DS490: Data Journalism for Modern Investigative Research Methodologies (“Data Journalism II”)

Directed Study under Langdon White (CDS)

Andrew Botolino, Inhye Kang, Claire Law, Crosby Nash

Boston University

Duan Family Center for Computing & Data Sciences

665 Commonwealth Ave, Boston, MA 02215

May 6, 2025

1. Introduction

Data journalism is quickly becoming more technically sophisticated, and today’s journalists must adapt and advance their own technical skillset to conduct investigative research, maintain security on the web, and effectively communicate their work and goals to high-tech teammates. In collaboration with journalists, those tech professionals can leverage their skills to help enable a fundamental mission of journalism: holding power to account.

This directed study, which we dubbed “Data Journalism II,” builds upon the foundations of data journalism learned in COM JO521 (Data Journalism), which taught how to identify and obtain government data, download and import data, and perform basic data scraping, cleaning, standardization, analysis, and visualization.

This report documents our work in this course during the Spring 2025 semester. Working as a team of four students, we explored advanced techniques for web scraping, privacy and security, open-source software, and more. The course structure was guided by a framework that will eventually make up an official class, Data Journalism II, though adapted to our specific learning needs and collaborative project goals.

Our directed study honed in on developing both theoretical knowledge and practical skills in several areas: command line tools, Google dorking, version control, large file management, web scraping, privacy and security considerations, and exploration of the dark web — all through an open-source mindset and lens. These topics provided us with a foundation for understanding how journalists can leverage technology to access and use data in investigative reporting.

The Data Journalism II course will have a central project as a part of its coursework. During the semester, we developed an automated scraper tool designed to make data acquisition via scraping more accessible to journalists. Our vision was to create a solution that would allow users to simply enter a URL and a prompt, instructing an LLM to extract specific data from the website, and have that relevant data automatically extracted behind the scenes, without requiring programming knowledge from the user. In building this tool, our project integrated many of the course's theoretical concepts into a practical application with real-world utility for investigative journalism.

Throughout the semester, we faced various challenges that provided valuable learning experiences. Most notably: start sooner! While researching and brainstorming potential project ideas was exciting and fun, considering the given time constraint of a semester, we could have benefited greatly from earlier project scoping and timeline management. While we made significant progress on our scraper tool, a quicker definition of project parameters would have allowed for more comprehensive implementation and testing.

These lessons will not only inform our future academic and professional work but also guide subsequent iterations of this course.

The rest of this report details our learning journey, the development of our technical solution, results from testing, and reflections on both the process and outcomes. It represents not only what we accomplished but also provides a roadmap for future exploration — for us and future students — in the rapidly evolving intersection of journalism and data science.

2. About the Class and Topic Areas

2.1 Core Topics Covered

Our directed study explored technical fundamentals for modern investigative journalism, focusing on practical tools and methodologies that enhance research, data collection, and security. While we didn't cover every topic in the original syllabus, we prioritized skills with immediate applications in the field. The following areas formed the core of our technical training:

2.1.1 Command Line Tools:

We learned the fundamentals of command-line operations, a foundational skill for efficient data handling and automation. For those of us who took CAS CS210 (Computer Systems), this provided valuable hands-on experience applying our theoretical knowledge of Unix systems in a real-world context. We explored essential terminal commands, including file navigation (cd, ls), text processing (grep, awk), and network tools (wget, cURL). This was immediately valuable, especially in the context of our project. It enabled us to automate document collection from public archives and extract structured data from websites. We implemented various CLI techniques such as piping (|) for data transformation, redirection operators (>, >>) for output management, and job control (&, bg, fg) for process management. By building Bash scripts to streamline repetitive tasks, we significantly reduced manual processing time while ensuring our workflows remained reproducible. These command-line skills became essential throughout our project lifecycle, from initial data gathering to final analysis and verification stages.

2.1.2 Google Dorking:

Advanced search techniques, or "Google Dorking," help uncover publicly available but poorly indexed information. We practiced using specialized operators (site:, filetype:, intitle:, inurl:) to locate sensitive documents, exposed data, and unsecured web directories. For instance, searching filetype:pdf site:gov "confidential report" can reveal improperly published government records. We also examined how cached pages and date-range filters help track changes to online content, useful for detecting edits or deletions in archived material.

2.1.3 Version Control:

We began by examining the history of version control systems, from early centralized systems like CVS to Linus Torvalds' creation of Git in 2005 to manage Linux kernel development. This historical context helped us better understand Git's distributed architecture and its advantages for collaborative work. Building on our prior coursework in software engineering and computer systems, we were able to apply advanced Git concepts in practice, including interactive rebasing from CAS CS411 to clean up commit histories and CI/CD pipeline integration from previous projects to automate testing. In our project, we implemented Git workflows to manage our evolving codebase, giving us firsthand experience of how critical systematic version control is for collaborative work. As multiple team members contributed to shared files, we saw how Git's branching and merging capabilities prevented conflicts while maintaining a clear history of changes. This practical application reinforced our classroom learning, demonstrating how proper version control enables seamless collaboration among distributed teams working on complex projects.

2.1.4 Large File Management:

Investigative journalism often involves handling substantially large datasets—leaks, FOIA responses, or multimedia evidence. An invaluable resource is containerization, which allows for packaging data processing tools and dependencies into portable, reproducible environments.

A popular containerization platform is Docker, a pioneering tool that standardized and popularized containers. However, for this project, we opted to use containers independently of Docker as much as possible—using Containerfiles instead of Dockerfiles, for instance—as it better aligned with our open source goals.

Containers are critical for reproducibility, security, and scalability. Not only do they ensure datasets can be processed identically across different team members' systems, they also act as isolated environments, which are ideal for sensitive data operations and minimize risks of system-wide contamination or accidental data leaks. Docker's lightweight nature lets us spin up multiple instances of tools like grep or awk to parallel-process large datasets.

By integrating containers with our existing CLI workflows, we can bridge the gap between raw data processing and reproducible investigation—showing that containerization is no longer just for software engineers, but a vital skill for data-driven journalists as well.

2.1.5 Web Scraping:

Web scraping allows journalists to systematically collect public data from websites, especially useful for non-governmental data and less structured data. We compared lightweight parsers like BeautifulSoup (for static HTML) with browser automation tools like Selenium (for JavaScript-heavy sites). Practical exercises included scraping government databases, news archives, and social media profiles. We also discussed legal boundaries — such as avoiding denial-of-service risks — and ethical considerations, like anonymizing scraped personal data. By combining scraping with data cleaning and formatting tools (e.g. Pandas, JSON), we transformed raw web content into structured datasets for analysis, which was a significant part of our project.

2.1.6 Privacy and Security:

Protecting sources and maintaining operational security is non-negotiable. We learned essential privacy measures for journalists, including how encrypted communication (ProtonMail, Signal), anonymization tools (Tor, Tails OS), and secure file storage (VeraCrypt, encrypted USBs) work. We learned about VPNs and SSH tunneling, specifically how they are configured to obscure network activity, and how GPG encrypts sensitive documents. We also discussed "burner" workflows, such as how to set up disposable emails.

2.1.7 Dark Web Exploration:

The dark web presents both unique opportunities and significant risks for investigative work. Using Tor, we accessed hidden services (.onion sites) hosting valuable resources such as leaked document archives, anonymous tip lines, and forums where whistleblowers may communicate. These platforms can serve as critical sources for uncovering corruption, human rights violations, or other public interest matters that surface in obscured corners of the internet.

We carefully considered the limitations and ethical boundaries of dark web research. While some sites host legitimate whistleblower materials, many others facilitate illegal activity, requiring journalists to navigate these spaces with clear ethical frameworks. We established strict protocols, never interacting with illicit content, avoiding personal identification, and using dedicated, secure devices isolated from our primary work systems. Operational security was also reinforced—we accessed sensitive material only through Tor on hardened machines, never from personal devices or unprotected networks.

Our exploration revealed that while the dark web can be a powerful tool for accessing censored information or communicating with protected sources, its utility for journalism is often overstated. Many sites lack reliable information, and verification of dark web-sourced materials requires extra scrutiny.

Ultimately, we approached the dark web as journalists, not as participants, maintaining detachment and learning about how its users communicate — jargon, tendencies, etc. — before booting in, while also recognizing its potential as one of many tools in investigative research. This allowed us to explore its capabilities while upholding ethical standards and minimizing legal risks.

2.2 Open Source Philosophy & Licensing in Investigative Journalism

The open-source movement has revolutionized software development by promoting transparency, collaboration, and accessibility. As outlined by

OpenSource.com, open-source software (OSS) grants users four freedoms: to use, study, modify, and distribute code. These principles align with journalistic values. Transparency ensures tools can be audited for biases or vulnerabilities, while collaborative development fosters innovation. Investigative reporters benefit immensely from OSS; tools like Maltego (link analysis), OCRopus (document scanning), and SecureDrop (whistleblower submissions) are all open-source, enabling newsrooms to adopt them without restrictive licensing costs.

However, not all open licenses are equal. The Business Source License (BSL), for example, temporarily restricts commercial use before converting to a standard OSS license. While useful for startups seeking monetization, BSL's limitations conflict with journalism's public-interest ethos. In our project, we rejected it in favor of permissive licenses (MIT for code, CC BY-NC for creative works) to maximize accessibility.

