TRIBHUVAN UNIVERSITY

INSTITUTE OF SCIENCE AND TECHNOLOGY

MADAN BHANDARI MEMORIAL COLLEGE

A PROJECT REPORT ON

Abstractify

Submitted by:
Firoj Paudel (79011003)
Nilima Mainali (79011011)
Priyanka Thapa (79011017)
Subodh Ghimire (79011027)

SUBMITTED TO:
Er. Ram Kumar Basnet
SOFTWARE ENGINEERING
(CSC 375)

December 25, 2025

CERTIFICATE OF APPROVAL

This is to certify that the project entitled **"Abstractify"** prepared by **Firoj Paudel, Nilima Mainali, Priyanka Thapa, Subodh Ghimire** as a part of the coursework in the Department of Computer Science and Information Technology is a record of original work carried out under our supervision.

External Examiner
Institute of Science and Technology
Tribhuvan University

Er. Ram Kumar Basnet
Supervisor
Madan Bhandari Memorial College

Phul Babu Jha
Head of CSIT Department
Madan Bhandari Memorial College

Acknowledgment

We would like to express our sincere gratitude to the Department of Computer Science and Information Technology, Madan Bhandari Memorial College, for providing an encouraging academic environment that fostered our growth in the field of Information Technology. The resources and mentorship offered by the department played a crucial role in the successful completion of this project.

We are especially thankful to our supervisor, **Er. Ram Kumar Basnet**, for his valuable guidance, continuous support, and encouragement throughout the project. His expertise and feedback were instrumental in refining our ideas and achieving our objectives.

This project has been a significant learning experience, allowing us to deepen our understanding of natural language processing, web application development, and real-world AI driven summarization systems.

We would also like to thank our Head of Department, **Phul Babu Jha**, for his administrative support and encouragement. Our special thanks go to all staff members of the CSIT department who directly and indirectly extended their hands in making this project work a success.

With respect,
Firoj Paudel
Nilima Mainali
Priyanka Thapa
Subodh Ghimire

Abstract

This project presents the complete development of **Abstractify**, an intelligent web-based arXiv research paper summarizer designed to address the challenge of information overload faced by students, researchers, and educators. Built using the *Flask* framework and leveraging advanced natural language processing models, the system accepts research papers via PDF upload or arXiv URL and generates concise, readable summaries with properly rendered LaTeX equations.

The core summarization engine intelligently selects between the preferred **Gemini 2.0 Flash API** and a local fallback model (BART or T5) to ensure reliability and high-quality abstractive summaries. The system achieves efficient performance with response times typically under 10 seconds, while maintaining accuracy and context preservation. Key features include text extraction and cleaning using *pdfplumber*, LaTeX rendering via *MathJax*, persistent storage of summaries in an SQLite database for history access, and user-friendly export options (copy to clipboard and download as Markdown). The project emphasizes usability, scalability, and robustness through fallback mechanisms and modular design.

Abstractify was developed as part of the Software Engineering coursework to provide a practical AI-powered tool that saves time and enhances comprehension of complex academic content. The system has been tested on a diverse set of arXiv papers and demonstrates strong generalization across scientific domains, making it a valuable resource for academic communities.

Table of Contents

List of Figures

List of Tables

List of Symbols and Acronym

AI Artificial Intelligence. 3

API Application Programming Interface. iii, 10, 11, 21, 23, 31

BART Bidirectional and Auto-Regressive Transformers. iii, 1, 3, 6, 7, 11, 21, 23, 31

CSS Cascading Style Sheets. 11

GPU Graphics Processing Unit. 10

HTML HyperText Markup Language. 11

HTTPS HyperText Transfer Protocol Secure. 7

NLP Natural Language Processing. 1, 3, 10, 11, 19, 23, 31

OCR Optical Character Recognition. 31

PDF Portable Document Format. iii, 6, 11, 19, 21, 23, 29–31

RAM Random Access Memory. 10

ROUGE-L Recall-Oriented Understudy for Gisting Evaluation - Longest Common Subsequence. 6

SQL Structured Query Language. 1, 6

T5 Text-To-Text Transfer Transformer. 3

UI User Interface. 6, 30

UML Unified Modeling Language. 12

URL Uniform Resource Locator. iii, 6, 19, 21, 23, 29, 31

WCAG Web Content Accessibility Guidelines. 7

1 Introduction

With thousands of research papers published daily on platforms like arXiv, keeping up with the latest work can be difficult and time-consuming. Researchers often spend hours reading long documents just to understand the main ideas.

Abstractify, also known as the *arXiv Summarizer*, is a web application designed to solve this problem. Built with *Flask*, it allows users to upload research papers or provide URLs and receive clear, well-structured summaries.

The system uses advanced NLP models such as BART and Gemini 2.0 Flash to generate concise summaries. It also supports LaTeX rendering for equations and stores summaries with timestamps using SQLite for easy access later.

Overall, **Abstractify** helps students, researchers, and educators quickly understand complex academic content and manage their reading workload more efficiently.

1.1 Problem Statement

In today's digital research environment, thousands of academic papers are published daily on platforms such as arXiv, IEEE, and Springer. While this rapid growth accelerates knowledge sharing, it also introduces several challenges for researchers, students, and educators. The major problems addressed by the Abstractify system are outlined below:

- **Information Overload:** The increasing volume of research papers makes it difficult for users to read and comprehend full documents within limited time.
- **Time-Consuming Manual Summarization:** Extracting key ideas and contributions from lengthy academic papers manually is inefficient and prone to human error.
- **Complex Academic Language:** Research papers often contain dense technical terminology and mathematical expressions that are difficult to quickly interpret.
- **Lack of Instant Summarization Tools:** Existing summarization tools often lack support for academic PDFs, LaTeX equations, or structured scientific content.
- **Dependency on Single Models:** Many systems rely on a single summarization model, making them unreliable when external APIs fail or become unavailable.
- **Poor Accessibility and Organization:** Users often lack an efficient way to store, revisit, and manage previously generated summaries for future reference.

This project addresses these challenges by developing Abstractify, a web-based academic paper summarization system that leverages advanced NLP models such as Gemini 2.0 Flash and BART/T5. The system provides fast, reliable, and well-formatted summaries, supports PDF and URL inputs, preserves mathematical expressions, and maintains a searchable summary history for improved research productivity.

1.2 Objectives

The primary objectives of the **Abstractify** project are defined using the SMART framework to ensure clarity, feasibility, and measurable outcomes:

- To design and develop a user-friendly, Flask-based web application that allows users to upload research papers or submit URLs for summarization.
  (Specific, Measurable: Supports PDF and URL inputs with 95% uptime, Achievable: Flask framework, Relevant: Improves research efficiency, Time-bound: By November 2025).

- To generate concise, accurate, and readable summaries using advanced NLP models such as BART, T5, and Gemini 2.0 Flash.
  (Specific, Measurable: ROUGE-L score > 0.4, Achievable: Pretrained and fine-tuned models, Relevant: High-quality summarization, Time-bound: Within 3 months).

- To ensure proper rendering of mathematical equations and structured content using LaTeX and Markdown formatting.
  (Specific, Measurable: 100% equation rendering accuracy, Achievable: MathJax integration, Relevant: Academic readability, Time-bound: Prototype phase).

- To maintain a persistent history of generated summaries using a local SQLite database for traceability and future reference.
  (Specific, Measurable: Storage of 100+ summaries, Achievable: SQLite database, Relevant: Research continuity, Time-bound: By Week 4).

- To optimize system performance to deliver fast and scalable summarization across different domains.
  (Specific, Measurable: Response time under 10 seconds, Achievable: Efficient model selection and fallback strategy, Relevant: User satisfaction, Time-bound: End of project).

