PROJECT REPORT

Web Crawler et Indexer

Supervisor : Antoine Ferszterowski

Pierre-Ange BILLA | 3D | October 2024 – January 2025

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deep gratitude to all the stakeholders who contributed, directly or indirectly, to the realization of this project.

I would like to extend my thanks to ESME Sudria for giving me the opportunity to carry out this project. In particular, I would like to thank Mr. Damien Romanet, without whom this project would not have existed.

My gratitude also goes to Mrs. Raouda Kamoun, researcher and faculty advisor for the Big Data program, for her insightful advice and valuable guidance.

Finally, I would like to express my sincere appreciation to Mr. Antoine Ferszterowski, my project supervisor, who guided and supported me at every stage. His expertise and availability greatly contributed to the successful completion of this work.

TABLE OF CONTENTS

TABLE OF FIGURES

Introduction

As part of my engineering studies, I undertook the development of a web crawler and an indexing system to efficiently collect, organize, and search data from web pages. The goal was not so much to deliver a fully finished and ready-to-use product but rather to engage in an in-depth learning project, exploring the fundamental principles of web scraping, indexing, and deployment in a Cloud environment.

In practice, this project follows the entire lifecycle of a simplified search engine: extracting raw web content (HTML) via a crawler developed from scratch and supplemented with the Scrapy library, cleaning and enriching the data, indexing it into a real-time searchable database, and developing a web application that allows querying the index and displaying a list of relevant links, much like a real search engine.

My primary objective was to gain a comprehensive understanding of the process, from understanding how bots operate to implementing best practices. Additionally, I opted for a Cloud-based configuration on AWS to grasp the challenges related to scalability and automation. Although full deployment was constrained by certain limitations, this approach allowed me to experiment with modern enterprise solutions for search and data analysis.

Throughout this report, I will discuss the key aspects of the project, the challenges encountered and how I addressed them, as well as the major lessons learned that can be applied to future projects involving large volumes of data or high-performance requirements.

1 – Web Crawler

This project involves building a web crawler technology from scratch. Throughout this document, we will explore what this entails and how it was designed and implemented.

1.1 – What is a web crawler?

A web crawler, sometimes referred to as a “spider” or “bot”, is an automated program designed to explore and collect information across the World Wide Web, encompassing the entire internet domain.

When a crawler is launched, it typically starts from a manually configured list of initial URLs, chosen based on specific needs. It then systematically visits these pages, extracting various elements such as content (text, images, metadata, etc.) and hyperlinks pointing to other pages. The discovered links are subsequently added to the list of pages to be visited, and the process continues iteratively.

This method enables the bot to navigate from site to site, continuously retrieving more data as it progresses.

Figure 1 : Web crawler

The primary objective of a web crawler is to gather raw data, which can later be analyzed or indexed. Search engines, for example, rely on millions of bots that continuously explore websites to build massive indexes (database structures) that allow users to quickly find relevant information.

Similarly, many competitive intelligence and reputation monitoring tools operate on the same principle, using crawling techniques to collect and aggregate valuable information such as pricing data, customer reviews, consumer feedback, and news updates.

In this project, we developed a crawler from scratch, starting from a blank slate, in order to control every step of the exploration, extraction, and processing of data. This custom approach allows us to precisely define which pages we want to analyze, how deep the crawl should go (number of links followed from the starting page), and what exclusion rules to apply (for example, ignoring specific types of content).

1.2 – Significance and Major Challenges of a Web Crawler

The usefulness of a web crawler is measured by the quantity and quality of the data it manages to collect. By automating the process of discovering and retrieving information, it saves considerable time and allows handling very large volumes of data that cannot be processed manually.

In a business context, for example, a crawler can serve several purposes:

- Monitoring the competition, as it enables the detection of price changes or the appearance of new products in a market.
- Conducting technological or media watch by identifying articles, news, or trends to stay informed about developments in a specific industry.
- Building knowledge bases by assembling articles, definitions, or various resources, then indexing them to simplify search and retrieval.

To make it clear, bots accounted for approximately half of the global web traffic in 2022, or 47.5%.

In 2019, Googlebot and crawlers from the company Google represented 28.5% of the bots present on the web actively collecting data.

Besides bots like those from Google designed to build a search engine, there are also SEO crawlers. These crawlers send requests to websites to optimize them by identifying their technical weaknesses or improving their structure while aiding in the SEO strategy.

In this field, AhrefsBot, which visits around six billion pages each day, ranks first. It is considered the second most active bot after Googlebot.

The presence of crawlers will only continue to grow with the significant increase in the amount of data each day, as well as the massive development of AI.

However, implementing such a tool comes with significant technical and ethical challenges. Many websites publish a file called robots.txt, which specifies the sections that are forbidden or allowed for bots. A “responsible” crawler must comply with these rules to avoid overloading the server or accessing areas that are not intended to be indexed or exploited.

It is also essential to pay attention to meta tags (noindex, nofollow) in the HTML code. The “noindex” tag prevents certain pages from being indexed, while the “nofollow” tag instructs the search engine not to consider a specific link in its analysis. Respecting these crawling policies is crucial for building an efficient and ethical web crawler.

Moreover, the web is of colossal scale: it contains billions of pages, and these pages can be frequently updated. A crawler must therefore manage its storage space efficiently (to avoid keeping duplicates) and its resource consumption (bandwidth, memory, CPU). Without a minimum level of optimization, the crawling speed and the number of pages visited can quickly become unmanageable.

There are various strategies for deciding the order in which newly discovered pages are visited. One can opt for a breadth-first approach (visiting all links at level 1, then those at level 2, etc.), a depth-first approach (following a link continuously until there are no more, then going back), or weighting certain links based on priority (page recently updated, page from a more “important” site, etc.)

An efficient crawler does not settle for a one-time visit: it must regularly revisit the same pages to capture updates. A scheduling system must therefore be put in place to determine the frequency of visits to each site. In some sectors, pages can change multiple times a day, such as e-commerce sites or news pages, while.e others remain stable for long periods, such as institutional pages

Websites do not appreciate large-scale data collection and protect themselves against bots by sometimes limiting the number of requests. It is therefore important to manage traffic properly, ensuring that servers are not overloaded by setting delays between requests and limiting the number of simultaneous connections.

1.3 – The web crawler's role in our project

In this project, the web crawler primarily serves as a technical foundation for all aspects related to data collection and preparation. Here are some key points that illustrate its concrete role :

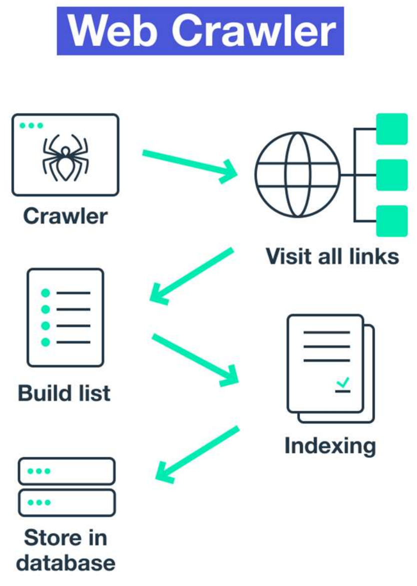


Figure 2 : Web Crawler operations

Everything begins with the configuration of a list of web addresses that we identify as relevant to our project. In our case, we used sites like lemonde.fr or openclassrooms.com, which allow a significant amount of web scraping and contain a variety of regularly updated topics. The crawler visits each of these URLs, retrieves the content (text, images, etc.), and stores it in raw form in our temporary storage system, represented by lists. This first step quickly provides a representative sample of the content we seek to analyze while giving the crawler concrete starting points to extend its exploration.

