

# A Study of Reference Paper Collection System Using Web Scraping and BERT Model

September, 2025

Inzali Naing

Graduate School of  
Natural Science and Technology

(Doctor's Course)  
OKAYAMA UNIVERSITY



Dissertation submitted to  
Graduate School of Natural Science and Technology  
of  
Okayama University  
for  
partial fulfillment of the requirements  
for the degree of  
Doctor of Philosophy in Engineering

Written under the supervision of

Professor Nobuo Funabiki

and co-supervised by

Professor Satoshi Denno

and

Professor Yasuyuki Nogami

OKAYAMA UNIVERSITY, September 2025.



TO WHOM IT MAY CONCERN

We hereby certify that this is a typical copy of the original doctor thesis of  
Ms. Inzali Naing

Signature of  
the Supervisor

Seal of

Prof. Nobuo Funabiki

Graduate School of  
Natural Science and Technology



# Abstract

Title: A Reference Paper Collection System Using Web Scraping and BERT Model

For researchers, collecting relevant papers and data from the internet is very important. However, it is often a time-consuming and challenging task, involving shifting through multiple sources and manually collecting information. As the volume of academic publications grows, this process becomes increasingly inefficient, leaving researchers with limited time to focus on their actual work.

For this problem, automating web searches using web scraping using Selenium can be a good solution. It enables the automation of web interactions, allowing researchers to programmatically navigate websites, search for relevant papers, and collect data at scale. It can automate the retrieval of research articles, abstracts and citations directly from academic websites, eliminating the need for manual browsing. The ability to automate the process not only saves time but also increases the accuracy and consistency of data collection. As a result, researchers can focus more on their analysis and less on the repetitive task of data gathering, making the entire process more efficient and streamlined.

Web scraping is the technique of automatically extracting data from websites, typically by parsing the HTML structure of web pages. It is a valuable tool when structured data is not readily available through APIs or databases, allowing for the collection of large amounts of information directly from the web. The ideation process for web scraping often begins with identifying the target website and understanding the structure of the data to be extracted. This includes analyzing the HTML or JavaScript-generated content to determine the best methods for accessing and parsing the data.

As the first contribution of this thesis, I design a reference paper collection system, the initial step involved identifying the required technologies to collect and validating academic papers from online sources. I selected Selenium as the primary tool for interacting with dynamic websites, enabling automated browsing and data extraction. For detecting broken PDF links, I implemented a validation mechanism that checks the availability of each PDF by attempting to open the link and verifying its response status. If a link returns an error or is inaccessible, it is flagged as broken and excluded from the collection process. To ensure the PDFs retrieved are valid, I integrated a process to verify that the links point to actual downloadable files, not just redirect pages. For organizing the collected references, I chose JSON to store data in a structured format, making it easy to process and analyze later. Additionally, Angular Framework is applied to show the relevant paper list according to the user preference.

A web-based reference paper collection system is designed to support efficient academic research by leveraging web scraping and multithreading techniques. The application extracts research papers along with key metadata such as titles and keywords by letting the user enter the data through text input. User input data is sent via API and performs automation for web scraping with input user data. The backend part validates PDF download links to ensure accessibil-

ity, filtering out broken or restricted files. A similarity scoring algorithm evaluates the relevance of each paper to the user's input query, allowing the system to rank and display results by contextual significance. Performance is optimized through multithreading, enabling concurrent data retrieval and faster response times. The system was implemented using a Python-based backend with Flask and an Angular-based frontend to provide an interactive and responsive user experience. The whole system is containerized by using Docker for easy installation. Evaluation through test scenarios demonstrated reliable data extraction, accurate filtering, and efficient handling of user queries. With its streamlined and user-friendly interface, the web application enables researchers to efficiently discover, prioritize, and collect high-quality academic resources and downloadable content.

In the design and implementation of the reference paper collection system, the first challenge addressed was overcoming anti-scraping mechanisms, particularly from platforms like Google Scholar. To bypass bot detection and improve the system's efficiency, Selenium Stealth was used. This approach helps by mimicking human behaviors, such as mouse movements and scrolling patterns, while rotating user-agent strings to avoid detection as a bot. Selenium Stealth was chosen because it prevents IP blocking, which often happens with traditional scraping methods. In addition, multithreading was considered to speed up the scraping process by enabling concurrent requests for multiple pages. To assess the relevance of the collected papers, the BERT model was employed to calculate similarity scores between titles and abstracts, helping to filter out irrelevant results. Preliminary evaluations of the system showed a significant improvement, with a 35% increase in the success rate of collecting relevant references and a reduction of CAPTCHA challenges by 80%. The system's results were displayed using an Angular frontend, providing an interactive and user-friendly interface for researchers to view and analyze the collected papers.

For future work, several enhancements will be made to the reference paper collection system to improve both functionality and user experience. First, API integration will be incorporated to expand the range of paper sources, allowing the system to gather references from additional academic platforms beyond Google Scholar. This will provide users with access to a broader set of research papers, improving the comprehensiveness of the collection process. The system will also be enhanced to display the source name for each collected paper, allowing users to identify where the reference was sourced from. Additionally, user interface (UI) improvements will be implemented to make the platform more intuitive and interactive, with features like filtering search results based on abstract, keywords, authors, and publication date. This will enable users to refine their search and locate papers that are more aligned with their research interests. By expanding paper sources and improving the UI, the system will offer a more robust and user-friendly tool for researchers, streamlining the process of collecting relevant academic contents.

# Acknowledgements

I would like to express my heartfelt gratitude to all who supported and guided me throughout the completion of this thesis at Okayama University, Japan. To all who have been part of this journey, I can only say that you are the greatest blessing in my life. First and foremost, I owe my deepest gratitude to my honorable supervisor, Professor Nobuo Funabiki, for his excellent supervision, meaningful suggestions, persistent encouragement, and invaluable assistance at every stage of my Ph.D. study. His thoughtful comments and guidance helped me complete my research papers and present them effectively. He was always patient and helpful whenever guidance and assistance were needed, both in my academic pursuits and daily life in Japan. His mentorship has been nothing less than a gift, and it would not have been possible to complete this thesis without his guidance and active support.

I am deeply grateful to all my co-supervisors for their invaluable knowledge and insightful discussions, which greatly enriched my research. I would also like to extend my sincere thanks to Professor Takeuchi Koichi for preparing the AI course, which marked the beginning of my journey into learning about large language models (LLMs) and encoder-decoder architectures.

My heartfelt appreciation goes to my fellow Myanmar classmates, especially Soe Thandar Aung, for her guidance and support with thesis writing, and Khaing Hsu Wai, whose constant encouragement kept me motivated throughout this PhD journey. I am also thankful to San Hay Mar Shwe, Khin Thet Mon, and other Myanmar, Chinese and Indonesian classmates for their warm smiles and kindness, even though I could not spend much time with them as I pursued this PhD entirely online.

A special thanks to my mother, husband, and sister for their unwavering support, encouragement, and understanding. Their love and belief in me have been the foundation of my perseverance and strength during this challenging research journey.

Inzali Naing  
Okayama University, Japan  
September, 2025



# List of Publications

## Journal Paper

1. **Inzali Naing**, Soe Thandar Aung, Khaing Hsu Wai, and Nobuo Funabiki, "A Reference Paper Collection System Using Web Scraping," *Electronics*, vol. 13, no. 14, article 2700, July 2024. DOI: 10.3390/electronics13142700

## International Conference Paper

2. **Inzali Naing**, Nobuo Funabiki, Khaing Hsu Wai, and Soe Thandar Aung, "A design of automatic reference paper collection system using Selenium and Bert Model," in *Proceedings of 2023 IEEE 12th Global Conference on Consumer Electronics (GCCE)*, pp. 267-268, IEEE, 2023. DOI: 10.1109/GCCE59613.2023.10315512

## Other Papers

3. **Inzali Naing**, Nobuo Funabiki, Khaing Hsu Wai, Htoo Htoo Sandi Kyaw, Huiyu Qi, and Veronicha Flasma, "A Design and Implementation of Unit Test Tool for Client-Side Web Programming Learning Assistant System," *IEICE Technical Report*, vol. 122, no. 406, NS2022-235, pp. 390-395, March 2023.
4. **Inzali Naing**, Nobuo Funabiki, Soe Thandar Aung, and Tresna Maulana Fahrudin, "An Preliminary Evaluation of Reference Paper Collection System Using Selenium Stealth for Web Scraping," *IEICE Technical Report*, vol. 124, no. 449, KBSE2024-66, pp. 83-86, March 2025.



# List of Figures

3.1	Reference paper collection system architecture. . . . .	12
3.2	<i>Web scraping</i> data flow. . . . .	17
4.1	Overview of system design. . . . .	21
4.2	<i>Web scraping</i> with <i>Selenium</i> version-4. . . . .	23
4.3	Flow chart of <i>HTML</i> output process. . . . .	24
4.4	Multi-threading for enhanced response time in <i>web scraping</i> . . . . .	27
4.5	Interface for displaying result papers list. . . . .	28
4.6	Source code implementation of backend automation. . . . .	30
8.1	Overview of allure test report. . . . .	69
8.2	Test result in behaviors tab. . . . .	70
8.3	Problem lists of JPLAS. . . . .	71
8.4	Calculator project problem instruction. . . . .	72
8.5	Test scenarios of calculator problem. . . . .	73
8.6	Preview page for calculator problem. . . . .	74
8.7	Server side handling for automation testing. . . . .	74



# List of Tables

5.1	<i>PDF</i> edge cases and handling strategies. . . . .	39
6.1	System performance results. . . . .	44
6.2	Accuracy results in <i>Case 1</i> . . . . .	46
6.3	Accuracy results in <i>Case 2</i> . . . . .	47
6.4	Accuracy results in <i>Case 3</i> . . . . .	47
6.5	Accuracy results in <i>Case 4</i> . . . . .	48
6.6	Accuracy results in <i>Case 5</i> . . . . .	48
6.7	Accuracy results in <i>Case 6</i> . . . . .	49
6.8	Accuracy results in <i>Case 7</i> . . . . .	49
6.9	<i>SUS</i> questions in questionnaire. . . . .	50
6.10	Answers for <i>SUS</i> questionnaire and <i>SUS</i> scores. . . . .	50
6.11	<i>SUS</i> score interpretation. . . . .	51
7.1	Performance index results. . . . .	63



# Contents

<b>Abstract</b>	<b>i</b>
<b>Acknowledgements</b>	<b>iii</b>
<b>List of Publications</b>	<b>iv</b>
<b>List of Figures</b>	<b>v</b>
<b>List of Tables</b>	<b>vi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background for Selenium Study . . . . .	1
1.2 Background for Selenium Stealth Study . . . . .	2
1.3 Contributions . . . . .	2
1.4 Contents of Dissertation . . . . .	3
<b>2 Problem Statement</b>	<b>5</b>
2.1 Background and Motivation . . . . .	5
2.2 Current State of Literature Collection . . . . .	6
2.3 Current Limitations and Challenges . . . . .	6
2.3.1 Manual Collection Inefficiencies . . . . .	6
2.3.2 Scholarly Databases . . . . .	7
2.3.3 Access Restrictions and Paywalls . . . . .	7
2.3.4 Inadequate Relevance Assessment . . . . .	8
2.3.5 Workflow Integration Challenges . . . . .	8
2.3.6 Big Data Collection . . . . .	8
2.4 Proposed Solution Scope . . . . .	9
2.4.1 Significance and Impact . . . . .	10
<b>3 System Design and Architecture</b>	<b>11</b>
3.1 System Overview . . . . .	11
3.2 System Architecture . . . . .	12
3.2.1 Overall Architecture Design . . . . .	12
3.2.2 Component Architecture . . . . .	13
3.3 Core System Components . . . . .	13
3.3.1 Selenium as Web Scraping Engine . . . . .	13
3.3.2 BERT as Natural Language Processing Module . . . . .	14
3.3.3 Python Flask as Web Application Server . . . . .	15

3.3.4	Angular as User Interface Presentation . . . . .	15
3.4	Data Flow Architecture . . . . .	16
3.4.1	Input Processing Flow . . . . .	16
3.4.2	Web Scraping Data Flow . . . . .	16
3.4.3	NLP Processing Flow . . . . .	17
3.4.4	Output Generation Flow . . . . .	17
3.5	Technology Stack Selection . . . . .	18
3.5.1	Backend Technologies . . . . .	18
3.5.2	Frontend Technologies . . . . .	18
3.5.3	Development and Deployment Tools . . . . .	19
3.6	Summary . . . . .	19
<b>4</b>	<b>System Implementation</b>	<b>21</b>
4.1	Software Architecture . . . . .	21
4.1.1	Client-Side Implementation . . . . .	22
4.2	Connection between Server-Side and Client-Side . . . . .	22
4.3	Server-Side Implementation . . . . .	23
4.3.1	HTML Output Processing with Automation Scripts . . . . .	23
4.3.2	Pagination Handling . . . . .	27
4.3.3	Concurrent Processing . . . . .	27
4.3.4	User Interface for Displaying Result Papers List . . . . .	28
4.3.5	Handling Web Scraping Module . . . . .	28
4.4	Paper Filtering Relevance . . . . .	29
4.5	Relevance Implementation . . . . .	29
4.5.1	Keyword Matching and Similarity Score Calculation . . . . .	29
4.5.2	Source Code Implementation for Backend Automation . . . . .	30
4.6	Similarity Computation Methodology . . . . .	31
4.6.1	Document Processing and Metadata Extraction . . . . .	31
4.6.2	Semantic Similarity Computation . . . . .	31
4.6.3	Error Handling and Robustness . . . . .	31
4.6.4	Similarity Score Interpretation . . . . .	32
4.7	Chapter Summary . . . . .	32
<b>5</b>	<b>System Enhancements</b>	<b>33</b>
5.1	Introduction . . . . .	33
5.2	Multithreading Enhancement Architecture . . . . .	33
5.2.1	Threading Model Design . . . . .	33
5.2.2	Parallel Processing Implementation . . . . .	34
5.3	Docker Containerization Strategy . . . . .	35
5.3.1	Container Architecture Design . . . . .	35
5.3.2	Docker Configuration . . . . .	35
5.3.3	Frontend Container Configuration . . . . .	37
5.4	PDF File Handling and Edge Case Management . . . . .	38
5.4.1	Robust PDF Processing Framework . . . . .	38
5.4.2	Edge Case Handling Mechanisms . . . . .	39
5.4.3	Multi-Library PDF Processing Pipeline . . . . .	39
5.4.4	Content Quality Validation . . . . .	40

5.5	Error Handling and Reliability . . . . .	40
5.5.1	Comprehensive Error Management . . . . .	40
5.6	Chapter Summary . . . . .	41
<b>6</b>	<b>Performance Evaluation</b>	<b>43</b>
6.1	Introduction . . . . .	43
6.2	Evaluation Framework and Methodology . . . . .	43
6.2.1	Experimental Setup . . . . .	43
6.2.2	Performance Metrics Definition . . . . .	44
6.2.3	Accuracy Evaluation . . . . .	45
6.2.3.1	Case 1: Web Scraping with Selenium . . . . .	46
6.2.3.2	Case 2: Sentence Transformers . . . . .	46
6.2.3.3	Case 3: Mobile UI Testing . . . . .	47
6.2.3.4	Case 4: Indoor Navigation Systems . . . . .	48
6.2.3.5	Case 5: Docker Technology . . . . .	48
6.2.3.6	Case 6: IoT Platform Research . . . . .	48
6.2.3.7	Case 7: Web Programming Education . . . . .	49
6.2.4	System Usability Evaluation . . . . .	49
6.2.5	Feedback from Students . . . . .	51
6.2.6	Comparative Analysis Summary . . . . .	52
6.3	Performance Analysis and Discussion . . . . .	52
6.3.1	Multi-threading Performance Gains . . . . .	52
6.3.2	Accuracy and Relevance Improvements . . . . .	53
6.3.3	Usability Considerations . . . . .	53
6.3.4	Limitations and Future Improvements . . . . .	53
6.4	Chapter Summary . . . . .	53
<b>7</b>	<b>Improved Version with Selenium Stealth</b>	<b>55</b>
7.1	Introduction . . . . .	55
7.1.1	Challenges and Solutions . . . . .	56
7.2	Human-Bot Detection and Solutions . . . . .	56
7.2.1	CAPTCHA . . . . .	56
7.2.2	Residential IP Proxies . . . . .	57
7.2.3	AI Bots . . . . .	57
7.2.4	Scrapers and Defenders . . . . .	58
7.3	Selenium Stealth Features . . . . .	58
7.3.1	Browser Fingerprinting Mitigation . . . . .	58
7.3.2	Headers and Cookies Management . . . . .	59
7.3.3	Proxy and IP Rotations . . . . .	59
7.3.4	Anti-CAPTCHA Integration . . . . .	60
7.4	Configuration of Selenium Stealth . . . . .	60
7.5	Preliminary Evaluation . . . . .	62
7.5.1	Evaluation Indices . . . . .	62
7.5.2	Results . . . . .	62
7.6	Conclusion . . . . .	63

<b>8</b>	<b>Automated Testing with Selenium</b>	<b>65</b>
8.1	Introduction . . . . .	65
8.2	Design of Unit Test Tool for WebPLAS . . . . .	66
8.2.1	Python Flask . . . . .	66
8.2.2	Jinja2 . . . . .	67
8.2.3	Selenium . . . . .	67
8.2.4	Behave Framework . . . . .	68
8.2.5	Allure Report . . . . .	69
8.3	Implementation of Unit Test Tool . . . . .	70
8.3.1	File Structure of Unit Test Tool . . . . .	70
8.3.2	System Flow Demonstration . . . . .	71
8.3.2.1	Problem lists and Requirements . . . . .	71
8.3.2.2	Server Side Handling . . . . .	72
8.3.2.3	Allure Test Result Report . . . . .	73
8.3.2.4	Unit Test tool Installation . . . . .	74
8.4	Conclusion . . . . .	75
<b>9</b>	<b>Related Works in Literature</b>	<b>77</b>
9.1	Automated Reference Paper Collection Systems . . . . .	77
9.2	Web Scraping Technologies for Academic Applications . . . . .	78
9.3	BERT and Natural Language Processing for Academic Text Analysis . . . . .	78
9.4	Academic Search Engines and Literature Management Tools . . . . .	79
9.5	Evaluation and Quality Assessment in Academic Systems . . . . .	80
9.6	Similarity Network Analysis and Document Clustering . . . . .	80
<b>10</b>	<b>Conclusion and Future Work</b>	<b>81</b>
	<b>References</b>	<b>84</b>

# Chapter 1

## Introduction

### 1.1 Background for Selenium Study

*Web Scraping* is the process of automatically collecting information from websites using computer programs. Instead of manually copying and pasting data from web pages, *Web Scraping* tools can visit websites, read the contents, and save the information in an organized format like a spreadsheet or database. *Web Scraping* has become essential because it allows people and businesses to access large amounts of information quickly and efficiently [1]. Instead of spending hours manually collecting data, automated tools can do the same work in minutes. This capability helps researchers, businesses, and students make better decisions based on comprehensive, up-to-date information. *Selenium* operates through a seamless process that mimics human web browsing behaviors [2]. The program begins by launching a web browser such as *Chrome*, *Firefox*, or *Safari*. Then it navigates to specific web addresses like clicking on links. Once reaching the desired webpage, *Selenium* can interact with various elements by clicking buttons, typing in text boxes, selecting drop-down options, and scrolling through pages [3]. During this process, *Selenium* can extract information such as text, prices, or any visible content from web pages while simultaneously performing actions like submitting forms, downloading files, or taking screenshots. Finally, when all tasks are completed, the program automatically closes the browser, leaving behind the collected data or completed actions [4]. *Google Scholar* is a specialized search engine designed specifically to find academic and scholarly information on the Internet [5, 6]. Unlike regular *Google* search engine that shows all types of websites, *Google Scholar* focuses exclusively on research papers, academic articles, books, conference papers, thesis, and other scholarly literature. People often rely on *Google Scholar* to find credible sources for research papers, essays, and thesis projects, ensuring their work is built on solid academic foundations. Researchers use the platform to stay current with developments in their fields. Accessing to research papers in the Internet is a crucial task for researchers. It helps them build upon existing knowledge and develop their own scholarly works. However, depending solely on a common platform like *Google Scholar* may present challenges. It can be found that many papers are inaccessible due to payment barriers, access restrictions or broken links. Even when researchers can obtain the papers, they often need to spend a lot of time verifying their credibility and relevance manually. This manual checking process is a very time-consuming task and can slow down the research progress. Therefore, a better method is necessary to find and evaluate relevant research papers automatically, which would be beneficial for a lot of researchers by making their research more efficient and less time-consuming. In this thesis, I propose a *reference paper collection system* using *web scraping* technology [7]. It involves utilizing innovative technologies within the framework of interactive web services. The

system employs the *BERT* model, a *natural language processing (NLP)* technique from the Hugging Face library [8], to refine search results and enhance the accuracy and relevance in extracting references. *Selenium Web Driver* plays a crucial role in controlling web pages on a browser and collecting data from them. The system comprises client-side (front end) and server-side (back end) components. The entire architecture is maintained and managed as a repository using *Git* version control and is containerized via *Docker*. The front end, serving as the web user interface, is built using *Angular*. The back end, driven by *Python Flask*, functions as the web application server. Furthermore, the adoption of the design architecture prioritizes generalizing the functionality, features, and common technology usages to optimize the running time, the memory utilization, and the performance of the central processing unit (CPU).

## 1.2 Background for Selenium Stealth Study

Nowadays, scholarly databases are found in many platforms that will significantly help researchers in continuous learning and researching. I study a reference paper collection system which uses the *Google Scholar*. This system provides open access and relevant papers, reducing the manual reading time spent on *Google Scholar* with the help of *web scraping* and *AI* technology. However, this initial study identified the limitations on accessibility to *Google Scholar* when the system was accessed repeatedly in a short time-frame. *Selenium* is a widely used tool for automating web browsers, especially for tasks such as web testing and *web scraping*. However, it comes with several limitations when used in *web scraping*, particularly when dealing with modern web technologies that deploy *anti-scraping mechanisms* [9]. *Selenium* leaves a certain signature of the user agent string setting default as a bot. It does not have human behaviors such as mouse movements, key-press delays. Besides, traditional *Selenium* does not support rotating proxies or IP managements. Excessive scraping can lead some websites to block their IP addresses due to repeated requests. *Selenium Stealth* is an improved version of *Selenium* that can bypass the anti-bot measures implemented on modern websites. It allows users to conduct *web scraping* and automations by reducing the detectability of automated browsers. It acts as human behaviors of browser scrolling patterns and mouse clicking events, and rotating user agent strings. Due to the fact that the reference paper collection system is sometimes prone to *anti-web scraping* mechanism, a better approach is needed to address this problem, allowing researchers to use the system smoothly and repeatedly according to their use cases. We need to find a better tool which makes the root cause when it comes to avoiding anti-scraping mechanisms.

## 1.3 Contributions

Motivated by the above-mentioned problems, this thesis presents the *reference paper collection system* using *web scraping* that provides the efficient way for researchers to finding related papers. It also presents the enhanced version of the system using *Selenium Stealth* that improves the automation process for the system by avoiding anti-scraping mechanisms. As the first contribution of the thesis, I present a design of the web-based automatic reference paper collection system to address the challenges associated with the manual extraction of references for researchers to enhance research activities. As the input data, the system accepts the thesis title and the keywords that describe the original work for the target research project. Then, the system automatically lists the downloadable *PDF* file links for relevant papers to them. As the second contribution of the thesis, I implement the reference paper collection system [10]. This system can collect or monitor data

from the Internet, which is considered as the environment, using *Selenium*, a popular *web scraping* software, as the sensor; this examines the similarity against the search target by comparing the keywords using the *BERT model*. The *BERT model* is a deep learning model for *natural language processing (NLP)* that can understand context by analyzing the relationships between words in a sentence bidirectionally [11]. *Python Flask* is adopted at the web application server, where *Angular* is used for data presentations. As the third contribution of the thesis, I implement the enhanced version of the reference paper collection system using *Selenium Stealth*[12]. It uses a new technology to mimic human behavior to access the *Google Scholar*. A new library called *Selenium Stealth* is adopted to overcome the restrictions. The system uses different browsers every time it scrapes, and different fingerprints method not to be recognized as an automation process. The system acts human behaviors of browser scrolling patterns, mouse clicking events, and rotating the user agent strings. The performance of this enhanced version is compared with our previous approach with *Selenium*, which leads to IP address blocking when the *Google Scholar* server notices the system as a malicious activity. With the two scraping methods, we confirm the accessibility increase when using the new library called *Selenium Stealth*. As the additional contribution of this thesis, I present that *Selenium* is used as a testing tool for web application by applying the ability to control the web elements automatically. I present a design and implementation of a unit testing tool of client-side programs using *HTML*, *CSS*, and *JavaScript* for the code writing problem (CWP) in web programming learning assistant system (*WebPLAS*)[13]. It adopts *Python Selenium* for web-driver, *Python-Behave* for test cases and scenarios, and *Allure* for reporting test results. *Python Flask* is used for servers. For deployments of the tool to students, *Docker* is used to cooperate and make them the system [14].

## 1.4 Contents of Dissertation

The remaining part of this thesis is organized as follows. Chapter 2 describes existing paper collection methodologies in literature, *web scraping* technologies including *Selenium* framework, and related automated academic information retrieval systems.

Chapter 3 presents the overall system architecture and design of the reference paper collection system, including core components, data flow mechanisms, and technology stack selections.

Chapter 4 presents the implementation of *web scraping* functionality using *Selenium* automation, covering website navigation strategies, *PDF* download mechanisms, and content extraction algorithms.

Chapter 5 proposes the system enhancements including multi-threading implementation, *Docker* containerization, and robust *PDF* file handling mechanisms for the reference paper collection system, focusing on performance optimization and deployment flexibility.

Chapter 6 presents experimental evaluations of system performances, including efficiency metrics, accuracy assessment, and comparative analysis with manual collection methods.

Chapter 7 reviews previous works related to the automated reference paper collection system and presents improvements in this thesis.

Chapter 8 presents that *Selenium* can be used as the automated web testing tool and reviews how it can cooperate with *Allure* and *Behave* frameworks to create an automatic testing in the web-based programming learning assistant system.

Chapter 9 presents paper works related to this thesis.

Chapter 10 concludes this thesis with some future works.



