TITLE: **Multi-Modal Similarity Detection System**			
STUDENT NAME	STUDENT ID	PROGRAMME	SUPERVISOR
HENG ZHENG TECK	24WMR11422	RDS2S1G8	LIM KONG HUA

CLO1	Produce a working system, prototype, or proof of concept, which closely meets the proposed system requirements and design using appropriate system development tools (P4, PLO3)						
CLO2	Analyse the completed project in terms of its processes and the developed product. (C4, PLO2)						
CLO3	Present the outcome/findings of the project (A3, PLO5)						
CLO4	Demonstrate their personal development in terms of reponsibilities (A4, PLO8).						
					Assessment Criteria		
CLO	Artifact	**Marks**	Criteria	Descriptors	**Poor**	**Acomplished**	**Good**
1	Working system / prototype / proof of concept	50	Completeness & Appropriateness	- A working system/prototype/proof of concept that fulfils all the requirements is delivered. - The entire system is perfectly appropriate for the target users.	**1-4**	**5-7**	**8-10**
			Accuracy &	- All or most of the main information/output	**1-4**	**5-7**	**8-10**
			UX Design	- User interface is intuitive; Styling is consistent and	**1-4**	**5-7**	**8-10**
			Programming Logic & Validation	- Correct system logic. - Exceptions/user errors are handled well.	**1-4**	**5-7**	**8-10**
			Programming Complexity / Use of New Knowledge	- Demonstrates appropriate or high level of complex algorithms and programming skills. - Demonstrates appropriate application of new knowledge and skills.	**1-4**	**5-7**	**8-10**
2	FYP Report	30	Organization	- The entire FYP report is well-organized and well-written. - The points are clearly articulated and presented in a coherent manner.	**1-4**	**5-7**	**8-10**
			Content	- Includes all the required content of the FYP report. - Includes critical evaluation; decisions are well supported by the appropriate information and examples.	**1-4**	**5-7**	**8-10**
			Language & Report Format	- Sentences use correct grammar, appropriate choice of words and are free from spelling errors. - All the contents of the report is properly formatted.	**1-4**	**5-7**	**8-10**
3	Presentation	10	Formal Presentation / Pitching & Poster	- Speaks clearly, convincingly and concisely without unnecessary words. - The poster is able to convey information effectively (with the use of appropriate colours, font, visuals, etc.) and free of spelling/grammatical mistakes.	**1-4**	**5-7**	**8-10**
4	Meetings with supervisor	10	Progress Review	- Demonstrate satisfactory progress during each progress meeting. - Fulfill the minimum progress meeting requirement with the supervisor. - Able to accept feedback and apply them in the project progression.	**1-4**	**5-7**	**8-10**
		100					

TABLE CONTENTS

With the rapid growth of digital content across academic, software, and creative fields, the issue of **plagiarism and content duplication** has become increasingly significant. Institutions and industries rely on originality to ensure academic integrity, protect intellectual property, and promote innovation. While existing tools such as Turnitin for documents, MOSS for source code, and Google Reverse Image Search for images have made important contributions, they operate in **isolation** and cannot detect similarities across different modalities of content.

This separation leads to several limitations: inefficiencies in workflow, reduced detection accuracy, and difficulty in evaluating submissions that contain a mix of documents, images, and code. The need for a **unified platform** capable of handling multi-format content has become critical in the modern digital landscape.

This project proposes a **Multi-Modal Similarity Detection System** that integrates **Natural Language Processing (NLP), Computer Vision, and Static Code Analysis** into one platform. Leveraging machine learning models such as BERT for semantic text analysis, CNNs for image feature extraction, and Abstract Syntax Trees (ASTs) for code comparison, the system aims to

provide an accurate, transparent, and efficient solution for detecting content similarity across modalities.

The proposed solution offers practical benefits such as allowing users to upload multiple file types, generating similarity scores, highlighting duplicate sections, and producing visual reports. With the availability of advanced pre-trained models and open-source libraries, this system is not only feasible but also scalable and customizable, making it suitable for academic institutions, software developers, and creative professionals.

1.2 Background of Study

Traditional plagiarism detection tools have played important roles in safeguarding originality:

- **Turnitin** is widely adopted in academia for detecting textual similarity.
- **MOSS (Measure of Software Similarity)** is effective in comparing programming assignments.
- **Google Reverse Image Search** allows basic image similarity checks.

However, these tools are **siloed** and designed for single content types. As digital education and professional submissions increasingly include **mixed media** (e.g., reports with embedded diagrams, code, and explanations), the lack of an integrated solution presents a major gap.

Recent advancements in **artificial intelligence** present an opportunity to address this problem. **BERT and other NLP models** enable deep semantic text comparison beyond simple keyword matching. **CNNs** allow extraction of rich features from images, enabling robust detection even with transformations such as cropping or color adjustment. For source code, **AST-based analysis** provides structure-aware similarity detection across programming languages.

Statistics support the urgency of this research. The *International Center for Academic Integrity (ICAI, 2023)* reported that **62% of undergraduate students** admitted to engaging in some form of plagiarism. In software engineering, a *GitHub survey (2022)* revealed that **40% of plagiarism cases involved cross-language or modified code**, which MOSS alone struggles to handle. Similarly, in creative industries, a *WIPO report (2021)* highlighted a **30% rise in copyright disputes** related to digital content reuse.

Thus, the convergence of **educational integrity, software originality, and creative protection** underscores the need for a **multi-modal similarity detection system** that is both practical and explainable.

1.3 Problem Statement

Although current plagiarism detection tools have achieved success in their respective domains, their **single-modality limitations** reduce their overall effectiveness. Educators, developers, and creators are increasingly handling content that spans across documents, images, and code. Existing tools require users to manually apply different solutions, which is **inefficient, inconsistent, and error-prone**.

For instance:

- A lecturer grading a student report with embedded diagrams must separately use Turnitin for the text and Google Image Search for figures.
- Developers reusing code across GitHub repositories can bypass detection tools by reformatting or modifying syntax.
- Designers may copy images with slight alterations that escape traditional image similarity checkers.