Creative Commons licenses extend open principles beyond software to articles, datasets, and multimedia. The CC BY-NC (Attribution-NonCommercial) license was chosen for our written work to prevent commercial exploitation while allowing educational/activist reuse. For code, aside from the CC BY-NC license, the MIT License was selected for its simplicity and compatibility — ensuring newsrooms, NGOs, and researchers can freely integrate our tools.

Legally, CC licenses are enforceable worldwide and irrevocable, offering strong protections. They also permit "license stacking," allowing derivative works to adopt more permissive terms (e.g., CC BY-NC content remixed into a CC BY project). This flexibility ensures our investigative methodologies can evolve while retaining attribution.

Ultimately, our licensing strategy reflects a commitment to ethical open knowledge: tools should be free to use, but research must remain non-commercial to avoid misuse. By combining MIT’s developer-friendly terms with CC BY-NC’s protective safeguards, we balance innovation with accountability, a model for modern investigative work.

2.3 Areas for Deeper Exploration

While our directed study established core technical competencies, several areas merit deeper exploration to address evolving investigative challenges.

2.3.1 Digital Forensics:

In the future, we could fully explore digital forensics techniques through hands-on exercises in image steganography using tools like StegExpose to detect hidden data in leaked documents, alongside advanced geolocation techniques available.

Multimedia verification skills are becoming increasingly critical as manipulated images and AI-generated content proliferate, necessitating deeper work with metadata analysis through tools such as ExifTool and error-level examination in FotoForensics.

2.3.2 Social Media Mining:

Social media mining presents both opportunities and obstacles for journalists. While platforms increasingly restrict API access, we could develop more robust frameworks for ethical data collection, combining legal scraping tools like Twint with human-centered methods such as crowdsourced verification. That said, legal scraping of social media is difficult enough that we might consider omitting this from future syllabi.

2.3.3 OSINT Techniques:

Open Source Intelligence techniques deserve particular attention given their centrality to modern investigations. A more comprehensive curriculum should include methodologies such as network mapping to visualize relationships between entities, temporal reconstruction of deleted content using archival services, and systematic verification pipelines for visual evidence. The depth of OSINT methodologies extends beyond what we could incorporate into our directed study, but they are still crucial to learn, which suggests this might justify an entire course dedicated to it.

2.4 Reflection on Course Structure

The directed study format offered both advantages and challenges for our learning experience:

2.4.1 Collaborative Learning Environment:

Working as a small team of four students allowed for close collaboration and peer learning, particularly valuable when tackling technical challenges in our scraper project. We were able to meet in person frequently, which, as we came to learn, was extremely valuable to the success of the project. Whether debugging scraping scripts or discussing structural adjustments, peer feedback created a real-world simulation of work done at a high-tech data desk or in small software engineering teams. Knowledge transfer happened organically, partly due to our team's chemistry as well: for instance, team members with CS backgrounds helped others understand and contribute to more complex LLM models and algorithms, while those with reporting experience guided journalistic considerations for data collection. However, we noted that a clearer role definition early on (e.g., assigning lead responsibilities for documentation, testing, or front-end development) could have further enhanced productivity later on for the project.

2.4.2 Flexible Topic Selection:

The directed study allowed us to adjust our focus areas based on our evolving project needs and interests, which perhaps would not have been as feasible in a traditional course structure. That said, a suggested improvement would be maintaining a "core curriculum" of non-negotiable skills (e.g., basic CLI, Git, and BeautifulSoup) while reserving flexibility for advanced topics.

2.4.3 Project-Based Learning:

Centering our work around a deliverable — the automated scraper tool — provided context for the technical skills we were developing. Our work on the tool bridged the gap between theoretical knowledge gained through our readings and discussions and the practical application of these concepts in a real-world setting. The hands-on experience of developing a tool enhanced our technical understanding of how different websites are structured, how scrapers and crawlers work, and our understanding of what data is valuable journalistically. For example, our project involved some forms of reverse-engineering website structures to reveal how DOM manipulation varies across news sites (e.g., NPR’s semantic HTML vs. CNN’s dynamic loading). In addition, we learned to identify what is considered "scrapable" data with reporting value.

2.4.4 Earlier Project Scoping:

As aforementioned, by the time we settled on what tool to develop, the semester timeline left limited opportunity for comprehensive implementation and testing. As such, our project represents a proof of concept rather than a fully developed implementation.

If this class were to be further developed, this could be accelerated by a structured “sprint 0” for requirements gathering — for instance, analyzing 3-5 potential target sites before committing to one. Future students could also benefit from template proposals listing data sources, ethical considerations, and successful implementations from the past.

2.4.5 Balancing Breadth and Depth:

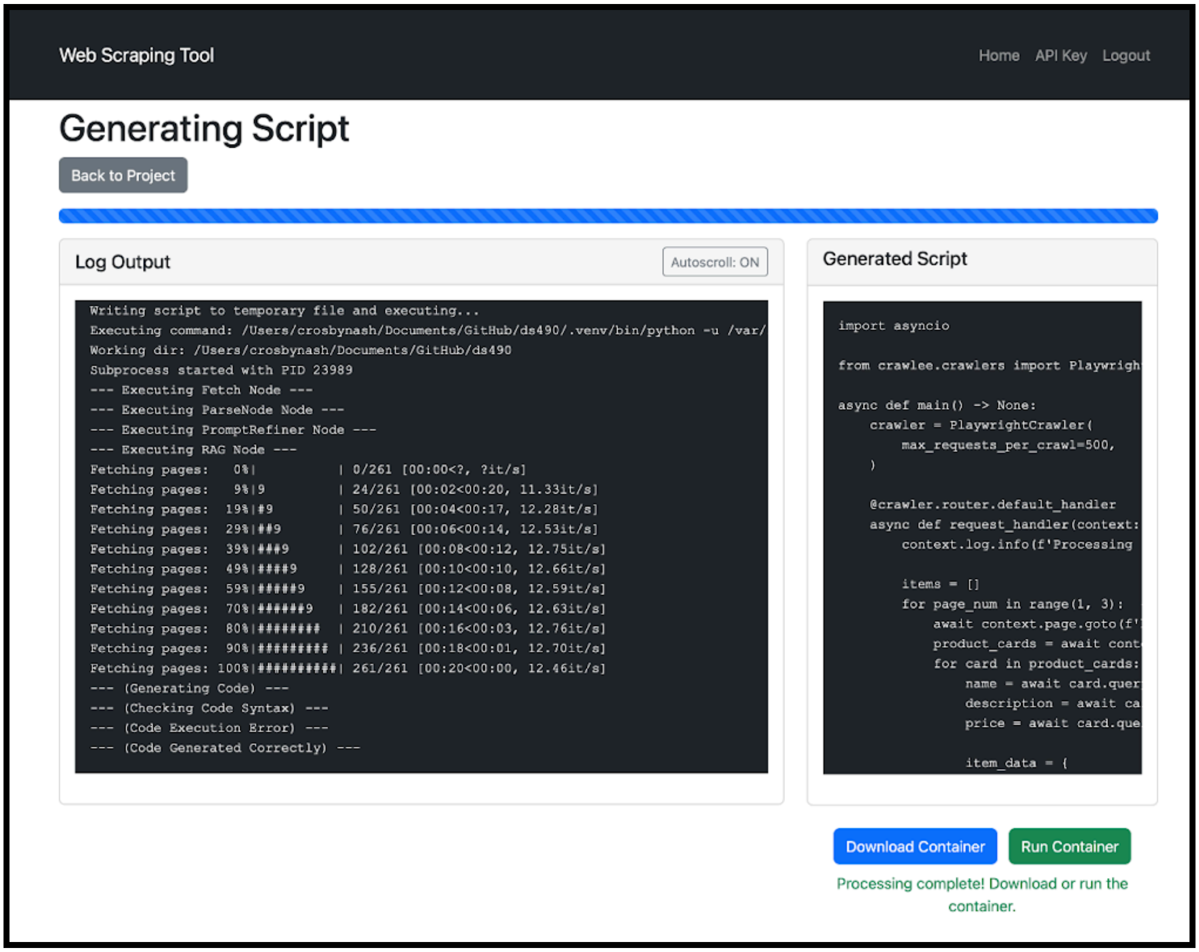
The range of topics in our outline was ambitious, skewed towards breadth. In future iterations, narrowing the scope of topics covered would allow for deeper dives into specific areas that would enhance mastery of core skills. For example, more hands-on time spent on Git to develop journalism students with a working knowledge of version control and less on Google dorking, which is taught to some extent in Data Journalism I, could be greatly beneficial in future iterations.

Overall, the directed study provided valuable exposure to the technical foundations of data journalism.

3. Automated Scraper Tool Project

3.1 Project Overview

The scraper tool's primary purpose is to provide a robust, end-to-end framework for extracting structured information from arbitrary web pages with minimal manual intervention. The tool is designed to guarantee reliability and repeatability by ensuring consistent extraction results across runs, even when underlying page layouts change. It incorporates mechanisms to automatically detect and adapt to standard web design patterns, such as HTML tables, card grids, and paginated lists.

