Optimized Web Crawling: Enhancing Navigation Efficiency with Crawlee

Bachelor Semester Project S2 (Academic Year 2024/25), University of Luxembourg

1 Introduction

In modern web-based systems, the ability to collect and analyze large pieces of data from different dynamic websites is quite important. However, this leads to various challenges related with content duplication. Many web pages contain different small components - such as timestamps, clocks, and advertisements - that change frequently but do not alter the underlying meaning of the page. Such elements are considered as "noisy", since they are semantically irrelevant.

Conventional web crawling approaches usually deal with these problems by utilizing different fingerprinting techniques, including hashing and similarity metrics, such as cosine distance. Although such methods are highly effective in detecting complete duplicates, they typically struggle in identifying semantic similarities. As a result, these insignificant changes on a web page can cause misclassification, and subsequently the content is treated as completely new. This results in redundant data processing and inefficiency in data collection.

To address these limitations, our project introduces a comprehensive and semantically aware solution by extending the comparison process from a basic content comparison to a semantic-level understanding. At the center of this approach is the integration of a Large Language Model (LLM), which is capable of advanced reasoning. For this purpose, the OpenAI's GPT-4.0 LLM [14] was chosen. This decision can reliably extend our project to identify content de-duplication, even with small semantically insignificant changes.

2 Background

2.1 Web Crawlers

Web crawlers are designed to automate the process of systematically visiting various web pages and collecting information from them. The main goal of web crawlers is to discover new pages and observe changes in previously discovered ones, as well as to index web content and enable large-scale data extraction. A typical web crawler operates by starting from a set of initial web pages, called seed URLs, extracting their content, parsing it for additional hyperlinks, and recursively extracting, retrieving, and storing relevant data from the following URLs. Beyond web archiving, web crawlers can be used for more sophisticated purposes such as test automation; for example, the Metamorphic Security Testing [1] toolset of the University of Luxembourg relies on web crawling to automatically assess the security properties of web systems.

The basic web crawler architecture consists of some essential components that work together. At the center of this system is the **request queue**, which maintains a record of all URLs gathered in the process of crawling that must be visited. This is typically done using priority queue strategies, such as breadth-first search and depth-first search. The second crucial element is a **HTTP client**, or **fetcher**, which retrieves the HTML of a web page. The retrieved content is then passed through a **parser** that is used to extract useful and meaningful data, including text, hyperlinks, images, and metadata. The extracted links go through a filtering process before they are added to the URL queue to prevent recrawling of already visited sites and the formation of infinite loops. In addition, a set of content de-duplication algorithms are used to mark and exclude similar or identical web pages. Finally, a **storage system**, most often in the form of a database or an external file, is used to maintain the extracted content for future analysis [17].

Even though web crawlers are highly effective and useful tools, they face several challenges. These consist of duplicate content handling (situations where identical content is duplicated under different URLs), detecting and avoiding crawler traps (pages specifically constructed to cause an infinite loop or "noisy" information), and handling pages that generate dynamic content on user interactions.

Popular web crawlers capable of processing pages with dynamic content are Brozzler, Puppeteer, Crawlee, and Crawljax. A recent empirical assessment reported in a master thesis of the University of Luxembourg has shown that the crawler capable of visiting the largest set of pages belonging to modern Web applications (namely Jenkins, Joomla, and PrestaShop) is Crawlee.

In this project, the Crawlee framework [2] is used for the development of the web crawler. Crawlee is a modern open source crawling and scraping library built based on Cheerio [4], Puppeteer [3], and Playwright [5], supporting headless browser automation along with many advanced features such as request queuing, page handling, and session management. It has been used because of its flexibility and strong support for JavaScript-based websites that are quite common on the Internet today.

2.2 Large Language Models

Large language models (LLMs) are deep learning models that are trained on large sets of data to understand and generate human-based language. Such models can be generalized to carry out an extremely large number of natural language tasks without task-specific training [10].