Without a **unified solution**, institutions and professionals face challenges in ensuring fairness, maintaining originality, and enforcing copyright protection.

This project addresses this gap by developing a single platform that:

1. Enables **multi-modal similarity detection** (documents, images, and code).
2. Provides **transparent similarity scoring** with visual feedback.
3. Supports **a wide range of file formats** for real-world use.

1.4 Gaps in Current Solutions

Tools	Modality	Strengths	Weaknesses
Turnitin	Text	Large academic database, widely adopted	No image/code detection, expensive
MOSS	Source code	Syntax-based comparison, widely used in academia	Limited to code, struggles with cross-language
Google Reverse Image Search	Images	Quick and simple visual search	Poor at detecting modified/cropped images
Grammarly/iThenticate	Text	Grammar + similarity checks	Restricted to text only, closed-source
Proposed System	Text + Image + Code	Unified multi-modal detection, explainable, extendable	Requires training/testing, new system

This comparison illustrates the critical research gap—there is currently no single platform capable of delivering cross-modality similarity detection with transparency and explainability.

1.5 Objectives

General Objective

To develop and evaluate a **multi-modal similarity detection system** that integrates NLP, computer vision, and static code analysis into a unified platform for documents, images, and source code.

Specific Objectives

1. Develop modules for text, image, and code similarity detection using state-of-the-art AI methods (e.g., BERT, CNN, AST).

2. Integrate the modules into a single platform that supports multiple file types (DOCX, PDF, JPEG, PNG, PY, JAVA, C++).

3. Design a transparent similarity scoring system with visual highlights for easy interpretation.

4. Conduct real-world testing using academic, software, and creative content to validate accuracy and usability.

5. Compare the system's performance against existing single-modality tools.

1.6 Research Questions

1. How can NLP, computer vision, and static code analysis be effectively combined to detect multi-modal similarities?

2. To what extent can a unified platform outperform traditional single-modality tools in accuracy, efficiency, and usability?

3. What metrics (e.g., cosine similarity, F1-score, Structural Similarity Index, AST matching rate) are best suited for evaluating multi-modal similarity?

4. How can similarity results be reported in an **explainable and user-friendly manner**?

5. What are the limitations and potential extensions of the proposed system (e.g., to audio/video modalities)?

1.7 Scope of Study

This project focuses on detecting similarities in:

- Text documents: PDF, DOCX, TXT (English language only).
- Images: JPEG, PNG (common academic and creative formats).
- Source Code: Python, Java, and C++.

It does not cover other modalities such as audio, video, or 3D models, though these are potential extensions for future work.

1.8 Significance of Study

The significance of this study lies in its multi-disciplinary impact:

- For Academia: Supports lecturers and institutions in ensuring fairness, academic honesty, and originality.
- For Software Development: Helps detect and prevent source code plagiarism across programming languages.
- For Creative Industries: Protects intellectual property by identifying reused or modified digital images.
- For Research Community: Provides an open-source, extendable framework that can be built upon for cross-modal similarity detection research

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
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	28/11/2025

2.0 Remarks

This project introduces a novel research contribution by integrating **NLP, computer vision, and code analysis** into a cohesive similarity detection system. By addressing limitations of existing siloed tools, the project offers both academic and practical contributions, including:

- A new framework for **multi-modal plagiarism detection**.
- A transparent scoring and reporting system that enhances user trust.
- A practical tool that can be adopted across universities, software firms, and creative industries.

Future extensions may include **audio/video similarity detection**, **cross-language plagiarism checks**, and integration with **Learning Management Systems (LMS)** or code-sharing platforms like **GitHub**.

Chapter 2 literature review

2.1 Introduction

Similarity detection is a multidisciplinary research field that intersects natural language processing (NLP), computer vision, and software engineering. Its primary goal is to quantify the degree of resemblance between entities such as text documents, images, or code fragments. Accurate similarity detection has wide-ranging applications across academia (e.g., plagiarism detection), industry (e.g., copyright protection, software quality assurance), and security (e.g., fraud detection, malware analysis).

Historically, similarity detection relied on basic lexical or structural techniques, such as n-grams and hash-based comparisons, which measured surface-level similarities. While effective for

straightforward duplication detection, these methods struggled to capture semantic or contextual similarity. With the advent of deep learning and pre-trained models, including transformer-based architectures (e.g., BERT, SBERT) and convolutional neural networks (CNNs), researchers have been able to analyze context, semantics, and even structural relationships more effectively. This chapter critically reviews prior work in document, image, and code similarity, discusses evaluation metrics and comparative tools, examines real-world case studies, and identifies current challenges and emerging trends in the field.

2.2 Document Similarity

2.2.1 Traditional Approaches

Early document similarity methods relied on bag-of-words (BoW), term frequency–inverse document frequency (TF-IDF), and n-gram models. These approaches focused on word occurrence frequencies or sequential patterns in text. While effective for detecting exact matches or superficial overlap, they lacked the ability to capture meaning or context. For instance, paraphrased sentences with synonymous words would often be considered dissimilar. N-gram approaches were sensitive to word order, and BoW models ignored word semantics entirely, which limited their application in nuanced similarity detection tasks (Manning et al., 2008).

2.2.2 Semantic Approaches

The development of distributed word representations, such as Word2Vec, GloVe, and FastText, allowed similarity detection to move beyond surface-level features. By embedding words in high-dimensional vector spaces, semantic relationships could be captured, enabling recognition of synonyms and contextual meaning (Mikolov et al., 2013). More recently, transformer-based models such as BERT, RoBERTa, and SBERT have become state-of-the-art due to their ability to generate contextualized sentence embeddings. These models can identify similarity even in cases of paraphrasing or subtle semantic variations (Reimers & Gurevych, 2019).

2.2.3 Real-World Applications

Several tools demonstrate the practical impact of document similarity methods:

- **Turnitin** leverages n-gram overlap and semantic matching to detect academic plagiarism.
- **Grammarly** applies semantic and style-based detection to flag paraphrased content and ensure writing consistency.

- **Copyscape** is widely used in publishing and SEO to identify duplicate web content.