Additionally, our crawler operates autonomously and can be scheduled to launch or relaunch regularly. For example, it can be triggered every night to detect new content or updates on already explored sites. In our case, we wanted it to run continuously.

During exploration, the crawler analyzes the HTML code of the already visited pages to detect all outgoing links or hyperlinks. Each new relevant link is visited and added to a list, allowing us to filter the links and avoid returning to sites that have already been visited before.

Once the raw content is collected, we apply preprocessing to transform the "as-is" text into more structured information. This is where scraping techniques and content analysis come into play.

First, scraping, which is the targeted extraction of specific elements (titles, meta tags, paragraphs, prices, reviews, etc.) from the HTML code. In our case, we wanted the raw text, so we had to remove all redundant content such as headers and footers.

Next comes the cleaning and normalization step. This involves removing unnecessary elements (useless tags, scripts, special characters, etc.) and converting the content into a more uniform textual format.

Finally, we reach the indexing stage, which consists of generating or updating the index that will allow for fast keyword searches. At this stage, we can also store metadata (visit date, update frequency, source, page language, etc...).

At this point, the list of websites and their content is stored in an index, which serves as a database that we can query to find useful information.

Implementing a custom crawler allowed us to finely tune our collection criteria (delays between requests, depth of link following, specific processing based on page format) both to maximize data relevance and to avoid aggressive behavior toward the explored sites.

Ultimately, the web crawler is the essential starting point of our processing pipeline: it explores, detects, collects, and supplies fresh data for all subsequent steps. Its proper functioning determines the success of the entire project, whether it be for indexing, building a web application, or setting up dashboards for data visualization.

2 – Project Architecture

After explaining what a web crawler is and how it works globally, we will now examine the architecture of the project in order to understand its mechanisms and detailed functioning.

Here is the overall architecture of the project:

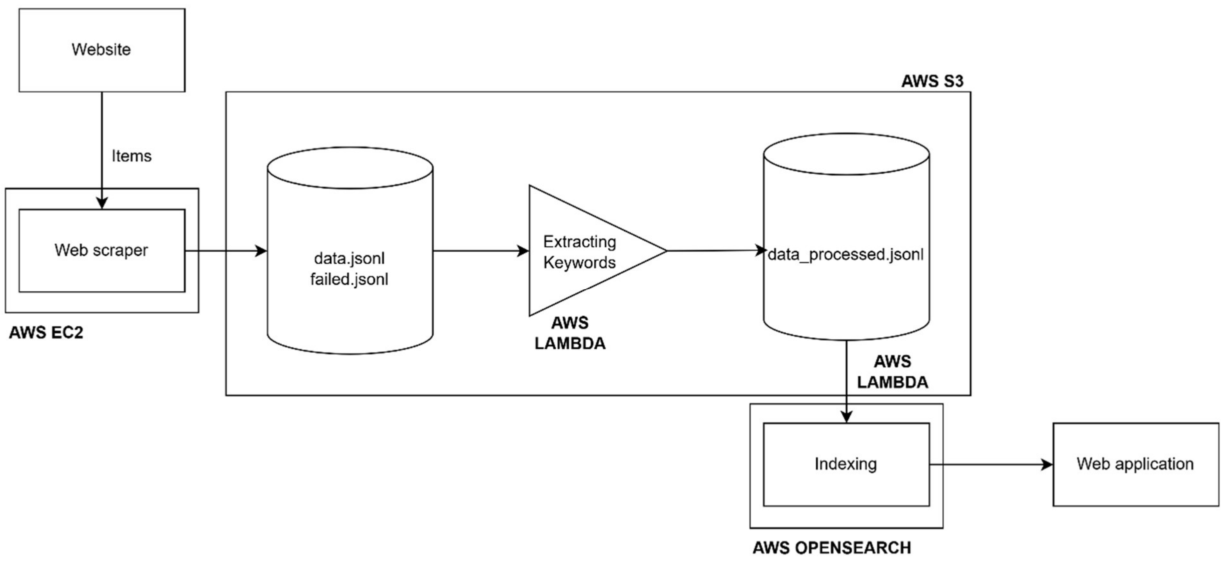


Figure 3 : Project's flowchart

2.1 – Web Scraping

Web Scraping consists of automating the extraction of data or content from websites. Concretely, this practice relies on retrieving the HTML code of a page and then extracting the relevant information (texts, images, titles, prices, etc.) to build a database or feed other processing tasks. It is one of the key components of web crawlers and marks the first step of the workflow.

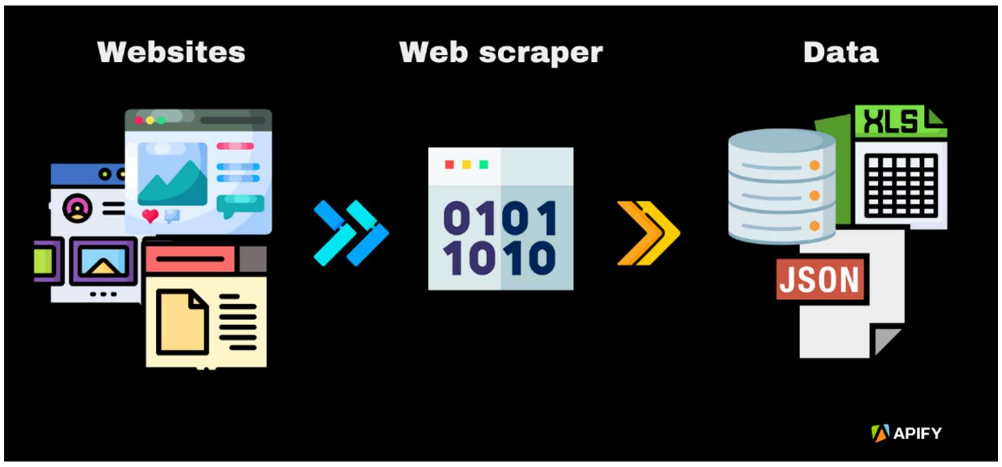


Figure 4 : Web scraper workflow

2.1.1 –Scrapy Framework

As part of our project, we chose the Scrapy framework, a powerful and flexible open-source tool specifically designed to handle large volumes of data. Scrapy offers modular architecture, making it easy to define the data to extract, manage HTTP request errors, ensure the quality of collected information, and finely configure the behavior of the scraping bot.

Scrapy is structured around four main components.

2.1.1.1 - Items

The items define the structure of the collected data (for example, a title, a URL, or content). In our project, we chose to store a minimal set of information to maintain consistency and avoid an excess of unnecessary fields.