LLMs have demonstrated remarkable success in various areas such as text processing and generation, question answering, coding, and analysis. When it comes to web crawling, LLMs open up new possibilities, especially in the context of semantic content analysis. They can be used to compare the meaningful part of a web page rather than relying on structural similarities, making them quite valuable tools for detecting content de-duplication that may appear in different forms or layouts [12].

For this project, OpenAI's GPT-4.0 LLM [14] was chosen via its API to determine the content similarity between two already crawled web pages. This approach enables more accurate decision making throughout the crawling process, especially while detecting the web page content as similar or completely identical, which other

algorithms such as hashing or string comparison might miss. In addition, the decision to use OpenAI's GPT-4.0 LLM was made based on its ease of integration and the possibility to be extended for the purpose of the project.

3 A benchmark for web crawling

3.1 Principles

To evaluate the effectiveness of web crawlers, it is essential to define a set of principles that should be followed to assess them. Some key principles include the crawler's ability to avoid similar or already visited web pages. Another key principle is the crawler's ability to handle JavaScript-based dynamic web pages that rely mainly on reload or user interactions. Additionally, Web crawlers must be able to identify and avoid situations that may work as traps that mislead them. Such traps may consist of pages that overwhelm the crawler by continuously generating new meaningless (or 'noisy') content or lead to infinite loops.

Another key principle is efficiency in data collection and storage for future analysis. Based on previous work of the SVV group [1], the crawled content must be stored both as HTML for further structural analysis and as plain text for meaningful content comparison. Finally, the extracted data must be properly structured, allowing for integration with tools, such as Large Language Models (LLMs) to support the content similarity detection between different pages. These principles form the foundation for the benchmark in this project and ensure that the crawler is capable to reliably operate on different websites.

3.2 Benchmark Design

The architecture of the system is built with flexibility to enable the integration of new features. At the center of the system is a web environment - designed and implemented as a "trap website" - carefully constructed to simulate real-world challenges that most web crawlers tend to face. The website is developed using modern front-end library, such as **React** - a widely used JavaScript library for building interactive web applications [15] - and utility-first framework, such as **Tailwind CSS** - used for styling, and simplifying the creation of responsive designs [16]. This approach ensures a safe and reliable environment. The "trap website" consists of four distinct web pages, each of them intentionally created to examine particular aspects of the crawler's behavior, such as duplicate content detection, user interaction handling, and content diversity.

The first trap is a **static clock page**, where the displayed time updates to the current time only after the initial page rendering. This tests the crawler's capability of detecting JavaScript-based content that is not persistent after each page rendering. The second trap consists of **duplicate content, rendered under different URLs**. This is helpful when examining whether the crawler can identify semantically identical content located under different URLs. In addition, this simulates real-world scenarios where the same content appears under different routes, which is useful to assess if the crawler uses content-based comparison - rather than relying on URL uniqueness - to detect redundant content. It also serves as a test case for the use of Large Language Models (LLMs) to assess semantic similarity and improve crawling efficiency.

The third trap is a **parameter-based page** that renders different content on each reload based on one of three predefined parameters. The parameter is chosen randomly after each page render and each parameter leads to a different predefined content. Even though the set of possible page outputs is finite, the randomness can lead to uncertainty in what should be considered as a correct output. This is designed to test crawler's ability to identify if the randomly rendered pages represent a similar information or not. Additionally, this approach challenges different duplicate detection strategies and caching techniques, since the predefined content is limited, but it appears dynamically from the crawler's perspective.

The fourth trap is a **same URL that renders one of three predefined contents**. The rendered content is chosen by a mechanism for randomness on each page visit. The possible page outputs are either a static page with text, a submission form that sends user input to the back-end, or a dynamic interface that generates new content after each button click. This trap is designed to simulate web interactivity and content variability, challenging the crawler's ability to handle dynamic behavior under a consistent URL.

3.3 Back-end server

The **back-end server** is a significant part of the project and it fits two purposes: support HTTP requests as a normal website and collect some statistics about the internal behavior of Web crawlers, to support a better understanding of their execution flow.