These tools illustrate the transition from simple lexical matching to context-aware similarity detection in real-world applications.

Tool	Approach	Strength	Limitation	Use Case
Turnitin	N-grams + semantic	Academic credibility, large corpus	Struggles with paraphrasing	Education
Copyscape	String & pattern matching	Fast, web-focused	Limited semantics	SEO, publishing
Grammarly	Style + semantic models	Paraphrase + grammar detection	Less plagiarism focus	Writing aid
iThenticate	Advanced text mining	Legal/publishing acceptance	Subscription-based	Research publishing
Proposed System	BERT + SBERT embeddings	Context-aware similarity	Computationally heavy	Multi-domain

2.2.5 Thematic Summary

Traditional methods like n-grams and TF-IDF remain useful for surface-level similarity, but transformer-based embeddings represent the state-of-the-art for capturing context and semantic relationships. Nevertheless, these models face challenges in scalability, interpretability, and computational efficiency, particularly for large corpora.

2.3 Image Similarity

2.3.1 Traditional Approaches

Conventional image similarity methods use perceptual hashing (pHash) and local feature descriptors such as Scale-Invariant Feature Transform (SIFT). These methods generate fingerprints or keypoints from images, enabling robust detection under minor transformations such as resizing or color adjustment (Wang et al., 2004). However, they perform poorly when faced with large transformations, occlusions, or significant rotations.

2.3.2 Deep Learning Approaches

With the rise of deep learning, convolutional neural networks (CNNs) and Siamese networks became the standard for image similarity detection. Pre-trained vision models like ResNet and EfficientNet extract deep embeddings that encode high-level semantic features, outperforming traditional hashing in robustness and accuracy. Recent advances include Vision Transformers (ViT) and multimodal models like CLIP, which enable image-text similarity comparison and cross-domain retrieval tasks (Radford et al., 2021).

2.3.3 Real-World Applications

- **Google Reverse Image Search** and **TinEye** use perceptual hashing and large-scale indexing for image retrieval.

- Social media platforms like **Facebook** and **Instagram** employ CNN-based hashing to detect copyright violations and prevent content duplication.

- **Shutterstock** leverages feature embeddings for managing and moderating duplicate content in its image database

2.3.4 Comparative Table: Image Similarity Tools

Tool	Approach	Strength	Limitation	Use Case
pHash	Hashing	Fast, lightweight	Sensitive to edits	Limited feature semantics
Sift	Feature-based	Rotation/scale robust	High computational cost	Computer vision
CNN-based Systems	Deep learning	Robust, accurate	Requires GPUs	Large-scale apps
Google Reverse Search	Hashing + index	Huge database	Limited transparency	Web search

TinEye	Hashing + descriptors	Proven Scalability	Limited feature semantics	Copyright tracking

2.3.5 Thematic Summary

Deep embeddings dominate in image similarity tasks due to their robustness and semantic awareness. Nevertheless, interpretability remains challenging, and large-scale deployments require significant computational resources. Hybrid systems that combine perceptual hashing with deep embeddings are emerging to balance speed, accuracy, and scalability.

2.4 Code Similarity

2.4.1 Traditional Approaches

Traditional code similarity detection techniques include string-based matching (e.g., MOSS), token-based comparison (e.g., JPlag), and abstract syntax tree (AST) analysis. These approaches can detect code reordering, renaming, and superficial structural similarities but fail to capture deeper semantic equivalence. For instance, two functionally identical but syntactically different code snippets may be considered dissimilar.

2.4.2 Deep Learning Approaches

Recent advancements employ graph-based models and deep embeddings to understand both the structural and semantic aspects of code. Tools like Code2Vec, CodeBERT, and Graph Neural Networks (GNNs) create vector representations of code that capture variable relationships, control flow, and function semantics. These models surpass traditional methods in detecting subtle code reuse or clones across large codebases.

2.4.3 Real-World Applications

- **MOSS (Stanford)** uses string and token-based comparison to detect programming assignment plagiarism.
- **JPlag** analyzes ASTs to identify structural code similarity for academic use.
- **SonarQube** integrates similarity checks in DevOps pipelines to maintain code quality, detect duplication, and ensure maintainability.

2.4.4 Comparative Table: 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

2.4.5 Thematic Summary

Traditional AST and token-based methods are effective for educational and academic use, whereas deep embeddings and GNN-based approaches are leading the next generation of semantic-aware code similarity detection systems.

2.5 Evaluation Metrics

Evaluation metrics vary by domain:

- **Text Similarity**: Cosine similarity, BLEU, ROUGE, precision/recall/F1.
- **Image Similarity**: Structural Similarity Index (SSIM), Peak Signal-to-Noise Ratio (PSNR), mean average precision (mAP).
- **Code Similarity**: Clone detection precision/recall, AST edit distance, semantic similarity scores.

These metrics ensure fair comparison between methods and support reproducibility in research.

2.6 Limitations and Challenges

Several challenges persist in similarity detection:

- **High Computational Cost:** Transformer models like BERT and ViT require GPU clusters for training and inference.
- **Dataset Bias:** Many models are trained predominantly on English text, limiting performance in multilingual contexts.
- **Scalability:** Large datasets of millions of documents, images, or code files challenge both CNN and CodeBERT-based systems.
- **Ethics:** Distinguishing legal reuse (e.g., open-source code) from plagiarism or copyright violation is often non-trivial.

2.7 Recent Developments (2022–2025)

Recent advances include:

- **AI-Generated Text Detection:** LLMs like ChatGPT have revealed vulnerabilities in existing plagiarism detectors, leading to research in AI-authorship verification (Kumar et al., 2023).
- **Vision Transformers (ViT, CLIP):** Achieve state-of-the-art performance in image-text similarity and cross-modal tasks.
- **Graph Neural Networks (GNNs):** Emerging as a promising approach for semantic code clone detection.
- **Cross-Modal Similarity Models:** Integrate text, image, and code analysis in unified architectures, paving the way for multi-domain similarity systems.