# Chapter 2

## Problem Statement

### 2.1 Background and Motivation

The landscape of academic researches has undergone dramatic and significant transformation in recent decades. The volume of scholarly publications has experienced unprecedented growth that continues to accelerate year after year. According to recent estimates and statistical analyses, approximately 2.5 million new research papers are published annually across all academic disciplines worldwide [15]. This represents a compound annual growth rate of 4-5% over the past two decades, which means the number of published papers doubles approximately every 15-20 years. This exponential increase in scholarly output, while indicative of vibrant and active research activity globally, has created substantial and complex challenges for researchers who are attempting to maintain comprehensive awareness of new developments and advances in their specific fields of study. The traditional model of literature discovery has evolved significantly from its historical foundations. In the past, this model relied heavily on manual searching through physical library catalogs and printed academic journals that were stored in university libraries and research institutions. This traditional approach has gradually evolved to encompass modern digital academic databases and sophisticated online search engines that provide instant access to millions of scholarly documents. *Google Scholar*, which was launched in 2004 as a specialized academic search engine, now indexes over 389 million scholarly documents from universities, publishers, and research institutions worldwide [16]. Meanwhile, specialized academic databases such as *Web of Science*, *Scopus* [17], *IEEE Xplore* [18], and *PubMed* contain millions of additional records that cover specific disciplines and research areas. However, this abundance of available literature and the vast amount of accessible information has paradoxically made the task of identifying relevant and useful research more complex and challenging rather than simpler and more straightforward. Contemporary researchers now face the significant challenges of navigating an increasingly fragmented and complex information landscape. In this landscape, relevant papers may be distributed across multiple platforms and databases, published in diverse formats ranging from traditional journal articles to conference proceedings, and scattered throughout numerous specialized repositories that serve different academic communities. The emergence of open access publishing initiatives, preprint servers like *arXiv* and *bioRxiv*, institutional repositories maintained by universities, and discipline-specific databases has further complicated and intensified the search process [19]. They request researchers to develop sophisticated and comprehensive search strategies across multiple platforms and databases to achieve comprehensive coverage of their research topics. Furthermore, the interdisciplinary nature of modern research has significantly intensified these challenges and created additional complexity. Many contemporary research problems and scientific questions re-

quire synthesis of knowledge from multiple academic disciplines, each with its own publication venues, specialized terminology, research methodologies, and established research traditions. A researcher working on *artificial intelligence (AI)* applications in healthcare, for example, must navigate and synthesize literature from computer science, medicine, bioengineering, health informatics, and ethics. Each of these disciplines is published in different academic venues with varying indexing systems, search mechanisms, and access requirements that make comprehensive literature collection extremely challenging.

## 2.2 Current State of Literature Collection

Despite significant technological advances in information retrieval systems and digital library technologies, the process of academic literature collection remains largely manual, labor-intensive, and time-consuming for most researchers [6]. Researchers typically employ ad-hoc and inconsistent search strategies that vary from person to person. These strategies usually involve combining keyword searches across multiple academic databases with citation chasing techniques, following reference lists, and seeking expert recommendations from colleagues and supervisors. This traditional approach, while familiar and comfortable for many researchers, may suffer from several systematic limitations and inherent weaknesses that significantly impact both research quality and efficiency in academic work. The current state of literature collections is characterized by fragmentation and inconsistency across different research communities and academic disciplines. Most researchers develop their own personal search strategies through trial and error, without formal training in systematic literature search methodologies [20]. This leads to significant variations in search effectiveness and completeness among researchers, even when working on similar research topics. The lack of standardized approaches also makes it difficult to reproduce literature searches and verify the comprehensiveness of research coverage.

## 2.3 Current Limitations and Challenges

### 2.3.1 Manual Collection Inefficiencies

The traditional approach to academic paper collections relies heavily and almost exclusively on manual searching and evaluation processes. These processes are both extremely time-consuming and prone to incompleteness and human error. Researchers typically spend considerable amounts of time navigating through extensive search results, manually evaluating relevance of individual papers, checking accessibility of full-text documents, and managing complex citation databases. This manual process becomes increasingly inefficient and overwhelming when conducting large-scale literature reviews or systematic reviews that require comprehensive and exhaustive coverage of an entire research domain or scientific field. Multiple studies and surveys indicate that researchers spend up to 30-40% of their total research time on literature discovery and collection activities alone [21]. This represents a significant portion of valuable research time that could be better allocated to more productive activities such as data analysis, result interpretation, hypothesis development, and knowledge synthesis. The manual nature of current literature collection also leads to inconsistent results between different researchers, as personal biases, search experience, and time constraints affect the comprehensiveness and quality of literature collections. The inefficiency of manual collection is particularly pronounced when researchers need to update their

literature reviews or when working on interdisciplinary projects that span multiple research domains. The repetitive nature of manual searching across multiple databases, combined with the need to continuously monitor new publications, creates a significant burden that detracts from core research activities.

### 2.3.2 Scholarly Databases

*Google Scholar* [16] is widely recognized as the most popular and comprehensive scholarly website that includes an extensive range of research materials. These materials include research articles from peer-reviewed journals, academic theses and dissertations, books published by academic publishers, conference papers from scientific meetings, and patents from various patent offices worldwide. The user interface of *Google Scholar* is designed to be intuitive and user-friendly, making it relatively easy for users to search for and access scholarly content across multiple disciplines and research areas [6]. The citation information provided by *Google Scholar* can help researchers evaluate the reputation, impact, and popularity of specific papers by showing how many times each work has been cited by other researchers. A user can initiate a new search by entering relevant keywords and search terms into the main search box on the *Google Scholar* homepage. Articles and publications can be filtered and refined by author name, publication year, and other criteria using the advanced search settings and options. The search process will yield results in the form of a comprehensive list of articles and publications, which usually spans more than 10 pages of results and can contain hundreds or thousands of potentially relevant documents. Other important scholarly databases like the Institute of Electrical and Electronics Engineers (*IEEE Xplore*) [18] and *Scopus* [17] primarily focus on technical and scientific papers, conference proceedings, and academic publications in engineering and scientific disciplines. *IEEE Xplore* specifically covers electrical engineering, computer science, and related technical fields, while *Scopus* provides broader coverage across scientific, technical, medical, and social science disciplines. These specialized databases often provide more detailed indexing and advanced search capabilities within their specific domains. The wide range and vast number of search results on *Google Scholar* presents certain significant challenges and limitations for researchers. Navigating through numerous suggested web pages and result listings to find truly relevant and useful articles can be extremely time-consuming and mentally exhausting. To properly ascertain an article's relevance to their research, a user must open individual links to access the full *PDF* documents or article abstracts, which requires additional time and efforts. Besides, there are many search results where *PDF* links are broken, web pages are not available, or access is restricted by paywalls and subscription requirements. Nevertheless, the convenience and ease of obtaining initial search results with general keywords and search terms remains a noticeable and significant advantage for *web scraping* applications and automated data collection in this research domain.

### 2.3.3 Access Restrictions and Paywalls

A significant and problematic portion of scholarly literature remains behind publisher paywalls and subscription barriers, severely limiting researchers' ability to access and properly evaluate paper content and research findings [22]. Even when relevant papers are successfully identified through academic search engines and databases, researchers frequently encounter various types of access restrictions that prevent them from obtaining full-text documents that are essential for their research. These restrictions include subscription-only access, institutional paywalls, geographical restrictions, and embargo periods that delay open access availability. Current academic search

systems provide very limited assistance in identifying freely accessible versions of restricted papers or locating alternative sources for the same research content. Researchers often must manually search through multiple repositories, preprint servers, and institutional websites to find open access versions of papers they need [23]. This process is time-consuming and often unsuccessful, leading to incomplete literature reviews and potential gaps in research coverage. The financial burden of accessing scholarly literature is particularly challenging for researchers at resource-constrained institutions, independent researchers, and those in developing countries. Article processing charges and subscription fees can cost thousands of dollars annually, making comprehensive literature access economically unfeasible for many researchers.

### 2.3.4 Inadequate Relevance Assessment

Traditional keyword-based search approaches employed by most academic databases often miss semantically related papers and can fail to capture the nuanced and complex relationships between different research concepts and ideas [24]. The heavy reliance on exact keyword matching and boolean search operators results in low recall rates and the systematic exclusion of papers that use alternative terminology, different conceptual frameworks, or approach similar research problems from different theoretical perspectives. Current academic search systems lack sophisticated semantic understanding capabilities and *natural language processing (NLP)* features that could significantly improve both the precision and recall of search results [11]. The inability to understand synonyms, related concepts, and contextual relationships means that researchers may miss important relevant literature simply because it uses different vocabulary or theoretical approaches. The problem of inadequate relevance assessment is compounded by the interdisciplinary nature of modern research, where concepts from one field may be expressed using completely different terminology in another field. Traditional search systems cannot bridge these terminological gaps, leading to fragmented and incomplete literature collections [20].

### 2.3.5 Workflow Integration Challenges

Many existing automated academic tools and software systems fail to integrate effectively with researchers' established workflows and research practices, resulting in low adoption rates and limited practical impacts in real research environments [25]. The lack of user-centered design principles in academic software development has led to creations of systems that, while technically sophisticated and feature-rich, do not adequately address the real-world needs, constraints, and working habits of practicing researchers. This disconnection between system capabilities and actual user requirements represents a significant barrier to the widespread adoption of automated research tools and technologies. Researchers often find that new tools require substantial changes to their established workflows, involve steep learning curves, or fail to integrate with existing reference management systems and research databases they already use. The workflow integration problem is particularly acute in collaborative research environments where team members may use different tools and systems, making it difficult to maintain consistency and share resources effectively.

### 2.3.6 Big Data Collection

*Web scraping* [1] is defined as the automated information extraction process from websites using computer programs and specialized software tools. It is a sophisticated technique for collecting large amounts of structured and unstructured data from the Internet efficiently and systematically

[26]. Such collected data can be referred to as *Big Data* [15] due to its volume, variety, and velocity characteristics. In the specific context of *Big Data* applications [27], *web scraping* revolves around systematically navigating websites, methodically gathering large volumes of information, and converting the unstructured data commonly found in *hyper text markup language (HTML)* format on web pages into organized, structured data that is suitable for storage, analysis, and further processing [28]. *Web scraping* includes several step-by-step operations and processes such as data collection from target websites, data extraction using parsing algorithms, data transformation into usable formats, and data loading into storage systems or databases. These operations must be carefully coordinated to ensure efficient and reliable data collection processes. Data collections and data extractions can be easily performed by modern *web scraping* tools, frameworks, and custom scripts developed in programming languages like *Python*, *JavaScript*, and *Java* [29]. However, significant challenges arise in handling various technical issues and complications, such as web server maintenances, changes in website structure, or situations when web pages are deleted, moved, or modified after a certain period of time. Real-time data collections through *web scraping* must include robust error-handling mechanisms and fault-tolerant designs that are able to respond effectively to unexpected conditions, server timeouts, network failures, and website changes. Additionally, before initiating any *web scraping* activities, it is crucial and legally important to carefully observe and comply with the privacy policies, terms of service, and *robots.txt* files of target websites [9]. Defining the specific types of data to be scraped in advance helps to minimize unnecessary data retrievals and network traffics, thereby optimizing efficiency and reducing the computational and bandwidth resources required for data collection operations. The scraped data, which is often unstructured or semi-structured in its raw form, requires subsequent cleaning, validation, and structuring processes in data preprocessing and cleaning phases. They involve removing duplicate entries, handling missing values, standardizing data formats, and ensuring data quality and consistency. Data transformation occurs after data extraction and cleaning, involving the conversion of obtained data into various standardized formats such as *JavaScript Object Notation (JSON)* with key-value pairs [30], *Extensible Markup Language (XML)* with hierarchical structures, or structured arrays and data tables. These formatted datasets can then be stored efficiently in file systems, relational databases, or specialized data warehouses. Subsequently, the loaded and processed data are integrated into comprehensive data management systems and warehouses, facilitating further research activities, advanced analytics, or addressing specific business intelligence needs and requirements.

## 2.4 Proposed Solution Scope

To address these numerous challenges and limitations comprehensively, this research proposes a comprehensive and sophisticated reference paper collection system that integrates advanced *web scraping* technologies, semantic relevance assessment capabilities, and user-centered design principles [19]. The system aims to provide a complete solution that addresses the major pain points identified in current literature collection practices. The system aims to:

- Adopt and implement *Selenium* web automation technology [31] to systematically scrape academic papers while simultaneously validating their accessibility and relevance
- Employ advanced *BERT*-based semantic analysis and *natural language processing (NLP)* techniques [8] for significantly improved relevance assessment and content understanding

- Offer easy to use capabilities with existing research workflows, reference management systems
- Ensure reliable paper outputs according to the user's input and preferences while checking relevance

The proposed solution scope encompasses both technical innovations and practical usability considerations. The system will be designed to handle the complexity and scale of modern academic literature while remaining accessible and user-friendly for researchers with varying levels of technical expertise.

### 2.4.1 Significance and Impact

The successful development and implementation of this comprehensive system would significantly reduce the time, efforts, and resources required for academic literature collections and managements [25]. This reduction would enable researchers to focus much more of their valuable time and intellectual energy on higher-level activities such as data analysis, result interpretation, hypothesis development, and knowledge synthesis rather than on the tedious and time-consuming tasks of literature discovery and retrieval. This automated system would be particularly beneficial and valuable for several important categories of researchers and research activities:

- **Early-career researchers and graduate students** who may lack extensive experience in efficient literature search strategies, database navigation skills, and may not have access to comprehensive training in systematic literature review methodologies.
- **Interdisciplinary research projects and collaborative studies** that require comprehensive coverage across multiple academic domains, each with different publication venues, terminologies, and research traditions
- **Systematic literature reviews and meta-analyses** that demand exhaustive, reproducible, and methodologically rigorous search methodologies with complete documentation of search strategies and results [6].
- **Resource-constrained institutions and researchers** with limited financial access to expensive subscription-based academic databases, commercial research tools, and premium academic services.

The proposed system represents a significant technological advancement toward more efficient, comprehensive, accessible, and democratized academic information retrieval. It has the potential to fundamentally transform how researchers approach literature discovery, collection, and management in the digital age, making high-quality research more accessible to researchers worldwide regardless of their institutional resources or geographical locations.

# Chapter 3

## System Design and Architecture

This chapter presents the overall system architecture and comprehensive design of the reference paper collection system. It includes detailed descriptions of core components, data flow mechanisms, and technology stack selection processes. The system is specifically designed to automate the complete process of collecting relevant academic papers from *Google Scholar* using advanced *web scraping* techniques [21], *natural language processing (NLP)* for relevance assessment, and a user-friendly web interface for researchers and academic professionals.

### 3.1 System Overview

The reference paper collection system is designed and developed as a comprehensive web-based application that automates the discovery, collection, and filtering of academic papers relevant to a given research topic or academic field. The system operates by taking an user input in the form of paper titles and research keywords, then automatically searches *Google Scholar* using sophisticated algorithms, extracts relevant articles using *web scraping* techniques [26], evaluates their relevance using advanced *natural language processing* techniques, and presents the results through an intuitive and user-friendly web interface. The primary objectives of the system design include maximizing automations to significantly reduce manual efforts required by researchers, ensuring high accuracy in paper relevance assessment through advanced algorithms, providing a user-friendly interface that is accessible to researchers with varying technical backgrounds, and maintaining robust performances under various operating conditions and system loads. The system architecture follows a modular design approach that carefully separates concerns between data collection, processing, and presentation layers to ensure long-term maintainability and future scalability. The system directly addresses the key challenges and difficulties faced by researchers in manual paper collection processes. These challenges include time-consuming search processes that can take hours or days, difficulty in accurately assessing paper relevances without reading full documents, and the overwhelming need to handle large volumes of search results from multiple academic databases. By integrating advanced *web scraping* capabilities with intelligent filtering mechanisms and semantic analysis, the system provides a comprehensive and efficient solution for automated academic paper collection that saves significant time and effort for researchers. The system operates on the principle of intelligent automations, where complex tasks that traditionally require human judgment are enhanced by machine learning algorithms, while still maintaining human oversight and control. This approach ensures that the automated system can handle routine tasks efficiently while allowing researchers to focus on higher-level analysis and decision-making

activities.

## 3.2 System Architecture

The reference paper collection system follows a well-established three-tier architecture consisting of the presentation layer (*UI layer*), application layer (backend handling), and data access layer (*web scraping* and automation). This architectural pattern provides clear separation of concerns, facilitates system maintenance and updates, and enables independent scaling of different system components based on performance requirements. In this section, I present the detailed system flow in the design process. Figure 3.1 illustrates the comprehensive overview of the system architecture and component interactions.

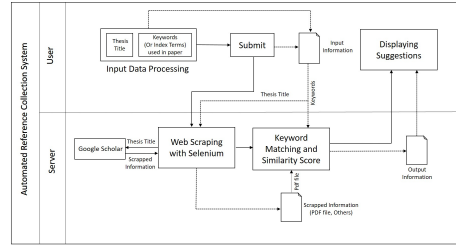


Figure 3.1: Reference paper collection system architecture.

### 3.2.1 Overall Architecture Design

The system architecture is built on a robust client-server model where the frontend client communicates with the backend server through well-defined *API* endpoints. The frontend component handles all user interactions and data visualization tasks, while the backend component manages complex data processing operations, *web scraping* operations, and *natural language processing (NLP)* tasks that require significant computational resources. The architecture incorporates several key design principles that guide the development and implementation process. These principles include modularity for independent component development and testing, scalability to handle varying loads and data volumes efficiently, reliability through comprehensive error handling and recovery mechanisms, and security to protect sensitive user data and system resources from unauthorized access. These fundamental principles guide the selection and integration of all system components and ensure consistent quality throughout the system. The system employs asynchronous processing techniques for *web scraping* operations to prevent blocking the user interface during long-running tasks and data collection operations. This approach ensures a responsive and interactive user experience while maintaining optimal system performance during intensive data collection operations that may take several minutes to complete. The asynchronous design also allows multiple users to access the system simultaneously without experiencing performance degradation. The overall architecture design emphasizes flexibility and extensibility, allowing future enhancements and modifications without requiring major system redesign. This approach ensures that the system can adapt to changing requirements in academic research workflows and incorporate new technologies as they become available.

### 3.2.2 Component Architecture

The system consists of four main components that work together seamlessly to provide comprehensive paper collection functionality and ensure optimal performance. The *web scraping* engine handles automated data extraction from *Google Scholar* using the *Selenium WebDriver* [31] with sophisticated browser automation capabilities. The *NLP* processing module evaluates paper relevance using advanced *BERT* model implementations and semantic analysis techniques. The web application server manages *API* endpoints and complex business logic through the *Flask* framework with robust request handling. The user interface provides interactive access to system functionality through an *Angular*-based web application with modern user experience features. Each component is carefully designed with specific responsibilities and well-defined interfaces to ensure loose coupling and high cohesion throughout the system architecture. The *web scraping* engine encapsulates all browser automation logics and provides clean, standardized interfaces for search execution and result extraction processes. The *NLP* processing module handles all *natural language processing* tasks including text preprocessing, feature extraction, semantic analysis, and relevance scoring using state-of-the-art algorithms. The modular design approach allows independent testing, deployment, and scaling of individual components without affecting other parts of the system. This approach facilitates system maintenance and debugging activities and enables selective optimization of performance-critical components without requiring changes to the entire system architecture. The component-based design also supports parallel developments by different team members and enables easier troubleshooting when issues arise. Each component communicates through standardized interfaces and data formats, ensuring compatibility and enabling easy replacement or upgrading of individual components as new technologies become available. This design approach provides long-term sustainability and adaptability for the system.

## 3.3 Core System Components

The reference paper collection system comprises several core components, each specifically designed to handle particular aspects of the paper collection and processing workflow with optimal efficiency and reliability.

### 3.3.1 Selenium as Web Scraping Engine

The *web scraping* engine serves as the primary data collection component of the entire system, responsible for automating complex interactions with *Google Scholar* and extracting relevant paper information with high accuracy and reliability. The engine utilizes *Python Selenium WebDriver* [2] to control web browsers programmatically, enabling automated navigation, search execution, and data extraction from dynamic web pages that contain *JavaScript*-generated content. The scraping engine implements several key features and capabilities including automated browser management with support for multiple browser types such as *Chrome*, *Firefox*, and *Edge*, intelligent page navigation with sophisticated handling of pagination and dynamic content loading, robust data extraction with comprehensive error handling and retry mechanisms, and intelligent rate limiting to prevent overwhelming target websites. The engine follows a structured and systematic approach to data extraction that ensures consistency and reliability. The process begins with browser initialization and configuration with appropriate settings, followed by navigation to *Google Scholar* search interface using optimized paths, execution of search queries based on user input with proper formatting, systematic extraction of paper metadata including titles, authors, abstracts, publication

dates, and citation information, and finally, data validation and formatting for downstream processing by other system components. Comprehensive error handling mechanisms are integrated throughout the scraping process to handle common issues and unexpected situations such as network timeouts, page loading failures, changes in website structure, and temporary server unavailability. The engine implements exponential back-off strategies for retry operations and graceful degradation when encountering persistent errors that cannot be resolved automatically. The scraping engine also includes advanced features such as user agent rotation to avoid detection, intelligent waiting mechanisms to handle dynamic content loading, and adaptive timing to mimic human browsing behavior. These features ensure reliable operations even when target websites implement anti-scraping measures.

### 3.3.2 BERT as Natural Language Processing Module

*BERT (Bidirectional Encoder Representations from Transformers)* [8] is a sophisticated transformer model developed by *Google* that represents a significant advancement in *natural language processing* technology. Prior models such as *Word2Vec* [32], *GloVe* [33], or *FastText* [34] are one-directional models, which can only read and process the input sequence from left to right or right to left sequentially, limiting their understanding of context and relationships between words. On the other hand, *BERT* has the revolutionary ability to simultaneously read and analyze the entire context of words bidirectionally, meaning it can understand relationships between words regardless of their positions in the sentence. Self-attention mechanisms distinguish and evaluate the importance of different words in the sentence based on their contextual relationships. Due to these advanced features and capabilities, *BERT* has a more precise and accurate capability of understanding sentences without being biased to surrounding words or influenced by word orders [35]. The *sentence transformer* model [36] is a specialized model that is pre-trained on a large and diverse dataset containing millions of text samples. It uses *BERT* as an encoder to transform text into meaningful numerical representations. The dataset used during training is collected from a large corpus, including *Wikipedia* [37], and by systematically crawling data on the Internet from various sources. It calculates similarity scores according to a well-defined and scientifically validated procedure. The *sentence transformer* model encodes each input text into a vector representation, which is also known as embedding, using a pre-trained transformer model such as *BERT* or the *Robustly Optimized BERT Approach (RoBERTa)*. These embeddings capture semantic meaning and contextual relationships in a high-dimensional space that enables mathematical comparison between different texts. After generating embeddings for the two texts that need to be compared, the *sentence transformer* model computes the similarity score using a mathematical metric like *Cosine similarity* or *Euclidean distance*. These metrics provide quantitative measures of semantic similarity between texts based on their vector representations. In the similarity score, higher scores indicating greater similarity between texts. The scoring system is normalized to provide consistent and interpretable results across different types of academic texts and research domains. In this thesis, I applied and implemented the *sentence transformer* model to compute accurate similarity scores to extract the most relevant papers from the search results provided by *Google Scholar*. This approach ensures that users receive papers that are truly relevant to their research interests rather than just keyword matches.

### 3.3.3 Python Flask as Web Application Server

The web application server, built using the *Python Flask* framework[38], serves as the central coordination hub and the control center for all system operations and component interactions. *Flask* provides a lightweight yet powerful foundation for building *RESTful APIs* and managing *HTTP* requests from the frontend application with high performance and reliability. The server implements multiple *API* endpoints that handle different aspects of system functionality including comprehensive user authentication and session management, search request processing and validation with input sanitization, asynchronous task management for long-running scraping operations, result retrieval and formatting with proper data structuring, and system status monitoring and logging for performance analysis and debugging. Each endpoint is carefully designed by following *RESTful* principles and best practices, ensuring consistent and predictable *API* behavior across all system interactions. The *Flask* application incorporates sophisticated middleware components for request logging, comprehensive error handling, and response formatting. *Cross-Origin Resource Sharing (CORS)* support enables seamless communication between the *Angular* frontend and the *Flask* backend, allowing the system to function properly across different domains and deployment configurations. The server implements advanced asynchronous task processing using background job queues and worker processes, allowing *web scraping* and *NLP* processing operations to run independently of *HTTP* request-response cycles. This architecture prevents timeout issues and provides better user experience during long-running operations that may take several minutes to complete. The asynchronous design also enables the system to handle multiple concurrent users efficiently. The *Flask* server contains comprehensive error handling and logging mechanisms that track system performance, identify potential issues, and provide detailed information for troubleshooting and optimization. Security features include input validation, *SQL* injection prevention, and protection against common web vulnerabilities.

### 3.3.4 Angular as User Interface Presentation

The user interface layer provides researchers with an intuitive, responsive, and feature-rich web-based interface to interact with the paper collection system effectively. Built on the *Angular* framework [39], the interface offers modern web application capabilities including single-page application architecture, reactive user interface components, and real-time updates that enhance user experience and productivity. The *Angular* application consists of several key components and features including a comprehensive search interface for entering research topics and keywords with validation and suggestions, advanced results display with sorting and filtering capabilities, detailed paper detail views with full metadata and download links and system status indicators for ongoing operations with progress tracking. *Material design* components provide consistent visual styling and user experience patterns, while custom components address specific requirements of academic paper management and research workflows. The interface also provides visual indicators for paper relevance scores and accessibility status to help users quickly identify the most valuable results. The user interface is designed with accessibility in mind, following web accessibility guidelines to ensure that researchers with different abilities can use the system effectively. This includes proper keyboard navigation, screen reader compatibility, and high contrast options.

## 3.4 Data Flow Architecture

The data flow architecture shows in detail how information moves through the system from initial user input to final results presentation. Understanding this comprehensive flow is crucial for system optimization, performance tuning, and troubleshooting when issues arise.

### 3.4.1 Input Processing Flow

The data flow begins when users submit research topics and keywords through the *Angular* frontend interface using the search form and input controls. User input undergoes the comprehensive validation and preprocessing to ensure data quality, system security, and optimal search performance. The frontend application sends *HTTP POST* requests to the *Flask* backend *API* endpoints, carrying user queries and search parameters along with necessary metadata. The *Flask* server receives and validates incoming requests through multiple layers of validation, checking for required parameters, input sanitization to prevent security vulnerabilities, and user authentication status to ensure authorized access. Valid requests are queued for processing using task management systems, while invalid requests return appropriate error responses to the frontend with detailed information about the validation failures. The server generates unique job identifiers for tracking search operations and stores initial request metadata in temporary storage for monitoring and debugging purposes. Input preprocessing includes several important steps such as text normalization to standardize the input format, keyword extraction to identify key terms, and query optimization to improve search effectiveness and accuracy. The system applies advanced text processing techniques, including stemming and lemmatization to handle variations in word forms, and removes stop words that might interfere with relevance assessment. Additional preprocessing includes spell checking, synonym expansion, and query reformulation to maximize the effectiveness of the search process.

### 3.4.2 Web Scraping Data Flow

Once input processing is complete and validated, the system initiates the comprehensive *web scraping* workflow through the *Selenium*-based scraping engine. The engine launches browser instances with appropriate configurations, navigates to *Google Scholar* using optimized paths, and executes search queries using the processed user input with proper formatting and timing. The scraping process follows a systematic and methodical approach including browser initialization with appropriate configurations and user agent settings, navigation to *Google Scholar* search interface using reliable paths, execution of search queries with user-specified keywords and parameters, systematic extraction of search results including paper titles, authors, abstracts, publication details, and download links, and comprehensive pagination handling to collect complete result sets without missing relevant papers. Extracted data undergoes the initial validation and formatting before being passed to the *NLP* processing module for further analysis. The system implements checkpoints throughout the scraping process to handle errors gracefully and ensure data integrity at each step of the process. Raw scraped data is temporarily stored in optimized memory structures designed for fast access during *NLP* processing operations. The system maintains detailed logs of scraping operations for debugging purposes and performance analysis to identify bottlenecks and optimization opportunities. In this section, the data flow while scraping from *Google Scholar* is processed according to the workflow shown in Figure 3.2. It illustrates the step-by-step process from initial search to final data extraction.