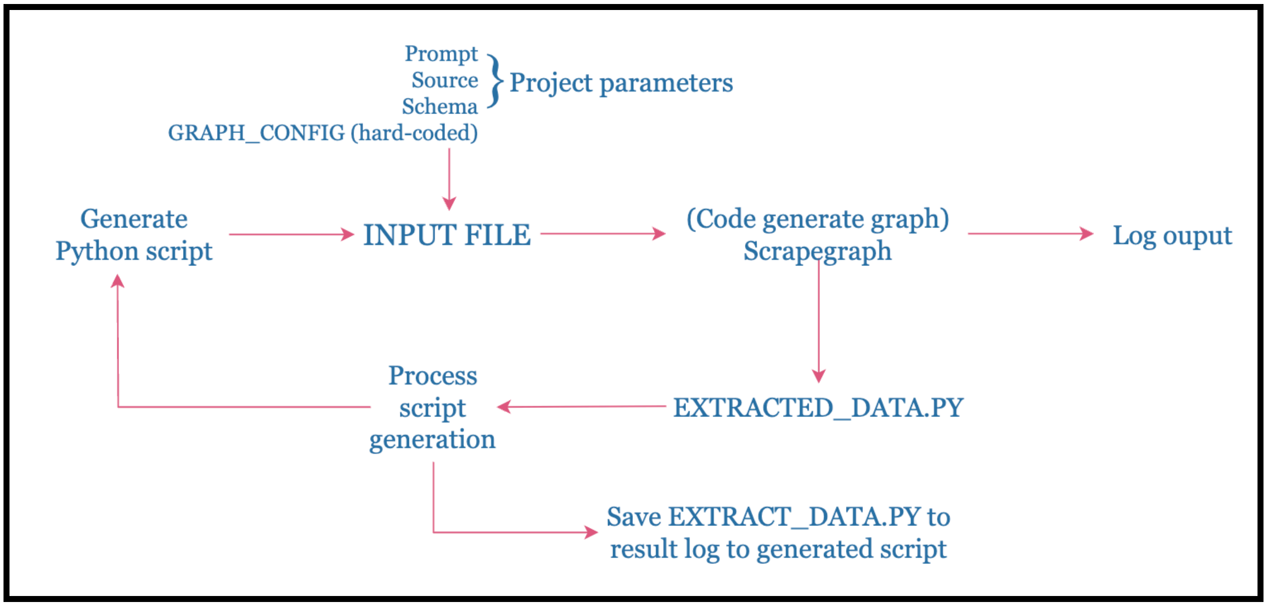


```
Writing script to temporary file and executing...
Executing command: /Users/crosbynash/Documents/GitHub/ds490/.venv/bin/python -u /var/
Working dir: /Users/crosbynash/Documents/GitHub/ds490
Subprocess started with PID 23989
--- Executing Fetch Node ---
--- Executing ParseNode Node ---
--- Executing PromptRefiner Node ---
--- Executing RAG Node ---
Fetching pages:   0%|          | 0/261 [00:00<?, ?it/s]
Fetching pages:   9%|9         | 24/261 [00:02<00:20, 11.33it/s]
Fetching pages:  19%|#9        | 50/261 [00:04<00:17, 12.28it/s]
Fetching pages:  29%|##9       | 76/261 [00:06<00:14, 12.53it/s]
Fetching pages:  39%|###9      | 102/261 [00:08<00:12, 12.75it/s]
Fetching pages:  49%|####9     | 128/261 [00:10<00:10, 12.66it/s]
Fetching pages:  59%|#####9    | 155/261 [00:12<00:08, 12.59it/s]
Fetching pages:  70%|######9   | 182/261 [00:14<00:06, 12.63it/s]
Fetching pages:  80%|########  | 210/261 [00:16<00:03, 12.76it/s]
Fetching pages:  90%|######### | 236/261 [00:18<00:01, 12.70it/s]
Fetching pages: 100%|##########| 261/261 [00:20<00:00, 12.46it/s]
--- (Generating Code) ---
--- (Checking Code Syntax) ---
--- (Code Execution Error) ---
--- (Code Generated Correctly) ---
```

```
import asyncio

from crawlee.crawlers import Playwrigh

async def main() -> None:
    crawler = PlaywrightCrawler(
        max_requests_per_crawl=500,
    )

    @crawler.router.default_handler
    async def request_handler(context:
        context.log.info(f'Processing

        items = []
        for page_num in range(1, 3):
            await context.page.goto(f'
            product_cards = await cont
            for card in product_cards:
                name = await card.quer
                description = await ca
                price = await card.que

                item_data = {
```

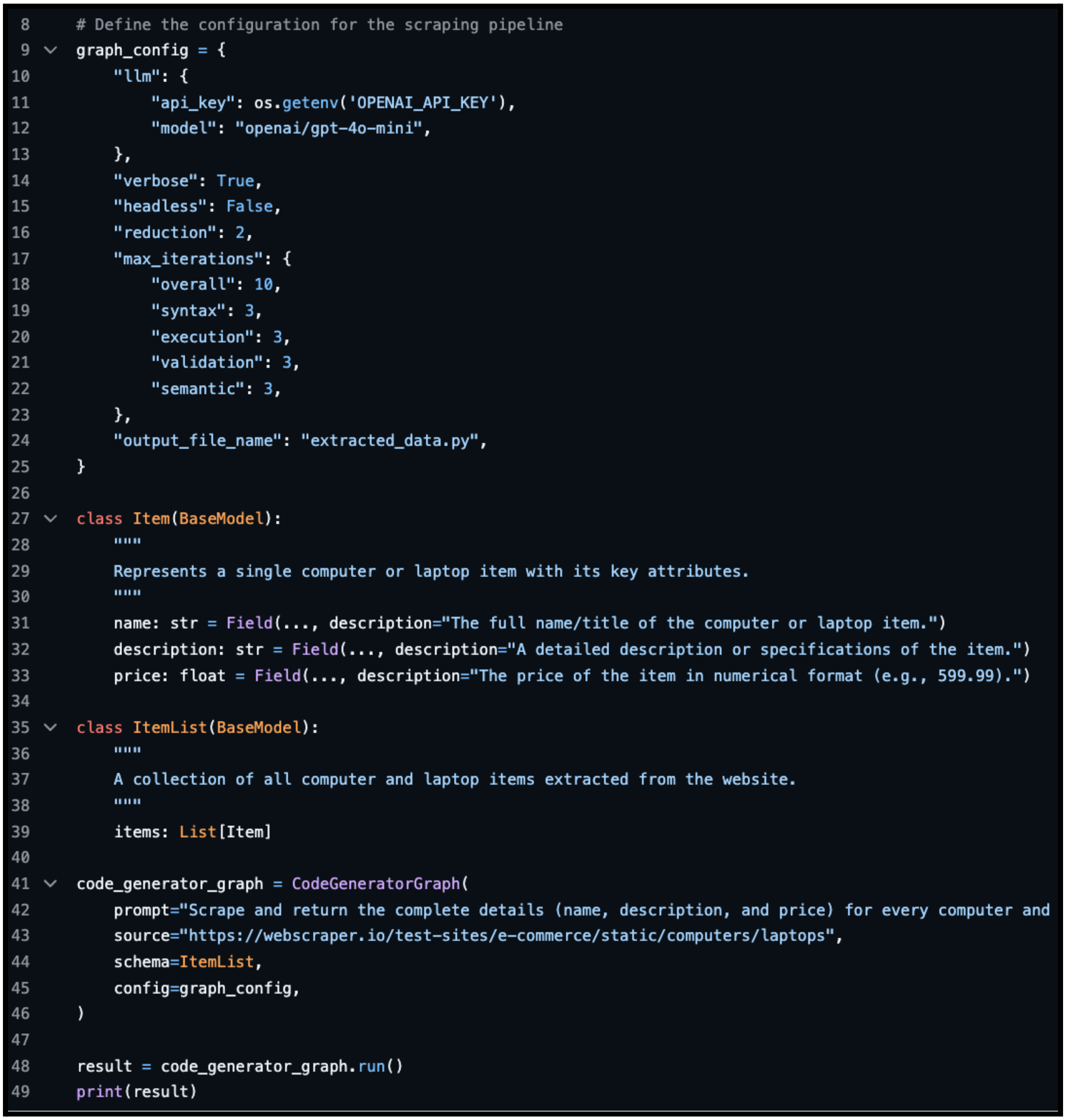
Processing complete! Download or run the container.