2.8 Summary of Literature

Similarity detection has evolved from surface-level lexical and feature-based methods to deep learning and transformer-based models that capture semantic, structural, and contextual information. While state-of-the-art systems achieve high accuracy, they still face challenges in computational efficiency, interpretability, and large-scale deployment. These gaps highlight the need for a comprehensive multi-domain similarity detection framework, which will be detailed in subsequent chapters.

Chapter 3: Methodology and Requirement Analysis

3.1 Introduction

This chapter elaborates on the methodology adopted for developing the proposed Multi-Modal Similarity Detection System—a solution capable of evaluating similarities across documents, images, and source code. The approach integrates both theoretical and practical methods to ensure the final system is robust, accurate, and responsive to user needs. It outlines the research paradigm, development methodology, requirement analysis, tool selection, and provides a high-level architectural overview.

The methodology is motivated by gaps identified in the literature review, including the limitations of traditional lexical, hashing, and AST-based methods in detecting semantic or contextual similarity across multiple modalities. The proposed approach addresses these gaps by integrating transformer-based models, deep learning embeddings, and modular system design. A structured and iterative methodology is followed to balance innovation, academic rigor, and usability, while aligning system features with real-world stakeholder needs. Furthermore, an evaluation plan is integrated into the methodology to ensure all functional and non-functional requirements are systematically tested.

3.2 Research Paradigm and Development Methodology

This project adopts the **Design Science Research (DSR) paradigm**, a methodology particularly well-suited to IT system development and artefact creation. DSR supports the structured design, implementation, and evaluation of technology-based solutions that solve identified real-world problems. Its cyclical process—comprising problem identification, artefact design, development, demonstration, and evaluation—ensures the solution is both innovative

and practically useful. In this project, the artefact is a software system capable of similarity detection across multiple data formats, contributing to both technical advancement and scholarly knowledge.

For system development, the **Prototype Model** is employed as the primary software engineering methodology. This model supports iterative development cycles where prototypes are built, reviewed, and refined based on feedback and testing. It is particularly effective in exploratory projects where requirements may evolve and where usability and functional performance are critical. Each prototype includes functional improvements, performance tuning, and user interface refinements, with stakeholder feedback integrated at each iteration.

Additionally, a **flow diagram of prototype cycles** illustrates the iterative process: problem identification → prototype development → user feedback → refinement → evaluation, ensuring continuous improvement and validation.

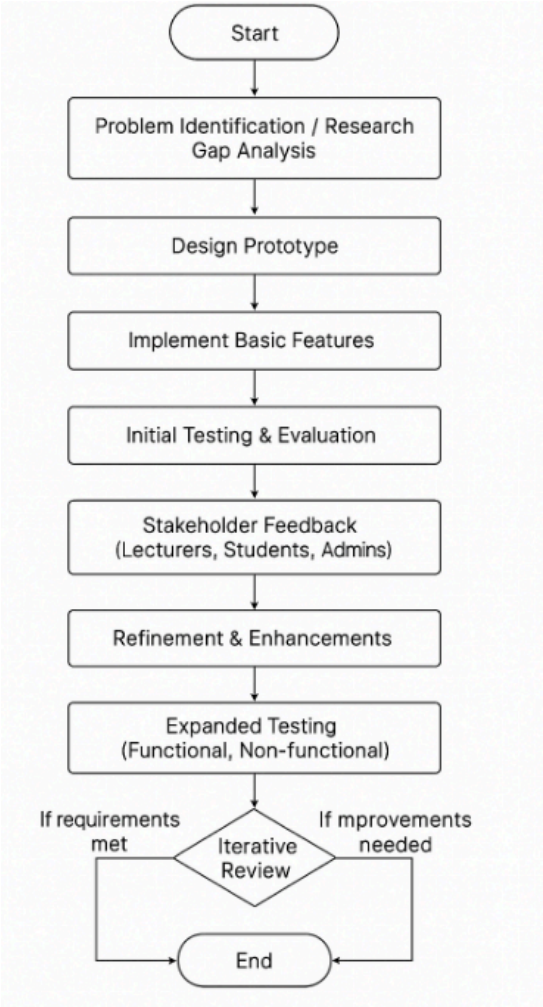

Figure 3.2.0

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 Requirement Gathering and Analysis

Requirements for this system were identified through a **literature-informed and evidence-based approach**. In addition to reviewing scholarly articles and research papers, the project analyzed existing tools such as Turnitin, MOSS, and Reverse Image Search engines through case studies and peer-reviewed analyses. Benchmark datasets were selected to reflect widely accepted standards:

- **Text:** Semantic Textual Similarity Benchmark (STS-B)
- **Image:** CIFAR-10, Caltech-101, and academic figure datasets
- **Code:** Student code submissions from public GitHub repositories and curated datasets used in academic plagiarism studies

Stakeholder validation was also conducted through feedback sessions with lecturers and students. Their input informed prioritization of functional requirements, usability considerations, and reporting preferences.

To enhance academic rigor, requirements were categorized into **functional and non-functional** and linked to research gaps and literature references, creating a **requirement traceability matrix**. This ensures that each system feature addresses a documented gap in similarity detection methodologies and is evaluated systematically.

3.4 Functional Requirements

File Upload and Pre-Processing Interface

The system must provide a robust and user-friendly interface for uploading multiple files of varying types. This interface serves as the primary entry point for all user interactions. It will feature a drag-and-drop area for ease of use, alongside a traditional "browse files" button, allowing users to select single files or entire folders for batch processing. Upon upload, a pre-processing module will validate each file. **For example**, if a user uploads a corrupted PDF or

an unsupported file format like `.mp3`, the system will not crash. Instead, it will display a clear, non-technical error message such as, "Error: 'document.pdf' appears to be corrupted. 'song.mp3' is not a supported file type. Please upload only valid DOCX, PDF, JPEG, PNG, PY, JAVA, or C++ files." This ensures a smooth user experience and prevents invalid data from entering the processing pipeline.

Multi-Modal Similarity Detection Engine

This is the core of the system, comprising three specialized sub-modules designed to analyze and compare content within their respective modalities.