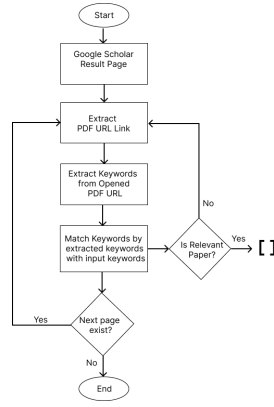


Figure 3.2: Web scraping data flow.

### 3.4.3 NLP Processing Flow

The *NLP* processing flow begins when the scraped paper data is passed to the *BERT*-based relevance assessment module from the *Hugging Face* library. This sophisticated component processes both user queries and paper abstracts to generate semantic embeddings that capture contextual meaning and complex relationships between concepts and terms. The *BERT* model from the *Hugging Face* library processes text through several sophisticated stages including tokenization to break text into manageable units and subwords, encoding tokens into numerical representations, embedding generation to create high-dimensional vector representations that capture semantic meaning, and similarity computation to compare query and paper embeddings using *cosine similarity* measures and other mathematical techniques. The relevance score is computed for each paper based on semantic similarity between user queries and paper abstracts using advanced mathematical algorithms. The system ranks and sorts papers with relevance scores in descending order, ensuring that the most relevant results are presented first to users for efficient review and selection. The *NLP* processing module also includes additional features such as the abstract summarization, keyword extraction from paper content, and topic modeling to provide users with additional insights about the collected papers. These features enhance the value of the search results and help users make better decisions about paper relevance.

### 3.4.4 Output Generation Flow

The final stage of data flow involves formatting and presenting processed results to users through the web interface in an organized and user-friendly manner. The *Flask* backend prepares response data including sorted paper lists, relevance scores, and comprehensive metadata information formatted as *JSON* objects[30] for frontend consumption and display. The *Angular* frontend receives processed results and renders them through web user interface. Results presentation includes tabular views with sortable columns, detailed paper information with expandable sections, direct download links for accessible papers, and relevance indicators to guide user attention to the most suitable results for their research needs. The system implements pagination and lazy loading mechanisms to handle large result sets efficiently without overwhelming the user interface or consuming excessive system resources. Users can navigate through results using intuitive controls, apply additional filters to narrow down results, and save interesting papers for future reference and follow-up research.

## 3.5 Technology Stack Selection

The selection of technologies for the reference paper collection system is based on specific requirements and careful evaluations including *web scraping* capabilities, *natural language processing* support, web application development efficiency, deployment flexibility, and long-term maintainability considerations.

### 3.5.1 Backend Technologies

*Python* serves as the primary programming language for backend developments, chosen specifically for its extensive ecosystem of libraries supporting *web scraping*, *natural language processing*, and web application development. *Python*'s exceptional readability and rapid development capabilities make it ideal for research-oriented applications that require frequent modifications and experimentation. *Selenium Web Driver* [31] provides robust and reliable web browser automation capabilities essential for scraping dynamic content from *Google Scholar*. *Selenium*'s comprehensive cross-browser support and mature *API* enable reliable data extraction from *JavaScript*-heavy academic websites that use complex dynamic content loading. The framework's ability to handle complex user interactions and dynamic page loading makes it superior to simpler scraping approaches that cannot handle modern web applications. *Python Flask* framework [40] offers lightweight yet powerful web application development capabilities with minimal overhead and maximum flexibility. *Flask*'s minimalist design philosophy allows developers to include only necessary components, resulting in efficient and maintainable applications that can be easily customized for specific requirements. The framework's extensive documentation and active community support facilitate rapid development and troubleshooting when issues arise. *BERT Model* from *Hugging Face* [41] provides state-of-the-art *natural language processing* capabilities for semantic similarity assessment and text understanding. The pre-trained models eliminate the need for extensive training data and computational resources while delivering high-quality results for academic text analysis. *Hugging Face*'s *Transformers* library offers easy integration and optimization features that simplify deployment and usage.

### 3.5.2 Frontend Technologies

*Angular* framework [42] provides comprehensive frontend development capabilities including component-based architecture, reactive programming support, and extensive tooling ecosystem for modern web development. *Angular*'s *TypeScript* [43] foundation ensures type safety and improved code maintainability, while its built-in features such as dependency injection and routing reduce development time and complexity significantly. *Angular Material* [44] components deliver consistent user interface design following *Material Design* principles and best practices. The pre-built components accelerate development while ensuring professional appearance and user experience standards that meet modern web application expectations. *HTML5* and *CSS3* technologies provide modern web standards support for semantic markup and advanced styling capabilities that enhance user experience. These technologies enable accessibility features and cross-browser compatibility essential for academic applications used across diverse computing environments and by users with different technical backgrounds.

### 3.5.3 Development and Deployment Tools

*Docker* containerization [14] enables consistent deployment environments across different platforms and simplified dependency management that reduces installation complexity. Containerization addresses common deployment challenges in academic settings where computing environments vary significantly between institutions and individual researcher setups. *Docker* containers ensure reproducible deployments and reduce installation complexity for end users who may not have extensive technical expertise. *Git* version control provides comprehensive source code management capabilities including branching strategies for feature development, collaborative development supports for team environments, and version history tracking for maintaining code quality and enabling rollback when necessary. *Git* repositories enable distributed development and facilitate code sharing among research collaborators and open-source contributors. *JSON* data formats enable efficient data exchange between system components and easy integration with web technologies and third-party tools. The standardized format ensures compatibility and simplifies data processing throughout the system.

## 3.6 Summary

The system design and architecture of the reference paper collection system successfully integrates modern web technologies, advanced *natural language processing* capabilities, and robust *web scraping* techniques to provide a comprehensive automated solution for academic paper collections. The three-tier architecture ensures clear separation of concerns while enabling scalable and maintainable system development that can adapt to changing requirements and growing user bases. The careful selections of *Python*, *Selenium*, *Flask*, *Angular*, and *BERT* technologies provide a comprehensive and well-integrated technology stack that addresses all system requirements including *web scraping* automation, intelligent paper filtering, user-friendly interfaces, and efficient backend processing. Each technology choice is carefully justified by specific capabilities and compatibility with overall system objectives and long-term sustainability goals. The data flow architecture ensures efficient information processing from user input through *web scraping*, *natural language processing*, and results presentation, creating a seamless user experience that maximizes productivity and research effectiveness. Design considerations for performance, scalability, reliability, and feasibility ensure that the system can operate effectively in diverse academic environments and handle varying usage patterns from individual researchers to large research institutions. The modular architecture facilitates future enhancements and adaptations to changing requirements in academic research workflows while maintaining system stability and performance.



# Chapter 4

## System Implementation

This chapter presents the implementation of *web scraping* functionality using *Selenium* automation [2]. It covers website navigation strategies, *PDF* download mechanisms, and content extraction algorithms. The *web scraping* implementation forms the core data collection component of the reference paper collection system, enabling automated extraction of academic paper information from *Google Scholar*.

### 4.1 Software Architecture

The system is implemented as a web application for helping facilitate academic paper searches and collect reference papers by researchers. It comprises three main components: the *user interface* of the *client-side*, the *model part* of the *server-side* that processes together the *Web Scraping*[26], and *data processing*. For the client-side, *Angular* is utilized to implement the *user interface* for filling out the title and keywords for a new search on a browser. For the server-side, *Python* is used in conjunction with *Flask*, which handles *web scraping* and *data processing* using *Selenium* and the *BERT* model, respectively. Figure 4.1 shows the overview of the system design.

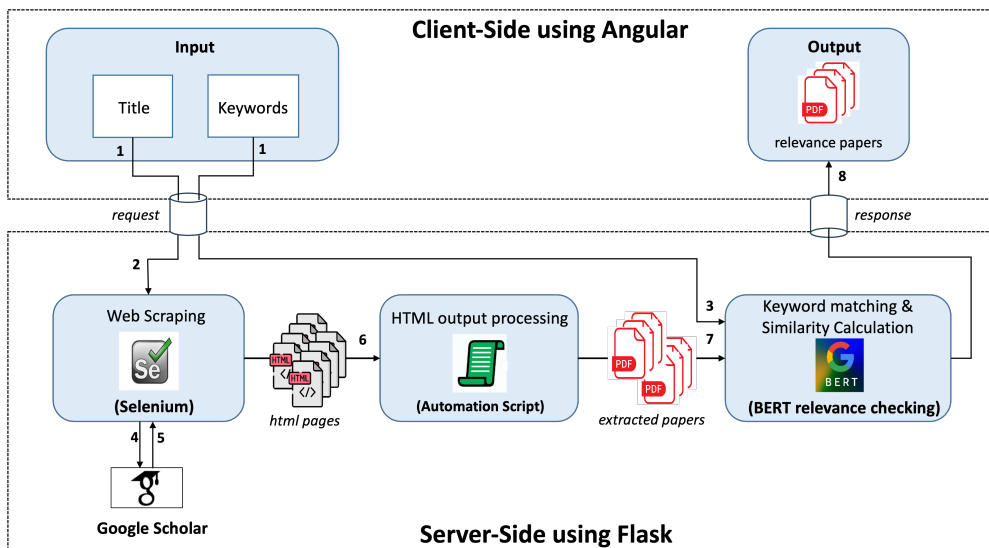


Figure 4.1: Overview of system design.

### 4.1.1 Client-Side Implementation

For the *client-side* implementation, *Angular* is adopted [39]. *Node.js* [45] version 12 is installed alongside *npm* or *Node Package Manager*, which is crucial for managing dependencies [46]. For *Angular* developments, *TypeScript* [43] is used instead of *JavaScript* for creating scalable and easy-to-maintain applications. Additionally, *Material UI* [44], which is an open-source *UI* component library optimized for *Angular* applications, is integrated. It ensures consistent user experiences across various platforms. The implemented function is designed to simplify the process of searching for titles and keywords related to the user's input. The title may vary based on the user's choices. After the application is launched via the port *localhost:4200*, the user interface implementation logic works as shown in Algorithm 1. After the user clicks the submit button, the client side sends *Hypertext Transfer Protocol (HTTP)* requests along with the user-input data, meanwhile showing the loading indicator. It then fetches the data from the server side. If the *server-side* response is successful and the *web scraping* procedure using *Selenium* and *BERT* is completed, the user can view the pages of the result papers on the browser.

---

**Algorithm 1** UserInterface Implementation Logic

---

```
1: function PAPERSEARCHFUNCTION(titleInput, keywordsInput)
2:   if titleInput is empty or keywordsInput is empty then
3:     display "Both fields are required."
4:     return
5:   end if
6:   display loadingIndicator
7:   requestData ← { "title": title, "keywords": keywords }
8:   responseData ← FETCHPAPERS(requestData)
9:   hide loadingIndicator
10:  if responseData is empty then
11:    display "No relevant papers found."
12:  else
13:    for all paper in responseData do
14:      showData(paper)
15:    end for
16:  end if
17: end function
```

---

## 4.2 Connection between Server-Side and Client-Side

In this application architecture, a user interaction with the client-side made on *Angular* and accessible via *localhost:4200*, initiates a sequence of events that involve communications with the *server-side* implemented with *Flask* and hosted at *localhost:5001*. When a user inputs data, such as *title* and *keywords*, and submit them, *Angular* generates the *HTTP* request containing this data. This request is directed towards the server's address of *localhost:5001*, where the *Flask* application resides.

## 4.3 Server-Side Implementation

For the *server-side*, *Flask* handles the *API* requests from the *client-side*. *Flask* provides *REST API* services using *Python* [40]. Upon receiving the request, typically via the *POST* method with additional parameters to the designated port of *localhost:5001*, *Flask* initiates a series of procedures. Firstly, it uses *web scraping* techniques to add more information to the input data by getting extra details from different websites. With the help of *automation scripts*, *Flask* handles the *HTML* documents to find important information from them [28]. *Flask* performs *keyword matching* and *similarity score calculation* to enhance the relevance and accuracy of search results or recommendations. Then, *Flask* displays the found papers on the *client-side* interface. We will explore each of these steps more comprehensively.

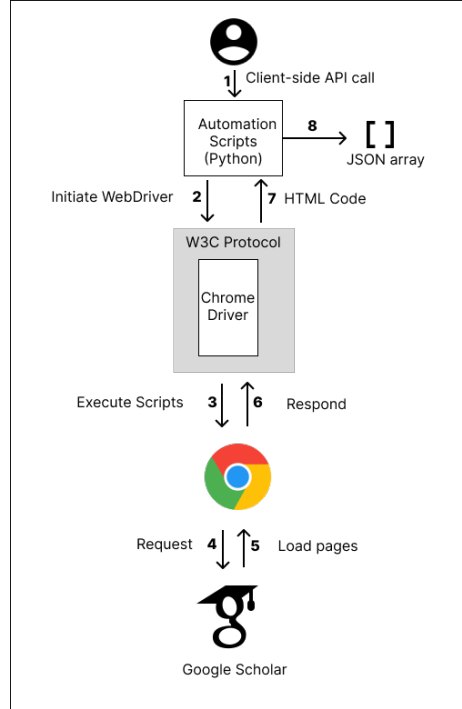


Figure 4.2: Web scraping with Selenium version-4.

### 4.3.1 HTML Output Processing with Automation Scripts

After getting the *HTML* output through *W3C* protocol [47], the data needs to be scraped, focusing on open-source reference papers. Typically, these papers are accessible in the *PDF* format via online *Uniform Resource Locator (URL)*. To collect these *URLs*, we create the automation script. Figure 4.3 shows the flow chart of the *HTML* output processing.

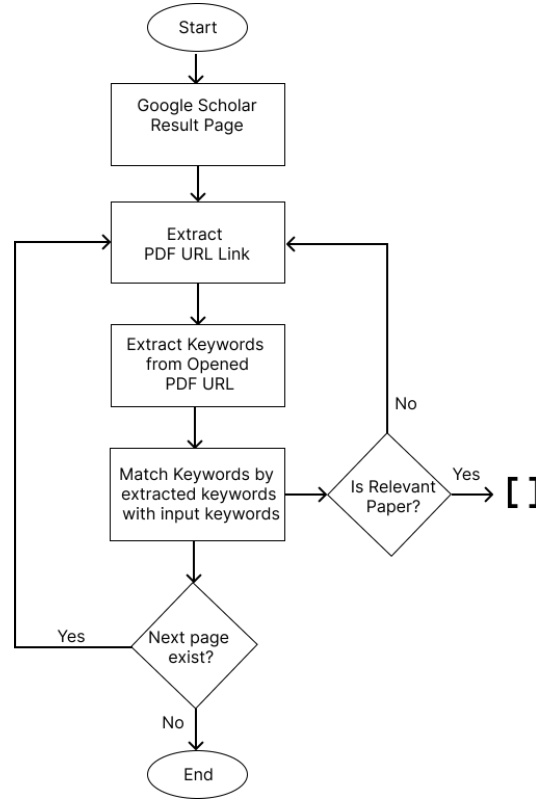


Figure 4.3: Flow chart of *HTML* output process.

1. **Process of Automation Script:** After the *Google Scholar* result list is loaded, the *automation script* locates the *URL* link of the *PDF* using *XPath* [48]. The dynamic nature of locating *DOM* (*Document Object Model*) elements on the website arises due to some tags lacking unique IDs or names. *XPath* is employed to pinpoint the *PDF* keywords among the *DOM* elements. Subsequently, the *regular expression* and the *Python PDF* reader library are utilized to extract keywords from the *PDF* online *URL*. Once the keywords are obtained, the *sentence transformer*-based *BERT* model[36] is used to calculate the similarity score. Further details on the data processing using the *sentence transformer* model will be provided in the next section. If the found paper is regarded as relevant, various information is collected from *Google Scholar*, including the paper title, the citation count, and the link to the *PDF*'s online *URL* according to Algorithm 2. Then, this data is transformed into a *JSON* array for further visualizations of the results.
2. **Multi-Threading:** Sequential retrieval of each page can be time-consuming, since it requires to wait for all pages to be searched and processed. As in Algorithm 5 and Figure 4, the *multi-threading* mechanism [49] is employed to enhance the system response time. First, the total number of pages is calculated. Then, the *web scraping* process across these pages is distributed to different threads, which significantly reduces the processing time [50]. However, increasing the number of threads also increases complexity. To ensure the efficient system resource utilization, available system resources are monitored.
3. **Error Handling Process:** Encountering dead links or invalid *PDF URL* links poses challenges. They can lead to potential system crashes when *Python* libraries like *PyPDF2* [51] and *PyMuPDF* [52] fail to read *PDF* texts. To address this challenge, we have implemented

---

**Algorithm 2** ScrapingData

```

1: function SCRAPINGDATA(url, keywords)
2:   Navigate to url using ChromeDriver
3:   Initialize empty list paper_data
4:   for i  $\leftarrow$  1 to 10 do
5:     while PDF links exist AND are valid do
6:       if CALCULATESIMILARITY(paper.pdf_link, keywords) then
7:         paper.pdf_link  $\leftarrow$  driver.By.XPATH, './span[contains(text(), "[PDF]")]')
8:         paper.similarity  $\leftarrow$  CALCULATESIMILARITY(paper.pdf_link, keywords)
9:         paper.title  $\leftarrow$  driver.get('title')
10:        paper.citation  $\leftarrow$  driver.get('citation')
11:        Add paper to paper_data
12:      end if
13:    end while
14:  end for
15:   $\hookleftarrow$  paper_data
16:   $\hookleftarrow$  FALSE
17:   $\hookleftarrow$  FALSE
18:   $\hookleftarrow$  FALSE
19:  DRIVER.CLOSE
20: end function

```


*error handling mechanisms*, including the validation of the *PDF* link, pages not found errors, and *SSL (Secure Socket Layer)* errors. Moreover, repeatedly accessing a large number of *PDF* links faces challenging cases. Extracting the *URL* and opening the link several times can trigger the browser to identify the human activity. To overcome this problem, headless *Chrome* is utilized, which lacks a *graphical user interface (GUI)*, consumes fewer system resources, and enables faster interactions. Additionally, the strategy to mimic human behaviors is implemented when interacting with the headless browser by introducing randomized time frames using waits and slower scrolling times, which can help avoid bot detection.

---

**Algorithm 3** MultiThreading

---

```

1: function MULTITHREADING(title, keywords)
2:   Initialize empty list urllist
3:   Construct base URL for Google Scholar search with the given title
4:   base_url  $\leftarrow$  'https://scholar.google.com/scholar?q=' + REPLACESPACES(title)
5:   Initialize ChromeDriver with base_url
6:   Count total result pages
7:   for page  $\leftarrow$  1 to totalPages do
8:     url  $\leftarrow$  base_url + '&?page=' + page
9:     Append url to urllist
10:  end for
11:  Initialize empty list results
12:  Initialize threads or async tasks to scrape data from url
13:  for each taskUrl in urllist do
14:    data  $\leftarrow$  SCRAPINGDATA(taskUrl, keywords)
15:    result  $\leftarrow$  taskUrl.execute(data)
16:    Append result to results
17:  end for
18:   results
19: end function

```

---

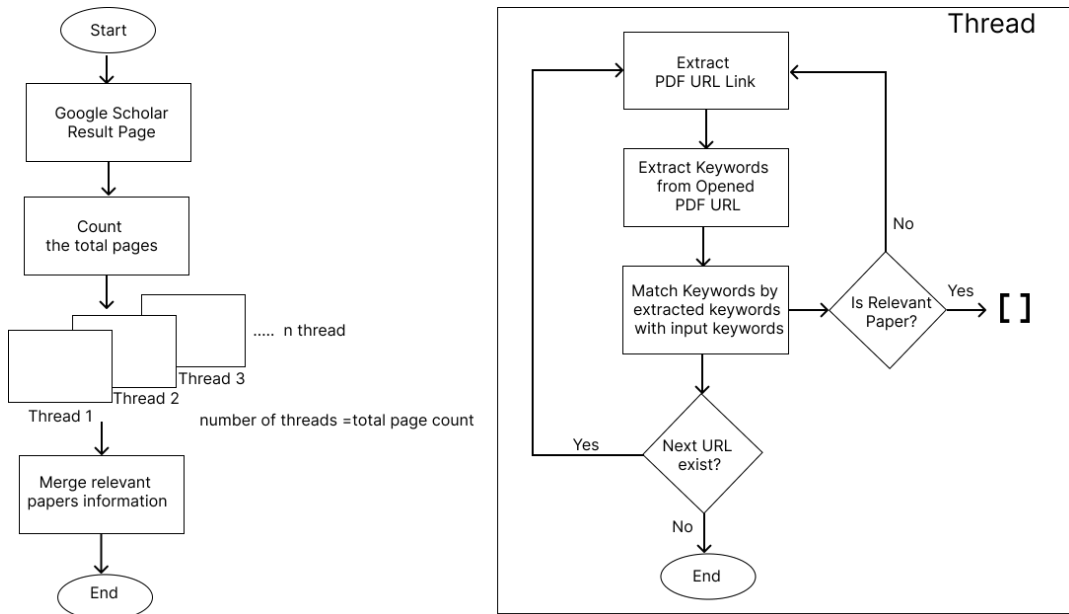


Figure 4.4: Multi-threading for enhanced response time in *web scraping*.

### 4.3.2 Pagination Handling

*Google Scholar* presents search results across multiple pages, requiring systematic pagination handling to collect comprehensive result sets. The implementation provides automatic pagination navigation with configurable limits to balance comprehensiveness with performance requirements. The pagination algorithm identifies next page navigation elements and clicks through result pages systematically. The system handles various pagination formats including numbered page links, next/previous buttons, and infinite scroll implementations. Robust element detection ensures reliable pagination navigation across different interface configurations. Page boundary detection prevents infinite loops when reaching the end of search results or encountering pagination errors. The system monitors the duplicate content, missing navigation elements, and error messages that indicate pagination completion or failure. Result aggregation combines data from multiple pages into coherent datasets while maintaining the sort order and removing duplicates. The system implements efficient data structures to handle large result sets without excessive memory consumption during pagination processing.

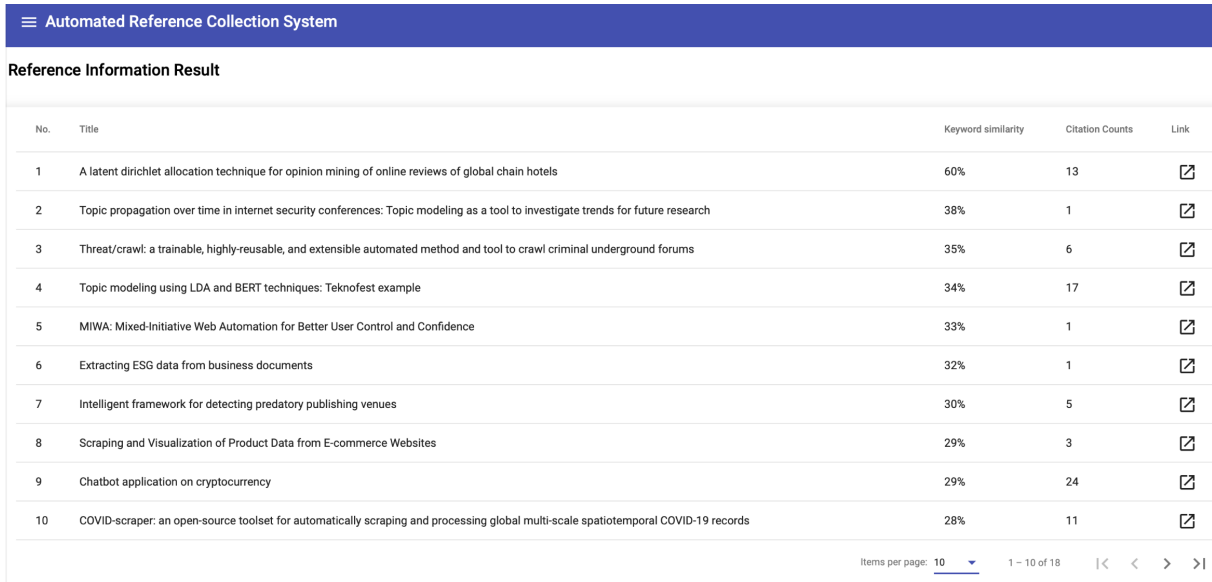
### 4.3.3 Concurrent Processing

Concurrent processing capabilities enable parallel execution of independent scraping operations to improve overall system throughput. The implementation provides thread-safe operation scheduling, resource coordination, and result aggregation mechanisms. Multi-threading support allows simultaneous execution of multiple scraping tasks including parallel browser instances for different search queries, concurrent *PDF* downloads for multiple papers, and simultaneous processing of extracted data. The system implements thread pools with configurable size limits to balance performance with resource consumption. Process coordination ensures that concurrent operations do not interfere with each other or overwhelm system resources. The system implements semaphores for resource access control, message queues for inter-process communication, and shared data structures for result aggregation. Load balancing distributes scraping tasks across available system

resources to optimize utilization and prevent bottlenecks. The system monitors resource availability and adjusts task distribution based on current system load and capacity. Synchronization mechanisms ensure data consistency when multiple processes access shared resources. The system implements appropriate locking strategies, atomic operations for critical sections, and conflict resolution mechanisms for concurrent data modifications.

#### 4.3.4 User Interface for Displaying Result Papers List

After completing the data processing stage, the backend server provides the reference papers in the *JSON* format [30]. They are sorted in descending order of the similarity scores. Then, the *client-side* receives the response and presents users with a visual representation of the sorted list of data. Figure 4.5 shows the user interface for displaying the result paper list. It includes the following output.



The screenshot shows a web application titled "Automated Reference Collection System". Below the title is a section labeled "Reference Information Result". It contains a table with 5 columns: "No.", "Title", "Keyword similarity", "Citation Counts", and "Link". The table lists 10 papers, sorted by keyword similarity in descending order. At the bottom right of the table, there is a pagination control showing "Items per page: 10" and "1 - 10 of 18", along with navigation arrows.

No.	Title	Keyword similarity	Citation Counts	Link
1	A latent dirichlet allocation technique for opinion mining of online reviews of global chain hotels	60%	13	<a href="#">↗</a>
2	Topic propagation over time in internet security conferences: Topic modeling as a tool to investigate trends for future research	38%	1	<a href="#">↗</a>
3	Threat/crawl: a trainable, highly-reusable, and extensible automated method and tool to crawl criminal underground forums	35%	6	<a href="#">↗</a>
4	Topic modeling using LDA and BERT techniques: Teknofest example	34%	17	<a href="#">↗</a>
5	MIWA: Mixed-Initiative Web Automation for Better User Control and Confidence	33%	1	<a href="#">↗</a>
6	Extracting ESG data from business documents	32%	1	<a href="#">↗</a>
7	Intelligent framework for detecting predatory publishing venues	30%	5	<a href="#">↗</a>
8	Scraping and Visualization of Product Data from E-commerce Websites	29%	3	<a href="#">↗</a>
9	Chatbot application on cryptocurrency	29%	24	<a href="#">↗</a>
10	COVID-scraper: an open-source toolset for automatically scraping and processing global multi-scale spatiotemporal COVID-19 records	28%	11	<a href="#">↗</a>

Items per page: 10 1 - 10 of 18 |< < > >|

Figure 4.5: Interface for displaying result papers list.