1.3 Scope

The scope of the **Abstractify** system defines the boundaries and functionalities covered by the project. It focuses on providing an efficient academic paper summarization solution while maintaining simplicity and usability.

- The system supports summarization of academic documents through PDF uploads and research paper URLs (e.g., arXiv).

- It performs abstractive summarization using cloud-based (Gemini 2.0 Flash) and local fallback NLP models (BART/T5).

- The application includes preprocessing steps such as text extraction, cleaning, and formatting to ensure high-quality summaries.

- Generated summaries are rendered with proper LaTeX equation support and Markdown formatting for improved readability.

- The system stores summaries along with metadata such as source, timestamp, and model used in a local SQLite database.

2 Overview

Abstractify is a web-based application designed to help users efficiently comprehend long and complex research papers. Leveraging state-of-the-art AI tools and advanced NLP models such as BART, T5, and Gemini 2.0 Flash, Abstractify generates concise and clear summaries of academic content. Users can provide input either by uploading a PDF of a research paper or by submitting the URL of an online publication.

Process

Upon receiving a document, the system performs the following steps:

1. **Text Extraction and Cleaning:** The document is parsed, and unnecessary formatting or noise is removed to obtain clean text.
2. **Tokenization and Embedding:** The text is divided into smaller units (tokens) and transformed into numerical representations (embeddings) that the AI models can process.
3. **Model Selection:** The system selects the most appropriate summarization model. Gemini 2.0 Flash is preferred if available; otherwise, BART or T5 serves as a backup.
4. **Summary Generation:** The chosen model generates a summary of the content.
5. **Post-processing:** The summary is formatted in markdown, and any LaTeX equations are rendered properly.
6. **Storage:** The final summary, along with metadata such as timestamp and optional source link, is stored in a local SQLite database.

The entire application is delivered via a simple and interactive web interface built with *Flask*, making it accessible for students, researchers, and professionals who need quick insights from extensive academic papers.

Constraints

- Limited to English-language papers.
- Dependent on the availability of external AI models (e.g., Gemini API).
- Local storage may restrict scalability.

Scope

In-scope: Summarization of academic PDFs and URLs, LaTeX rendering, and SQLite-based storage.
Out-of-scope: Live streaming, mobile application development, and multi-language support.

Stakeholders

Primary stakeholders include students, researchers, educators, and developers.

3 Working Mechanism

3.1 System Flowchart

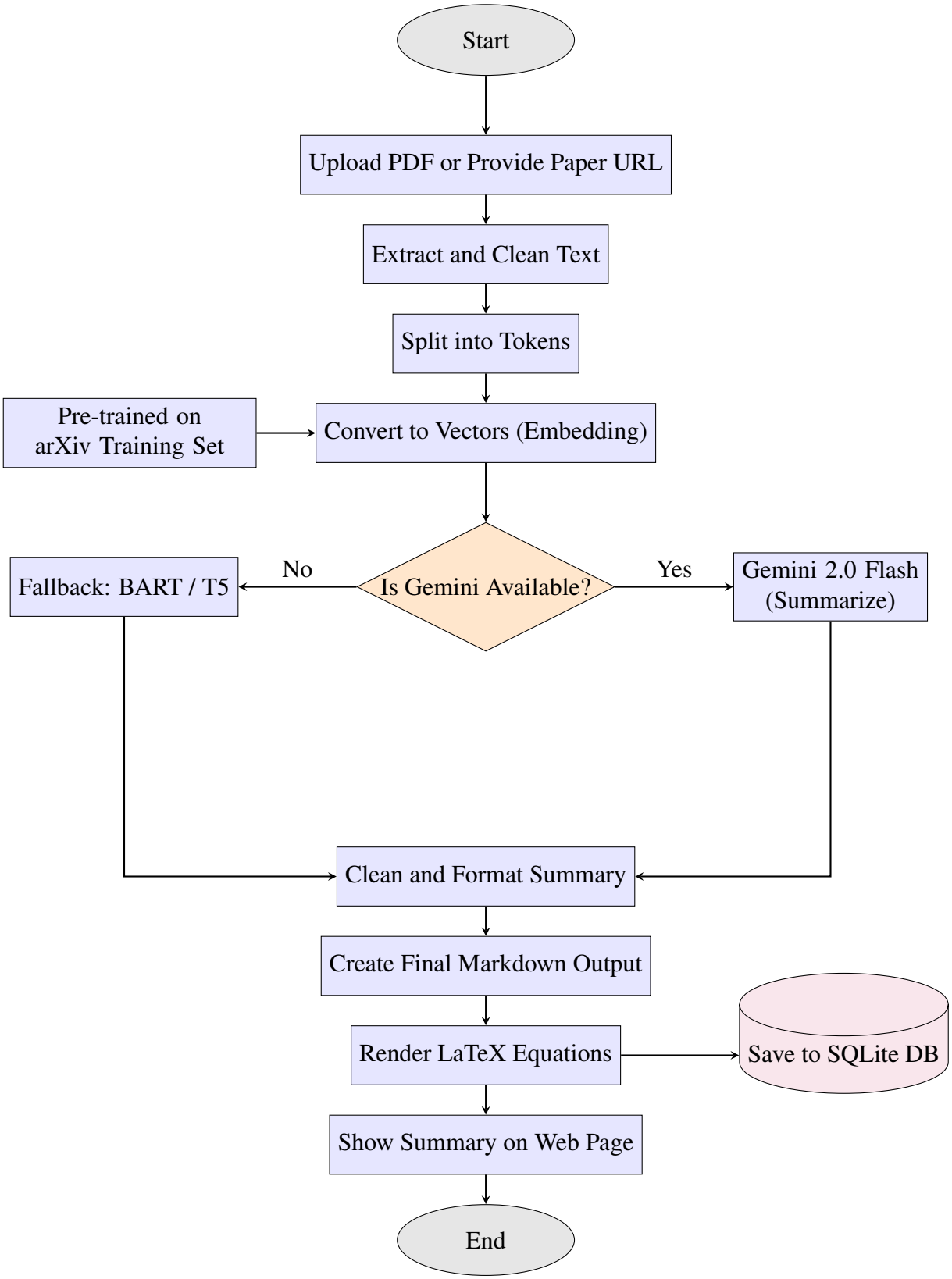

Figure 1: Abstractify Workflow Diagram

3.2 Algorithm for Summarization Models

Algorithm 1 Fine-Tuning Transformer-Based Model for Summarization

Require: Dataset D of (paper, abstract) pairs from arXiv, pre-trained model M (e.g., BART or T5), learning rate η, batch size B, number of epochs E, validation split V
Ensure: Fine-tuned model M_f

Preparing dataset
Split D into training set D_{train} (80%) and validation set D_{val} (20%)
Preprocess D_{train} and D_{val}: clean text, remove noise (e.g., headers, footers), normalize equations
Tokenize D_{train} and D_{val} using M's tokenizer, ensuring input length ≤ 1024 tokens and output length ≤ 256 tokens
Configuring model
Initialize M with pre-trained weights
Set optimizer to AdamW with learning rate $\eta = 3 \times 10^{-5}$
Configure loss function as cross-entropy for sequence-to-sequence tasks
Set batch size $B = 8$ and gradient accumulation steps if memory is constrained
Training loop
for $epoch = 1$ to E **do**
    for each batch $b \in D_{\text{train}}$ **do**
        Encode input paper text to input IDs
        Encode target abstract to target IDs
        Compute model output $M(b_{\text{input}})$
        Calculate loss L between $M(b_{\text{input}})$ and b_{target}
        Backpropagate L and update M's weights using optimizer
    end for
    Evaluate M on D_{val} using ROUGE-2 and ROUGE-L metrics
    if validation ROUGE-L improves **then**
        Save checkpoint of M as M_{best}
    end if
end for
Finalizing model
Select M_{best} based on highest ROUGE-L score
Perform inference on test set to verify summary quality
Save fine-tuned model M_f for deployment
Deployment preparation
Integrate M_f into the summarization pipeline
Test M_f with Gemini 2.0 Flash as fallback if unavailable
Monitor summary quality and retrain periodically with new arXiv data
return M_f

4 Requirement Analysis

Requirement analysis is the process of identifying, documenting, and analyzing the functional and non-functional needs of a system. It serves as the foundation for system design and development by ensuring that the solution aligns with user expectations and business objectives.