Concretely, this is implemented by specifying in the construction of the web crawler the fields we want to retrieve from each visited page. In our case, the fields are :

- Title : The title or the name of the webpage
- URL : The link to the page
- Content : The body of the main text or all the essential elements

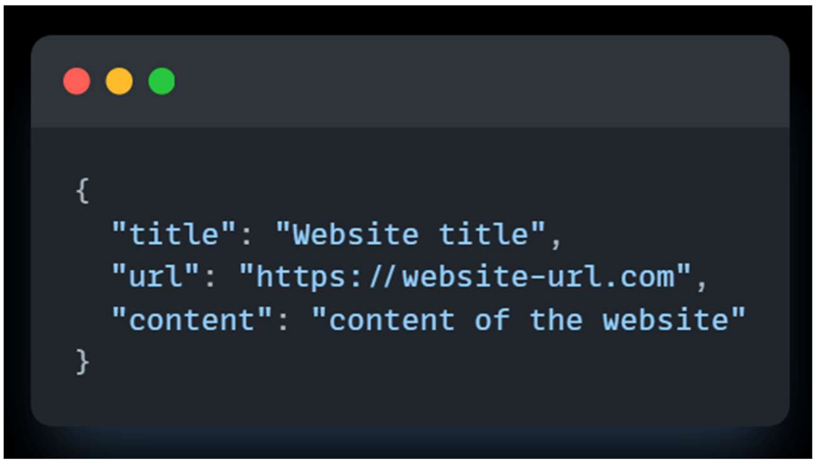

```
{
  "title": "Website title",
  "url": "https://website-url.com",
  "content": "content of the website"
}
```

Figure 5 : Item

The information is stored in JSONL (JSON Lines) format in an output file.

Unlike a standard JSON file, where items would be structured within arrays or objects, using JSONL allows each item to be completely independent, even though they share the same structure. This facilitates subsequent processing and interoperability between different tools.

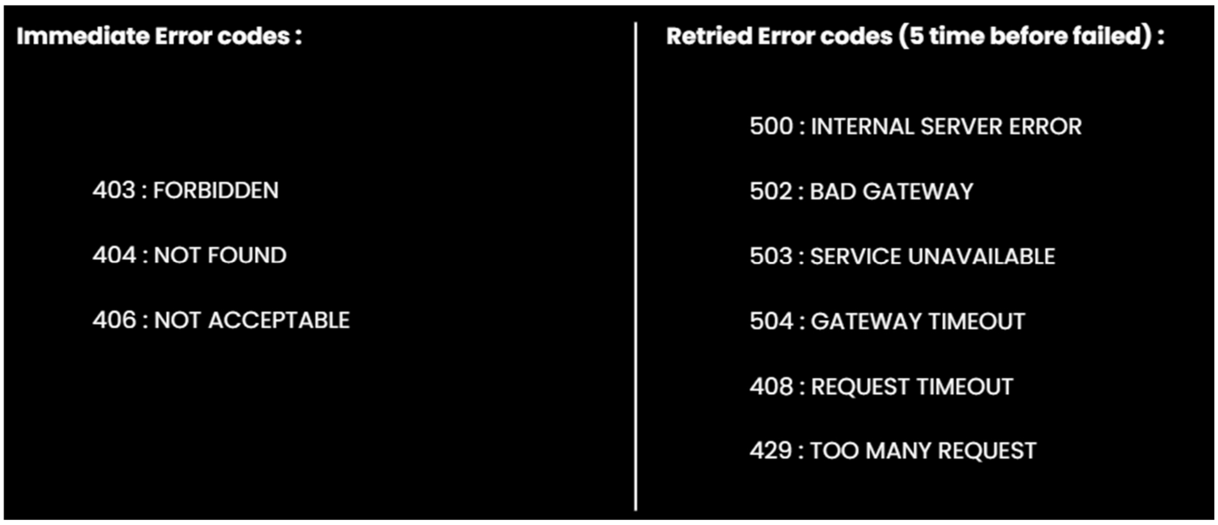
The use of items ensures data homogeneity, simplifying their manipulation and updating, which is particularly useful for a project that retrieves large volumes of data.

2.1.1.2 – Middlewares

Middlewares act as intermediate layers to adapt the scraping behavior based on specific needs or errors encountered. They allow us to customize the crawler’s behavior when sending requests or receiving responses.

In our case, they were primarily used to intercept and automatically handle HTTP errors (HyperText Transfer Protocol). Specifically, we defined a list of possible HTTP errors and configured the crawler's behavior when encountering them.

Managing these errors allowed us to create a list (failed.jsonl) in which we recorded all the sites considered inaccessible.


Immediate Error codes :

403 : FORBIDDEN

404 : NOT FOUND

406 : NOT ACCEPTABLE

Retried Error codes (5 time before failed) :

500 : INTERNAL SERVER ERROR

502 : BAD GATEWAY

503 : SERVICE UNAVAILABLE

504 : GATEWAY TIMEOUT

408 : REQUEST TIMEOUT

429 : TOO MANY REQUEST

Figure 6 : HTTP errors

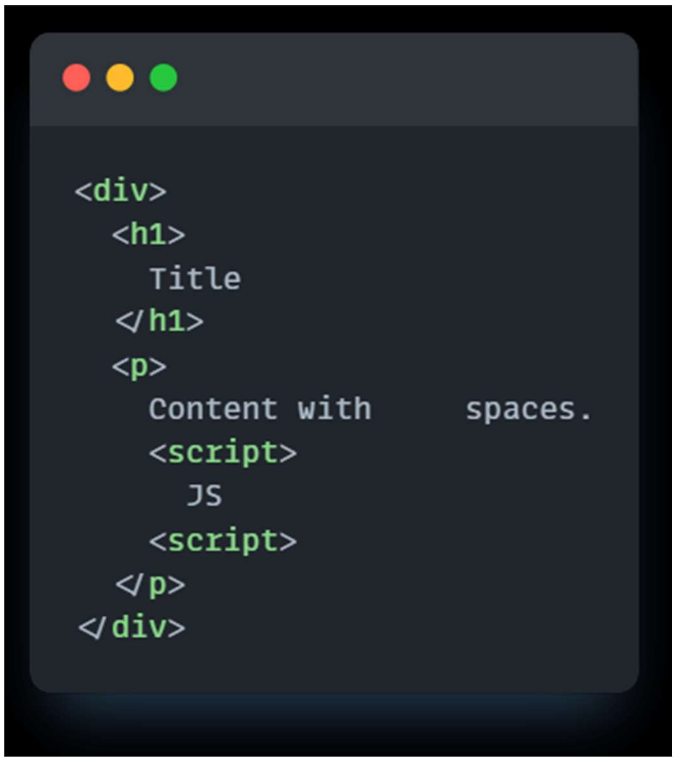
Some error codes immediately placed sites into our list of inaccessible sites because they represent permanent errors. However, for temporary errors, we implemented a condition: when encountered, the program attempts to access the site up to 5 times before finally considering it inaccessible. This helps reduce the frequency of errors over time.

The failed.jsonl file can later be used as an input for the crawler, specifying which sites should be skipped to avoid unnecessary access attempts.

Thanks to these middlewares, we gain robustness: the scraper can handle failures, nonexistent pages, or access refusals without interrupting the overall process.

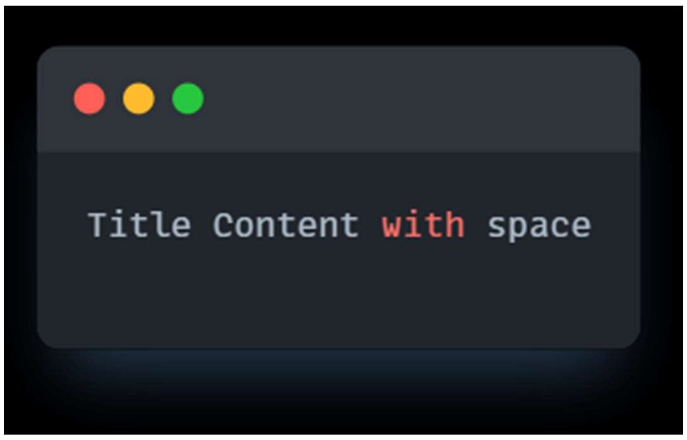
2.1.1.3 – Pipelines

In the Scrapy framework, pipelines handle the post-processing of extracted data. The first step involves cleaning the information (removing HTML tags, extra spaces, scripts, styles, special characters). This is achieved using the BeautifulSoup library, which allows us to eliminate unnecessary elements and obtain a more readable text.



```
<div>
  <h1>
    Title
  </h1>
  <p>
    Content with     spaces.
    <script>
      JS
    <script>
  </p>
</div>
```

Figure 7 : Input data before the pipeline



```
Title Content with space
```

Figure 8 : Output data after the pipeline

The pipeline also allows us to validate the presence of mandatory fields (title, URL, content). If an item is missing one or more of these elements, it is rejected and added to the list of unusable sites mentioned earlier. This ensures a high-quality dataset.