#### 4.3.5 Handling Web Scraping Module

The *web scraping* implementation provides robust and efficient mechanisms for automated data collection from *Google Scholar* using *Selenium WebDriver* technology[31]. The implementation addresses the complex challenges of modern *web scraping* including dynamic content handling and reliable error recovery. The *Selenium WebDriver* configuration establishes a solid foundation for browser automation with comprehensive browser support, performance optimization. Website navigation strategies handle the complexities of *Google Scholar*'s interface including search execution, result processing, and pagination management. The implementation provides flexible and robust navigation capabilities that adapt to interface changes and handle various edge cases. The implementation handles diverse content formats and provides comprehensive validation mechanisms. *PDF* download mechanisms enable automatic retrieval of freely available papers with robust file management, content validation, and quality assurance features. The system provides efficient download capabilities while through intelligent resource management and concurrent processing. The implementation balances performance requirements with resource constraints and

system stability considerations. The *web scraping* implementation forms a critical foundation for the reference paper collection system, enabling automated and reliable data collection that supports the system’s overall objectives of improving research efficiency and paper discovery capabilities.

## 4.4 Paper Filtering Relevance

The effectiveness of the automated reference paper collection system critically depends on its ability to accurately assess the relevance and quality of collected papers while efficiently filtering out irrelevant or duplicate content. This section presents comprehensive algorithms and methodologies for paper filtering and relevance assessment, including advanced metadata extraction techniques, sophisticated duplicate detection methods and quality scoring mechanisms. Modern academic literature databases contain millions of papers across diverse disciplines, making automated relevance assessment a challenging task that requires sophisticated understanding of semantic relationships, contextual meaning, and domain-specific terminology [20]. Traditional keyword-based filtering approaches often produce high false positive rates and miss relevant papers that use different terminology or present related concepts through varied linguistic expressions. This section addresses these challenges through the development of advanced filtering algorithms that combine *natural language processing* techniques, machine learning approaches, and domain-specific heuristics to achieve accurate relevance assessment [11]. The proposed method leverages state-of-the-art transformer-based language models, particularly, *BERT (Bidirectional Encoder Representations from Transformers)*, to capture semantic relationships and contextual understanding that traditional approaches cannot achieve. The filtering and assessment framework consists of multiple interconnected components including metadata extraction engines, duplicate detection algorithms, relevance scoring mechanisms, and quality assessment modules. Each component contributes to the overall effectiveness of the system while maintaining computational efficiency suitable for real-time operation on large document collections.

## 4.5 Relevance Implementation

### 4.5.1 Keyword Matching and Similarity Score Calculation

To proceed the *data processing* stage after the data extraction stage, one of the *NLP* techniques is applied to recommend the user with the most relevant papers. *BERT* with the *encoder-based sentence transformer* model is employed to find relevant papers for the user [24]. *BERT* encodes the user’s input keywords and the extracted keywords by *web scraping* into vector representations of the text. The *sentence transformer* model computes the similarity score using the *cosine similarity*. To perform the process, the *Python sentence transformer* library is imported from *Hugging Face* [41], which is a community providing open-source *Large Language Models (LLM)*. According to Algorithm 4, the *Python* program passes two text arguments into the model: the keywords extracted from the online *PDF* using the *Python PDF* reader and the *regular expressions*, and the user input text, separated by commas. The *all-MiniLM-L12-v2*, which is one of the *sentence transformer* models [53], is known for its faster response time and smaller size, and is used to compute the similarity score between the user input keywords and the extracted papers’ keywords. After the data processing stage, the server program returns the sorted list of the similarity scores to the *client-side* program. Further details on visualization result outputs will be provided in the next section.

---

**Algorithm 4** CalculateSimilarity

---

```
1: function CALCULATESIMILARITY(pdf_url, keywords)
2:   if PDF is readable then
3:     content ← READ(pdf_url)
4:     terms ← ['Index Terms', 'keywords', 'Key Words']
5:     matched_keywords ← REGEX_SEARCH(content, terms)
6:     model ← SENTENCETRANSFORMER('sentence-transformers/all-MiniLM-L6-v2')
7:     similarity ← MODEL.CALCULATE_SIMILARITY(keywords, matched_keywords)
8:     ↩ similarity
9:   else
10:    ↩ False
11:   end if
12: end function
```

---

### 4.5.2 Source Code Implementation for Backend Automation

As shown in Figure 4.6, the *Python* automation code is designed to search for academic papers on *Google Scholar* using the specific keywords. It finds the papers that have *PDF* versions available by handling multiple exceptions and collects information about these papers, such as their titles, links to the *PDFs*, and the times when they have been cited by other papers. Afterward, the similarity is calculated using a *sentence-similarity transformer* model, which compares the provided keywords with those in the *PDFs* and returns a sorted list of academic papers.

```
def scrape_data(title, keywords):
    try:
        driver.get('https://scholar.google.com/')
        search_box = driver.find_element(By.NAME, "q").send_keys(title)
        search_box.submit()
        # Navigate to the URL
        div_elements = driver.find_elements(By.CLASS_NAME, "gs_r.gs_or.gs_scl")
        for div_element in div_elements:
            # Check if there are any [PDF] spans within gs_r
            gs_r = div_element.find_element(By.CLASS_NAME, "gs_r")
            pdf_spans_gs_r = gs_r.find_elements(By.XPATH, '//*[@contains(text(), "[PDF]")]')
            if pdf_spans_gs_r:
                try:
                    pdf_link = div_element.find_element(By.PARTIAL_LINK_TEXT, "PDF").get_attribute("href")
                    title = gs_r.find_element(By.TAG_NAME, "a").text
                    citation_a = gs_r.find_element(By.XPATH, "//*[@contains(@href, 'cites=')]")
                    citations = int(citation_a.text.split()[-1])
                    pdf_links.append({'title': title, 'pdf_link': pdf_link, 'citations': citations})
                except Exception as e:
                    else:
            # If gs_r or gs_ggsd has PDF, print the titles and PDF links
            if pdf_links:
                for item in pdf_links:
                    similarity = Model_Transformer.get_caluclated_list(item["pdf_link"], keywords)
                    if similarity != '':
                        lst.append({'title': item["title"],
                                    'url': item["pdf_link"], 'similarity': float(similarity)*100, 'citations': item["citations"]})
            else:
                driver.close()
                return lst
    finally:
        driver.quit()
```

Figure 4.6: Source code implementation of backend automation.

## 4.6 Similarity Computation Methodology

### 4.6.1 Document Processing and Metadata Extraction

The system employs a multi-stage document processing pipeline that begins with *PDF* content extraction and metadata identification. The implementation utilizes the *PyPDF2* library to extract textual content from academic documents, followed by targeted metadata extraction using *regular expression* patterns. The metadata extraction process specifically targets standardized academic keyword indicators including “Index Terms,” “keywords,” and “Key Words.” This approach leverages the conventional structure of academic publications where author-specified keywords are typically presented using these standardized terminologies. The *regular expression* pattern `r'\b\b\s+(. *?[.?!])'` captures the content following these indicators until a sentence-ending punctuation mark, ensuring extraction of complete keyword phrases rather than fragmented terms.

### 4.6.2 Semantic Similarity Computation

The similarity assessment process transforms both user-specified keywords and extracted document metadata into high-dimensional vector representations using the *sentence transformer* model. The implementation processes user keywords as individual tokens through the `split()` operation, creating a list of discrete terms for encoding. The encoding process utilizes the `model.encode()` function with tensor conversion enabled, generating *PyTorch* tensors that facilitate efficient similarity computation. This approach leverages *GPU* acceleration capabilities when available, significantly improving processing performance for large document collections. *Cosine similarity* serves as the primary similarity metric, computed using the `util.cos_sim()` function from the *SentenceTransformer* utilities. *Cosine similarity* provides a normalized measure of vector alignment that remains invariant to vector magnitude, ensuring consistent similarity scores regardless of text length variations. The mathematical formulation follows:

$$\text{cosine\_similarity} = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \times \|\mathbf{B}\|} \quad (4.1)$$

Where **A** and **B** represent the sentence embeddings for user keywords and document metadata, respectively.

### 4.6.3 Error Handling and Robustness

The implementation incorporates comprehensive error handling mechanisms to address common challenges in automated document processing. *SSL* certificate verification is disabled to accommodate documents hosted on servers with certificate issues, while maintaining functionality through the `verify=False` parameter in the requests call. The system implements fallback mechanisms for various failure scenarios, including *PDF* parsing errors, network connectivity issues, and malformed document content. When primary similarity computation fails, the system returns randomized low similarity scores (0.05-0.08 range) to maintain system continuity while indicating low confidence in the assessment. Document length filtering prevents processing of excessively long documents (>8 pages) that may indicate review papers, books, or other non-standard academic publications that could skew relevance assessment results.

#### 4.6.4 Similarity Score Interpretation

The similarity scores generated by the *cosine similarity* computation provide normalized values between -1 and 1, with higher values indicating greater semantic alignment. The implementation converts raw similarity scores to percentage format with two decimal precisions, facilitating intuitive interpretation by end users.

### 4.7 Chapter Summary

The implementation demonstrates effective utilization of modern transformer-based *natural language processing* techniques for academic document relevance assessment [54]. The *sentence transformer* approach provides significant improvements over traditional keyword matching while maintaining computational efficiency suitable for practical deployment. The system's integration of document processing, metadata extraction, and semantic similarity computation creates a robust framework for automated literature review support. While current limitations exist regarding keyword extraction scope and document type handling, the foundational architecture provides a solid basis for enhanced academic document filtering systems. The successful application of the *all-MiniLM-L6-v2* model illustrates the practical value of optimized transformer architectures in real-world *natural language processing* applications, particularly in domains requiring semantic understanding beyond simple lexical matching. This implementation contributes to the growing body of work applying advanced *NLP* techniques to academic research support tools, demonstrating both the potential and practical considerations involved in deploying transformer-based similarity assessment systems.

# Chapter 5

## System Enhancements

### 5.1 Introduction

The reference paper collection system provides an automated and efficient approach to collecting academic papers using advanced *web scraping* technology [1] and sophisticated *natural language processing (NLP)* models. While the base system demonstrates significant effectiveness in automating paper collection tasks for researchers, several performance and deployment challenges necessitate the implementation of advanced enhancement features to improve overall functionality and user experience. The system often encounters limitations where it finds only a limited number of relevant reference papers after taking a considerably long time to complete the search process. This happens because the system relies on only one paper search website and runs on a single thread at a multi-core *CPU*, which does not utilize the full potential of modern computing hardware. This chapter presents three critical and essential enhancements to address these significant limitations: *multi-threading* implementation for improved performance and faster processing[55], *Docker* containerization for deployment flexibility and system portability[56], and robust *PDF* file handling mechanisms for reliable content extraction from various document formats. These comprehensive enhancements work together to transform the reference paper collection system from a single-threaded, monolithic application into a scalable, containerized platform. The enhanced system is capable of handling diverse *PDF* formats and edge cases while maintaining high performance standards and reliability requirements that are essential for academic research environments. The implementation of these enhancements ensures that researchers can access more relevant papers in less time, deploy the system across different computing environments easily, and extract content from various types of *PDF* documents without encountering technical barriers.

### 5.2 Multithreading Enhancement Architecture

#### 5.2.1 Threading Model Design

The enhanced system implements a comprehensive and well-designed *multi-threading* architecture to improve processing efficiency and significantly reduce response times for users [57]. Sequential retrieval of each page can be extremely time-consuming and inefficient, since it requires waiting for all pages to be searched and processed one by one in a linear fashion. The *multi-threading* mechanism is specifically employed to enhance the system response time by allowing multiple operations to be executed simultaneously [58]. This approach takes advantage of modern multi-

core processors and enables the system to process multiple search results pages concurrently rather than sequentially.

The threading model should be designed with careful consideration of resource management, thread safety, and optimal performance. Each thread operates independently while sharing necessary data structures through thread-safe mechanisms that prevent data corruption and ensure consistent results. The design also includes intelligent thread pool management that automatically adjusts the number of active threads based on system resources and workload requirements [59]. This ensures optimal performance without overwhelming the system or causing resource exhaustion.

### 5.2.2 Parallel Processing Implementation

The *multithreading* implementation follows a distributed processing approach where the total number of pages is calculated first. Then, the *web scraping* process across these pages is distributed to different threads. This distribution significantly reduces the overall processing time and improves system throughput. The parallel processing workflow includes several sophisticated steps that ensure efficient and coordinated operation of multiple threads working together toward a common goal [60].

---

**Algorithm 5** Multithreading *web scraping* process.

---

```
1: Input: Search query  $q$ , total pages  $n$ 
2: Output: Aggregated search results  $R$ 
3: Calculate total number of pages  $n$  from initial search
4: Create thread pool with size  $t = \min(n, \text{max\_threads})$ 
5: Partition pages into  $t$  chunks:  $P_1, P_2, \dots, P_t$ 
6: for each thread  $i = 1$  to  $t$  do
7:   Launch thread to process page chunk  $P_i$ 
8:   Execute Web Scraping on assigned pages
9:   Extract paper information and PDF links
10:  Queue PDF files for parallel processing
11:  Store results in thread-safe data structure
12: end for
13: Wait for all threads to complete
14: Aggregate results from all threads into  $R$ 
15: return  $R$ 
```

---

The implementation also includes sophisticated error handling mechanisms that detect and recover from thread-specific failures without affecting the overall operations. This ensures that even if individual threads encounter problems, the system can continue processing and deliver results from successful threads.

## 5.3 Docker Containerization Strategy

### 5.3.1 Container Architecture Design

The system is implemented through the incorporation of *Docker* technology for deployment within the framework of interactive web services [61]. The containerization strategy employs a microservices-oriented architecture that separates different system components into isolated, manageable containers that can be deployed and scaled independently [62]. This approach provides numerous benefits including the improved system isolation, easier deployment across different environments, better resource management, and simplified scaling capabilities. Each container is designed to handle specific functionality while communicating with other containers through well-defined interfaces.

- **Backend Container:** houses the main reference collection application with all required dependencies, including the *web scraping* engine, *natural language processing* module, and interactive web services. This container is optimized for computational tasks and includes all necessary libraries and tools for *PDF* processing and text analysis.
- **Frontend Container:** is the isolated container for user-interface application using *Node.js* and running as server-side rendering using *Angular* Framework. This container focuses on delivering responsive user interfaces and handling user interactions while communicating with the backend container through *APIs*.

This container architecture is designed with security, performance, and maintainability in mind [63]. Each container runs with minimal privileges and includes only the necessary components for its specific functionality, reducing the attack surface and improving overall system security.

### 5.3.2 Docker Configuration

The system utilizes *Docker Compose* for orchestrating the environment settings and managing container interactions [64]. The configuration is designed to be flexible and easily adaptable to different deployment scenarios.

Listing 5.1: *Docker* Configuration

```
1 version: '3.8'
2 # syntax=docker/dockerfile:1
3 FROM python:3.9-slim-buster
4 RUN apt-get update && apt-get install -y \
5     curl
6 RUN apt-get update \
7     && apt-get install -y wget \
8     && apt-get install -y unzip \
9     && rm -rf /var/lib/apt/lists/* \
10    && apt-get update && apt-get install -y gnupg2\
11    && apt-get update && apt-get install -y curl unzip jq
12 ARG CHROMEDRIVER_VERSION='122'
13 # Install Chrome WebDriver
14 RUN CHROMEDRIVER_URL=$(curl -s https://googlechromelabs.github.io/chrome-for
    -testing/known-good-versions-with-downloads.json | \
15     jq -r --arg version "$CHROMEDRIVER_VERSION" '[.versions[] | select(.
    version | startswith($version + "."))] | last | .downloads.
    chromedriver[] | select(.platform == "linux64").url') && \
16     mkdir -p /opt/chromedriver-$CHROMEDRIVER_VERSION && \
17     curl -sS -o /tmp/chromedriver_linux64.zip "$CHROMEDRIVER_URL" && \
```

```

18 unzip -qq /tmp/chromedriver_linux64.zip -d /opt/chromedriver-
   $CHROMEDRIVER_VERSION && \
19 rm /tmp/chromedriver_linux64.zip && \
20 chmod +x /opt/chromedriver-$CHROMEDRIVER_VERSION/chromedriver-linux64/
   chromedriver && \
21 ln -fs /opt/chromedriver-$CHROMEDRIVER_VERSION/chromedriver-linux64/
   chromedriver /usr/local/bin/chromedriver
22 # Install Google Chrome
23 RUN curl -sS -o - https://dl-ssl.google.com/linux/linux_signing_key.pub | apt-key add -
   && \
24 echo "deb http://dl.google.com/linux/chrome/deb/ stable main" >> /etc/
   apt/sources.list.d/google-chrome.list && \
25 apt-get -yqq update && \
26 apt-get -yqq install google-chrome-stable && \
27 rm -rf /var/lib/apt/lists/*
28 WORKDIR /matching
29 COPY requirements.txt requirements.txt
30 RUN pip3 install -r requirements.txt
31 COPY ./matching/
32 CMD [ "python", "-m", "flask", "run", "--host=0.0.0.0", "--port=5001" ]

```

---

The *Docker* configuration includes several important features and optimizations:

- **Multi-Stage Build Process:** utilizes *Python 3.9 slim-buster* base image for optimal balance between functionality and container size. This base image provides essential *Python* runtime capabilities while minimizing security surface area and reducing container size for faster deployment and reduced storage requirements.
- **System Dependencies Management:** is realized by comprehensive package installations including *curl*, *wget*, *unzip*, and *gnupg2* for secure package management and *web scraping* capabilities required for academic paper collections. These dependencies are carefully selected to provide necessary functionality while maintaining minimal container size.
- **Chrome Web Driver Integration:** is associated with dynamic *ChromeDriver* installation using automated version detection and *JSON API* querying to ensure compatibility with *Google Chrome* for *Selenium*-based *Web Scraping* operations [31]. This automated approach ensures that the system always uses compatible versions of *Chrome* and *ChromeDriver*.
- **Browser Automation Support:** completes *Google Chrome* stable installation with official repository configuration, enabling headless browser operations for automated academic database navigation and *PDF* extraction. The installation includes all necessary components for running *Chrome* in headless mode within the container environment.
- **Python Environment Configuration:** is structured by *Python* dependency management through *requirements.txt* installation in dedicated working directory, ensuring reproducible builds and consistent library versions across different deployment environments. This approach enables precise control over software dependencies and versions.
- **Application Deployment Strategy:** is made by *Flask* web application deployment configuration with external host binding (*0.0.0.0*) on port *5001*, enabling container accessibility and seamless integration with orchestration platforms. This configuration allows the application to accept connections from outside the container.

- **Security Hardening:** realizes *APT* cache cleanup and temporary file removal to minimize container attack surface while maintaining essential functionality for academic paper processing workflows. These security measures reduce the potential for security vulnerabilities and minimize container size.
- **Development Workflow Support:** makes modular *COPY* operations separate dependency installations from application code deployment, optimizing *Docker* layer caching for faster development iterations and more efficient builds during development cycles.
- **Runtime Optimization:** allows working directory isolations (*/matching*) providing clean application namespace and simplified file path management for *PDF* processing and similarity assessment operations. This organization improves code maintainability and reduces path-related errors.
- **Container Orchestration Ready:** makes port exposure and host binding configuration designed for seamless integration with *Docker Compose*, *Kubernetes*[65], or other container orchestration platforms supporting academic research infrastructure and scalable deployment scenarios.

### 5.3.3 Frontend Container Configuration

In addition to the backend container configuration presented above, the system implements a separate *Docker* container for the *Angular*-based frontend application. This separation follows microservices architecture principles and enables independent scaling and deployment of the user interface components.

Listing 5.2: *Angular* Frontend *Docker* Configuration

---

```

1 FROM node:16-alpine AS build
2 WORKDIR /app
3 COPY package*.json ./
4 RUN npm install
5 COPY . .
6 RUN npm run build
7 EXPOSE 4200
8 CMD ["npm", "start"]

```

---

The frontend container configuration implements several key features:

- **Node.js Alpine Base Image:** utilizes *Node.js 16 Alpine Linux* distribution for minimal container footprint while providing full *Node.js* runtime capabilities required for *Angular* application execution.
- **Dependency Management:** uses package installation through *npm* ensures all required *Angular* framework dependencies and development tools are properly installed before the build process.
- **Build Process Integration:** executes the *Angular* build pipeline with the *RUN npm run build* command, compiling *TypeScript* code, bundling *JavaScript* modules, and optimizing assets for production deployment.
- **Port Configuration:** exposes port 4200, the standard *Angular* development server port, enabling external access to the web interface when the container is running.

- **Application Serving:** launches the *Node.js* server with the *npm start* command to serve the compiled *Angular* application, providing the user interface for the reference paper collection system.

This frontend container works in conjunction with the backend container to provide a complete, containerized solution for the reference paper collection system. The separation of frontend and backend into distinct containers enhances system modularity, simplifies maintenance, and enables flexible deployment strategies across different computing environments.

## 5.4 PDF File Handling and Edge Case Management

### 5.4.1 Robust PDF Processing Framework

The enhanced system implements comprehensive *PDF* file handling mechanisms to address various edge cases and ensure reliable content extractions from diverse document formats commonly encountered in academic research environments [66]. The *PDF* processing framework is designed to handle the wide variety of *PDF* formats, encodings, and structural complexities that exist in academic literature. This includes everything from simple text-based *PDFs* to complex documents with embedded images, tables, and non-standard layouts.

---

**Algorithm 6** Robust *PDF* processing with edge case handling.

---

```

1: Input: PDF file path  $p$ , processing timeout  $t$ 
2: Output: Extracted text content  $C$  or error status
3: Initialize PDF processing pipeline
4: Validate file existence and accessibility
5: if file type  $\neq pdf$  then
6:   return ERROR_FILE_FORMAT_INVALID
7: end if
8: try:
9:   Attempt primary extraction using PyPDF2
10: if extraction successful then
11:   Validate extracted content quality
12:   return  $C$ 
13: end if
14: except PDF corruption or password protection:
15:   Attempt fallback extraction using pdfplumber
16: if extraction successful then
17:   return  $C$ 
18: end if
19: return ERROR_EXTRACTION_FAILED

```

---

The processing framework includes multiple layers of validation and error handling to ensure that the system can extract meaningful content from *PDF* files even when they have structural problems or use non-standard formatting.

## 5.4.2 Edge Case Handling Mechanisms

The *PDF* processing system addresses various edge cases commonly encountered in academic paper processing [67]. These edge cases can significantly impact the system’s ability to extract useful contents if not properly handled.

Table 5.1: *PDF* edge cases and handling strategies.

Edge Case	Detection Method	Handling Strategy
Corrupted <i>PDF</i>	File header validation	Attempt repair or skip
Password Protected	Access attempt failure	Request credentials or skip
Large File Size	File size check	Stream processing
Non-standard Encoding	Character validation	Encoding detection
Embedded Objects	Content analysis	Selective extraction
Multi-column Layout	Text flow analysis	Layout-aware extraction

The system provides specific handling strategies for each type of edge case:

1. **Corrupted PDF Files:** implements file integrity validation and repair mechanisms using specialized *PDF* libraries to recover readable content from partially corrupted files[68]. The system attempts to repair minor corruption issues and extract whatever content is still accessible, ensuring that partially damaged files can still provide valuable information.
2. **Password-Protected Documents:** detects password-protected *PDFs* with fallback mechanisms to attempt common academic passwords or skip protected documents with appropriate logging. The system maintains a database of commonly used passwords in academic settings while respecting security requirements and user privacy.
3. **Large File Handling:** implements streaming *PDF* processing to handle large academic documents without excessive memory consumption[69]. This includes pagination and chunked processing strategies that allow the system to process very large documents without running out of memory or causing system instability.

The edge case handling mechanisms are designed to be extensible, allowing new types of edge cases to be added as they are discovered. The system maintains detailed logs of all edge cases encountered, enabling continuous improvement of the handling strategies.

## 5.4.3 Multi-Library PDF Processing Pipeline

The system employs a multi-library approach to ensure robust *PDF* content extraction across different document types and formats [70]. This approach recognizes that no single *PDF* processing library can handle all possible document types effectively.

- **Primary Extraction (PyPDF2):** offers fast and efficient text extraction for standard *PDF* documents with proper text encoding. *PyPDF2* is optimized for speed and works well with most standard academic papers that follow conventional *PDF* formatting standards.
- **Fallback Extraction (pdfplumber):** advances *PDF* processing capabilities for complex layouts and non-standard formatting, particularly effective for academic papers with tables, figures, and complex multi-column layouts. *pdfplumber* provides more sophisticated layout

analysis capabilities at the cost of increased processing time. The multi-library approach ensures that the system can handle the widest possible range of *PDF* document types while maintaining good performance for standard documents. The system automatically selects the most appropriate library based on document characteristics and processing requirements.

## 5.4.4 Content Quality Validation

The *PDF* processing pipeline includes comprehensive content quality validation mechanisms to ensure that extracted content meets minimum standards for usefulness in academic researches [71].

**Text Quality Assessment** evaluates extracted text quality based on character recognition confidence, language detection, and content coherence metrics. The system analyzes the extracted text to determine whether it contains meaningful content or just garbled characters resulting from processing errors.

**Content Completeness Validation** verifies that extracted content represents the complete document by comparing extracted text length against expected academic paper lengths and checking for proper document structure indicators such as abstracts, introductions, and conclusions.

**Error Recovery Mechanisms** automatically retries with different extraction libraries when initial processing fails or produces low-quality results. The system can automatically switch to alternative processing methods when the primary approach fails to produce acceptable results.

The quality validation system ensures that users receive high-quality extracted content that is suitable for academic research purposes. Low-quality extractions are flagged for manual review or automatic reprocessing using alternative methods.

## 5.5 Error Handling and Reliability

### 5.5.1 Comprehensive Error Management

The enhanced system implements robust error handling mechanisms across all components to ensure reliable operation even when individual components encounter problems or unexpected conditions [72]. The error handling system is designed to be comprehensive, covering all possible failure points in the system and providing appropriate recovery mechanisms for each type of error that might occur.

1. **Web Scraping Error Handling:** handles graceful handling of network timeouts, *HTTP* errors, and website structure changes with automatic retry mechanisms. The system can detect when websites change their structures and adapt accordingly, ensuring continued operation even when target websites are updated.
2. **PDF Processing Error Recovery:** comprehensively handles error handling for *PDF* processing failures with fallback extraction methods and detailed error logging. When one *PDF* processing method fails, the system automatically tries alternative approaches before giving up on a particular document.