We built a custom scraping pipeline using the ScrapeGraph library, an LLM-powered scraper generation system. Users specify a target website, a Pydantic schema (comprising field names and types), and a natural language prompt. These inputs are passed into CodeGeneratorGraph, a modified version of the one provided by the ScrapeGraph library, which emits a fully-formed Python script housing scraping code based on the Crawlee library. When executed, this script performs the actual scraping and outputs the results to CSV or JSON files that are saved to the disk. The frontend is built in Django, chosen for its batteries-included approach offering built-in authentication, routing, templating, ORM, CSRF protection, and robust form handling. Django’s secure defaults and straightforward mapping between views, forms, and models allowed for rapid development with minimal glue code. The core data models include Project (containing metadata like URL, LLM prompt, and crawl options), FieldSpecification (defining the user’s output schema), APIKey (for securely storing each user’s model key), and ScrapingResult (which records logs, output content, and execution status). The UI spans the full lifecycle: account signup, API key entry, project creation/editing with dynamic schema rows, and a project detail page that lists prior runs and schema settings. Triggering script generation launches a background thread that composes a template via generate_python_script_template(), executes the script in a subprocess, and captures output and logs into the ScrapingResult object. A dedicated status page uses AJAX polling of the /get_logs/ endpoint to stream real-time logs, monitor progress, preview the generated script, and enable container downloads once complete. Scraped data is currently not viewable within the UI but is planned to be viewable in a results page that pretty-prints the output and allows downloading the full container package. Containerization is also not currently implemented but will be handled in-process via the Containerizer class, bundling the generated script, a Containerfile, and requirements.txt into a downloadable ZIP archive.


Prompt
Source
Schema
Project parameters
GRAPH_CONFIG (hard-coded)
Generate Python script
INPUT FILE
(Code generate graph) Scrapegraph
Log ouput
Process script generation
EXTRACTED_DATA.PY
Save EXTRACT_DATA.PY to result log to generated script

3.1.1 Input

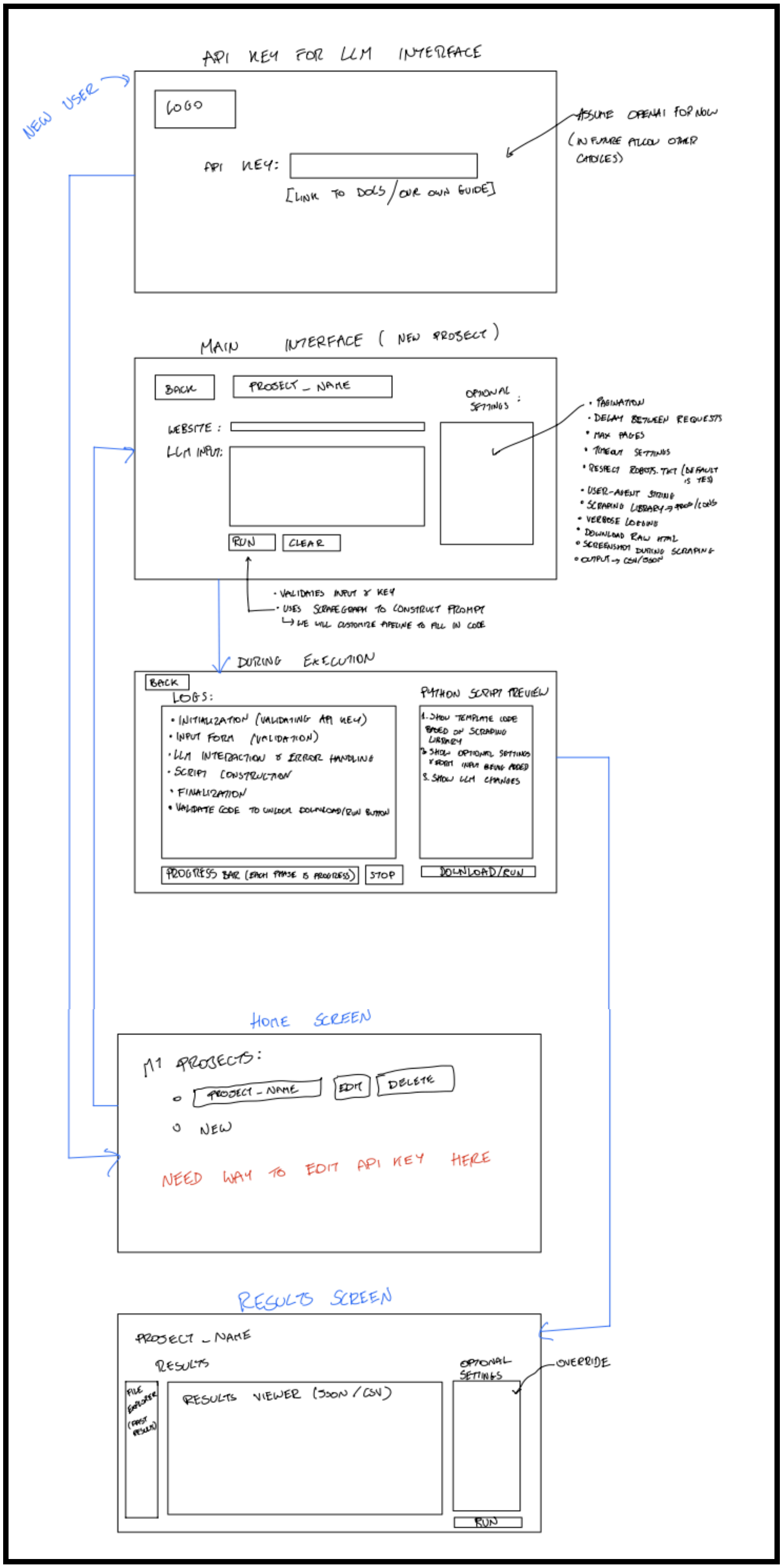
The image below shows what the input looks like from the backend:



```python
# Define the configuration for the scraping pipeline
graph_config = {
    "llm": {
        "api_key": os.getenv('OPENAI_API_KEY'),
        "model": "openai/gpt-4o-mini",
    },
    "verbose": True,
    "headless": False,
    "reduction": 2,
    "max_iterations": {
        "overall": 10,
        "syntax": 3,
        "execution": 3,
        "validation": 3,
        "semantic": 3,
    },
    "output_file_name": "extracted_data.py",
}

class Item(BaseModel):
    """
    Represents a single computer or laptop item with its key attributes.
    """
    name: str = Field(..., description="The full name/title of the computer or laptop item.")
    description: str = Field(..., description="A detailed description or specifications of the item.")
    price: float = Field(..., description="The price of the item in numerical format (e.g., 599.99).")

class ItemList(BaseModel):
    """
    A collection of all computer and laptop items extracted from the website.
    """
    items: List[Item]

code_generator_graph = CodeGeneratorGraph(
    prompt="Scrape and return the complete details (name, description, and price) for every computer and
    source="https://webscraper.io/test-sites/e-commerce/static/computers/laptops",
    schema=ItemList,
    config=graph_config,
)

result = code_generator_graph.run()
print(result)
```

CodeGeneratorGraph takes four main inputs:

- **Prompt**: A natural language instruction describing what data to scrape and any specific requirements. In the image above, it specifies scraping computer/laptop details (name, description, price) from both the first and second pages of the target website. The prompt guides the AI in understanding the scraping objectives.
- **Source**: The target URL or local directory containing the data to be scraped. Here, it points to a test e-commerce website (webscraper.io) specifically configured for scraping practice, focusing on the laptops section. This tells the system where to find the data.
- **Schema**: A Pydantic model (ItemList) that defines the exact structure and validation rules for the output data. It specifies that the result should contain a list of items, where each item has three validated fields: name (string), description (string), and price (float). This ensures the generated code will extract and format data correctly.
- **Config**: A dictionary containing various settings, including the AI model configuration (GPT-4 with an API key), verbosity level, browser mode (headless or visible), HTML reduction factor, iteration limits for different validation steps, and the output filename for the generated script. These parameters control how the code generation process executes.


API KEY FOR LLM INTERFACE
NEW USER
LOGO
API KEY:
[LINK TO DOCS / OUR OWN GUIDE]
ASSUME OPENAI FOR NOW
(IN FUTURE ALLOW OTHER CHOICES)
MAIN INTERFACE (NEW PROJECT)
BACK
PROJECT _ NAME
OPTIONAL SETTINGS :
WEBSITE :
LLM INPUT:
RUN
CLEAR
• PAGINATION
• DELAY BETWEEN REQUESTS
• MAX PAGES
• TIMEOUT SETTINGS
• RESPECT ROBOTS.TXT (DEFAULT IS YES)
• USER-AGENT STRING
• SCRAPING LIBRARY → PROS / CONS
• VERBOSE LOGGING
• DOWNLOAD RAW HTML
• SCREENSHOT DURING SCRAPING
• OUTPUT → CSV/JSON
• VALIDATES INPUT & KEY
• USES SCRAPEGRAPH TO CONSTRUCT PROMPT
↳ WE WILL CUSTOMIZE PIPELINE TO FILL IN CODE
DURING EXECUTION
BACK
LOGS:
• INITIALIZATION (VALIDATING API KEY)
• INPUT FORM (VALIDATION)
• LLM INTERACTION & ERROR HANDLING
• SCRIPT CONSTRUCTION
• FINALIZATION
• VALIDATE CODE TO UNLOCK DOWNLOAD/RUN BUTTON
PYTHON SCRIPT PREVIEW
1. SHOW TEMPLATE CODE BASED ON SCRAPING LIBRARY
2. SHOW OPTIONAL SETTINGS & FORM INPUT BEING ADDED
3. SHOW LLM CHANGES
PROGRESS BAR (EACH PHASE IS PROGRESS)
STOP
DOWNLOAD/RUN
HOME SCREEN
MY PROJECTS:
○ PROJECT _ NAME
EDIT
DELETE
○ NEW
NEED WAY TO EDIT API KEY HERE
RESULTS SCREEN
PROJECT _ NAME
RESULTS
OPTIONAL SETTINGS
OVERRIDE
FILE EXPLORER (FIRST RESULT)
RESULTS VIEWER (JSON / CSV)
RUN

3.2 CodeGeneratorGraph Overview

The CodeGeneratorGraph, as implemented in version 1.48.0 of the ScrapeGraph library, represents the baseline architecture for orchestrating large language model-driven web scraping workflows. Defined in scrapegraphai/graphs/code_generator_graph.py, it subclasses AbstractGraph and encapsulates a fully linear DAG of processing nodes that operate over a shared mutable state. Upon instantiation, the graph accepts four main inputs: a user-supplied natural language prompt, a source identifier (which may be a URL or local directory), a configuration dictionary containing flags and model parameters, and a Pydantic BaseModel schema describing the structure of the desired output. Based on whether the source string begins with http, the graph dynamically sets the appropriate input key (url or local_dir) for the pipeline.