Valid data, meaning items that are fully completed, are then passed to a storage module, which will be covered in the next section.

These different steps ensure reliable extraction, reduce redundancy (same pages explored multiple times), and make the information easier to use later for text analysis, indexing, etc…

2.1.1.4 – Settings

The settings of our crawler allow us to deeply configure the behavior of the Scrapy project. Concretely, this allows us to :

- Respect exploration rules by enabling ROBOTSTXT_OBEY and defining an appropriate USER_AGENT, preventing violations of the access policies described in robots.txt.
- Manage request load by limiting the number of simultaneous requests to avoid overloading servers. We also introduce a waiting time between requests to be less invasive. In our case, the delay between requests is one second, which helps prevent HTTP 429 “too many requests” errors. Similarly, we have limited the number of simultaneous requests to 15.
- Specify the storage location by choosing a local directory or a cloud service for data storage. This is an option that will be detailed in a later section, but it allows us to switch from a development environment to a production environment in seconds.
- Enable or disable certain modules (middlewares and pipelines): depending on the testing or production environment, we can disable some features to improve performance or, on the contrary, activate additional checks to enhance reliability.

2.1.1.5 – Main Spider

Now that we have presented the four components of the Scrapy framework, we will examine the operation of the core of the program, which we have named the “Main Spider”.

It represents the main script responsible for exploring pages and extracting the elements defined in our Items. It specifically handles the discovery of new links, meaning that when a page is visited, the spider identifies outgoing URLs and adds them to the exploration queue if they are relevant. This process starts from the list of websites provided as input.

It also plays a role in duplicate management. To prevent crawling the same page multiple times and polluting the database with duplicates, Scrapy includes a DupeFilter system that detects and ignores URLs that have already been processed.

Additionally, each time the program is restarted, it takes as input the previously collected data file as well as the file of inaccessible sites, ensuring that it does not revisit the same paths.

This component serves as the interface between the raw data extraction (HTML) and the application of various mechanisms (items, pipelines, settings...) that transform the content into a final dataset.

2.1.2 – Main challenges

Despite the robustness of the Scrapy framework, we encountered several obstacles.

First, duplicate management is not straightforward. Even with the two solutions mentioned in the previous section (DupeFilter and input item list), filtering is not always sufficient.

For example, we faced URLs that looked very similar but were technically different while still redirecting to the same page. For instance, if we take this website :

- https://www.esme.fr/formation-ingenieur/cursus-ingenieur/

We might encounter a second link that redirects to the same page :

- https://shorturl.at/Aeky5

These two links lead to the same site. So, despite having a different URL, the content remains identical. We consider this a duplicate, but handling this case is very complex, as each website has its own unique structure.

Additionally, we faced difficulties in providing reliable entry links to our crawler. Many websites are protected by the rules defined in the robots.txt file, which limit the number of links we could use as input.

406 "Not Acceptable" errors were very complex to handle. Websites recognize bot behavior, and despite limiting simultaneous requests and introducing delays, many sites still blocked access. One technique we used was to simulate web headers when making requests, but this did not fully resolve the issue, meaning that this error could not be entirely bypassed.

Finally, in data retrieval, the main challenge was figuring out how to extract content given that each website had a unique HTML structure. In a basic web scraper, we would simply specify the HTML tags containing the information we wanted to extract, but in this case, that was not possible.

Despite these challenges, Scrapy and our customized configuration allowed us to overcome most issues and efficiently retrieve large amounts of information.

In summary, the Web Scraping approach we implemented, using Scrapy, forms the essential building block for enriching our database. It integrates directly into the overall workflow: the crawler handles exploration, the Scrapy spider performs extraction, and the cleaned and validated data is then passed to our indexing and storage system for further use.

2.2 – Storage and Automation solutions

Once all this data has been collected, the aim is to know how to store this large volume of data and automate its collection in a robust way.

2.2.1 – Data Storage Solution

Once the scraping process is launched and the data is collected and cleaned, the question arises, where and how to store it in a sustainable manner.

During the development and testing phase, it is common to store data locally, for example, in a simple JSONL file or in a database hosted on our machine. However, as the volume of data increases, it becomes necessary to move to a more robust production environment. This is also the case when an application is made available to the public, it requires a stronger hosting solution than a local setup, especially as the number of end users grows.

Throughout the testing phase, we opted for local storage. Concretely, the Scrapy pipeline saves the extracted elements in JSONL files, located on the disk of the machine running the crawler. This approach offers several advantages during development.

Indeed, this approach simplifies execution, as the configuration is minimal and reduces dependencies. Additionally, debugging is faster and easier with direct access to files, allowing verification and correction of potential extraction or cleaning errors. This solution incurs no hosting or cloud service costs, which is unnecessary when the data volume is low, and the program remains relatively small.

However, this storage method quickly reaches its limits. As soon as there is a need to share data or handle large volumes, it becomes necessary to switch to a more suitable storage solution.

To implement the project in a production environment, meaning an environment that would, in theory, allow for the industrialization of our project, facilitate data collection, and make files centrally accessible, we turned to Amazon Web Services (AWS). More specifically, we opted for Amazon S3 (Simple Storage Service).

AWS is a cloud service platform that offers a wide range of hosting, computing, database, and storage solutions. Among these services, S3 is designed for object storage (files of all sizes, images, JSON, logs, etc.) and offers several features that are particularly relevant to our needs.

First, AWS S3 provides high scalability, as it can handle massive volumes of data (up to petabytes) without requiring users to manage infrastructure. It also ensures high availability, with files replicated across multiple Availability Zones (AZs) to minimize the risk of data loss.

AWS offers secure and flexible access management via access keys, allowing users to restrict or grant read/write permissions and even automate lifecycle policies (file expiration, archiving to other storage classes).

For our project, we created an S3 bucket (a storage container) specifically dedicated to data retrieval. The files generated by Scrapy (in JSONL, CSV, or other appropriate formats) are automatically transferred to this bucket, making them shareable between multiple servers and accessible to other services or applications

Thus, we have two environments:

- A development environment, which allows us to experiment and run tests in a simple and local way.
- A production environment, which provides a more robust infrastructure for scalability and reliability.