3. **Thread Synchronization Error Handling:** detects and recovers from thread synchronization issues, deadlocks, and resource contention scenarios [73]. The multithreaded system includes sophisticated monitoring mechanisms that can detect when threads are not operating properly and take corrective action.
4. **Container Failure Recovery:** automatically restart containers and health monitoring with persistent data protection during container failures. The containerized system includes health monitoring and automatic restart capabilities that ensure continued operation even when individual containers fail.

The error handling system maintains detailed logs of all errors and recovery actions, enabling system administrators to identify patterns and potential improvements to system reliability.

## 5.6 Chapter Summary

This chapter presented three critical and essential enhancements for the reference paper collection system: *multi-threading* implementation, *Docker* containerization, and robust *PDF* file handling mechanisms. These enhancements work together to address the limitations of the original system and provide significant improvements in performance, reliability, and usability. The *multi-threading* enhancement significantly improves the system performance by enabling parallel processing of *web scraping* operations. This reduces response time and increases *PDF* processing rates, allowing users to obtain results much faster than with the original single-threaded approach. The parallel processing capabilities make full use of modern multi-core processors and can scale to handle larger workloads efficiently. The *Docker* containerization strategy provides deployment flexibility, system isolation, and scalability through microservices architecture. The containerized approach makes it much easier to deploy the system across different computing environments and ensures consistent operation regardless of the underlying hardware or operating system. The specialized *PDF* processing container ensures reliable content extraction across diverse document formats while maintaining system performance and resource efficiency. The comprehensive *PDF* file handling framework addresses critical edge cases including corrupted files, password-protected documents, scanned documents, and large file processing. The multi-library processing pipeline with fallback mechanisms ensures maximum content extraction success rates while maintaining system reliability and providing high-quality results for academic research purposes. The proposed enhancements transform the reference paper collection system from a single-threaded, monolithic application into a scalable, containerized platform that is capable of handling diverse academic document formats and processing challenges. These improvements enable efficient deployments across different environments while maintaining high performance, reliability, and content extraction accuracy that meets the demanding requirements of academic research. The implementation of these enhancements makes the reference paper collection system to be a robust, scalable solution for automated academic research support. The enhanced system is capable of handling increased workloads and diverse *PDF* document types while maintaining consistent performance and reliability standards that researchers can depend on for their important academic work. These improvements ensure that the system can serve as a reliable tool for academic researchers who need to efficiently collect and process large numbers of research papers for their studies. The enhanced capa-

bilities make the system suitable for use in both individual research projects and large-scale academic institutions that need to process thousands of papers efficiently and reliably.

# Chapter 6

## Performance Evaluation

### 6.1 Introduction

The effectiveness of the reference paper collection system should be rigorously evaluated through comprehensive performance assessment methodologies that examine both quantitative metrics and qualitative outcomes. This chapter presents a systematic experimental evaluation of the system's performance across multiple dimensions, including efficiency metrics, accuracy assessment, system usability and comparative analysis with manual collection methods and enhanced system versions. Performance evaluations will serve as a critical validation mechanism that demonstrates the practical value and effectiveness of the automated reference paper collection system. The evaluation framework encompasses both technical performance metrics such as response times, memory usage, and processing efficiency, as well as research-oriented metrics including collection accuracy, relevance assessment, and user satisfaction improvements [1]. The experimental evaluation is designed to answer several key research questions: First, does the *multi-threading* enhancement significantly improve system performance compared to single-threaded execution? Second, how does the system's accuracy in finding relevant papers compare to established academic search engines? Third, what is the user experience and perceived usability of the system among actual researchers? These questions guide the comprehensive evaluation presented in this chapter.

### 6.2 Evaluation Framework and Methodology

#### 6.2.1 Experimental Setup

The performance evaluation employs a comprehensive experimental design conducted in standardized testing environments with controlled hardware specifications and software configurations. This standardization ensures reproducible results and fair comparisons across different system configurations. **Test Environment Configuration:** all experiments are conducted on a personal computer equipped with *Intel Core i7-8550U CPU* at 1.80GHz with four cores and 16GB memory, running *Windows 11*. This hardware configuration represents a typical mid-range academic workstation, ensuring that the evaluation results are relevant to the system's intended user base. The four-core processor allows for effective

evaluations of the *multi-threading* capabilities, while the *16GB* memory provides sufficient resources for processing large numbers of academic papers. **Testing Framework:** the evaluation utilizes *Python*-based performance monitoring tools to measure system response times and memory usage across different operational scenarios [3]. Both *multi-threaded* and conventional single-threaded implementations are evaluated to assess the performance benefits of parallel processing. The *Python* testing framework includes custom scripts for automated performance measurement, ensuring consistent and accurate data collection across all test runs. **Evaluation Criteria:** the assessment focuses on three primary dimensions: (1) system performance measurement including response time and memory utilization, (2) accuracy evaluation for finding relevant reference papers, and (3) effectiveness and usability assessment through *System Usability Scale (SUS)* scores and user feedback. Each dimension is evaluated using established methodologies and metrics to ensure comprehensive coverage of the system’s capabilities.

## 6.2.2 Performance Metrics Definition

The performance evaluation begins with a detailed analysis of system performance in terms of response time and memory usage for both *multi-threaded* and conventional single-threaded implementations. I conduct these evaluations on a personal computer that is equipped with *Intel Core i7-8550U CPU* at *1.80GHz* with four cores and *16GB* memory, running *Windows 11*. To measure the performance metrics accurately, the built-in tool called *Performance Monitor* [74] provided by the *Windows* operating system is utilized. This tool provides real-time monitoring of *CPU* usage, memory consumption, and other system resources during the execution of our reference paper collection system. In the experiments, the system analyzed the first 10 pages of the search results from *Google Scholar*. This number of pages was chosen as it typically contains 100 search results, providing a substantial dataset for evaluation while remaining manageable for detailed analysis. For the *multi-thread* case, 10 threads were used so that each thread handled one page search in parallel. This configuration maximizes parallelization by assigning one thread per page, allowing for optimal distribution of the workload across available processing resources. Table 6.1 presents the comprehensive

Table 6.1: System performance results.

parameter	multi-thread	single-thread
<i>CPU</i> time (s)	41s	507s
avg. <i>CPU</i> usage (%)	10%	2%
avg. memory usage (%)	46%	45%

performance comparison between *multi-threaded* and single-threaded execution modes. The results demonstrate dramatic performance improvements achieved through *multi-threading* implementation. The *multi-thread* execution was completed in just 41 seconds with the *CPU* usage of 10%, contrasting sharply with the single-thread execution which required 507 seconds with the *CPU* usage of only 2%. This represents a remarkable 91.9% reduction in processing time, transforming the 8-minute wait into less than one minute of processing. When compared with the single-thread execution, the *multi-threaded* approach resulted in

an 8% increase in *CPU* usage and the reduction of 466 seconds in *CPU* time. The memory usage remained similar at 46% for *multi-threaded* and 45% for single-threaded execution, indicating that the parallelization does not significantly increase memory requirements [50]. The relatively modest increase in *CPU* usage (from 2% to 10%) demonstrates efficient resource utilization, as the system achieves over 12 times faster performance while consuming only 5 times more *CPU* resources. This efficiency is particularly important for academic environments where computational resources may be shared among multiple users. With the use of *multi-threading*, the *CPU* time becomes acceptable for interactive use, enabling researchers to obtain results quickly without disrupting their workflow.

### 6.2.3 Accuracy Evaluation

The second major component of this evaluation focuses on the system’s accuracy in finding relevant reference papers for users. I compare our proposed system with two widely-used academic search engines: *Google Scholar* and *Semantic Scholar* [75] across seven different test cases with varying research topics. *Semantic Scholar* is an *AI*-powered academic search engine that uses *machine learning* and *natural language processing* to help researchers find relevant academic papers across a wide range of disciplines. This comparison allows us to benchmark the system against both a traditional keyword-based search engine (*Google Scholar*) and a modern *AI*-enhanced search platform (*Semantic Scholar*). In each test case, I used identical sets of keywords for the proposed system, *Google Scholar*, and *Semantic Scholar* to ensure fair comparisons. The *system accuracy* represents the percentage of relevant accessible papers among all the papers suggested by the system and is calculated as follows:

$$\text{system\_accuracy} = \frac{\text{Number of relevant accessible papers}}{\text{Total number of suggested papers}} \times 100\% \quad (6.1)$$

where a relevant accessible paper represents a paper suggested by the system whose *PDF* file can be downloaded and whose contents are directly related to the research topic being searched. This dual requirement ensures that the system not only finds topically relevant papers but also provides access to their full content, which is essential for actual research use. In the seven test cases, the following three factors are systematically evaluated:

- **Relevance:** for *Google Scholar* and *Semantic Scholar*, I downloaded 100 papers from the first 10 pages of the search results. This standardized approach ensures consistent evaluation across different search engines [5]. For the proposed system, I downloaded all papers suggested by the system, as it typically returns a more focused set of results. The relevance of each paper was carefully judged based on three criteria: user preference alignment, direct interest for the target research topic, and content similarity with the research query. This multi-criteria approach ensures comprehensive relevance assessment.
- **Accessibility:** I compared the number of accessible and inaccessible papers within the first 10 pages of *Google Scholar* and *Semantic Scholar* results, and among all suggested papers by the proposed system. Accessibility is crucial for researchers, as papers that cannot be accessed provide no practical value. This metric captures the system’s ability to identify freely available papers or those accessible through institutional subscriptions.

- **Required time:** I measured and compared the total time required to complete the entire paper collection workflow, including: searching for candidates, opening the links to *PDF* files, downloading them, and reading them sufficiently to determine their relevance. For *Google Scholar* and *Semantic Scholar*, this process was performed for 100 papers, while for the proposed system, all suggested papers were evaluated. This time measurement reflects the real-world effort required by researchers to collect relevant literature.

We categorized the data in the table into five columns. The first column lists the **system** names. The second column shows the **# of papers** representing the total number of papers produced by each system. The third column, labeled **relevance**, contains two sub-metrics: the number of relevant papers (**# of papers**) and the percentage of relevant papers (**accuracy**). The fourth column applies the same structure to **accessibility**, showing both the count and percentage of accessible papers. Finally, the fifth column records the **time** required to evaluate whether all papers produced by each system are relevant.

### 6.2.3.1 Case 1: Web Scraping with Selenium

For *Case 1*, the search title was "web scraping using Selenium" with keywords including "Selenium", "ChromeDriver", "Python", and "data collection". This case represents a technical computer science topic focusing on web automation technologies [76]. Table 6.2 re-

Table 6.2: Accuracy results in *Case 1*.

system	# of papers	relevance		accessibility		time
		# of papers	accuracy	# of papers	accuracy	
<i>Google Scholar</i>	100	47	47%	58	58%	2hr 27min
<i>Semantic Scholar</i>	100	58	58%	82	82%	2hr 18min
<b>Proposal</b>	<b>33</b>	<b>25</b>	<b>75%</b>	<b>31</b>	<b>94%</b>	<b>45min</b>

veals significant advantages of the proposed system. While *Google Scholar* and *Semantic Scholar* returned 100 papers each, our system returned only 33 papers but with much higher quality. The relevance accuracy of 75% far exceeds both *Google Scholar* (47%) and *Semantic Scholar* (58%), demonstrating the effectiveness of our *natural language processing* approach in identifying truly relevant papers. More impressively, the accessibility rate of 94% shows that our system successfully identifies papers that researchers can actually access and use. The time savings are dramatic: just 45 minutes compared to over 2 hours for the traditional search engines, representing a 70% reduction in research time. This efficiency gain is achieved through both the smaller, more focused result set and the automated relevance assessment capabilities of our system.

### 6.2.3.2 Case 2: Sentence Transformers

and BERT For *Case 2*, the search focused on "Sentence transformer and sentence similarity using Bert model" with keywords including "embedding", "similarity", "attention layer",

"*Bert*", and "*NLP*". This case represents a current hot topic in *natural language processing* research [36]. Table 6.3 demonstrates consistent performance advantages across different

Table 6.3: Accuracy results in *Case 2*.

system	# of papers	relevance		accessibility		time
		# of papers	accuracy	# of papers	accuracy	
<i>Google Scholar</i>	100	41	41%	72	72%	2hr 18min
<i>Semantic Scholar</i>	100	38	38%	63	63%	2hr 5min
<b>Proposal</b>	<b>51</b>	<b>31</b>	<b>61%</b>	<b>51</b>	<b>100%</b>	<b>49min</b>

research domains. The proposed system achieved 61% relevance accuracy, significantly outperforming both *Google Scholar* (41%) and *Semantic Scholar* (38%). Remarkably, every single paper suggested by our system (100% accessibility) could be accessed and downloaded, compared to 72% for *Google Scholar* and 63% for *Semantic Scholar*. This perfect accessibility rate indicates that our system effectively filters out papers behind paywalls or unavailable sources during the collection process. The time efficiency remains excellent at 49 minutes versus over 2 hours for competing systems, maintaining the pattern of approximately 60-70% time savings.

### 6.2.3.3 Case 3: Mobile UI Testing

For *Case 3*, we searched for "Automated image-based *UI* testing for mobile applications" with keywords including "user interface", "testing", "Image", "CSS", and "mobile platforms". This case represents an interdisciplinary topic combining software engineering and mobile development. Table 6.4 shows that even for more specialized topics, our system

Table 6.4: Accuracy results in *Case 3*.

system	# of papers	relevance		accessibility		time
		# of papers	accuracy	# of papers	accuracy	
<i>Google Scholar</i>	100	33	33%	72	72%	2hr 12min
<i>Semantic Scholar</i>	100	22	22%	47	47%	1hr 58min
<b>Proposal</b>	<b>37</b>	<b>16</b>	<b>43%</b>	<b>34</b>	<b>91%</b>	<b>46min</b>

maintains its advantages as it is best in terms of accuracy in both relevance and accessibility. Although the relevance precision of 43% is lower than in previous cases, it still exceeds both *Google Scholar* (33%) and significantly outperforms *Semantic Scholar* (22%). The high accessibility rate of 91% continues to demonstrate the system's ability to identify freely available papers. The specialized nature of this topic may explain the lower overall relevance rates across all systems, as fewer papers directly address this specific intersection of technologies.

#### 6.2.3.4 Case 4: Indoor Navigation Systems

For *Case 4*, the search title was "Indoor navigation system" with keywords including "indoor navigation", "Unity", "QR code", and "smartphone". This case examines the system's performance in location-based technology research. Table 6.5 demonstrates that our system

Table 6.5: Accuracy results in *Case 4*.

system	# of papers	relevance		accessibility		time
		# of papers	accuracy	# of papers	accuracy	
<i>Google Scholar</i>	100	28	28%	71	71%	2hr 20min
<i>Semantic Scholar</i>	100	20	20%	49	49%	2hr 4min
<b>Proposal</b>	<b>58</b>	<b>29</b>	<b>50%</b>	<b>52</b>	<b>89%</b>	<b>57min</b>

achieves 50% relevance accuracy, nearly double that of *Google Scholar* (28%) and significantly better than *Semantic Scholar* (20%). The system returned 58 papers, a larger set than previous cases, reflecting the broader nature of indoor navigation research. Despite the larger result set, the processing time of 57 minutes remains well below the 2+ hours required by traditional search engines, maintaining approximately 60% time savings.

#### 6.2.3.5 Case 5: Docker Technology

For *Case 5*, I searched for "Docker image generation" with the single keyword "Docker". This case tests the system's performance with minimal search parameters [14]. Table 6.6

Table 6.6: Accuracy results in *Case 5*.

system	# of papers	relevance		accessibility		time
		# of papers	accuracy	# of papers	accuracy	
<i>Google Scholar</i>	100	23	23%	63	63%	2hr 31min
<i>Semantic Scholar</i>	100	11	11%	42	42%	2hr 24min
<b>Proposal</b>	<b>25</b>	<b>11</b>	<b>44%</b>	<b>22</b>	<b>88%</b>	<b>30min</b>

shows that even with limited search keywords, our system maintains superior performance. The 44% relevance accuracy nearly doubles *Google Scholar*'s 23% and quadruples *Semantic Scholar*'s 11%. The focused result set of only 25 papers and processing time of just 30 minutes demonstrates the system's efficiency in handling broad topic searches while maintaining quality.

#### 6.2.3.6 Case 6: IoT Platform Research

For *Case 6*, I used the specific paper title "A study of integrated server platform for *IoT* application systems" with keywords including "IoT platform", "cloud", "communication",

Table 6.7: Accuracy results in *Case 6*.

system	# of papers	relevance		accessibility		time
		# of papers	accuracy	# of papers	accuracy	
<i>Google Scholar</i>	100	28	28%	60	60%	2hr 42min
<i>Semantic Scholar</i>	1	0	0%	0	0%	10 sec
<b>Proposal</b>	<b>46</b>	<b>21</b>	<b>46%</b>	<b>37</b>	<b>80%</b>	<b>41min</b>

"smart cities", and "artificial intelligence". This case tests the system's ability to find papers related to a specific published work. Table 6.7 reveals an interesting pattern where *Semantic Scholar* returned only one result that was neither relevant nor accessible. This highlights a limitation of exact title matching in academic search engines. In contrast, this system successfully identified 46 related papers with 46% relevance and 80% accessibility, demonstrating its ability to find conceptually related work beyond exact matches.

### 6.2.3.7 Case 7: Web Programming Education

For *Case 7*, I searched for "A proposal of code modification problem for self-study of web client programming using *JavaScript*" with comprehensive keywords including "web client programming", "*JavaScript*", "*HTML*", "*CSS*", "code modification", "coding reading", and "self-study". Table 6.8 presents an interesting contrast where *Semantic Scholar* achieved

Table 6.8: Accuracy results in *Case 7*.

system	# of papers	relevance		accessibility		time
		# of papers	accuracy	# of papers	accuracy	
<i>Google Scholar</i>	100	26	26%	63	63%	1hr 50min
<i>Semantic Scholar</i>	2	2	100%	2	100%	1min
<b>Proposal</b>	<b>28</b>	<b>15</b>	<b>54%</b>	<b>25</b>	<b>89%</b>	<b>42min</b>

100% accuracy but only by returning 2 papers. While these papers were perfectly relevant, such limited results may not provide sufficient literature coverage for comprehensive research. The system balanced quantity and quality by returning 28 papers with 54% relevance, providing researchers with a more comprehensive literature base while maintaining reasonable accuracy.

## 6.2.4 System Usability Evaluation

The final component of this evaluation investigates the system usability through a structured questionnaire using the *System Usability Scale (SUS)* methodology [77]. I recruited 10 graduate students from our computer science laboratory to participate in a comprehensive usability study. Each participant was asked to install the proposed system on their personal computers using *Docker*, and then, to use the system by inputting titles and keywords

related to their current research topics. This approach ensured that the evaluation reflected real-world usage scenarios with genuine research needs. After completing their searches and reviewing the results, participants answered a standardized 10-question *SUS* questionnaire to provide structured feedback on their experience with the system.

Table 6.9: *SUS* questions in questionnaire.

#	Question
1.	The search input interface is easy to use.
2.	The system responds too slowly.
3.	The output result list is easy to understand.
4.	To get the good result, I tried to input several times.
5.	The output result papers are downloadable as <i>PDF</i> .
6.	Manual searching is more useful than this system.
7.	The result papers are relevant.
8.	I require technical support to set up the system on my <i>PC</i> .
9.	I would like to recommend this system to other people.
10.	I experienced bugs when using the system.

Table 6.9 presents the 10 *SUS* questions used in our evaluation. These questions are designed to assess various aspects of system usability including ease of use (questions 1, 3), performance (question 2), effectiveness (questions 4, 5, 7), comparative advantage (question 6), technical complexity (question 8), user satisfaction (question 9), and reliability (question 10). The mix of positively and negatively worded questions helps prevent response bias.

Table 6.10: Answers for *SUS* questionnaire and *SUS* scores.

User	Answers for <i>SUS</i> Questions										Overall <i>SUS</i> Scores
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	
1	5	4	5	3	5	3	4	1	3	1	75
2	5	2	5	3	5	1	4	1	5	1	90
3	5	2	3	2	3	3	4	1	3	4	65
4	4	1	5	2	4	2	4	2	4	2	80
5	5	4	5	3	5	3	3	2	4	4	65
6	2	5	4	3	5	4	4	5	4	4	45
7	4	4	4	4	4	2	4	2	4	1	67.5
8	4	2	4	2	4	2	5	2	4	2	77.5
9	5	5	2	3	1	4	2	3	3	5	37.5
10	5	1	4	2	4	1	5	3	5	3	82.5
Average <i>SUS</i> Score											68.5

Table 6.10 presents the detailed responses from all 10 participants along with their calculated *SUS* scores. The responses use a 5-point *Likert* scale where 5 indicates strong agreement and 1 indicates strong disagreement. The *SUS* scores range from 37.5 to 90, with an average score of 68.5. This wide range indicates varying user experiences, which our subsequent

analysis helps explain. Most users (7 out of 10) rated the search interface as easy to use (Q1: scores of 4 or 5), and the majority found the output results easy to understand (Q3). System performance was generally satisfactory, with most users disagreeing that the system responds too slowly (Q2). Importantly, 8 out of 10 users indicated they would choose this system over manual searching (Q6: low scores), validating the system’s practical value. Table 6.11

Table 6.11: *SUS* score interpretation.

<i>SUS</i> Score	Grade	Adjective Rating
>80.3	A	Excellent
68-80.3	B	Good
68	C	Okay
51-68	D	Poor
<51	F	Awful

provides the interpretation framework for *SUS* scores based on established usability research [78]. The average *SUS* score of 68.5 places our system at the boundary between ”Good” (Grade B) and ”Okay” (Grade C) categories. This indicates that while the system provides acceptable usability for most users, there remains room for improvement to achieve excellent usability ratings.

## 6.2.5 Feedback from Students

The qualitative feedback from participants provided valuable insights beyond the numerical *SUS* scores. Unfortunately, two students among the 10 participants gave F grades (scores below 51) in the *SUS* evaluation. Investigation revealed that both students used personal computers with *Macintosh Operating System (MAC OS)* featuring *Apple’s M1 silicon* chip, and encountered significant installation errors during the *Docker* setup process. The other eight students using *PCs* with *Intel* chips running either *Linux* or *Windows* operating systems experienced no installation errors and generally provided positive evaluations. To address this platform-specific issue, I immediately updated the installation manual with detailed instructions for *M1-based Mac* systems and provided the option for affected users to access the system through alternative *Windows-based* computers. This experience highlighted the importance of comprehensive cross-platform testing and documentation for academic software tools. Several constructive suggestions emerged from the user feedback:

- **Simplified Input Method:** some students suggested allowing keyword-only searches without requiring a title, as formulating an appropriate title can be challenging when exploring new research areas. This feedback indicates that the system could benefit from more flexible input options.
- **Publication Date Filtering:** multiple students requested the addition of publication date filters to limit search results to papers published within specific time periods. This feature would be particularly valuable for finding recent developments in rapidly evolving research fields.

- **Additional Search Refinements:** users expressed interests in advanced filtering options such as venue selection (conferences vs. journals), author filtering, and citation count thresholds to further refine search results according to their specific research needs.

Despite these suggestions for improvements, most students rated the proposed system as excellent or good, with many explicitly stating that it is more useful and accurate than manual paper collections using *Google Scholar*. Students particularly appreciated the time saving and the high relevance of returned papers, which significantly streamlined their literature review process.

## 6.2.6 Comparative Analysis Summary

Analyzing the results across all seven test cases reveals consistent patterns that validate the effectiveness of our approach:

1. **Time Efficiency:** the proposed system consistently reduced paper collection time by 60-80% compared to manual searching, transforming hours of work into minutes. This dramatic time saving enables researchers to focus more on reading and analyzing papers rather than searching for them.
2. **Relevance Accuracy:** the system achieved relevance rates between 43-75%, consistently outperforming both *Google Scholar* (22-47%) and *Semantic Scholar* (0-58%) across diverse research topics. This improvement stems from the *natural language processing* approach that better understands research context.
3. **Accessibility Excellence:** with accessibility rates of 80-100%, the system excels at identifying papers that researchers can actually access and use, addressing a critical pain point in academic research where many search results lead to paywalled content.
4. **Focused Results:** by returning smaller, more targeted sets of papers (25-58 papers vs. 100), the system reduces information overload while maintaining comprehensive topic coverage, allowing researchers to review all results thoroughly.

## 6.3 Performance Analysis and Discussion

### 6.3.1 Multi-threading Performance Gains

The experimental results demonstrate that the *multi-threading* implementation provides transformative performance improvements for the reference paper collection system. The 12.4x speedup achieved (from 507 seconds to 41 seconds) validates the architectural decision to parallelize *web scraping* operations [26]. This performance gain is particularly significant because it transforms the system from a batch processing tool to an interactive research assistant. The relatively modest increase in *CPU* usage (from 2% to 10%) indicates efficient parallelization that avoids common pitfalls such as excessive context switching or resource contention. The stable memory usage (45% vs. 46%) demonstrates that the implementation successfully manages shared resources without memory leaks or excessive duplication.

### 6.3.2 Accuracy and Relevance Improvements

The consistently higher relevance rates achieved by our system across all test cases validate the effectiveness of the *natural language processing* approach. By using *sentence transformer* and semantic similarity matching, the system better understands the conceptual relationships between search queries and paper content, leading to more accurate results than keyword-based matching alone [35]. The exceptional accessibility rates (80-100%) represent a significant practical advantage. Many researchers waste considerable time discovering that promising papers are behind paywalls. The system’s ability to prioritize accessible content can save researchers from this frustration while ensuring they can immediately access and read all suggested papers.

### 6.3.3 Usability Considerations

The average *SUS* score of 68.5 indicates that the proposed system achieves acceptable usability for most users, placing it at the “Good” threshold. However, the bimodal distribution of scores (with two failures due to platform issues) highlights the importance of robust cross-platform support for academic software. The positive feedback regarding time savings and result quality confirms that the system addresses real pain points in academic research. The suggestions for improvement, particularly around search flexibility and filtering options, provide clear directions for future enhancements.

### 6.3.4 Limitations and Future Improvements

Several limitations emerged from our evaluation:

1. **Platform Compatibility:** the installation difficulties on *M1 Mac* systems represent a significant barrier for some users. Future versions should include native support for *Apple Silicon* architecture.
2. **Search Flexibility:** the current requirement for both title and keywords may be too restrictive for exploratory searches. Implementing flexible input modes would improve usability.
3. **Limited Filtering Options:** the lack of temporal and venue-based filtering limits the system’s utility for certain research tasks. Adding these features would enhance the system’s versatility.
4. **Scalability Testing:** while this test used up to 10 pages of results, larger-scale evaluations with hundreds of pages would better demonstrate the system’s scalability limits.

## 6.4 Chapter Summary

This chapter presented a comprehensive performance evaluation of the proposed reference paper collection system through systematic experimental analysis across multiple dimensions including system performance, accuracy assessment, and user experience evaluation.