The back-end server is implemented using **Node.js** - a cross-platform JavaScript runtime environment that enables the development of scalable network applications [13] - and **Express.js** - a minimal and flexible Node.js web application framework that provides a robust set of features for building web and mobile applications [9] - and plays a vital role in the overall architecture and maintenance of the project. It is designed to receive, organize, and analyze the data gathered from the crawler. To enhance functionality, the OpenAI's GPT-4.0 LLM is integrated, allowing for advanced content comparison and analysis.

The back-end provides two main endpoints: one for the plain text (/crawler-data) and the other for the complete HTML content (/crawler-data-as-html). Whenever the crawler sends data to the server, it contains a session ID, which is used to organize the data from each crawl. This approach guarantees that the data is stored separately during multiple crawler runs. For each session, two separate log files are generated — one for plain text and one for HTML — and stored in dedicated directories (*/logs* and */logs_as_HTML*, respectively). This structure supports efficient session tracking and simplifies later analysis of crawled data.

To avoid the storage of duplicate content, a mechanism for duplicate content detection is implemented - based on the integration of OpenAI's GPT-4.0 LLM - in the back-end to compare each incoming page with the previously visited ones. If the page content, extracted from the crawler seems to be completely different, it is logged in the file. Otherwise, the content is considered similar and, therefore, ignored. The logic behind this comparison approach will be described in detail in Section 4, which gives a detailed explanation of the integration of OpenAI's GPT-4.0 LLM to assess page similarity.

In addition, the back-end provides an example use case for the submission form endpoint: (/submit-report). This endpoint receives

user input from the submission form via a "POST" request and stores the data in the memory, which simulates a real-world application scenario and can be a good approach for further crawler improvements and test cases.

Other useful features, such as the **CORS** middleware, enable smooth operation in a local development environment. **CORS** is a middleware that allows or restricts resources on a web server to be requested from another domain, helping to control cross-origin requests and enhance web application security; for the purpose of this project, it is configured to accept requests only from the front-end [11]. In addition, the **body-parser** middleware is included to parse incoming request bodies before they reach the route handlers, making the data accessible via *req.body* [8]. Lastly, environment variables such as API keys or other sensitive information are managed securely and flexibly using the **dotenv** package [7].

Overall, the back-end server is intentionally designed to provide extensibility. It is required to ensure a controlled environment, enabling a comprehensive evaluation of the crawler's behavior.

4 Crawling with LLM

The second objective of the project is to design a web crawler capable of leveraging LLMs to control crawling decisions (hereafter, *LLM-based crawler*); further, the crawler shall also allow a comparison of the proposed LLM-based solution with the state-of-the-art crawling practice (hereafter, *basic crawler*). For this reason, a crawler that integrates basic crawling features and is extensible with LLM features has been designed. These two characteristics are described in the following subsections.

4.1 Basic crawler design

The basic crawler is implemented using the **Crawlee framework**, utilizing its PuppeteerCrawler to enable headless browser automation and dynamic content handling [2][3]. Crawlee enables the crawler to interact with dynamic content, simulate user behavior, and manage a request queue of pages that must be visited throughout the crawling process. The crawler's logic begins by initiating a session with a unique timestamp-based ID to track each crawling process independently. For every visited page in the request queue, it waits for the *<body>* element to be rendered, ensuring that the page is fully loaded before processing. This approach is quite useful, when preventing the crawler from being impacted by a heavy JavaScript-based content that comes late after the page is rendered.

A key feature of Web crawlers is content de-duplication; following common developers' practice, it was designed to rely on a hash-based content comparison approach. This implementation aimed to avoid duplicate content reprocessing using a unique fingerprint to store each page's meaningful content. Each time the crawler extracted the plain text from the *<body>* tag of a web page, it was passed through the MD5 hashing algorithm based on the Node.js's **Crypto** library [6]. The hashing mechanism represented the content as a 128-bit hexadecimal string that was stored in a Set object in the memory. Additionally, each newly extracted page content was compared - if its hash string was already in the Set, the page was classified as a duplicate and, therefore, excluded from fetching to the back-end; otherwise, it was added to the Set.