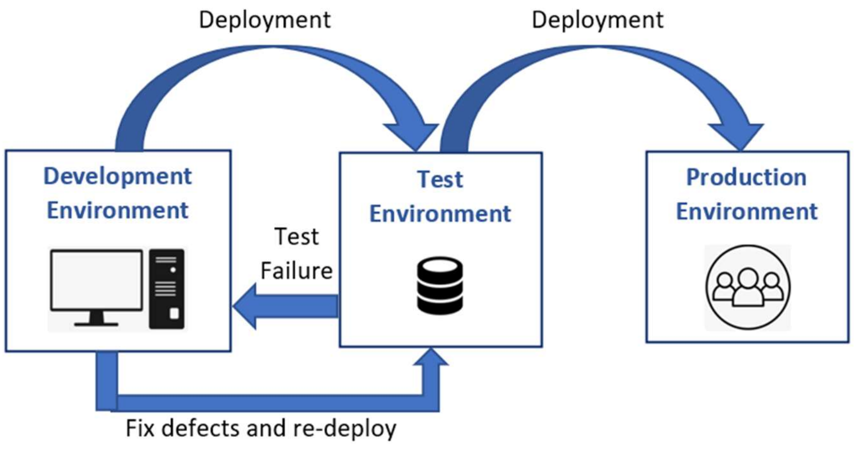


Figure 9 : Transition from development to production environment

The transition from one environment to another was simplified by implementing an abstract class in our program. This allowed us to configure both environments independently and switch between them quickly.

To switch from one environment to another, we simply change a parameter in the Scrapy settings under the corresponding component name. This makes the workflow more flexible, allowing us to run tests easily and quickly revert to a development phase when needed.

2.2.2 – Process Automation

After implementing our scraper and choosing a storage solution (S3 in production), the next step is to ensure that it runs continuously, or at least that it is automatically restarted at regular intervals. This allows for data collection and updates without human intervention, an essential element for a production project aiming to always provide fresh information.

To host and execute our program autonomously, we opted for Amazon EC2 (Elastic Compute Cloud), which provides virtual machines (instances) in the cloud. An EC2 instance allows us to install and configure an operating system on which we can run any application. It is on this virtual machine that we decided to copy our program, thanks to its hosting on GitHub.

This has advantages such as ease of deployment, flexibility (choice of CPU resources, memory, storage), and native integration with other AWS services (S3, IAM, etc.). However, it also has disadvantages, such as cost, which can become high if an instance runs 24/7, especially with powerful configurations.

To reduce costs, we decided to use Spot Instance, which is an option that allows renting AWS's excess computing capacity at a more advantageous rate (sometimes significantly lower than the standard on-demand price, up to ten times cheaper, for example).

Unfortunately, this technique has a major drawback: AWS can stop the Spot Instance at any time (with short notice) if capacity decreases or if the Spot price exceeds the maximum price set.

In summary, using Spot Instances is very cost-effective, but it introduces the risk of sudden instance termination. For a crawler meant to run continuously, this can be a problem.

Since Spot Instances can be interrupted at any time, our machine may shut down or restart unpredictably. In a typical scenario, if we manually launch the scraping program using a command (in our case: “scrapy crawl web_crawler”), it stops as soon as the instance shuts down. Upon restarting, everything would then have to be manually relaunched, which goes against our goal of full automation.

To address this issue and make the crawler autonomous, we configured systemd, an initialization and service management system present on most modern Linux distributions.

We define a unit file that specifies how and when to launch the Scrapy script. This file includes details such as the user, the execution commands, and the behavior to adopt if the service stops or encounters an error. Using a specific systemd command, we configure it to automatically start at every system boot. Thus, even if the Spot Instance restarts, the crawler relaunches without any manual action.

Additionally, if the service unexpectedly stops, systemd can be configured to automatically restart it after a set delay.

By combining a Spot Instance (to significantly reduce costs) with systemd (to ensure automatic program relaunch), we achieve an architecture that is both cost-effective and robust. The crawler runs continuously as long as the Spot Instance is available. In case of a sudden restart, systemd ensures that the process is reactivated. This configuration meets the needs of automation and cost reduction for a web crawler in a production environment.

2.3 – Keyword Extraction and Indexing

Now that our program is continuously retrieving data and storing it in a container on AWS. We need to be able to index it using keywords in particular. In this section, we'll look at how we've configured this process.

2.3.1 – Keyword Extraction

First of all, regarding keyword extraction, it must also be automated.

Our web scraping pipeline regularly sends new files (in batches of 10 items) to an Amazon S3 bucket. As soon as this bucket is updated, an S3 trigger (a native event mechanism) automatically calls an AWS Lambda function. This Lambda function forms the core of the keyword extraction process.

AWS Lambda is a serverless computing service that executes code in response to events (such as an S3 bucket update) without requiring management or provisioning of servers. Users only pay for the compute time consumed, making it ideal for sporadic and asynchronous processing like this.

Now, let's detail what happens inside this Lambda function that we have configured :

Once the Lambda function is triggered by the bucket update event, we start by identifying the language of each document. For this, we use the langid library. This library reads the text content of the item stored in the "content" field. It returns a language code (for example, “en” for English, “fr” for French). This step is crucial because once the language is detected, it allows us to perform the next step more effectively.

After determining the document's language, we apply the NLTK (Natural Language Toolkit) library to first load the appropriate stopwords list (meaningless words such as “et”, “le”, “la” in French or “and”, “the”, “of” in English). This is why language detection beforehand is important, as it ensures the correct stopwords are removed from the start.

The library filters the text content by systematically removing these terms, which do not provide any useful semantic information.

This cleaning process significantly improves the relevance of later analyses, as high-frequency words (conjunctions, articles, prepositions) no longer affect the calculation of word relevance.

Next comes the core of the machine learning processing: the analysis of term frequency and relevance within the text.

We use Scikit-Learn and its TF-IDF (Term Frequency – Inverse Document Frequency) algorithm to assign a weight to each word.

First, TF (Term Frequency) measures how often a term appears in a document (the more it appears, the more important it is for that document).

Then, IDF (Inverse Document Frequency) measures the rarity of a term across all documents (the rarer a word is globally, the more significant its presence in a given document).

The combination of TF and IDF allows us to highlight words that truly characterize a document (frequent in that document but uncommon elsewhere) while reducing the importance of overly generic or recurring words found in most texts.

After these initial processing steps, we continue using Scikit-Learn with the TFIDFVectorizer module. This module allows us to exclude words that appear in an excessively high percentage of documents (in our case, we remove the terms present in more than 90% of the texts). This option helps eliminate words that are more specific than common stopwords but still too frequent to be meaningful (e.g., “article”, “chapter”, “section”, or other recurring terms depending on the corpus).

Once this calculation is completed, each document is assigned a score for each term. We can then rank the terms in descending order based on their TF-IDF score and finally retain the top 10 most relevant words.