4.1 Functional Requirements

The functional requirements define the core capabilities that the **Abstractify** system must implement to achieve its objective of efficiently summarizing academic research papers. These requirements were derived from workflow, system architecture, and text-processing stages described in the proposal. The MoSCoW technique (Must, Should, Could, Won't) is applied to prioritize them. Each requirement includes a unique identifier, description, priority, acceptance criteria, and verification method, ensuring traceability and testability throughout the development lifecycle.

S.N.	Description	Priority	Acceptance Criteria	Verification Method
1	Enable users to upload a PDF or provide a URL of an academic document.	Must have	System successfully parses the file/URL and extracts accurate text without errors.	Unit test: Provide a sample arXiv PDF or URL and verify extracted text.
2	Extract and preprocess text by removing noise, splitting into 1024-token chunks, and generating embeddings using arXiv-trained models.	Must have	Clean, segmented text and embeddings free from irrelevant artifacts.	Integration test: Process noisy input and inspect logs for token and embedding integrity.
3	Generate a concise summary using Gemini 2.0 Flash (fallback BART/T5).	Must have	Summary $<$256 tokens, contextually correct, ROUGE-L $>$ 0.4.	System test: Compare with human summary and compute ROUGE-L.
4	Format summary in Markdown and render LaTeX equations correctly.	Should have	Equations and Markdown formatting display correctly.	User acceptance test: Visual inspection.
5	Store summaries with timestamp and optional source link in SQLite.	Must have	All entries saved and retrieved without corruption.	Database test: Query by timestamp and verify stored content.
6	Display the formatted summary in a Flask-based web interface.	Must have	Displayed summary matches generated output clearly.	End-to-end test: Run full workflow and verify UI output.

Table 1: Functional Requirements

4.2 Non Functional Requirements

Non-functional requirements define the system's performance, usability, security, and maintainability attributes, beyond core functionalities. These are essential for making Abstractify a reliable, user-centric, and scalable web application, based on AI tool standards. Each includes a unique identifier, description, and measurable target for evaluation and improvement.

S.N.	Attribute	Description	Measurable Target
1	Performance	System must process and deliver summaries efficiently, minimizing latency during peak usage.	Response time $\leq$ 10 seconds for 95% of requests under typical load.
2	Scalability	Application should handle growing users and data without performance loss.	Support 100 simultaneous users with $\leq$ 20% response time degradation.
3	Reliability	System must operate consistently with fallback mechanisms for disruptions.	Achieve 99% uptime, switching to BART/T5 if Gemini 2.0 Flash fails, over 30 days.
4	Usability	Web interface must be intuitive and accessible for diverse users.	Meet WCAG 2.1 guidelines, with user satisfaction $>$ 4/5 in testing.
5	Security	System must protect user data and prevent breaches with secure practices.	Use HTTPS for uploads, restrict to non-sensitive data, verified by penetration testing.
6	Maintainability	Codebase must support updates and debugging with modular design.	Maintain $>$ 80% code coverage and PEP 8 compliance via automated tools.
7	Portability	Application must work across platforms and browsers for global access.	Compatible with Chrome, Firefox, Windows, Linux, confirmed by cross-testing.

Table 2: Non-Functional Requirements

These requirements set a high quality standard, focusing on user satisfaction, resilience, and sustainability for Abstractify's deployment.

4.3 Feasibility Study

The feasibility study provides a comprehensive evaluation of the Abstractify project's viability across five critical dimensions—technical, economic, operational, legal, and schedule feasibility. This analysis is conducted in accordance with established software engineering methodologies to ascertain whether the project can be executed within the given constraints, leveraging insights from the proposal's technical scope and resource descriptions.

4.3.1 Technical Feasibility

The technical feasibility hinges on the utilization of well-established technologies, including the Flask web framework for the user interface, Hugging Face Transformers library for deploying NLP models such as BART and T5, and the Gemini API for advanced summarization capabilities. SQLite serves as a lightweight database solution for storing summary histories. The required skill set—encompassing Python programming, natural language processing integration, and web development—is presumed to be within the team's competencies, as inferred from their enrollment in a software engineering course. Potential technical risks, such as intermittent downtime of the Gemini API, are mitigated by the inclusion of fallback models (BART and T5). Given the availability of open-source tools and the team's academic preparation, the project is deemed technically feasible.

4.3.2 Economic Feasibility

The economic feasibility assessment considers the financial implications of development, focusing on cost-effectiveness for an academic setting. The primary expense is the team's time investment, estimated at 3 to 6 months for a student group, with no significant hardware or licensing costs. Open-source frameworks like Flask and SQLite, along with the Hugging Face library, eliminate software expenses, while the Gemini API offers a free tier suitable for prototyping. The anticipated benefit lies in significant time savings for users (e.g., researchers and students), translating into a positive return on investment through enhanced productivity and educational value. Thus, the project is economically viable within the academic context.

4.3.3 Operational Feasibility

Operational feasibility examines the system's post-deployment sustainability and ease of use. The Abstractify application requires minimal ongoing maintenance, limited to periodic retraining of NLP models with updated arXiv datasets to maintain accuracy. The intuitive Flask-based web interface eliminates the need for specialized training, enabling end-users—such as students, researchers, and educators—to integrate it into their daily workflows seamlessly. The low operational overhead and absence of dedicated staff requirements suggest that the system is operationally feasible with minimal resource demands.

4.3.4 Legal Feasibility

Legal feasibility ensures compliance with relevant regulations and intellectual property considerations. The system processes only publicly available arXiv papers, avoiding the storage or processing of personal data, which aligns with data privacy standards. Usage of external APIs, such as Gemini, must adhere to their respective terms of service, particularly regarding non-commercial use and data redistribution prohibitions. Intellectual property rights

for any fine-tuned models remain with the development team, posing no legal conflicts. Based on these considerations, the project presents no significant legal barriers, confirming its feasibility.

4.3.5 Schedule Feasibility

Schedule feasibility evaluates whether the Abstractify project can be completed within the allocated time frame using the available resources. The project is planned to be executed over a period of approximately 2 to 4 months, aligning with the academic semester schedule. The development timeline is structured into clearly defined phases, including requirement analysis, system design, implementation, testing, and documentation. Each phase is allocated sufficient time based on task complexity and team capacity.

