the way. Thank you for the great experience, UB. Horns up!

Acknowledgments

I would like to both dedicate and acknowledge Dr. Cassandra Jacobs, and Dr. Kenny A. Joseph for guiding me through my Masters journey, and allowing me to work with them. Thank you for taking care of me and helping me grow over the past two years. Take care.

Abstract

Computational Social Science (CSS) is a specialized field in the intersection of computing and social theory to analyze complex human behavior and social phenomena. To accomplish this, CSS uses many computing techniques and paradigms to retrieve data for predictive and analytical tasks. These computing techniques range from web scraping, NLP text analysis, network analysis, machine learning, causal inference and statistics, multimodal analysis, and much more. With such powerful compute power in the modern age, CSS has become even more prominent to find underlying meaning in what humans say and post, and why they do it.

This project focuses on the web scraping portion of Computational Social Science. Specifically, web scraping Instagram post comments. Instagram is notoriously difficult to scrape due to Meta placing countermeasures for scrapers, such as rate limiting, frequent HTML/CSS structure changes, and bot detection. Web scrapers break all the time due to countermeasures placed by the website origin. It is time consuming when you run a scraper, leave it alone for a while, and come back the next day to see that no data was retrieved because something failed. This project aims to combat the problems of scraping by using a self-correcting agentic AI system. With Python, Selenium, and Claude, this system eliminates much of the tedious process that comes with manually fixing web scrapers.

Table of Contents

Introduction

Computational Social Science (CSS) is an interdisciplinary field that includes mathematics, statistics, data science, and of course social science. A computational social science is emerging that leverages the capacity to collect and analyze data with an unprecedented breadth and depth and scale (Lazer et al., 2009). CSS introduces various computational methods such as text analysis, image analysis, web scraping, machine learning, causal inference and statistics, and much more. Using computational methods, we can study society faster, and at scale by analyzing data. We can answer many questions, such as: Why do people post on a particular subreddit? What are the hidden intentions of posting lifestyle videos on YouTube? How is the comments environment different on TikTok vs. Instagram? When, how, and why do influencers/content creators change their content? Do political podcasters change their framing based on the guests on an episode? There are so many questions and ideas to be answered in the vast field of Computational Social Science. Studying what humans say, and do, both in-person and online is important to see how humans behave and evolve over time.

Web scraping is a significant part of Computational Social Science, allowing researchers to retrieve large amounts of data from online platforms, such as Reddit, YouTube, Facebook, etc. Web scraping is the process of extracting information and data from websites through either two approaches: (1) manually writing programs to extract data from the HTML/CSS and DOM of a webpage, and (2) using the API of the webpage of interest. Although web scraping is so detrimental in CSS, it is often time-

consuming and difficult. Many challenges may arise, such as legality, web site structure, dynamic contents, blocking mechanisms, which can prevent scrapers from receiving data (Waheed et al., 2014). This project aims to eliminate the challenges that come with web scraping.

This project focuses on web scraping Instagram, a popular social media platform developed by Meta. To accomplish this, the scraper was developed using Python, and Selenium (Programmatic Browser Control). With the previously mentioned challenges that come with scraping, this project aims to solve those challenges with Agentic AI (Artificial Intelligence). Artificial Intelligence, specifically Generative AI, is growing at an extremely fast rate, with applications such as Claude Code, Cursor, ChatGPT, Coding Agents, Microsoft Office Agents, and so much more. AI has evolved from being a mere computational tool to a transformative force that is reshaping industries, economies, and societies. Agentic AI—a category of AI systems capable of independently making decisions, interacting with their environment, and optimizing processes without direct human intervention (Hosseini & Seilani, 2025). AI has increased our efficiency through automating tasks such as writing code, summarizing text, providing information, etc. Agentic AI is an even more powerful way to use AI by providing an LLM (Large Language Model) additional context and control of an environment for it to make decisions to modify and produce outputs. The key importance of Agentic AI is it being able to make it’s own decisions, without a human in the loop, automating processes, and often eliminating time-consuming tasks, such as fixing a web scraper. Using Python, and Claude (LLM of choice), this project introduces a self-correcting agentic AI

system that repairs the Instagram scraper upon failure, and continuously runs the scraper until all data is retrieved.

Problem Solution

Python 3.10

Python is a well-known high-level, interpreted programming language that is useful for fast and easy development of ideas. The scraper and agentic system were built using Python, involving various Python scripts to allow for parameterized command-line execution, extract data, structure extracted data, and prompt Claude.