This approach offered certain benefits. The MD5 hashing algorithm is quick and inexpensive, which makes it suitable for a fast and safe page processing environment. In addition, the use of a Set data structure ensures a scalable and efficient duplicate detection. However, this mechanism had its limitations and quickly became insufficient when tested in real-world scenarios. Web pages that were identical often generated completely distinct hashes due to some "noisy" elements, such as timestamps, clocks, or whitespaces. These differences caused semantically equivalent pages to be treated as entirely unique, reducing the effectiveness of the crawler. Furthermore, the mechanism lacked the ability to assess the actual meaning of the content. These limitations became quite noticeable when pages with minor, meaningless changes were considered completely new.

To ensure a better crawler functioning, a special logic was implemented for the */Rules* page, which contains the clock element that updates after each reload of the page. First, the crawler extracts the initial value of the clock and then waits for a change in the corresponding DOM element. Afterwards, the crawler extracts both the plain text and the HTML content of the page. For all the other pages, the content extraction is performed without any additional logic. This is used to demonstrate the crawler's ability to detect post-render DOM content changes. Even though the implementation is not designed to be useful for various purposes and websites, it serves as a valuable test case to evaluate the dynamic content handling of the crawler.

The collected data is then sent to the back-end server, where it is stored as JSON and HTML for future analysis. Two separate API calls are used: one for the plain text and another for the HTML content of the page. This mechanism provides flexibility, allowing both structural and semantic evaluation of the pages.

Additionally, the crawler uses the Crawlee's *enqueueLinks* feature that automatically discovers, extracts and queues all anchor tags *<a>* to the request queue.

The system can also be further extended to include other even more advanced functionalities such as screenshot capture or the simulation of real user interactions through the use of Large Language Models (LLMs).

The overall workflow of this implementation—including page rendering, content hashing, and API communication—is illustrated in Figure 1 , which presents the crawler's internal state flow during execution.

4.2 LLM-based crawler design

To address the limitations of a basic Web crawler, an LLM-based crawler was designed. It implements a new, more reliable, and semantically aware approach, not relying only on text content matching. The crawler's implementation leverages a Large Language Model (LLM), namely OpenAI's GPT-4.0 (before selecting this LLM a preliminary study on its capability to compare the semantic similarities between different web pages has been conducted).

The implementation of the LLM-based crawler represents a direct extension of the basic crawler. The overall architecture remains unchanged: the crawler is based on the PuppeteerCrawler tool provided by the Crawlee framework. It is utilized to visit and extract the plain text and the HTML content of each page, as long as to