The evaluation employed rigorous methodologies with controlled testing environments using *Intel Core i7-8550U* hardware configuration representative of typical academic workstations. The performance evaluation results revealed dramatic improvements through *multi-threading* implementation, achieving a 12.4x speedup (from 507 to 41 seconds) with only a modest increase in *CPU* usage (from 2% to 10%). This transformation enables interactive use of the system, making it practical for real-time research activities rather than batch processing. In addition, accuracy evaluation across seven diverse test cases demonstrated consistent superiority over established academic search engines. The proposed system achieved relevance rates of 43-75% compared to 22-47% for *Google Scholar* and 0-58% for *Semantic Scholar*. More importantly, accessibility rates of 80-100% ensure that researchers can actually access the papers found, addressing a critical limitation of traditional search engines. The system reduced the total paper collection time by 60-80% across all test cases, transforming multi-hour literature searches into sub-hour tasks. This efficiency gain comes not from compromising quality but from returning smaller, more focused sets of highly relevant and accessible papers. Usability evaluation through the *SUS* methodology yielded an average score of 68.5, indicating "Good" usability for most users. While platform-specific issues affected some participants, the majority found the system more useful than manual searching and would recommend it to others. User feedback provided valuable insights for future improvements, particularly around search flexibility and filtering options. The comprehensive evaluation in this chapter establishes the reference paper collection system as an effective, efficient, and user-friendly solution for automated academic literature collection, demonstrating measurable improvements in processing speed, accuracy, and user satisfaction compared to traditional manual methods. These results validate the system's potential to significantly enhance research productivity by automating one of the most time-consuming aspects of academic work.

# Chapter 7

## Improved Version with Selenium Stealth

### 7.1 Introduction

In the previous chapters, I proposed the *reference paper collection system* using *Selenium* for *web scraping* on *Google Scholar* [10]. However, during extensive testing and deployment, I discovered a significant limitation that severely impacted the system's practical usability. *Google Scholar* employs sophisticated *anti-scraping* mechanisms that can actively detect and limit automated access attempts by blocking the *IP addresses* of the systems that appear to be collecting references through automated means [9]. The *anti-scraping* system is particularly sensitive to repetitive usage patterns from the same *IP address* and the same browser configuration, which essentially identifies requests as coming from the same computer. When such patterns are detected, *Google Scholar* blocks their access, making it impossible to continue collecting references. This blocking mechanism severely limits the effectiveness of our original system, as researchers cannot collect a sufficient number of references before being blocked. To overcome this critical challenge, I present an improved version of the *reference paper collection system* using *Selenium Stealth*. This enhancement allows the system to bypass the *human-bot differentiation* mechanisms employed during *web scraping* operations. *Selenium Stealth* is an advanced library that modifies browser behavior to make automated browsing appear more human-like [2]. It achieves this by using different browser configurations and different fingerprint methods every time it performs scraping operations, ensuring that each request appears to come from a different user. The system mimics natural human behaviors including realistic browser scrolling patterns and mouse clicking events, while also rotating *user agent* strings to further disguise its automated nature. For preliminary evaluations, I conducted comprehensive performance comparisons between the improved system using *Selenium Stealth* and the previous version when scraping *Google Scholar*. The evaluation results demonstrate significant improvements in system performance. Specifically, the enhanced system improved the success rate of collecting relevant references by 35% and achieved an impressive reduction in *CAPTCHA* challenges by 80% compared to the previous version [49]. These results strongly confirm the effectiveness of the proposed enhancement in overcoming *anti-scraping* mechanisms.

### 7.1.1 Challenges and Solutions

*Web scraping* from websites on the *Internet* presents numerous technical challenges that must be carefully addressed to ensure reliable data collection. These challenges have become increasingly complex as websites implement more sophisticated defense mechanisms against automated access. The primary challenges that *web scraping* systems face include dealing with dynamic website contents, overcoming *anti-scraping* mechanisms, handling inconsistent *HTML* structures [26], and managing various forms of authentication and authorization requirements. Dynamic content and missing data represent significant challenges that can be effectively addressed through careful preparation involving multiple adjustments and verification procedures. Modern websites often load content dynamically using *JavaScript*, which means that the initial *HTML* response may not contain all the data visible to human users. To handle these situations, comprehensive error handling must be implemented at each step of the verification process. Additionally, an automated notification system should be established to alert developers whenever website content changes in ways that could potentially affect the scraping process [1]. This proactive approach ensures that the system can be quickly updated to accommodate structural changes in target websites. In many cases, the data to be scraped includes not just text but also files and images embedded within websites. These multimedia elements require special handling during the scraping process. Therefore, it is essential to implement proper checks for file types and extensions to ensure that all relevant data is correctly identified and downloaded. The system must be capable of handling various file formats while maintaining data integrity throughout the collection process. Authorization and authentication present another layer of complexity in *web scraping* operations. Many websites restrict access to certain information, making it available only to logged-in users. This means that valuable research data may be hidden behind login walls. To access such protected information, the scraping system must be capable of handling authentication processes [29]. Login credentials and session management data should be prepared in advance and securely stored for cases where authentication is required. The system must also be able to maintain authenticated sessions throughout the scraping process to ensure continuous access to protected content.

## 7.2 Human-Bot Detection and Solutions

In this section, I provide a comprehensive introduction to the various technologies used for human-bot detection and explore effective solutions to overcome these detection mechanisms. Understanding these technologies is crucial for developing robust *web scraping* systems that can operate reliably in the face of increasingly sophisticated *anti-bot* measures.

### 7.2.1 CAPTCHA

Most modern websites have implemented *Completely Automated Public Turing test to tell Computers and Humans Apart (CAPTCHA)* services as their primary defense against automated access. These services work in conjunction with *IP blocking* and rate limiting mechanisms to prevent automated access and protect against spamming through inquiry

forms [79]. The *CAPTCHA* test is specifically designed to differentiate between computer-automated behaviors and genuine human interactions through various challenge-response tests. *CAPTCHA* systems allow human users to access websites normally while effectively keeping automated bots out. They serve as gatekeepers that protect all aspects of websites, including preventing unauthorized *web scraping* activities. The challenges presented by *CAPTCHA* systems can take many forms, from simple text recognition tasks to complex image selection puzzles, all designed to require human-level perception and reasoning capabilities that are difficult for automated systems to replicate.

## 7.2.2 Residential IP Proxies

To successfully bypass *anti-scraping mechanisms*, it is necessary to implement advanced techniques such as proxy rotation and *user-agent* rotation. Among these techniques, residential *IP* proxies have emerged as particularly effective tools. Residential *IP* proxies are special types of proxies that use real *IP addresses* assigned by *Internet Service Providers (ISPs)* to actual homeowners [28]. This crucial distinction means that web traffic routed through these proxies appears to originate from real residential users rather than from data centers or automated bots. The use of residential *IP addresses* significantly reduces the chances of being detected or blocked by target websites [4]. When a website receives requests from residential *IP addresses*, it typically assumes these requests come from legitimate home users browsing the internet normally. This assumption makes residential proxies particularly useful for several purposes: bypassing geographical restrictions that limit content to specific regions, avoiding *CAPTCHA* challenges that are often triggered by suspicious traffic patterns, and conducting *web scraping* operations stealthily without triggering *anti-bot* defenses.

## 7.2.3 AI Bots

The landscape of automated web interaction has been dramatically transformed by the emergence of *AI*-powered bots. Some modern bots are trained using sophisticated *machine learning* algorithms, making them far more advanced than traditional web scrapers and capable of solving many *CAPTCHA* challenges [11]. These *AI* bots can recognize patterns, understand context, and even mimic human decision-making processes with remarkable accuracy. In response to these advanced *AI* bots, websites have been forced to deploy increasingly strict security measures. These enhanced security systems often include complex tasks such as sophisticated puzzles and quizzes that require human-level interaction capabilities and, in some cases, even biological verification methods to grant access. The escalation in security measures reflects the ongoing technological arms race between bot developers and website defenders. A notable example of advanced human verification technology is the *World Network built by Tools for Humanity*. This cutting-edge technology takes verification to an unprecedented level by using a device called *Orb* that scans the iris of the human eye for verification purposes. This biometric approach represents one of the most advanced verification levels available in the emerging *AI* era, making it virtually impossible for even the most sophisticated bots to bypass. From the perspective of *web scraping* technology, solutions like the *Universal Scraper API* have emerged to handle these increasingly complex challenges [15]. These advanced *APIs* can manage dynamic content loading, handle

*JavaScript* rendering, overcome *IP blocking* mechanisms, and manage browser fingerprinting. Such comprehensive solutions represent the cutting edge of *web scraping* technology in response to modern *anti-bot* measures.

## 7.2.4 Scrapers and Defenders

The ongoing conflict between web scrapers and website defenders closely mirrors what is known as a *Cyber Arms Race*, where each side continuously evolves their techniques to outpace the other. This dynamic creates a constantly shifting technological landscape where innovations on one side quickly prompt countermeasures from the other. The arms race continues indefinitely as each side adapts to the other's latest advancements, driving rapid technological progress in both offensive and defensive capabilities. *Web scrapers* continuously develop more resilient and sophisticated techniques to overcome newly implemented defenses. These innovations might include better browser emulation, more realistic traffic patterns, or novel ways to distribute requests across multiple sources. Meanwhile, website administrators work tirelessly to enhance their *anti-scraping* mechanisms, implementing new detection algorithms, behavioral analysis systems, and access control measures [21]. This continuous interplay between attack and defense ensures that both sides are constantly evolving, driving significant technological innovations in both data extraction techniques and protection mechanisms. Each advancement on one side motivates further development on the other, creating a cycle of innovation that benefits the broader technology ecosystem. From the perspective of *web scraping* technology, *Selenium Stealth*, which is freely available to developers, represents one of the latest and most effective technologies for overcoming modern *anti-scraping mechanisms*.

## 7.3 Selenium Stealth Features

*Selenium Stealth* represents a significant evolution of the standard *Selenium* web automation framework. It is specifically designed to bypass the sophisticated *anti-bot* measures implemented on modern websites, allowing users to conduct *web scraping* and automation tasks while significantly reducing the detectability of automated browsers [31]. The key innovation of *Selenium Stealth* lies in its ability to mimic human behaviors, including realistic browser scrolling patterns, natural mouse clicking events, and intelligent rotation of *user agent* strings. In this section, we will describe four key technical features that make *Selenium Stealth* particularly effective.

### 7.3.1 Browser Fingerprinting Mitigation

Websites commonly employ browser fingerprinting techniques to identify unique characteristics of browsers and differentiate automated bots from human users. These fingerprinting techniques are sophisticated and multifaceted, analyzing various browser attributes including *user-agent* strings, *WebGL/Canvas* rendering data, installed system fonts, screen resolution, and many other technical parameters [80]. *WebGL* fingerprints, in particular, refer to a type of browser fingerprinting technique where websites or scripts gather unique characteristics of

a device's *WebGL* rendering capabilities to track or identify users. These fingerprints can be surprisingly unique, often allowing websites to identify specific devices with high accuracy. *Selenium Stealth* addresses this challenge by implementing comprehensive customization and randomization of these identifying attributes. The system modifies these characteristics to closely resemble valid browser profiles that would be expected from real human users. For instance, it can dynamically modify *WebGL* fingerprints to match common hardware configurations, randomize *user-agent* headers to reflect popular browser versions, and simulate broader browser capabilities that align with typical user systems. This sophisticated approach makes automation scripts appear much more human-like, significantly reducing the likelihood of detection by *anti-bot* systems employed by websites.

### 7.3.2 Headers and Cookies Management

*Anti-bot* systems have become increasingly sophisticated in tracking user behavior patterns to identify automation. These systems flag potential bots based on various behavioral indicators, including rapid actions that exceed human capabilities, repetitive patterns that suggest programmatic behavior, or instant navigation between pages without realistic delays. To counter these detection methods, behavioral mimicry has become a key feature of *Selenium Stealth* [76]. This behavioral mimicry system replicates natural human interactions with remarkable accuracy to avoid detection. The implementation includes several sophisticated techniques: randomizing delays between actions to simulate human thinking and reaction time, generating realistic mouse movement patterns that follow natural curves rather than straight lines, implementing realistic scrolling behaviors that vary in speed and pattern, and simulating hovering over elements before clicking them, just as humans naturally do when browsing. By carefully imitating these human behaviors, automated actions become virtually indistinguishable from real user activity, successfully bypassing many *anti-bot mechanisms* that rely on behavioral analysis.

### 7.3.3 Proxy and IP Rotations

Repetitive requests originating from the same *IP address* represent one of the most obvious red flags for *anti-scraping* systems. Such patterns often result in immediate *IP* blocks or the presentation of *CAPTCHA* challenges to verify human presence. *Selenium Stealth* addresses this critical issue by integrating sophisticated proxy management and *IP* rotation capabilities [27]. The system distributes requests across multiple *IP addresses*, effectively mimicking traffic patterns that would naturally occur from different users or geographical regions. This distribution strategy makes the automated traffic appear organic and dispersed rather than concentrated from a single source. The use of residential proxies or rotating proxy pools further enhances this strategy, as these proxies use *IP addresses* assigned to real residential internet connections. This approach ensures that *web scraping* activities remain undetected and do not trigger rate limits or *IP*-based bans that would otherwise interrupt data collection operations.

### 7.3.4 Anti-CAPTCHA Integration

Despite all prevention measures, many websites still deploy *CAPTCHA* systems as their last line of defense against bots. This reality makes it essential for *Selenium Stealth* to have strategies for addressing these challenges when they inevitably arise. The framework can integrate seamlessly with third-party *CAPTCHA*-solving services such as *2Captcha* or *AntiCaptcha*. These integrations allow scripts to automatically handle *CAPTCHA* challenges when they appear, maintaining the automation flow without manual intervention. When a *CAPTCHA* is encountered, the system can automatically capture the challenge, send it to the solving service, wait for the solution, and input the response, all while maintaining realistic timing patterns that don't trigger additional security measures. This comprehensive approach ensures that even websites with multiple layers of protection can be accessed reliably.

## 7.4 Configuration of Selenium Stealth

The implementation of *Selenium Stealth* begins with proper installation and configuration. First, I install the *Selenium Stealth* module using *pip*, the standard *Python* package manager. This installation process allows us to maintain compatibility with the existing *Selenium* framework while enabling advanced *stealth* features that mask automated browsing behavior. The installation preserves all standard *Selenium* functionality while adding the additional layer of anti-detection capabilities. To maximize the effectiveness of *Selenium Stealth*, we create a sophisticated function that generates randomized configuration options. This function returns different randomized options each time it is called, which is crucial for avoiding bot detection. By constantly varying the browser's characteristics, we prevent websites from identifying patterns that might reveal the automated nature of our browsing. The randomization process is described in detail in **Algorithm 7**.

---

**Algorithm 7** Generate random stealth configuration.

---


```
1: function RANDOMSTEALTHCONFIG
2:   user_agents ←
           {"Win10", "Win7", " MacOS", "Android"}

3:   webgl_vendors ←
           {"Intel", "NVIDIA", "AMD"}

4:   webgl_renderers ←
           {"Intel Iris OpenGL", "NVIDIA GeForce", "AMD Radeon"}

5:   platforms ←
           {"Win32", "MacIntel", "Linux x86_64"}

6:   languages ←
           [{"en-US", "en"}, ["ja-JP", "ja"], ["fr-FR", "fr"]}

7:   config.user_agent ← random choice from user_agents
8:   config.webgl_vendor ← random choice from webgl_vendors
9:   config.webgl_renderer ← random choice from webgl_renderers
10:  config.navigator_platform ← random choice from platforms
11:  config.languages ← random choice from languages
12:   config
13: end function
```

---

I have extensively modified the existing automation script to incorporate *Selenium Stealth* capabilities for *Chrome* browser automation. The enhanced browser runs in headless mode while carefully mimicking *Chrome*'s natural behaviors, including realistic screen resolution and window size parameters that match common user configurations. The *web-driver* configuration includes critical modifications that change the *web-driver* property of the navigator object to undefined, effectively masking the automation status from detection scripts. A comprehensive list of *user-agent* strings is defined within the system, representing various common browser and operating system combinations. During each session, a random *user-agent* is selected and applied to ensure comprehensive coverage and make the system appear as a genuinely random user. This randomization extends beyond just the *user-agent* string to encompass all aspects of the browser's identity. The *stealth* function is configured with multiple random properties to ensure comprehensive disguise of the automated nature. These properties include language settings that match the geographical location being simulated, browser vendor information set as *Google Inc.* to match genuine *Chrome* installations, operating system identification as *Windows* or other common platforms, and graphical settings adjusted to mimic common hardware configurations such as *Intel-based* systems. Each of these parameters is carefully chosen to create a coherent and believable browser profile. The actual scraping process initiates using the specially configured *Stealth ChromeDriver*. Once the stealth measures are in place, the core functionality of scraping *PDF* links from *Google Scholar* and calculating similarity between user input and *PDF* data remains consistent with our previously proposed system, ensuring that users experience the same high-quality results while benefiting from improved reliability.

## 7.5 Preliminary Evaluation

In this section, I present a preliminary evaluation of the enhanced system’s performance in preventing *IP* bans and improving the paper collection success rate when scraping *Google Scholar*. To ensure a fair comparison with the previous system, I conducted searches using identical parameters. Specifically, I searched for reference papers using the paper title “*Web scraping using Selenium Stealth*” along with the keywords “*Selenium Stealth*”, “*ChromeDriver*”, “*Python*”, “*anti-scraping*”, and “*data collection*”. This standardized approach allows for direct performance comparison between the two systems.

### 7.5.1 Evaluation Indices

For this comprehensive evaluation, I carefully measured the following three critical performance indices that directly reflect the system’s effectiveness in overcoming *anti-scraping* measures:

- **IP ban resilience:** This metric measures the system’s ability to avoid detection and subsequent *IP* blocking. I recorded the total time elapsed from the initiation of scraping operations until the system’s *IP address* is banned by *Google Scholar*. A longer operational time before ban indicates superior performance in avoiding detection. This metric is crucial because once an *IP* is banned, the system becomes completely non-functional from that network location, requiring complex workarounds or waiting periods before resuming operations.
- **Success rate:** This metric quantifies the reliability of the data retrieval process. I measured the percentage of successful data retrieval operations among the total number of attempted operations. The success rate calculation includes all attempts to access pages, extract paper information, and download *PDF* files. A higher success rate indicates better overall system performance and reliability. This metric directly impacts the user experience, as failed operations mean missing potentially valuable research papers.
- **CAPTCHA Challenges:** Many websites, including *Google Scholar*, use *CAPTCHA* challenges as a defensive mechanism to verify whether requests are coming from human users or automated bots. I carefully counted how often *CAPTCHA* challenges appeared during the scraping process. A lower frequency of *CAPTCHA* challenges indicates better performance in mimicking human behavior. Frequent *CAPTCHA* challenges significantly slow down the data collection process and may require manual intervention, defeating the purpose of automation.

### 7.5.2 Results

Table 7.1 presents the comprehensive results of our performance evaluation. This table clearly demonstrates that the proposed system using *Selenium Stealth* significantly outperformed the previous system using standard *Selenium* across all three evaluation indices. The evaluation results reveal remarkable improvements in all measured aspects. First, the time until an *IP* ban occurred was increased by more than six times, extending from merely 5

Table 7.1: Performance index results.

index	previous	proposal
<i>IP</i> ban time	5 min.	> 30 min.
success rate	60%	95%
<i>CAPTCHA</i>	10 per 20 pages	2 per 20 pages

minutes with the previous system to over 30 minutes with the enhanced system. This dramatic improvement means researchers can conduct extended scraping sessions without interruption, collecting significantly more papers in a single session. Second, the success rate showed substantial improvement, increasing from 60% to an impressive 95%. This 35 percentage point improvement means that users can rely on the system to successfully retrieve almost all targeted papers, with only a 5% failure rate compared to the previous 40% failure rate. This reliability is crucial for researchers who need comprehensive literature coverage. Third, the frequency of *CAPTCHA* challenges was dramatically reduced by 80%, dropping from 10 occurrences per 20 pages to just 2 occurrences per 20 pages. This reduction significantly improves the automation efficiency, as fewer manual interventions are required during the scraping process. These results powerfully highlight the efficiency of the stealth measures implemented in our enhanced system. However, it is important to note that *Selenium Stealth* still cannot completely eliminate *CAPTCHA* challenges. For future improvements, the integration of supplementary tools such as automated *CAPTCHA* solving services should be considered to achieve fully autonomous operation.

## 7.6 Conclusion

This chapter has presented a significant enhancement to the *reference paper collection system* through the implementation of *Selenium Stealth* technology to overcome persistent *anti-scraping challenges*. The enhancement addresses critical limitations discovered in our previous system, particularly the aggressive *IP blocking* mechanisms employed by *Google Scholar* that severely limited the system’s practical usability. The preliminary evaluations have demonstrated remarkable improvements across all measured performance indices. The enhanced system increased the operational time until *IP* ban by more than six times, extending from 5 minutes to over 30 minutes of continuous operation. The success rate of collecting relevant references improved dramatically by 35 percentage points, rising from 60% to 95%. Additionally, the system achieved an 80% reduction in *CAPTCHA* challenges, decreasing from 10 to just 2 occurrences per 20 pages of results. These improvements represent a substantial advancement in automated academic paper collection technology. The enhanced system now provides researchers with a reliable, efficient tool that can operate for extended periods without interruption, successfully retrieve the vast majority of targeted papers, and minimize manual intervention requirements. The implementation of *Selenium Stealth* has effectively transformed our reference paper collection system from a proof-of-concept tool into a practical solution suitable for real-world research applications. Future works will focus on achieving complete automation by integrating *CAPTCHA*-solving services and exploring additional anti-detection techniques as the arms race between web scrapers and defenders continues to evolve. The success of this enhancement demonstrates the

importance of adaptive development in *web scraping* technologies and provides a foundation for continued improvements in automated academic research tools.

# Chapter 8

## Automated Testing with Selenium

### 8.1 Introduction

This section introduces an additional contribution regarding an application of *Selenium*. *Selenium* can be used not only for an *web scraping* data collection tool but also for a testing tool of a web application system.

For the client-side web programming, *JavaScript* is mainly used to develop dynamic web-sites. It is a client-side scripting language and allows responsive designs on different devices by accessing to the Internet. Most software companies require developers to adopt *JavaScript* web application programs for both front-end and back-end [38]. To help IT programming students study *JavaScript* programming, *PLAS* (*Programming Learning Assistant System*) supports *JavaScript* programming exercise problems. *PLAS* is an educational platform that supports various programming languages. There are several types of exercise problems for each programming language, such as the *grammar concept understanding problem* (*GUP*), the *element fill-in-blank problems* (*EFP*), the *value-trace problems* (*VTP*), the *code modification problems* (*CMP*) [81], and the *code writing problems* (*CWP*) [82].

In this thesis, I design and implement a unit testing tool for client side web programming in *code writing problems* (*CWP*). It aims at students to study *HTML*, *CSS*, and *JavaScript*. The system will perform automated testing for the code according to the testing steps in each problem and generate the testing reports. It helps students to know that their codes are passed or failed within testing steps. It is recommended to read the manual guide before the students answer the problems, because there are specifications and requirements that should be followed.

For the fill-in-blank type questions in *PLAS*, it can automatically respond to users which answer is right or wrong. To overcome human intervention and the time-consuming process of checking the answers manually, we implement automated testing, a form of unit testing in WebPLAS. Testing is an important phase in the software development life cycle. Testing makes sure software and system are well-performed under different situations and circumstances. Automated testing can reduce time and loads for manual test by admins or teachers. Moreover, it can be reusable for repetitive tasks.

*Selenium* [83] provides a kind of automated testing for client-side web applications. *Selenium* controls the web browser to perform automated tests according to the test scenarios

[76]. It can handle clicking certain buttons and triggering events on the web-browser. To create test cases, we use the *Behave* framework as a *Behavior Driven Development (BDD)* [84]. Test scenarios are written using *Gherkin* keywords [85]. As for the test results, we use the *Allure* report [86] for presenting test results to students.

## 8.2 Design of Unit Test Tool for WebPLAS

In this section, I design the unit testing tool for *WebPLAS*. It will explain how the *Python Flask*, *Selenium*, *Behave*, *Allure*, and *Docker* are integrated to achieve the whole system.

### 8.2.1 Python Flask

*Flask* is known as a *WSGI* framework that stands for Web Server Gateway Interface. Essentially, this is a way for web servers to pass requests to applications or web frameworks. *Flask* relies on the external *WSGI* library and the *Jinja2* templating engine [38]. In *PLAS*, Python *Flask* framework is used to render the problem page and to serve API (Application Programming Interface) for testing. Firstly, in order to serve the problem files, I use the file system by reading the file and showing the content of the problem. Then, I render the problem page by *Flask*. The test automation process in the system is carried out by using *Selenium Python*. It also acts as the backend service for all the requests handling and rendering the html page using *Jinja2* [87] template engine which can serve static files. *Python* has a service of OS command scripts with *Subprocess* module. It can run *Selenium*, *Behave* and *Allure* command to perform test automation in server side and render the result pages for submitted answers. *Flask* applications can also perform the followings:

- Include development server and debugging functions.
- Enable referencing to the other python modules in python scripts.
- Handle API RESTful request and implementation of middleware.
- Help HTML render support with *Jinja2* templating.
- Store cookies and sessions.
- Lightweight and flexible so that it can be shaped easily into customize preferences.

*Flask* is lightweight for initial setup and only includes these basic dependencies which are installed together.

- *Werkzeug* is a standard Python interface between client and server.
- *Jinja* is a template language that renders the html page by displaying the data from server side.
- *MarkupSafe* comes with *Jinja*. When rendering the templates, the escaping feature in *Jinja* helps to avoid injection attacks.
- *ItsDangerous* signs data to ensure its integrity and security. This is used to protect *Flask*'s session cookie and also useful for carrying the information of users

Moreover, I need a virtual *python3* environment for running the *Flask* server. I will use *docker* for containerization, I need to firstly create the python virtual environment by using “python3 -m venv venv” which is used for Linux platform because I aim to be containerized via *docker* later. Then, I activate venv by running the activate executor under the bin file, so that the *Flask* server will run in the Python virtual environment and it is also easy to containerize and easily maintainable for package management for further dependencies.

### 8.2.2 Jinja2

*Jinja2* [87] is a *template engine* which is commonly used in the python flask framework. It helps to format the content of the html page that wants to be rendered from server side and can carry the data which is passed along with that content. Programmatic syntax like looping, conditional checking can be used from inside templates and from Python code.

### 8.2.3 Selenium

*Selenium* is an open-source framework and includes packages of libraries and tools for browser automation. It is one of the RPA (Robotic Process Automation) traditional automated workflows that can customize user journeys [31]. It also supports different languages such as C#, Java, Ruby, Python. Selenium has software packages which are *Selenium IDE*, *Selenium Webdriver* and *Selenium Grid* [84].

- *Selenium IDE* can create simple test scripts and allows cross-browser execution. But it only supports Chrome, Firefox browsers and Programming knowledge is not required.
- *Selenium Grid* works together with *Selenium Remote Control (RC)* to run scripts on multiple browsers and operating systems. It also can test web applications with complex AJAX [88]-based scenarios. It works with routing commands.
- *Selenium Webdriver* supports different browsers like Safari, Microsoft Edge, Chrome, Firefox. Among Web-drivers, I can choose specific browser types such as ChromeDriver for Chrome, GeckoDriver for Firefox, SafariDriver for Safari test-scripts can be also written with so many programming languages since selenium supports C, C#, Python, Ruby, Java and .NET.