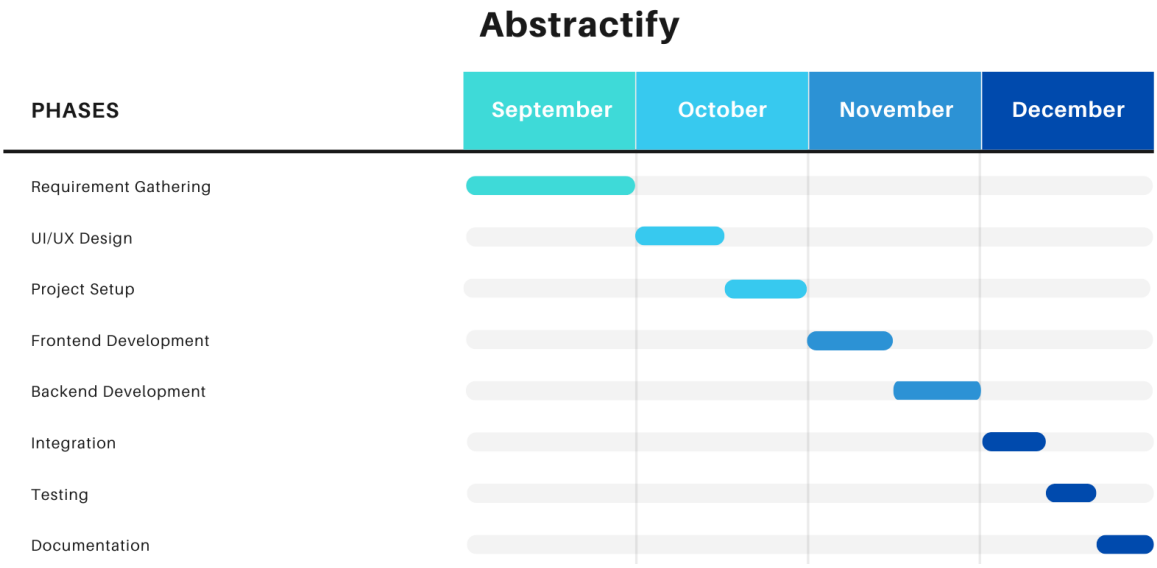

Figure 2: Gantt Chart

4.4 Hardware and Software Requirements

The development, testing, and deployment of **Abstractify** — the arXiv research paper summarizer web application — require carefully selected hardware and software components to ensure efficient performance, reliable NLP processing, and a smooth user experience. Below is a comprehensive overview of the resources used and recommended.

4.4.1 Hardware Requirements

The following hardware specifications are recommended to ensure smooth development, efficient model inference, and optimal end-user accessibility for the Abstractify system.

Development Machine
A moderately powerful machine is required to support model fine-tuning, training, and local inference tasks efficiently. Recommended specifications include:

Component	Minimum Specification	Recommended Specification
Processor	Intel Core i5 (8th Gen) or AMD Ryzen 5	Intel Core i7/i9 (11th Gen+) or AMD Ryzen 7/9
RAM	8 GB	16–32 GB (for faster model inference)
Storage	256 GB SSD	512 GB NVMe SSD
GPU (Optional)	Integrated GPU	NVIDIA GPU $\geq$ 8 GB VRAM
Operating System	Windows 10/11, macOS, or Linux	Linux Ubuntu 20.04+

Table 3: Hardware Requirements – Development Machine

End-User Device
For end-users accessing the Abstractify system via the web interface, basic hardware requirements are sufficient:

Component	Minimum Specification	Recommended Specification
Device	Any modern laptop or desktop	Any modern laptop or desktop
RAM	4 GB	8 GB or higher
Browser	Chrome, Firefox, Edge (latest)	Chrome, Firefox, Edge (latest)
Internet Connection	Required for Gemini API	Stable high-speed connection

Table 4: Hardware Requirements – End-User Device

4.4.2 Software Requirements

The Abstractify project leverages modern, open-source tools and libraries to provide high-quality abstractive summarization while keeping resource usage efficient.

Programming Languages and Core Technologies
The system is primarily developed using Python, which offers extensive support for machine learning, natural language processing, and web development. Core technologies include:

Component	Technology / Library	Purpose / Version
Programming Language	Python	3.10 or higher
Web Framework	Flask	2.3+
NLP Libraries	Hugging Face Transformers Sentence-Transformers google-generativeai	4.35+ (BART, T5 models) Text embedding Gemini 2.0 Flash integration
Text Extraction	PyPDF2 or pdfplumber	Extract text from research PDFs
LaTeX Rendering	MathJax	Client-side equation rendering
Database	SQLite	3.37+ (lightweight local storage)
Frontend Technologies	HTML5, CSS3, Bootstrap 5 JavaScript + MathJax	Responsive user interface Dynamic content rendering
Deployment (Planned)	Render / Heroku / PythonAnywhere Docker (optional)	Free-tier cloud hosting Containerization

Table 5: Software Requirements: Core Software Technologies

Development and Supporting Tools
Industry-standard tools are used for efficient coding, collaboration, testing, and environment management.

Tool	Name	Purpose
Code Editor	Visual Studio Code	Primary development environment
Version Control	Git & GitHub	Code versioning and collaboration
API Testing	Postman	Testing Gemini and local endpoints
Environment Management	venv / virtualenv	Isolated Python environments

Table 6: Software Requirements: Development Tools

5 System Models

The process of creating abstract representations of a system, where each model provides a distinct perspective of the system, is known as system modeling. This process can also be described as depicting a system using graphical notations in the UML. The UML notations employed to represent Abstractify are outlined as follows:

5.1 Use Case Diagram

A Use Case Diagram is a behavioral diagram in UML that illustrates a system's functional requirements from the perspective of its users. It depicts the interactions between actors (such as users or external systems) and the system, focusing on what the system should accomplish rather than how it will be implemented. Use case diagrams are useful for identifying key functionalities, defining system boundaries, and ensuring that all user requirements are captured prior to the design phase.

