# 

# 

# 

Multi-Modal Similarity Detection System

Name: HENG ZHENG TECK

Programme: Bachelor in Data Science

Student ID : 24WMR11422

Supervisor: NOOR AIDA BINTI HUSAINI

# 

# **Table Content**

[**Chapter 1 Introduction**](#_aujff474xkfp) **4**

[1.1 Introduction](#_rsoohwp5e94z) 4

[1.2 Background of Study](#_afhj5ag6nz6u) 5

[1.3 Problem Statement](#_q5hymno15bbu) 6

[1.4 Gaps in Current Solutions](#_6kvw7x3ivcls) 7

[1.5 Objectives](#_sg1885gs587h) 8

[General Objective](#_823p39dkmupt) 8

[Specific Objectives](#_h1noy033qi7m) 8

[1.6 Research Questions](#_3qkwrv5mnaek) 8

[1.7 Scope of Study](#_qmeb2ypa6xnb) 8

[1.8 Significance of Study](#_jdhp6zz4za8n) 9

[1.9 Project Scheduling](#_v6prtc1m0uao) 9

[2.0 Remarks](#_v6tkf0mdtxiw) 11

[**Chapter 2 literature review**](#_5kkl5qgp9sc) **11**

[2.1 Introduction](#_bb4pwusvxycs) 11

[2.2 Document and Text Similarity](#_gilpmbufqr5j) 11

[2.2.1 Traditional Text Similarity Approaches](#_siktw1a5242) 11

[2.2.2 Semantic and Transformer-Based Approaches](#_8upy2o3doclp) 12

[2.2.3 URL-Based and Web Content Similarity Detection](#_wsx5vftox5ol) 12

[2.2.4 Real-World Tools and Applications](#_jygkx1kzt0dj) 13

[2.2.5 Thematic Summary](#_qsmpn8vlg8al) 13

[2.3 Image Similarity](#_8mxybpolxnt6) 13

[2.3.1 Traditional Image Similarity Methods](#_u3299qgryovk) 13

[2.3.2 Deep Learning-Based Approaches](#_7ez3c4z5n0mp) 14

[2.3.3 Real-World Applications](#_9n6im42h6cx4) 14

[2.4 Code Similarity](#_zfqs7m4hq982) 14

[2.4.1 Traditional Code Similarity Methods](#_ud1xyscjsbu0) 15

[2.4.2 Structural and Hybrid Approaches](#_osub5vkgu78y) 15

[2.4.3 Comparative Tools](#_l2yh6vm6qqc1) 15

[2.5 Evaluation Metrics](#_w523vba81pyn) 15

[2.6 Limitations and Challenges](#_y1mo5juzxd6r) 16

[2.7 Recent Developments (2022–2025)](#_iazbugxxjdj5) 16

[2.8 Summary of Literature](#_v6azg796zi61) 16

[**Chapter 3: Methodology and Requirement Analysis**](#_yc2f0q523fpi) **16**

[3.1 Introduction](#_y6enkgoc81x8) 16

[3.2 Research Paradigm and Development Methodology](#_ve8yu4vulcpu) 17

[3.3 System Development Lifecycle](#_e1jz3abx0hye) 21

[3.4 Requirement Gathering and Analysis](#_oda246lb20a) 21

[3.5 Functional Requirements](#_3e71b2cxkj0z) 21

[3.6 Non-Functional Requirements](#_vadbw066ux0n) 23

[3.7 Tools and Technology Selection](#_wjonx2ol7dty) 24

[3.7 Gantt Chart](#_6a6aab9hmkvi) 25

[3.8 Summary](#_on7ihkjiqusb) 25

[**Chapter 4: System Design**](#_djxbfcmyng6g) **26**

[4.1 Introduction](#_qsitysjlha83) 26

[4.2 System Architecture](#_lz7wl5s4ged) 27

[4.3 Component-Level Design](#_w3zyy9xk6nfu) 28

[4.4 System Workflow and Data Flow](#_rqqc58tmarp) 29

[4.5 Algorithm and Model Justification](#_w273mcrbhf4z) 29

[4.6 Database Schema Design](#_umd97zaliek) 29

[**Chapter 5 Development and Testing**](#_eiwnwysy0xd) **34**

[5.1 Introduction](#_4nhavi1wksui) 34

[5.2 Development Environment and Tools](#_42m3kuv2b7pi) 34

[5.3 Backend Development](#_l5gfyat2rrq8) 35

[5.3.1 Flask extension module](#_a0uk96xc1e36) 35

[5.3.1 Authentication Module](#_cc74do8b5675) 36

[5.3.2 Database Engine](#_sxnyxn4lvj2y) 43

[5.3.3 Code Similarity Engine](#_mf1ydq9akhh) 47

[5.3.4 Image Similarity Engine](#_tglqpyvm3sq5) 51

[5.3.5 Document Similarity](#_vbr92g4xtbhp) 53

[5.3.6 System Integration and Application Controller Module](#_on73ve35ippq) 55

[5.4 Frontend Development](#_3fefq813g8yj) 67

[5.4.1 User Interface Components](#_jsg6e9jww78e) 67

[5.4.1.1 header](#_7b90lf3485dy) 68

[5.4.1.2 history](#_nbtyw4vdf49f) 70

[5.4.1.3 index](#_31y96bnvei7e) 73

[5.3.1.4 profile](#_v8zl0xocs90z) 77

[5.3.1.5 report](#_l3nldin9zb35) 78

[5.3.1.6 reset password](#_gg9vjbickxx) 79

[5.3.1.7 view text](#_azua487n0ptz) 80

[5.4.2 CSS](#_gs8fa6afwxdq) 81

[5.4.2.1 home](#_8v7led2lr6jm) 81

[5.4.2.2 report](#_vakjvc15y6a2) 91

[5.4.2.3 reset password](#_ueusjohat1oy) 92

[5.4.3 JavaScript Logic](#_24spqg4ckrgq) 94

[5.4.3.1 home.js](#_vcwdvas97cum) 94

[5.4.3.2 history](#_ay9p29z52j7l) 109

[5.4.3 Result Visualization and Difference Highlighting](#_vdixounq0gnc) 113

[Signup](#_61zqidd4lwdw) 113

[Login](#_8lzi5z2ck7ui) 114

[Forgot password](#_28l8wdbtb4yc) 115

[Profile](#_2lczm1k0darp) 116

[Document comparison](#_h934viqtync0) 116

[Image comparison](#_cfy9y18d6yq7) 117

[Code comparison](#_veud4ygttwce) 118

[Text comparison](#_mwf9es7x1r3) 119

[URL comparison](#_gfc5b2lvzedu) 120

[History](#_i03kksx6yr23) 121

[Filter and search](#_63fjxjp1rdfg) 122

[View and Download report](#_bbpzty79oxfy) 124

[5.5 System Integration](#_669nyz505r67) 125

[5.6 Error Handling and Validation](#_uyyt4qwhycti) 126

[5.7 System Testing](#_xr3lkb558ric) 126

[5.7.1 Testing Strategy](#_8pfvqejzu2fa) 127

[5.7.2 Testing Results and Discussion](#_j04hbb3ha6h) 128

[5.8 Security Considerations](#_oivj9rdbizd1) 128

[5.9 Summary](#_79f2lduc9u16) 129

[**Chapter 6 Summary and Conclusions**](#_ilt3v6r5aocc) **130**

[6.1 Introduction](#_toyc0pqxwpgx) 130

[6.2 Summary of Chapter 1: Introduction and Problem Definition](#_dv80na7phi3b) 130

[6.3 Summary of Chapter 2: Literature Review](#_iylcrxcvabvq) 130

[6.4 Summary of Chapter 3: Methodology and Requirement Analysis](#_tjpe7lubgl5n) 131

[6.5 Summary of Chapter 4: System Design](#_xnfbgt9j3sps) 132

[6.6 Summary of Chapter 5: Implementation and Evaluation](#_ayhxop3jlnj0) 132

[6.7 Overall Contributions and Achievements](#_8fplqys944ot) 133

[6.8 Conclusion](#_ff8mtz34vq3j) 133

[**References**](#_bqu7kmdzrb57) **134**

# 

# **Chapter 1 Introduction**

## **1.1 Introduction**

With the rapid growth of digital content across academic, software, and creative fields, the issue of plagiarism and content duplication has become increasingly significant (Clough, 2010). Educational institutions rely on originality to uphold academic integrity, software organizations must protect intellectual property, and creative industries depend on copyright enforcement to encourage innovation and fair use (ICAI, 2023; WIPO, 2021). As a result, automated similarity detection systems have become essential tools in modern digital environments (Bex et al., 2020).

While existing tools such as Turnitin for textual documents, MOSS for source code, and Google Reverse Image Search for images have contributed significantly to plagiarism detection, these tools operate independently and are limited to single content modalities (Schleimer et al., 2003; Clough, 2010). Consequently, they are unable to effectively evaluate submissions that contain a combination of text, images, and source code, which is increasingly common in modern academic and professional workflows (Alzahrani et al., 2012).

This separation introduces several limitations, including fragmented workflows, inconsistent similarity measurements, and reduced detection accuracy when dealing with mixed-media submissions (Bennett et al., 2018). Users are often required to apply multiple tools to analyze a single submission, increasing time consumption and the likelihood of oversight (Foltynek et al., 2019). Therefore, there is a growing need for a unified platform capable of performing similarity detection across multiple content modalities within a single system (Potthast et al., 2016).

This project proposes a Multi-Modal Similarity Detection System that integrates Natural Language Processing (NLP), image feature analysis, source code similarity detection, and URL-based comparison into a single web-based platform. For textual similarity, the system employs transformer-based sentence embedding models using Sentence-BERT (*all-MiniLM-L6-v2*), which have demonstrated strong performance in capturing semantic similarity beyond lexical overlap (Reimers & Gurevych, 2019). Image similarity detection is achieved through deep convolutional neural networks using feature extraction from a pre-trained ResNet50 model, enabling robust comparison even under transformations such as resizing or color variation (He et al., 2016). Source code similarity is analyzed using a combination of token-based hashing and Abstract Syntax Tree (AST) structural analysis, which has been shown to be effective against superficial code modifications (Baxter et al., 1998; Schleimer et al., 2003).

In addition, the system supports direct text input, URL-based comparison, side-by-side difference visualization, similarity scoring, historical tracking, and automated report generation, enhancing transparency and usability. By leveraging pre-trained machine learning models and open-source technologies, the proposed system is scalable, efficient, and adaptable, making it suitable for academic institutions, software developers, and creative professionals (Devlin et al., 2019; Reimers & Gurevych, 2019).

## **1.2 Background of Study**

Traditional plagiarism detection tools have played an essential role in safeguarding originality within their respective domains. Turnitin is widely adopted in academic institutions for detecting textual similarity by comparing submissions against extensive proprietary databases (Clough, 2010). MOSS (Measure of Software Similarity) is commonly used in programming education to detect similarities in source code by analyzing token sequences and structural patterns (Schleimer et al., 2003). Google Reverse Image Search provides basic visual similarity detection by matching images against indexed web content but offers limited robustness against image transformations (WIPO, 2021).

Despite their effectiveness, these tools are fundamentally siloed and limited to single content types (Foltynek et al., 2019). Modern educational and professional submissions increasingly combine written explanations, programming code, diagrams, screenshots, and web-based references (Potthast et al., 2016). The lack of an integrated similarity detection solution forces users to manually correlate results from multiple systems, which is inefficient and error-prone (Bennett et al., 2018).

Recent advancements in artificial intelligence provide an opportunity to overcome these limitations. Transformer-based NLP models such as BERT enable deep semantic text comparison, allowing detection of paraphrased or restructured content rather than relying solely on surface-level similarity (Devlin et al., 2019). In computer vision, convolutional neural networks allow robust extraction of visual features that remain consistent despite common image modifications (He et al., 2016). For source code analysis, AST-based techniques provide structure-aware comparison that is resilient to formatting changes, variable renaming, and minor logic reordering (Baxter et al., 1998).

Statistical evidence highlights the urgency of this research. The International Center for Academic Integrity (ICAI, 2023) reported that approximately 62% of undergraduate students admitted to engaging in some form of plagiarism. A GitHub survey (2022) revealed that nearly 40% of software plagiarism cases involved modified or reformatted code designed to evade detection. Furthermore, a World Intellectual Property Organization (WIPO, 2021) report identified a significant increase in copyright disputes related to digital content reuse. These findings emphasize the necessity of a unified, explainable, and multi-modal similarity detection system.

## **1.3 Problem Statement**

Although existing plagiarism detection tools perform well within individual domains, their single-modality focus limits their effectiveness in modern, mixed-content environments (Foltynek et al., 2019). Educators, developers, and content creators increasingly handle submissions that combine documents, images, source code, and online content, which current tools are not designed to evaluate holistically (Potthast et al., 2016).

For example, a lecturer grading a student project that includes written explanations, diagrams, and programming code must rely on multiple independent tools to assess originality, increasing workload and inconsistency (Bennett et al., 2018). Developers may bypass source code plagiarism detection by renaming variables or reformatting syntax, while designers may reuse images with minor visual modifications that escape traditional image matching techniques (Baxter et al., 1998; He et al., 2016).

Without a unified solution, institutions and professionals face challenges in ensuring fairness, maintaining originality, and enforcing intellectual property protection (ICAI, 2023). This project addresses these challenges by developing a single platform that enables multi-modal similarity detection, supports file uploads, text input, and URL comparison, and provides transparent similarity scoring with visual difference reporting.

## **1.4 Gaps in Current Solutions**

| **Tools** | **Modality** | **Strengths** | **Weaknesses** |
| --- | --- | --- | --- |
| Turnitin | Text | Large academic database; widely adopted in higher education institutions | Does not support image or source code similarity detection; expensive and proprietary (Clough, 2010) |
| MOSS | Source code | Effective token-based comparison for programming assignments; commonly used in academia | Limited to source code only; weak against structural variation and cross-language plagiarism (Schleimer et al., 2003) |
| Google Reverse  Image Search | Images | Fast and accessible image matching across the web | Performs poorly on modified, cropped, or transformed images; lacks detailed similarity scoring (WIPO, 2021) |
| Grammarly/iThentica te | Text | Combines grammar checking with similarity detection | Restricted to textual analysis; closed-source and not extensible (Foltynek et al., 2019) |
| Proposed System | Text, Image, Source Code, URL | Unified multi-modal similarity detection; semantic text analysis; image feature | Requires extensive testing and evaluation as a newly developed system |

## **1.5 Objectives**

### **General Objective**

To develop and evaluate a multi-modal similarity detection system that integrates NLP, image analysis, source code comparison, and URL-based analysis into a unified web-based platform.

### **Specific Objectives**

* To implement semantic text similarity detection using transformer-based sentence embedding models (Reimers & Gurevych, 2019).
* To develop image similarity detection using deep learning-based feature extraction techniques (He et al., 2016).
* To analyze source code similarity using token-based and AST-based methods (Baxter et al., 1998).
* To integrate all similarity modules into a single platform supporting file uploads, direct text input, and URL comparison.
* To conduct comprehensive system testing using real-world document , url , image and text input to evaluate accuracy, usability, and performance (Potthast et al., 2016).

## **1.6 Research Questions**

1. How can text, image, source code, and URL-based similarity detection techniques be effectively integrated into a unified system (Potthast et al., 2016)?
2. To what extent can a multi-modal similarity detection platform improve efficiency and usability compared to single-modality tools (Foltynek et al., 2019)?
3. What similarity metrics are most suitable for evaluating semantic text similarity, visual similarity, and source code similarity (Reimers & Gurevych, 2019; He et al., 2016)?
4. How can similarity results be presented in a transparent and user-friendly manner (Bennett et al., 2018)?
5. What are the limitations of the current system and potential extensions for future research (Devlin et al., 2019)?

## **1.7 Scope of Study**

This project focuses on similarity detection for text documents in PDF, DOCX, and TXT formats, limited to the English language (Clough, 2010). Image similarity detection supports JPEG and PNG formats, which are widely used in academic and creative contexts (WIPO, 2021). Source code similarity analysis covers Python, Java, and C++ programming languages using structural and token-based techniques (Schleimer et al., 2003). The system also supports direct text input and URL-based comparison to analyze online content. Other modalities such as audio and video are outside the scope of this study but are considered potential future extensions (Potthast et al., 2016).

## **1.8 Significance of Study**

The significance of this study lies in its academic, technical, and practical contributions. For academic institutions, the system supports fair assessment and strengthens academic integrity (ICAI, 2023). For software developers, it assists in detecting reused or modified source code across projects (Baxter et al., 1998). For creative professionals, it helps protect visual content from unauthorized reuse (WIPO, 2021). From a research perspective, the project contributes a unified and extensible framework for multi-modal similarity detection, supporting future advancements in explainable and cross-modal plagiarism detection (Potthast et al., 2016).

## **1.9 Project Scheduling**

| **ACTIVITIES** | **EXPECTED OUTCOME** | **COMPLETION** |
| --- | --- | --- |

|  |  | **DATE** |
| --- | --- | --- |
| proposal writing | Project Proposal Approved | 4/15/2025 |
| Chapter 1  Introduction | Project Introduction Completed | 3/7 /2025 |
| Chapter 2 Research Background | Research Context and Motivation Drafted | 14 / 7 /2025 |
| Chapter 3  Methodology and  Requirements  Analysis | System Methodology and  Requirement Specification | 28 / 7 /2025 |
| Chapter 4 System Design | Architecture and Design Diagrams Finalized | 18 / 8 /2025 |
| Project I Portfolio  (Individual) | Individual Report Completed | 8 / 9 /2025 |
| Document Similarity Module | Functional text similarity checker | 11/9/2025 |
| Image Similarity  Module | Working image comparison module | 16/9/2025 |
| Code Similarity  Module | Code analysis and comparison  module ready | 21/9/2025 |
| Initial System Preview with Supervisor. | The system can work without  any error | 26/9/2025 |
| Improvement | Allow user to input paragraph in textarea or url in textfield and can download report with analysis | 5/10/2025 |
| Integration of  Modules | Combined Multi-modal Similarity Checker | 26/10/2025 |
| User Testing and  Evaluation | Usability and Accuracy Feedback | 1/11/2025 |
| Bug Fixing and  Optimization | Improved Performance and Accuracy | 10/11/2025 |
| Preparation of test  plan/cases or experiment plan System Preview with Supervisor | Test Plan Ready and Initial System Preview | 21/11/2025 |
| Final System  Testing with  Supervisor and  Moderator | Fully Tested and Finalized System | 12/12/2025 |

## **2.0 Remarks**

This project presents an integrated and practical approach to similarity detection by combining text, image, source code, and URL-based analysis into a single system. By addressing the limitations of existing siloed tools, the system provides transparent similarity scoring, visual difference reporting, historical tracking, and automated report generation. Leveraging modern machine learning models and open-source technologies, the proposed solution demonstrates strong academic relevance and real-world applicability. Future work may extend the system to additional modalities, enhance cross-language code detection, and integrate with learning management systems or online repositories (Devlin et al., 2019).

# Chapter 2 literature review

## **2.1 Introduction**

Similarity detection is a multidisciplinary research field that integrates natural language processing (NLP), computer vision, and software engineering to quantify the degree of resemblance between digital entities such as documents, images, source code, and online textual resources. With the rapid expansion of web-based content and digital submissions, similarity detection systems are increasingly required to handle heterogeneous inputs, including uploaded files, raw text, and content retrieved from URLs (Manning et al., 2008; Foltynek et al., 2019). Accurate similarity detection is essential in academic integrity enforcement, copyright protection, software quality assurance, and online content moderation.

Traditional similarity detection techniques relied on surface-level matching such as lexical overlap, hashing, and syntactic comparison. Although effective for identifying exact duplication, these approaches failed to capture semantic similarity, paraphrasing, and structural variations commonly found in modern digital content. Recent advancements in deep learning, particularly transformer-based language models and convolutional neural networks, have significantly improved semantic understanding and robustness across modalities (Devlin et al., 2019; He et al., 2016). This chapter reviews existing literature on document, image, source code, and URL-based similarity detection, evaluates commonly used metrics and tools, and highlights research gaps addressed by the proposed multi-modal similarity detection system.

## **2.2 Document and Text Similarity**

### **2.2.1 Traditional Text Similarity Approaches**

Early text similarity detection methods employed statistical and lexical representations such as Bag-of-Words (BoW), TF-IDF, and n-gram models. These approaches represented text based on term frequency and word sequences, enabling detection of direct copying but lacking semantic awareness. Consequently, paraphrased content or synonym substitution often resulted in low similarity scores despite conceptual equivalence (Salton & Buckley, 1988; Manning et al., 2008).

String-based comparison methods were also applied to web content and URL-based plagiarism detection, where exact phrase matching was used to identify duplicated online text. However, such methods were highly sensitive to minor edits and rewording, limiting their effectiveness in detecting paraphrased or summarized web content (Clough, 2010).

### **2.2.2 Semantic and Transformer-Based Approaches**

Semantic similarity detection advanced significantly with the introduction of distributed word embeddings such as Word2Vec and GloVe, which enabled vector-based representation of words based on contextual usage (Mikolov et al., 2013). However, these models assigned fixed representations to words, ignoring contextual meaning. Transformer-based models such as BERT addressed this limitation by generating contextualized embeddings, enabling deeper semantic understanding of sentences and paragraphs (Devlin et al., 2019).

Sentence-BERT (SBERT) further optimized BERT for similarity detection by producing fixed-length sentence embeddings suitable for efficient cosine similarity computation (Reimers & Gurevych, 2019). These models are particularly effective for comparing uploaded documents, raw text input, and text extracted from URLs. The proposed system adopts SBERT to perform semantic similarity analysis across all text-based inputs, ensuring consistent and accurate detection regardless of content source.

### **2.2.3 URL-Based and Web Content Similarity Detection**

With the increasing reliance on online sources, URL-based similarity detection has become an essential component of plagiarism analysis. Web plagiarism detection involves extracting textual content from URLs and comparing it with submitted documents or text inputs. Early approaches relied on search-engine-based matching and fingerprinting, which were limited by accessibility and scalability constraints (Potthast et al., 2010).

Recent research has demonstrated that semantic embeddings can significantly improve web content similarity detection by identifying conceptual overlap between local documents and online sources, even when the content is paraphrased or summarized (Foltynek et al., 2019). The proposed system integrates URL text extraction and semantic embedding generation, allowing direct comparison between uploaded files, raw text input, and online web pages within a unified framework.

### **2.2.4 Real-World Tools and Applications**

Document and web text similarity detection is widely used in academic and professional tools. Turnitin compares student submissions against academic databases and online sources using a combination of lexical and semantic techniques. Grammarly and iThenticate apply semantic analysis for originality checking and writing assistance, while Copyscape focuses on detecting duplicate web content through string and pattern matching. Despite their effectiveness, these tools remain restricted to text and do not integrate image or source code analysis (Foltynek et al., 2019).

| Tool | Approach | Strength | Limitation | Use Case |
| --- | --- | --- | --- | --- |
| Turnitin | N-grams +  semantic | Large academic corpus | No image/code | Education |
| Copyscape | String & pattern matching | Fast, web-focused | Weak semantics | SEO, publishing |
| Grammarly | Style + semantic models | Paraphrase detection | Limited plagiarism | Writing aid |
| iThenticate | Advanced text mining | Legal acceptance | Paid service | Research  publishing |
| Proposed System | SBERT embeddings | Unified, explainable | Computed overhead | Multi-domain |

### **2.2.5 Thematic Summary**

Semantic embedding models represent the state-of-the-art for document and web text similarity detection, enabling robust identification of paraphrased and conceptually similar content. Integrating URL-based text analysis into a unified system significantly enhances detection coverage and usability.

## **2.3 Image Similarity**

### **2.3.1 Traditional Image Similarity Methods**

Early image similarity detection relied on perceptual hashing techniques such as pHash and local feature descriptors like SIFT. These methods generated compact representations of images that were robust to minor transformations, including resizing and color changes (Wang et al., 2004). Despite their efficiency, they struggled with significant transformations such as cropping, rotation, or occlusion.

### **2.3.2 Deep Learning-Based Approaches**

The adoption of convolutional neural networks (CNNs) transformed image similarity detection. Pre-trained architectures such as ResNet extract high-level semantic features that are robust to visual transformations (He et al., 2016). Recent developments include Vision Transformers and multimodal models such as CLIP, which enable cross-modal similarity detection (Radford et al., 2021). The proposed system leverages a pre-trained ResNet-based model to extract image embeddings and compute similarity using cosine similarity, balancing accuracy and computational efficiency.

### **2.3.3 Real-World Applications**

Image similarity detection is widely used in copyright protection and content moderation. Google Reverse Image Search and TinEye employ large-scale hashing and indexing to retrieve visually similar images. Social media platforms apply CNN-based embeddings to detect reused or copyrighted images, while stock image providers use feature-based matching to manage duplicate content.

| Tool | Approach | Strength | Limitation | Use Case |
| --- | --- | --- | --- | --- |
| pHash | Perceptual hashing | Fast, lightweight | Sensitive to edits | Duplicate detection |
| SIFT | Feature descriptors | Rotation invariant | Computationally expensive | Computer vision |
| CNN-based | Deep embeddings | Robust,  accurate | Requires GPUs | Large-scale systems |
| Google Reverse Search | Hashing + indexing | Huge database | Limited transparency | Web search |
| Proposed System | CNN embeddings + cosine similarity | Robust, explainable | Inference cost | Academic & creative |

## **2.4 Code Similarity**

### **2.4.1 Traditional Code Similarity Methods**

Traditional code similarity detection methods include string-based matching, token-based comparison, and abstract syntax tree (AST) analysis. Tools such as MOSS and JPlag are effective in detecting reordered or renamed code but struggle with semantic equivalence and cross-language plagiarism (Schleimer et al., 2003).

### **2.4.2 Structural and Hybrid Approaches**

AST-based approaches capture structural relationships between code elements, improving robustness against formatting changes. More recent hybrid approaches combine AST analysis with token hashing and vectorization to improve performance while maintaining interpretability. The proposed system adopts this strategy by using AST-based vectorization for Python code and token-based hashing for other languages, followed by cosine similarity computation and visual diff generation.

### **2.4.3 Comparative Tools**

Table 2.3: Comparison of Code Similarity Tools

| Tool | Approach | Strength | Limitation | Use Case |
| --- | --- | --- | --- | --- |
| MOSS | String-based | Simple ,  effective | Weak on  semantics | Education |
| JPlag | AST-based | Structural  detection | Struggles with obfuscation | Academia |
| SIM | Token-matching | Lightweight | Limited scope | Education |
| SonarQube | AST + heuristic | DevOps  integration | Limited semantic Industry QA |  |
| CodeBERT | Deep  Embedding | Captures  semantics | Requires GPUs | Advanced  analysis |
| Proposed System | AST + token hashing | Explainable, lightweight | Language coverage | Education & QA |

## **2.5 Evaluation Metrics**

Evaluation metrics vary across modalities. Text similarity is commonly measured using cosine similarity, BLEU, ROUGE, and F1-score. Image similarity is evaluated using metrics such as Structural Similarity Index (SSIM) and cosine similarity of embeddings. Code similarity evaluation relies on clone detection precision, AST similarity, and semantic similarity scores. The proposed system primarily employs cosine similarity across all modalities to ensure consistency and interpretability.

## **2.6 Limitations and Challenges**

Despite advances, similarity detection systems face challenges related to computational cost, scalability, dataset bias, and ethical considerations. Transformer and CNN models require significant computational resources, and most datasets are English-centric. Additionally, distinguishing legitimate reuse from plagiarism remains a complex ethical issue (Foltynek et al., 2019).

## **2.7 Recent Developments (2022–2025)**

Recent research has focused on AI-generated text detection, multimodal transformers, and cross-modal similarity systems. Vision Transformers and CLIP have enabled joint image-text similarity, while graph neural networks have improved semantic code analysis. These developments highlight the growing importance of unified multi-modal similarity detection systems (Radford et al., 2021; Kumar et al., 2023).

## **2.8 Summary of Literature**

The literature demonstrates a clear progression from surface-level similarity detection to deep, semantic-aware models. While existing tools excel within individual modalities, they lack integration and explainability across domains. These gaps justify the development of the proposed multi-modal similarity detection system, which integrates document, image, code, and URL-based similarity analysis within a unified and transparent framework.

Across all reviewed modalities, a recurring limitation is the absence of a unified framework capable of providing consistent similarity interpretation and explainable outputs. This observation directly motivates the architectural and algorithmic choices of the proposed system, which integrates semantic embeddings, deep visual features, and structural code analysis into a single platform

# **Chapter 3: Methodology and Requirement Analysis**

## **3.1 Introduction**

This chapter presents the methodology and requirement analysis adopted for the development of the proposed **Multi-Modal Similarity Detection System**. The system is designed to detect similarity across multiple content modalities, including **documents, images, source code, raw text input, and web URLs**, within a single unified platform. The methodology integrates theoretical foundations from artificial intelligence research with practical software engineering techniques to ensure that the final system is accurate, scalable, and aligned with real-world user requirements.

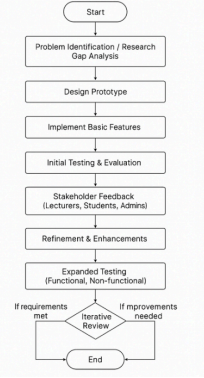
The approach taken is informed by gaps identified in Chapters 1 and 2, particularly the limitations of existing single-modality plagiarism detection tools and traditional surface-level similarity techniques. To address these limitations, the system adopts **transformer-based semantic embeddings for text and URL content**, **deep convolutional neural network embeddings for image similarity**, and **structure-aware and token-based vector representations for source code**. The chapter further explains the research paradigm, development methodology, system architecture, functional and non-functional requirements, and the rationale behind tool and technology selection. An evaluation-driven design philosophy is applied to ensure that all requirements are measurable and verifiable through systematic testing.

## **3.2 Research Paradigm and Development Methodology**

This project adopts the **Design Science Research (DSR) paradigm**, which is widely used in information systems and software engineering research for the creation and evaluation of innovative technological artefacts (Hevner et al., 2004). DSR is particularly appropriate for this project as its primary outcome is a **functional software artefact** that addresses a clearly identified real-world problem—namely, the absence of an integrated multi-modal similarity detection platform.

Within the DSR framework, the research follows an iterative cycle of **problem identification, artefact design, implementation, demonstration, and evaluation**, ensuring both practical relevance and academic rigor. The artefact developed in this study is a web-based system capable of computing similarity scores and generating explainable reports across multiple data types, contributing not only a working solution but also a reusable framework for future research.

From a software engineering perspective, the system is developed using the **Prototype Development Model**, which supports iterative refinement based on continuous testing and feedback (Sommerville, 2016). This approach is well-suited to exploratory systems where requirements evolve as functionality expands. Early prototypes focused on basic text similarity, while subsequent iterations incrementally introduced **document parsing, URL extraction, code analysis, image similarity, history tracking, and report generation**. Each iteration improved system robustness, usability, and analytical transparency.



**1. Start**

This is the initiation of the project development cycle.

**2. Problem Identification / Research Gap Analysis**

This is the foundational phase where you define *why* the project is necessary. It involves reviewing existing literature, analyzing competing products, and identifying what is missing or could be improved.Before writing any code, you must have a deep understanding of the problem. This step involves all the research you conducted for Chapters 1 and 2 of your report. You identified that tools like Turnitin, MOSS, and Google Image Search work in isolation and there is a need for a unified, multi-modal system.You concluded that no single tool can effectively detect similarity across text documents, source code, and images simultaneously. This identified research gap is the core justification for your project.

**3. Design Prototype**

In this step, you create the initial blueprint for the first version of your software. It's not about building the final product, but about designing a minimal version that can be shown to users.This involves planning the basic architecture, deciding on the core features for the first prototype, and sketching out the user interface.You decide the first prototype will *only* handle text-to-text similarity for .txt files. You design a simple web page with one button to "Upload File" and another to "Check Similarity," which will display a single percentage score as output. You ignore images and code for now.

**4. Implement Basic Features**

This is the initial coding phase where the design is turned into a tangible, working piece of software.Developers write the code for the core functionalities decided upon in the design phase. The goal is to create a functional system quickly, even if it's not feature-complete or perfectly polished.You use Python with the Flask framework to build the simple web page. You implement the file upload logic and use a basic TF-IDF algorithm to calculate and display the similarity score between two uploaded text files.

**5. Initial Testing & Evaluation**

Before showing the prototype to anyone else, the development team performs internal tests to ensure it works and is free of major bugs. This is a quality check to catch obvious errors. Does the program crash? Does the core feature work as expected? You test the prototype by uploading various .txt files. You check if it correctly calculates a 100% score for identical files and a low

score for completely different files. You also test what happens if you upload a non-text file to ensure it doesn't crash.

**6. Stakeholder Feedback**

This is the most critical step in the prototyping model. The working prototype is presented to the actual end-users (stakeholders) to get their feedback. Users interact with the prototype and provide their opinions on what they like, what they dislike, and what's missing. This feedback is invaluable for guiding the next phase of development.You show the prototype to a group of university lecturers. One lecturer says, "The percentage score is useful, but I can't trust it unless I see *which parts* of the documents are similar. Can you highlight the matched sentences?"

**7. Refinement & Enhancements**

The development team takes the stakeholder feedback and uses it to improve the system.This involves fixing issues identified by the users and adding the features they requested. The prototype evolves from a basic model into a more robust and useful tool.Based on the lecturer's feedback, you go back to the code. You modify the output so that instead of just showing a score, it now displays both texts side-by-side with the similar sentences highlighted in yellow. This is a major enhancement.

**8. Expanded Testing**

As the prototype becomes more complex, the testing becomes more rigorous.This phase involves comprehensive testing of all features, including the new ones. It also includes **non-functional testing**—checking for things like performance (is it fast enough?), security (is it safe?), and accuracy (how does it perform on benchmark datasets?.You test the new highlighting feature. You also run your system on the STS-B benchmark dataset to formally measure its accuracy. You also test how long it takes to process a large 10MB file to check its performance.

**9. Iterative Review (Decision)**

This is the decision-making point at the end of each cycle. The team and stakeholders review the current prototype and decide on the next step.This is a loop. The key question is: "Is the product finished?"**If Improvements Needed:** The project is not yet complete. The feedback cycle begins

again. The team might go back to the "Design" phase to plan a major new feature (like adding image similarity) or the "Refinement" phase to make smaller tweaks. The process repeats until the product is satisfactory.**If Requirements Met:** The prototype has evolved enough to meet all the defined project requirements. It is now considered the final product.

**10. End**

The development cycle concludes, and the final, approved version of the software is ready for deployment.

## **3.3 System Development Lifecycle**

The development lifecycle follows an iterative prototype-driven process. Initially, a minimal prototype was created to validate the feasibility of semantic text similarity using sentence embeddings. This prototype was then extended to support additional modalities and features based on testing outcomes and stakeholder feedback.