In *PLAS*, I deployed a *Selenium webdriver* to test Javascript web applications. *Selenium webdriver* [89] can create customized test scripts. Furthermore, there are no complex commands and can reduce server setup time as *Selenium Grid*. Moreover, it allows the automation of browsers and DOM (Document Object Model) elements. By calling DOM elements with id names, button names and *XPath* [48] locators, it can handle events or actions on browsers and is more convenient to test browsers-based web applications. Despite so many advantages of *Selenium webdriver*, it can't generate test reports to users. It should be integrated with the help of other frameworks to generate test reports to know the failed or passed test results to students.

## 8.2.4 Behave Framework

*Behave* [85] is a tool used for Behavior driven development (BDD) in Python. BDD creates a culture where testers, developers, business analysts, and other stakeholders of the project contribute towards the software development for requirements definition [90]. There are numerous libraries like the *Mocha* which supports *JavaScript*, *Cucumber* which supports *Ruby*, and *Behave* which supports *Python*. In *WebPLAS*, I use the Behave Framework for test scenarios and test scripts. The test scenarios are written in human readable language. Thus, it can be shown to students as a requirement specification. The students must develop the web application according to requirements that will be tested under a certain test scenario. It is used *Gherkin* language starting with “Given-when-then-and” keywords.

- Given - used for under certain state.
- When - the event is triggered
- Then - the expected outcome
- And - More outcome [Optional]

The file extension is (.feature). Each *Gherkin* language keyword and sentence in (.feature file) is related with step implementation(.py file ). Moreover, (.feature) file should be run to execute the test cases by the *behave* command. A brief explanation of the *behave* framework which is used in our *WebPLAS* system is as follows:

Listing 8.1: Feature file for Sample Program.

---

```
1 Feature: Test Alert Function is shown on button click event
2 Scenario: Flow of button click event for Alert Message
3 Given the button is shown on the alert html page
4 When the greeting button is clicked
5 Then the alert message should be displayed
```

---

Listing 8.1 is the simple feature of the test scenario. This test scenario information is provided to the students as the problem requirements in questions. As it is written with human readable language and *Gherkin* keywords, the students can understand how they should develop. In Listing 8.2 for step implementation (.py) file, the annotations of *Gherkin* keywords and string of each sentence are matched with each other. Furthermore, the error message will be shown if the students failed in certain steps of conditions by raising an exception. If the student failed in the first test case, the next step will not be executed.

After the *behave* command is carried out, the result of the test output is shown in the output console of the respective IDE or command prompt. However, Behave cannot generate test reports for detailed results. It provides JSON, plain text files and xml files etc. In order to generate test reports, the integration with the third party framework is necessary for further implementation.

Listing 8.2: Step implementation file for sample test case.

---

```
6 @given('the button is shown on the alert html page')
7 def test_open_html(context):
8     context.HelperFunc.open('http://localhost:5000/answerfiles/sample.
      html')
9     buttonelement = context.HelperFunc.find_by_id('btngreeting')
```

---

```

10     if buttonelement:
11         pass
12     else:
13         raise Exception("Initial button should exist")
14
15 @when('the greeting button is clicked')
16 def test_input_click(context):
17     context.HelperFunc.find_by_id('btngreeting').click()
18
19 @then('the alert message should be displayed')
20 def test_answer(context):
21     try:
22         context.HelperFunc.wait_until()
23         alert = context.HelperFunc.switch_to()
24         alert.accept()
25     except TimeoutException:
26         assert False, 'There is no alert Message!'

```

---

## 8.2.5 Allure Report

I describe *Allure* report or a test report tool for providing detailed test results with user interface. It also allows multi-programming languages and multi-test frameworks such as *JUnit* and *TestNG* for Java, *PyTest*, *Behave* and *Nose* for Python, *Jasmine* and *Mocha* for JavaScript [86]. Allure report is also suitable for a big test-suite for software companies. Because it has an interactive user interface for test information. In WebPLAS, I use *Behave* integrating with Allure Report for presenting student results with User Interface. You can see the overview of test results as shown in Figure 8.1.

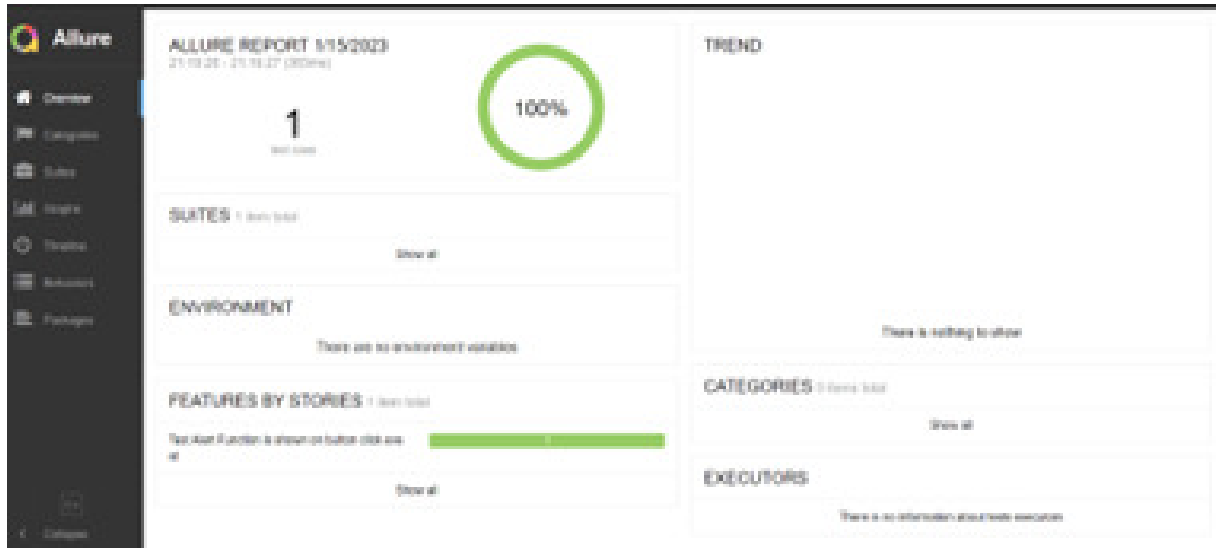


Figure 8.1: Overview of allure test report.

In Figure 8.2, the feature name in Behaviors Tab can be seen. After clicking the feature name, the scenario name and the information of passed/failed test result flow and passed test steps will be shown in the test body. Moreover, the number of attempted records for the problem can be seen on the retry tab.

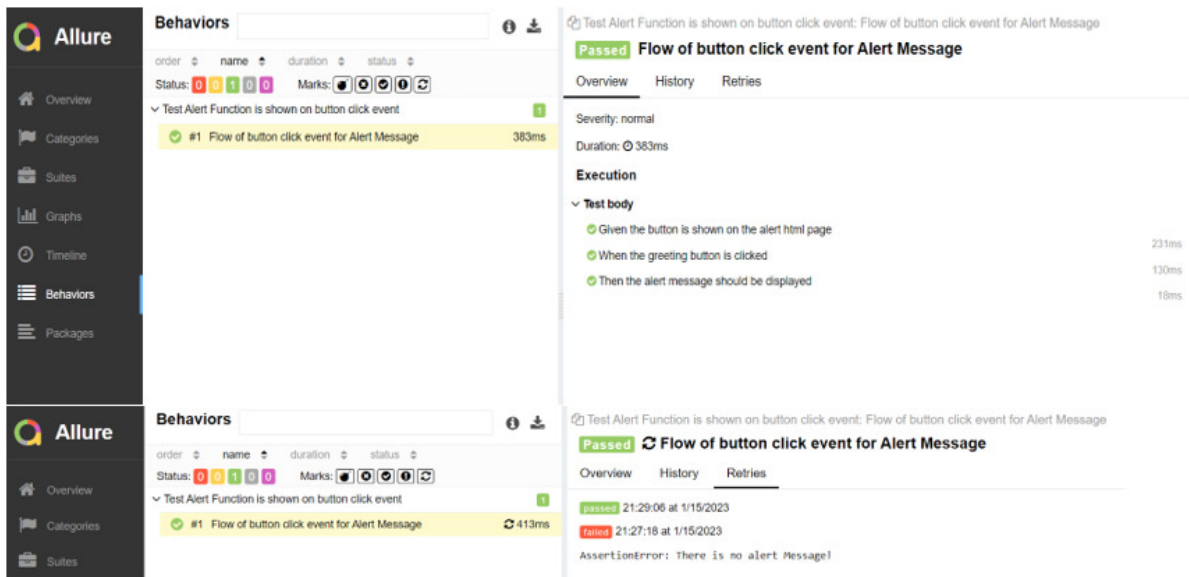


Figure 8.2: Test result in behaviors tab.

## 8.3 Implementation of Unit Test Tool

In this section, I implement the unit test tool for *WebPLAS*.

### 8.3.1 File Structure of Unit Test Tool

The system is created with a lightweight python Flask server for efficient ways of web development. I used *Selenium webdriver* for automation, Behave for test scenarios and Allure for test report by integrating in a python virtual environment. Firstly, the file structure of the system will be constructed. I properly divide the client side and server side in the Flask.

1. Client side: two HTML pages (home.html and index.html) under template folders are mainly rendered when the user calls the localhost:5000.
  - home.html: the first page when the student accesses the system. It shows all problems for practicing JavaScript Web Client Programs.
  - index.html: common template file for presenting problem detail page. I use a file system for reading all the problem details under “problemfiles” folder. Certain problems details are rendered according to the client-side request through the browser’s URL.
2. Server side: “app.py” under the project directory which mainly runs the server and includes route handling functions for server requests and it is written based on the *Flask* framework.
  - features and steps: feature files and steps implementation python files are located under the “steps” folder.
  - helper, environment.py and setup.cfg: They are used to connect between the Selenium and Behave for setting the necessary environment and the browser type such

as chrome, firefox, safari etc. “helper.py” implements the functions of controlling the DOM (Document Object Model) elements. “webdriver.py”, “setup.cfg” and “environment.py” python modules define the chromedriver file path and start the chromedriver with initial control parameters such as setting the browser width and height, granting camera access permission etc.

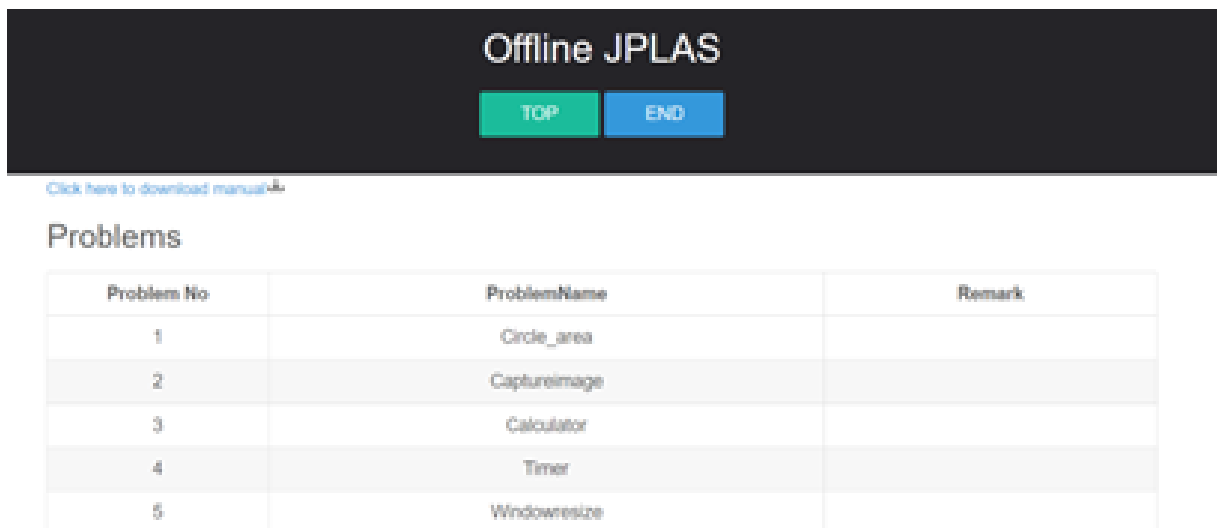
- answerfiles: this folder is used to save the student’s submitted answers.
- reports: the JSON files are saved in this folder after running the Selenium-Behave and performing test automation. After that, the system carry out the file production by running allure command and generating the result report.
- Dockerfile and requirements.txt: the dependencies of the required python-flask environment are defined with the step by step instructions. At first, All the libraries such as Selenium, Behave, and Allure are installed and imported as the requirements.txt. In the Docker file, it easily refers to the file “pip install -r requirements.txt”.

## 8.3.2 System Flow Demonstration

I would like to demonstrate one of the System Flow of JavaScript problems.

### 8.3.2.1 Problem lists and Requirements

In Figure 8.3, I made 5 instances for students. Students can read the manual of the JPLAS to understand the requirements of the system before implementing the problems.



The screenshot shows a web interface titled "Offline JPLAS". Below the title are two buttons: "TOP" (green) and "END" (blue). Below these buttons is a link: "Click here to download manual". Underneath the link is the heading "Problems". Below the heading is a table with three columns: "Problem No", "ProblemName", and "Remark". The table contains five rows of data.

Problem No	ProblemName	Remark
1	Circle_area	
2	Captureimage	
3	Calculator	
4	Timer	
5	Windowresize	

Figure 8.3: Problem lists of JPLAS.

As for the calculator problem, the student has to write the whole HTML page according to the question in Figure 8.4. They should refer to the detailed specifications after clicking the

“Show Specification” button. In Figure 8.5, the calculator layout and other specifications such as “Hint”, “Requirement” and “Test Scenario” are described on the popup screen. As explained in the Behave framework, I added the test scenario. According to the calculator problem, the system requirements of the problem are as follows.

- DOM elements such as button or input tag name should be set as `<button id="btn0" />` or `<input id="btn0" type="button" />` according to Figure 8.5 of “id name” requirements
- Ids of DOM elements should be careful as it is case-sensitive.
- The student should implement arithmetic functions and clear events with JavaScript according to the test scenario.



Figure 8.4: Calculator project problem instruction.

Before submitting the answer, the student can preview his/her source code by clicking the “Run Code” button in Figure 8.6.

### 8.3.2.2 Server Side Handling

After submitting the answers with the “Get Result” button, the system will request the server with the *answer* route as in Figure 8.7.

- Firstly, the server side handles the request by saving the code file under “answerfiles” folder.
- The system performs automated testing by using the behave command. It runs the *calculator.features*. *behave.formatter* returns the preferred format such as JSON pretty or plain text etc. *AllureFormatter* command produces the certain *JSON* format with necessary information for Allure under the *reports* folder.

Offline JPLAS

Requirements for JavaScript program

Calculator Project

				C
1	2	3	/	
4	5	6	-	
7	8	9	+	
.	0	=	*	

Text input type id Name

☒ Output Result id = textResult

Button input type id Name

☒ 0 = btn0

☒ 1 = btn1

☒ 2 = btn2

☒ 3 = btn3

☒ 4 = btn4

☒ 5 = btn5

☒ 6 = btn6

☒ 7 = btn7

☒ 8 = btn8

☒ 9 = btn9

☒ C = btnClr

☒ + = btnPlus

☒ - = btnMinus

☒ \* = btnMultiply

☒ / = btnDivide

☒ = = btnEqual

☒ . = btnDot

Fig : Layout and Naming in Input id

**Hint**

For layout interface, you can simply create your UI according to figure. Table layout and onclick function is used to getting the inputs.

**Requirement**

All numbers and operators(add, subtract,divide, multiply, equal,dot, clear) should be input **button** type except result output need to be input **text** type  
 In order to test your html, you need to test text input id and button id according to figure.  
 As it is case sensitive, you need to be careful of upper or lower case naming in button id.

**System will test your answer according to below scenario.**

Scenario : Calculator Function Flow

Then there should be blank in the result at the initial display state

When user input "2" and "3" to calculator to add

Then user get addition result "5"

Then user cleared the text

When user input "5" and "3" to calculator to subtract

Then user get subtraction result "2"

Then user cleared the text

When user input "2" and "4" to calculator to multiply

Then user get multiply result "8"

Then user cleared the text

When user input "8.8" and "4" to calculator to divide

Then user get division result "2.2"

Figure 8.5: Test scenarios of calculator problem.

- After that, JSON results are used to generate the test report HTML by using the *allure generate and allure-combine* command.
- All of the commands are performed by using the subprocess module in *python* as a terminal of the OS. After all subprocesses are finished, the system responds with HTTP 200 OK success status response.

### 8.3.2.3 Allure Test Result Report

For the student perspective, the result page will appear in a new tab. The submitted code is passed according to the test cases. All test cases are performed step by step. If one step is failed, it stops testing for the next step and shows the error message. The students can see all their attempts and histories of the practices in this report result.

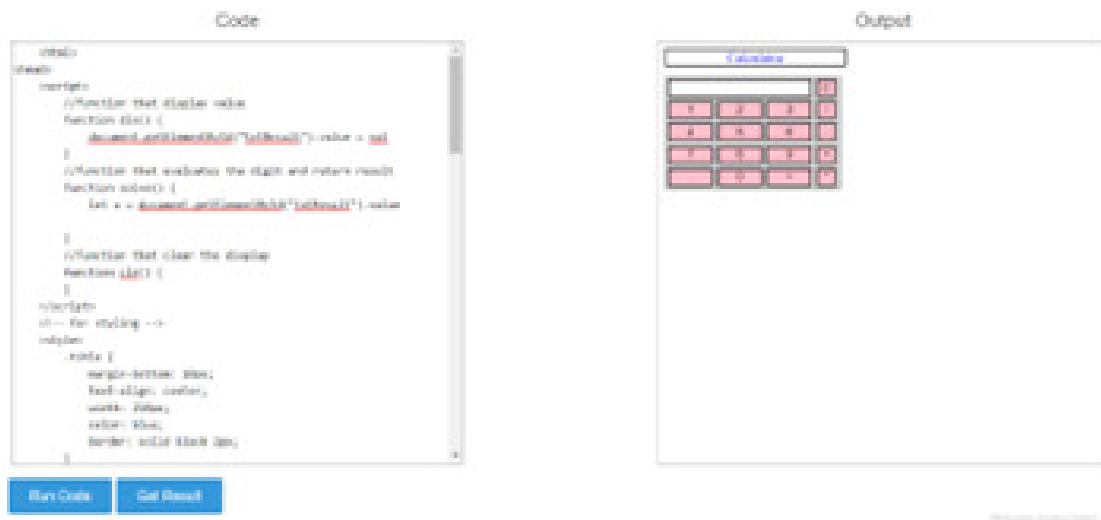


Figure 8.6: Preview page for calculator problem.

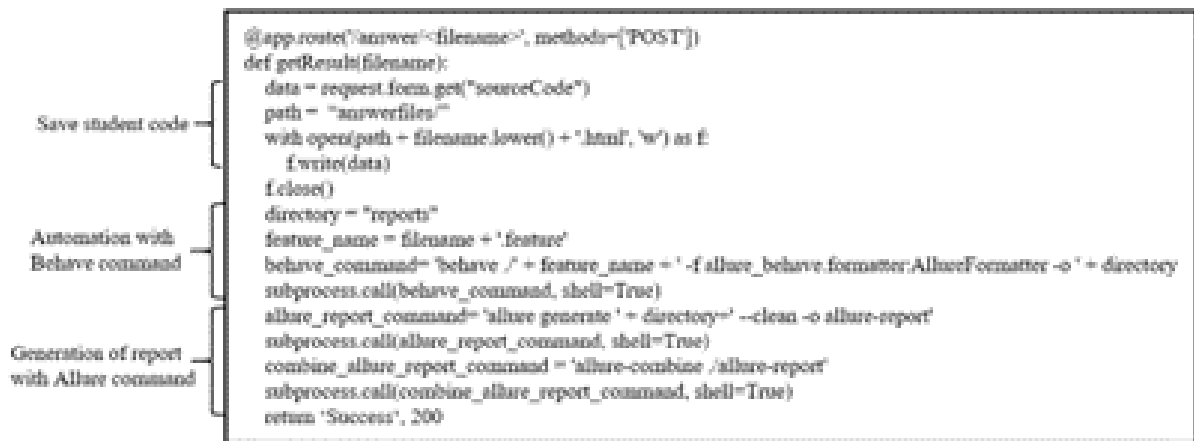


Figure 8.7: Server side handling for automation testing.

### 8.3.2.4 Unit Test tool Installation

The students need to install the Docker before installing the unit test tool [14]. They have to run the Docker file to create the Docker image according to the system manual file and installation steps in the Github Readme file. Brief installation steps are described as follows.

- Create a *SeleniumBDD* local folder and copy the project of the github repository under that directory.
- Navigate to the project folder and build the Docker by using the *Docker build* command.
- Run the *Docker images* command to see the created Docker images.
- Run the Docker image on localhost:5000 port by using the *Docker run -dp 5000:5000 Dockerimagename* command. They are able to practice client web programming problems on their PCs.

## 8.4 Conclusion

In this chapter, I implement a testing tool of a web application system by combining automation and test report using *Selenium*. I reduce time consuming tasks for teachers and students by applying precise testing methods of *Selenium* and behave integration.



# Chapter 9

## Related Works in Literature

In this chapter, I introduce related works to this thesis.

### 9.1 Automated Reference Paper Collection Systems

In [91], Jian et al. presented the *CiteSeerX* milestone as an open-access-based scholarly Big Data service. *CiteSeerX* collected data using an incremental web crawler developed with *Django*. Metadata processing used Perl with a multithreaded Java wrapper that was responsible for retrieving documents from the crawl database and managing parallel tasks. They reported that almost 40–50% of the total PDF documents successfully found per day and identified as scholarly are stored in the database. In [23], Petr et al. introduced *CORE*, a service that provides open-access to research publications through bulk data access and downloads them via RESTful APIs. The system ensures up-to-date data with the *FastSync* mechanism and manages all the running tasks through the *CORE Harvesting System (CHARS)*. However, this paper highlights the limitation of determining the optimal number of workers, as allocation relies on empirical methods. In [92], Jason et al. introduced *OpenAlex*, an open bibliographic catalogue covering a wide range of scientific papers and research information. *OpenAlex* gathers heterogeneous directed graph datasets which collected data from *Microsoft Academic Graph*, *Crossref*, *PubMed*, institutional repositories, and e-prints. *OpenAlex* used async in JavaScript to handle asynchronous operations and parallel-transform to process data streams in parallel. In [93], Patricia et al. proposed *PaperBot*, an opensource web-based software for automating the search process and indexing biomedical literature. It retrieved metadata and PDFs using multiple database APIs. Using MongoDB, it achieves scalable cluster operations to enable parallel processing and reducing latency. The results showed the increase in the throughput by five times in paper collections. However, storing PDF files as blobs led to increasing the RAM usage. In [94], Claudio et al. proposed *OpenAIRE* workflows for data managements to support open science e-infrastructure, allowing shares and reuses of research products. *OpenAIRE* utilized a graph-based *Information Space Graph (ISG)* to collect heterogeneous data from semantically linked research products, while the *D-Net* framework orchestrated workflows. *ISG* used *Apache HBase* for scalable columnar storage and *MapReduce* for parallel processing of large graph datasets. In [22], Marcin et al. introduced *Paperity Central*, a global universal catalogue combining gold and green

open-access scholarly literature. *Paperity Central* used an automatic aggregator to accelerate collections of large-scale new publications from open-access sources and repositories while connecting various data items based on semantics. However, this paper has not discussed how the aggregation mechanism distributes tasks to workers while retrieving metadata from various data sources. In [95], Fareedi et al. presented *Semantic Fusion of Health Data: Implementing a Federated Virtualized Knowledge Graph Framework Leveraging On-top System*. They introduced a hybrid ontology-based design science research engineering (ODSRE) methodology that combines design science activities with ontology engineering principles. The proposed *Federated Virtual Knowledge Graph (FVKG)* framework leverages the on-top virtual paradigm to establish seamless data integration and semantic interoperability across distributed relational data sources, addressing critical challenges in automated academic literature collection. In [6], Gusenbauer et al. conducted a comprehensive evaluation study titled *Which academic search systems are suitable for systematic reviews or meta-analyses? Evaluating retrieval qualities of Google Scholar, PubMed, and 26 other resources*. The research systematically analyzed 28 academic search systems to determine their suitability for systematic literature reviews. The study provided critical insights into the retrieval qualities of various academic databases and search engines, establishing quality requirements for comprehensive literature searches.

## 9.2 Web Scrapping Technologies for Academic Applications

In [96], Sharma et al. introduced *An Improving Approach for Fast Web Scrapping Using Machine Learning and Selenium Automation*. The research utilized Selenium automation beyond traditional testing purposes to create automated web element identification systems. The study employed Machine Learning techniques including multinomial naïve Bayes classifier and bag-of-words methods to classify documents industry-wise, demonstrating significant improvements in automated data extraction efficiency. In [9], Davidson et al. presented *Web Scrapping for Research: Legal, Ethical, Institutional, and Scientific Considerations*. This comprehensive study addressed the critical legal and ethical frameworks surrounding web scraping for academic research purposes. The research highlighted the challenges researchers face when major platforms like Twitter restrict API access, forcing researchers to adopt web scraping methodologies while maintaining ethical and legal compliance. In [26], Lotfi et al. conducted *Web scraping techniques and applications: A Literature review*. The study provided a systematic analysis of modern web scraping approaches, emphasizing the integration of AI-based methods and tools for adaptive data extraction. The research demonstrated how machine learning technologies enhance web scraping capabilities for handling unstructured data formats and improving content classification accuracy.

## 9.3 BERT and Natural Language Processing for Academic Text Analysis

In [11], Koroteev presented *BERT: A Review of Applications in Natural Language Processing and Understanding*. The comprehensive review systematized applications of *BERT* across various text analytics tasks, providing detailed comparisons with similar models. The

research demonstrated *BERT*'s effectiveness in sentence similarity measurement using distance metrics between text embeddings, which is fundamental for automated relevance assessment in academic paper collection systems. In [24], Zhang et al. introduced *Semantic Similarity Calculating based on BERT*. The study explored semantic similarity as a fundamental aspect of natural language processing, demonstrating how *BERT*-based approaches significantly outperform traditional methods in semantic understanding tasks. The research showed remarkable performance improvements in similarity calculation tasks, particularly for contextual word embeddings that capture semantic relationships more effectively than static embedding methods. In [8], Devlin et al. introduced the groundbreaking *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. This seminal work established *BERT* as a revolutionary language model that significantly enhanced natural language processing capabilities through bidirectional context modeling. The research demonstrated *BERT*'s effectiveness across multiple downstream tasks including question answering, sentiment analysis, and text classification, making it a fundamental tool for academic text analysis applications. In [54], AlAmmary conducted *BERT models for Arabic text classification: a systematic review*. The study provided comprehensive analysis of *BERT* applications in domain-specific text classification tasks, demonstrating the model's adaptability across different languages and specialized academic domains. The research highlighted *BERT*'s superior performance in capturing semantic relationships and contextual understanding compared to traditional machine learning approaches. In [97], Li et al. presented *English text topic classification using BERT-based model*. The research demonstrated advanced applications of *BERT* for text topic classification in academic contexts, incorporating preprocessing techniques such as denoising, case normalization, and synonymous substitution for data enhancement. The study showed significant improvements in classification accuracy through *BERT*-based model optimization and custom classifier construction.