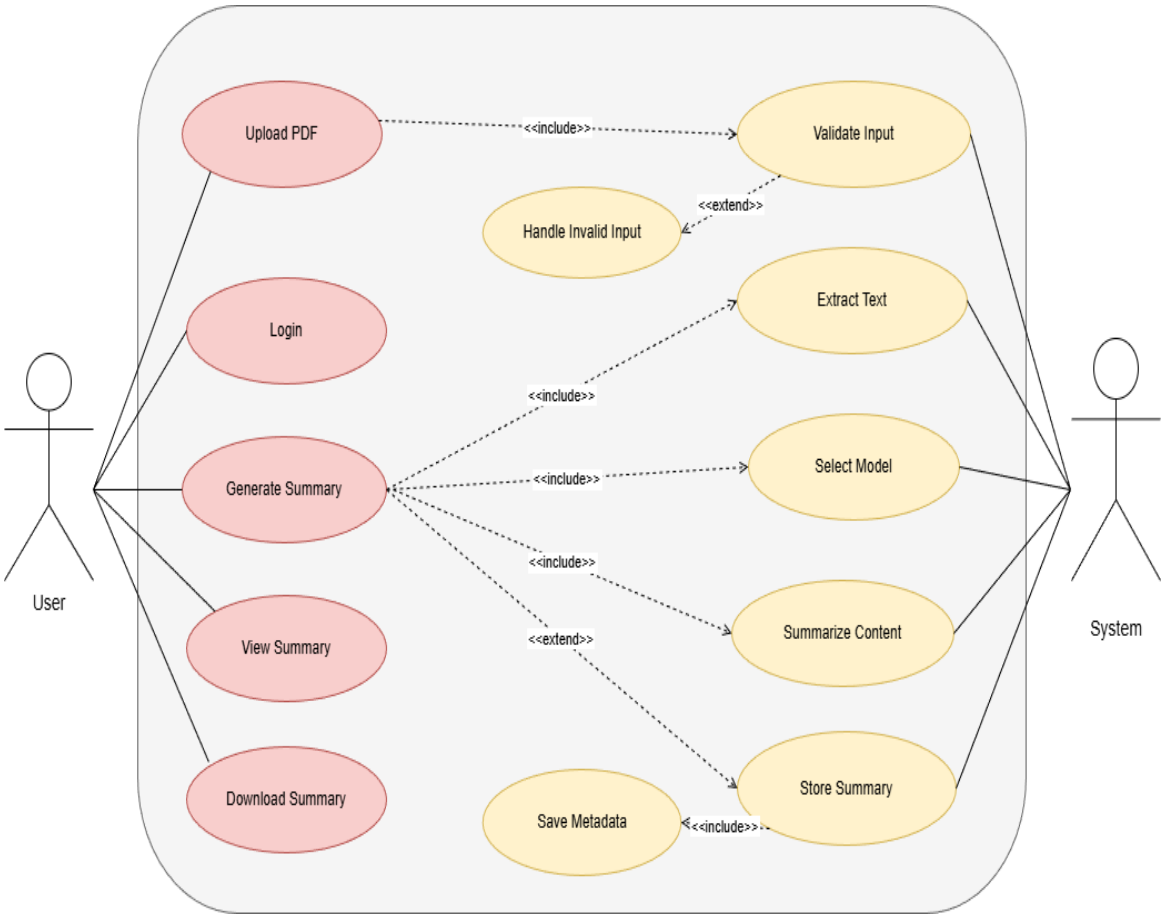

Figure 3: Use Case Diagram for Abstractify

Use Case	Actors	Description
Login	User	Allows the user to securely log in. The system validates credentials and grants access.
Upload PDF	User	Allows the user to upload a PDF document for summarization.
Generate Summary	User, System	Generates a summary of the uploaded document. Includes Extract Text, Select Model, Summarize Content; optionally extends Store Summary.
View Summary	User	Allows the user to view a previously generated summary.
Download Summary	User	Enables the user to download a summary as a file.
Validate Input	System	Checks uploaded PDFs or URLs for validity. Extends Handle Invalid Input if the input is invalid.
Extract Text	System	Extracts clean text from uploaded documents. Included in Generate Summary.
Select Model	System	Selects the appropriate AI/NLP model for summarization. Included in Generate Summary.
Summarize Content	System	Generates the actual summary using the selected model. Included in Generate Summary.
Store Summary	System	Stores the generated summary and metadata in the database. Extends Save Metadata.
Save Metadata	System	Stores metadata such as timestamp and source link. Included in Store Summary.
Handle Invalid Input	System	Provides feedback when uploaded document or URL is invalid. Extended from Validate Input.

Table 7: Use Case Description: Use Case Diagram for Abstractify

5.1.1 Login and Authentication

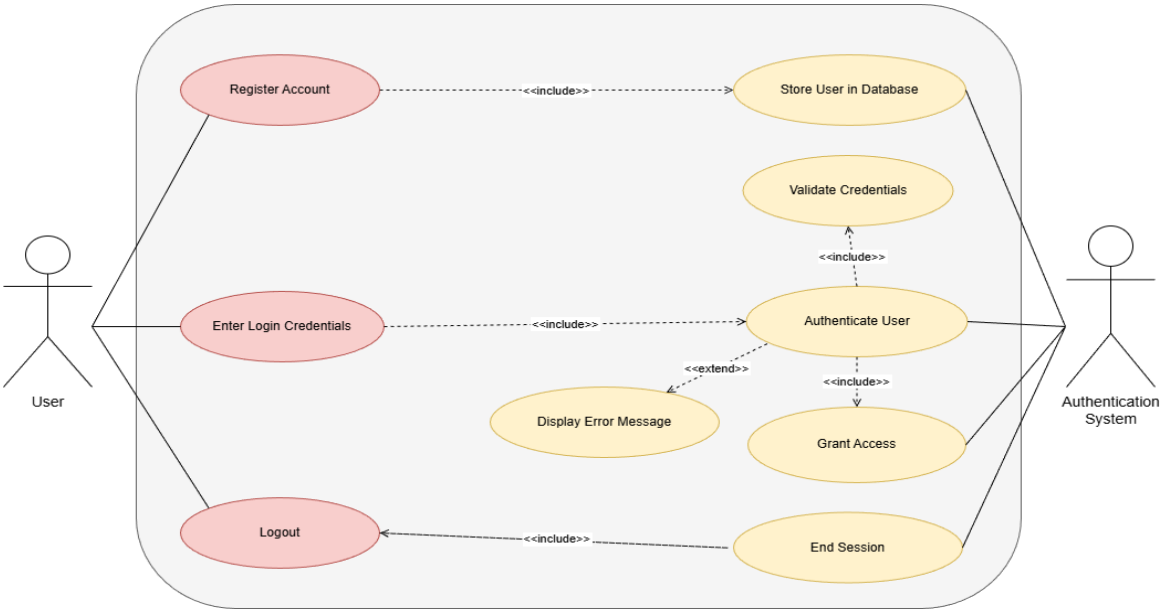

Figure 4: Use Case Diagram for the Login and Authentication Module

Description:

Abstractify – Login and Authentication Module	
Actors	User, Authentication System
Description	Enables registered users to securely access the Abstractify system. The system verifies the provided credentials and grants access to authorized users while preventing unauthorized access.
Data	Username/Email and Password
Stimulus	User initiates a login attempt by entering credentials into the system.
Response	The system establishes an active session for successfully authenticated users and redirects them to the dashboard. In case of invalid credentials, an error message is displayed.
Comments	Includes the following use cases: Validate Credentials, Authenticate User, Grant Access, and End Session. Display Error Message is an extended use case, triggered when login fails.

Table 8: Use Case Description: Login and Authentication Module

5.1.2 Input Handling

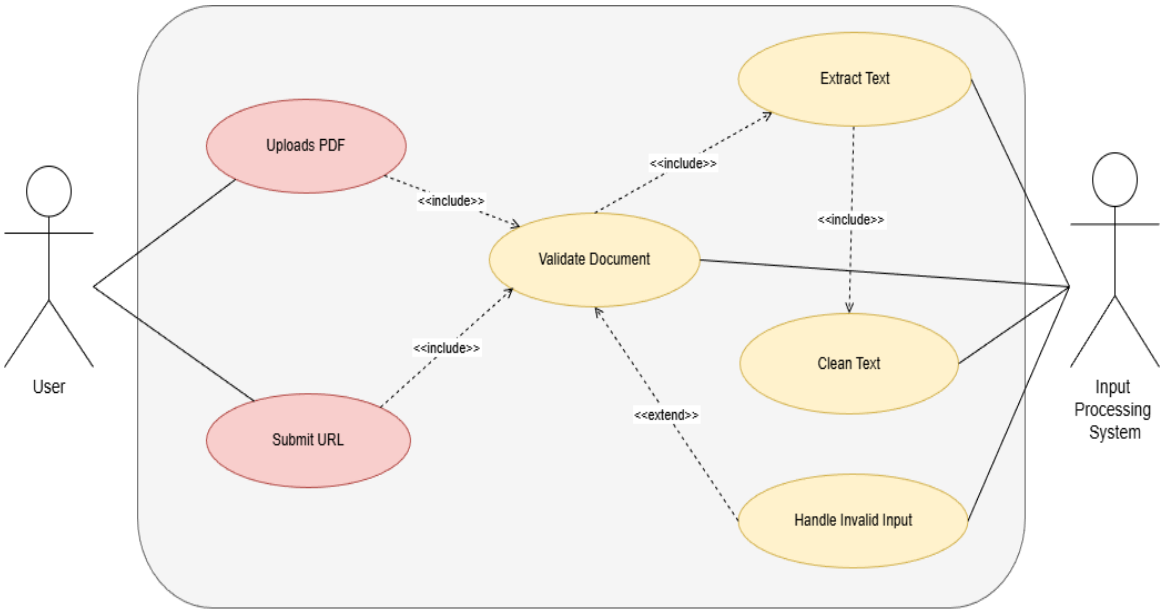

Figure 5: Use Case Diagram for the Input Handling Module

Description:

Abstractify – Input Handling Module	
Actors	User, Input Processing System
Description	Handles the input provided by users, either as uploaded PDFs or submitted URLs. The system validates the input, extracts relevant text, and cleans it for further processing. Invalid inputs are detected and appropriate feedback is provided.
Data	PDF files, URLs, extracted raw text
Stimulus	User uploads a PDF or submits a URL for summarization
Response	The system validates the input, extracts and cleans the text, and provides error messages for invalid inputs. Valid input proceeds to the summarization module.
Comments	Includes the following use cases: Validate Document, Extract Text, Clean Text. Display Error Message is an extended use case triggered when invalid input is detected.