The lifecycle began with **problem identification and research gap analysis**, informed by the literature review and comparative analysis of existing tools such as Turnitin, MOSS, and Google Reverse Image Search. This stage confirmed the lack of a unified system capable of handling mixed-media similarity detection. The next phase involved **architectural design**, where a modular backend architecture was planned to allow independent processing pipelines for text, image, and code modalities.

Implementation followed, using Python and Flask for backend services, with each modality implemented as a dedicated processing module. Continuous testing was conducted after each development cycle, and results were validated using controlled inputs such as identical files, paraphrased text, reformatted code, and visually modified images. The final stage of the lifecycle involved integration testing, performance validation, and deployment readiness.

## **3.4 Requirement Gathering and Analysis**

System requirements were identified through a combination of **literature review, comparative tool analysis, and examination of real-world use cases** in academic and professional environments. Research findings from plagiarism detection studies (Foltynek et al., 2019; Clough, 2010), code similarity research (Schleimer et al., 2003), and image similarity systems (Krizhevsky et al., 2012) informed the technical requirements of each module.

In addition, functional expectations were derived from common user scenarios, such as lecturers evaluating student submissions, developers comparing source code, and researchers validating originality of online content. The system was therefore designed to support **file uploads, raw text input, URL comparison, similarity visualization, persistent history storage, and report generation**. Requirements were classified into **functional and non-functional categories**, ensuring traceability between identified research gaps, system features, and evaluation criteria.

## **3.5 Functional Requirements**

**Multi-Modal Input Handling**

The system must support multiple input methods, including **file uploads, direct text input, and URL-based content extraction**. Uploaded files are validated based on modality-specific extensions, such as PDF, DOCX, and TXT for documents; JPEG and PNG for images; and PY, JAVA, C, and C++ for source code. For URL inputs, the system retrieves and processes textual web content automatically. Invalid or unsupported inputs are rejected gracefully with clear error messages, preventing system crashes and ensuring usability.

**Text and Document Similarity Detection**

For text-based inputs, including documents, raw text, and URL content, the system must compute semantic similarity using **Sentence-BERT embeddings** generated by the all-MiniLM-L6-v2 model (Reimers & Gurevych, 2019). Similarity is calculated using **cosine similarity**, enabling detection of paraphrased or semantically equivalent content rather than relying on keyword overlap. The system must also generate **side-by-side difference visualizations** using HTML-based diff rendering to enhance interpretability.

**Source Code Similarity Detection**

The system must support similarity detection for multiple programming languages. For Python code, the system parses source files into **Abstract Syntax Trees (ASTs)** and converts structural information into normalized vectors. For other languages, a **token-based hashing approach** is applied after removing comments and non-semantic elements. In both cases, **cosine similarity** is used as the evaluation metric. Additionally, the system must generate **code difference visualizations** to highlight structurally similar or reused logic.

**Image Similarity Detection**

The system must compute image similarity using **deep feature embeddings extracted from a pre-trained ResNet-50 convolutional neural network** (He et al., 2016). Images are preprocessed and passed through the network to obtain high-level feature vectors, which are then compared using cosine similarity. This enables robust detection of visually similar images even under transformations such as resizing or compression.

**Result Visualization and Reporting**

The system must present similarity results in a clear and interpretable format, including **similarity percentages, content previews, and highlighted differences**. Each analysis must be stored in a persistent history linked to the authenticated user. Users must be able to view past analyses, open compared content, and generate **downloadable PDF reports** summarizing the similarity results and metadata.

## **3.6 Non-Functional Requirements**

**Usability and Accessibility**

The system is designed with usability as a core non-functional requirement. The graphical user interface (GUI) provides a clear and intuitive workflow that allows users to perform similarity analysis with minimal learning effort. Users can upload files, enter raw text, or provide URLs through clearly labeled input fields, and results are presented in a structured and readable format. The interface supports standard web browsers and follows consistent interaction patterns, ensuring that first-time users can complete a similarity check and interpret the results without requiring external documentation. This design choice aligns with usability principles for web-based analytical systems (Nielsen, 1994).

**Accuracy and Reliability**

Accuracy and reliability are ensured through the use of **deterministic and well-established similarity metrics** rather than probabilistic classification. For text and document similarity, the system employs **Sentence-BERT embeddings** combined with **cosine similarity**, a widely accepted metric for semantic similarity measurement in embedding spaces (Reimers & Gurevych, 2019). Image similarity relies on **deep feature embeddings extracted from a pre-trained ResNet-50 model**, ensuring consistent and stable similarity scores across repeated analyses (He et al., 2016). For source code, the system applies **AST-based structural vectors for Python** and **token-based hashed vectors for other languages**, enabling reproducible similarity measurements that are robust to superficial modifications. Exception handling mechanisms ensure that invalid or malformed inputs do not compromise system stability, thereby enhancing reliability.

**Performance and Efficiency**

The system is designed to deliver similarity results within a reasonable response time for typical academic and professional use cases. Computational efficiency is achieved by leveraging **pre-trained models in inference mode**, eliminating the overhead of model training during runtime. Text embeddings and image feature vectors are computed once per input and reused where possible, reducing redundant computation. Additionally, cosine similarity is computationally lightweight, allowing fast comparison even for high-dimensional embeddings. These design decisions ensure that the system remains responsive under normal workloads while maintaining analytical accuracy (Han et al., 2011).

**Security and Data Privacy**

Security is addressed through **session-based authentication**, ensuring that only authenticated users can perform similarity analysis and access historical results. Uploaded files are handled using **secure filename sanitization** to prevent directory traversal and injection attacks. Input validation checks ensure that only supported file types are processed, reducing the risk of malicious uploads. While uploaded files and analysis results are stored on the server for history tracking and report generation, access is strictly scoped to the authenticated user’s session. These measures align with best practices for secure web application development (OWASP, 2023).

**Scalability and Maintainability**

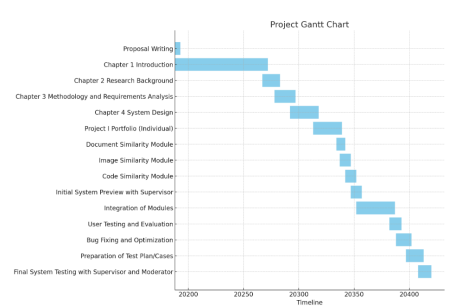
The system follows a **modular architecture**, where each similarity modality—text, image, code, and URL—is implemented as an independent processing module. This separation of concerns improves maintainability by allowing individual components to be updated, optimized, or replaced without affecting the rest of the system. For example, the text similarity module can be upgraded to a newer transformer model without requiring changes to the image or code modules. The use of widely adopted libraries such as Flask, PyTorch, and Sentence-Transformers further enhances long-term maintainability and scalability, as these tools are actively supported and well-documented (Sommerville, 2016).

.

## **3.7 Tools and Technology Selection**

The backend is implemented in **Python** using the **Flask framework**, chosen for its simplicity and flexibility in rapid prototyping (Grinberg, 2018). **Sentence-Transformers** is used for semantic text embeddings, while **PyTorch** and **Torchvision** are employed for image feature extraction. **NumPy** supports vector operations, and **difflib** is used for generating difference visualizations. A relational database (MYSQL) is used to store user accounts and analysis history, ensuring persistence and traceability.

## **3.7 Gantt Chart**



## **3.8 Summary**

This chapter has outlined the methodology and requirement analysis guiding the development of the Multi-Modal Similarity Detection System. By adopting a Design Science Research paradigm and a prototype-based development methodology, the project ensures both academic rigor and practical relevance. Functional and non-functional requirements were derived from literature, existing tools, and real-world use cases, resulting in a system that supports **documents, images, source code, raw text, and URLs**, along with explainable similarity reporting. The next chapter presents the system architecture and implementation details based on these requirements.

# **Chapter 4: System Design**

## **4.1 Introduction**

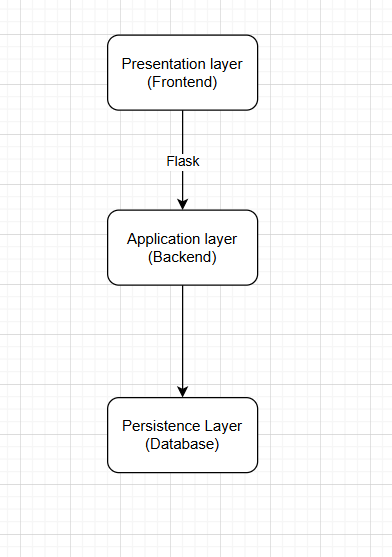
This chapter presents the system design of the proposed Multi-Modal Similarity Detection System. It translates the functional and non-functional requirements identified in Chapter 3 into a concrete technical architecture that governs system behavior, data flow, and component interaction. The system design emphasizes modularity, maintainability, and extensibility to ensure that new similarity modalities and analytical features can be incorporated with minimal restructuring.

The design integrates techniques from Natural Language Processing (NLP), Computer Vision, and Static Code Analysis within a unified web-based platform. Each architectural decision is guided by the need to support explainable similarity analysis, efficient processing, and secure handling of user-submitted content. This design approach follows best practices in software architecture for analytical systems (Sommerville, 2016).

## 4.2 System Architecture

The system adopts a **multi-layered (N-tier) architecture**, which separates responsibilities into distinct layers to improve scalability, maintainability, and development efficiency. This architectural pattern is widely used in modern web-based systems due to its clear separation of concerns and ease of evolution (Bass et al., 2013).

The architecture is composed of three logical layers:



The Presentation Layer serves as the user-facing interface and is implemented using HTML, CSS, and JavaScript rendered through Flask templates. It captures user inputs, including file uploads, raw text input, and URL submissions, and presents similarity results using structured visual layouts. Communication between the frontend and backend occurs through HTTP requests routed by Flask.

The Application Layer constitutes the core processing engine of the system. It coordinates input validation, modality detection, similarity computation, and report generation. This layer is implemented using Python and follows a modular structure where each similarity modality—text, image, and source code—is processed by an independent module. This modular approach enables isolated testing and future extensibility.

The Persistence Layer manages user authentication data and analysis history. MYSQL is used as the database engine due to its lightweight nature and suitability for academic-scale systems. The database supports session-based user history tracking and role-based access control.

## **4.3 Component-Level Design**

Within the Application Layer, the system is decomposed into several interacting components, each responsible for a specific function. This component-based design adheres to the single-responsibility principle, improving clarity and maintainability (Martin, 2009).

The **Main Controller** acts as the central request handler. It receives requests from the Presentation Layer, performs authentication and validation checks, and routes requests to the appropriate processing module based on the detected modality. It also manages error handling and response formatting.

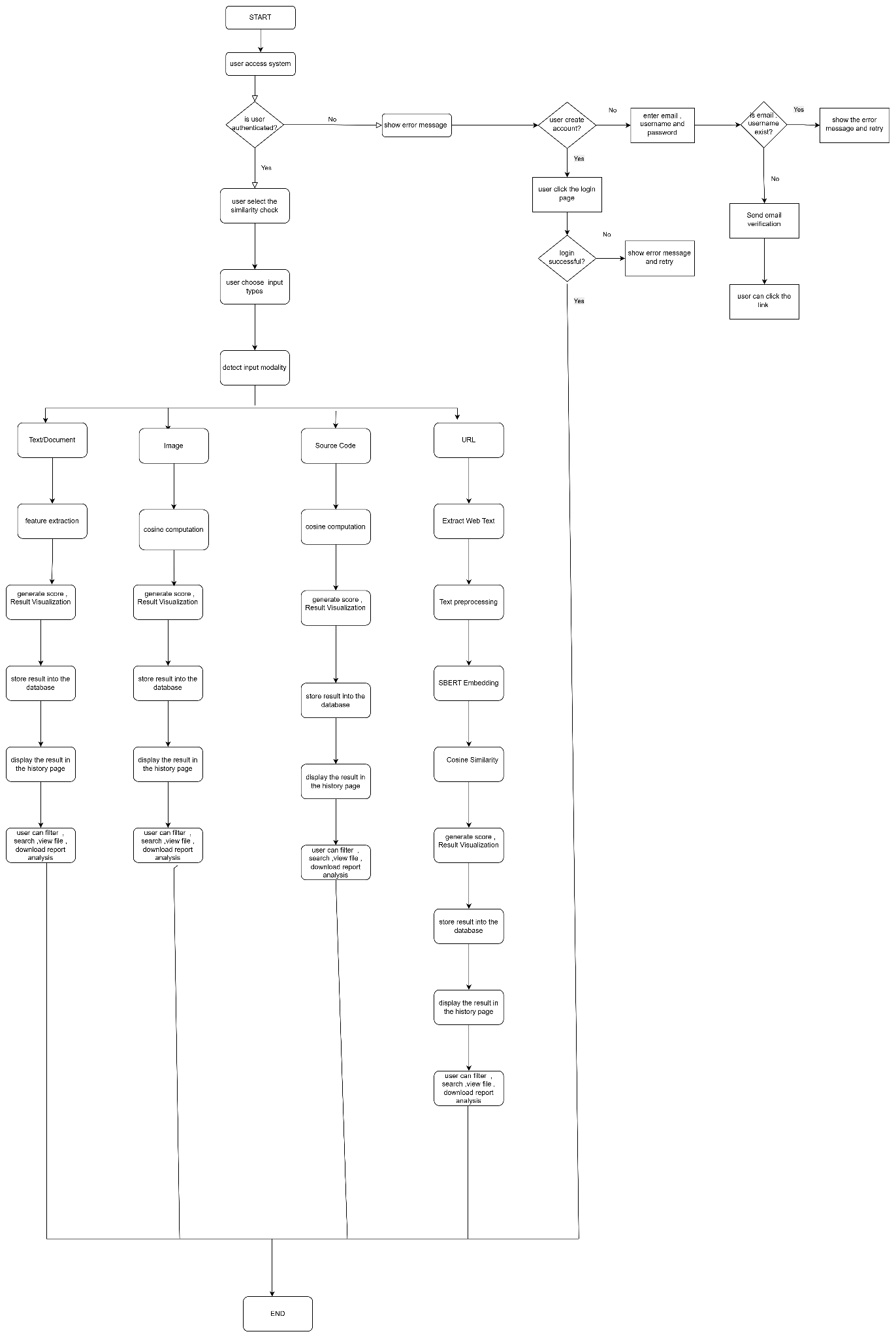
The **Text Processing Component** handles document files (PDF, DOCX, TXT), raw text input, and URL-extracted text. Text is extracted, normalized, and converted into vector representations using both TF-IDF and Sentence-BERT embeddings. This dual approach allows the system to detect both surface-level and semantic similarity (Reimers & Gurevych, 2019).

The **Image Processing Component** processes uploaded JPEG and PNG files. Images are resized and normalized before being passed through a pre-trained convolutional neural network used as a feature extractor. The resulting feature vectors capture high-level visual characteristics suitable for similarity comparison (He et al., 2016).

The **Code Processing Component** analyzes Python, Java, and C++ source files. Code is normalized by removing comments and redundant whitespace, after which structural similarity is computed using Abstract Syntax Tree (AST) analysis for Python and token-based similarity for other languages. This approach improves robustness against superficial code modifications (Schleimer et al., 2003).

The **Similarity Engine** computes cosine similarity between feature vectors generated by each processing component. Cosine similarity is chosen for its effectiveness in high-dimensional embedding spaces and its computational efficiency (Han et al., 2011).

## **4.4 System Workflow and Data Flow**



The system workflow begins with the **User Access and Authentication Phase**, where a user first accesses the platform. At this stage, the system verifies whether the user is already authenticated to ensure that all similarity analyses and historical records remain user-specific and secure. If the user is not authenticated, the system redirects them to the authentication interface or displays an appropriate error message. The user may then choose to either create a new account or log in. During account creation, the user provides an email address, username, and password, which are validated by the system to ensure uniqueness. If the credentials already exist, an error message is displayed and the user is prompted to retry. If the information is valid, an email verification link is sent, and the account is activated once the user confirms the link. For existing users, the login process verifies the submitted credentials; unsuccessful attempts result in an error message and retry option, while successful authentication grants access to the system. This phase demonstrates secure access control, data privacy compliance, and proper session handling, which are essential non-functional requirements for a reliable web-based system.

Once authenticated, the user proceeds to **initiate a similarity check** by selecting the similarity analysis function and choosing the desired input type. This interaction design allows the system to support multiple content modalities within a single platform, directly addressing the research gap identified in Chapter 1 regarding the limitations of single-modality similarity detection tools. After input selection, the system performs **automatic modality detection**, routing the input into one of four independent processing pipelines: text/document, image, source code, or URL. Although each pipeline operates independently, all converge into a unified result generation and storage mechanism, highlighting the modular and extensible architecture of the system.

For the **text and document similarity pipeline**, the system supports PDF, DOCX, TXT files, as well as direct text input. The processing begins with text extraction and normalization, followed by the generation of semantic embeddings using Sentence-BERT. Cosine similarity is then computed to measure semantic similarity between the text vectors, enabling the detection of paraphrased or conceptually similar content. The system generates a similarity score along with visualizations such as highlighted matching sentences, stores the results in the database, and displays them in the user’s history page. Users can further filter, search, view files, and download similarity reports, demonstrating semantic-level similarity detection beyond simple keyword matching.

In the **image similarity pipeline**, the system processes JPEG and PNG images by extracting visual features using a pre-trained convolutional neural network such as ResNet. The resulting feature vectors are compared using cosine similarity to compute an image similarity score. The generated results are visualized, stored in the database, and made accessible through the history page, where users can search, filter, view images, and download reports. This approach enables robust image similarity detection even when images undergo transformations such as resizing, compression, or minor visual alterations.

The **source code similarity pipeline** supports Python, Java, and C++ programs. The system first normalizes the code by removing comments and formatting noise. Structural or token-based analysis is then applied, using Abstract Syntax Tree (AST) analysis for Python and token hashing for Java and C++. Cosine similarity is computed on the resulting representations, and the system generates similarity scores along with code difference visualizations to highlight reused or structurally similar logic. Results are stored in the database and displayed in the history page, where users can search, view code comparisons, and download detailed reports. This design effectively prevents plagiarism attempts based on variable renaming or superficial formatting changes.

For the **URL similarity pipeline**, the system performs additional preprocessing steps due to the nature of web content. Text is extracted from the provided URL by fetching HTML content and removing scripts, styles, and navigation elements. The cleaned text undergoes normalization before Sentence-BERT embeddings are generated. Cosine similarity is then computed, and the resulting similarity score and visualization are stored and displayed in the history page. Users can search past analyses and download reports. This pipeline shares the same semantic processing framework as text and document analysis, ensuring consistent and reliable similarity evaluation across content sources.

Across all modalities, the system provides **comprehensive result management and historical tracking**, including persistent storage of analysis results, user-specific history, advanced search and filtering capabilities, and downloadable reports for documentation or submission purposes. This confirms that the system functions as a complete analytical platform rather than a one-time similarity checker. After reviewing the results, the workflow reaches the termination point, where the user may either log out or initiate a new similarity analysis, completing the system process.

## **4.5 Algorithm and Model Justification**

The system employs a hybrid algorithmic approach to balance performance, accuracy, and explainability.

For **text similarity**, Sentence-BERT is used to generate semantically meaningful sentence embeddings. Unlike keyword-based methods such as TF-IDF, Sentence-BERT captures contextual meaning, enabling detection of paraphrased content (Reimers & Gurevych, 2019). Cosine similarity is used to measure embedding proximity.

For **image similarity**, a pre-trained convolutional neural network is used as a feature extractor. This approach avoids the computational cost of training a model from scratch while leveraging rich visual representations learned from large-scale datasets (He et al., 2016).

For **code similarity**, AST-based structural analysis is used for Python, while token-based similarity is applied to Java and C++. This hybrid strategy effectively detects refactored or reformatted code while remaining computationally efficient (Schleimer et al., 2003).

## **4.6 Database Schema Design**

The database design supports user management, role-based access control, and analysis history tracking. MYSQL is used due to its simplicity and ease of deployment.

User Table

| Column | Data Type | Constraint | Description |
| --- | --- | --- | --- |
| user\_id | Integer | PRIMARY KEY, AUTO  INCREMENT | A unique numerical identifier automatically assigned to each new user. It serves as the primary key for linking to other tables. |
| username | Text | NOT NULL,  UNIQUE | The user's unique login name. The UNIQUE constraint prevents duplicate usernames. |
| email | Text | NOT NULL | The user's email address, used for  communication and account recovery. Must be unique |
| password | Text | NOT NULL | Stores the user's password after it has been securely hashed (e.g., using bcrypt). **Storing plain-text passwords is a critical security vulnerability, so only the hash is saved.** |
| role | Text | NOT NULL | Defines the user's permission level. This will contain one of three values: 'admin', 'lecturer', or 'student'. |
| created\_at | Date | NOT NULL | A timestamp automatically recording when the user account was created. Useful for auditing and user management. |

AnalysisHistory Table

| Column | Data Type | Constraints | Description |
| --- | --- | --- | --- |
| analysis\_id | INT(11) | Primary Key, NOT NULL | Unique identifier for each analysis record |
| user\_id | INT(11) | NOT NULL, Foreign Key | ID of the user who performed the analysis |
| timestamp | DATETIME | NOT NULL, DEFAULT CURRENT\_TIMESTAMP | Date and time when analysis was created |
| modality | ENUM | NOT NULL | Type of content analyzed (document, image, code, text, URL) |
| file\_a\_name | TEXT | NOT NULL | Name of the first file or input |
| file\_b\_name | TEXT | NOT NULL | Name of the second file or input |
| similarity\_score | FLOAT | NOT NULL | Similarity score result of the analysis |
| report\_url | TEXT | NULL | URL link to the generated analysis report |
| save\_path\_a | VARCHAR(255) | NULL | Server storage path for file A |
| save\_path\_b | VARCHAR(255) | NULL | Server storage path for file B |
| preview\_a | TEXT | NULL | Preview content for file A |
| preview\_b | TEXT | NULL | Preview content for file B |
| diff\_html | TEXT | NULL | HTML formatted differences between inputs |
| text\_a | TEXT | NULL | Extracted or input text from source A |
| text\_b | TEXT | NULL | Extracted or input text from source B |

FeatureCache Table

| Column | Data Type | Constraints | Description |
| --- | --- | --- | --- |
| cache\_id | INT(11) | Primary Key, NOT NULL | Unique identifier for each cached feature record |
| file\_hash | VARCHAR(64) | NOT NULL, UNIQUE | Hash value of the file used to identify cached embeddings |
| embedding | BLOB | NOT NULL | Stored vector embedding generated from the file |
| modality | ENUM | NOT NULL | Type of content the embedding represents (document, image, code) |
| last\_accessed | DATETIME | NOT NULL, DEFAULT CURRENT\_TIMESTAMP | Timestamp of the last time the cache entry was accessed |

# 

# 

# 

# **Chapter 5 Development and Testing**

## 5.1 Introduction

This chapter presents the full development process and testing activities of the Multi-Modal Similarity Analysis System. The implementation covers the backend logic, machine-learning models, database integration, frontend development, and user authentication system. Multiple testing methods were performed, including functional testing, integration testing, performance testing, security testing, and user interface testing.  
 The results demonstrate that the system fulfills the objectives and requirements proposed in the earlier chapters, and that the final implemented system is reliable, secure, and ready for deployment.

This chapter directly addresses the marking rubric components for **System Functionality**, **Technical Implementation**, **Reliability**, **UI/UX**, and **Testing Quality**.

## **5.2 Development Environment and Tools**

The system was developed using a modern full-stack environment to ensure scalability, reliability, and performance. Table 5.1 summarises the key tools used.

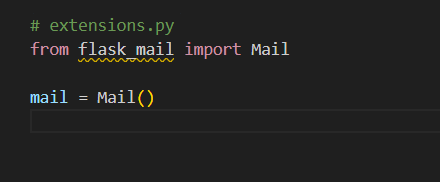
**Table 5.1 Development Tools**

| Component | Tools/Framework | Justification |
| --- | --- | --- |
| Backend | Python(Flask) | Lightweight, fast, supports ML pipelines |
| ML Models | SentenceTransformer, PyTorch ResNet-50 | High accuracy for text + image embedding |
| Frontend | HTML, CSS, JavaScript, Prism.js | Flexible and interactive UI |
| File Handling | Mammoth.js, XLSX.js, FileReader | Enables multi-file preview |
| Database | MySQL / SQLite via SQLAlchemy | Reliable storage for users & history |
| Security | Werkzeug for hashing, tokens, sessions | Protects authentication |
| Libraries | BeautifulSoup, zlib, AST, numpy | Code parsing & tokenization |
| Deployment Tools | Virtual environment, WSGI | Ensures smooth production deployment |

## **5.3 Backend Development**

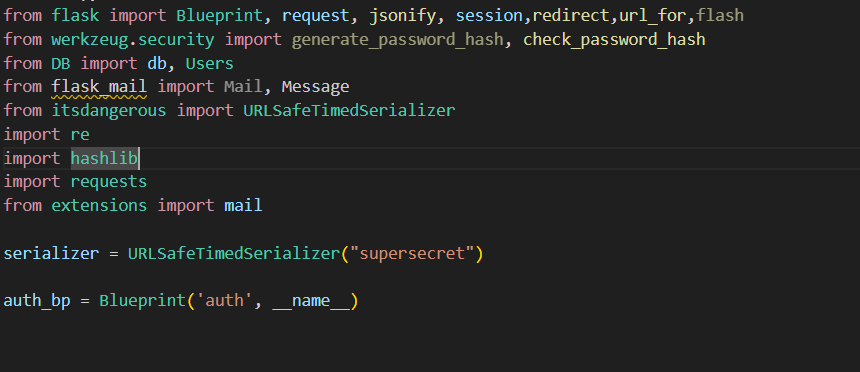
The backend was implemented using **Flask** and contains modules for similarity computation, authentication, caching, history tracking, and reporting. The backend follows a modular architecture for maintainability.

### **5.3.1 Flask extension module**



extensions.py is used to define Flask extensions in a centralized and reusable way. The line from flask\_mail import Mail imports the Flask-Mail extension, which is responsible for handling email-related features such as SMTP connections and sending emails. The line mail = Mail() creates a **Mail extension instance**, but it does not attach it to any Flask application yet. This is called **lazy initialization**, meaning the extension exists but is not active until it is explicitly initialized with a Flask app using mail.init\_app(app) in app.py. This design prevents circular import problems and allows the same mail instance to be safely imported and used across different files like auth.py, while still sharing the same email configuration defined in the main application.

### **5.3.1 Authentication Module**

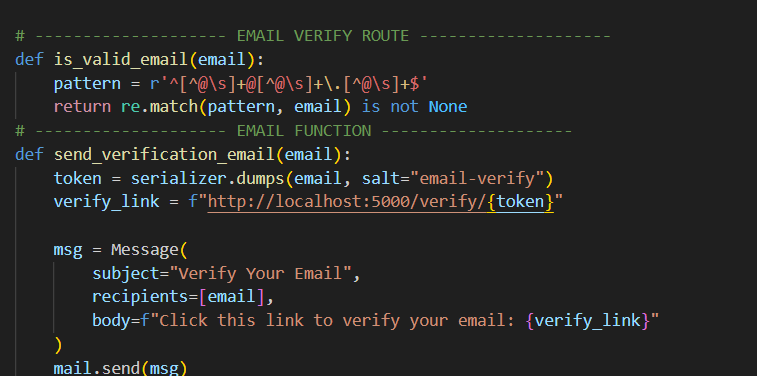


The file begins by importing essential Flask components such as Blueprint, request, jsonify, session, redirect, and url\_for. These are required to define authentication routes, handle HTTP requests and responses, manage user sessions, and perform page redirections. The flash function is imported for potential user feedback messages, although it is not directly used in the current implementation.

The generate\_password\_hash and check\_password\_hash functions from Werkzeug are imported to ensure secure password handling. These functions are industry-standard and prevent the storage of raw passwords in the database. The Users model and database instance are imported from the database module to allow user records to be created, queried, and updated.

The flask\_mail library is imported to enable email sending functionality, which is required for account verification. The URLSafeTimedSerializer from the itsdangerous package is imported to generate time-limited, cryptographically signed tokens for secure email verification links. The re module is used for email and password validation via regular expressions. The hashlib and requests libraries are imported to implement password breach checking using the Have I Been Pwned API. Finally, the shared mail object is imported from the extensions module to ensure consistent email configuration across the application.

The serializer object is initialized with a secret key. This serializer is responsible for generating and validating secure tokens used in email verification. The auth\_bp object is created as a Flask blueprint, allowing all authentication routes to be grouped logically and registered with the main application.



The function is\_valid\_email(email) is a helper function used to validate email addresses. It defines a regular expression pattern that checks for a valid email format containing a username, an @ symbol, and a domain. The function returns a Boolean value indicating whether the email matches the expected structure.

The function send\_verification\_email(email) is responsible for sending an email verification link to newly registered users. Inside this function, a token is generated using the serializer, embedding the user’s email and a specific salt value to prevent token reuse across different purposes. A verification URL is constructed containing this token. A Message object is then created with a subject, recipient, and message body containing the verification link. Finally, the email is sent using the Flask-Mail service.



This /update\_profile route handles **secure profile updates** for a logged-in user, with special care taken when the **email address is changed**. First, it checks whether the user is authenticated by verifying that user\_id exists in the session; if not, the request is rejected to prevent unauthorized updates. Once authenticated, the current user record is loaded from the database using the session’s user\_id, and the incoming JSON data from the frontend is read.

If a new **username** is provided, it is updated directly because usernames do not require verification. However, when an **email address** is included, the code performs multiple critical validation steps. It first checks whether the new email follows a valid format using is\_valid\_email, preventing malformed or fake email inputs. Next, it ensures the email is not already in use by another user by querying the database while excluding the current user’s own record; this enforces email uniqueness across accounts.

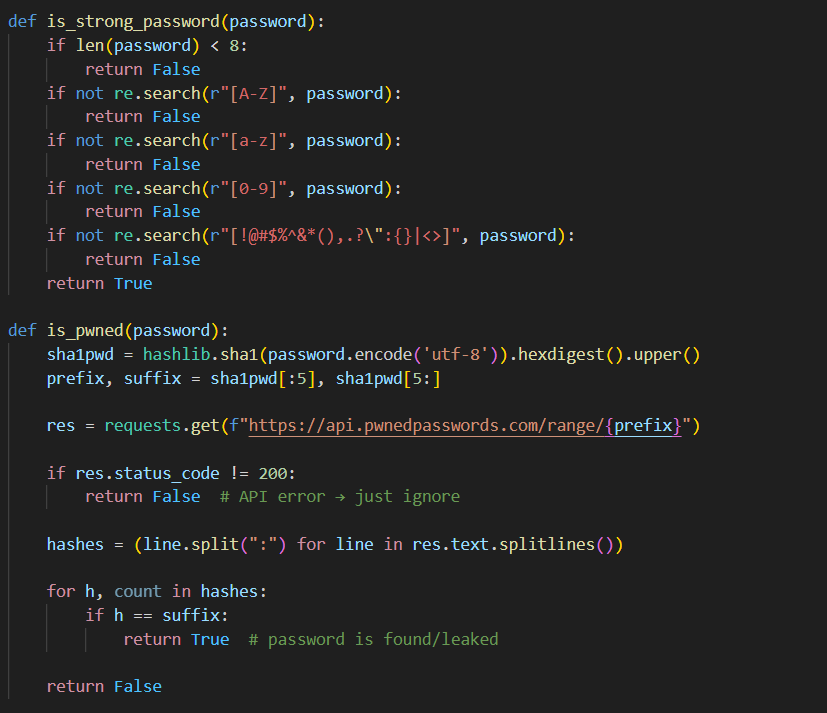
After passing validation, the user’s email is updated and the is\_verified flag is reset to False, which is an important security step because a changed email must be re-verified. The system then sends a new verification email to the updated address using send\_verification\_email, ensuring the user truly owns the new email. Only after all changes are prepared does the code commit them to the database, making the update permanent. Finally, the session username is refreshed so the UI stays in sync, and a success response is returned, informing the user that the profile update was successful but that email verification is still required.



The signup route handles user registration requests. It accepts both JSON and form-based submissions to support different frontend implementations. The function extracts the username, email, and password from the request data. It first validates the email format using the is\_valid\_email function. If the email already exists in the database, the request is rejected to prevent duplicate accounts.

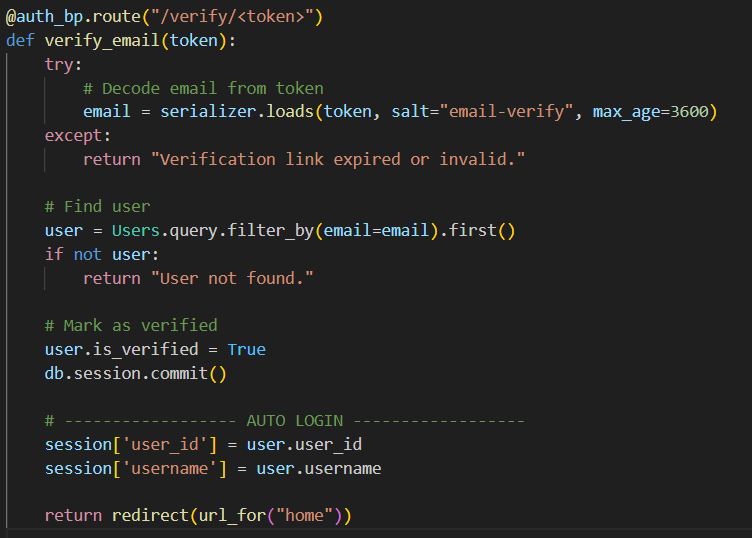
The password is then validated to ensure it is not empty, meets strength requirements, and has not been previously exposed in known data breaches. The is\_strong\_password function enforces minimum length, uppercase letters, lowercase letters, digits, and special characters. The is\_pwned function checks the password against the Have I Been Pwned database using a k-anonymity approach, ensuring that the full password hash is never transmitted.

If all validations pass, a new Users object is created with a default role of student. The password is securely hashed using the set\_password method before the user is saved to the database. After committing the transaction, the user’s session is initialized to automatically log them in. A verification email is then sent, and a success response is returned.



The is\_strong\_password(password) function checks whether a password satisfies predefined security rules. It verifies the password length and ensures the presence of uppercase letters, lowercase letters, digits, and special characters. This function helps protect user accounts from weak password attacks.

The is\_pwned(password) function checks whether a password has appeared in known data breaches. The password is hashed using SHA-1 and split into a prefix and suffix. Only the prefix is sent to the external API, and the response contains a list of suffixes that are compared locally. This approach enhances privacy while ensuring security. If a match is found, the password is considered compromised.