The _create_graph() method wires together six core nodes in a fixed sequence: FetchNode → ParseNode → GenerateAnswerNode → PromptRefinerNode → HtmlAnalyzerNode → GenerateCodeNode. These nodes are chained such that each one reads from and writes to a shared state dictionary, enabling modular state transformation and dependency resolution. When .run() is invoked, the graph constructs the initial inputs dictionary using the user prompt and source, executes the DAG with self.graph.execute(inputs), extracts the final generated_code from the resulting state, and writes it to disk using the save_code_to_file() utility before returning the code as a string.

Each node inherits from BaseNode and implements an execute(state: dict) -> dict interface. The FetchNode handles content acquisition, either by loading local data formats (e.g., CSV, JSON, PDF) or by launching a Playwright-driven ChromiumLoader for dynamic scraping of web pages. This node outputs the raw HTML and optionally transforms it into Markdown or cleans it depending on the configuration. ParseNode consumes this output and converts the HTML into plain text using LangChain’s Html2TextTransformer, splitting it into manageable chunks while extracting auxiliary information such as links and images.

The GenerateAnswerNode selects a prompting strategy based on the number of chunks and routes the LLM call through a single chain or a fan-out/merge pattern using LangChain's RunnableParallel. If a schema is provided, the node activates a structured output parser to enforce schema conformity. Prompt refinement is handled by PromptRefinerNode, which injects the schema into a concise rewriting prompt and generates a schema-aligned instruction for downstream code generation. HtmlAnalyzerNode then consumes the refined prompt and original

HTML to produce an annotated understanding of the page’s structure, including a pruned subset of the HTML for input efficiency.

The culminating step is the GenerateCodeNode, which conducts a multi-stage reasoning loop to synthesize a Python script. It begins with initial code generation using a Jinja-style template, followed by successive refinement phases focused on syntax correctness, runtime stability, schema validation, and semantic fidelity. Each phase incorporates feedback—whether from ast.parse, test execution of the extract_data(html) function, or a comparison against the LLM-generated example output—to iteratively regenerate and improve the code. This process repeats until the script either satisfies all constraints or exhausts the maximum number of iterations, at which point the node raises an error.

The execution of the CodeGeneratorGraph thus proceeds as a tightly integrated pipeline where each transformation stage incrementally builds upon the last, and all intermediate artifacts—including raw HTML, tokenized text, refined prompts, and analysis summaries—are stored in the shared state. The result is a standalone, executable Python script tailored to the user’s specification, output schema, and target site.

However, the vanilla implementation of CodeGeneratorGraph revealed two critical limitations that constrained its scalability and reliability. It regenerated an entirely new script on every run, even for minor prompt or schema changes, leading to unnecessary LLM usage and redundant downstream processing. Also, the scripts it produced were based on the requests and BeautifulSoup libraries, which, while lightweight, proved fragile when handling modern, JavaScript-heavy websites. Parsing dynamic content, paginated lists, or deeply nested structures often failed without explicit user intervention. To overcome these limitations, we transitioned to a template-driven approach using the Crawlee framework, which offers robust headless browser automation via Playwright. Rather than tasking the LLM with emitting fully novel code, we now ask it to populate structured placeholders within a predefined Crawlee script template. This shift not only improved reliability and maintainability but also made it significantly easier to validate, cache, and debug the generated scripts.

To adapt the vanilla CodeGeneratorGraph to the needs of a robust Crawlee-based code generation pipeline, several core components were customized or replaced. These changes centered around enhancing generation reliability, incorporating retrieval-augmented generation (RAG), and introducing aggressive caching to reduce redundant computation during iterative development.

At the node level, two major substitutions were made. The upstream scrapegraphai.nodes.GenerateCodeNode was replaced by a custom Crawlee-flavored implementation located in nodes/generate_crawlee_code_node.py. Similarly, a new RAGNode was introduced via nodes/crawlee_rag_node.py to inject retrieval capabilities into the graph. This RAG step was placed between the existing HtmlAnalyzerNode and the final code generation node, creating a seven-node graph. The RAGNode constructs or restores a Qdrant vector store populated with Crawlee's official Python documentation, which it embeds and stores as state["vectorial_db"]. This enables downstream nodes to ground LLM output in authoritative, task-relevant examples retrieved via similarity search.

In support of RAG, the configuration system was extended to allow embedding and retrieval of parameters to be passed through the graph constructor. These include the number of documents to retrieve at various stages (initial_k, execution_k, validation_k), the LLM's reasoning loop limits (max_iterations), and the embedding model used to vectorize queries (defaulting to OpenAI's embeddings). These parameters are sourced from a centralized NODE_DEFAULTS module but can be overridden via the config argument.

The construction of GenerateCodeNode itself was also modified to accept a wide range of inputs via keyword expansion. In addition to the LLM model and retrieval parameters, this configuration passes through schema definitions, template injection fields, embedding models, and optional debugging metadata. This allowed for tight integration between prompt templates, code refinement logic, and retrieved vector data, without sacrificing node modularity.

The .run() method of CodeGeneratorGraph was fully rewritten to support intelligent caching. Whereas the upstream version simply executed the full graph from scratch on every invocation, the revised logic computes a SHA-256 hash over the concatenated prompt and source URL to generate a cache key. On first run—or if force=True is set—all nodes except the final GenerateCodeNode are executed, and their intermediate outputs are serialized into a JSON cache file. On subsequent invocations, the system reloads this cached state, rehydrates the vector store into a Qdrant client, and proceeds directly to the final code generation step. This dramatically reduces iteration time when only minor schema or prompt changes are being tested.

The RAGNode itself is responsible for ingesting Crawlee's documentation via a DocusaurusLoader, filtering down to /python/api/ endpoints, and embedding each page's content using the configured model. These embeddings are indexed in Qdrant under a cosine distance metric and stored as points enriched with metadata for later lookup. The node is idempotent, rebuilding the full collection from scratch on each invocation to ensure freshness and eliminate drift.

The custom GenerateCodeNode was heavily modified to align with Crawlee's Playwright-based scripting model. Rather than emitting a standard BeautifulSoup parser with Pydantic validation, the new node populates a Jinja-style DEFAULT_CRAWLEE_TEMPLATE with dynamically selected documentation snippets and schema-driven logic. The node orchestrates a series of reasoning loops—initial generation, syntax validation, execution testing, and schema verification—at each of which it constructs targeted similarity search queries to retrieve documentation excerpts relevant to the current failure. These examples are embedded into the prompt before regeneration, grounding each iteration in concrete reference material from the Crawlee API.

In the execution loop, for example, the system writes the candidate script to disk and runs it in a subprocess. If an error or timeout occurs, a retrieval query is constructed using the error message and the original prompt. The top execution_k matches are retrieved from Qdrant, prepended to the execution-focused analysis prompt, and fed back into the model for a new candidate script. A similar loop is implemented for schema validation using jsonschema.validate(), with fallback prompts grounded in documentation that references relevant field types or formats. Semantic comparison is stubbed but present, enabling future support for evaluating content-level fidelity against the LLM's original “answer.”

Together, these changes transform the CodeGeneratorGraph from a generic scraping pipeline into a hybrid retrieval-generation framework specifically optimized for Crawlee’s ecosystem. By combining precise documentation grounding, state-aware regeneration, and aggressive caching, the modified system achieves a significantly faster, more reliable development loop for generating production-grade scraping scripts.

3.3 Current Status

CodeGeneratorGraph Enhancements

The underlying graph engine has been significantly extended to support retrieval-augmented generation (RAG), modular caching, and fine-grained regeneration control:

- **Retrieval-Augmented Generation (RAG)**

  A new RAGNode integrates a Qdrant-based vector store built from Crawlee’s Python API documentation. This node loads or restores the vector database and injects it into the graph via state["vectorial_db"], enabling downstream components to perform similarity searches over official reference material.