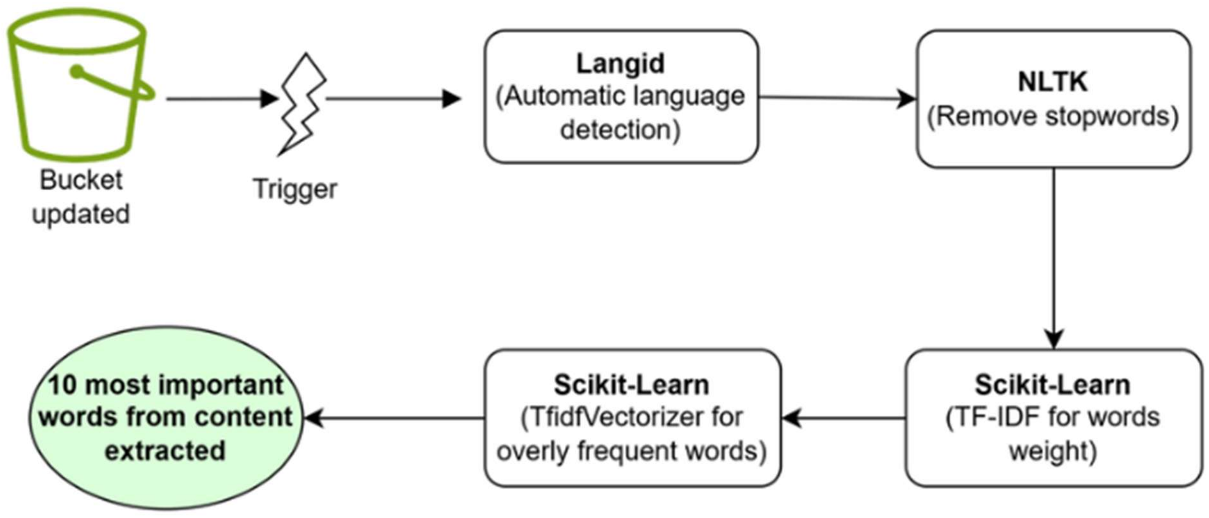
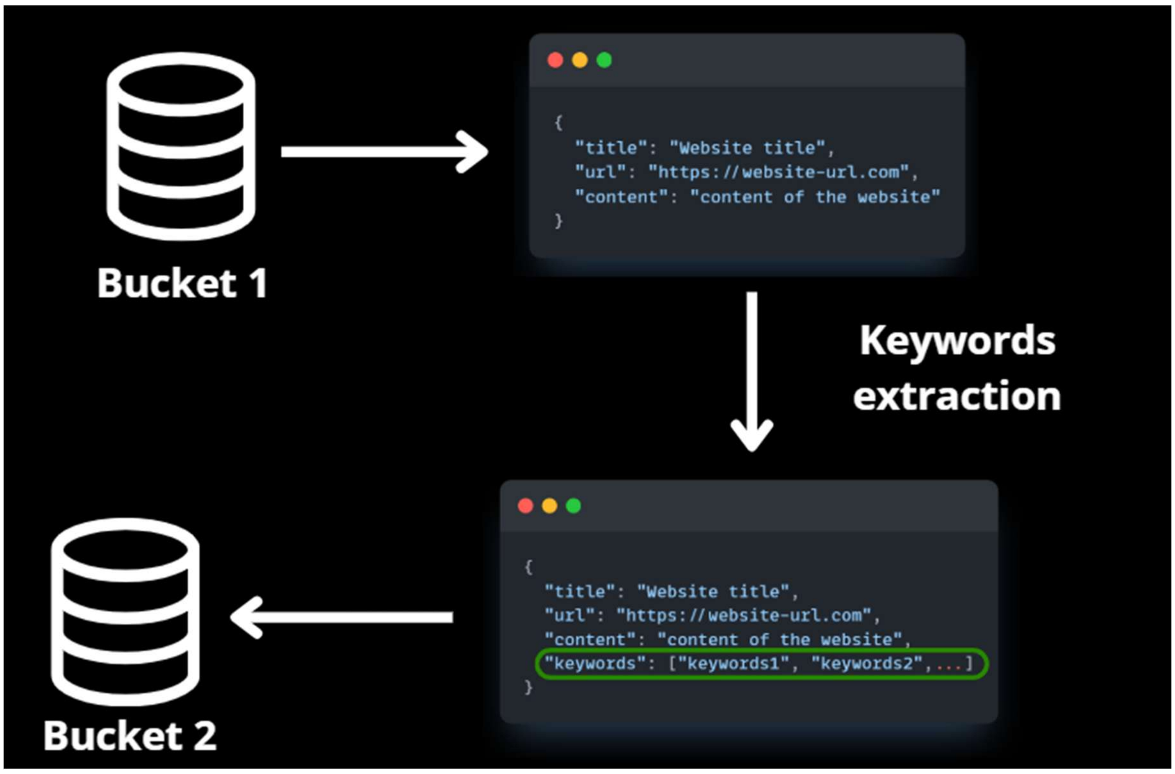

Bucket updated
Trigger
Langid
(Automatic language detection)
NLTK
(Remove stopwords)
Scikit-Learn
(TF-IDF for words weight)
Scikit-Learn
(TfidfVectorizer for overly frequent words)
10 most important words from content extracted

Figure 10 : Keywords extraction workflow

The goal of these operations is to add an additional field, “keywords”, to the Item, containing this list of keywords. This enhancement provides a concise overview of the content of the article or web page and significantly facilitates research and thematic groupings.

After this post-processing step, the Lambda function generates a new file containing the enriched items (with the new “keywords” field). This file is then stored in a second dedicated S3 bucket, specifically for data ready for indexing.

This approach clearly separates the source bucket, which contains the raw, unanalyzed data, from the target bucket, which holds the finalized files, ready for indexing and consumption by other services (internal search engines, web applications, etc...).



```
{
  "title": "Website title",
  "url": "https://website-url.com",
  "content": "content of the website"
}
```

```
{
  "title": "Website title",
  "url": "https://website-url.com",
  "content": "content of the website",
  "keywords": ["keywords1", "keywords2",...]
}
```

Figure 11 : keywords field added to item

Thanks to this machine learning approach, the larger the document corpus grows, the more accurate the TF-IDF model becomes. Words that were previously considered rare may become more frequent as they appear in additional documents, automatically rebalancing the scores.

This ensures that keyword extraction remains consistently adapted to the evolution of the dataset. This machine learning method is particularly well-suited for our project, given the massive amounts of data the system must handle.

Moreover, deployment via AWS Lambda provides scalability and flexibility: if the volume of documents to process increases, we can simply increase the parallel execution of Lambda functions, without having to worry about additional infrastructure.

In short, keyword extraction is a key asset in enhancing the value of the collected data. By leveraging language identification, cleaning (stopwords), and TF-IDF algorithms, each document is assigned a list of relevant keywords, strengthening the quality of indexing.

Combined with automated triggering via an S3 trigger and execution through a Lambda function, this architecture delivers a fully serverless and scalable solution, which is crucial for an ambitious web crawler project.

2.3.2 – Indexing

The final step in the processing pipeline is to index the enriched items, now containing their associated keywords, into a search engine. The goal is to make this information easily searchable and usable later (via a web interface, an API, or any other data-consuming service). In our project, we aim to use them through a web interface.

Following the keyword extraction process, once our Lambda function has generated a file containing the enriched items (with the “keywords” field) and stored it in a second S3 bucket, a new S3 trigger comes into play. This trigger calls a second Lambda function, specifically dedicated to indexing.

The function retrieves the JSONL file stored in the new bucket and processes each item to index it into a search engine. For this, we chose OpenSearch, the search and analytics solution provided by AWS.

Originally, OpenSearch was developed by AWS as a fork of Elasticsearch (v7.10) after Elasticsearch changed its licensing model. This means that OpenSearch is a new software created from the source code of an existing software, in this case, Elasticsearch. OpenSearch remains an open-source project under the Apache 2.0 license.

Just like Elasticsearch, OpenSearch is a distributed search and analytics engine. It enables fast storage and querying of large volumes of textual data using inverted indexing and efficient REST APIs.

Its integration within the Amazon OpenSearch Service simplifies management (creation, scaling, backups, etc.) of an OpenSearch cluster, eliminating the need to manage the infrastructure manually.

With OpenSearch (or Elasticsearch, its historical equivalent), it is possible to perform fast and relevant full-text searches, aggregated analyses (facets, histograms, keyword aggregations), as well as advanced features such as autocomplete, similarity-based search, and highlighting (emphasizing found terms).

In our project, we will utilize its similarity-based search function using keywords.

To go into more detail about the indexing workflow, the second Lambda function manages the connection to the OpenSearch cluster (hosted on Amazon OpenSearch Service).