Selenium

Selenium is a Python library that provides a convenient API to access Selenium WebDrivers like Firefox, Ie, Chrome, Remote etc. (Muthukadan, n.d.). Selenium allows us to write Python programs that launch browser instances, to be able to extract HTML/CSS elements and selectors from the DOM of a webpage, and retrieve data from Instagram. Selenium is the core component of the standalone Instagram scraper. With Selenium, the scraper works in steps to extract comments from Instagram posts.

First, the scraper is called through the command line, with post URL and scroll count specified. Next, the first scraper step is the browser setup, starting up the initial Firefox browser instance, via Selenium's WebDriver class. The second scraper step is navigating Firefox to the Instagram login page, and programmatically entering username and password fields with Selenium. The account login information is set in a .env file, and set into memory on scraper startup. The scraper tries to find the username and password containers through **name**, **XPATH,** and **CSS selectors**, in that order, moving to the next if one fails. Once the username and password field are found,

Selenium inputs the respective account login information, and instructs the browser to execute the “Enter” command, in order to send the login HTTP request. After the login completes, the browser is redirected to another page, where popups occur, that are most likely in-place by Meta to prevent bot detection. To combat this, the scraper sleeps for 5 seconds, refreshes, sleeps 5 more seconds, and moves on to the next scraper step. In early development, the scraper would attempt to find the actual “Login” button in the DOM, and send the HTTP request there, but that proved to be difficult and unnecessary since we can just instruct the browser to execute the “Enter” command. This small tweak eliminates a lot of running time in the grand scheme of the system.

The third scraper step is navigating to the post indicated in the URL command-line argument. The scraper navigates to the target URL through a HTTP GET request, sleeps for a random number of seconds from 3.5 - 6.5 seconds (to avoid bot detection), and then waits 10 seconds for the page to fully load.

The fourth scraper step is retrieving post metadata through post caption, date, comment count, and post like count. This is done through trying various span, article, and meta selector tags. Regular expressions are also used to determine if data extracted signifies dates, or comment count descriptions.

The fifth scraper step is finding the “open comments” button, in order to load the comments container onto the webpage. This is done through attempting various XPATH selectors. After finding the button, the button is executed, and the scraper waits 5 seconds for the comments container, and first wave of comments to fully load.

The sixth scraper step is retrieving the actual container element of the comments container. The scraper searches for a scrollable div on the page, finds it, and stores it in memory.

Finally, the last scraper step is finding comment “blocks”, meaning finding the specific container that holds one comment. Each comment box has its own container. The scraper does this by finding CSS span selectors that have been manually extracted through human eyes, and then uses the found container to walk up 15 levels within that container, in the DOM, to retrieve the high-level container that holds the entire comment block. The scraper then finds the commenter’s username, comment text, and comment likes. The first wave of comments is around 15 before needing to scroll the comments container. After the 15th comment is extracted, or the scraper does not find any new comments, it scrolls the container using the div found in the sixth step, and repeats the comment extraction process, until the amount of scrolls specified in the command-line argument is reached, or no new comments have been found. In order to prevent bot detection, the scraper attempts to scroll the comments container in a human-like way, through randomized incremental scrolling distances.

Agent

The agentic repair system uses Anthropic’s Claude Sonnet 4.6 model. The system calls this model through Anthropic’s API, using an API key with some number of paid credits. When the scraper fails at a certain step, some error message is outputted, which is fed into the agentic system, along with the indicated failed step. The agentic system then attempts to fix the scraper through editing scraper source code files,

rerunning the scraper, and repeats the process until the scraper is fixed, or the maximum number of repair attempts have been reached.

The agentic repair system is built through Python scripting and specialized LLM prompts. The first Claude API call involves a system prompt, defining the context of the scraper system, agent role, agent goals, and limitations. Along with this system prompt, a set of "tools" are attached for the agent to use when the scraper fails. When the agent receives the scraper error message and step failure, it references this set of tools and independently decides which one to use in order to accomplish the task of repairing the scraper. The set of tools include: `list_files` ,`read_file`, `write_file`, `get_page_html`, `get_element_html`, `read_element_cache`, `run_visual_inspection`, `mark_done`, and `mark_failed`. These tools are stored in JSON format, with a description of what each tool does, and its restrictions mapped to the tool name. The tools functionalities and logic are defined in a Python script that is called after the LLM decides which one to use. Some guardrails in-place are not restricting the agent to only be able to write to the scraper/ directory, not allowing it to modify the agent's own source code, and not allowing it to modify other directories within the codebase. The LLM also can not view nor read its own source code.