- **Crawlee-Specific Code Generation**

  The vanilla code generator has been replaced with a custom GenerateCodeNode that emits Crawlee Playwright crawler scripts. It uses a Jinja2-based template (DEFAULT_CRAWLEE_TEMPLATE) to inject only the scraping logic and schema-matching dictionary into a pre-defined scaffold, ensuring structural consistency across runs.

- **Multi-Stage Regeneration Loops**

  The code generation phase is structured around iterative “reasoning loops,” each targeting a different failure mode:

  - Syntax Loop: Attempts AST parsing and regenerates on parse errors
  - Execution Loop: Executes the candidate script in a subprocess; on error or timeout, queries Crawlee docs for context and regenerates accordingly
  - Validation Loop: Loads real output files (from storage/datasets/default) and checks them against the user-defined Pydantic schema. On

mismatch, relevant documentation snippets are retrieved and used to repair the code

  - (Semantic Loop: Currently scaffolded but inactive—intended for future use to enforce semantic consistency against earlier LLM outputs.)

- **Node-Level Caching**

  The .run() method has been re-architected to cache intermediate node outputs keyed by a SHA-256 hash of the prompt and target URL. On initial run, all nodes except the final generator are executed and serialized to disk. On subsequent runs, the cached state is rehydrated, including reloading the Qdrant vector DB, and only the final node is executed, dramatically reducing LLM overhead.

- **Configurable LLM and Retrieval Parameters**

  Runtime behavior is fully parameterized: retrieval settings (e.g., initial_k, execution_k, validation_k), iteration caps, LLM temperature, max tokens, embedding models, and seed values are all pulled from a central NODE_DEFAULTS dictionary or overridden via the user config.

Frontend Design and User Experience

The Django web interface is engineered to mirror the graph's backend logic, offering users a responsive, transparent interaction model:

- **Persistent Data Models**

  Projects, field specifications (which map directly to Pydantic schemas), API keys, and result logs are modeled via Django ORM and stored in SQLite for simplicity and portability.

- **Schema-Driven Project Creation**

  Project creation and editing pages feature dynamic JavaScript-driven forms that allow users to add, remove, and reorder schema fields client-side. These are submitted as parallel lists and saved atomically.

- **Authentication and API Key Management**

  All routes are protected by Django’s built-in @login_required decorator. Each user maintains a single API key object, which is scoped per user and required before code generation can begin.

- **Script Generation and Background Execution**

  Triggering “Generate Script” on a project spawns a background thread that executes the full CodeGeneratorGraph, writes logs incrementally to a ScrapingResult object, and flips the status to completed or failed based on runtime outcome.

- **Live Execution Status Interface**

  A real-time dashboard polls the server every second via AJAX. It displays a progress bar reflecting pipeline stage, a live log window with an auto-scroll toggle, and a read-only code preview panel for inspecting the final script. The progress bar will need future work as it currently does not reflect progress made from each node in the CodeGeneratorGraph.

- **Results and Error Handling**

  The results screen provides a centralized view of scraped data or diagnostic output. It automatically formats structured output for legibility and surfaces error messages inline.

One of the most persistent issues involved the validation and semantic reasoning loops within the GenerateCodeNode. These stages, designed to refine generated scripts by checking JSON schema conformity and semantic agreement with the model's earlier output, often stalled. In particular, the semantic comparison step was prone to false negatives, flagging acceptable scripts as incorrect due to minor phrasing mismatches. To improve resilience, we exposed key LLM hyperparameters — such as temperature, top_p, max_tokens, and seed — on a per-node basis. This allowed us to fine-tune the model's behavior during regeneration. For example, increasing temperature helped the model explore alternative code paths when stuck, while decreasing top_p made outputs more deterministic and aligned with prior results. Furthermore, commenting out the semantic and validation checks altogether helped improve reliability, especially when data extracted using pagination differed from each page. Thus, these checks will need to be improved in the future.

Another challenge centered around retrieval-augmented generation (RAG) over Crawlee documentation. Early attempts to vectorize a local clone of the Crawlee GitHub repository produced noisy results, as the repo was cluttered with tutorials, partial examples, and legacy content that diluted retrieval accuracy. The solution was to pivot to the live Docusaurus site at https://crawlee.dev/python/, which organizes API references into structured pages. By filtering specifically for URLs matching the /python/api/ path, we ensured that similarity searches returned concise, relevant function and class signatures that were well-suited for error-driven code regeneration.

Token usage emerged as a bottleneck when dealing with JavaScript-heavy pages. Dynamically rendered content required expensive full-page captures and large chunk sizes to preserve context, which inflated both cost and latency. While this remains an area for future optimization, several mitigations are planned: introducing smarter HTML chunking strategies (e.g., extracting only visible or above-the-fold content), caching raw HTML responses between runs, and limiting unnecessary re-fetches by integrating lightweight pre-rendering steps.

Model selection introduced a trade-off between throughput and quality. More capable models like GPT-4o could usually generate valid scripts in a single pass but were limited by rate constraints and cost per token. In contrast, faster models like GPT-4o-mini required more iterations to converge but could handle bulk workloads. In the end, we decided to use GPT-4o-mini due to its higher token per minute limit, 128k TPM, compared to GPT-4o (30k TPM).

3.4 Results on Test Cases

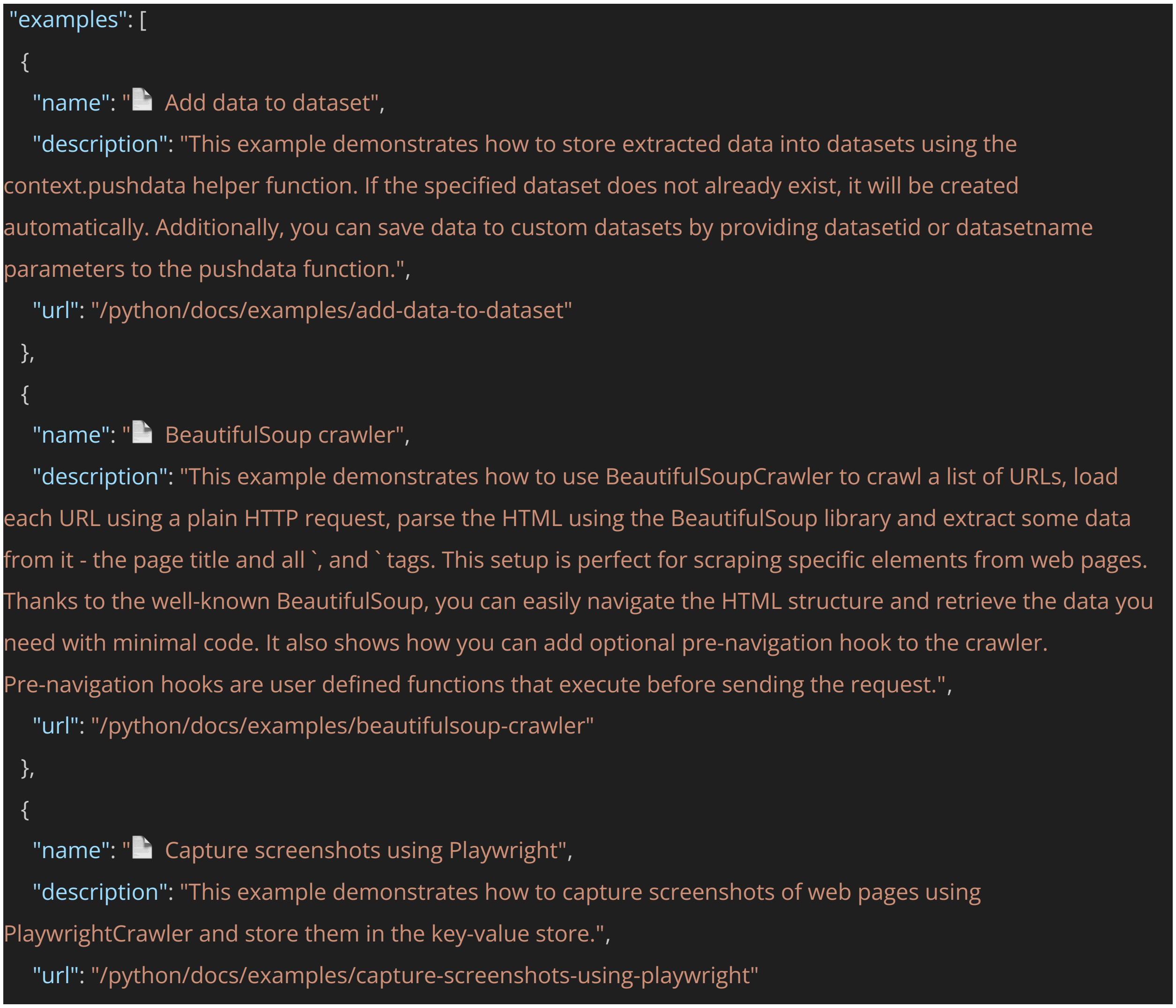
A simple working example is scraping Crawlee documentation. Create a project and enter the inputs as follows:

- website url = https://crawlee.dev/python/docs/examples
- Prompt = "Please give me the name, description, and URL for examples of crawlee applications that involve pagination"
- fields:
  - name: str = Name of the example
  - description: str = One sentence description of the example
  - url: str = URL of the example