Table 9: Use Case Description: Input Handling Module

5.1.3 Summarization

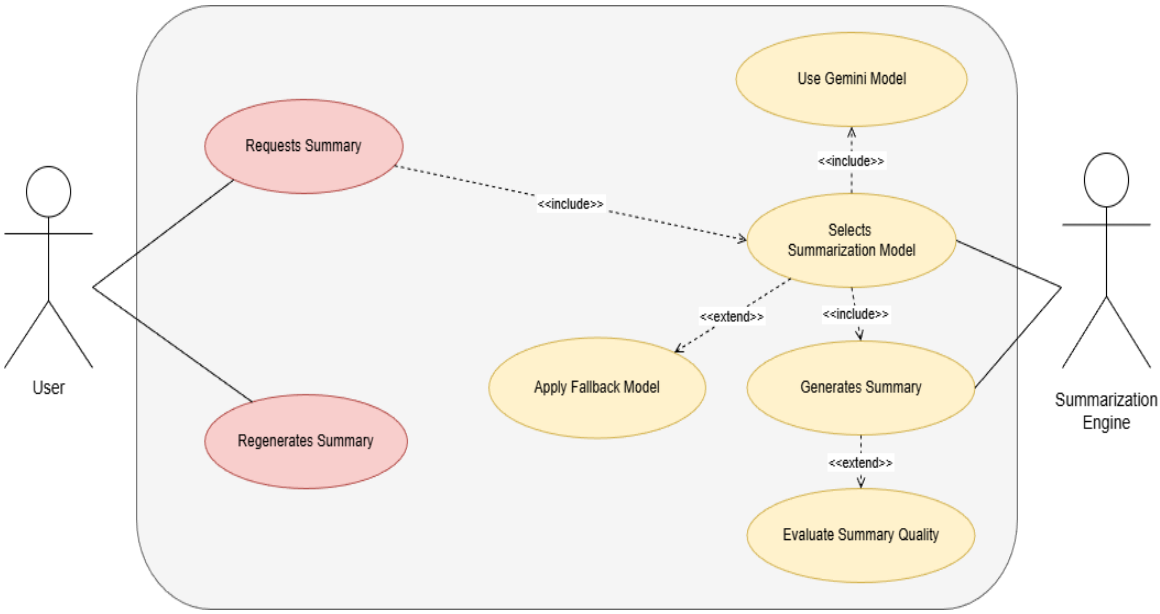

Figure 6: Use Case Diagram for the Summarization Module

Description:

Abstractify – Summarization Module	
Actors	User, Summarization Engine
Description	Generates concise and accurate summaries of academic documents submitted by users. The system selects the appropriate AI/NLP model, applies fallback models if necessary, formats the output with Markdown and LaTeX, and returns the summary to the user.
Data	Cleaned text, tokenized embeddings, generated summary
Stimulus	User requests a summary for an uploaded PDF or submitted URL
Response	The system generates and returns a formatted summary. If the preferred model fails, a fallback model is applied. Summary quality may also be optionally evaluated.
Comments	Includes the following use cases: Selects Summarization Model, Use Gemini Model, Generate Summary. Apply Fallback Model and Evaluate Summary Quality are extended use cases triggered under specific conditions.

Table 10: Use Case Description: Summarization Module

5.1.4 Database Management

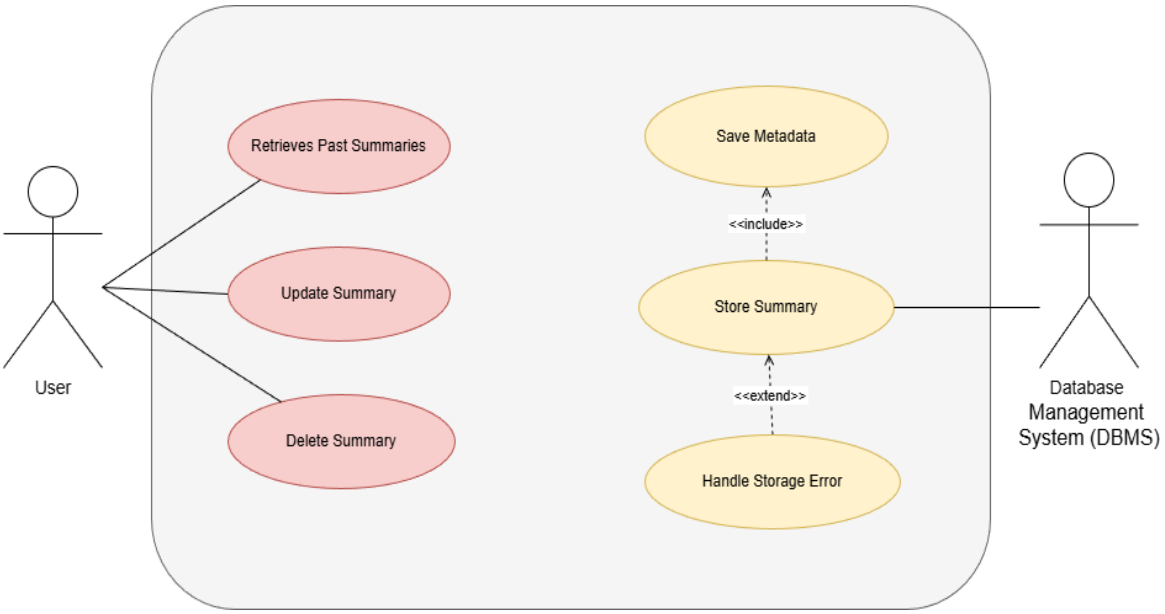

Figure 7: Use Case Diagram for the Database Module

Description:

Abstractify – Database Management Module	
Actors	End User, Database Management System (DBMS)
Description	Manages storage, retrieval, and updating of summaries in the system. The module ensures data integrity, saves metadata such as timestamps and source links, and handles any storage errors that may occur.
Data	Summaries, timestamps, source links, user identifiers
Stimulus	User requests to store, retrieve, update, or delete a summary
Response	The system successfully stores, retrieves, updates, or deletes summaries. Metadata is automatically recorded, and any storage errors trigger error handling procedures.
Comments	Includes the following use cases: Store Summary, Save Metadata. Handle Storage Error is an extended use case triggered under specific conditions. End User actions include Retrieve Past Summaries, Update Summary, and Delete Summary.

Table 11: Use Case Description: Database Management Module

5.2 Activity Diagram

An activity diagram shows the workflow of a system through a sequence of actions and decision points. In Abstractify, it illustrates the process from user login and document submission to input validation, text extraction, summary generation, storage, and final display of the summary.