The core tools that allow for scraper repair are `get_page_html`, and `run_visual_inspection`. `Get_page_html` gets the HTML of the page of the failing step, and feeds it into the LLM for it to attempt to figure out why the current scraper is failing, and ultimately deciding to use `read_file` and `write_file` to replace the current Selenium selectors with updated ones, based on the HTML. `run_visual_inspection` is indicated as a last resort if `get_page_html` does not return

useful information in order to fix the scraper. Since Claude is multimodal, it has a vision model feature. `Run_visual_inspection` takes a screenshot of the failing step page, feeds it into the LLM, where the LLM visually finds the HTML container needed to fix the failing step, extracts the coordinates on the page, and uses JavaScript to retrieve the element based on the coordinates indicated. `Run_visual_inspection` is a last resort due to a higher API cost with using the vision model feature of Claude. In experimenting, this tool has worked very well in finding failing step elements on the first attempt, and is quite fascinating how efficient it can be.

Systems Summary

The scraper and agentic system are standalone, and can be run separately. The agentic system calls the scraper itself. The scraper uses Python and Selenium to spin up a Firefox browser instance, and extract data through HTML/CSS selectors. The agentic system uses scraper error output, and functional tools for it to independently decide how to repair the scraper, without human intervention. Separate bash scripts are defined for running the systems on CSVs of Instagram accounts. After a successful run of the scraper, the output is structured as CSVs of post metadata, and post comments. If the agentic system is run, there is a repair report, showing LLM prompts, outputs, decisions made, and files changed.

Conclusions

This agentic system eliminates a moderate amount of the tedious process required to repair scrapers once they break. It has successfully updated HTML/CSS selectors for each step, through both `get_page_html`, and `run_visual_inspection`, and continued running the scraper. It has been fascinating to both me, my supervisor,

and fellow peers in our research lab how easily the agent repaired the scraper. However, there are improvements to be made, open-ended questions, and many suggestions for future work.

The agentic system is entirely terminal-based, with no user interface. It would be interesting to expand on this project so that non-computing professionals can use the system with ease. The system is also limited to my specific Instagram scraper as of now, but can potentially be expanded into a framework for any web scraper, or even other systems. The system currently uses Claude Sonnet 4.6 as the LLM, which is powerful, but not as powerful as other models such as Claude Opus, potentially Claude Mythos in the future, and GPT's Codex. It may be worth experimenting with more powerful models, or even less powerful and less costly models and see how the agent performs. Expanding on this, metrics can be added as well, to examine when the agent fails, number of successful repairs vs. number of failed repairs, number of repair attempts before success, etc. Another important expansion could be another tool, similar to `get_page_html` and `run_visual_inspection`, but not as token-cost heavy, and still providing enough context for the agent to repair the scraper.

As for open-ended questions, Instagram data is not very abundant, unless you pay hefty prices for it. There are so many research questions to be explored with Instagram data, such as: What are the differences between creator content a few years ago vs. now? Why did they change their content? What patterns do we see in social groups of commenters on Instagram posts? What posts receive more likes than others, and why? How prominent is race fetishization on Instagram, and why does it occur? Some other interesting non-related Instagram questions are: With the rapid

advancements of AI, can we analyze data through images and videos, rather than text and numbers? How can self-repairing agents be used in other scrapers, or systems? Can we create models that are solely for repairing scrapers?

This project is just a starting point for agentic self-repairing scrapers, and can very likely be expanded into a framework for all scrapers, or other systems. Computational Social Science enables us to analyze social human data at scale, and find patterns we haven't seen before. Agentic AI shows much promise in the CSS space, eliminating time-consuming tasks, and allowing for more in-depth data analysis. I hope that my project can be expanded on, and potentially be used throughout the research lab, to make web scraping and other CSS systems an easier task.

References

Lazer, D., Pentland, A., Adamic, L., Aral, S., Barabási, A.-L., Brewer, D., Christakis, N., Contractor, N., Fowler, J., Gutmann, M., Jebara, T., King, G., Macy, M., Roy, D., & Van Alstyne, M. (2009). Computational social science. *Science*, *323*(5915), 721–723. https://doi.org/10.1126/science.1167742

Wahed, M. A., Alzboon, M. S., Alqaraleh, M., Ayman, J., Al-Batah, M., & Bader, A. F. (2024). Automating web data collection: Challenges, solutions, and Python-based strategies for effective web scraping. In *2024 7th International Conference on Internet Applications, Protocols, and Services (NETAPPS)* (pp. 1–6). IEEE. https://doi.org/10.1109/NETAPPS63333.2024.10823528

Hosseini, S., & Seilani, H. (2025). The role of agentic AI in shaping a smart future: A systematic review. *Array*, *26*, 100399. https://doi.org/10.1016/j.array.2025.100399

Muthukadan, B. (n.d.). *Selenium with Python*. Read the Docs. Retrieved May 11, 2026, from https://selenium-python.readthedocs.io/