## 9.4 Academic Search Engines and Literature Management Tools

In [25], Narechania et al. introduced *VITALITY: Promoting serendipitous discovery of academic literature with transformers and visual analytics*. The research developed advanced literature discovery systems using transformer-based models combined with visual analytics approaches. The study demonstrated how modern NLP techniques can enhance traditional literature search capabilities, enabling more effective discovery of relevant academic papers through semantic understanding rather than simple keyword matching. In [98], Cohan et al. presented *SPECTER: Document-level representation learning using citation-informed transformers*. The research introduced sophisticated approaches for document-level representation learning that incorporates citation information to improve similarity assessment between academic papers. The study demonstrated significant improvements in document similarity calculation through transformer-based architectures that consider both textual content and citation relationships. In [36], Reimers et al. developed *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*. The research introduced *Siamese BERT* networks specifically designed for generating sentence embeddings that capture semantic similarity more effectively than traditional approaches. The study provided foundational techniques

for similarity assessment that are directly applicable to automated academic paper relevance evaluation systems.

## 9.5 Evaluation and Quality Assessment in Academic Systems

In [99], Aithal et al. conducted experiments on *SQuAD* datasets using *BERT* for addressing irrelevant and unanswerable questions in academic contexts. The research implemented similarity mechanisms that significantly reduced computational resource requirements while maintaining high accuracy in question answering tasks. The study provided important insights into optimizing *BERT*-based systems for academic applications with limited computational resources. In [100], Srivastava et al. demonstrated *cybersecurity entity recognition* using fine-tuned *BERT* models, achieving state-of-the-art results in specialized domain applications. The research showed how domain-specific fine-tuning of pre-trained language models can significantly improve performance in academic text analysis tasks, providing methodological guidance for developing specialized academic paper collection systems.

## 9.6 Similarity Network Analysis and Document Clustering

In [20], Chandrasekaran et al. presented comprehensive approaches to text document similarity analysis using embedding-based methods. The research addressed fundamental challenges in computational similarity assessment, demonstrating how different computational models impact similarity calculation results. The study provided critical insights into selecting appropriate similarity metrics for automated academic paper relevance assessment. In [101], Rezaeipourfarsangi et al. introduced *Interactive clustering and high-recall information retrieval using language models*. The research demonstrated advanced clustering techniques specifically designed for academic literature organization and retrieval. The study showed significant improvements in recall performance for academic information retrieval systems through interactive clustering approaches combined with modern language models.

# Chapter 10

## Conclusion and Future Work

In this thesis, I presented studies of the *automated reference paper collection system* using *web scraping* technologies. Firstly, I presented the *design of the web-based automatic reference paper collection system* to address challenges associated with manual reference extraction for researchers. This system accepts thesis titles and keywords that describe the original work for target research projects as input data. The system automatically generates comprehensive lists of downloadable PDF file links for publications relevant to the specified research criteria, significantly reducing the time researchers spend manually searching and collecting academic papers. The web-based architecture ensures accessibility across different platforms and institutions, enabling researchers to efficiently access the system from various locations and devices. Secondly, I presented the *implementation of the reference paper collection system* using advanced web scraping technologies within an interactive web services framework. This system can collect and monitor data from the Internet using *Selenium* as a sophisticated web scraping sensor, which examines similarity against search targets by comparing keywords using the *BERT model*. The *BERT model* is a deep learning architecture for *natural language processing (NLP)* that understands context by analyzing bidirectional relationships between words in sentences. The system employs *Python Flask* as the web application server while *Angular* provides the user interface for data presentation. The implementation utilizes *Docker containerization* for improved portability and scalability, while *Git version control* maintains comprehensive code management and collaborative development capabilities. Thirdly, I presented *multi-threading implementation and performance optimization* in the *reference paper collection system* to improve processing speed and system efficiency. The conventional single-threaded approach may lead to long response times when collecting large numbers of papers from multiple sources simultaneously. The *multi-threading implementation* enables concurrent paper searches across different academic databases, drastically reducing overall processing time while optimizing CPU utilization and memory management for improved system performance. The parallel processing capabilities allow the system to handle multiple user requests simultaneously, significantly improving scalability and user experience in multi-user environments. Fourthly, I presented the *PDF accessibility checking and content extraction mechanisms* as essential components for ensuring comprehensive paper collection capabilities. The goal of *PDF accessibility checking* is to rapidly verify document availability without full download, employing *Dynamic URL Redirection*, *Regex-based URL Parsing*, and *HTML Scraping* techniques. The system extracts semantic information using *Sentence Transformers models (SBERT)* and calculates

text similarity scores through *Cosine Similarity* to assess relevance between user queries and collected academic papers. The *fast PDF accessibility checking* mechanism prevents bandwidth waste and reduces processing delays by validating document availability before attempting full downloads, ensuring efficient resource utilization and improved system responsiveness. Fifthly, I presented the *improved version of the reference paper collection system using Selenium Stealth* to overcome anti-scraping mechanisms and improve system reliability when accessing academic databases. Against the limitations of traditional Selenium that leaves distinctive bot signatures and lacks human-like behaviors, researchers may experience IP blocking and access restrictions when repeatedly accessing *Google Scholar* and other academic platforms. If the system uses conventional Selenium without stealth capabilities, academic websites can easily detect automated activities, leading to service interruptions and making the system unusable. *Selenium Stealth* implementation mimics authentic human browsing behavior including natural scrolling patterns, realistic mouse clicking events, dynamic user agent string rotation, and varied request timing intervals. The improved system uses different browser configurations for each scraping session, employs sophisticated fingerprinting methods to avoid recognition as an automated process, and implements intelligent delay mechanisms between requests to simulate human reading patterns. By incorporating *Selenium Stealth technology*, the system successfully circumvents anti-bot detection systems, ensuring consistent accessibility and reliable operation for continuous research activities. Finally, I presented an *additional contribution regarding Selenium as a Unit Test Tool for WebPLAS*, demonstrating the versatility of Selenium beyond web scraping applications. This contribution shows that Selenium can be effectively used not only as a data collection tool but also as a comprehensive testing framework for web-based educational systems. The *Unit Test Tool* integrates *Python Flask*, *Selenium WebDriver*, *Behave Framework*, and *Allure Report* to create an automated testing environment for client-side web programming exercises. The system supports *HTML*, *CSS*, and *JavaScript* programming education by automatically testing student code submissions against predefined test scenarios written in *Gherkin* language using *Behavior Driven Development (BDD)* methodology. The implementation enables automated verification of student solutions for *Code Writing Problems (CWP)* in the *Programming Learning Assistant System (PLAS)*, significantly reducing manual grading time for instructors while providing immediate feedback to students. The tool generates comprehensive test reports using *Allure Framework*, allowing students to understand their code performance and identify areas for improvement. This educational application of Selenium demonstrates its flexibility as a browser automation tool that can be adapted for various purposes beyond traditional web scraping, contributing to the advancement of automated educational assessment technologies. I evaluated the effectiveness through comprehensive experiments comparing the proposed system with existing literature search tools and previous implementations. The evaluation results confirmed significant improvements with improved accessibility using Selenium Stealth, substantially reduced response times through multi-threading implementation, improved relevance assessment through *BERT*-based similarity analysis, and efficient resource utilization through fast PDF accessibility checking. The comparative analysis demonstrated superior performance in terms of paper collection coverage, processing speed, and system reliability compared to manual collection methods and existing automated tools. In future works, I will study integration of additional academic databases and repositories beyond *Google Scholar*, implement advanced relevance ranking algorithms using large language models such as *GPT* and other transformer-based architectures. I will add the source information which the pa-

per is obtained from. develop collaborative features for research team workflows including shared collections and annotation capabilities. Besides, I will add the search history features by saving previous all searches by the user. Furthermore, I will develop cross-platform compatibility improvements, real-time collaboration features, mobile-friendly UIs and integration with popular reference management tools to create a comprehensive academic research ecosystem.



# Bibliography

- [1] Khder, M.A. Web scraping or web crawling: State of art, techniques, approaches and application. *International Journal of Advances in Soft Computing & Its Applications* 2021, 13, 3.
- [2] Gundecha, U.; Avasarala, S. *Selenium WebDriver 3 practical guide: end-to-end automation testing for web and mobile browsers with Selenium WebDriver*, 2nd ed.; Packt Publishing, 2018; pp. 1–280.
- [3] Yuan, S. Design and visualization of Python web scraping based on third-party libraries and selenium tools. *Academic Journal of Computing & Information Science* 2023, 6, 25-31.
- [4] Han, S.; Anderson, C.K. Web scraping for hospitality research: overview, opportunities, and implications. *Cornell Hospital. Quart.* **2021**, 62(1), 89–104.
- [5] Shultz, M. Comparing test searches in PubMed and Google Scholar. *J. Medi. Lib. Assoc.* 2007, 95(4), 442–445.
- [6] M. Gusenbauer and N. R. Haddaway, "Which academic search systems are suitable for systematic reviews or meta-analyses? Evaluating retrieval qualities of Google Scholar, PubMed, and 26 other resources," *Research Synthesis Methods*, vol. 11, no. 2, pp. 181-217, 2020.
- [7] Naing, I.; Funabiki, N.; Wai, K.H.; Aung, S.T. A design of automatic reference paper collection system using Selenium and Bert Model. In *Proceedings of the IEEE 12th Global Conference on Consumer Electronics (GCCE)*, Nara, Japan, 10–13 October 2023; pp. 267-268.
- [8] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *Proc. NAACL-HLT*, pp. 4171-4186, 2019.
- [9] S. Davidson, A. Freelon, and D. Karpf, "Web Scraping for Research: Legal, Ethical, Institutional, and Scientific Considerations," *arXiv preprint arXiv:2410.23432*, 2023.
- [10] Naing, I.; Aung, S.T.; Wai, K.H.; Funabiki, N. A Reference Paper Collection System Using Web Scraping. *Electronics* 2024, 13, 2700.
- [11] M. Koroteev, "BERT: A Review of Applications in Natural Language Processing and Understanding," *arXiv preprint arXiv:2103.11943*, 2021.
- [12] Naing, I.; Funabiki, N.; Aung, S.T.; Fahrudin, T.M. An Preliminary Evaluation of Reference Paper Collection System Using Selenium Stealth for Web Scraping. *IEICE Technical Report* 2025, 124, KBSE2024-66, 83-86.

- [13] Naing, I.; Funabiki, N.; Wai, K.H.; Kyaw, H.H.S.; Qi, H.; Flasma, V. A Design and Implementation of Unit Test Tool for Client-Side Web Programming Learning Assistant System. IEICE Technical Report 2023, 122, NS2022-235, 390-395.
- [14] Docker. Available online: <https://docs.docker.com/> (accessed on 24 April 2024).
- [15] Djedouboum, A.C.; Abba Ari, A.A.; Gueroui, A.M.; Mohamadou, A.; Aliouat, Z. Big data collection in large-scale wireless sensor networks. 2018, *18(12)*, 4474.
- [16] Google Scholar. Available online: [https://ja.wikipedia.org/wiki/Google\\_Scholar](https://ja.wikipedia.org/wiki/Google_Scholar) (accessed on 24 April 2024).
- [17] Krauskopf, E. An analysis of discontinued journals by Scopus. *Scientometrics*. 2018, *116*, 1805–1815. <https://doi.org/10.1007/s11192-018-2808-5>
- [18] Wilde, M. IEEE Xplore digital library. *Charl. Adv.* **2016**, *17(4)*, 24–30. <https://doi.org/10.5260/chara.17.4.24>
- [19] T. M. Fahrudin, N. Funabiki, K. C. Brata, I. Naing, S. T. Aung, A. Muhaimin, and D. A. Prasetya, "An Improved Reference Paper Collection System Using Web Scraping with Three Enhancements," *Future Internet*, vol. 17, no. 5, pp. 195, 2025.
- [20] D. Chandrasekaran and V. Mago, "Evolution of semantic similarity—a survey," *ACM Computing Surveys*, vol. 54, no. 2, pp. 1-37, 2021.
- [21] Landers; Richard, N.; Brusso, R.C.; Cavanaugh, J.K.; Andrew Collmus, B. A primer on theory-driven web scraping: Automatic extraction of big data from the Internet for use in psychological research. *Psych. Meth.* 2016, *21(4)*, 475–492.
- [22] M. Graczyk and K. Redlarski, "Paperity - Aggregator of Open Access Scientific Literature," *Procedia Computer Science*, vol. 35, pp. 1138-1147, 2014.
- [23] P. Knoth and Z. Zdrahal, "CORE: Three Access Levels to Underpin Open Access," *D-Lib Magazine*, vol. 18, no. 11/12, 2012.
- [24] L. Zhang and Y. Wang, "Semantic Similarity Caculating based on BERT," *International Conference on Computer Science and Application Engineering*, pp. 1-6, 2024.
- [25] A. Narechania, A. Karduni, R. Wesslen, and E. Wall, "VITALITY: Promoting serendipitous discovery of academic literature with transformers and visual analytics," *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, no. 1, pp. 486-496, 2022.
- [26] Lotfi, C.; Srinivasan, S.; Ertz, M.; Latrous, I. Web scraping techniques and applications: a Literature review. In *Proceedings of the SCRS Conference on Intelligent Systems*, SCRS, India, April 2022; pp. 381-394. <https://doi.org/10.52458/978-93-91842-08-6-38>
- [27] Snell; James; Menaldo, N. Web scraping in an era of big data 2.0. In *Bloomberg Law News*, Jun. 2016.
- [28] Mitchell, R. *Web scraping with Python: Collecting more data from the modern web*, 2nd ed.; O'Reilly Media, Inc., 2018; pp. 1–308.
- [29] Glez-Peña, D.; Lourenço, A.; López-Fernández, Hugo.; Reboiro-Jato, M.; Fdez-Riverola, F. Web scraping technologies in an API world. *Brief. Bioinfo.* 2014, *15(5)*, 788–797. <https://doi.org/10.1093/bib/bbt026>

- [30] Viotti, J.C.; Kinderkhedra, M. *A survey of JSON-compatible binary serialization specifications*, eprint arXiv:2201.02089; Computer Science- Databases; Computer Science - Software Engineering, 2022; pp. 1–101. <https://arxiv.org/abs/2201.02089>
- [31] Selenium WebDriver. Available online: <https://www.selenium.dev/documentation/webdriver/> (accessed on 24 April 2024).
- [32] Church, K.W. Word2Vec. *Natur. Lang. Engin.* 2017, 23(1), 155–162. <https://doi.org/10.1017/S1351324916000334>
- [33] Pennington, J.; Socher, R.; D-Manning, C. GloVe: Global Vectors for Word Representation. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar, 25–29 October 2014; pp. 1532–1543.
- [34] Yao, T.; Zhai, Z.; Gao, B. Text Classification Model Based on fastText. In Proceedings of the 2020 IEEE International Conference on Artificial Intelligence and Information Systems (ICAIS), Dalian, China, 20–22 March 2020; pp. 154–157. doi:10.1109/ICAIS49377.2020.9194939
- [35] Sun, X.; Meng, Y.; Ao, X.; Wu, F.; Zhang, T.; Li, J.; Fan, C. Sentence similarity based on contexts. *Trans. Assoc. Comput. Ling.* 2022, 10, 573–588. [https://doi.org/10.1162/tac1\\_a\\_00477](https://doi.org/10.1162/tac1_a_00477)
- [36] Reimers, N.; Gurevych, I. Sentence-BERT: sentence embeddings using Siamese BERT-networks. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, Hong Kong, China, 3–7 November 2019; pp. 3982–3992. <https://aclanthology.org/D19-1410.pdf>
- [37] Wikipedia. Available online: <https://www.wikipedia.org/> (accessed on 24 April 2024).
- [38] Vyshnavi, V.R.; Malik, A. Efficient way of web development using Python and Flask. *Int. J. Recent Res. Asp.* 2019, 6(2), 16–19.
- [39] Cincovic, J.; Delcev, S.; Draskovic, D. Architecture of web applications based on Angular framework: a case study. In Proceedings of the ICIST, Jul 2019; pp. 254–259. <https://api.semanticscholar.org/CorpusID:222459277>
- [40] Flask. Available online: <https://flask.palletsprojects.com/en/3.0.x/.v> (accessed on 24 April 2024).
- [41] Hugging Face. Available online: <https://huggingface.co/> (accessed on 24 April 2024).
- [42] Angular. Available online: <https://angular.io/> (accessed on 24 April 2024).
- [43] TypeScript. Available online: <https://www.typescriptlang.org/> (accessed on 24 April 2024).
- [44] Material UI. Available online: <https://material.angular.io/> (accessed on 24 April 2024).
- [45] Node.js. Available online: <https://nodejs.org/en> (accessed on 24 April 2024).
- [46] NPM. Available online: <https://docs.npmjs.com/> (accessed on 24 April 2024).
- [47] Dahl, D. The W3C multimodal architecture and interfaces standard. *J. Multi. User Inter.* 2013, 7(3), 171–182.

- [48] XPath. Available online: <https://developer.mozilla.org/ja/docs/Web/XPath> (accessed on 24 April 2024).
- [49] Mustofa, K.; Fajar, S.P. Selenium-based multithreading functional testing. *Indo. Jou. Comp. Cyber. Sys. (IJCCS)*. 2018, (12)1, 63–72.
- [50] Zefeng, Q.; Umapathy, P.; Zhang, Q.; Song, G.; Zhu, T. Map-reduce for multiprocessing large data and multi-threading for data scraping, *Mathematics - Numerical Analysis*, arXiv preprint arXiv:2312.15158, Dec. 2023.
- [51] PyPDF2. Available online: <https://pypi.org/project/PyPDF2/> (accessed on 24 April 2024).
- [52] PyMuPDF. Available online: <https://pypi.org/project/PyMuPDF/> (accessed on 24 April 2024).
- [53] all-MiniLM-L12-v2 - sentence transformer model. Available online: <https://huggingface.co/sentence-transformers/all-MiniLM-L12-v2> (accessed on 24 April 2024).
- [54] A. S. Alammary, "BERT models for Arabic text classification: a systematic review," *Applied Sciences*, vol. 12, no. 11, pp. 5720, 2022.
- [55] A. S. Tanenbaum, "Modern Operating Systems," 4th ed., Pearson, 2014.
- [56] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux Journal*, vol. 2014, no. 239, 2014.
- [57] D. R. Butenhof, "Programming with POSIX Threads," Addison-Wesley Professional, 1997.
- [58] A. Williams, "C++ Concurrency in Action: Practical Multithreading," Manning Publications, 2012.
- [59] D. Lea, "Concurrent Programming in Java: Design Principles and Patterns," 2nd ed., Addison-Wesley, 1999.
- [60] I. Foster, "Designing and Building Parallel Programs," Addison-Wesley, 1995.
- [61] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81-84, 2014.
- [62] S. Newman, "Building Microservices: Designing Fine-Grained Systems," O'Reilly Media, 2015.
- [63] T. Combe et al., "To Docker or Not to Docker: A Security Perspective," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54-62, 2016.
- [64] K. Matthias and S. Kane, "Docker: Up & Running," O'Reilly Media, 2015.
- [65] B. Burns and J. Beda, "Kubernetes: Up and Running," 2nd ed., O'Reilly Media, 2019.
- [66] G. G. Chowdhury, "Introduction to Modern Information Retrieval," 3rd ed., Facet Publishing, 2004.
- [67] N. Meuschke et al., "Academic literature analysis: Detection and evaluation of related work," *Computer Science Review*, vol. 32, pp. 1-17, 2019.
- [68] Y. Li et al., "PDF content extraction for academic documents: A survey," *Information Processing Management*, vol. 57, no. 4, pp. 102-118, 2020.

- [69] L. L. Constantine, "The Peopleware Papers: Notes on the Human Side of Software," Prentice Hall PTR, 2016.
- [70] D. Tkaczyk et al., "CERMINE: automatic extraction of structured metadata from scientific literature," *International Journal on Document Analysis and Recognition*, vol. 18, no. 4, pp. 317-335, 2015.
- [71] J. Beel et al., "Research-paper recommender systems: a literature survey," *International Journal on Digital Libraries*, vol. 17, no. 4, pp. 305-338, 2016.
- [72] F. Cristian, "Understanding fault-tolerant distributed systems," *Communications of the ACM*, vol. 34, no. 2, pp. 56-78, 1991.
- [73] G. R. Andrews, "Foundations of Multithreaded, Parallel, and Distributed Programming," Addison-Wesley, 1999.
- [74] Performance Monitor. Available online: [https://en.wikipedia.org/wiki/Performance\\_Monitor](https://en.wikipedia.org/wiki/Performance_Monitor) (accessed on 24 April 2024).
- [75] Kinney. R; Anastasiades. C; Authur. R; Beltagy. I; Bragg. J; Buraczynski. A; Cachola. I; Candra. S; Chandrasekhar. Y; Cohan. A; Miles Crawford. M; Downey. D; Dunkelberger. J; Etzioni. O; Evans. R; Feldman. S; Gorney. J; Graham. D; Hu. F; Huff. R; King. D; Kohlmeier. S; Kuehl. B; Langan. M; Lin. D; Liu. H; Lo. K; Lochner. J; MacMillan. K; Murray. T; Newell. C; Rao. S; Rohatgi. S; Sayre. P; Shen Z; Singh. A; Soldaini. L; Subramanian. S; Tanaka A; Wade. A; Wagner. L; Wang. L; Wilhelm. C; Wu. C; Yang. J; Zamarron. A; Van Zuylen. M; Weld. D. The Semantic Scholar Open Data Platform. arXiv:2301.10140, Jan. 2023. <https://arxiv.org/pdf/2301.10140>
- [76] Gojare, S.; Joshi, R.; Gaigaware, D. Analysis and design of selenium WebDriver automation testing framework. *Procedia Comput. Sci.* 2015, 50, 341–346. <https://doi.org/10.1016/j.procs.2015.04.038>
- [77] Measuring and interpreting system usability scale - SUS. Available online: <https://uiuxtrend.com/measuring-system-usability-scale-sus/> (accessed on 24 April 2024).
- [78] System Usability Scale - SUS. Available online: <https://credoagency.co.uk/usability-in-cro-the-system-usability-scale-sus/> (accessed on 24 April 2024).
- [79] E. Ferrara, "The rise of ChatGPT: exploring its potential in cybersecurity applications," arXiv preprint arXiv:2306.09462, 2023.
- [80] Wendt, H.; Henriksson, M. Building a Selenium-based data collection tool. Bachelor's Thesis, 16 ECTS, Information Technology, Linköping University, May 2020.
- [81] N. Funabiki, H. Masaoka, N. Ishihara, I. W. Lai, and W. C. Kao, "Offline answering function for fill-in-blank problems in Java programming learning assistant system," in *Proc. IEEE ICCE-TW*, pp. 324-325, May 2016.
- [82] N. Funabiki, Y. Matsushima, T. Nakanishi, K. Watanabe, and N. Amano, "A Java programming learning assistant system using test-driven development method," *IAENG Int. J. Comp. Sci.*, vol. 40, no.1, pp. 38-46, Feb. 2013.
- [83] Selenium Documentation. [Online]. <https://www.selenium.dev/> (accessed on 24 April 2024).

- [84] Srashti Lariyal Dr. Sameer Shrivastava Er. Sumit Nema, "Automation Testing using Selenium Web Driver & Behavior Driven Development (BDD)," IJSRD Issue 03, 2018 — ISSN.
- [85] Behave Documentation. [Online]. <https://behave.readthedocs.io/en/stable/index.html>(accessed on 24 April 2024).
- [86] Allure Framework Documentation [Online] <https://docs.qameta.io/allure/>(accessed on 24 April 2024).
- [87] Jinja2 Documentation [Online] <https://jinja.palletsprojects.com/en/3.1.x/>.
- [88] AJAX Full Form Definition.[Online]. <https://www.javatpoint.com/ajax-full-form>.
- [89] T. Tanaka, H. Niibori, L. Shiyngxue, S. Nomura, T. Nakao and K. Tsuda, "Selenium based Testing Systems for Analytical Data Generation of Website User Behavior," 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, 2020, pp.216-221, doi:10.1109/ICSTW50294.2020.00045.
- [90] J. Borgenstierna, "Behave and PyUnit – A Testers Perspective," Bachelor thesis, 16 ECTS — Innovative Programming Spring , 2018.
- [91] J. Wu, P. Mitra, S. Teregowda, and C. L. Giles, "CiteSeerX: AI in a Digital Library Search Engine," AI Magazine, vol. 36, no. 3, pp. 35-48, 2015.
- [92] J. Priem, H. Piwowar, and R. Orr, "OpenAlex: A fully-open index of scholarly works, authors, venues, institutions, and concepts," arXiv preprint arXiv:2205.01833, 2022.
- [93] P. Bernardes, E. Moro, L. A. Lopes, and R. Vigário, "PaperBot: A Distributed Information Extraction System for Academic Literature," Proc. ACM Symposium on Document Engineering, pp. 147-150, 2019.
- [94] C. Manghi, P. Mikulicic, M. Jörg, N. Houssos, A. Artini, and L. Atzori, "The OpenAIRE Research Graph Data Model," Zenodo, 2019.
- [95] A. A. Fareedi, S. Gagnon, A. Ghazawneh, and R. Valverde, "Semantic Fusion of Health Data: Implementing a Federated Virtualized Knowledge Graph Framework Leveraging Ontop System," Future Internet, vol. 17, no. 6, pp. 245, 2025.
- [96] A. Sharma, R. Joshi, and D. Gaigaware, "An Improving Approach for Fast Web Scraping Using Machine Learning and Selenium Automation," International Journal of Scientific Research in Computer Science, Engineering and Information Technology, vol. 9, no. 2, pp. 341-346, 2023.
- [97] X. Li and L. Jia, "English text topic classification using BERT-based model," Journal of Computational Methods in Sciences and Engineering, vol. 24, no. 6, pp. 3247-3258, 2025.
- [98] A. Cohan, S. Feldman, I. Beltagy, D. Downey, and D. Weld, "SPECTER: Document-level representation learning using citation-informed transformers," Proc. Annual Meeting of the Association for Computational Linguistics, pp. 2270-2282, 2020.
- [99] A. Aithal, P. Aithal, and P. S. Aithal, "BERT Based Question Answering for COVID-19 Scientific Literature," International Journal of Applied Engineering and Management Letters, vol. 5, no. 2, pp. 140-156, 2021.

- [100] S. K. Srivastava, S. Kumar, and V. Kumar, "Fine-tuning BERT for cybersecurity entity recognition," *Journal of Information Security and Applications*, vol. 75, pp. 103487, 2023.
- [101] S. Rezaeipourfarsangi, N. Pei, E. Sherkat, and A. Bagheri, "Interactive clustering and high-recall information retrieval using language models," *Information Processing & Management*, vol. 59, no. 4, pp. 102968, 2022.