Clicking “Generate Scraping Script” would begin the script generation process, showing you the log outputs as ScrapeGraph generates and improves a script. The output was written to extract_data.py as follows:

```
import asyncio

from crawlee.crawlers import PlaywrightCrawler, PlaywrightCrawlingContext
from bs4 import BeautifulSoup

async def main() -> None:
    crawler = PlaywrightCrawler(
    max_requests_per_crawl=500,
    )

    @crawler.router.default_handler
    async def request_handler(context: PlaywrightCrawlingContext) -> None:
    context.log.info(f'Processing {context.request.url}')

    page_content = await context.page.content()
    soup = BeautifulSoup(page_content, 'html.parser')
    examples = []

    for article in soup.select('article.col.col--6.margin-bottom--lg'):
    url = article.find('a')['href']
    name = article.find('h2').get_text(strip=True)
    description = article.find('p').get_text(strip=True)
    if 'pagination' in description.lower():
    examples.append({
        'name': name,
        'description': description,
        'url': url
    })

    data = {'examples': examples}

    await context.push_data(data)

    await crawler.run(['https://crawlee.dev/python/docs/examples'])


if __name__ == '__main__':
    asyncio.run(main())
```

The data scraped is saved into the following path storage/datasets, that houses the JSON file containing the scraped data. Here is a snippet of the data scraped:



```
"examples": [
  {
    "name": "📄 Add data to dataset",
    "description": "This example demonstrates how to store extracted data into datasets using the context.pushdata helper function. If the specified dataset does not already exist, it will be created automatically. Additionally, you can save data to custom datasets by providing datasetid or datasetname parameters to the pushdata function.",
    "url": "/python/docs/examples/add-data-to-dataset"
  },
  {
    "name": "📄 BeautifulSoup crawler",
    "description": "This example demonstrates how to use BeautifulSoupCrawler to crawl a list of URLs, load each URL using a plain HTTP request, parse the HTML using the BeautifulSoup library and extract some data from it - the page title and all `, and ` tags. This setup is perfect for scraping specific elements from web pages. Thanks to the well-known BeautifulSoup, you can easily navigate the HTML structure and retrieve the data you need with minimal code. It also shows how you can add optional pre-navigation hook to the crawler. Pre-navigation hooks are user defined functions that execute before sending the request.",
    "url": "/python/docs/examples/beautifulsoup-crawler"
  },
  {
    "name": "📄 Capture screenshots using Playwright",
    "description": "This example demonstrates how to capture screenshots of web pages using PlaywrightCrawler and store them in the key-value store.",
    "url": "/python/docs/examples/capture-screenshots-using-playwright"
```

One test case involved a two-page static pagination demo hosted at webscraper.io. The LLM occasionally generates logic for only the first page. The underlying issue stemmed from a lack of consistent signal in the DOM about the presence of a “next page” link, causing the LLM to halt prematurely. We attempted a schema-based workaround that flagged runs with “too few items” to trigger a regeneration cycle, but the validator sometimes failed due to small discrepancies (e.g., off-by-one errors). Ultimately, re-running the graph with a fresh prompt proved more effective than incremental correction.

Next, we tested a JavaScript-powered dropdown menu on OpenSecrets (https://www.opensecrets.org/elections-overview), which cycles through election years from 1990 to 2024 and displays corresponding party tables. The pipeline was only partially successful, as very few managed to iterate through all available cycles. Common failure modes included the LLM hardcoding a single year or omitting the loop entirely. Moreover, when a script did succeed in gathering data across cycles, our validator sometimes misclassified it as invalid due to missing expected sections, likely because some election cycles lacked presidential tables. Again, the validation layer surfaced useful signals, but full regeneration remained the most reliable recovery path.

Furthermore, we targeted Zillow’s homepage — a highly dynamic, bot-sensitive site — where we tasked the LLM with finding the top ten most expensive properties in Boston. This test failed entirely across all runs. The generated scripts did not reach the appropriate execution paths, and all requests were met with HTTP 403 errors. The RAG fallback, which attempted to repair failed code via documentation lookup, was ineffective against such aggressive anti-bot measures. This underscored the need for true browser-level automation, session management, and potentially paid API access when targeting hardened production endpoints.

Across all tests, recurring patterns emerged. Pagination and multi-step workflows exposed the brittleness of schema and semantic validation, where error recovery required complete regeneration rather than fine-grained correction, as the validation and semantic checks did not provide much use. JavaScript-heavy pages burned through token limits without guaranteeing improved fidelity, highlighting the need for smarter chunking, caching, and rendering strategies. While retrieval-augmented generation improved resiliency during code generation, it proved insufficient in cases where real-world anti-scraping defenses blocked progress at the network level.

The dedicated validation and semantic reasoning loops are meant to improve result accuracy, but these mechanisms often fall short in practice. The JSON schema validation logic, anchored in the validate_dict() method, only reports the first schema violation it encounters, offering the LLM limited visibility into broader structural issues. Compounding this, Pydantic's default schemas are permissive by design, frequently allowing extra fields or missing attributes unless explicitly tightened via additionalProperties: false and a complete required list. As a result, many malformed outputs pass unnoticed, and the validation loop fails to trigger. Even when the loop activates, it operates on coarse feedback, making isolated LLM patches unreliable, especially for nested or multi-record payloads. Similarly, the semantic comparison layer relies on an overly strict are_content_equal() check, which performs a normalized but exact match of dictionary contents, leading to false negatives due to trivial differences in value formatting, key order, or whitespace. This brittleness often causes the semantic loop to stall or misfire, and in the current implementation, the loop is even disabled by default due to its low return on cost. Ultimately, both validation and semantic checks remain too fragile and token-expensive to serve as dependable repair mechanisms, underscoring the need for more expressive schema enforcement, full-batch error reporting, fuzzy comparators, and fallback strategies that regenerate entire scripts rather than piecemeal patches.

3.5 Future Improvements

After the code is generated, there are two placeholder buttons to either download the container or run it. In the future, these two features can actually be implemented. Clicking “Download Container” would download the container with the generated script and requirements.txt inside, and the user would then be able to run the container locally. Clicking “Run Container” would upload the container to a public registry, and we could then run it for the user as a service. We can also improve more quality of life features in the UI, such as a button to copy the generated Python script.

Another future improvement would be scaling to more complex websites, which is currently difficult because we start running out of tokens. We could try to make the prompt generation process more concise.

Our current process for generating the script is also not the most secure since the web server runs as root, so a way to solve this would be to put the entire project into a container as well.

And, because our target usership is low to non-technical, we want the scraper to be as seamless and easy to use as possible, so we are considering the following implementations: a prompt guide, where users are instructed on how to best write their scraper prompt, a list of preset questions about the website the user wants to scrape, an interactive preview that shows sample data during configuration, a feature to automatically run scraping jobs at regular intervals (daily, weekly) to monitor changes in data sources for ongoing investigations, and export options to formats journalists commonly use (e.g., CSV).

4. Reflections and Journalism Tools

4.1 Project Scoping and Timeline Management

While we worked on the automated scraper tool, we came away with several important lessons regarding the landscape of journalism tools, how we can improve upon it, and how we can contribute to it with our scraper.

Underscoring this point: The most significant challenge we faced in delivering a working tool was defining the scope of our project too late in the semester. By the time we focused our vision on the automated scraper tool, the semester was about halfway through.

Earlier project scoping would have allowed us to:

- Develop a more robust testing framework

- Identify and solve issues such as pagination earlier on
- Create additional features beyond the core functionality
- Potentially refine the user interface based on feedback iterations

An ideal approach would dedicate the first three weeks of the semester to exploration and concept development, followed by a firm decision point where we commit to project parameters. This would have left approximately ten weeks for implementation, testing, and refinement.

For future directed studies, we recommend implementing structured decision gates at specific points in the semester, and recommend students follow an agile development methodology for their project.

4.2 Technical Skills Development

Despite our timeline challenges, the directed study facilitated significant growth in our technical capabilities, particularly in:

4.2.1 Programming proficiency:

Each team member, regardless of varying technical proficiencies, enhanced their technical skills through practical application. The necessity of writing functional code for our scraper tool pushed us beyond theoretical understanding to implementation-level knowledge.

4.2.2 Web scraping expertise:

Through our project, we gained hands-on experience with both static and dynamic web scraping challenges. We learned to identify when simple HTML parsing is sufficient versus when browser automation is necessary, and developed strategies for handling common obstacles like pagination, AJAX loading, and varying page structures.

4.2.3 Security and privacy literacy:

Our exploration of privacy and security enhanced our understanding of responsible data collection and web scraping. We learned to implement measures that respect site terms of service while protecting both the scraper user and potentially sensitive sources.

4.2.4 Containerization knowledge:

The project introduced us to Docker and containerization principles, providing valuable experience with modern deployment methods that ensure reproducibility and ease of use.

4.3 Investigative Journalism Applications

The skills developed and knowledge gained during our directed study are vital to understanding and working in today's data journalism landscape.