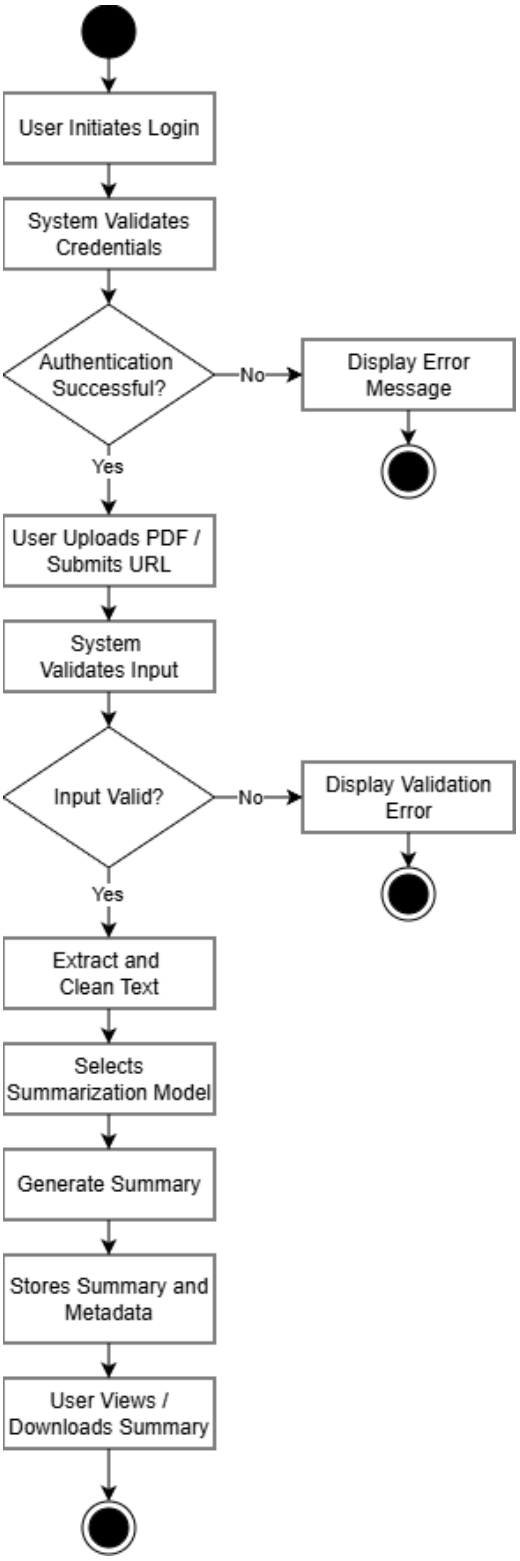

Figure 8: Activity Diagram of Abstractify

Description:

The activity diagram for the Abstractify System illustrates the step-by-step workflow of user interactions, starting from accessing the application to generating and viewing summarized content.

1. **User Authentication:**
    - The user accesses the Abstractify web application.
    - The system checks whether the user is authenticated.
    - If the user is not logged in, they are prompted to enter login credentials.
    - Upon successful authentication, access is granted; otherwise, an error message is displayed and the user is asked to re-enter credentials.
2. **Document Submission:**
    - After login, the user uploads a PDF or submits a document URL.
    - The system validates the provided input.
    - If the input is invalid, an error message is shown and the user is prompted to re-submit the input.
3. **Text Processing:**
    - The system extracts text from the valid document.
    - The extracted text is cleaned and prepared for summarization.
4. **Summary Generation:**
    - The system selects an appropriate summarization model.
    - A concise summary is generated using NLP techniques.
    - If the primary model fails, a fallback model is applied.
5. **Storage and Display:**
    - The generated summary is stored in the database along with relevant metadata.
    - The formatted summary is displayed to the user.
6. **Final Actions:**
    - The user may view, download, update, or retrieve past summaries.
    - The session continues until the user logs out or exits the system.

5.3 Sequence Diagram

A sequence diagram illustrates the dynamic behavior of a system by depicting the chronological exchange of messages among its components. In the Abstractify system, it demonstrates the interactions between the user, the application logic, the summarization engine, and the database, clearly showing the flow from document submission and validation to summary generation, storage, and final presentation to the user.

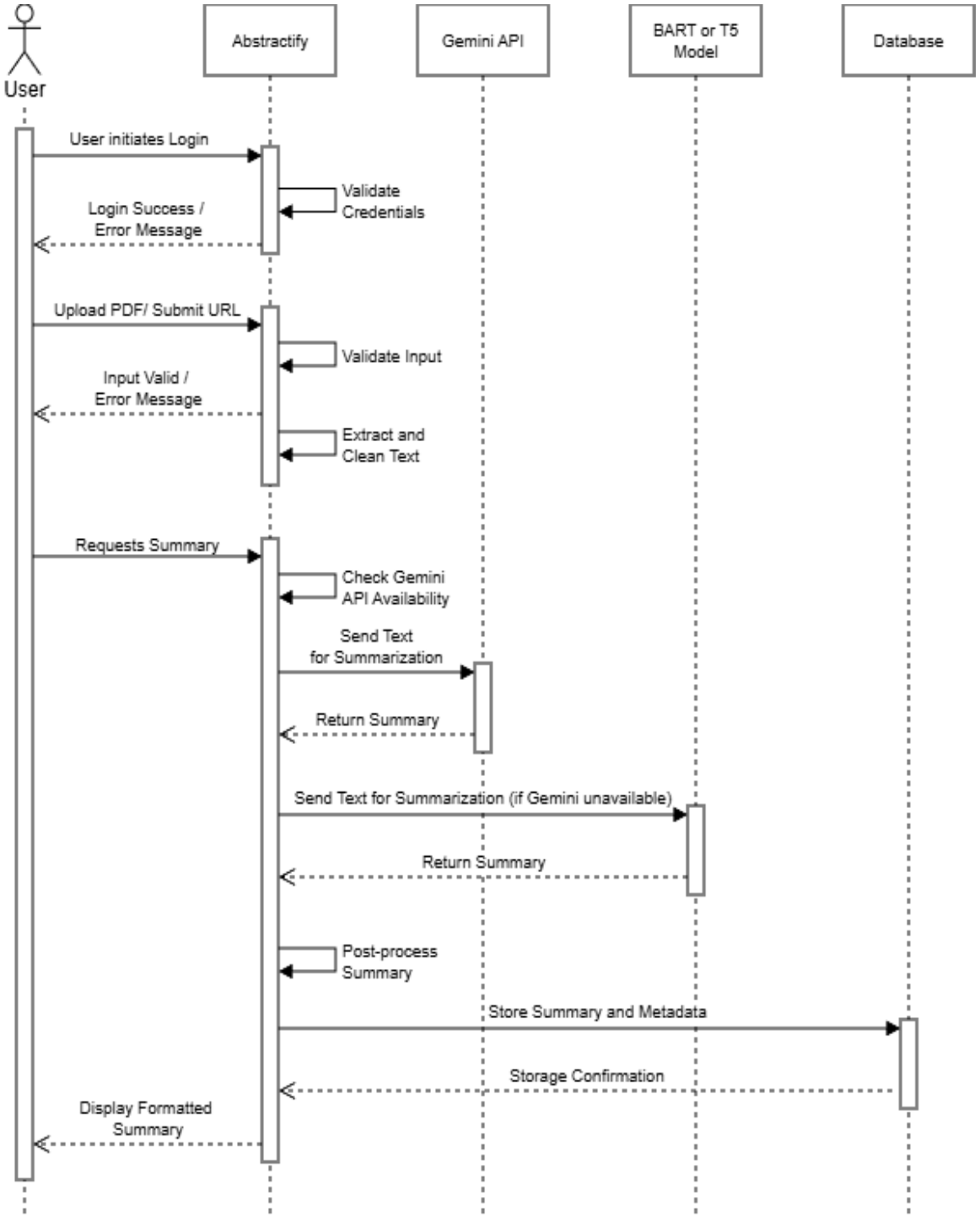

Figure 9: Sequence diagram of Abstractify

Description:

The sequence diagram of the Abstractify System illustrates the complete execution flow of the document summarization process, beginning with user interaction and concluding with result presentation and data persistence.