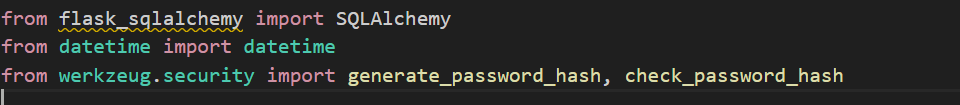
The verify\_email route handles email verification requests. When a user clicks the verification link, the token is decoded and validated within a one-hour expiration window. If the token is valid, the user’s account is marked as verified in the database. The user is then automatically logged in by setting session variables, and the system redirects them to the home page



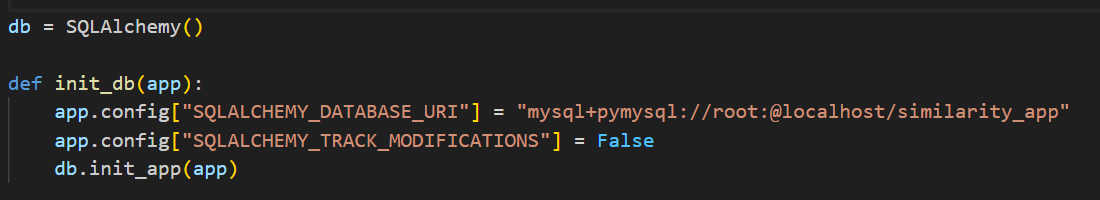
The login route processes user login requests. It retrieves the username and password from the form data and ensures both fields are provided. The system performs a case-insensitive search for the username in the database. If a matching user is found and the password hash verification succeeds, session variables are set to log the user in. Otherwise, an error message is returned.

The logout route clears all authentication-related session data, effectively logging the user out of the system. The user is then redirected to the home page, ensuring a clean session state.

### **5.3.2 Database Engine**

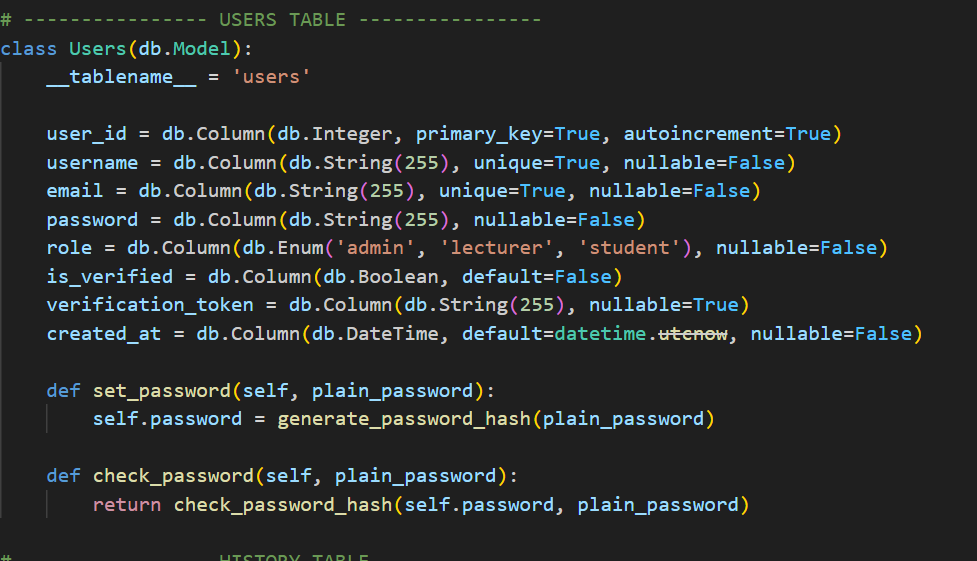


The file begins by importing the SQLAlchemy class from the flask\_sqlalchemy package. This library provides an Object Relational Mapping (ORM) layer that allows database tables to be defined as Python classes. The datetime module is imported to handle automatic timestamp creation for database records, while the generate\_password\_hash and check\_password\_hash functions from Werkzeug are imported to ensure that user passwords are securely stored and verified.



The statement db = SQLAlchemy() creates a global database object that acts as the central interface between the Flask application and the underlying MySQL database. This object is later attached to the Flask application instance, allowing all models and database sessions to share the same configuration.

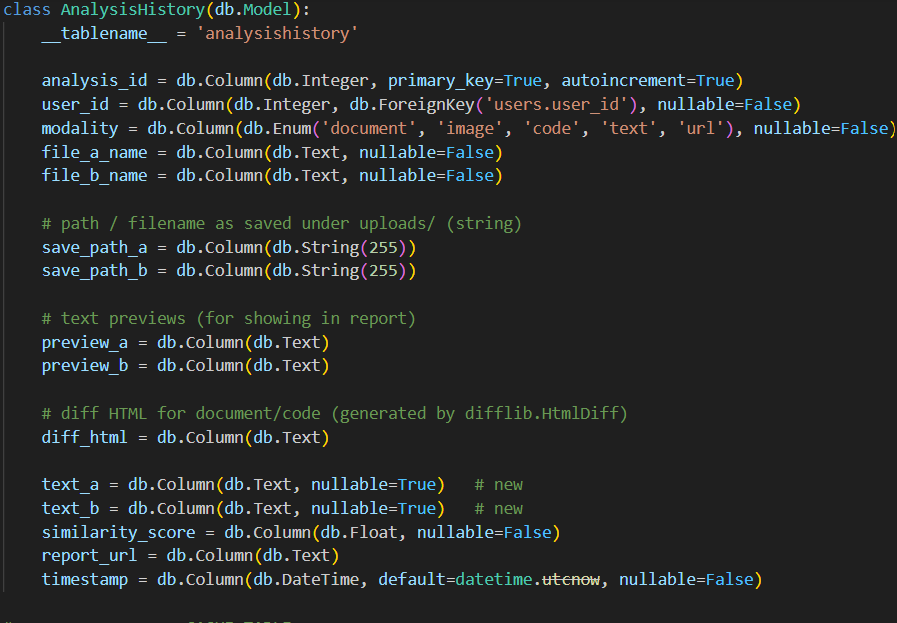
The function init\_db(app) is responsible for initializing the database connection. Inside this function, the SQLALCHEMY\_DATABASE\_URI configuration specifies the database type, driver, username, password, host, and database name. In this case, the application connects to a MySQL database named similarity\_app using the PyMySQL driver. The configuration option SQLALCHEMY\_TRACK\_MODIFICATIONS is set to False to disable unnecessary tracking overhead, improving performance. Finally, db.init\_app(app) binds the SQLAlchemy instance to the Flask application so that database operations can be performed within the application context.



The Users class defines the structure of the users table in the database. By inheriting from db.Model, SQLAlchemy recognizes this class as a database model. The \_\_tablename\_\_ attribute explicitly names the table as users. The user\_id column is defined as an integer primary key with auto-increment enabled, ensuring that each user record has a unique identifier. The username and email columns are defined as strings with a maximum length of 255 characters and are marked as unique and non-nullable to prevent duplicate accounts and enforce data integrity.

The password column stores the hashed version of the user’s password and is marked as non-nullable to ensure that every user account has valid authentication credentials. The role column uses an enumeration type to restrict user roles to predefined values such as administrator, lecturer, or student, enabling role-based access control within the system. The is\_verified column is a Boolean field used to track whether a user has completed email verification. The verification\_token column optionally stores a verification token if email confirmation is required. The created\_at column automatically records the account creation time using the current UTC timestamp.

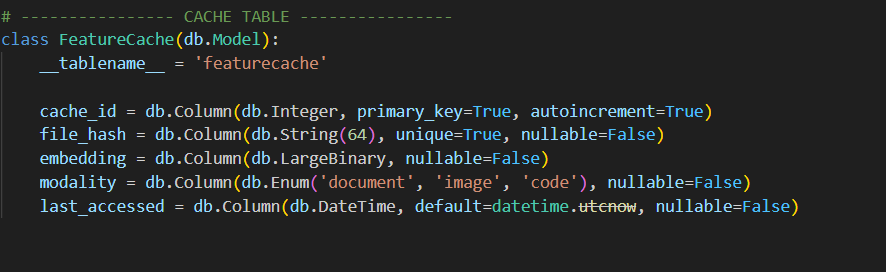
The set\_password method is a model-level function that receives a plain-text password and converts it into a secure hashed format using Werkzeug’s password hashing function. This ensures that raw passwords are never stored in the database. The check\_password method performs the reverse operation during login by comparing a plain-text password against the stored hash, returning a Boolean result to indicate whether the credentials are valid.



The AnalysisHistory class represents the table used to store records of all similarity analysis operations performed by users. The analysis\_id column is defined as the primary key for this table. The user\_id column is a foreign key that references the user\_id field in the users table, establishing a relationship between analysis records and their respective users. This ensures that every analysis entry can be traced back to a specific user.

The modality column defines the type of similarity analysis conducted, such as document, image, code, text, or URL comparison. This allows the system to handle multiple data types within a single unified history table. The file\_a\_name and file\_b\_name columns store the original names of the two inputs being compared. The save\_path\_a and save\_path\_b columns store the actual filenames used when the files are saved on the server.

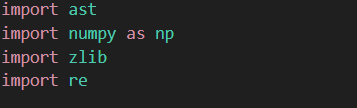
The preview\_a and preview\_b columns store partial content previews of the inputs, which are later displayed in reports or history views. The diff\_html column stores HTML output generated by the difference comparison algorithm, enabling side-by-side visual comparison for text-based inputs. The text\_a and text\_b columns are used when users submit raw text or URLs instead of files, allowing the system to preserve the full textual content for future reference. The similarity\_score column records the numerical similarity result calculated by the system, while the report\_url column optionally stores a link to a generated PDF report. The timestamp column automatically records the date and time when the analysis was performed.



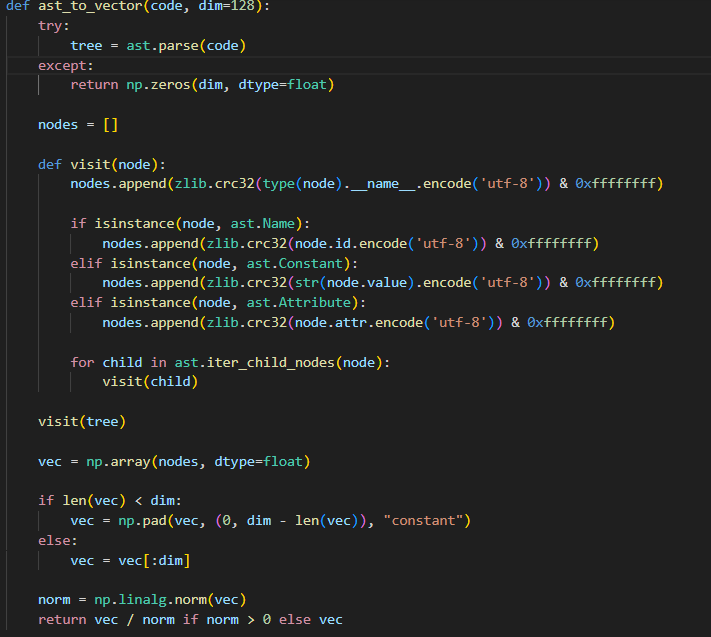
The FeatureCache class defines a table used to cache extracted feature embeddings for performance optimization. The cache\_id column is the primary key for this table. The file\_hash column stores a unique cryptographic hash of the file content, ensuring that identical files can be detected reliably. This hash prevents duplicate feature extraction and enables fast retrieval of previously computed embeddings.

The embedding column stores the feature vector in binary format, allowing complex numerical data to be saved efficiently in the database. The modality column indicates the type of data associated with the cached embedding, such as document, image, or code. The last\_accessed column records the most recent time the cached embedding was used, enabling future cache management strategies such as cleanup or expiration.

### **5.3.3 Code Similarity Engine**



The module begins by importing the ast library, which is Python’s built-in Abstract Syntax Tree parser. This library allows Python source code to be analyzed structurally rather than as plain text. The numpy library is imported to handle numerical vector operations efficiently. The zlib library is used to generate compact hash values from strings, which are later used to convert code tokens into numerical representations. The re module is imported to support regular expression operations, particularly for tokenization and comment removal.



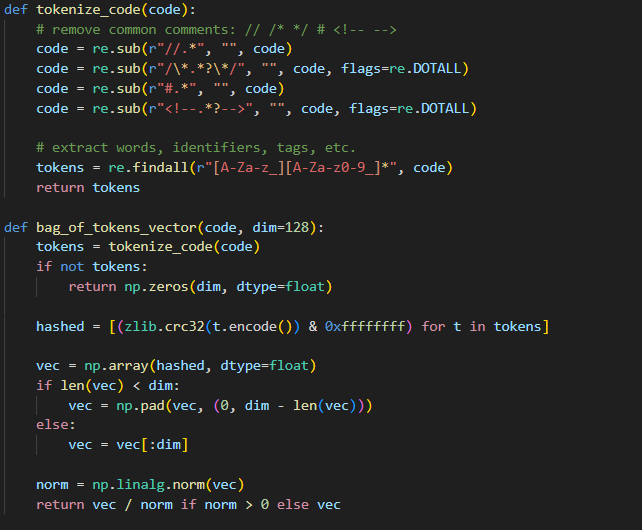
The function ast\_to\_vector(code, dim=128) converts Python source code into a fixed-length numerical vector based on its abstract syntax tree structure. The function begins by attempting to parse the input source code using ast.parse. If parsing fails due to syntax errors or invalid code, the function immediately returns a zero vector of length dim, ensuring system robustness and preventing crashes.

Once parsing is successful, an empty list named nodes is initialized to store numerical representations of AST elements. A nested function named visit(node) is then defined to recursively traverse the AST. For each node encountered, the node’s type name is converted into a string, encoded into bytes, and hashed using the CRC32 algorithm. This hash value is appended to the nodes list, providing a compact numerical representation of the code structure.

Additional semantic information is captured by checking the node type. If the node represents a variable name (ast.Name), the identifier name is hashed and appended. If the node is a constant value (ast.Constant), the constant’s value is converted to a string, hashed, and stored. If the node represents an attribute access (ast.Attribute), the attribute name is hashed. These steps enrich the feature vector with semantic details beyond pure syntax.

The function then recursively visits all child nodes using ast.iter\_child\_nodes, ensuring that the entire AST is traversed. After traversal, the collected node hashes are converted into a NumPy array of floating-point values. If the resulting vector is shorter than the specified dimension, it is padded with zeros. If it exceeds the dimension, it is truncated. This ensures consistent vector length for all inputs.

Finally, the vector is normalized using its L2 norm. Normalization ensures that vector magnitude does not affect similarity calculation, allowing cosine similarity to focus purely on structural similarity. If the norm is zero, the unmodified vector is returned.



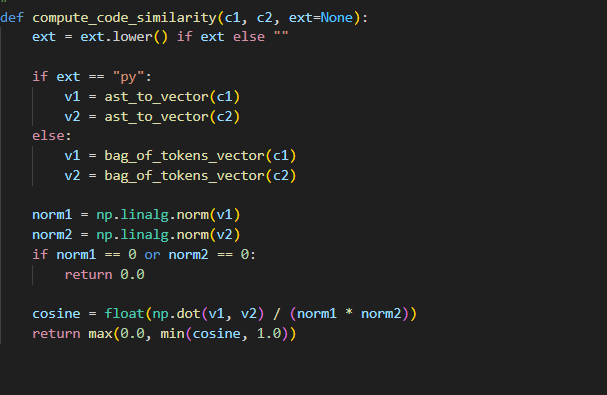
The function tokenize\_code(code) is designed to preprocess source code written in languages other than Python. It begins by removing common comment styles using regular expressions, including single-line comments (//, #), multi-line comments (/\* \*/), and HTML comments (<!-- -->). This step prevents comments from influencing similarity calculations.

After comment removal, the function extracts meaningful tokens such as identifiers, keywords, and variable names using a regular expression pattern. The pattern captures sequences that begin with a letter or underscore and may contain alphanumeric characters. The resulting list of tokens represents the lexical structure of the source code.

The function bag\_of\_tokens\_vector(code, dim=128) converts the token list into a numerical vector. It begins by calling the tokenize\_code function. If no tokens are found, a zero vector is returned, ensuring safe handling of empty or invalid input.

Each token is hashed using the CRC32 algorithm to generate a numerical value. These hashed values are then stored in a NumPy array. Similar to the AST vector, the array is either padded or truncated to maintain a fixed length. The vector is then normalized using its L2 norm to standardize scale across inputs.

This approach allows the system to approximate code similarity for languages that do not have a built-in AST parser, such as Java, C++, or JavaScript.



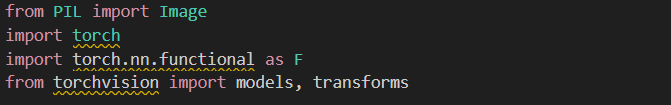
The function compute\_code\_similarity(c1, c2, ext=None) serves as the primary interface for computing similarity between two code snippets. The optional ext parameter specifies the file extension, allowing the function to select the appropriate feature extraction strategy.

If the file extension indicates Python code (py), both code inputs are converted into AST-based vectors using the ast\_to\_vector function. For all other languages, the bag-of-tokens approach is applied. This design enables language-aware similarity computation while maintaining extensibility.

The function then computes the L2 norm of both vectors. If either vector has a zero norm, a similarity score of zero is returned to prevent division errors. Otherwise, cosine similarity is calculated using the dot product of the two vectors divided by the product of their magnitudes.

The final similarity score is constrained to the range between 0 and 1 to ensure numerical stability and consistency. This value represents how structurally and lexically similar the two source code inputs are.

### **5.3.4 Image Similarity Engine**



The module begins by importing the Image class from the Python Imaging Library (PIL), which is used to load and manipulate image files. The torch library and torch.nn.functional module are imported to support tensor operations and similarity computation. The torchvision.models and torchvision.transforms modules are imported to load pre-trained neural networks and define image preprocessing pipelines.



The variable img\_model is initialized by loading the ResNet50 architecture from the torchvision.models module, with weights pre-trained on the ImageNet dataset. ImageNet contains over one million labeled images across a wide range of categories, enabling the model to learn rich and general-purpose visual representations.

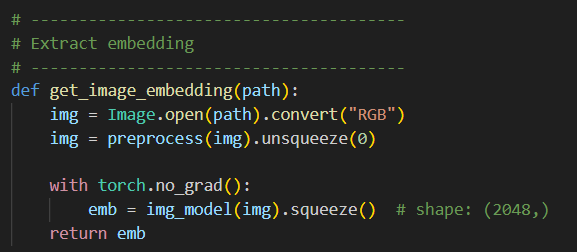
The next line removes the final fully connected classification layer of the ResNet50 model by keeping only the convolutional feature extractor. This is achieved by converting the model’s layers into a list and excluding the last layer. The modified model outputs a 2048-dimensional feature vector instead of class predictions, making it suitable for similarity comparison rather than classification.

The model is then set to evaluation mode using the eval() function. This disables training-specific behaviors such as dropout and batch normalization updates, ensuring consistent and deterministic feature extraction during inference.

### 

The preprocessing pipeline is defined using transforms.Compose, which applies a sequence of transformations to each input image. First, the image is resized to 224×224 pixels, which is the input size expected by ResNet50. Next, the image is converted into a PyTorch tensor, transforming pixel values into numerical form suitable for neural network input.

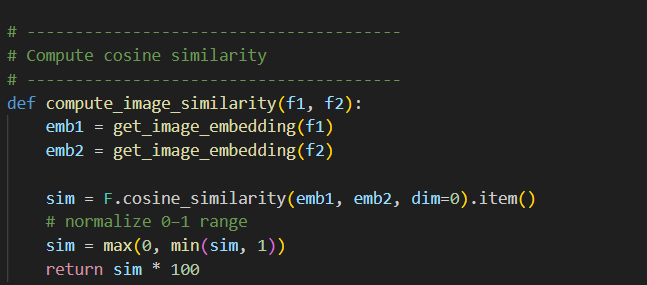
The final step normalizes the image tensor using mean and standard deviation values derived from the ImageNet dataset. This normalization ensures that the input distribution matches the data distribution used during the model’s training phase, which is critical for accurate feature extraction.



The function get\_image\_embedding(path) is responsible for converting an image file into a numerical feature vector. The function begins by loading the image from the specified file path and converting it to RGB format to ensure consistency across different image types.

The image is then passed through the preprocessing pipeline and reshaped using unsqueeze(0) to add a batch dimension, as neural networks expect batched input even for a single image. The model inference is wrapped within a torch.no\_grad() block to disable gradient computation, reducing memory usage and improving performance.

The processed image is fed into the ResNet50 feature extractor, producing a 2048-dimensional embedding vector. The output is squeezed to remove unnecessary dimensions and returned as a tensor. This embedding represents high-level semantic features of the image.



The function compute\_image\_similarity(f1, f2) computes the similarity between two image files. It begins by generating embeddings for both images using the get\_image\_embedding function. These embeddings capture the visual characteristics of the images in a numerical form.

Cosine similarity is then computed between the two embedding vectors using F.cosine\_similarity. This metric measures the angular similarity between vectors and is widely used for comparing high-dimensional feature representations. The result is a scalar value ranging from –1 to 1.

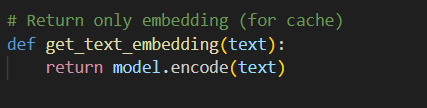
To ensure stability and interpretability, the similarity score is clamped to the range between 0 and 1. The score is then multiplied by 100 to produce a percentage-based similarity value, which is more intuitive for end users.

### **5.3.5 Document Similarity**



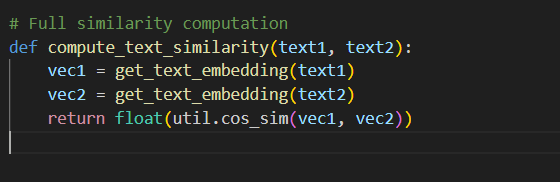
The module begins by importing SentenceTransformer and util from the sentence\_transformers library. The SentenceTransformer class provides access to pre-trained transformer models optimized for producing sentence-level embeddings, while the util module contains helper functions such as cosine similarity for comparing embedding vectors.  


The variable model is initialized using the SentenceTransformer class with the model name "all-MiniLM-L6-v2". This model is a lightweight yet powerful transformer architecture trained on large-scale sentence similarity datasets. It converts textual input into fixed-length dense vectors that preserve semantic meaning. Loading the model once at the module level ensures that it is reused across requests, improving system efficiency and reducing redundant computation.



The function get\_text\_embedding(text) is designed to convert raw text into a numerical embedding vector. The function takes a string as input and passes it to the transformer model using the encode method. Internally, the model tokenizes the text, processes it through multiple transformer layers, and produces a dense vector representation. This vector captures the semantic relationships between words and sentences, allowing texts with similar meanings to have similar embeddings even if they use different vocabulary.

This function returns only the embedding vector and does not perform similarity calculation. Separating embedding generation from similarity computation allows embeddings to be cached or reused, which improves performance when the same text is compared multiple times.



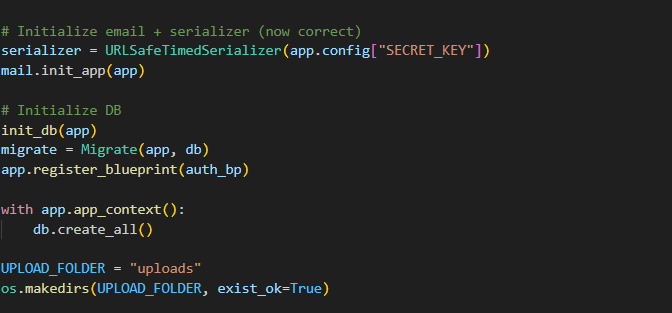
The function compute\_text\_similarity(text1, text2) computes the semantic similarity between two text inputs. It first generates embeddings for both input texts by calling the get\_text\_embedding function. The resulting vectors represent the semantic meaning of each text in a high-dimensional space.

The cosine similarity between the two vectors is then calculated using util.cos\_sim. Cosine similarity measures the angular distance between vectors, producing values between –1 and 1, where higher values indicate greater semantic similarity. The result is converted to a Python float to ensure compatibility with downstream processing, database storage, and JSON serialization.

### **5.3.6 System Integration and Application Controller Module**

## 

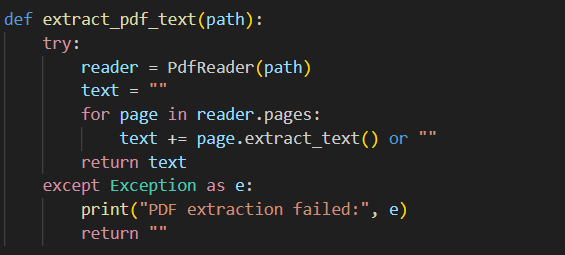
The first section imports all required libraries and internal modules. Standard Python libraries such as os, hashlib, pickle, datetime, traceback, and io are used for file handling, hashing, serialization, timestamping, and error reporting. Third-party libraries such as NumPy and PyTorch are imported to support numerical computation and cosine similarity operations. Specialized libraries including PyPDF2, python-docx, and BeautifulSoup are used for document and web content extraction. Flask-related modules handle routing, sessions, JSON responses, and template rendering. Internal system modules such as DB, auth, and the similarity utilities are imported to enable database access, authentication, and multimodal similarity analysis.



The Flask application object is created and configured with a secret key for session security. Email server settings are defined to enable automated email delivery for password reset and verification features. A URLSafeTimedSerializer is initialized to securely generate and validate time-limited tokens used in email-based authentication workflows.

The database is initialized using the init\_db function, database migrations are enabled through Flask-Migrate, and the authentication blueprint is registered. The database tables are created within the application context to ensure that all models are properly initialized before runtime.

An upload directory is defined and created if it does not already exist. This directory is used to store user-uploaded files for similarity analysis. This approach ensures controlled file management and prevents runtime errors caused by missing directories.



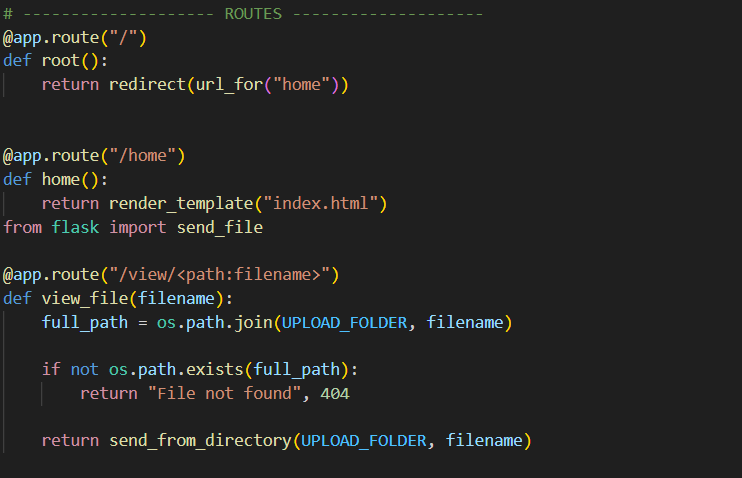
The extract\_pdf\_text function reads the content of a PDF file using PdfReader. It iterates through each page and extracts text while handling potential extraction failures gracefully. If an error occurs, an empty string is returned, ensuring that the system remains stable even when malformed documents are uploaded.



The hash\_file function computes a SHA-256 hash of a file’s binary content. This hash uniquely represents the file and is used as a cache key for previously computed embeddings. This mechanism avoids redundant computation and improves system performance.

The get\_or\_cache\_embedding function retrieves embeddings from the database cache if they already exist. If no cached entry is found, the function computes a new embedding using the provided embedding function, serializes it using pickle, and stores it in the database. This design significantly reduces processing time for repeated similarity comparisons involving identical files.

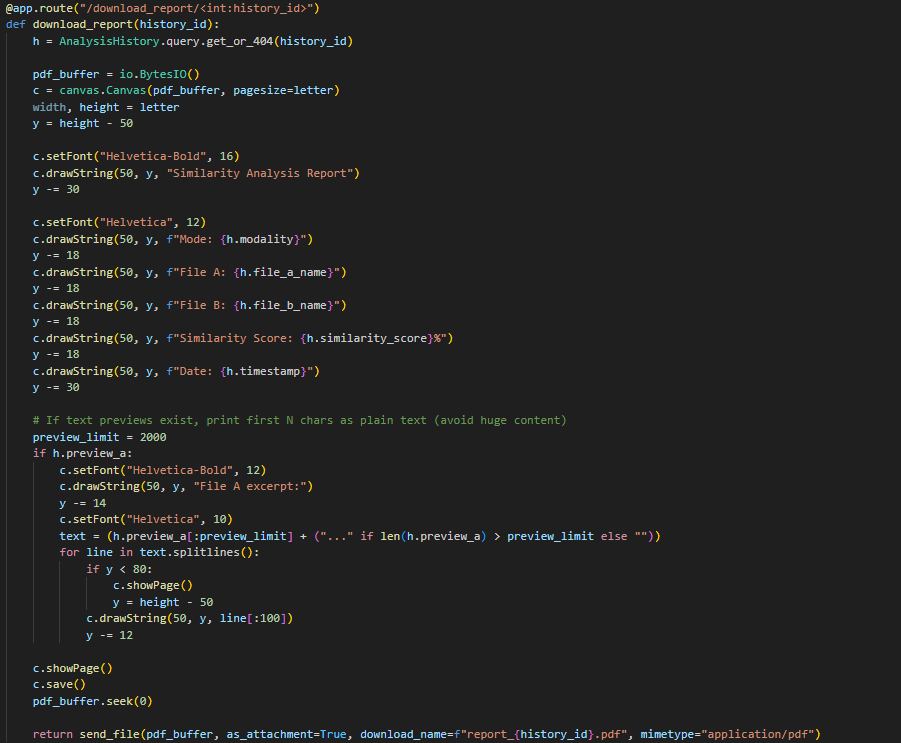
The generate\_diff\_html function produces a side-by-side HTML representation of textual differences using Python’s difflib. This visual comparison aids users in understanding where similarities and differences occur within documents or code files.



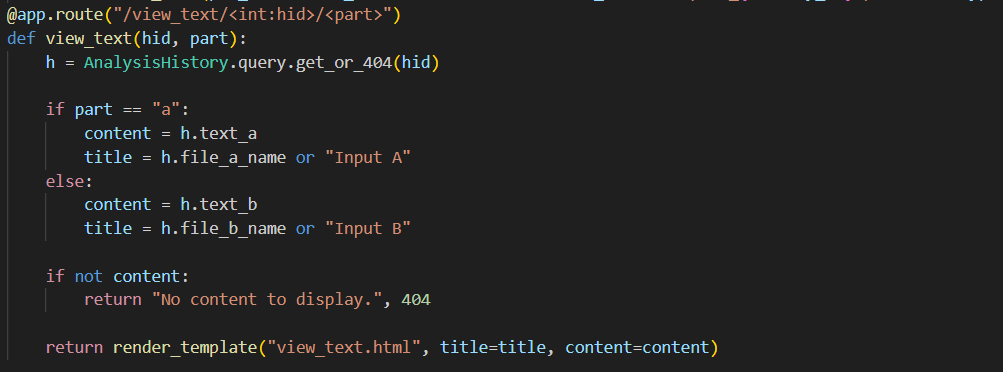
The root route redirects users to the home page, while the home route renders the main landing page. The file viewing route securely serves uploaded files from the upload directory, ensuring that only existing files are accessed.



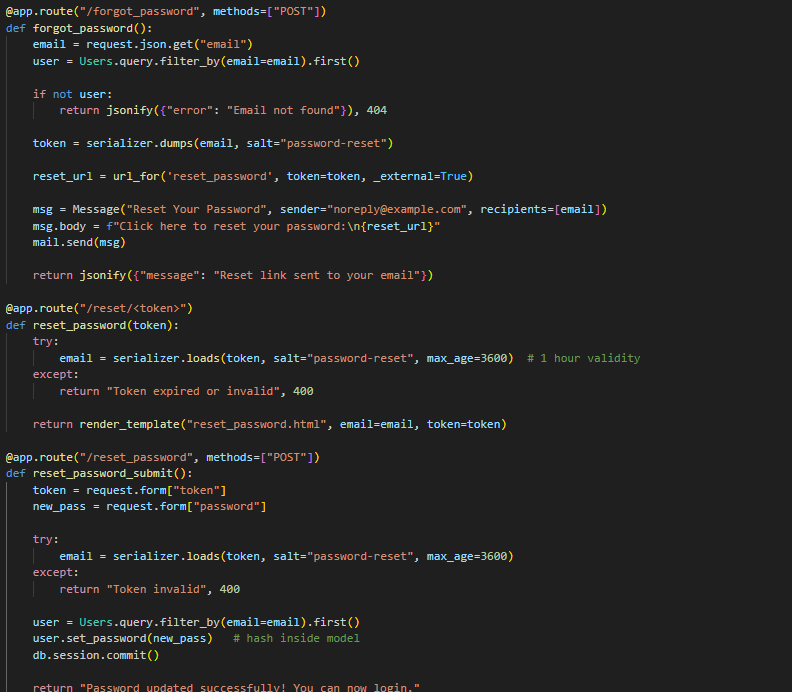
The history route retrieves all similarity analysis records associated with the logged-in user and displays them in descending chronological order. The profile route retrieves and displays user account information. These routes enforce session-based authentication to prevent unauthorized access.



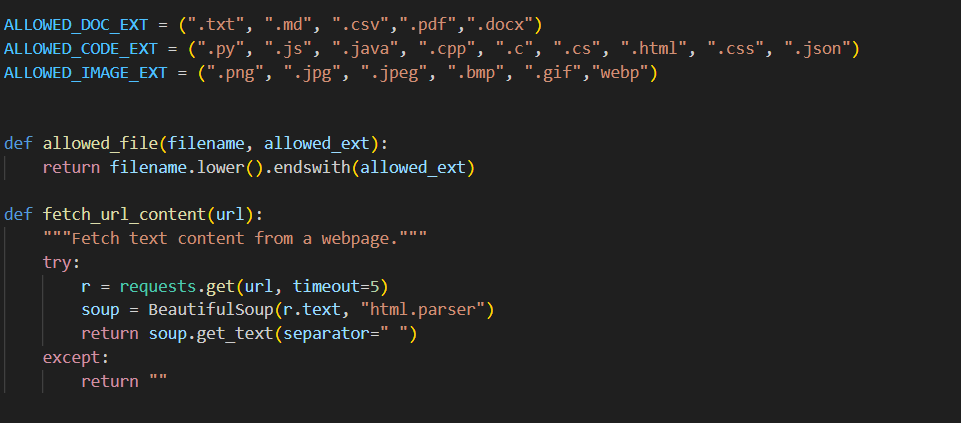
The report route renders a detailed similarity report using stored database records. The report download route dynamically generates a PDF file using the ReportLab library. It formats analysis details such as modality, filenames, similarity score, and timestamps, ensuring that users can download official, printable reports.