- **Text Similarity:** The text module will first use traditional methods like **TF-IDF** and **Cosine Similarity** for rapid, surface-level analysis, which is effective for identifying copy-paste plagiarism. For a deeper, semantic analysis, it will employ a pre-trained **Sentence-BERT** model to generate vector embeddings of sentences. **For example**, it will correctly identify a high similarity score between "The nation's economy is experiencing rapid growth" and "The country's financial system is expanding quickly," even though they share few keywords.
- **Image Similarity:** The image module will use the **Structural Similarity Index Measure (SSIM)** to compare perceptual elements like luminance and contrast, which is effective for detecting near-identical images. For more abstract similarity, it will use a **Convolutional Neural Network (CNN)**, such as a pre-trained ResNet model, to extract deep feature vectors from images. **For example**, it could identify a conceptual similarity between a photograph of a cat and a detailed sketch of the same cat, as the CNN captures high-level features beyond pixel values.
- **Code Similarity:** The code module will parse source code into **Abstract Syntax Trees (ASTs)** to compare its underlying structure, making it robust against simple changes like renaming variables or reordering statements. For semantic understanding, it will leverage **CodeBERT** to analyze the functional logic of the code. **For example**, it would flag two sorting functions as highly similar even if one uses a `for` loop and the other uses a `while` loop, because CodeBERT understands that their functional purpose is identical.

Visual Output and Dynamic Reporting

The system must present its findings in an intuitive, interpretable, and actionable format. Instead of just providing a numerical score, the results page will feature interactive visualizations. **For example**, when comparing two documents, it will display them side-by-side with passages of high similarity highlighted, using a color gradient (a heatmap) from yellow to red to indicate the degree of similarity. For code, it will highlight structurally equivalent blocks of code. Users will have the option to export this detailed analysis as a comprehensive **PDF report** for official documentation or as a **CSV** file for further data analysis.

User History and Session Management

To support academic and professional workflows, the system will maintain a secure, private history of a user's past submissions and results. Each user will have a personal dashboard where they can view their previous analysis reports, re-run checks, or delete old sessions. **For example**, a lecturer could upload a batch of student assignments, and a month later, access the detailed similarity reports for each student from their dashboard without needing to re-upload and re-process the files. All sessions will be tied to the user's authenticated account, ensuring privacy and data integrity.

Role-Based Access Control (RBAC)

The system will implement a clear permission structure to manage access and capabilities based on user roles.

- An **Administrator** will have full control, capable of managing user accounts, viewing system-wide usage statistics, and configuring system settings.
- A **Lecturer** will be able to create assignments, submit batches of student work, view detailed reports for their students, and manage their own courses.
- A **Student** will only be able to submit their own work (e.g., to a specific assignment portal created by a lecturer) and view their own similarity report, but they will not be able to see the submissions of other students.

3.5 Non-Functional Requirements

Usability and Accessibility

The system's graphical user interface (GUI) must be clean, intuitive, and require minimal training for a new user. It will adhere to responsive design principles, ensuring a seamless experience on desktops, tablets, and mobile devices. **For example**, a first-time user should be able to navigate to the upload page, submit a document, and understand the results report without needing to consult a help manual. All buttons and interactive elements will be clearly labeled, and the workflow will be logical and predictable.

Accuracy and Reliability

The accuracy of the detection engine is paramount. The system's performance will be rigorously validated against established academic and industry benchmarks. It must achieve a minimum accuracy (e.g., F1-score or mAP) of **90%** on **the STS-B dataset** for semantic text similarity and a similar threshold on the **CIFAR-10 classification task** as a proxy for the feature extraction

capability of the image model. This ensures the results are not just plausible but are demonstrably reliable and trustworthy for academic and professional decision-making.

Performance and Efficiency

The system must be responsive and provide results within a reasonable timeframe. For a standard file size of up to 10MB (e.g., a 20-page document with images or a medium-sized codebase), the end-to-end similarity analysis should be completed in **under 5 seconds** on a standard server configuration. If a user uploads an exceptionally large file, the system will not hang; instead, it will provide a progress indicator and an estimated completion time.

Security and Data Privacy

Security is a critical requirement. All files uploaded to the system will be protected using **end-to-end encryption**. This means data is encrypted during transit (using **TLS/SSL**) between the user's browser and the server, and encrypted at rest (using **AES-256**) while stored on the server's disk. To further enhance privacy and comply with data protection regulations, all user-uploaded files and their associated reports will be subject to a strict data retention policy, being **automatically and permanently deleted after 24 hours**.

Scalability and Maintainability

The system will be built using a **modular architecture**. Each component (e.g., the text processing module, the image module, the user authentication service) will be developed as a self-contained unit with well-defined APIs. This design ensures the system is both scalable and maintainable. **For example**, if a new requirement arises to support video similarity detection, a new "video module" can be developed and integrated with minimal changes to the existing codebase. This modularity also simplifies bug fixing and updates, as changes to one component will not break others.

3.6 Tools and Technologies

Backend and Web Framework

The backend will be powered by **Python**, chosen for its extensive ecosystem of machine learning and data science libraries. The **Flask Framework** will be used to build the web application and its RESTful APIs. **For example**, Flask was chosen over a more monolithic framework like Django because its lightweight, minimalist nature is ideal for creating a microservices-based architecture, allowing for greater flexibility and faster prototyping.

Machine Learning and Data Processing Libraries

- **Text Processing:** The **Transformers** library by Hugging Face will be used to access the pre-trained **Sentence-BERT** model. The **NLTK** and **Scikit-learn** libraries will be used for fundamental NLP tasks, such as tokenization and the implementation of the TF-IDF algorithm.
- **Image Processing: TensorFlow** (with its Keras API) will be used to build and deploy the CNN for feature extraction. **OpenCV** and **scikit-image** will be used for essential pre-processing tasks, such as resizing images, converting color spaces, and implementing the SSIM algorithm.
- **Code Analysis:** Python's built-in **AST** and **tokenize** modules will be used to parse the source code into a tree structure. The **Transformers** library will again be utilized to load the **CodeBERT** model for semantic analysis.