First, the Lambda function is provided with the necessary parameters (OpenSearch cluster URL, index name, access credentials) to retrieve the index references. Then, for each batch of documents, the function calls the OpenSearch API using HTTP POST or PUT requests in JSON format to insert or update records in the index.

The function receives a success or error response, which is then logged in CloudWatch to ensure tracking and debugging.

The indexing of our items provides many essential advantages for the realization of our project. Thanks to inverted indexing, OpenSearch allows us to instantly retrieve documents based on their keywords, titles, or any other relevant fields.

For example, keywords can be used to group processed pages by theme, measure the frequency of term occurrences, or even build analytical dashboards if needed.

As with most AWS services, it is possible to increase the size of the OpenSearch cluster or add additional nodes to handle a higher query load. Scalability is an important factor in projects of this scale.

Finally, OpenSearch provides APIs compatible with the Elasticsearch ecosystem, meaning that most tools and libraries built for Elasticsearch (Kibana, Beats, Logstash, etc.) can also be used in an OpenSearch environment.

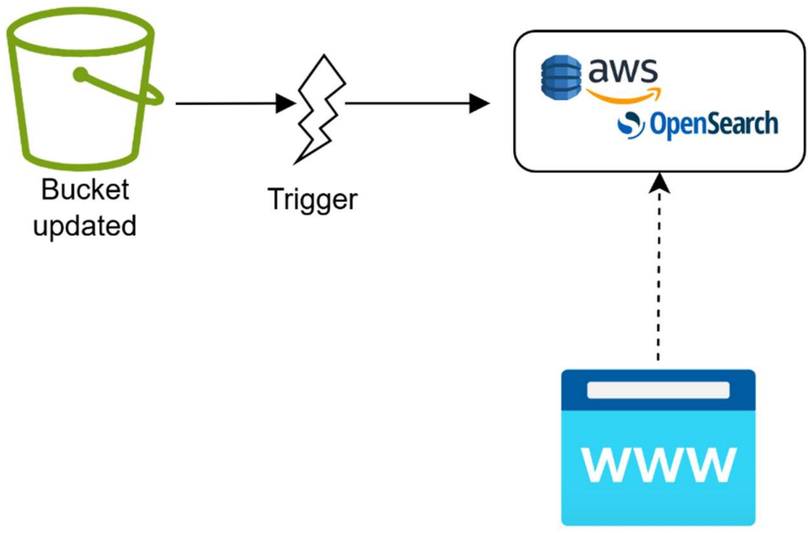


Figure 12 : Indexing workflow

With this indexing step in OpenSearch, our data pipeline reaches its completion:

- Raw documents are collected by the crawler and stored.
- They are cleaned and enriched using keywords and machine learning.
- They are then indexed into a high-performance search engine, enabling advanced queries and efficient navigation among thousands or even millions of documents.

This architecture ensures optimal utilization of content in various scenarios: data visualization, trend analysis, a search engine integrated into a web application, etc. It highlights the importance of a complete ecosystem (S3, Lambda, OpenSearch) to cover all needs, from data collection to fast and relevant access.

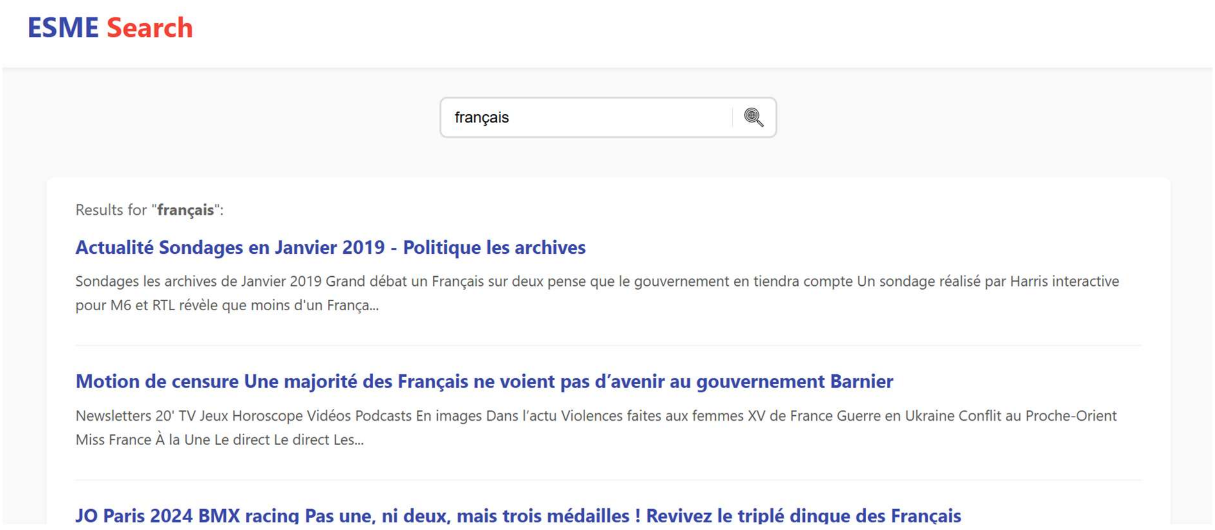
2.4 – Web application

After collecting, cleaning, and indexing the data, the final step to make the information accessible is to develop a web application that allows the user to search and view relevant content.

In our project, we designed a small search engine offering an interface similar to what is commonly found on the web (Google, Bing, etc.). The application is structured as a search bar:

- The user enters a keyword or a phrase.
- The system sends the query to our search service, which queries the index stored in OpenSearch.
- The results are then displayed in the form of a list of documents, where each result includes:
    - The clickable title of the page
    - An excerpt of the content or keywords to provide context

By clicking on a title, the user is redirected to the original page, allowing them to view the source site if needed. The interface is paginated for better navigation.


ESME Search

français

Results for "**français**":

Actualité Sondages en Janvier 2019 - Politique les archives

Sondages les archives de Janvier 2019 Grand débat un Français sur deux pense que le gouvernement en tiendra compte Un sondage réalisé par Harris interactive pour M6 et RTL révèle que moins d'un França...

Motion de censure Une majorité des Français ne voient pas d'avenir au gouvernement Barnier

Newsletters 20' TV Jeux Horoscope Vidéos Podcasts En images Dans l'actu Violences faites aux femmes XV de France Guerre en Ukraine Conflit au Proche-Orient Miss France À la Une Le direct Le direct Les...

JO Paris 2024 BMX racing Pas une, ni deux, mais trois médailles ! Revivez le triplé dingue des Français

Figure 13 : Web browser application

Concretely, when the user enters a query, the application transmits it to the OpenSearch cluster (via a REST API). The engine performs a "full-text" search or a keyword-based search (keywords) to identify the most relevant documents.

The results are sorted by score of relevance, and OpenSearch can apply filters (for example, the language, date of update, etc...) if desired. The application retrieves the

JSON response and generates the results page, displaying the title, the excerpt, and a link redirecting to the original resource.

The interface offers an intuitive way to perform searches, comparable to usual search engines. Thanks to OpenSearch indexing, the responses are returned in a few milliseconds, even on large volumes of data, which is an important parameter for the acceptability of such a project and a smoother user experience.

As the crawler feeds the database and keyword extraction methods improve, the relevance of the results increases, and the web application presents increasingly updated data. This is a key strength of our architecture, which, being designed to accommodate more and more data, improves its performance as resources grow.

By offering a web application that interacts directly with the OpenSearch index, we provide the user with familiar experience: enter a keyword, obtain a list of relevant links, and quickly access the original pages.