Our automated scraper tool addresses a critical need in newsrooms where technical expertise varies widely. At the local level, there is often a lack of technical proficiency, but these small newsrooms are often the ones that could benefit the most from easier data acquisition, which translates into more impactful stories and a wider breadth of topic coverage. By democratizing access to web scraping capabilities, we aim to empower journalists who may lack programming knowledge but need to monitor or collect data from websites for their reporting beats. Potential applications include:

Political Beats

- **City Council Meeting Records**: Scrape council agendas, minutes, and voting records to track decisions at the state and municipal level
- **Municipal Contract Awards**: Monitor local government spending patterns and identify potential conflicts of interest
- **Local Campaign Finance Data**: Track political donations to local candidates to reveal influence patterns

- **Property Records**: Monitor real estate transactions to identify development trends or potential red flags

Health and Safety Beats

- **Restaurant Health Inspection Reports**: Scrape restaurant inspection data to analyze local restaurant safety scores
- **Local Crime:** Scrape police department data portals for incident reports and response times
- **Building Code Violations**: Track housing code violations to identify systemic problematic landlords

Economy and Consumer Beats

- **Local Pricing Disparities**: Compare prices for essential services (internet, utilities) across different neighborhoods, and incorporate census data to measure socioeconomic impact
- **Rental Market Analysis**: Track apartment listings to document housing affordability trends
- **Job Market Monitoring**: Analyze local job listings for wage trends and potential employment scams

- **Public Transit Performance**: Track on-time performance and service changes in local transit systems, and monitor how public transit is serving various local communities

Education Beats

- **School Performance Data**: Track local school metrics, including test scores and funding
- **School Board Decisions**: Monitor policy changes and budget allocations
- **College Affordability**: Compare costs and financial aid at local institutions

These applications extend beyond individual reporting projects to support data-driven newsroom operations more broadly. They give small and hyperlocal newsrooms the potential to establish consistently updated data-driven projects that otherwise would only be possible with the resources, funding, and technical prowess of a larger news organization.

4.4 Justice Media Co-Lab Integration

Our directed study curriculum aligns with the mission of BU's Justice Media Co-Lab (JMCL), which trains "a new generation of computational investigative journalists and news technologists who are equipped to leverage the power of computing and data sciences to advance justice and transparency."

This course, in the future, could serve as a valuable component within the JMCL's educational framework by providing journalism students with the technical capabilities needed for computational journalism in justice-focused contexts. In this class, those students will learn how to better communicate their ideas and concepts to technical teammates, and vice versa. Students will also develop tools in Data Journalism II that can be utilized in JMCL and by JMCL’s news partners.

Students participating in both Data Journalism II and JMCL programs would benefit from complementary skill development. While JMCL focuses on interdisciplinary collaboration between journalism and computing/data science students to report a story, our course provides a foundational understanding of programs, tools, and protocols common in advanced data journalism, and hands-on experience with those programs, tools, and protocols that are essential for the technical implementation of investigative projects. For journalism students, this means gaining technical self-sufficiency; for computing students, it means understanding journalistic applications of their skills.

The tools and methodologies we've explored could enhance JMCL projects with local media partners like The Bay State Banner, WGBH, Commonwealth Beacon, and others.

5. Center for Open Tools for Journalism (COTJ)

Building on the insights gained through our semester-long project, we have ambitions to establish the Center for Open Tools for Journalism (COTJ). This center would address a critical issue of journalism tool maintenance, development, user support, and organization, as well as provide students with real-world software experience. There is potential for COTJ to be integrated into JMCL as well, and it could be a part of an eventual 4+1 data journalism program offered by COM to rising senior journalism students.

5.1 Mission and Structure

The COTJ would support the development, maintenance, and adoption of open-source tools for data-driven journalism. Its core mission is to ensure that essential civic technologies remain usable, supported, and accessible to journalists of all technical skill levels. The Center would operate on five fundamental values:

- **Public Interest First**: Supporting journalism as a public good

- **Open and Transparent**: Making all software, documentation, and governance public
- **Sustainable Software**: Providing ongoing maintenance, not just minimum viable products
- **Inclusive Design**: Creating tools usable by non-technical journalists
- **Community-Driven**: Engaging journalists, students, and developers as co-creators

More on COTJ can be found here.

6. Conclusion

This directed study has been a journey of exploration at the intersection of data journalism and technical development. Our goal for the automated scraper tool is not just a functional product but a tool that can help democratize hard-hitting, data-driven, investigative journalism. Through the challenges we faced with the ScrapeGraph architecture and user interface design, we've gained valuable insights into both the technical implementation of data collection tools and their practical application in investigative reporting.

While our timeline constraints prevented us from implementing all desired features, the core functionality demonstrates the potential for AI-assisted data collection to level the playing field for local newsrooms. The skills we've developed, from web scraping expertise to containerization knowledge to the process of learning how an open-source project works, like what Crosby did with ScrapeGraph — even to learning how to approach the question of what language to use for a project — these have have equipped us with a deeper understanding of software engineering and computational journalism that will serve us well in our future careers.

We are tremendously grateful for Professor Langdon White's guidance, stories, box of random cables, milieu of stickers, burner laptops, and expertise throughout this semester. His approach to this course allowed us to explore topics aligned with our interests while maintaining focus on developing practical skills for modern journalism.

7. Appendices

7.1 Project Code and Documentation

Github: https://github.com/cros-nash/ds490/tree/main

7.1.1. Installation & Setup

i. Prerequisites:

   1. Python 3.10+
   2. Docker (for containerized execution)
   3. OpenAI API key (for LLM processing)

ii. Clone the Repository:

   1. git clone https://github.com/cros-nash/ds490.git

iii. Create and activate a virtual environment + Install Dependencies

   1. python -m venv mvenv
   2. source myenv/bin/activate
   3. cd ds490
   4. pip install -e .
   5. playwright install

iv. Configure Environment Variables

   1. Create a .env file and add your OpenAI API key:
      - OPENAI_API_KEY=your_api_key_here

v. Create database:

   1. python frontend/manage.py makemigrations
   2. python frontend/manage.py migrate

vi. Create a superuser (admin account):

1. python frontend/manage.py createsuperuser
2. Follow the prompts to set username, email, and password.

7.2 Resources and References

7.2.1 Key readings and tutorials used

1. Open Source
    a. General overview of open source
    b. What does Creative Commons do?
    c. Business Source License
    d. The Legal Side of Open Source
2. GNU & Linux
    a. GNU/Linux Naming Controversy
    b. XZ Backdoor
        i. In early 2024, a carefully planted backdoor was found in XZ Utils, a widely used Linux compression tool, after a pseudonymous contributor spent years building trust to insert malicious code that could allow remote access, exposing major risks in open-source software supply chains.

3. Journalism Conferences
    a. USC Inaugural Open Source Journalism Conference (February 7-8)
    b. NICAR (National Institute of Computer-Assisted Reporting) Conference (March 6-9)
        i. 50 Free Datasets in 50 Minutes
4. Command Line Tools
    a. Repository of all the tools
    b. Introduction to shell
    c. Command line tutorial & practices
5. GitHub Repositories
    a. Semantra
        i. Multipurpose tool for semantically searching documents. Queries by meaning rather than just by matching text.
    b. meaningfully
        i. Semantic search tool for text data in spreadsheets; works best for semi-structured data, where you have thousands of instances of a type and want to find certain instances.
    c. Crawl4AI
        i. Open source tool that generates clean markdown; great for direct ingestion into LLMs. Also performs structured extraction:

parses repeated patterns with CSS, XPath, or LLM-based extraction.

   d. NamUs Scraper

      i. Python scraper for collecting metadata and files for Missing-, Unidentified-, and Unclaimed-person cases from the National Missing and Unidentified Persons System (NamUs) organization.

   e. crawlee

      i. Web scraping library for JavaScript and Python that handles blocking, crawling, proxies, and browsers for the user.

6. Project Proposal Research Phase

   a. Missing People in America (MPIA)

   b. ICEWatch: ICE Raids Tactics Map - Immigration Defense Project

   c. ICEWatch: Project of the Immigrant Defense Project and the Center for Constitutional Rights

   d. Civic Tech Hackathon Devpost

   e. The National Missing and Unidentified Persons System (NamUs)

7. Tools Referenced

   a. Rowboat

      i. Tool for understanding large tabular datasets.

   b. Fakespot

   c. IntelTechniques

      i. Search many online databases including LinkedIn, Facebook, and Twitter from one page.

   d. CUNY Craig Newmark Graduate School of Journalism: Fact Checking & Verification for Reporting

   e. Whonix

      i. Anonymous operating system that runs like an app and routes all Internet traffic through the Tor anonymity network. It offers privacy protection and anonymity online.

   f. r/DeGoogle

   g. Chatbox AI

      i. AI client application and smart assistant.

8. Project Development References

   a. ScrapeGraphAI

   b. WASM Scrapers

   c. Web Scraping with Elixir

9. Dark Web

   a. Dark web tour: A “sneak peak” into the dark web

   b. Beginner’s tour of the Dark Web

7.2.2 Tools and libraries leveraged for project

- Scrapegraph
- Django
- Crawlee
- OpenAI API
- Playwright

7.3 Slides

A few slideshows we created this semester:

- What is Open Source?
- Google Dorking
- The Dark Web