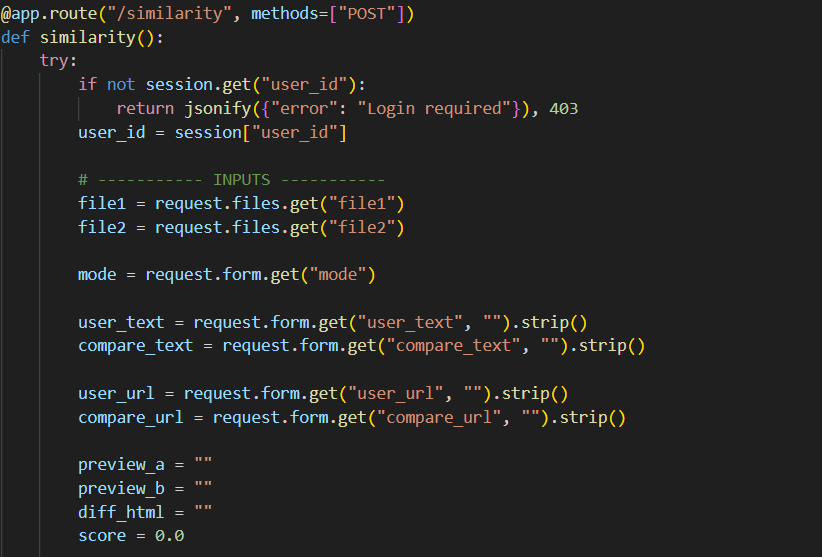
The text viewing route allows users to view full textual content stored during similarity analysis. It dynamically selects content based on whether the user requests the first or second input, improving transparency and usability.



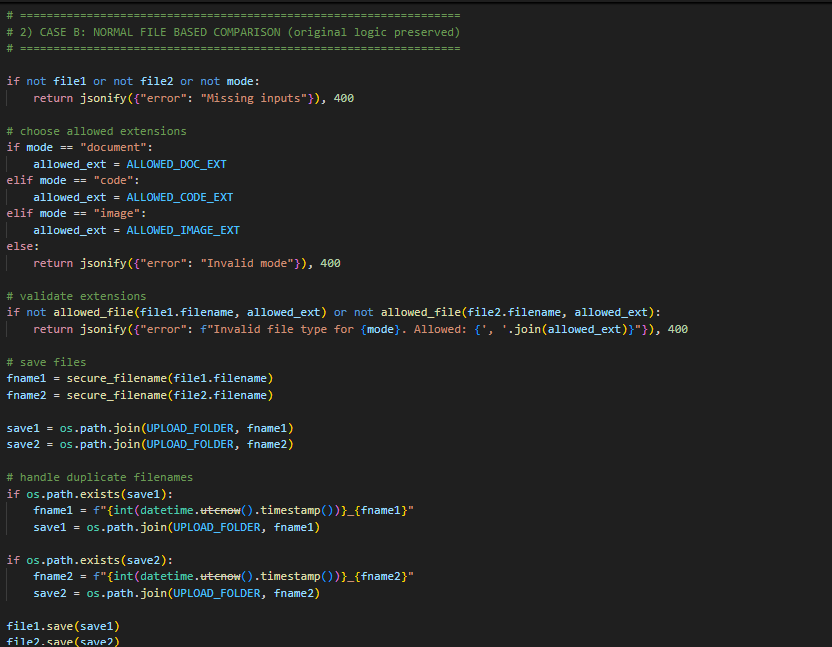
The password reset functionality includes routes for requesting a reset, validating reset tokens, rendering reset forms, and updating user passwords. Secure, time-limited tokens ensure that password reset operations are protected against misuse.



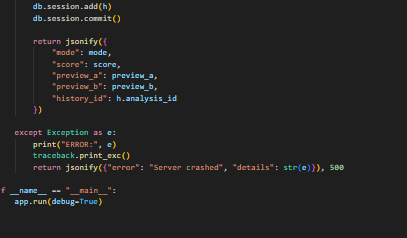
The system defines allowed file extensions for documents, code files, and images to prevent invalid uploads. The URL content extraction function retrieves and cleans webpage text using HTTP requests and HTML parsing, enabling URL-based similarity analysis.











The /similarity route is the central processing function of the system. It begins by validating user authentication and input completeness. The route supports two major use cases: text or URL-based comparison and file-based comparison.

For text and URL inputs, content is directly embedded using a transformer-based language model, and cosine similarity is computed. Differences are visualized using HTML diff generation.

For file-based comparison, the route dynamically selects the appropriate processing logic based on the chosen modality. Document files are parsed into text and compared using sentence embeddings. Code files are analyzed using syntax-aware and token-based similarity methods. Image files are processed using deep learning embeddings extracted from a pre-trained convolutional neural network.

All similarity results are normalized, stored in the database, and returned to the client as structured JSON responses.

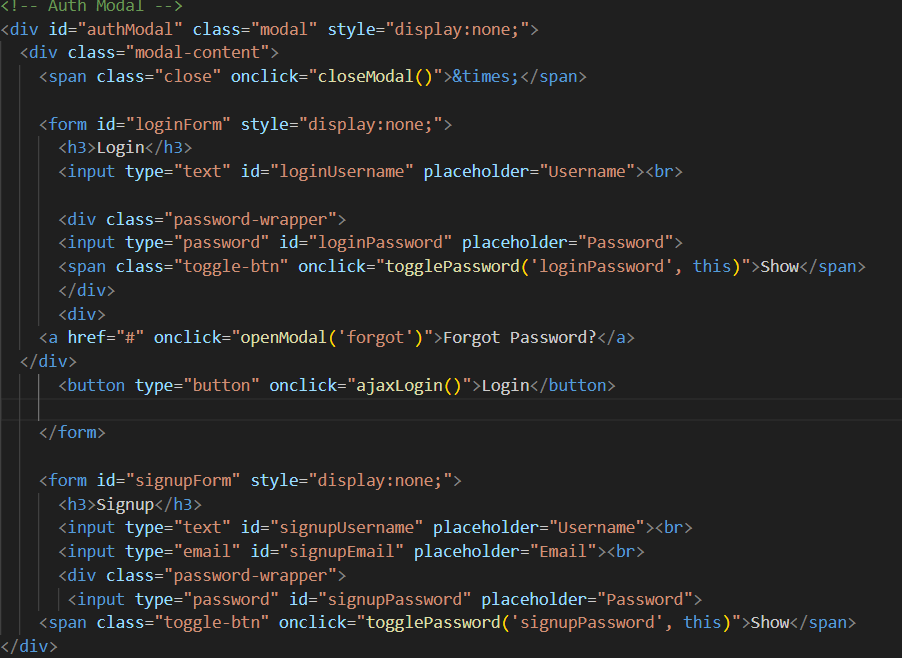
All similarity processing is wrapped in a try-except block to capture unexpected errors and prevent application crashes. When executed directly, the Flask application runs in debug mode to support development and testing.

## **5.4 Frontend Development**

### **5.4.1 User Interface Components**

#### **5.4.1.1 header**







This HTML file defines the **base layout (template)** for the Similarity System web application. It uses **Flask’s Jinja2 templating engine**, which allows dynamic content rendering based on the user’s login state and page context. The file mainly controls the **navigation bar, authentication modal (login, signup, password reset), and page structure**, while allowing other pages to inject their own content.

At the top of the document, the standard HTML5 structure is declared, including the document type, language setting, and character encoding. The <title> element is dynamically generated using Jinja2 syntax. If a page-specific title is provided, it is displayed; otherwise, a default title, “Similarity System,” is used. The stylesheet home.css is loaded from the Flask static directory, ensuring consistent styling across all pages. An empty <script> tag is included as a placeholder, allowing additional JavaScript to be injected if required.

The navigation bar is divided into two sections. On the left side, navigation links are provided for “Home” and “History,” both generated using Flask’s url\_for function to ensure correct routing. On the right side, the content changes dynamically depending on whether the user is logged in. This is determined by checking the existence of user\_id in the session. If the user is authenticated, their username is displayed as a clickable link to the profile page, along with a logout option. If the user is not logged in, two buttons—Login and Signup—are shown instead. These buttons trigger JavaScript functions that open the authentication modal.

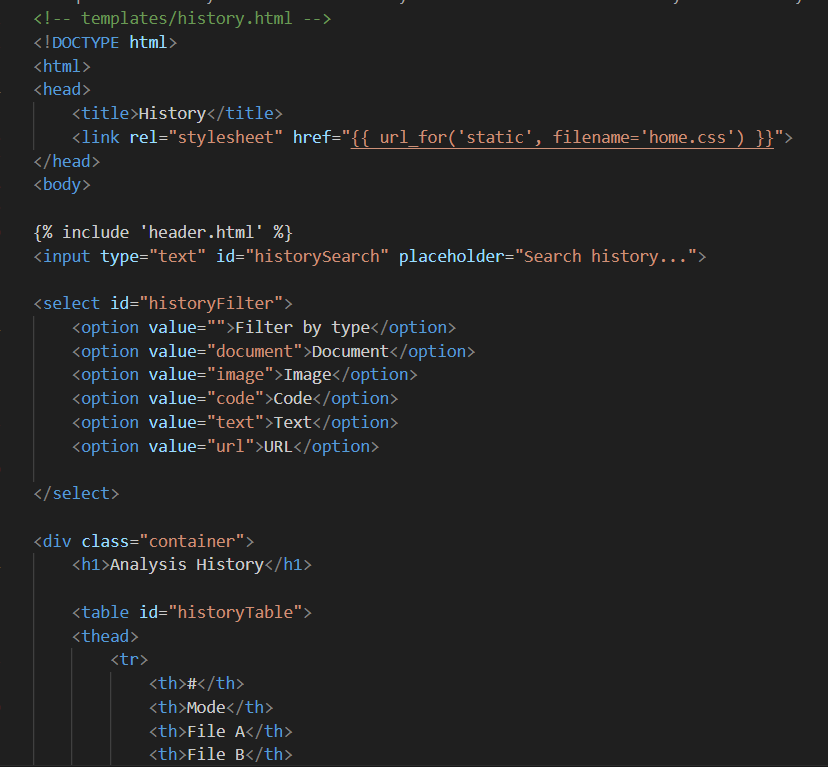
The authentication modal is a hidden overlay component designed to handle user authentication without navigating away from the current page. It contains three separate forms: login, signup, and password reset. Only one form is visible at a time, depending on user interaction. The modal includes a close button that allows users to dismiss it easily.

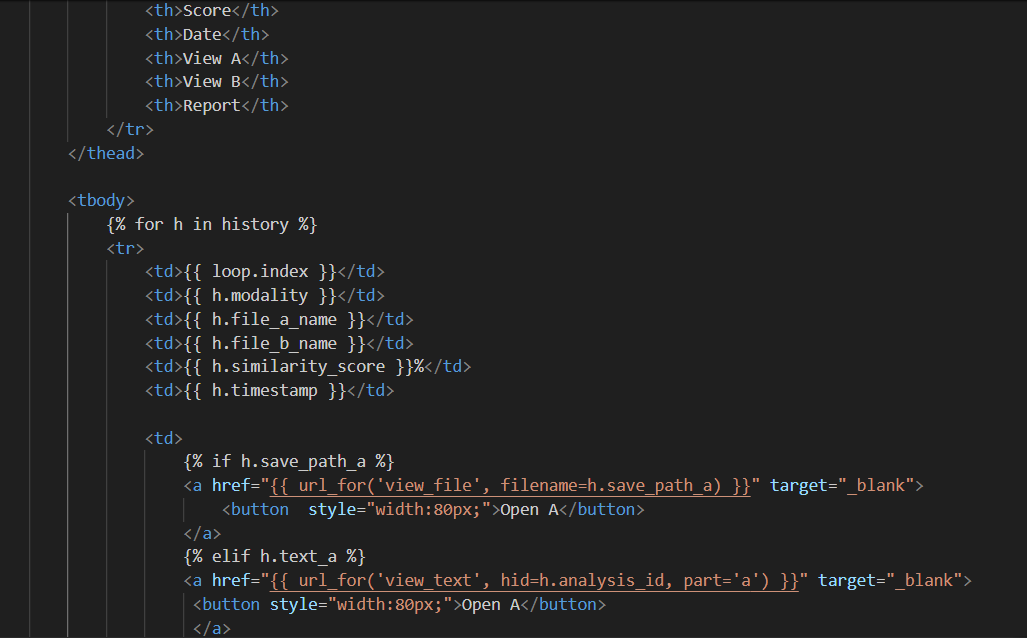
The login form collects the username and password, with a “show password” toggle that improves usability. A “Forgot Password?” link switches the modal to the password reset form. The login button triggers an AJAX-based login process, allowing authentication without a full page reload. Similarly, the signup form collects a username, email, and password, also providing a password visibility toggle. The signup action is handled via JavaScript, enabling real-time feedback and smoother user experience. The password reset form allows users to request a reset link by entering their email address.

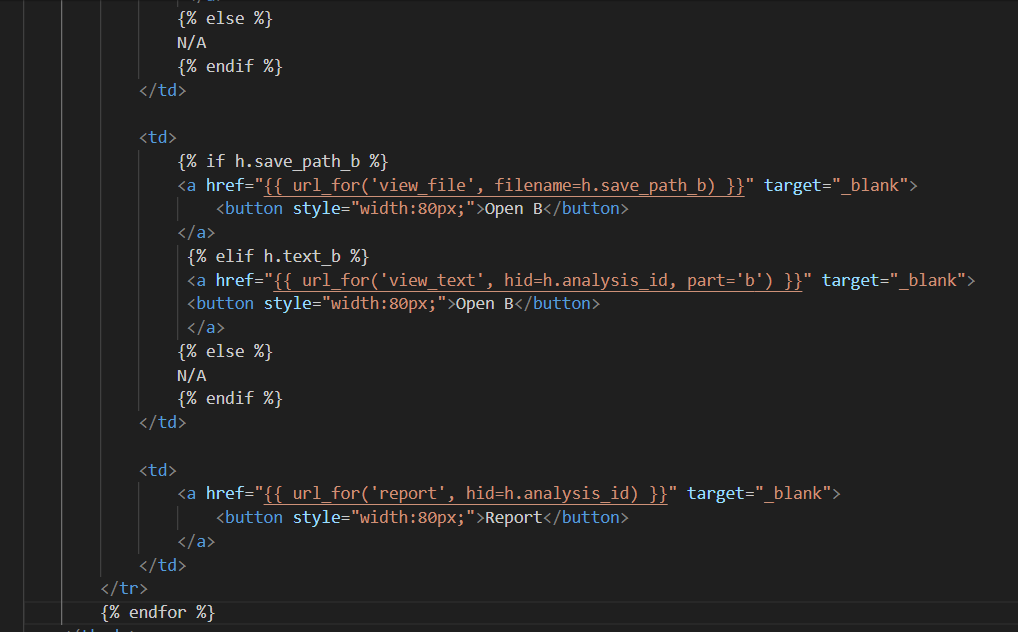
Below the forms, an empty message container is included to display authentication-related feedback such as error or success messages. This ensures users receive immediate visual confirmation of their actions.

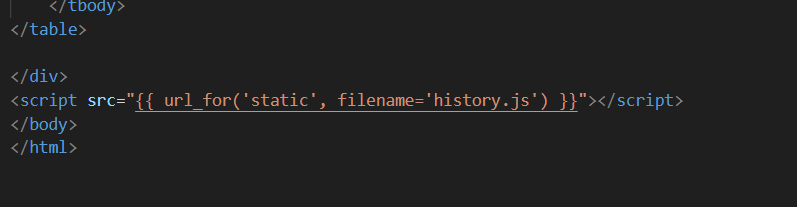
Finally, the main content area is defined using a Jinja2 block. This allows other HTML templates to extend this base layout and insert page-specific content while keeping the navigation bar and authentication modal consistent throughout the system. Overall, this template serves as the structural foundation of the web application, integrating user authentication, navigation, and dynamic content rendering in a clean and modular way.

#### **5.4.1.2 history**









This **history.html** file defines the History page of the similarity system and is used to display a complete record of all similarity analyses previously performed by a logged-in user. The page begins with standard HTML structure and loads a shared CSS file to maintain consistent styling across the application. The {% include 'header.html' %} statement inserts the common navigation bar and authentication controls, ensuring that users can easily navigate between pages and manage their accounts without duplicating code.

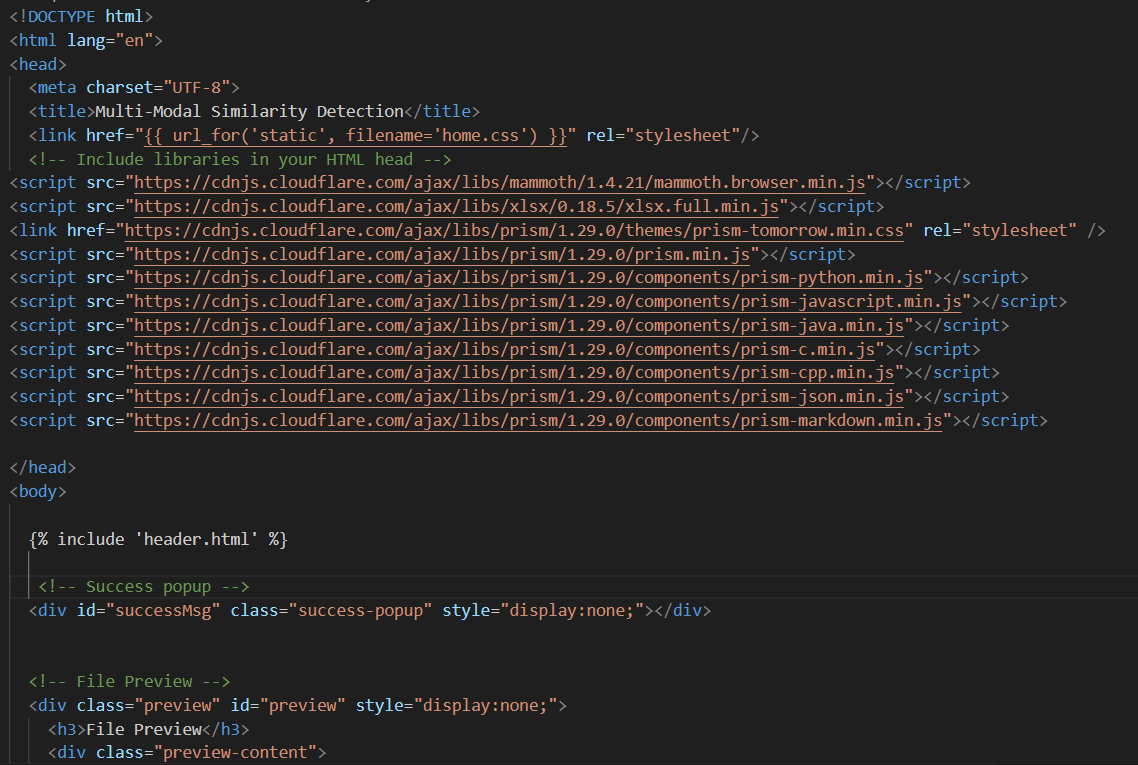
Below the header, the page provides two interactive controls: a text input field for searching analysis history and a dropdown menu for filtering results by comparison type, such as document, image, code, text, or URL. These inputs allow users to quickly locate specific analysis records based on keywords or modality, improving usability when the history list becomes large.

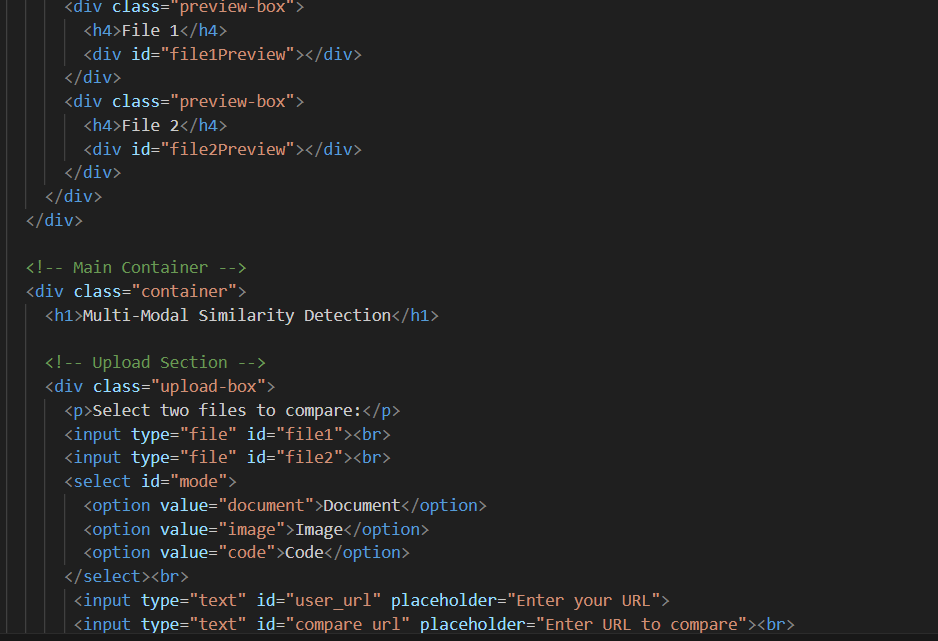
The main content of the page is a table that displays the analysis history in a structured format. Each column in the table represents key information about a similarity comparison, including the comparison mode, file names, similarity score, and timestamp. The table rows are dynamically generated using a Jinja2 loop that iterates over the history data passed from the Flask backend. This ensures that the table automatically reflects the user’s stored analysis records in the database.

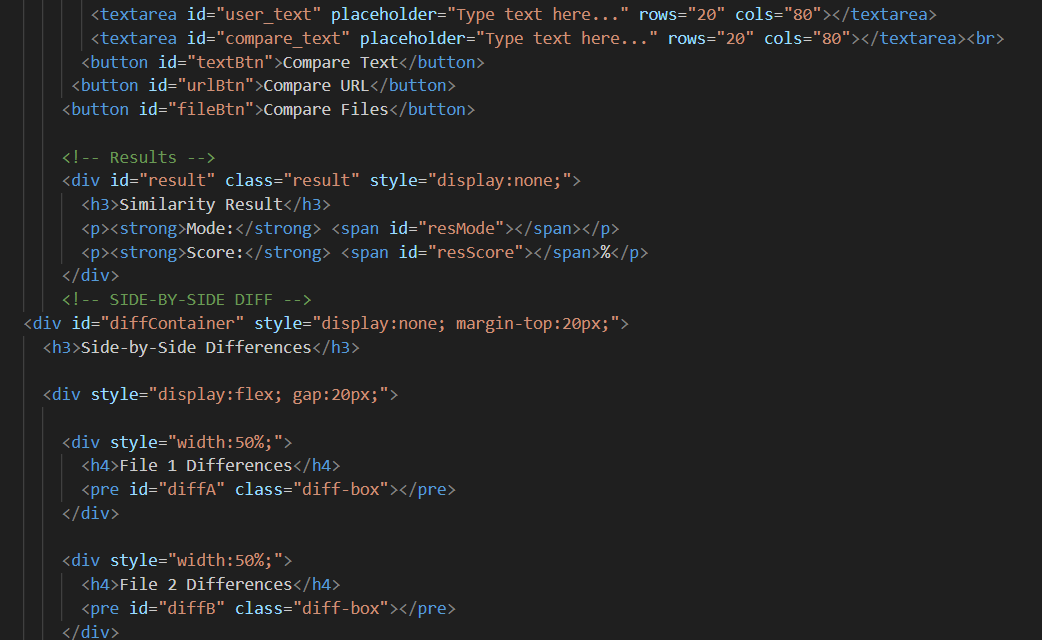
For each analysis entry, conditional logic is used to determine how the original inputs can be viewed. If a file was uploaded, an “Open A” or “Open B” button links to the file viewer route. If the comparison was text-based or URL-based, the buttons instead open a dedicated text viewer page. When no preview is available, the system clearly indicates this by displaying “N/A.” Additionally, each row includes a “Report” button that opens a detailed similarity report for that specific analysis, allowing users to review results in greater depth.

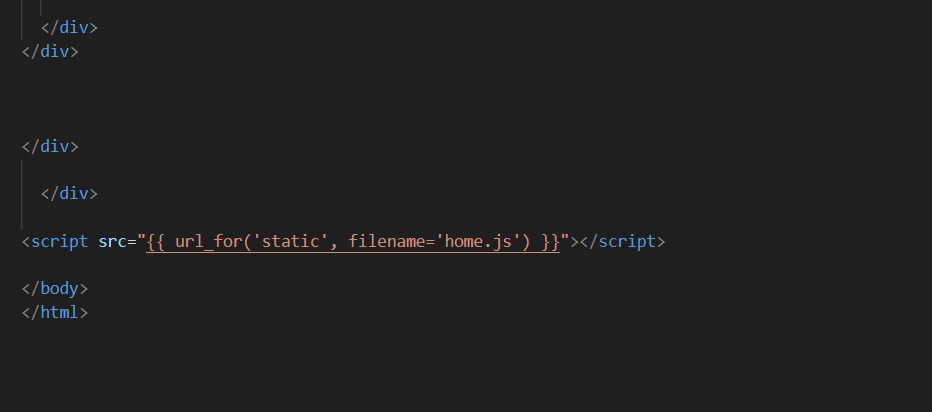
At the bottom of the page, a JavaScript file is loaded to enhance interactivity. This script handles client-side features such as live searching, filtering, and pagination of the history table without requiring page reloads. Overall, this History page functions as a centralized dashboard where users can review, manage, and revisit all their past similarity analyses in a clear, efficient, and user-friendly manner.

#### **5.4.1.3 index**









This HTML file defines the **main user interface of the Multi-Modal Similarity Detection system**, where users can upload content and perform similarity comparisons across different data types. The document begins with standard HTML structure and metadata, ensuring proper character encoding and page title. A shared CSS file is loaded to maintain a consistent layout and visual design throughout the system. Several external JavaScript and CSS libraries are included in the header, such as Mammoth for Word document previewing, XLSX for spreadsheet handling, and Prism for syntax highlighting of code files. These libraries enhance client-side functionality and improve user experience by enabling rich previews and readable code displays.

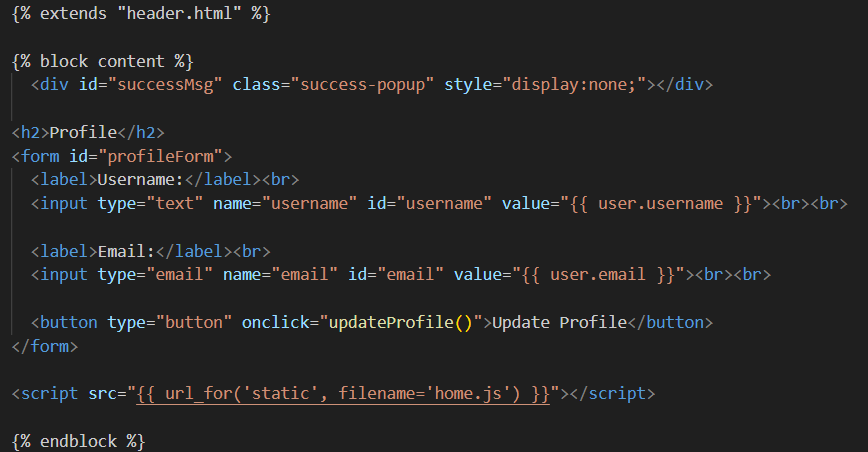
At the top of the body, the page includes a reusable header template that contains navigation links and authentication controls. This is followed by a hidden success message popup, which is dynamically displayed by JavaScript to provide feedback for actions such as login, signup, or profile updates. The page also defines a file preview section that remains hidden until files are selected. Once activated, this section displays two preview panels side-by-side, allowing users to visually inspect the contents of the files they intend to compare before running the similarity analysis.

The main container of the page presents the core functionality of the system. It provides input elements for multiple comparison modes, including file upload inputs for document, image, or code comparison, a dropdown selector to choose the comparison modality, text fields for URL-based comparison, and large text areas for direct text input. This flexible design allows users to compare content from different sources without being restricted to a single input method. Dedicated buttons are provided for each comparison type, ensuring clarity in user interaction and minimizing input ambiguity.

Once a comparison is performed, the results section becomes visible and displays the comparison mode and the computed similarity score as a percentage. In addition to the numerical score, the page includes a side-by-side difference viewer that highlights similarities and differences between the two inputs. This diff view is especially useful for text and code comparisons, as it allows users to visually identify matching and differing content line by line.

Finally, a JavaScript file is loaded at the bottom of the page to handle all client-side logic, such as file previews, event handling, form submission, result rendering, and difference highlighting. Overall, this page serves as the central interaction hub of the system, integrating multiple comparison modalities into a single, user-friendly interface that supports rich previews, clear feedback, and detailed similarity analysis.

#### **5.3.1.4 profile**



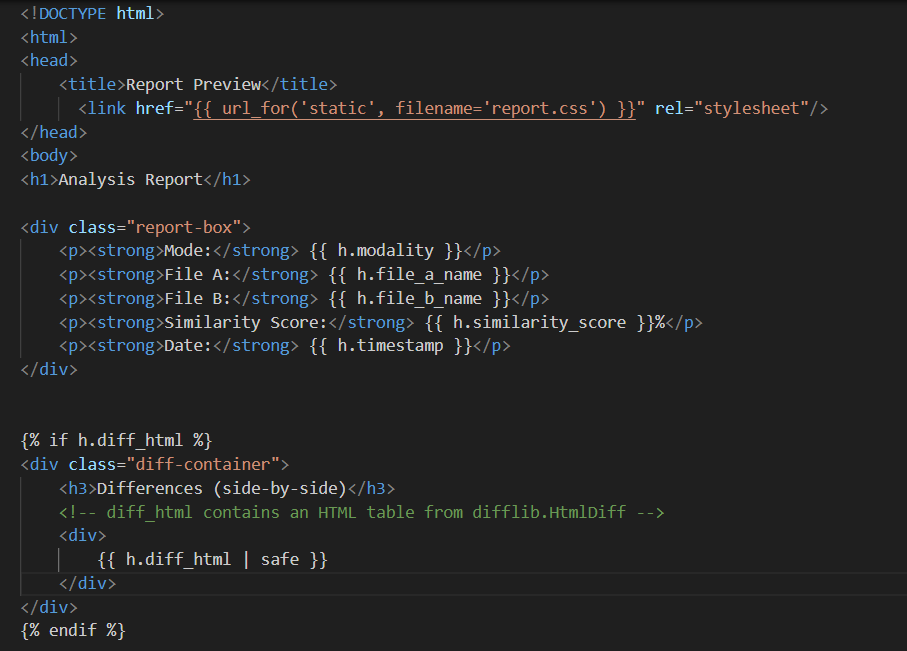
This template defines the **user profile page** of the system and is built using Jinja2’s template inheritance mechanism. The page extends the header.html template, which means it automatically includes the navigation bar and authentication controls shared across the application. This approach ensures consistency in layout and reduces code duplication. All profile-related content is placed inside the content block, allowing it to be injected into the main layout defined by the parent template.

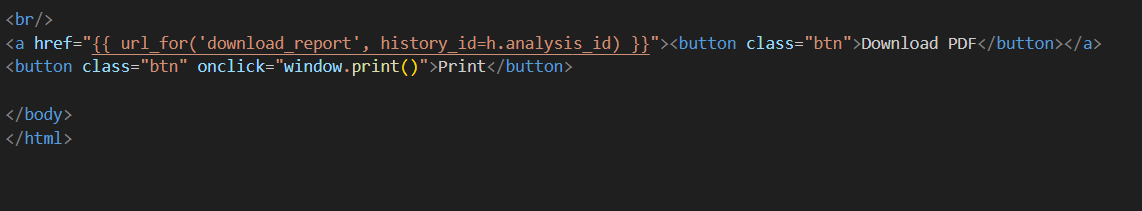
At the top of the content block, a hidden success message popup is included. This element is dynamically displayed by JavaScript to provide user feedback after a successful profile update, such as confirming that changes have been saved. The main body of the page contains a simple profile form that allows users to view and update their personal information. The form displays the current username and email address, which are pre-filled using server-side data passed from Flask through the user object. This ensures that the user always sees their most recent account information when accessing the profile page.

The form does not submit data using a traditional HTTP POST request. Instead, the update action is handled by a button that triggers the updateProfile() JavaScript function. This function sends the updated username and email to the backend asynchronously using AJAX, providing a smoother user experience without requiring a full page reload. This design improves responsiveness and allows immediate feedback to be shown to the user through the success popup.

At the bottom of the template, the main JavaScript file is loaded to enable client-side functionality such as handling form submissions, displaying success messages, and updating the user interface dynamically. Overall, this profile page provides a clean and efficient interface for account management, combining server-side rendering for data display with client-side scripting for interactive updates.

#### **5.3.1.5 report**





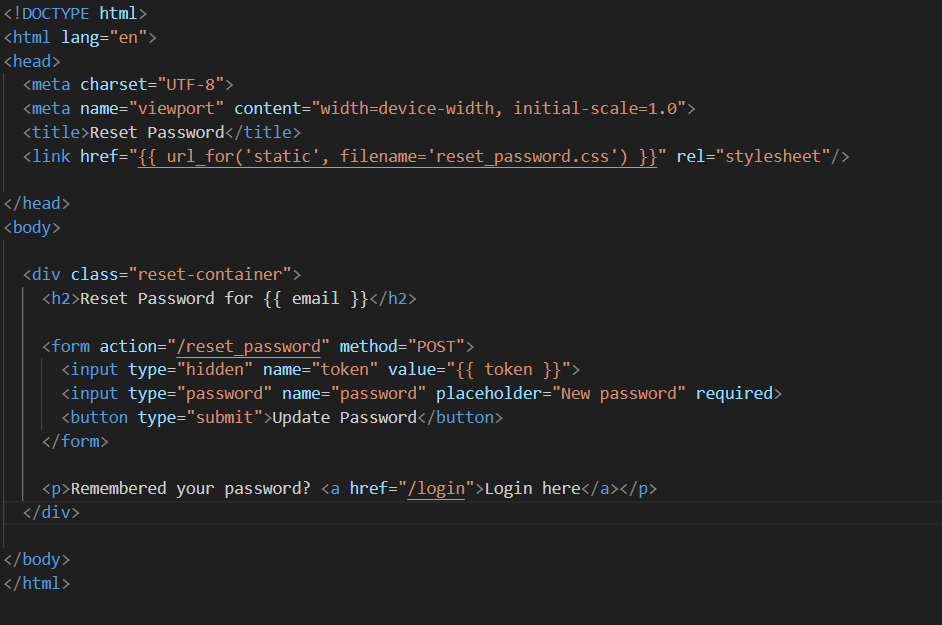
This template represents the **analysis report preview page**, which displays the detailed results of a similarity comparison performed by the system. It is a standalone HTML document styled using an external CSS file (report.css) to ensure the report has a clean and professional layout suitable for viewing, printing, or exporting. The page title and main heading clearly indicate that the content shown is an analysis report, helping users immediately understand the purpose of the page.