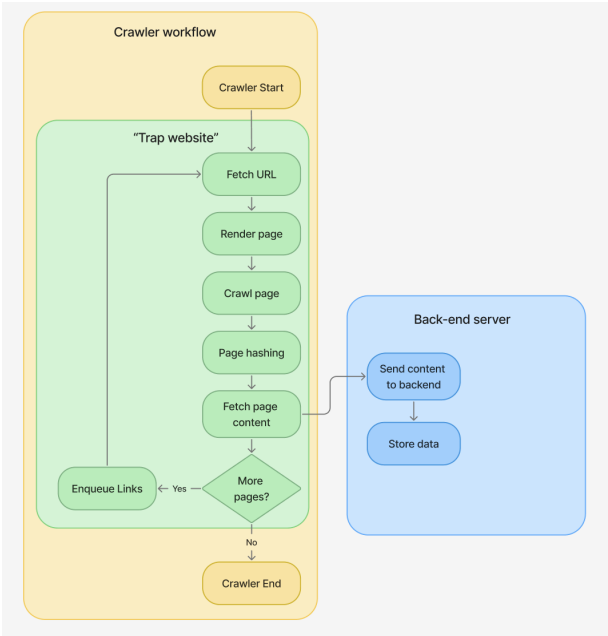

Figure 1: Workflow of the basic crawler using hash-based approach

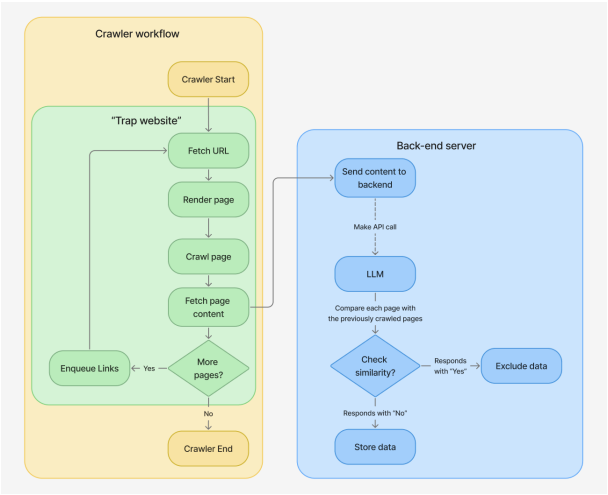

Figure 2: Workflow of the LLM-based crawler integrated with OpenAI's GPT-4.0 API for semantic de-duplication.

handle dynamic page elements when necessary. However, the key difference is in the mechanism used to determine the content similarity between various web pages. The basic crawler relied on the MD-5 hash-comparison implementation, whereas the updated version leverages the capabilities of the selected LLM, to improve the accuracy of similarity detection and the decision-making process (see Figure 2 for an overview of the updated workflow).

The LLM integration is implemented entirely on the back-end server. After the crawler extracts the content of each web page, both as plain text and as HTML, the data is sent to the back-end via a HTTP *POST* request with a structured JSON payload that includes the URL, the plain text, the HTML, and a session ID for further semantic de-duplication analysis. The logic for these analysis is built on top of a prompt-based interaction with the GPT-4.0 LLM, provided through the OpenAI's public API. This API-based integration allows the system to dynamically query the LLM with relevant page content and receive real-time responses to guide content filtering decisions.

The prompt is carefully designed, instructing the LLM to act as an AI that measures a content similarity between web pages. It includes a short system-level instruction to define the model's role - "Hello, GPT-4. You are an AI that measures a content similarity between web pages." - and a user-level message - "You are an AI that compares two web pages to decide if their main textual content is semantically the same or largely redundant. Respond **only** with "Yes" (if content is essentially the same) or "No" (if content is meaningfully different). Page1:${content1}, Page 2: ${content2}" - that provides the textual content of both pages, explicitly asking the model to reply only with a binary answer: "Yes" if the content is semantically equivalent, or "No" if it provides a new content.

Upon receiving the response of the LLM, the back-end uses it as a criterion for storing the unique web pages. If the model returns "Yes", meaning that the current page is semantically similar to a previously visited page, the back-end marks the page as duplicate, and the content is excluded from storing or further analysis. This approach effectively prevents unnecessary data duplication. Otherwise, the LLM returns "No", indicating that the current content is meaningfully different. The content is then stored in two locations: an array that maintains a record of all unique web pages and their URLs, and into *log* files that store the crawled data for potential future analysis.

5 Empirical assessment

To validate the effectiveness of the proposed crawler architecture and the integration of a Large Language Model (LLM), namely the OpenAI's GPT-4.0 LLM, into the web crawling process, an empirical assessment with two phases was made. The assessment aimed to test both the feasibility of semantic similarity classification using a state-of-the-art LLM and the performance of the developed crawler, executed within a dynamic front-end environment. The evaluation was structured into two stages: (1) a manual feasibility study via the ChatGPT interface to assess the model's capability to detect semantic similarities, and (2) an automated execution of the LLM-based crawler to observe its performance and capabilities in a controlled environment.

5.1 Feasibility study with ChatGPT interface

Before integrating the LLM into the back-end server, a feasibility study was made using the ChatGPT interface. The primary objective of the study was to manually evaluate the OpenAI's GPT-4.0 LLM capabilities; whether it could reliably differentiate semantic similarities between various web pages, based on understanding the actual meaning of the content. This assessment aimed to verify the model's ability to perform accurate semantic reasoning, by excluding minor formatting differences or "noisy" elements.