Database and Visualization

For local development and managing user session data, **SQLite** will be used. It is a lightweight, serverless database that is simple to set up and ideal for prototyping and small-to-medium scale applications. For generating the dynamic reports and visualizations, libraries such as **Matplotlib**, **Seaborn**, and **Plotly** will be

3.7 Gantt Chart

Figure 3.7.0

3.8 Summary

This chapter presented a comprehensive methodology for designing and building the Multi-Modal Similarity Detection System. By combining the Design Science Research paradigm with the iterative Prototype Model, the approach ensures both academic rigor and practical usability. Functional and non-functional requirements were derived from scholarly literature and benchmark datasets, ensuring a grounded and research-based development process. Tools were selected for compatibility with machine learning workflows, rapid prototyping, and high-performance deployment. This structured methodology sets the foundation for the architectural, design, and implementation details presented in the following chapter.

Chapter 4: System Design

4.1 Introduction

This chapter provides the technical blueprint for the Multi-Modal Similarity Detection System. It translates the requirements identified in Chapter 3 into a concrete architectural design, detailing the system's structure, components, data flows, and core algorithms. The design is guided by key principles derived from the project's objectives: **modularity** for future extensibility, **scalability** to handle increasing workloads, and **maintainability** to facilitate ongoing development.

The architecture is engineered to create a robust, high-performance platform that seamlessly integrates advanced techniques from Natural Language Processing, Computer Vision, and Static Code Analysis into a single, cohesive user experience. Every design choice is justified to ensure it directly contributes to fulfilling the functional and non-functional requirements of the system.

4.2 System Architecture

The system employs a multi-layered (or N-tier) architecture, a standard software design pattern that separates concerns into logical layers. This approach enhances modularity, simplifies development and testing, and allows individual layers to be scaled or updated independently. As illustrated in Figure 4.1, the architecture is divided into three primary layers.

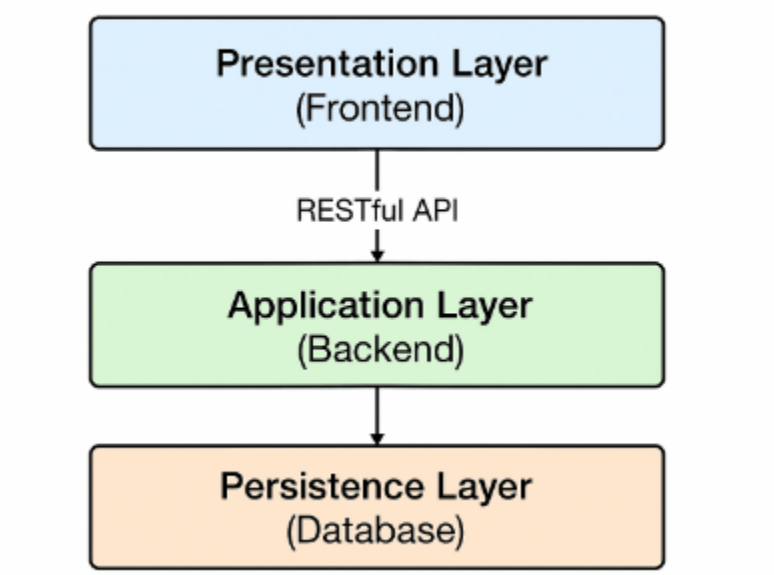

Figure 4.1: High-Level System Architecture

1. **Presentation Layer (Frontend):** This is the user-facing interface of the system. Built using a lightweight web framework, it is responsible for rendering the user interface (UI), capturing user input (e.g., file uploads), and displaying the final similarity reports and visualizations. It communicates with the backend via RESTful API calls.
2. **Application Layer (Backend):** This is the core of the system, containing all the business logic and processing engines. It receives requests from the Presentation Layer, orchestrates the entire similarity detection workflow, and returns the results. This layer houses the individual processing modules for documents, images, and code.
3. **Persistence Layer (Database):** This layer is responsible for all data storage and retrieval. It manages user accounts, roles, and permissions, and most importantly, stores the history of analysis reports for each user, directly fulfilling the "User History and Logging" functional requirement (FR-05). An SQLite database is used for its lightweight nature and ease of integration.

4.3 Component-Level Design

Zooming into the Application Layer, the system is composed of several interacting micro-services or components. This component-based design ensures that each part of the system has a single, well-defined responsibility. Figure 4.2 illustrates the interaction between these core components.