At the top of the page, a report summary section presents the key metadata of the similarity analysis. This includes the comparison mode (such as document, image, code, text, or URL), the names of the two compared files or inputs, the calculated similarity score expressed as a percentage, and the date and time when the analysis was performed. These values are dynamically injected using Jinja2 variables from the AnalysisHistory object (h) passed by the Flask backend, ensuring that each report accurately reflects a specific comparison session.

Below the summary, the template conditionally displays a side-by-side difference view if difference data is available. The diff\_html field contains pre-generated HTML produced by Python’s difflib.HtmlDiff module, which visually highlights similarities and differences between the two inputs line by line. The safe filter is applied to allow this HTML content to be rendered correctly in the browser without escaping, enabling formatted tables and highlighted changes to appear as intended.

At the bottom of the page, user action buttons are provided to enhance usability. One button allows the user to download the report as a PDF file, which is generated dynamically by the backend using the stored analysis data. Another button triggers the browser’s print function, allowing the report to be printed directly. Together, these features make the report page suitable not only for on-screen review but also for documentation, submission, or archival purposes.

#### **5.3.1.6 reset password**



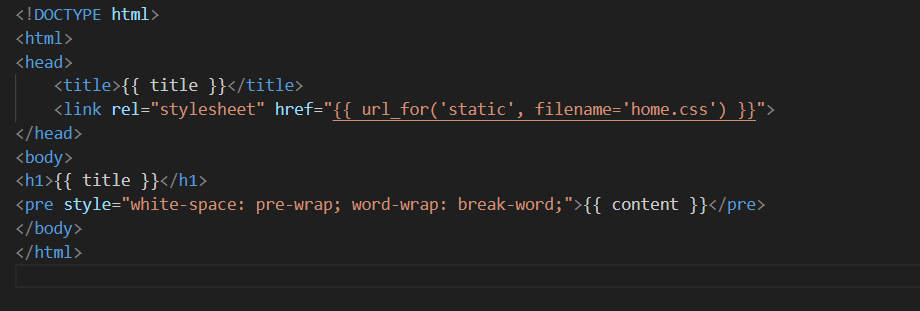
This HTML page implements the **password reset interface** for the system and is displayed after a user clicks a valid password-reset link sent to their email. The document begins with standard HTML metadata, including character encoding and viewport settings, to ensure proper rendering across different browsers and devices. An external CSS file (reset\_password.css) is linked to provide consistent styling and to visually separate this page from other parts of the application.

The main content is wrapped inside a container element that centers and structures the reset form on the page. At the top of this container, a heading clearly informs the user that they are resetting the password for a specific email address. The email value is dynamically inserted using a Jinja2 template variable, which helps reassure the user that the reset operation applies to the correct account and reduces the risk of confusion or phishing.

The core of the page is the password reset form itself. This form submits a POST request to the /reset\_password route, which is handled by the Flask backend. A hidden input field carries the secure reset token that was generated earlier and embedded in the reset link sent via email. This token allows the backend to verify that the request is legitimate and has not expired. The visible password input field requires the user to enter a new password, and the required attribute ensures that the form cannot be submitted with an empty value.

Once the user submits the form, the backend validates the token, hashes the new password, and updates the user’s account securely in the database. Below the form, a small helper message provides a navigation link back to the login page, offering a convenient path for users who remember their password or want to sign in immediately after completing the reset. Overall, this page plays a critical role in account recovery by combining security, clarity, and user-friendly design.

#### **5.3.1.7 view text**



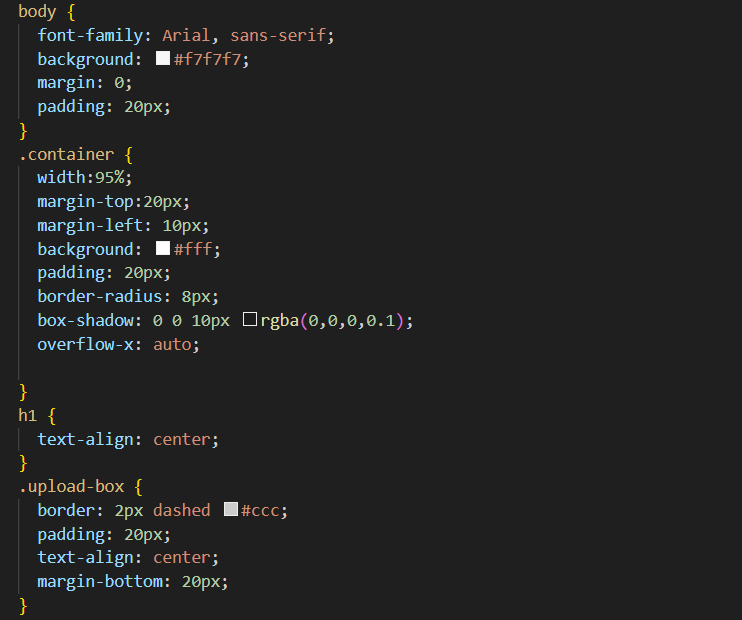
This HTML template is used to **display the full text content of a file or input in a readable format**, such as when a user chooses to view one side of a similarity comparison in detail. The document begins with a standard HTML structure and dynamically sets the page title using a Jinja2 variable. This allows the same template to be reused for different files or text inputs while clearly indicating what content is being viewed.

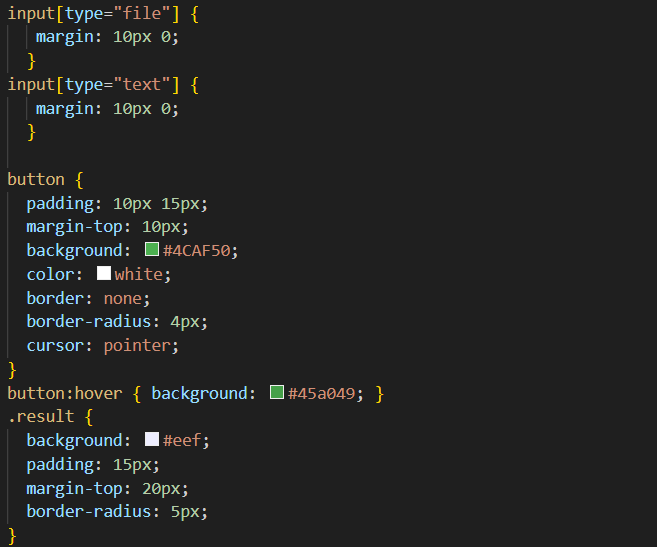
The stylesheet home.css is linked to maintain a consistent visual appearance with the rest of the system, including fonts, spacing, and color themes. At the top of the page body, a heading displays the title of the content, which is typically the filename or a descriptive label such as “Input A” or “Input B.” This helps users easily identify what they are viewing.

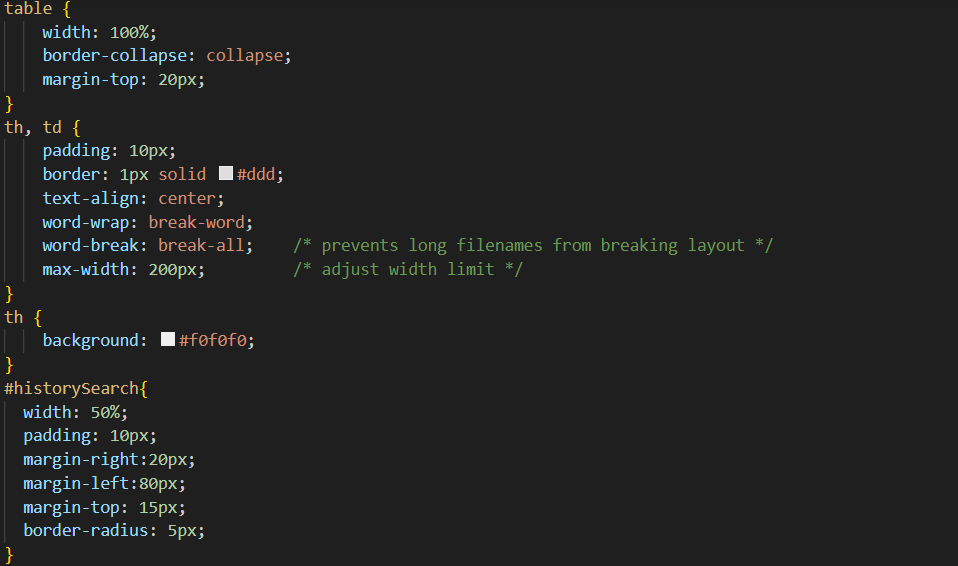
The main content is rendered inside a <pre> element, which preserves the original formatting, line breaks, and spacing of the text. The inline CSS styles white-space: pre-wrap and word-wrap: break-word ensure that long lines of text automatically wrap within the page width instead of overflowing horizontally. This is especially important for displaying large documents, code files, or extracted text from PDFs in a user-friendly way.

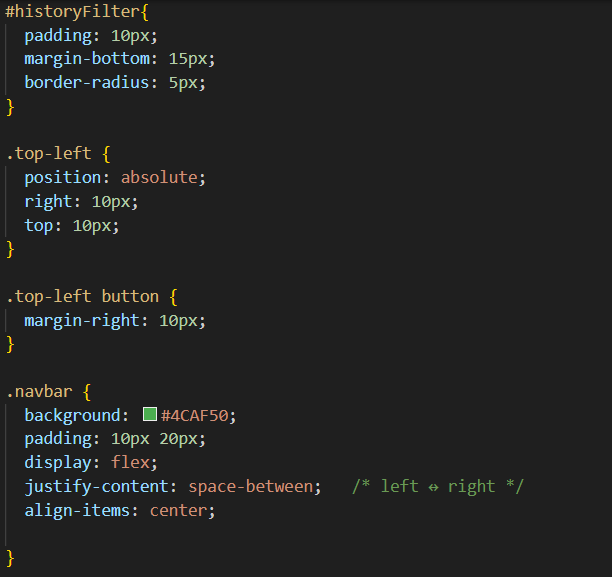
### **5.4.2 CSS**

#### **5.4.2.1 home**



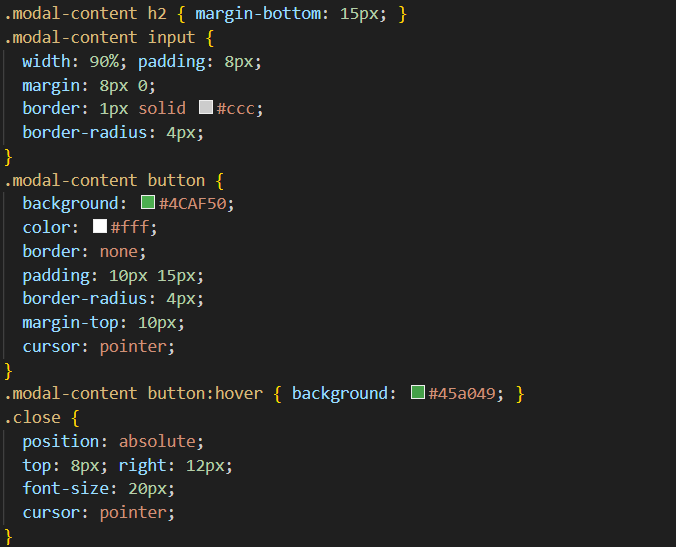






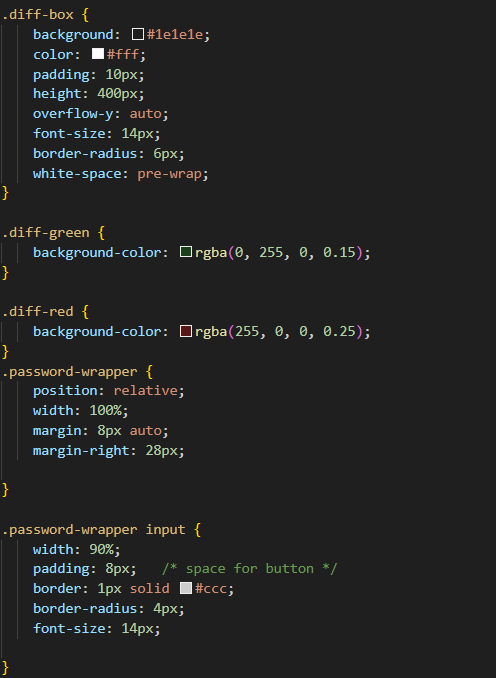




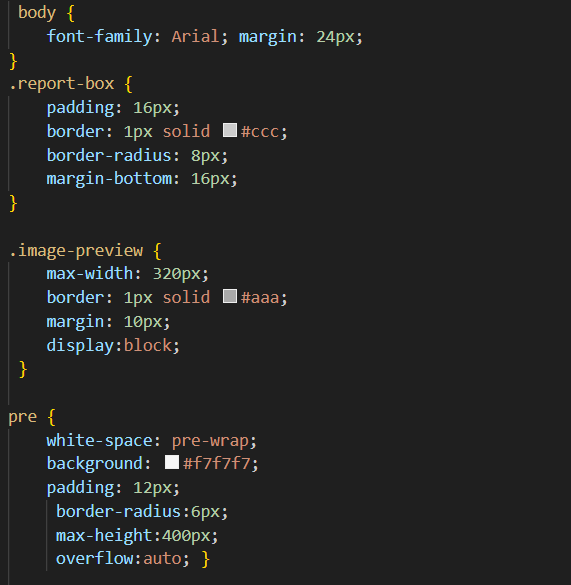


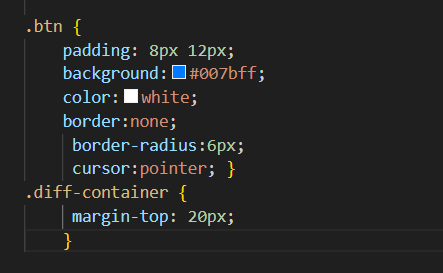




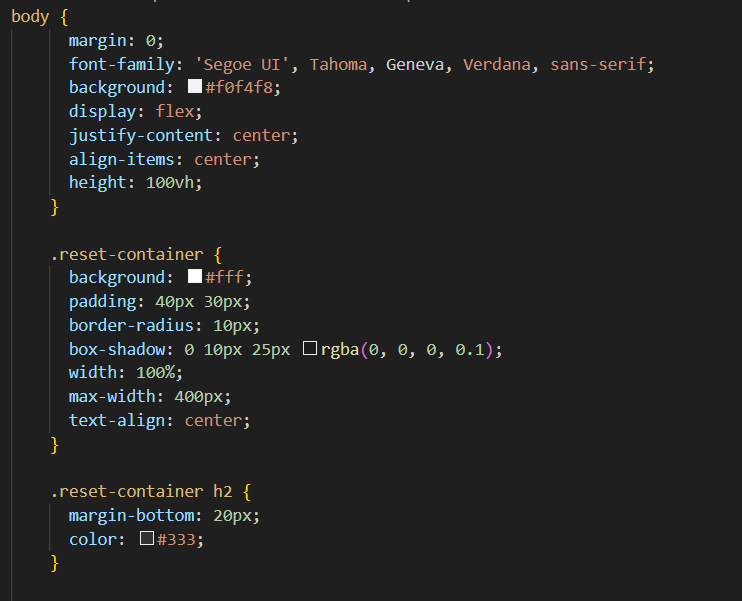


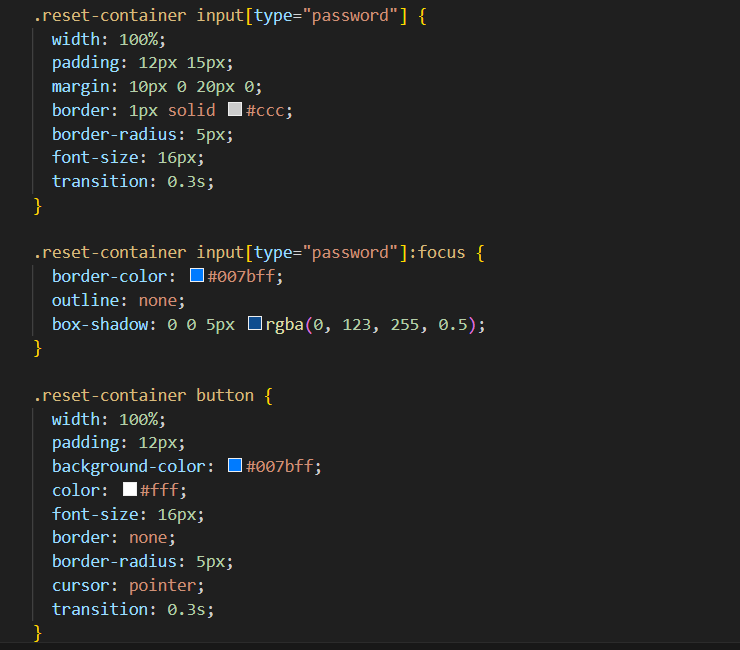
#### 5.4.2.2 report

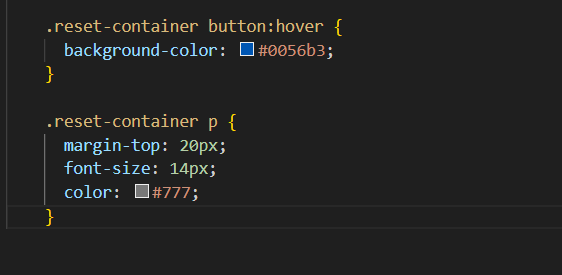




#### **5.4.2.3 reset password**

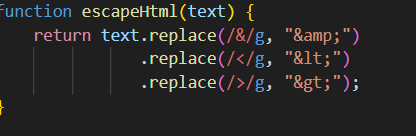






### **5.4.3 JavaScript Logic**

#### **5.4.3.1 home.js**



This function sanitizes user-provided text by replacing special characters such as <, >, and & with their corresponding HTML entities. This prevents cross-site scripting (XSS) vulnerabilities when rendering user content dynamically on the page. It is primarily used when previewing text and code files.



This function handles text-based similarity comparison. It retrieves text input values from the user interface, packages them into a FormData object, and submits them to the backend /similarity endpoint using the Fetch API. Upon receiving the response, it displays the similarity score and triggers a visual difference comparison if previews are available.

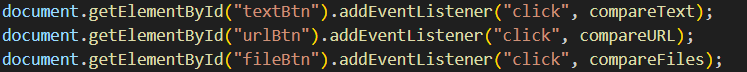


This function enables similarity comparison between two web URLs. It collects URL inputs from the user, sends them to the backend for content extraction and comparison, and displays the returned similarity results. This allows users to compare online articles or web pages directly without downloading content.

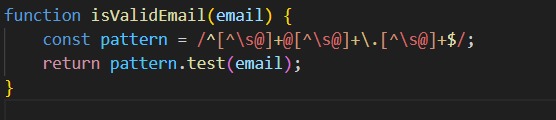


This function manages file upload-based similarity analysis. It retrieves selected files and the chosen comparison mode (document, code, or image), sends them to the backend, and processes the returned similarity score and previews. This function supports multiple file types and delegates actual similarity computation to the backend.

This function updates the user interface with the similarity analysis results returned by the server. It displays the comparison mode, similarity score, and optionally triggers line-by-line difference highlighting. It also handles error messages gracefully.



This section binds user interface buttons to their corresponding comparison functions, ensuring that user actions such as clicking “Compare Text,” “Compare URL,” or “Compare File” correctly trigger backend requests.



This function validates email addresses using a regular expression. It ensures that users provide properly formatted email addresses during signup or password recovery processes.

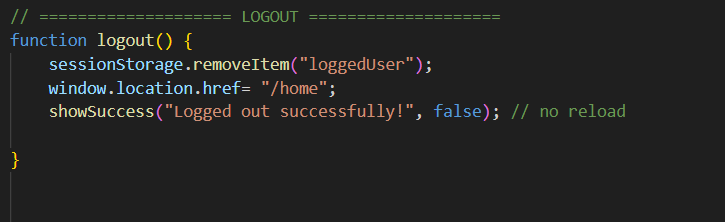


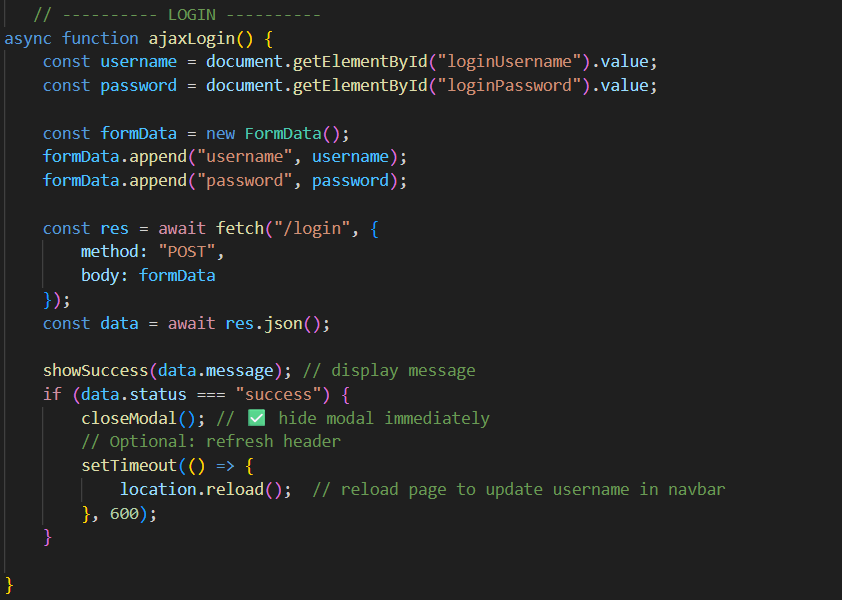
This function opens the authentication modal and switches between login, signup, and forgot-password forms based on user selection.

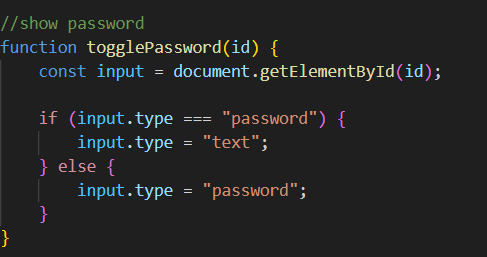
This function closes the authentication modal, improving user experience by avoiding page reloads.

This function dynamically toggles the visibility of authentication forms, ensuring that only the relevant form is displayed at a given time.

This function updates the navigation header based on the user’s login status stored in session storage. It dynamically switches between login/signup buttons and user display elements.

This function clears session data on the client side and redirects the user to the home page. It also displays a success message to confirm logout.

This asynchronous function handles user login by sending credentials to the backend /login route. It processes the server response, displays feedback messages, closes the authentication modal upon success, and refreshes the page to reflect the logged-in state.

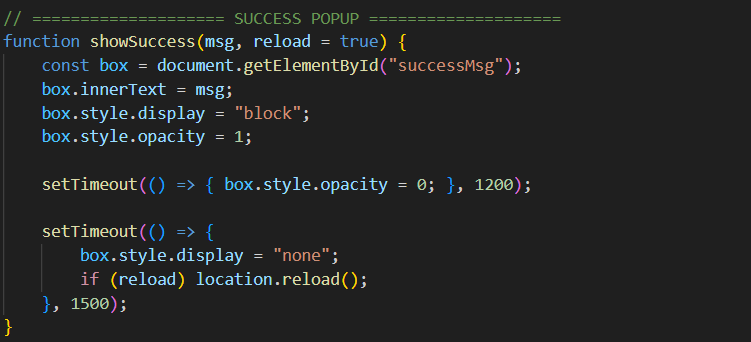


This utility function toggles password input visibility, allowing users to view or hide their password during entry.

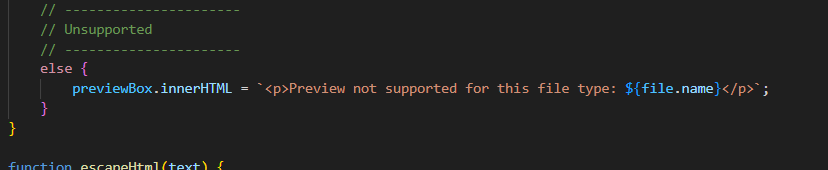
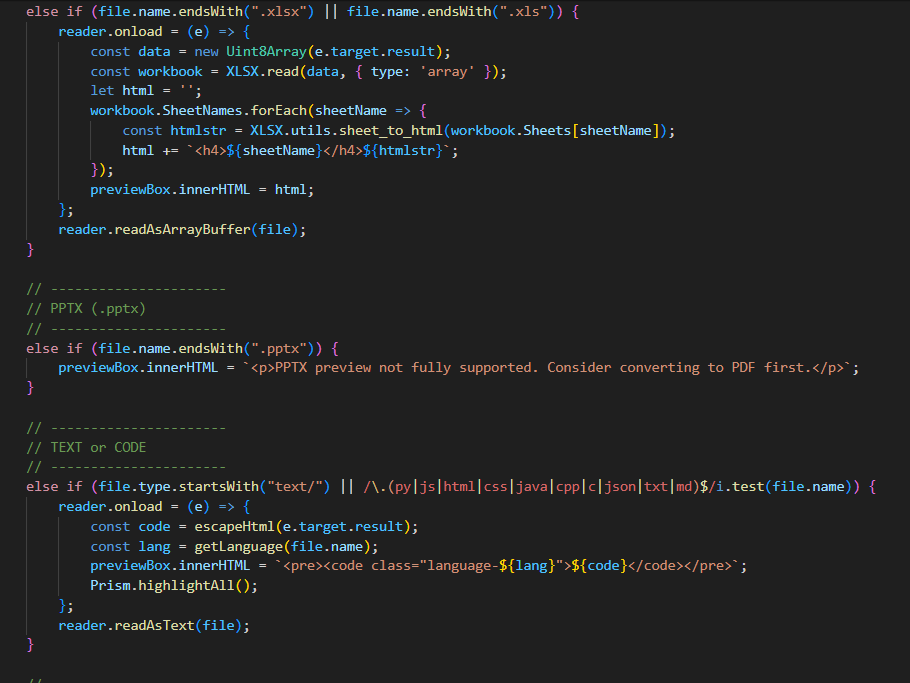
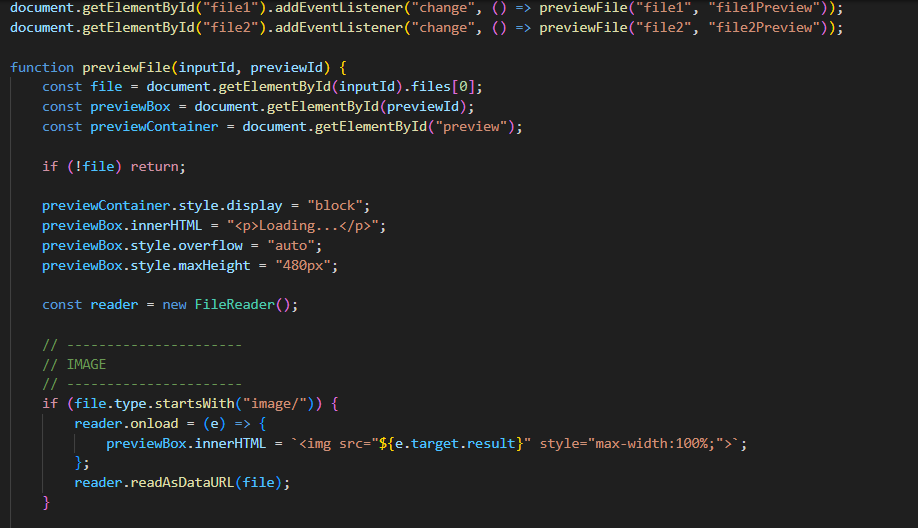
This function initiates the password reset process by sending the user’s email address to the backend. It handles server responses and displays appropriate success or error messages.

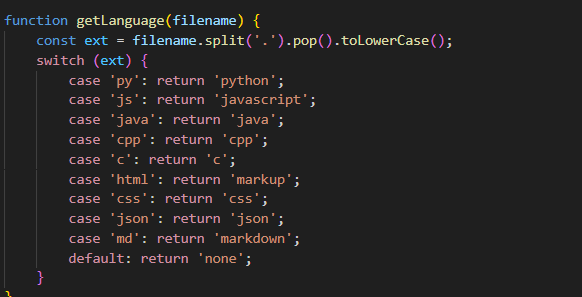
This function captures user registration details, validates input completeness, sends signup data to the backend, and handles server responses. Upon successful registration, the modal is closed and the page is refreshed to update the UI.

This function allows logged-in users to update their profile information. It submits updated data to the backend and dynamically updates the displayed username without refreshing the page.

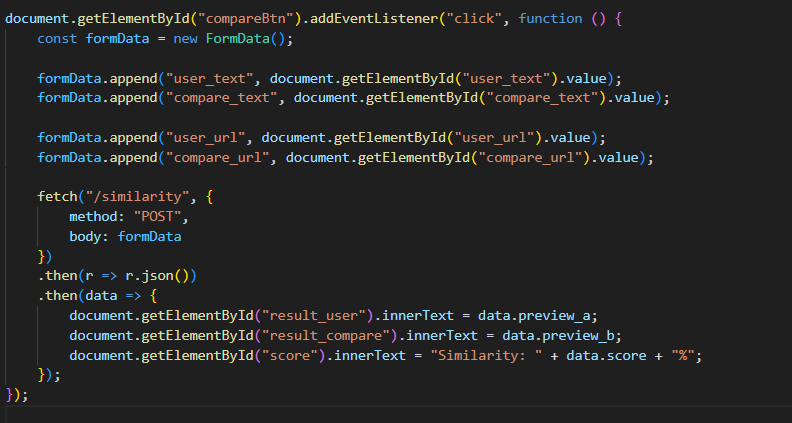
This function displays transient success or feedback messages to users. It optionally reloads the page depending on the context, improving both responsiveness and usability.

This function retrieves similarity analysis history from the backend and dynamically populates the history table. It allows users to review previous comparisons without navigating away from the page.

This function dynamically previews uploaded files before submission. It supports multiple file types including images, PDFs, Word documents, Excel files, text files, and source code. Syntax highlighting is applied to code files using Prism.js, enhancing readability.

This function determines the programming language of uploaded source code files based on file extension. It is used to enable correct syntax highlighting during preview.

This function performs a line-by-line comparison between two texts and visually highlights matching and differing lines using color-coded styling. It improves interpretability by allowing users to quickly identify similarities and differences.

This function allows users to submit text or URL inputs through a unified comparison button, simplifying the interface and improving user workflow.

#### **5.4.3.2 history**



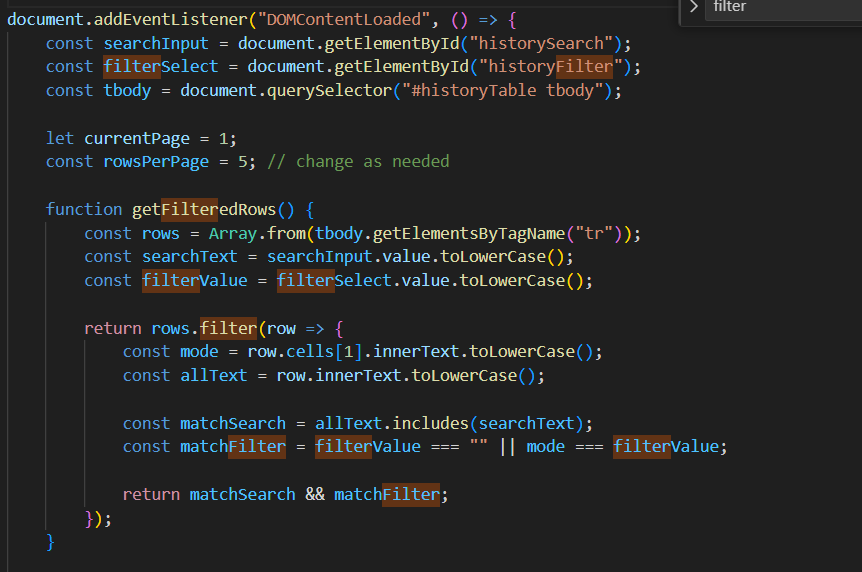
This JavaScript code implements a **client-side search and filter mechanism** for the history table and is executed only after the web page has fully loaded. The logic begins with the DOMContentLoaded event listener, which ensures that all required HTML elements are available in the Document Object Model (DOM) before any JavaScript operations are performed. This prevents common runtime errors that occur when scripts attempt to access elements that have not yet been rendered by the browser.

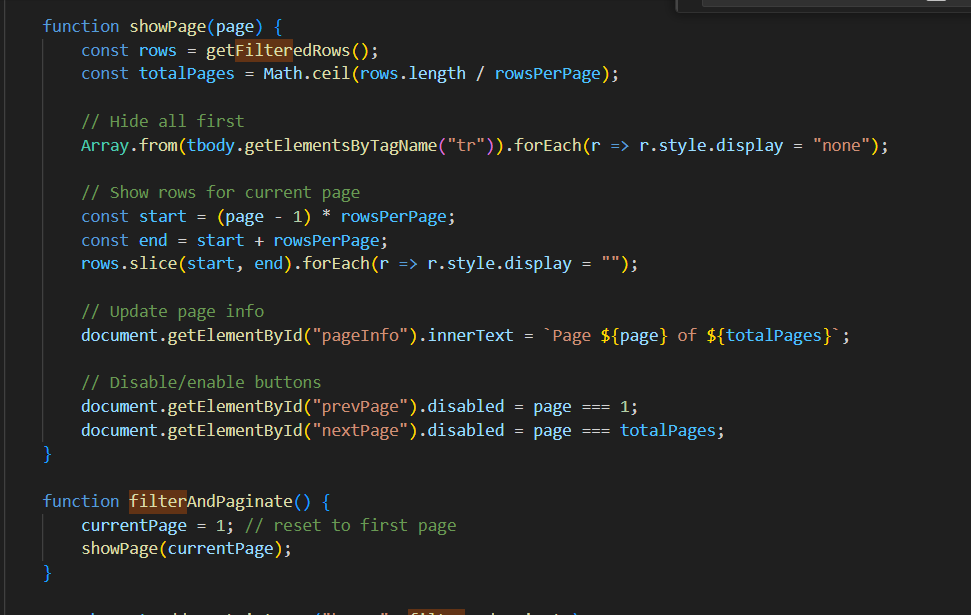
Once the page is ready, the script retrieves three essential elements: the search input field (historySearch), the filter dropdown menu (historyFilter), and the table body (tbody) that contains the history records. These elements serve as the primary user interaction points for filtering the displayed results. A validation check is then performed to confirm that all required elements exist. If any of these elements are missing, an error message is logged to the browser console and the script terminates early, ensuring graceful failure rather than breaking the application.