For this purpose, the plain text of any two pages from the custom-designed website was selected. These pages were carefully chosen to represent cases where the content was either semantically identical or completely unique. Precisely, a total of 12 pages has been considered; 1 page pair was semantically identical, 5 were different. The plain text of each page was extracted directly from the page's HTML and used without any changes to simulate the input that the crawler would later submit to the back-end server.

To perform the assessment, the plain text of each pair of pages was submitted to ChatGPT though a well structured prompt, designed to detect semantic similarities. The prompt followed a consistent structure across all test cases. It followed this format:

> I am performing an automated web crawling and I need to decide if the page that I am currently visiting was already visited. I am going to provide you with the content of the pages already visited and the current page. Decide if the current page is semantically similar to an already visited one. Respond only with "Yes" or "No".
> **Visited page:** [Page content]
> **Current page:** [Page content]

The model's consistently accurate responses validated its effectiveness in evaluating semantic similarities. It was able to reliably determine whether two web pages are semantically identical or represent an entirely distinct content.

This manual testing successfully demonstrated that GPT-4.0 LLM was capable of performing semantic reasoning, which is required for advanced crawler development. As a result, OpenAI's GPT-4.0 LLM was integrated into the back-end via its public API, replacing the previous, limited hashing-based comparison approach.

5.2 Assessment of Automated Crawling

After the OpenAI's GPT-4.0 LLM was fully integrated into the back-end server, an automated crawling was made to assess its performance in a practical environment. The main goal was to evaluate the performance and effectiveness of the new semantic de-duplication mechanism in real-time crawling scenarios and to compare it with the basic implementation that relied on hash-based content comparison.

To perform this assessment, the crawler was executed multiple times on the custom-designed website, described earlier in the paper. Each crawler execution resulted in the data being stored in a dedicated */logs* directory, which was implemented as part of the back-end server to support systematic data collection. For every new crawled page, a request was sent to the back-end, where the page content was analyzed using an API call to the GPT-4.0 LLM. Additionally, the log files included data such as a timestamp, showing the exact time when the data was processed by the LLM, a full record of URLs visited every time, and the corresponding page content analyzed by the model. A total of 12 pages were visited, 1 page was not further explored because semantically similar to others, our investigation of the logs confirmed that the LLM-based approach was capable of detecting all the semantically similar pages and avoid unnecessary data collection.

However, the execution time of the LLM-based crawler was approximately 10 seconds, while the execution time of the basic crawler was approximately 4 seconds. This contrast highlights a key trade-off introduced by the integration of a Large Language Model (LLM): while the LLM-based approach significantly improved the semantic accuracy of de-duplication, it also introduced a higher computational cost. Nonetheless, in scenarios where semantic precision is essential - such as content classification or indexing — the added latency may be a worthwhile compromise, offering substantial improvements in data classification and uniqueness.

This LLM-based approach significantly improved the traditional hash-based approach, which was incapable of identifying semantic equivalences in the content of different web pages. Additionally, the LLM was able to interpret the underlying meaning of the content and recognize paraphrased information as duplicate. This results in greater precision in filtering out the duplicate data, ensuring that only unique data is processed and stored. Furthermore, the system remains scalable, supporting integration with diverse content formats and allowing for future extensions in content analysis or classification.

6 Conclusion

This paper presented the design, development, and empirical assessment of an automated web crawling system, extended to semantic reasoning using a Large Language Model (LLM). The project started from a basic content-based crawler, which relied on hash-based algorithm to detect duplicate pages. Although this approach was effective in handling exact content duplicates, it was insufficient to recognize semantic similarities with small changes.

To address this limitation, the crawler was extended by integration of a Large Language Model (LLM), namely the OpenAI's GPT-4.0 LLM. This enabled the system to perform semantic reasoning and to detect page similarities, even with some structural changes. The LLM was integrated into the back-end server via an API call, replacing the hash-based approach with a new, more sufficient prompt-based algorithm. Before the full integration of the LLM, it was tested manually using the ChatGPT interface to assess the model's capabilities to detect page similarities.