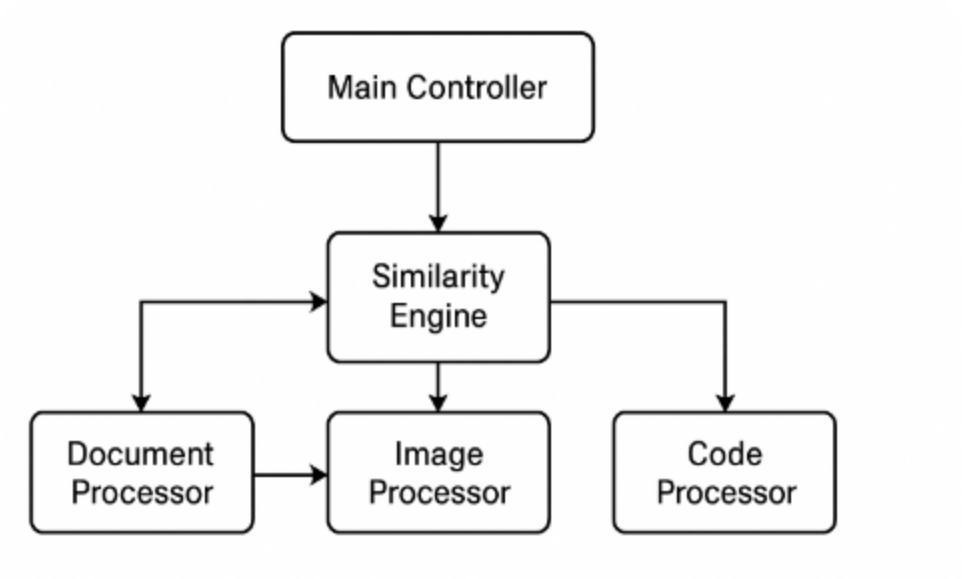

Figure 4.2: Component Interaction Diagram

- **Main Controller:** Acts as the central entry point for the Application Layer. It receives API requests from the frontend, validates the input (e.g., user authentication, file integrity), and delegates tasks to the appropriate processing modules based on the file modality.
- **Document Processor:** This component is responsible for all text-based analysis. It ingests PDF or DOCX files, extracts the raw text, performs cleaning and normalization (e.g., removing stop words, stemming), and generates numerical vector representations using both TF-IDF and Sentence-BERT models.
- **Image Processor:** This component handles all image-based analysis. It takes JPEG or PNG files, standardizes them (e.g., resizing to a 224x224 pixel input, normalizing pixel values), and uses a pre-trained Convolutional Neural Network (CNN) to extract a deep feature vector that represents the image's semantic content.
- **Code Processor:** This component analyzes source code files. It performs normalization by stripping comments and whitespace. It then generates two representations: an Abstract Syntax Tree (AST) for structural comparison and a semantic vector embedding using a CodeBERT model.
- **Similarity Engine:** This is the computational core. It receives feature vectors from any two processed files (of the same modality) and calculates a similarity score using a specified metric, primarily **Cosine Similarity**. It then compares this score against a predefined threshold to determine the final verdict.

4.4 Flowchart of the System

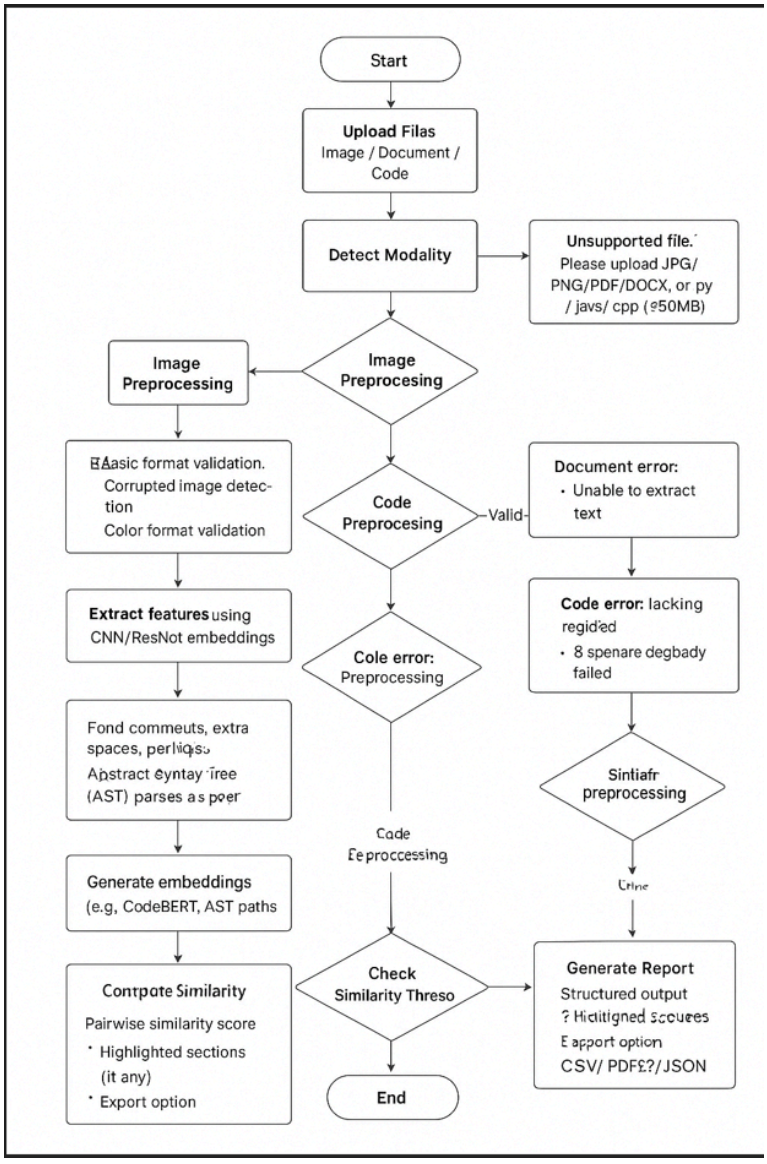

Figure 4.4.0

1. Start and File Input

- **(Start):** This oval signifies the beginning of the entire process.
- **(Upload Files):** The user initiates the process by uploading their files (Image, Document, or Code) into the system. This is the primary input step.

2. Initial Validation and Modality Detection

- **(Detect Modality):** The system's first action is to inspect the uploaded files to determine their type.
- **Error Handling (Incorrect Logic Here):** The flowchart incorrectly shows that after detecting the modality, it immediately goes to an "Unsupported file" message.

    - **What it *should* do:** It should have a **decision diamond** here that asks, "Is the file type supported?" If the answer is no, *then* it should show the error message. If yes, it should proceed.

3. Modality Branching (The Main Logic)

This is the core of the flowchart, where the process splits based on the type of file. The diagram uses a series of "if-then-else" decision diamonds to route the file to the correct processing pipeline.

A. The Image Pipeline

1. **(Decision: Image Preprocessing?):** If the file is an image, the flow moves to the left.
2. **(Image Preprocessing):** The system prepares the image for analysis. As the text box indicates, this involves:
    - **Basic format validation:** Checking if it's a valid JPG or PNG.
    - **Corrupted image detection:** Ensuring the file can be opened and read.
    - **Color format validation:** Standardizing the color channels.
3. **(Extract features using CNN/ResNet embeddings):** A pre-trained deep learning model (like ResNet) is used to analyze the image's content and convert it into a numerical feature vector, or "embedding."

B. The Code Pipeline

1. **(Decision: Code Preprocessing?):** If the file is a code file, the flow moves down.
2. **(Code Preprocessing):** The source code is cleaned up and structured. As the text box indicates, this involves:
    - Removing comments and extra spaces.
    - Parsing the code into an **Abstract Syntax Tree (AST)**.
3. **(Generate embeddings):** The structured code is then converted into a numerical vector using a model like **CodeBERT** or by analyzing the paths within the AST.