The core functionality is implemented within the filterHistory function, which dynamically controls the visibility of table rows based on user input. Each time the function is triggered, it retrieves the most up-to-date set of table rows from the table body. This approach ensures that the filtering logic remains accurate even if the table content changes dynamically. The function then reads the user’s search keyword and the selected filter value, converting both to lowercase to enable case-insensitive matching.

For each row in the table, the function extracts the comparison mode from the second table column and also retrieves the full text content of the row. Two matching conditions are evaluated: whether the row contains the search keyword and whether the row’s mode matches the selected filter option. If the filter dropdown is empty, all modes are considered valid. Rows that satisfy both conditions remain visible, while non-matching rows are hidden by setting their display style to none. This provides instant feedback to the user without reloading the page or querying the server.

Finally, event listeners are attached to both the search input field and the filter dropdown. The keyup event listener ensures that filtering occurs in real time as the user types, while the change event listener updates the table immediately when a different filter option is selected. Together, these interactions create a responsive and user-friendly interface that allows users to efficiently locate specific history records, improving overall usability and system interaction efficiency.







This JavaScript code implements **client-side filtering combined with pagination** for the history table, allowing users to search, filter, and navigate through records efficiently. The entire script is wrapped inside the DOMContentLoaded event listener to ensure that all HTML elements are fully loaded and accessible before any JavaScript logic is executed. This prevents errors related to missing or undefined DOM elements.

At the beginning of the script, references are obtained for the search input field, the filter dropdown menu, and the table body that contains the history records. Two pagination-related variables are then initialized: currentPage, which tracks the currently displayed page, and rowsPerPage, which defines how many rows should be displayed on each page. This configuration allows the system to divide large datasets into manageable segments, improving readability and user experience.

The getFilteredRows function is responsible for applying both the search and filter logic. It retrieves all rows from the table body and converts the user’s search input and selected filter option to lowercase to enable case-insensitive matching. For each row, the function extracts the comparison mode from the second table column and the entire row’s text content. It then evaluates whether the row matches the search keyword and whether it satisfies the selected filter criteria. Only rows that meet both conditions are returned, ensuring that pagination operates only on relevant data.

The showPage function controls the pagination display. It first retrieves the filtered rows and calculates the total number of pages based on the number of results and the defined rows per page. All table rows are initially hidden to reset the display. The function then calculates the range of rows that should appear on the current page and makes only those rows visible. Additionally, it updates the page indicator to inform the user of the current page number and total pages. The previous and next navigation buttons are also enabled or disabled dynamically to prevent invalid page navigation.

The filterAndPaginate function serves as a helper that resets pagination whenever the search term or filter selection changes. By resetting the page number to the first page and re-rendering the table, it ensures that users always see the most relevant results immediately after modifying the filter criteria.

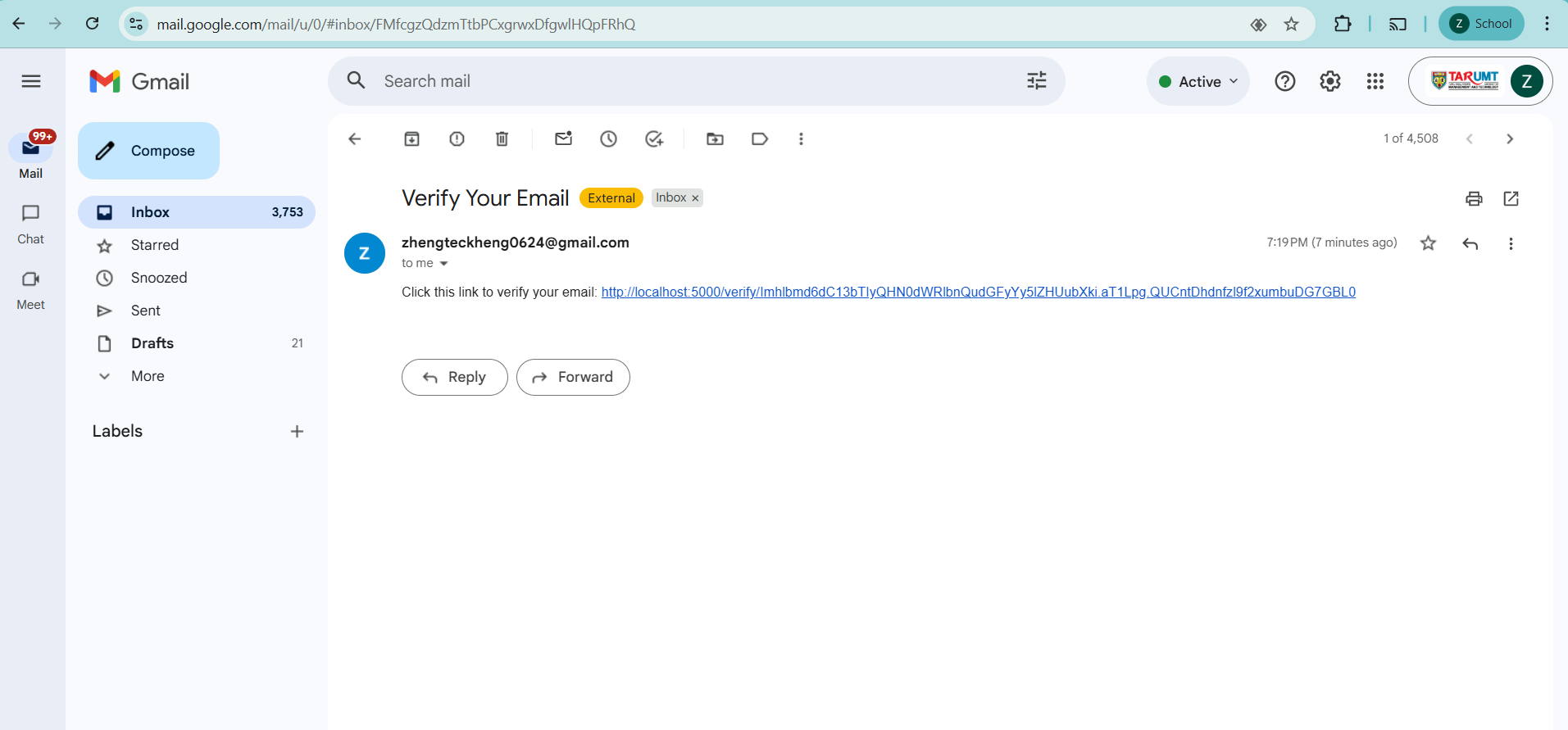
Event listeners are attached to the search input field and filter dropdown so that filtering and pagination update automatically as the user types or changes selection. Additional event listeners are attached to the “Previous” and “Next” buttons to allow navigation between pages. Each button updates the current page number and refreshes the displayed rows accordingly.

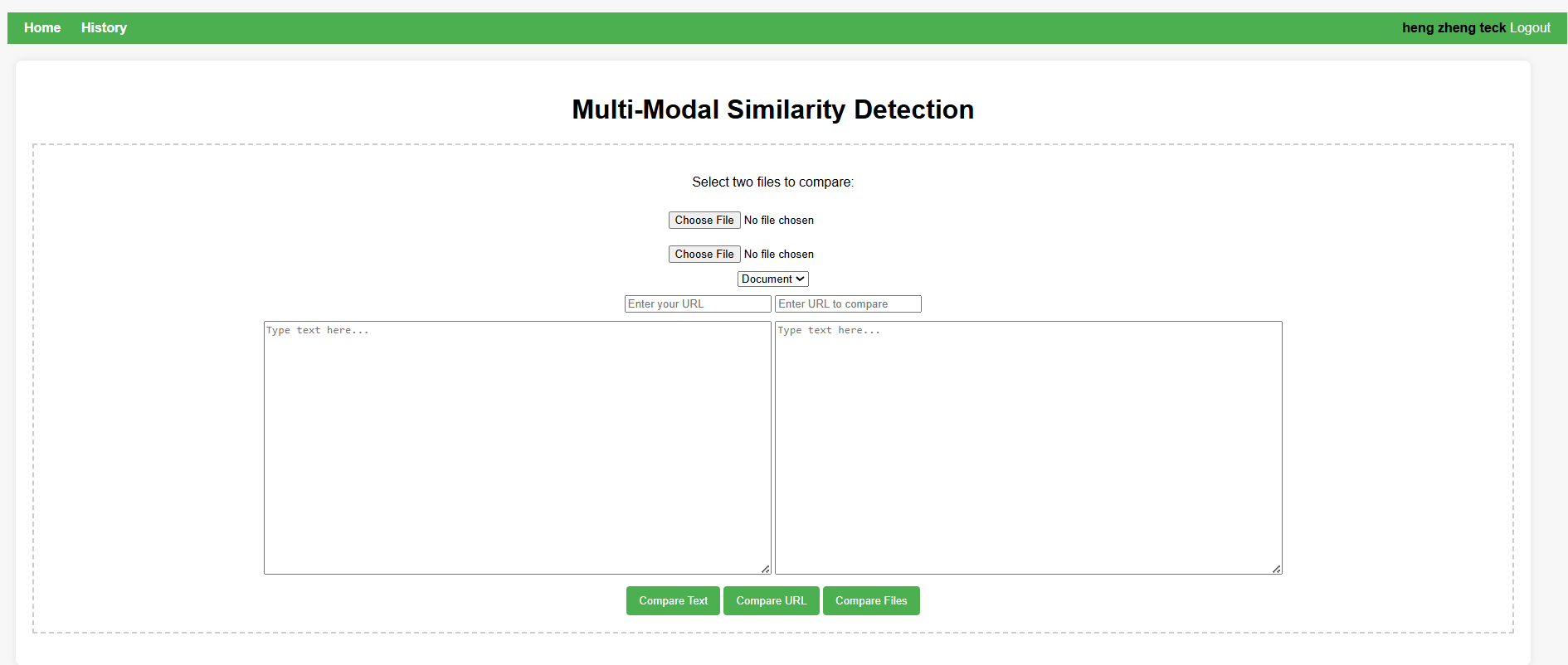
Finally, the script calls showPage(currentPage) when the page loads, ensuring that the initial dataset is displayed correctly with pagination applied. Overall, this implementation provides a responsive and efficient client-side solution for managing large sets of history records without requiring server-side requests, enhancing both performance and usability.

### **5.4.3 Result Visualization and Difference Highlighting**

#### **Signup**

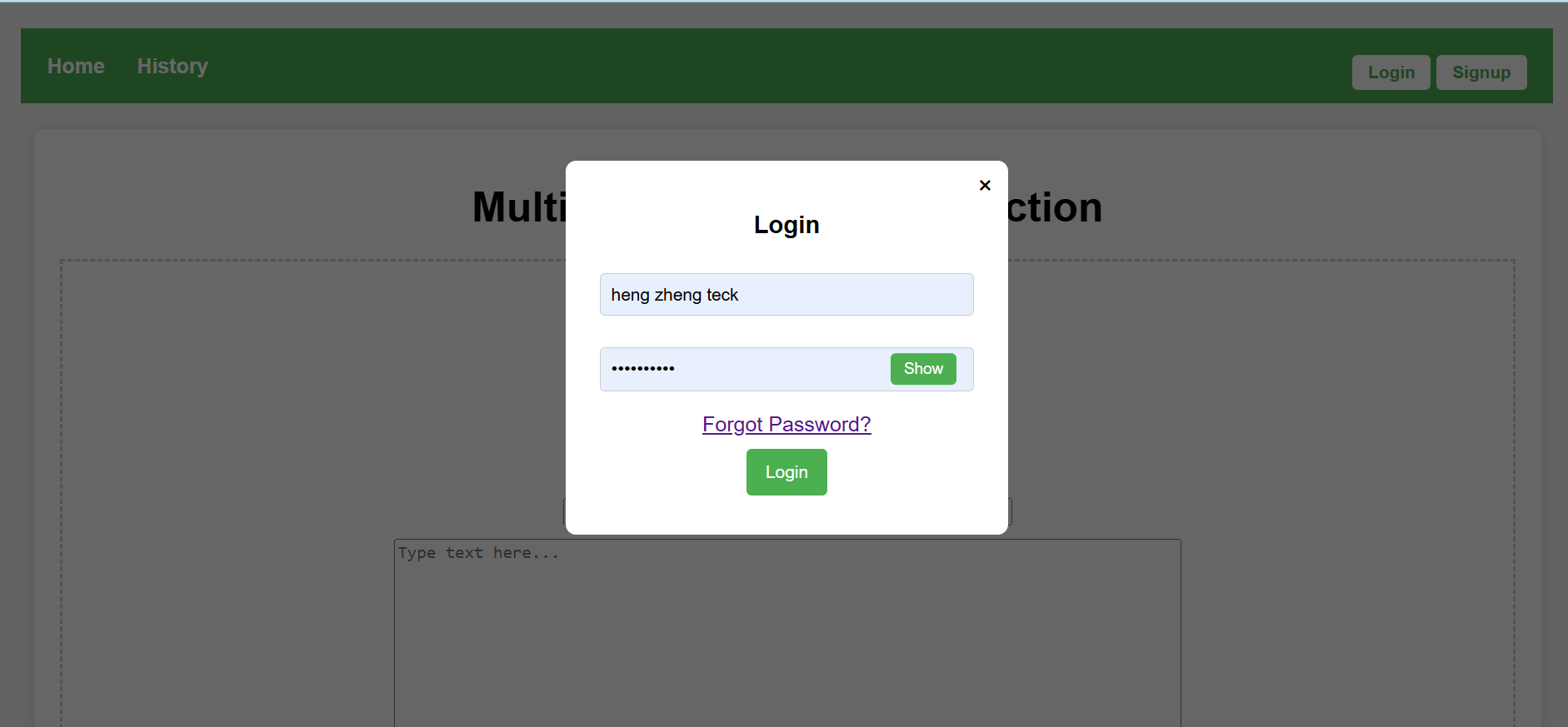
### 





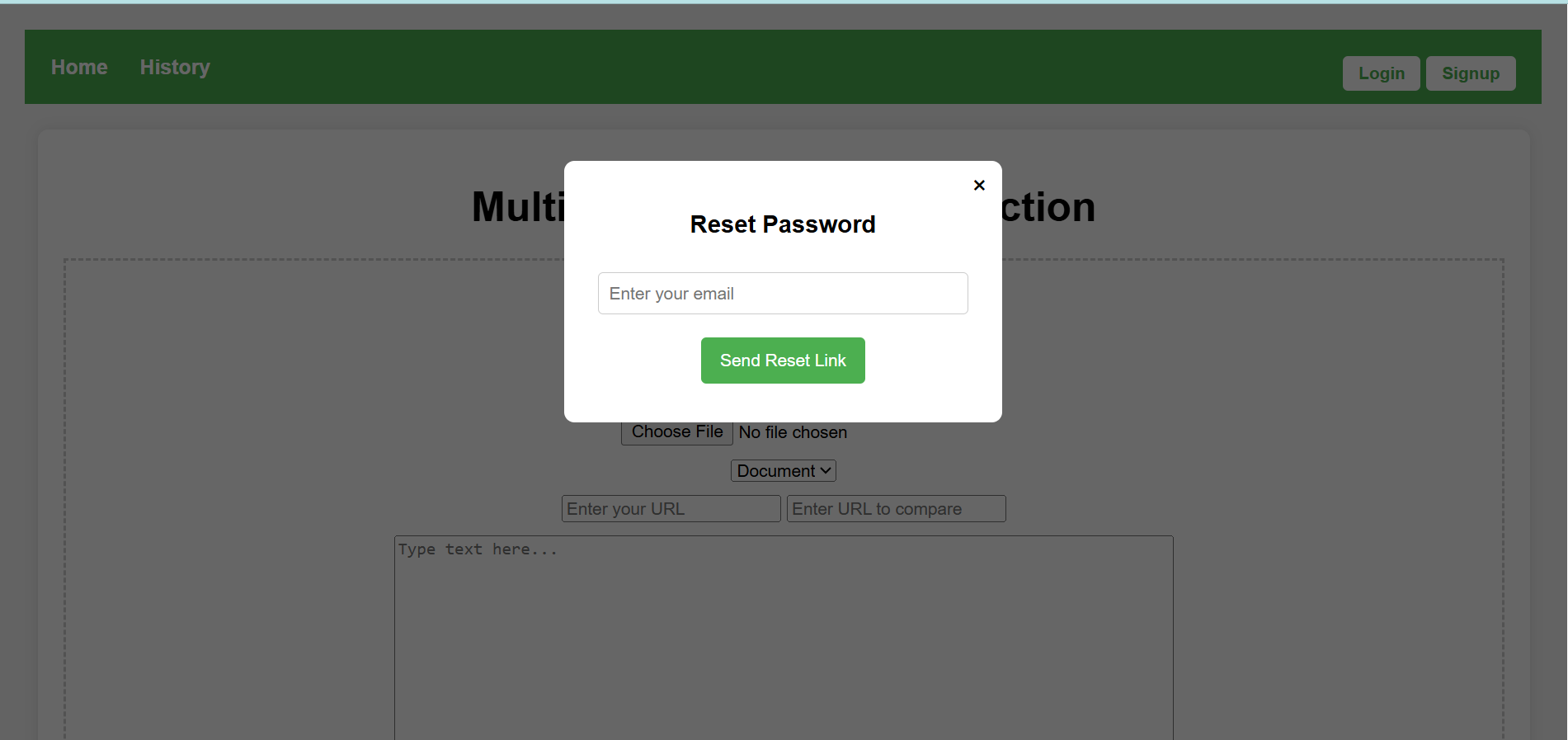
User create the account by using email , username and password. When user click the button , the system will send the email verification to confirm that email existence. User can click the show button to show actual password user input in order to ensure password is correct.

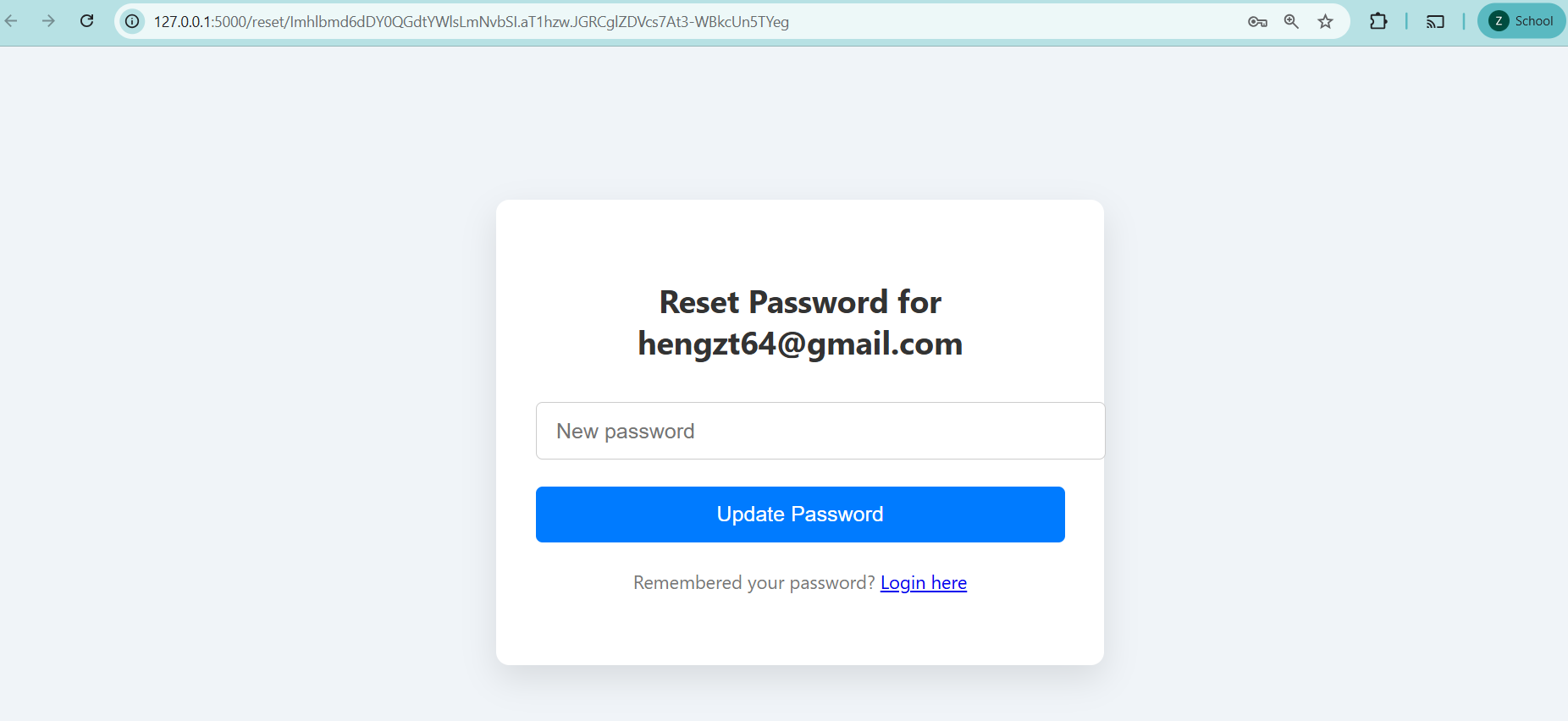
#### **Login**



After creating account, user can login account by correct username and password.

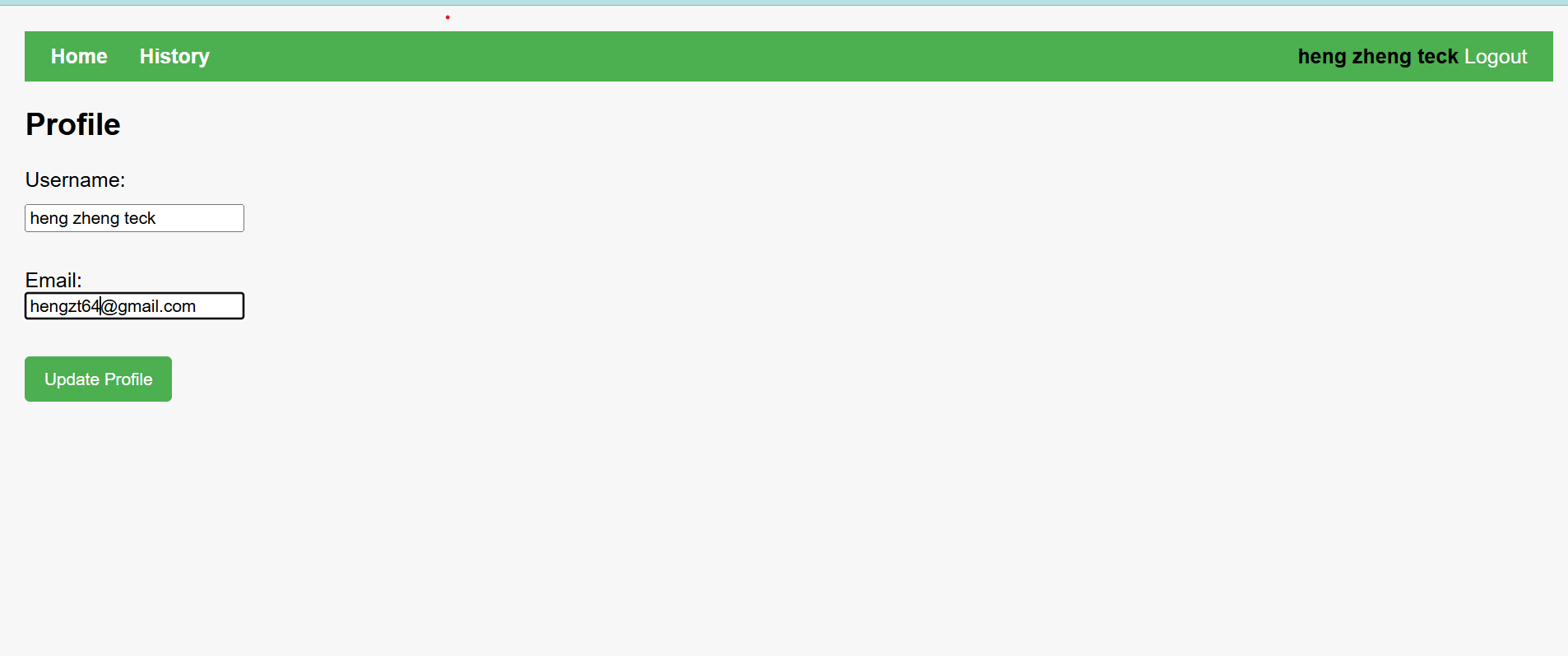
#### **Forgot password**





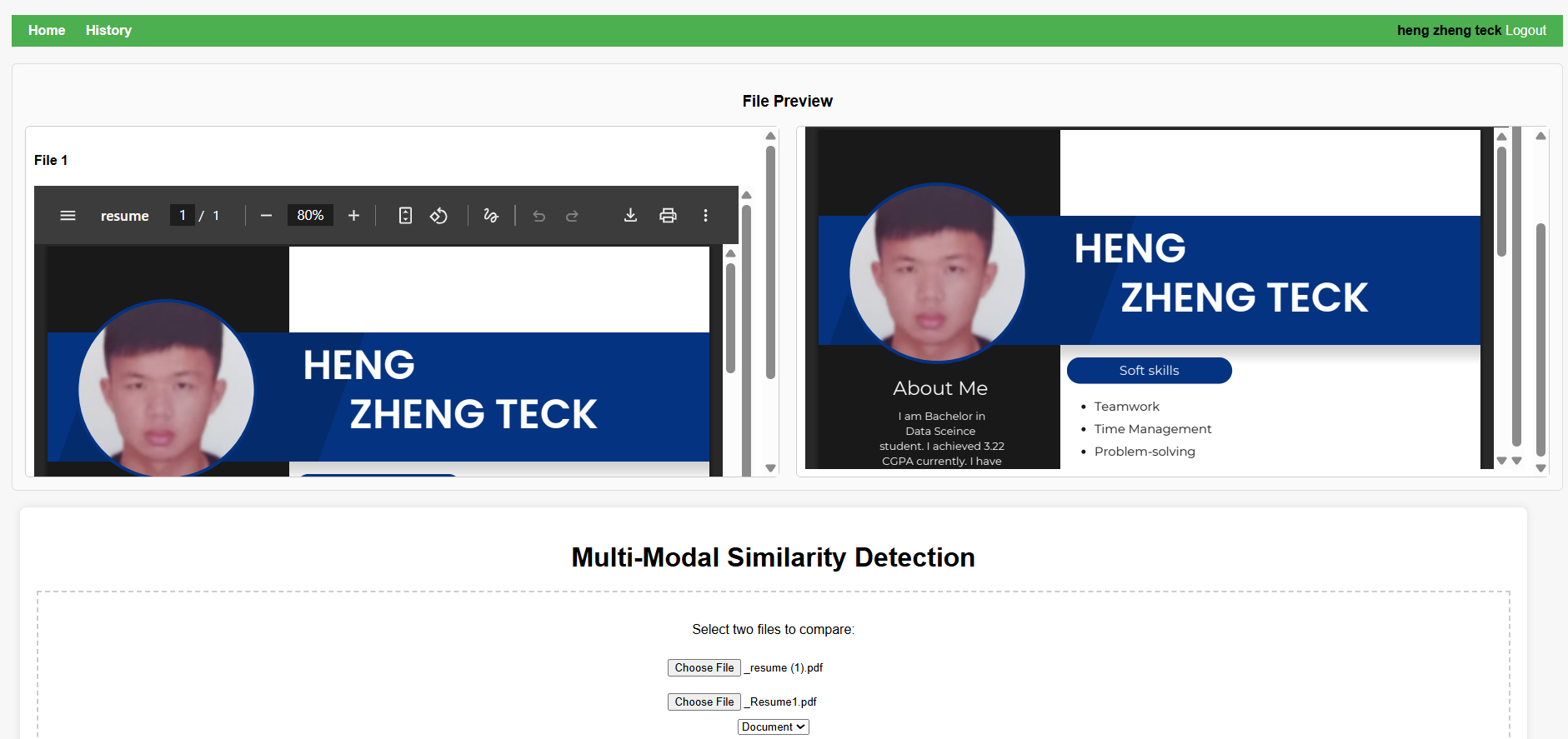
If user forgot password , user can enter email address that user registered and system will send email to allow user click the link and type new password.

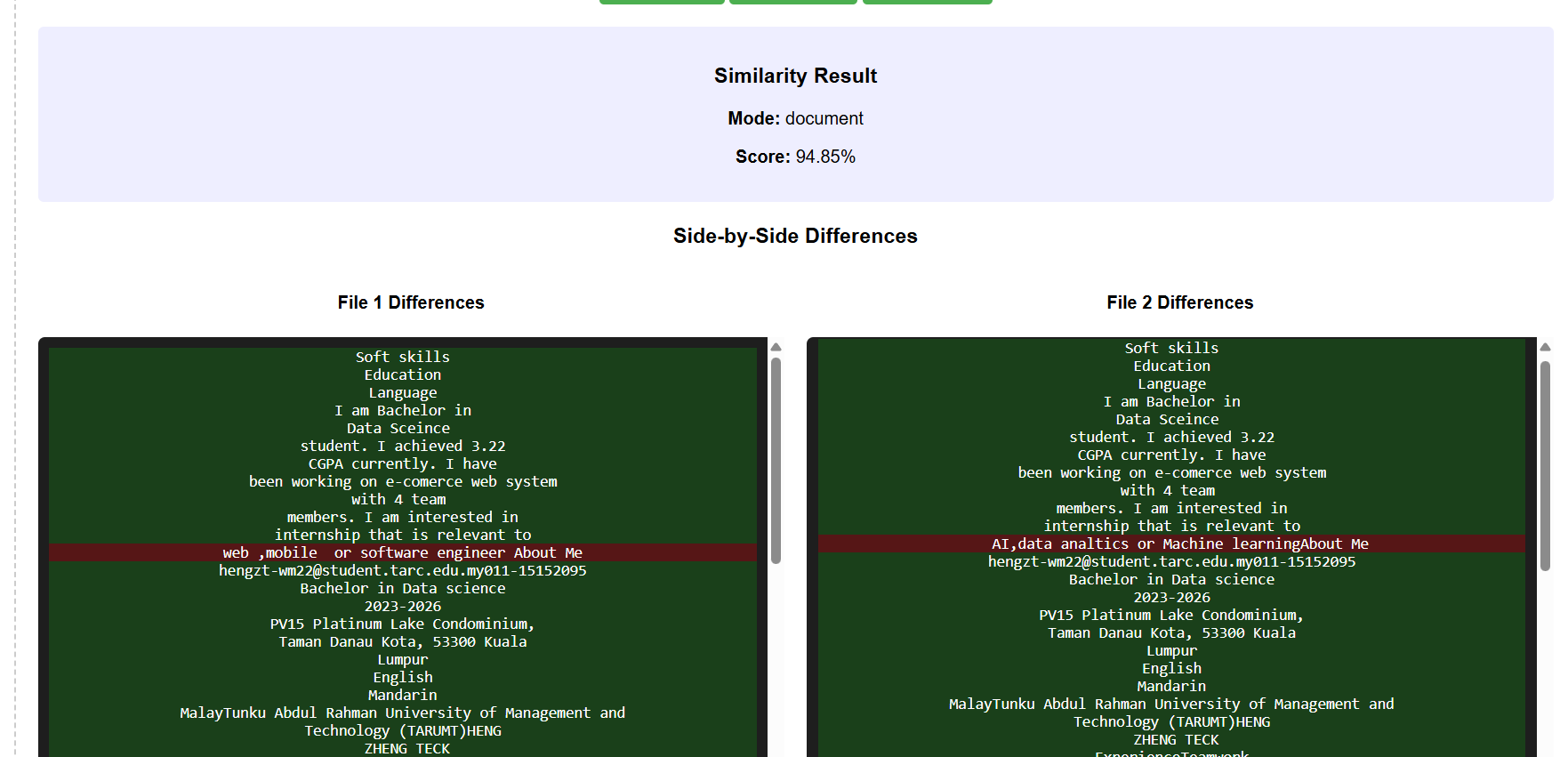
#### **Profile**



This is user profile page and allow user to update their email and username.