The extended crawler was then executed in a custom-designed website to assess its practical behavior. Multiple crawling sessions were made and the results were stored in a dedicated */logs* directory for later analysis. These results demonstrated the feasibility and potential advantages of the integration of LLM into an automated web crawling process.

Although the system is a prototype, the results received while testing were encouraging. They have shown that semantic reasoning performed by LLMs can be utilized to significantly improve the accuracy of web crawlers. Future improvements will focus on optimizing the crawling effectiveness and designing more advanced prompt engineering strategies to increase the crawler's performance and accuracy.

References

[1] Nazanin Bayati Chaleshtari, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand. 2023. Metamorphic Testing for Web System Security. *IEEE Transactions on Software Engineering* 49, 6 (2023), 3430–3449. doi:10.1109/TSE.2023.3256322

[2] Crawlee Contributors. 2024. Crawlee: Web Scraping and Automation Library. https://crawlee.dev/. Accessed: 2025-05-23.

[3] Crawlee Contributors. 2024. PuppeteerCrawler Class – Crawlee. https://crawlee.dev/js/api/puppeteer-crawler/class/PuppeteerCrawler. Accessed: 2025-05-23.
[4] Crawlee Contributors. 2025. CheerioCrawler Guide. https://crawlee.dev/js/docs/guides/cheerio-crawler-guide. Accessed: 2025-05-26.
[5] Crawlee Contributors. 2025. Playwright Crawler Example. https://crawlee.dev/js/docs/examples/playwright-crawler. Accessed: 2025-05-26.
[6] CryptoJS Contributors. 2025. crypto-js: JavaScript Library of Crypto Standards. https://www.npmjs.com/package/crypto-js. Version 4.2.0, Accessed: 2025-05-26.
[7] Dotenv Contributors. 2025. dotenv: Loads environment variables from .env file. https://www.npmjs.com/package/dotenv. Version 16.5.0, Accessed: 2025-05-26.
[8] Express.js contributors. 2025. body-parser: Node.js body parsing middleware. https://www.npmjs.com/package/body-parser. Version 2.2.0, Accessed: 2025-05-22.
[9] Express.js contributors. 2025. Express.js — Fast, unopinionated, minimalist web framework for Node.js. https://expressjs.com/. Accessed: 2025-05-22.
[10] Henry Gilbert, Michael Sandborn, Douglas C. Schmidt, Jesse Spencer-Smith, and Jules White. 2023. Semantic Compression With Large Language Models. https://arxiv.org/abs/2304.12512. doi:10.48550/arXiv.2304.12512 arXiv:2304.12512 [cs.AI].
[11] Troy Goode. 2017. cors: Node.js CORS middleware. https://www.npmjs.com/package/cors. Version 2.8.5, Accessed: 2025-05-22.
[12] Izzeddin Gur, Ofir Nachum, Yingjie Miao, Mustafa Safdari, Austin Huang, Aakanksha Chowdhery, Sharan Narang, Noah Fiedel, and Aleksandra Faust. 2023. Understanding HTML with Large Language Models. https://arxiv.org/abs/2210.03945. doi:10.48550/arXiv.2210.03945 arXiv:2210.03945 [cs.LG].
[13] Node.js contributors. 2025. Node.js — Run JavaScript Everywhere. https://nodejs.org/en. Accessed: 2025-05-22.
[14] OpenAI. 2023. GPT-4. https://openai.com/index/gpt-4/. Accessed: 2025-05-26.
[15] React Contributors. 2025. React – A JavaScript Library for Building User Interfaces. https://react.dev/. Accessed: 2025-05-26.
[16] Tailwind Labs. 2025. Tailwind CSS: A Utility-First CSS Framework. https://tailwindcss.com/. Accessed: 2025-05-26.
[17] Wikipedia contributors. 2024. Web crawler — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Web_crawler. Accessed: 2025-05-20.