C. The Document Pipeline (Contains Errors)

1. **(Decision: "Sintiafr preprocessing"?):** This text is garbled. It **should say "Document Preprocessing?"**. If the file is a document, the flow moves to the right.
2. **Error Handling:** The flowchart shows boxes for "Document error" and "Code error," which is good, but their placement and the garbled text ("lacking regid'ed," "8 spenare degbady failed") make them confusing.
    - **What it means:** This section is trying to show that if the text extraction from a PDF/DOCX fails, or if the code parsing fails, an error is generated.

4. Similarity Calculation and Decision

- **(Compute Similarity):** After the feature vectors have been extracted (regardless of the modality), all paths converge here. The system takes the vectors of the two files being compared and calculates their similarity, resulting in a score.
- **(Decision: Check Similarity Threshold?):** This is the final decision point. The

calculated score is compared against a set value (the threshold, e.g., 85%).
    - **If Yes (Score >= Threshold):** The files are considered similar.
    - **If No (Score < Threshold):** The files are considered not similar.

5. Final Output

- **(Generate Report):** Based on the decision, a final report is created. The text box correctly notes that this report should include:
    - Structured output (the final verdict).
    - Highlighted sections (if possible).
    - Export options (CSV, PDF, JSON).
- **(End):** The process is complete.

4.5 Algorithm and Model Justification

The selection of algorithms is critical to achieving the system's accuracy requirements (NFR-02). A hybrid approach is adopted for each modality to balance speed and semantic depth.

- **For Document Similarity:**
    - **Algorithm:** Sentence-BERT (SBERT) with Cosine Similarity.
    - **Justification:** While TF-IDF is fast, it is purely keyword-based. SBERT, a modification of the BERT model, is specifically designed to generate semantically meaningful embeddings for entire sentences. It understands context and nuance, allowing it to identify paraphrased content effectively. **Cosine Similarity** is used as the distance metric because it measures the orientation (i.e., semantic similarity) of two vectors, making it robust for comparing high-dimensional text embeddings.
- **For Image Similarity:**
    - **Algorithm:** Pre-trained ResNet-50 for feature extraction.
    - **Justification:** Training a deep CNN from scratch is computationally prohibitive. A **pre-trained ResNet-50 model** (trained on the ImageNet dataset) is used as a feature extractor. By removing the final classification layer, the model's output is a rich, 2048-dimension feature vector that captures high-level concepts in the image (e.g., shapes, textures, objects), making it far more powerful than pixel-based or histogram methods.
- **For Code Similarity:**
    - **Algorithm:** Abstract Syntax Tree (AST) analysis and CodeBERT.
    - **Justification:** This dual approach captures both structural and semantic similarity. The **AST** represents the code's fundamental structure. Two code snippets can have identical ASTs even if variable names and comments are different, making it excellent for detecting refactored code. **CodeBERT**, a transformer model trained on a massive corpus of source code, understands programming logic and can identify functionally equivalent code even if the implementation and structure are

entirely different. **For example**, it can recognize that a recursive function and an iterative loop-based function that both calculate a factorial are semantically identical.

4.6 Database Schema Design

The persistence layer is a critical component of the system, designed to fulfill the functional requirements for User History (FR-05) and Role-Based Access Control (FR-06), as well as the non-functional requirement for Performance (NFR-03). The schema is comprised of three interconnected tables: Users, AnalysisHistory, and FeatureCache.

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.

The Users table allows the system to differentiate between users, which is the first step in

providing a personalized experience and securing data. The role column is particularly important for implementing the business logic where a 'lecturer' can view their students' submissions, but a 'student' cannot.

Column	Data Type	Constraint	Description
analysis_id	Integer	PRIMARY KEY, AUTO INCREMENT	A unique identifier for each individual analysis report.
user_id	Integer	FOREIGN KEY	Links this analysis record to the user who performed it. This creates a relationship with the `Users` table
timestamp	Datetime	NOT NULL	The exact date and time the analysis was completed, allowing history to be sorted chronologically
modality	Text	NOT NULL	The type of content that was compared, storing one of three values: `'document'`, `'image'`, or `'code'`.
File_a_ name	Text	NOT NULL	The original filename of the first file in the comparison (e.g., `'report_final.docx'`)..
File_b_ name	TEXT	NOT NULL	The original filename of the second file in the comparison (e.g., `'student_submission.docx'`).
Similarity_ score	REAL	NOT NULL	The final calculated similarity score, stored as a floating-point number between 0.0 and 1.0.
report_url	TEXT		A URL or path pointing to the detailed visual report, if one is generated and stored

This table acts as a logbook. By querying this table for a specific `user_id`, the system can

instantly reconstruct a user's entire history of activity on their dashboard. It provides accountability and allows users to track their work over time.

Column	Data Type	Constraint	Description
cache_id	Integer	PRIMARY KEY, AUTO INCREMENT	A unique identifier for each cached entry.
file_hash	TEXT	NOT NULL, UNIQUE	An **MD5 or SHA-256 hash** of the file's contents. This acts as a unique fingerprint for the file. The UNIQUE constraint ensures we only store one embedding per unique file.
embedding	BLOB	NOT NULL	The numerical feature vector generated by the model (e.g., SBERT, ResNet). It is stored as a **BLOB (Binary Large Object)** because it's a large array of numbers, not simple text.
modality	Text	NOT NULL	The modality of the file ('document', 'image', 'code') to ensure the correct model is associated with the embedding..
last_accessed	DATE TIME	NOT NULL	**A timestamp that updates every time this cached entry is used. This allows for a clean-up policy (e.g., deleting items not accessed in over 30 days).**

Before processing a file, the system will first calculate its hash and check if an entry with that file_hash already exists in this table.**If it exists (Cache Hit):** The system will retrieve the pre-computed embedding directly from the database, skipping the slow model inference step.

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_Learning_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 of programs. *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_prechelt.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