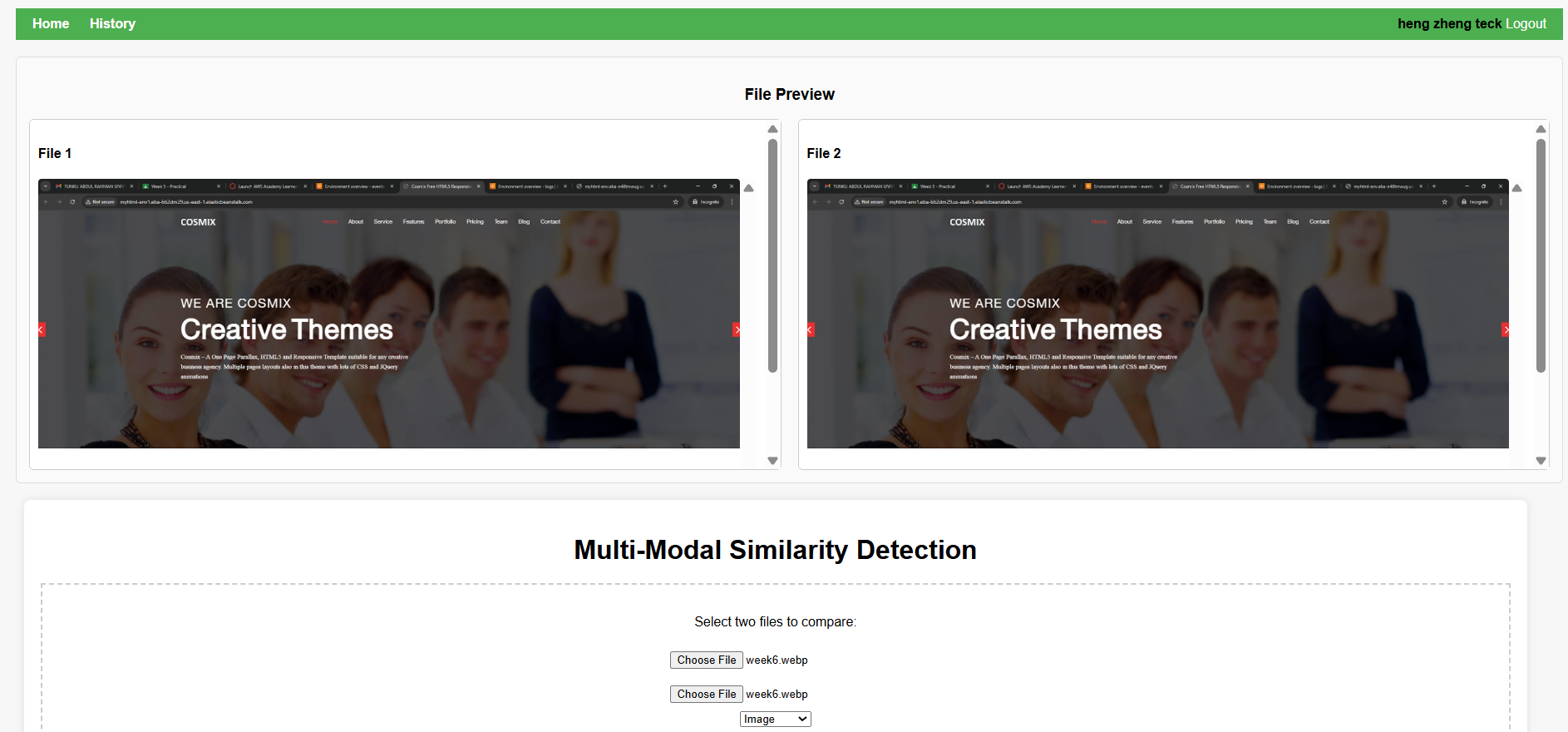
#### **Document comparison**

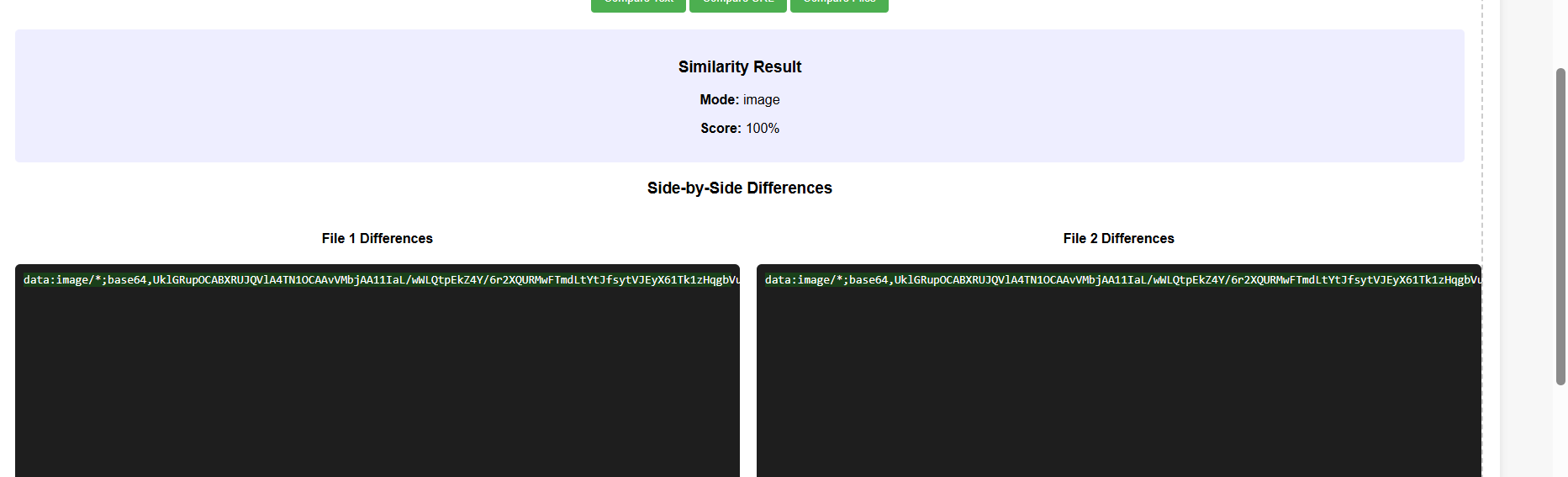




When user upload two pdf file and choosedocument mode , we can preview the file content what i upload . After clicking the compare file button , the system will show score, mode and area differences with highlighted for both files. The green is same while the red color is different. The files use upload are 94.85% similar because only a few area is different.

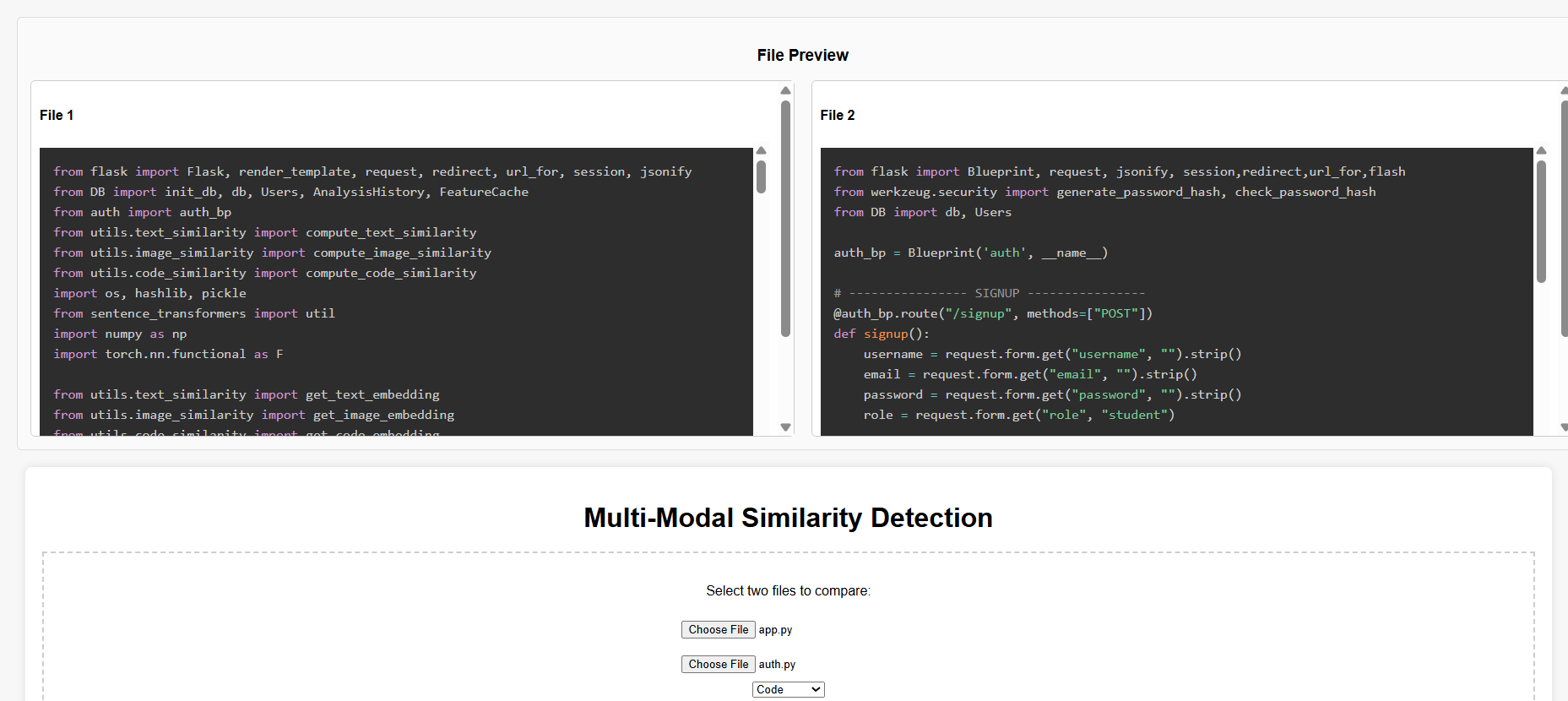
#### **Image comparison**

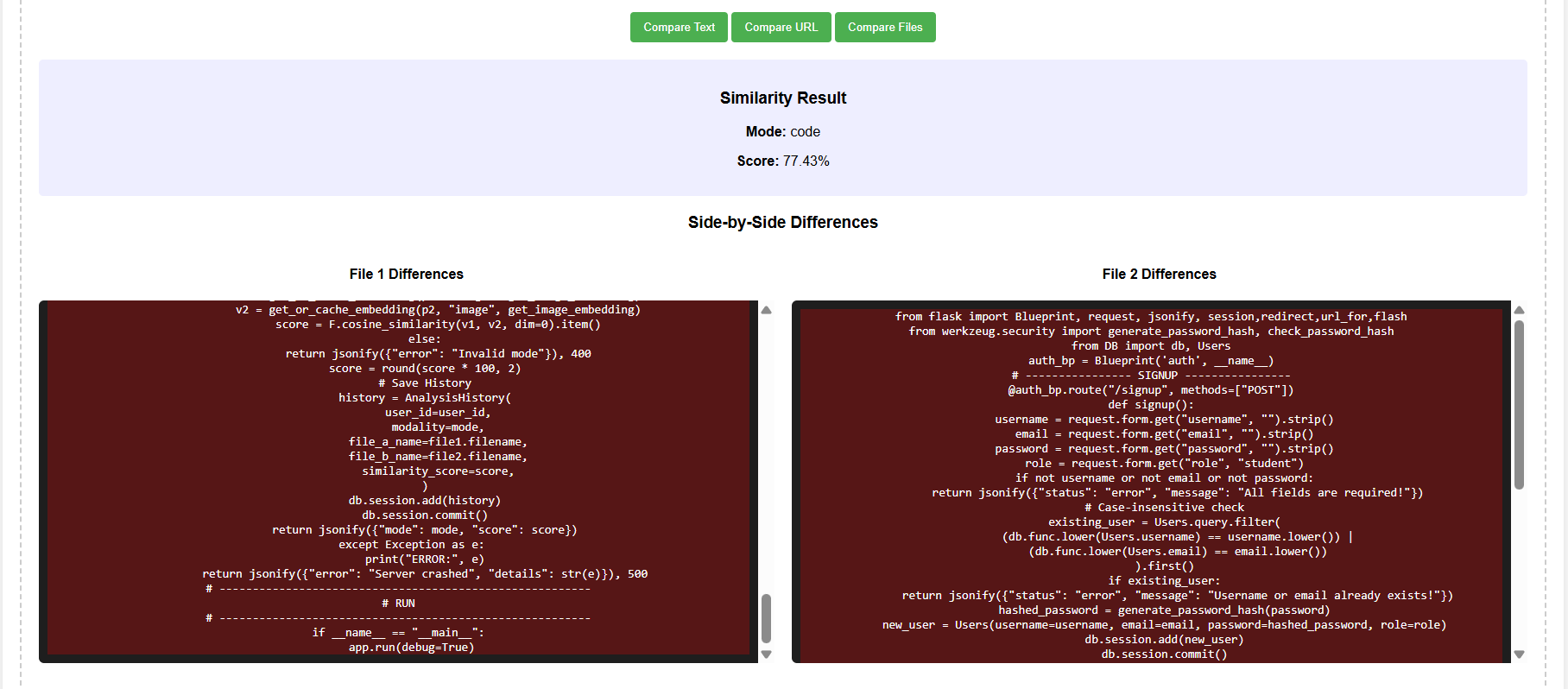
****

****

When user upload two image and choose image mode , we can preview the image content what i upload . After clicking the compare file button , the system will show score, mode and area differences with highlighted for both files. The green is same while the red color is different. The image use upload are 100% similar because of no difference in both images.

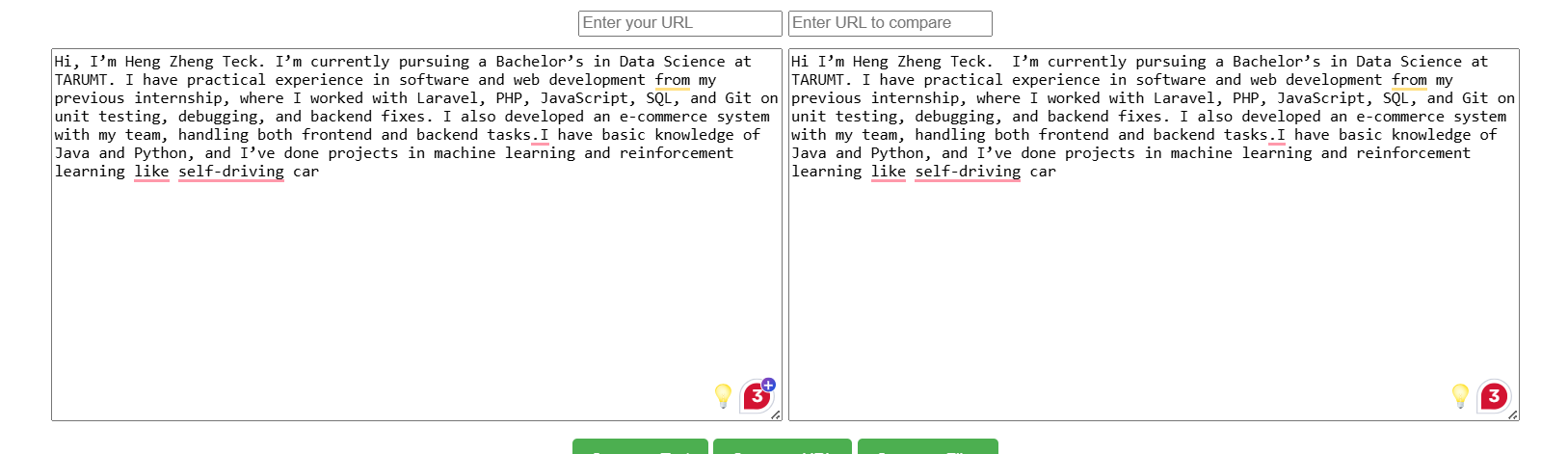
#### **Code comparison**

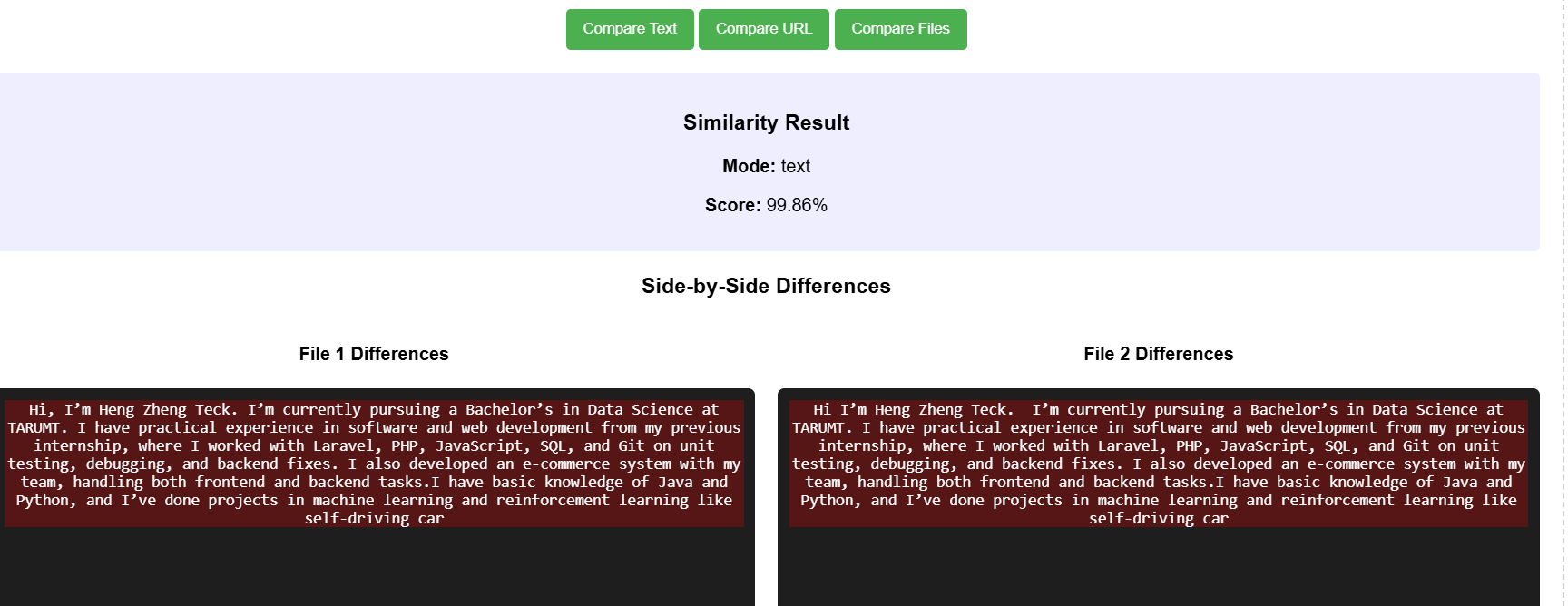
****

****

When user upload two code file and choose code mode , we can preview the code content what i upload . After clicking the compare file button , the system will show score, mode and area differences with highlighted for both code files. The green is same while the red color is different. The files use upload are 77.43% similar .

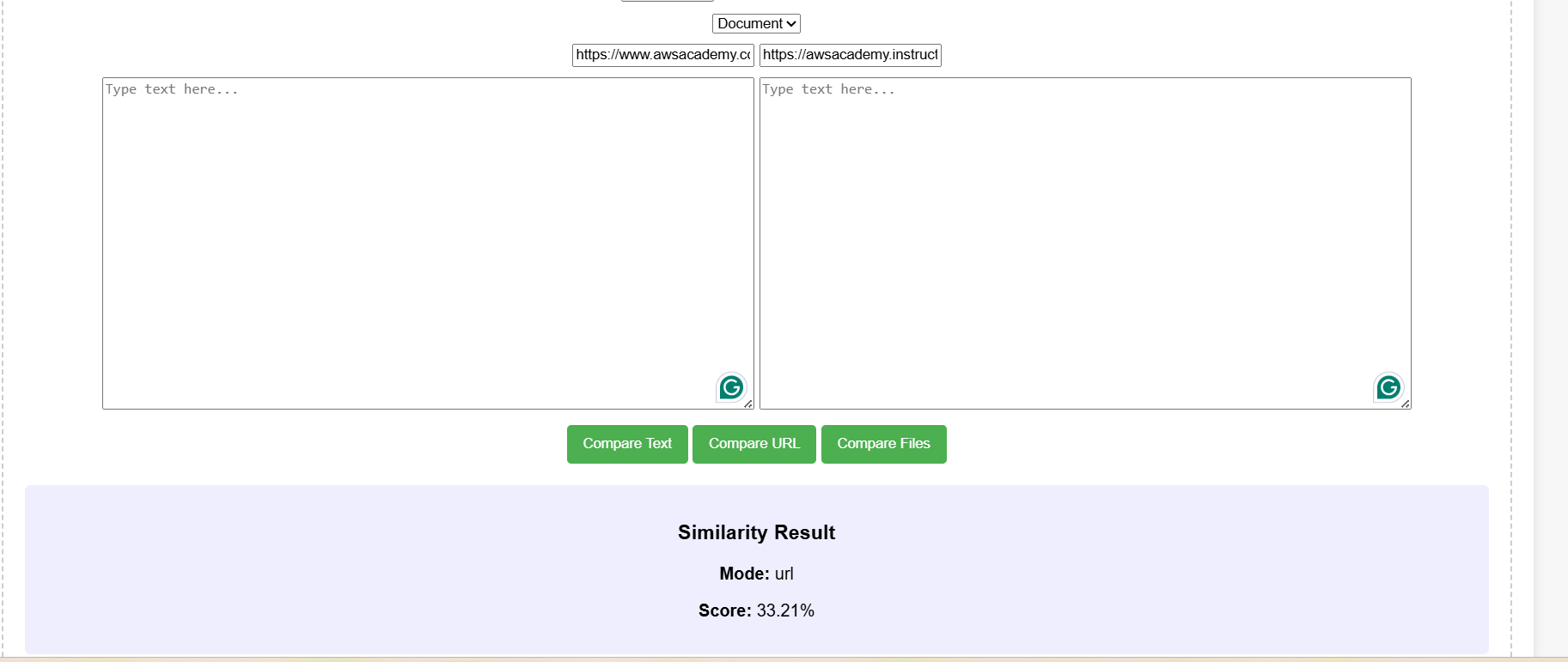
#### **Text comparison**

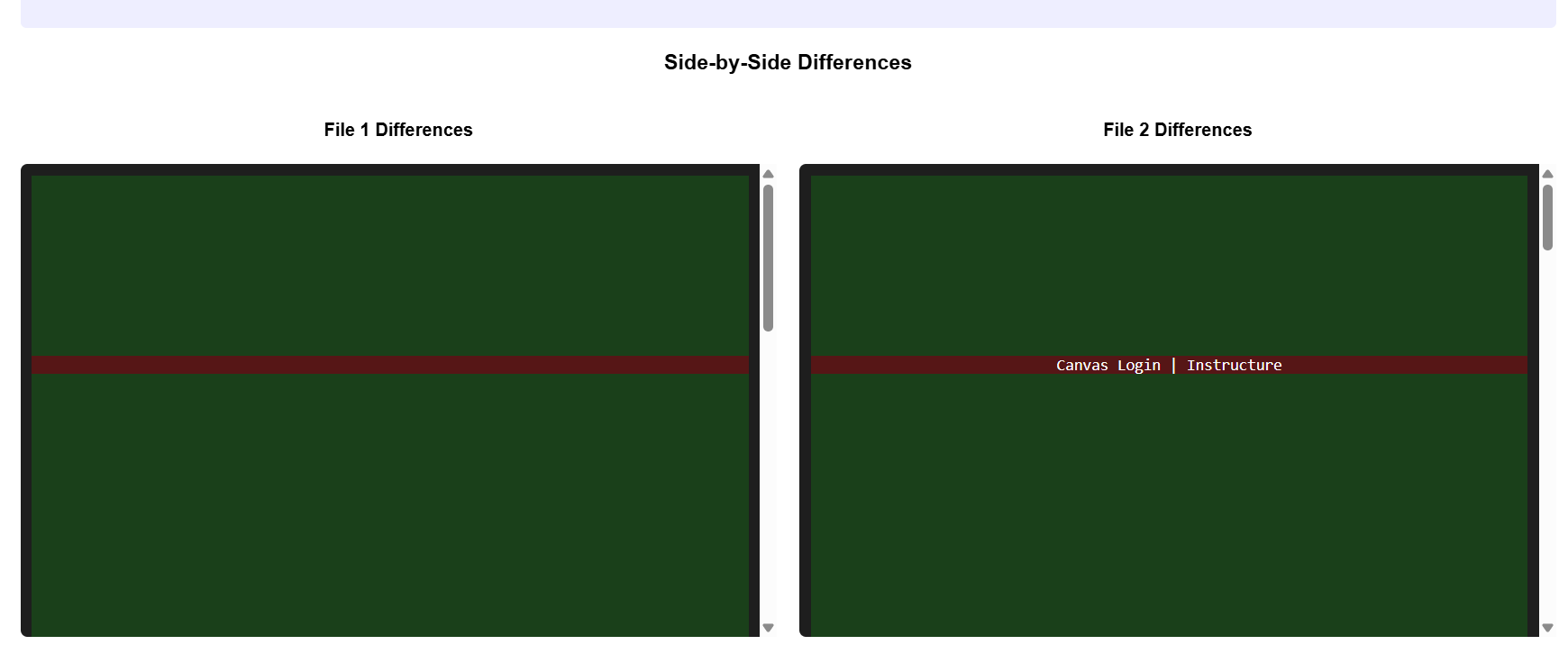
****

****

User can input two different paragraphs in two textarea .When user click the compareText button, the system will show mode , score and difference of both paragraph. The paragraph user input are 99.86% similar because only difference for comma and space in first line of paragraph.

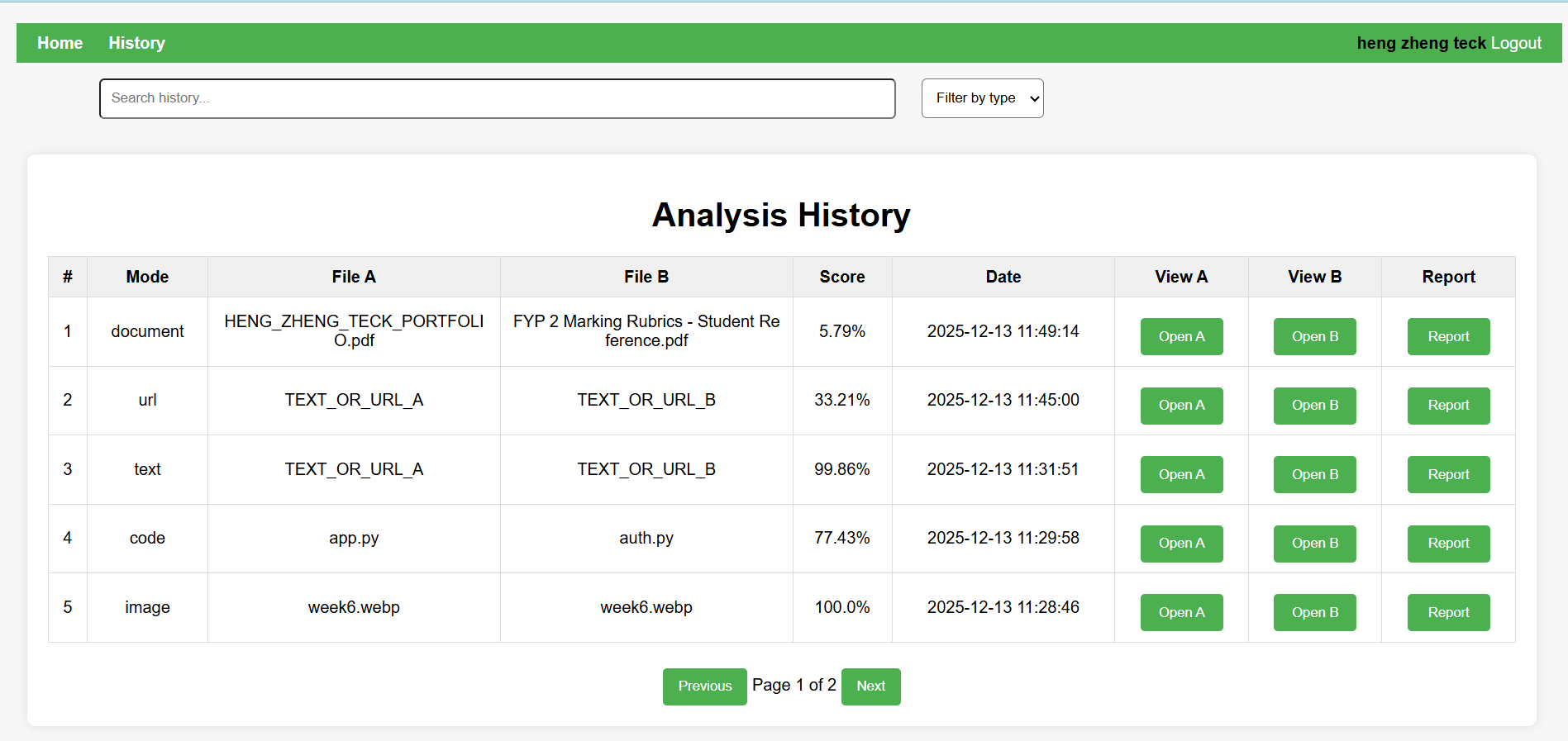
#### **URL comparison**

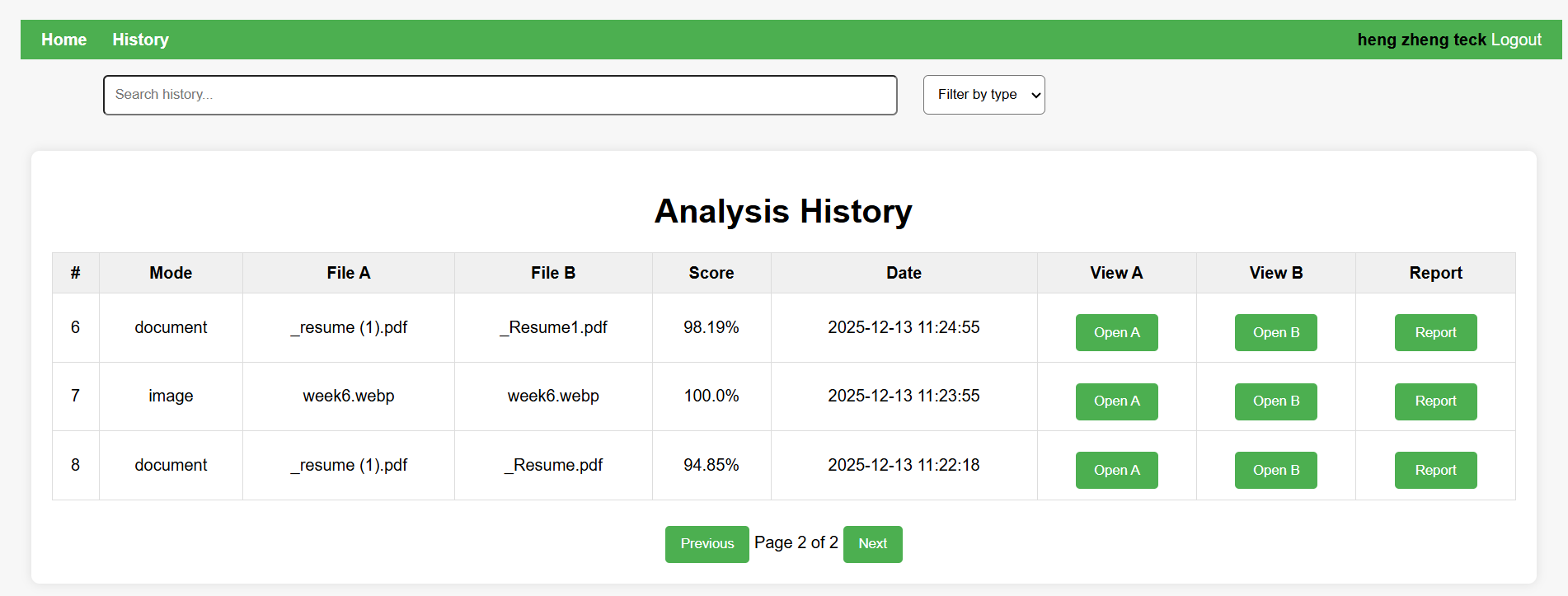
****

****

User can input two different URL in two input field .When user click the compareUrl button, the system will show mode , score and difference of both URL. The url user input are 33.21% similar.