In this way, the vast volume of data collected by the crawler is transformed into a scalable and usable service, useful for monitoring, document research, or thematic consultation. Ultimately, we have implemented a true vertical search engine, tailored to our specific needs, which serves as the final achievement of the architecture developed throughout this project.

3 – Challenges

The development of our web crawler project was not without difficulties. While the architecture described earlier (with AWS EC2, S3, Lambda, OpenSearch...) provides a fully developed solution in a production environment, certain financial and technical obstacles prevented its full implementation within the scope of this project.

3.1 – Budget constraints

Despite the considerable advantages offered by the AWS stack (notably scalability, high availability, and automation), the use of certain services can quickly lead to significant costs. However, we did not have the necessary budget to complete the deployment.

The continuous operation of EC2 instances and the setup of an OpenSearch cluster on Amazon OpenSearch Service incur substantial expenses, often exceeding the Free Tier quotas. OpenSearch is a recent and costly service. I was charged around $30 just after configuring it, before even starting any indexing.

Although there is a limited free tier, costs can quickly increase if large files are frequently updated or transferred. Amazon S3 allows significant storage volumes, even within the Free Tier. However, frequent updates to these buckets are not feasible at a large scale without allocating a dedicated budget.

The AWS Free Tier is not designed for a project that heavily relies on multiple resource-intensive services (EC2, OpenSearch, S3, Lambda, etc.) on a continuous basis.

Due to these constraints, we could not fully validate the production deployment of the entire architecture. Testing and demonstrations were therefore limited to a local environment or short-term partial deployments on AWS.

The project will run only in a development environment, with manual data extraction, keyword extraction to be performed later, and standard indexing via ElasticSearch, which we have configured in a Docker container to make it accessible.

Nevertheless, the configuration of these services has been completed, paving the way for full industrialization once an adequate budget becomes available.

3.2 – Learning

Despite the budget limitations, this project has fulfilled its primary objective, which was to enable a concrete and in-depth learning experience about web crawling technologies and cloud architecture best practices.

I specifically learned how to configure a production environment with the creation of S3 buckets, the configuration of Lambda functions, the deployment (even partial) on EC2, and the setup of an OpenSearch cluster.

The project was made resilient through error handling and the implementation of unit tests. For example, we tested item retrieval using mock HTML pages, ensuring the proper functioning of the scraper and data cleaning pipelines.

We also integrated HTTP error handling and a retry mechanism to prevent blockages or unexpected program interruptions.

I learned how to implement abstract classes, allowing for a quick switch from a local environment to a cloud environment by simply modifying the Scrapy framework settings.

I configured systemd on an EC2 instance to automatically launch the crawler at startup and restart it in case of failure, ensuring continuous operation in case of interruptions or reboots (especially useful for Spot Instances).

In summary, even though it was not possible to keep the solution "in production" due to the costs associated with AWS, the software infrastructure is in place and ready to operate in a professional environment with an appropriate budget.

This experience provided a very concrete insight into the deployment and automation of a web crawling project, which can later be replicated or expanded without difficulty.

Conclusion

Throughout this report, we have explored how to design a complete web crawler project from end to end, from the initial retrieval and exploration of web pages to the development of a web search application.

On a technical level, choosing to develop a crawler from scratch, combined with the Scrapy framework, proved to be an excellent way to master each component: from meticulous exploration of target sites to data extraction, including error handling and the implementation of best practices.

The subsequent stages of the pipeline, from storage on AWS to data enrichment through keyword extraction, followed by indexing via OpenSearch and the creation of a user-friendly interface, highlight the relevance of adopting modular and scalable architecture. This structuring makes it possible to confidently scale up the volume of data or add new features in the future.

However, deploying to a cloud environment encountered budgetary challenges, limiting the scope of our deployments. Despite these constraints, the software infrastructure remains operational and ready to be activated in a professional setting with adequate resources.

Moreover, this project fulfilled its primary objective: enabling skills development across the entire ecosystem (Scrapy, AWS, Lambda, OpenSearch, etc.), configuring production environments, and automating a robust crawler.

Ultimately, this project demonstrates that a well-designed web crawler can serve as the foundation for multiple services: competitive intelligence, internal search engines, analytical applications, and document monitoring platforms.

The key lies in the ability to establish a complete pipeline, from data collection to visualization, and to scale it for large volumes. The various learnings and experiments conducted within this framework will provide a solid foundation for anyone looking to deploy a full-fledged web crawling and data analysis solution in the future.

Appendix

Glossary

AWS (Amazon Web Services) – Amazon's cloud platform for deploying and running online applications.

AWS Lambda – AWS serverless service for executing code in response to events, without a dedicated server.

BeautifulSoup – Python HTML parser for content extraction and tree navigation.

Cluster (OpenSearch/Elasticsearch) – Set of nodes (servers) cooperating to store and process distributed data.

Docker Container – Lightweight, isolated environment for running applications without setting up an entire virtual machine.

EC2 (Amazon Elastic Compute Cloud) – Hosting service for configurable virtual machines on AWS.

Elasticsearch – Lucene-based search engine for fast, full-text queries.

Index (Search Engine) – Inverse structure for quickly finding documents containing a given term.

Instance Spot – Type of EC2 instance at a reduced rate, interrupted if AWS needs the capacity.

JSONL (JSON Lines) – Format where each line is a self-contained JSON object, simplifying the processing of large sets.

Langid – Python library that detects the dominant language of a text.

Middleware (Scrapy) – Intermediate layer that intercepts and modifies requests/responses to handle errors or HTTP headers.

NLTK (Natural Language Toolkit) – Python library dedicated to natural language processing (stopwords filtering, etc...).

OpenSearch – Distributed search and analytics engine born of an Elasticsearch fork.

Pipeline (Scrapy) – Post-processing chain to clean, validate and store extracted data.

robots.txt – File at the root of a website specifying the allowed or restricted crawling rules for bots.

S3 (Amazon Simple Storage Service) – Highly available, massive object storage service on AWS.

SEO (Search Engine Optimization) – Improving a site's ranking on search engines.

Scrapy – Modular, high-performance open-source framework specialized in web scraping.

Stopwords – Common words deemed to have little meaning (e.g. “the”, “and”, “et”), often suppressed in textual analysis.

Systemd – Linux initialization system managing the execution and monitoring of services (automatic startup).

TF-IDF (Term Frequency – Inverse Document Frequency) – Text analysis method that evaluates the importance of a word in a document.

Web Crawler (Spider/Bot) – Automated program that scans the Web and collects information (HTML, images, metadata).

Web Scraping – Extraction of structured data (text, prices, etc.) from the HTML code of web pages.

Sitography

What is a crawler ? - https://semji.com/fr/guide/crawler-un-site-web-methodes-et-techniques/

Scrapy documentation – https://docs.scrapy.org/en/latest/

AWS documentation – https://docs.aws.amazon.com/

systemd documentation – https://systemd.io/

NLTK documentation – https://www.nltk.org/

Scikit-Learn documentation – https://scikit-learn.org/stable/

BeautifulSoup documentation – https://www.crummy.com/software/BeautifulSoup/bs4/doc/

Langid documentation – https://pypi.org/project/langid/

Docker documentation – https://docs.docker.com/

Datadome – https://datadome.co/fr/bot-management-protection/la-liste-complete-des-crawlers-2024-identifiez-tous-les-robots-dindexation-web/

Kinsta – https://kinsta.com/fr/blog/liste-crawler/