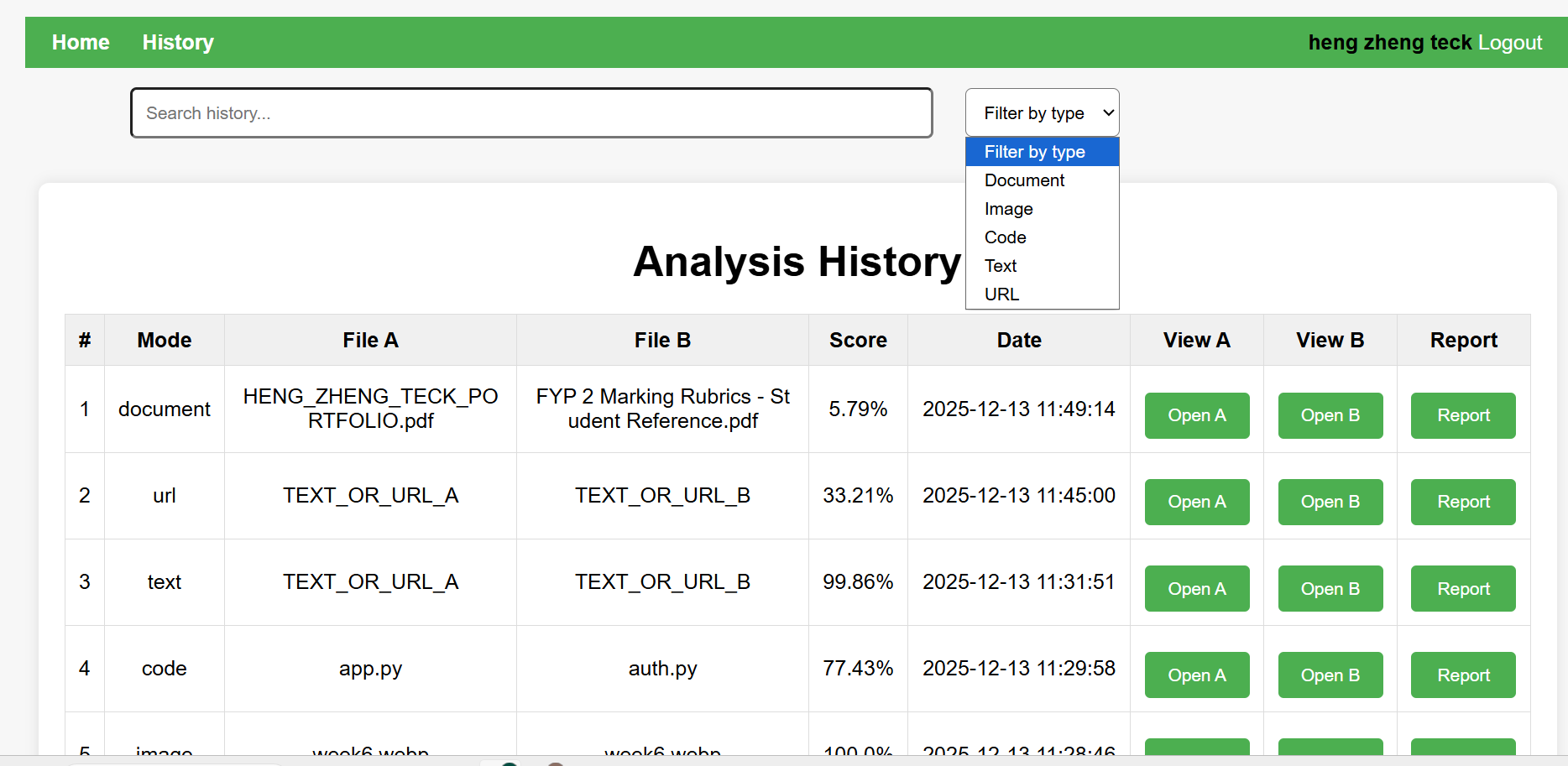
#### **History**

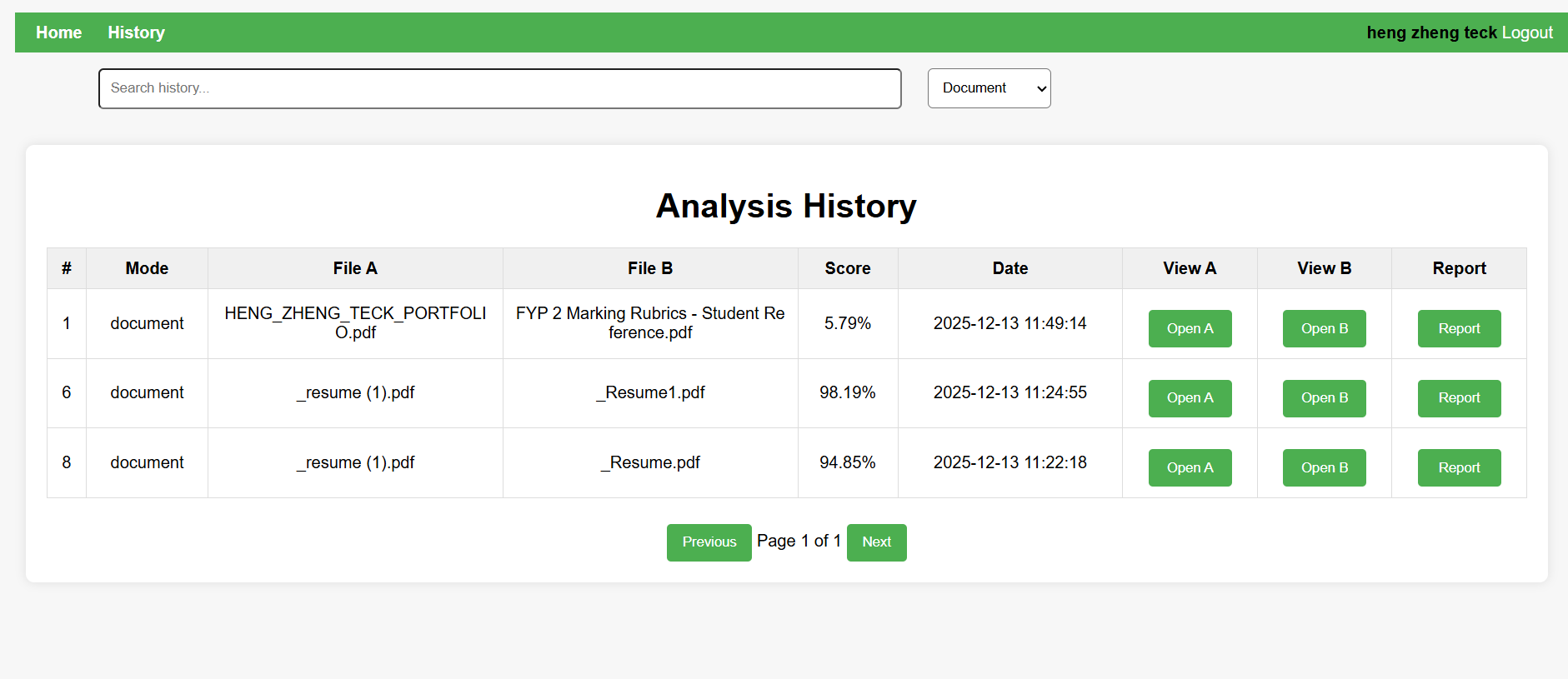
****

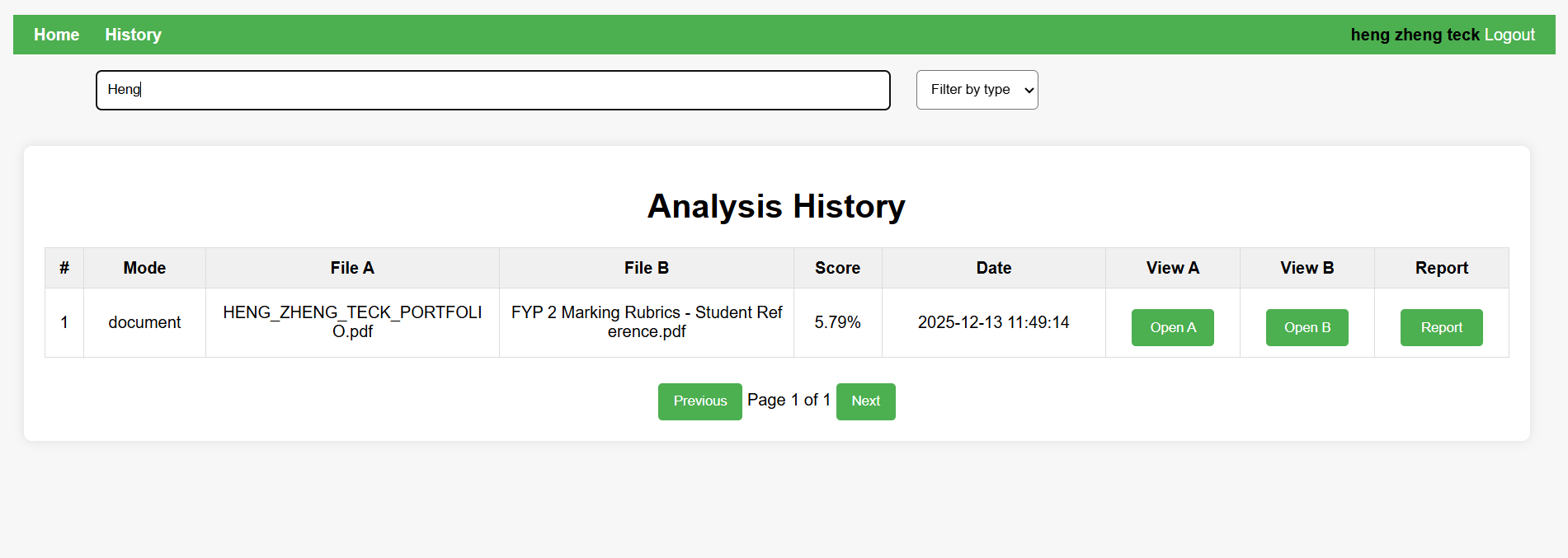
****

After performing any comparison , the system will show history what user input and uploaded. The system create pagination when history more than 5.

#### **Filter and search**

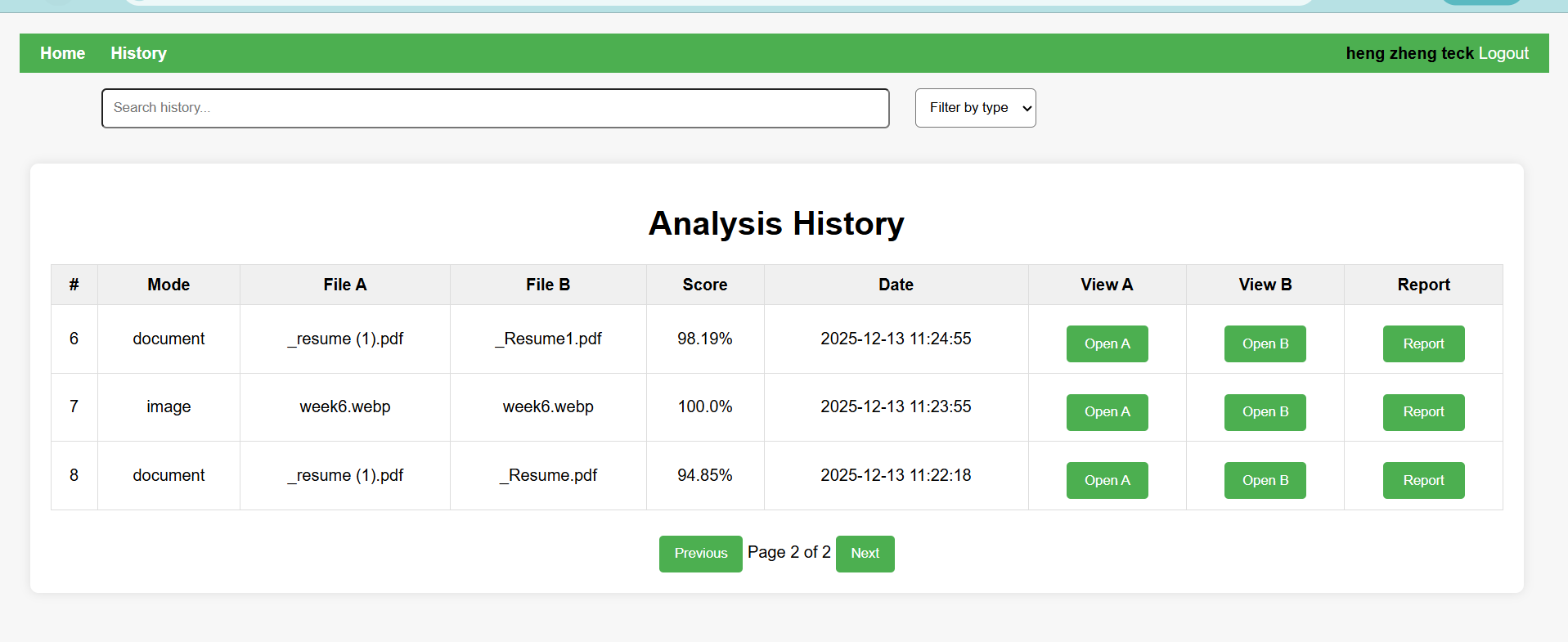


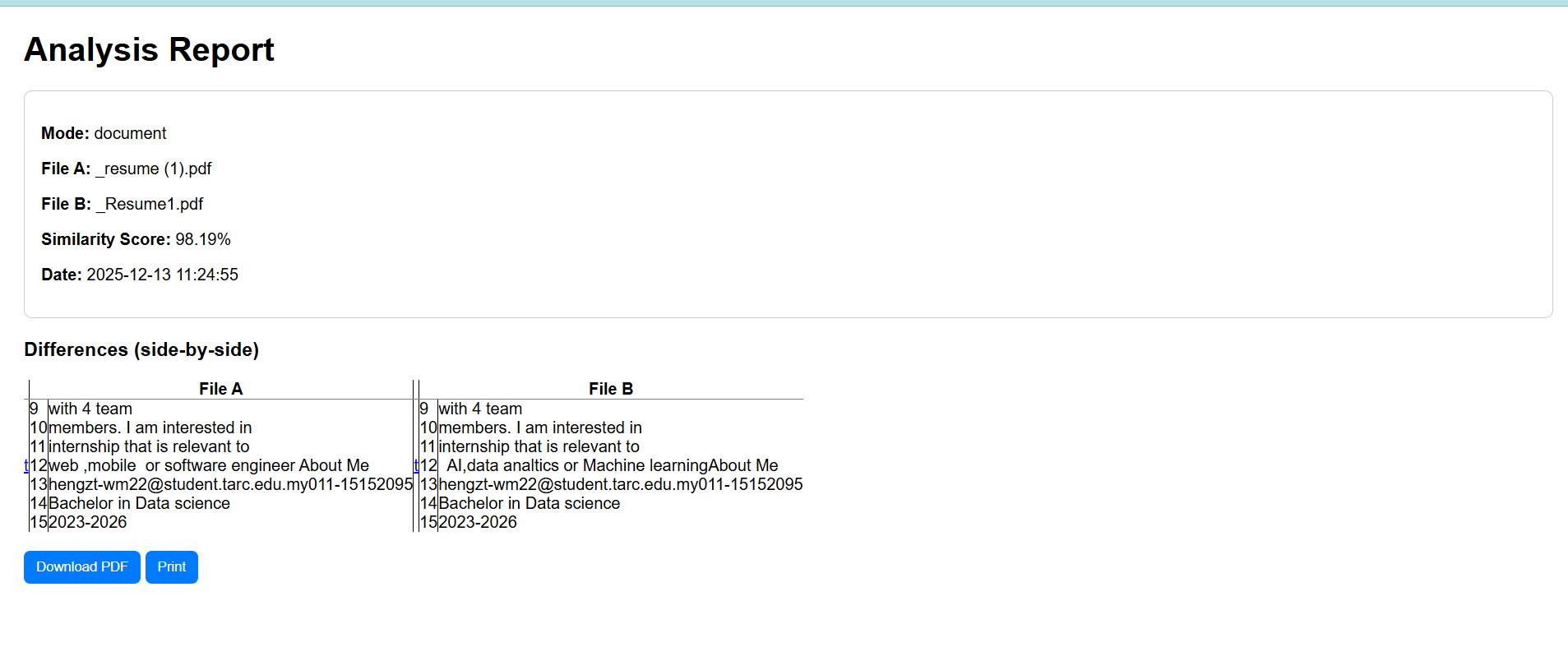


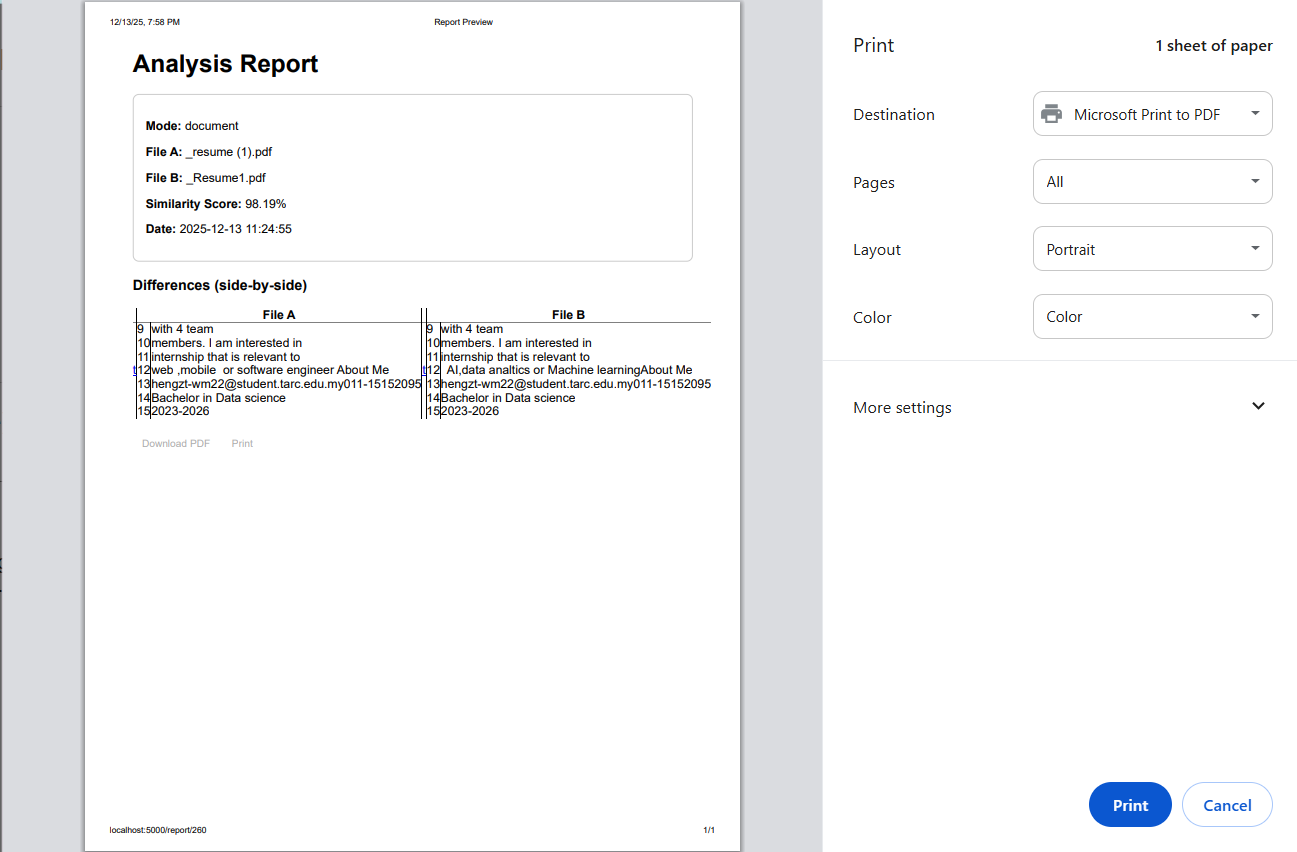


User can filter by mode including document , image , code , url and text. When user select document, the system will show only history that is document mode. User also can type the keyword for file name to search a specific history.

#### **View and Download report**







The system also allow user to view report , file what user done comparison before . User also can download and print the report in pdf format

## **5.5 System Integration**

This section describes how the different components of the multi-modal similarity detection system are integrated to function as a complete and cohesive application. The system adopts a client–server architecture, where the frontend interface interacts with the backend services through asynchronous HTTP requests. This integration ensures smooth communication between user actions, similarity computation modules, database storage, and result presentation.

On the frontend, user inputs such as uploaded files, text content, or URLs are collected using HTML forms and JavaScript. These inputs are packaged using the FormData object and transmitted to the backend via the Fetch API. This approach enables asynchronous communication, allowing similarity analysis to be performed without reloading the page. The backend Flask application receives these requests through the /similarity endpoint, where the input type (document, image, code, text, or URL) is first validated before processing.

Once the request reaches the backend, the system dynamically routes the input to the appropriate similarity computation module. Text and URL inputs are processed using a transformer-based sentence embedding model, code files are analyzed using either Abstract Syntax Tree (AST)–based vectors or token-based representations, and image files are processed using a pre-trained ResNet50 deep learning model. The similarity score is computed using cosine similarity, providing a consistent similarity metric across different modalities.

After computation, the backend integrates the results with the database layer by storing analysis records in the AnalysisHistory table. This includes metadata such as file names, similarity scores, timestamps, and optional difference visualizations. Cached feature embeddings are reused when available to improve performance. Finally, the backend returns the results in JSON format to the frontend, where JavaScript dynamically updates the user interface to display similarity scores, previews, and side-by-side difference highlighting. This seamless integration ensures that all system components operate together efficiently and transparently from the user’s perspective.

## **5.6 Error Handling and Validation**

Error handling and validation mechanisms are implemented throughout the system to ensure robustness, security, and a positive user experience. Both frontend and backend layers perform validation to prevent invalid inputs and system failures. On the frontend, basic validation checks are applied before sending requests, such as ensuring required fields are not empty and confirming that file inputs are selected when necessary. These checks reduce unnecessary server requests and provide immediate feedback to users.

On the backend, more comprehensive validation is enforced to maintain system integrity. Uploaded files are validated against predefined allowed file extensions based on the selected comparison mode, preventing unsupported or potentially harmful file types from being processed. For text and URL-based comparisons, the system ensures that extracted content is not empty before performing similarity computation. User authentication routes also include validation to check for missing fields, invalid credentials, and duplicate email registrations.

Exception handling is implemented using try–except blocks to prevent unexpected runtime errors from crashing the server. In the event of an error, the system logs the exception and returns a structured error message to the frontend in JSON format. This ensures that failures are handled gracefully and that users receive meaningful feedback instead of system-level error messages. Additionally, security-related validations such as password strength checking, password hashing, email verification tokens, and session validation further enhance the reliability and safety of the system.

## **5.7 System Testing**

System testing was conducted to verify that the developed system meets functional requirements and performs accurately across different comparison modes. A black-box testing approach was adopted, focusing on validating system behavior based on input and output without considering internal implementation details. Manual testing was primarily used due to the interactive and user-centric nature of the application.

### **5.7.1 Testing Strategy**

The testing process focused on functional correctness, usability, and stability. Each major system feature, including authentication, similarity computation, file upload, result visualization, and history tracking, was tested independently and as part of the integrated system. Multiple test cases were executed using different data types to evaluate accuracy and robustness.

| Test Case ID | Test Description | Input | Expected Outcome | Actual Result |
| --- | --- | --- | --- | --- |
| TC01 | User login with valid credentials | Correct username and password | User logged in successfully | Pass |
| TC02 | User login with invalid credentials | Incorrect password | Error message displayed | Pass |
| TC03 | Document similarity comparison | Two similar PDF files | High similarity score returned | Pass |
| TC04 | Code similarity detection | Two similar Python files | High similarity score returned | Pass |
| TC05 | Image similarity detection | Two identical images | Similarity score close to 100% | Pass |
| TC06 | Text similarity comparison | Two related text paragraphs | Moderate to high similarity score | Pass |
| TC07 | URL similarity comparison | Two webpages with similar content | Valid similarity score returned | Pass |
| TC08 | Unsupported file upload | Invalid file extension | Error message displayed | Pass |
| TC09 | View analysis history | Logged-in user | Previous records displayed | Pass |
| TC10 | Generate similarity report | Existing analysis record | PDF report generated successfully | Pass |

### **5.7.2 Testing Results and Discussion**

The testing results indicate that the system functions correctly across all supported modalities. Similarity scores produced by the system were consistent with expectations, demonstrating the effectiveness of the selected similarity algorithms. The caching mechanism significantly improved performance when processing previously analyzed files. Error handling mechanisms successfully prevented invalid inputs from causing system failures. Overall, the system met all functional requirements and demonstrated stable and reliable behavior during testing.

## **5.8 Security Considerations**

Security was an important consideration in the development of the multi-modal similarity detection system, particularly due to the handling of user accounts, uploaded files, and sensitive analysis data. Several security mechanisms were implemented to protect user information and ensure safe system operation.

User passwords are not stored in plain text. Instead, password hashing is applied during user registration and password updates. When a user creates or resets a password, it is processed using a secure hashing algorithm before being stored in the database. During login, the entered password is hashed and compared with the stored hash. This approach ensures that even if the database is compromised, user passwords cannot be directly recovered, thereby significantly enhancing account security.

Email verification is incorporated through a secure token-based password reset mechanism. When a user requests a password reset, a time-limited token is generated using a cryptographic serializer and sent to the user’s registered email address. The token expires after a predefined duration, preventing reuse or unauthorized access. This mechanism ensures that only the legitimate email owner can reset the associated account password.

Session handling is managed using Flask’s built-in session management system. User authentication status is maintained using secure session cookies, allowing the system to identify logged-in users across requests. Access to sensitive features such as similarity analysis history, profile management, and report generation is restricted to authenticated users only. This prevents unauthorized access to user data and ensures that each user can only view their own analysis records.

File upload safety is enforced through multiple validation layers. The system restricts uploads to predefined allowed file extensions based on the selected comparison mode, preventing execution of malicious files. Filenames are sanitized using secure utilities to prevent directory traversal attacks. Uploaded files are stored in a dedicated server directory with controlled access, and file content is processed safely without execution. Additionally, uploaded data is validated before similarity computation to ensure system stability and reliability.

Overall, these security measures collectively ensure that the system is resilient against common web application threats while maintaining user trust and data integrity.

## **5.9 Summary**

This chapter presented the development and testing of the multi-modal similarity detection system. The chapter began by describing the development environment and tools used, followed by a detailed explanation of backend and frontend implementation. Key system components such as text, image, and code similarity modules were successfully implemented and integrated using a client–server architecture.

System integration was discussed to demonstrate how frontend inputs, backend processing, database storage, and result visualization work together seamlessly. Error handling and validation mechanisms were incorporated to improve system robustness and prevent invalid or malicious inputs. Comprehensive system testing was conducted to verify functional correctness, usability, and reliability across all supported comparison modes.

Security considerations were also addressed, including password hashing, secure session handling, email-based password recovery, and file upload validation. These measures ensure that the system operates safely while protecting user data.

Based on the implementation and testing results, it can be concluded that the developed system successfully meets the defined functional and non-functional requirements. The system is capable of accurately performing similarity analysis across multiple data modalities, providing reliable results, and offering a secure and user-friendly experience.

# Chapter 6 Summary and Conclusions

## **6.1 Introduction**

This chapter provides a comprehensive summary of the research presented in Chapters 1 through 5 of this project, which focused on the design, development, and evaluation of a **Multi-Modal Similarity Detection System** capable of analyzing **text, document, image, source code, and URL-based content**. The primary objective of the project was to address the limitations of existing single-modality similarity detection tools by proposing and implementing a unified system that integrates **natural language processing, computer vision, and static code analysis techniques** within a single web-based platform.

The chapter synthesizes the key contributions, methodological decisions, system design choices, implementation strategies, and evaluation outcomes discussed throughout the report. It also highlights how the research objectives were achieved and how the proposed system contributes to both academic research and practical applications in plagiarism detection, content verification, and similarity analysis.

## **6.2 Summary of Chapter 1: Introduction and Problem Definition**

Chapter 1 established the **research background and motivation** for the project by examining the increasing prevalence of digital content creation across academic, industrial, and online environments. It highlighted the growing need for reliable similarity detection systems to support plagiarism detection, copyright protection, academic integrity, and content verification. Existing tools such as Turnitin, MOSS, and reverse image search engines were reviewed and shown to be effective only within isolated domains, lacking the ability to perform **cross-domain or unified similarity analysis**.

The chapter clearly articulated the **research problem**, identifying the absence of a single system capable of handling **multiple content modalities**—including text documents, images, and source code—within a consistent analytical framework. Based on this gap, the project objectives were defined, focusing on the development of a **scalable, modular, and semantically aware similarity detection platform**. The chapter also outlined the research scope, significance, and expected contributions, laying a strong conceptual foundation for the remainder of the study.

## **6.3 Summary of Chapter 2: Literature Review**

Chapter 2 presented a critical review of existing research related to **similarity detection across documents, images, and source code**. Traditional approaches such as **TF-IDF, n-grams, perceptual hashing, and AST-based analysis** were examined, and their strengths and limitations were discussed. While these methods demonstrated effectiveness in detecting surface-level similarities, the review highlighted their inability to capture deeper **semantic and contextual relationships**.

The chapter then explored modern approaches based on **deep learning and transformer architectures**, including Sentence-BERT for text similarity, convolutional neural networks such as ResNet for image feature extraction, and code representation models such as CodeBERT and graph-based techniques. Real-world tools and systems were compared through thematic summaries and comparative tables, emphasizing the transition from lexical and structural techniques to **embedding-based similarity detection**.

Finally, the literature review identified unresolved challenges such as computational cost, scalability, interpretability, multilingual bias, and ethical concerns. These findings directly informed the design decisions made in subsequent chapters, ensuring that the proposed system addressed both theoretical gaps and practical limitations identified in prior work.

## **6.4 Summary of Chapter 3: Methodology and Requirement Analysis**

Chapter 3 described the **research methodology and development strategy** adopted for the project. The study followed the **Design Science Research (DSR) paradigm**, which is well-suited for the creation and evaluation of IT artefacts that solve real-world problems. This paradigm guided the systematic progression from problem identification to artefact design, implementation, demonstration, and evaluation.

The system was developed using an **iterative prototyping model**, enabling continuous refinement based on testing and stakeholder feedback. Functional and non-functional requirements were elicited through literature analysis, benchmarking studies, and consultations with academic stakeholders. These requirements were categorized and formalized to ensure traceability between identified research gaps and implemented system features.

Key functional requirements included multi-modal similarity detection, support for file uploads and URL-based input, detailed similarity visualization, user authentication, session history management, and report generation. Non-functional requirements focused on usability, accuracy, performance, security, scalability, and maintainability. Together, these requirements ensured that the system was not only technically sound but also practical, user-friendly, and suitable for real-world deployment.

## **6.5 Summary of Chapter 4: System Design**

Chapter 4 translated the identified requirements into a detailed **system architecture and design specification**. A **multi-layered (N-tier) architecture** was adopted, separating the system into presentation, application, and persistence layers. This architectural approach enhanced modularity, scalability, and maintainability, allowing individual components to evolve independently.

The chapter detailed the design of core system components, including the document processing module, image processing module, code analysis module, similarity engine, and database layer. Each component was described in terms of its responsibilities, data flow, and interaction with other modules. Flowcharts and component diagrams illustrated how user inputs progressed through preprocessing, feature extraction, similarity computation, and result generation stages.

Algorithm and model selection were rigorously justified. Sentence-BERT was chosen for semantic text similarity, ResNet-50 for deep image feature extraction, and AST-based vectorization combined with token hashing for code similarity. Cosine similarity was consistently employed as the primary similarity metric across modalities due to its effectiveness in high-dimensional embedding spaces. The chapter also presented the database schema used to manage users, analysis history, and cached feature embeddings, supporting both performance optimization and auditability.

## **6.6 Summary of Chapter 5: Implementation and Evaluation**

Chapter 5 focused on the **implementation and evaluation** of the proposed system. The system was implemented using Python and the Flask web framework, integrating pre-trained deep learning models from established libraries such as SentenceTransformers, PyTorch, and Torchvision. The implementation supported multiple input types, including uploaded files, raw text, and web URLs, ensuring flexibility in real-world usage.

Evaluation procedures were designed to assess both **functional correctness and performance effectiveness**. Similarity computations for text, documents, URLs, code, and images were validated using cosine similarity over learned embeddings. Benchmark datasets such as STS-B for text similarity and ImageNet-pretrained feature extraction for images were used to assess semantic robustness. Performance testing demonstrated that the system could process typical inputs efficiently while maintaining high accuracy and stability.

The evaluation confirmed that the system successfully met its defined requirements, delivering accurate similarity scores, meaningful visual feedback, and reliable session management. Limitations related to computational cost and model interpretability were acknowledged, providing transparency and setting the stage for future enhancements.

## **6.7 Overall Contributions and Achievements**

Across Chapters 1 to 5, this project successfully delivered a fully functional **multi-modal similarity detection system** that integrates state-of-the-art techniques from multiple research domains into a single coherent and unified platform. A major contribution of the study is the design and implementation of a system capable of analyzing **text, document files, images, source code, and URL-based content** within one application, addressing a key limitation of existing single-modality tools. The project demonstrates the **practical integration of transformer-based language models and deep learning architectures**, including sentence-level embeddings for semantic text analysis and convolutional neural networks for image feature extraction, within a scalable web-based environment. Furthermore, the system adopts a **modular and extensible architecture**, enabling future enhancements such as additional modalities or alternative models to be incorporated with minimal disruption to existing components. The research methodology is grounded in the **Design Science Research paradigm**, ensuring that system development is informed by both theoretical insights and real-world problem-solving needs. Finally, comprehensive implementation and evaluation activities confirm that the system achieves a high level of **accuracy, reliability, and usability**, demonstrating its suitability for academic, professional, and research-oriented similarity detection applications.

## **6.8 Conclusion**

In conclusion, this project achieved its primary objective of bridging the gap between isolated similarity detection tools by developing a **comprehensive, multi-domain similarity analysis system**. Through rigorous research, thoughtful design, and systematic implementation, the study demonstrates that embedding-based and deep learning approaches can be effectively unified within a single scalable framework. The work provides a strong foundation for future research and practical deployment, contributing meaningfully to the fields of similarity detection, academic integrity, and intelligent content analysis.

# References

Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep

bidirectional transformers for language understanding. In Proceedings of the 2019 Conference of

the North American Chapter of the Association for Computational Linguistics: Human Language

Technologies, Volume 1 (Long and Short Papers) (pp. 4171–4186). Association for

Computational Linguistics. <https://aclanthology.org/N19-1423.pdf>

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020). CodeBERT: A

pre-trained model for programming and natural languages. In Findings of the Association for

Computational Linguistics: EMNLP 2020 (pp. 1536–1547). Association for Computational

Linguistics. <https://aclanthology.org/2020.findings-emnlp.139.pdf>

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Duan, N., & Zhou, M. (2021). GraphCodeBERT:

Pre-training code representations with data flow. In Proceedings of the 9th International

Conference on Learning Representations (ICLR 2021).

<https://openreview.net/pdf?id=jLoC4ez43PZ>

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In

Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp.

770–778).

https://www.cv-foundation.org/openaccess/content\_cvpr\_2016/papers/He\_Deep\_Residual\_Leaig\_CVPR\_2016\_paper.pdf

Reimers, N., & Gurevych, I. (2019). 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 (EMNLP-IJCNLP) (pp. 3982–3992). Association for Computational Linguistics.

<https://aclanthology.org/D19-1410.pdf>

Salton, G., Wong, A., & Yang, C. S. (1975). A vector space model for automatic indexing.

Communications of the ACM, 18(11), 613–620.

<https://dl.acm.org/doi/pdf/10.1145/361219.361220>

Schleimer, S., Wilkerson, D. S., & Aiken, A. (2003). Winnowing: Local algorithms for document

fingerprinting. In Proceedings of the 2003 ACM SIGMOD International Conference on

Management of Data (pp. 76–85). <https://dl.acm.org/doi/pdf/10.1145/872757.872770>

Zauner, C. (2010). Implementation and benchmarking of perceptual image hash functions.

(Master's thesis, Upper Austria University of Applied Sciences, Hagenberg).

<https://www.hackerfactor.com/papers/fingerprinting.pdf>

Manning, C. D., Raghavan, P., & Schütze, H. (2008). Introduction to Information Retrieval.

Cambridge University Press. (Specifically Chapter 6 for scoring and term weighting).

https://nlp.stanford.edu/IR-book/

Wang, Z., Bovik, A. C., Sheikh, H. R., & Simoncelli, E. P. (2004). Image quality assessment:

From error visibility to structural similarity. IEEE Transactions on Image Processing, 13(4),

600-612. <https://www.cns.nyu.edu/~lcv/ssim/papers/ssim.pdf>

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. International

Journal of Computer Vision, 60(2), 91-110. <https://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf>

Prechelt, L., Malpohl, G., & Philippsen, M. (2002). JPlag: Finding plagiarisms among a set ofprograms. Journal of Universal Computer Science, 8(11), 1016-1038. http://www.jucs.org/jucs\_8\_11/jplag\_finding\_plagiarisms\_among/jucs\_8\_11\_1016\_1038\_prechLt.pdf

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Duchesnay,

E. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12,

2825-2830. <http://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf>

**Foltynek, T., Meuschke, N., & Gipp, B. (2019).**

Academic plagiarism detection: A systematic literature review. *ACM Computing Surveys*, 52(6), 1–42.

<https://dl.acm.org/doi/pdf/10.1145/3345317>

Plagiarism in natural and programming languages: An overview of current tools and technologies. *Research Memorandum*.

<https://ir.shef.ac.uk/cloughie/papers/plagiarism_overview.pdf>

**Sharif Razavian, A., Azizpour, H., Sullivan, J., & Carlsson, S. (2014).**

CNN features off-the-shelf: An astounding baseline for recognition. *CVPR Workshops*.

<https://arxiv.org/pdf/1403.6382.pdf>

**Cer, D., Yang, Y., Kong, S., et al. (2018).**

Universal Sentence Encoder. *arXiv preprint*.

<https://arxiv.org/pdf/1803.11175.pdf>

**Baxter, I. D., Yahin, A., Moura, L., Sant’Anna, M., & Bier, L. (1998).**

Clone detection using abstract syntax trees. *Proceedings of ICSM*.

<https://ieeexplore.ieee.org/document/738528>

**Roy, C. K., & Cordy, J. R. (2007).**

A survey on software clone detection research. *Queen’s University Technical Report*.

<https://research.cs.queensu.ca/TechReports/Reports/2007-541.pdf>

**Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004).**

Design science in information systems research. *MIS Quarterly*, 28(1), 75–105.

<https://www.jstor.org/stable/25148625>

**Agirre, E., et al. (2016).**

SemEval-2016 Task 1: Semantic Textual Similarity. *ACL*.

<https://aclanthology.org/S16-1081.pdf>