Component	Role in the System	Interactions / Actions
User	Primary actor interacting with the system	Initiates the workflow by providing login credentials, uploading a PDF document or submitting a research paper URL, requesting an abstractive summary, and viewing or downloading the generated output.
Abstractify	Central controller and application interface	Authenticates users, validates input documents, extracts and preprocesses text, orchestrates communication with summarization models, stores generated summaries in the database, and delivers formatted results back to the user interface.
Gemini API	Cloud-based abstractive summarization service	Receives preprocessed text from the Abstractify system and generates a high-quality abstractive summary using the Gemini language model, returning the result to the application when the service is available.
BART / T5 Model	Local fallback summarization engine	Produces summaries locally when the Gemini API is unavailable or unreachable, ensuring system reliability and uninterrupted service delivery. The generated summary is returned to the core system for further processing.
Database	Persistent data storage component	Stores finalized summaries along with relevant metadata such as document source, timestamp, and the summarization model used, and confirms successful storage to the Abstractify system for future retrieval.

Table 12: Sequence Diagram Components and Their Roles

5.4 Class Diagram

A class diagram represents the static structure of a system by showing its classes, attributes, methods, and the relationships between them. In the Abstractify system, the class diagram illustrates how user interactions, input processing, summarization logic, and database storage are organized and connected.

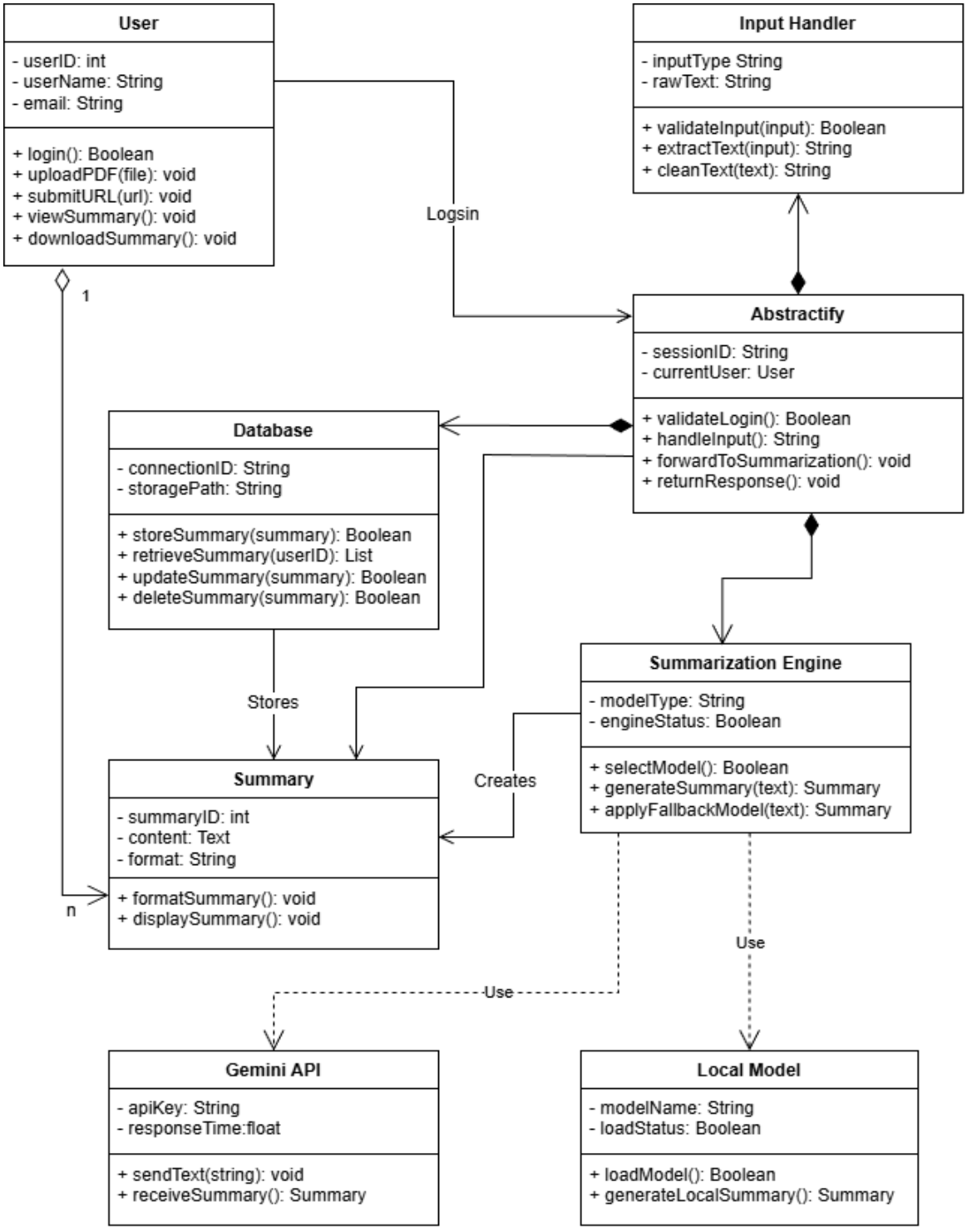

Figure 10: Class diagram of Abstractify

Description:

The following table describes the main classes used in the Abstractify system and their responsibilities within the overall architecture:

Class Name	Description
User	Represents an end user of the Abstractify system. The user initiates actions such as logging in, uploading a PDF, submitting a URL, requesting summaries, viewing generated summaries, and downloading results.
Abstractify	Acts as the central controller of the system. It manages user sessions, validates inputs, coordinates text processing and summarization, communicates with external services, and returns formatted results to the user interface.
InputHandler	Responsible for handling and validating user inputs. This class processes uploaded PDF files or submitted URLs, extracts raw text, and performs cleaning and preprocessing before passing the text to the summarization engine.
Summarization Engine	Manages the summarization process by selecting the appropriate NLP model. It communicates with the Gemini API for cloud-based summarization or applies a local BART/T5 model as a fallback, and generates a concise summary.
GeminiAPI	Represents the external cloud-based summarization service. It receives cleaned text from the system and returns an abstractive summary using the Gemini 2.0 Flash model.
LocalModel (BART/T5)	Serves as the local fallback summarization component. It generates summaries when the external API is unavailable, ensuring system reliability and continuity.
Database	Handles persistent storage of generated summaries. It stores, retrieves, updates, and deletes summaries along with related metadata such as timestamps, source information, and model usage.
Summary	Represents the summarized content produced by the system. It stores the summary text, format information, and supports display and formatting operations.

Table 13: Description of Classes in the Abstractify System

5.5 Deployment Diagram

A deployment diagram illustrates the physical deployment of software components across hardware nodes in a system. In Abstractify, it depicts the distribution of the Flask web application on a server, the client-side web browser on the user's device, the external Gemini API and arXiv API in the cloud, and the local SQLite database, highlighting the flow of data from user requests to summarization and persistent storage.

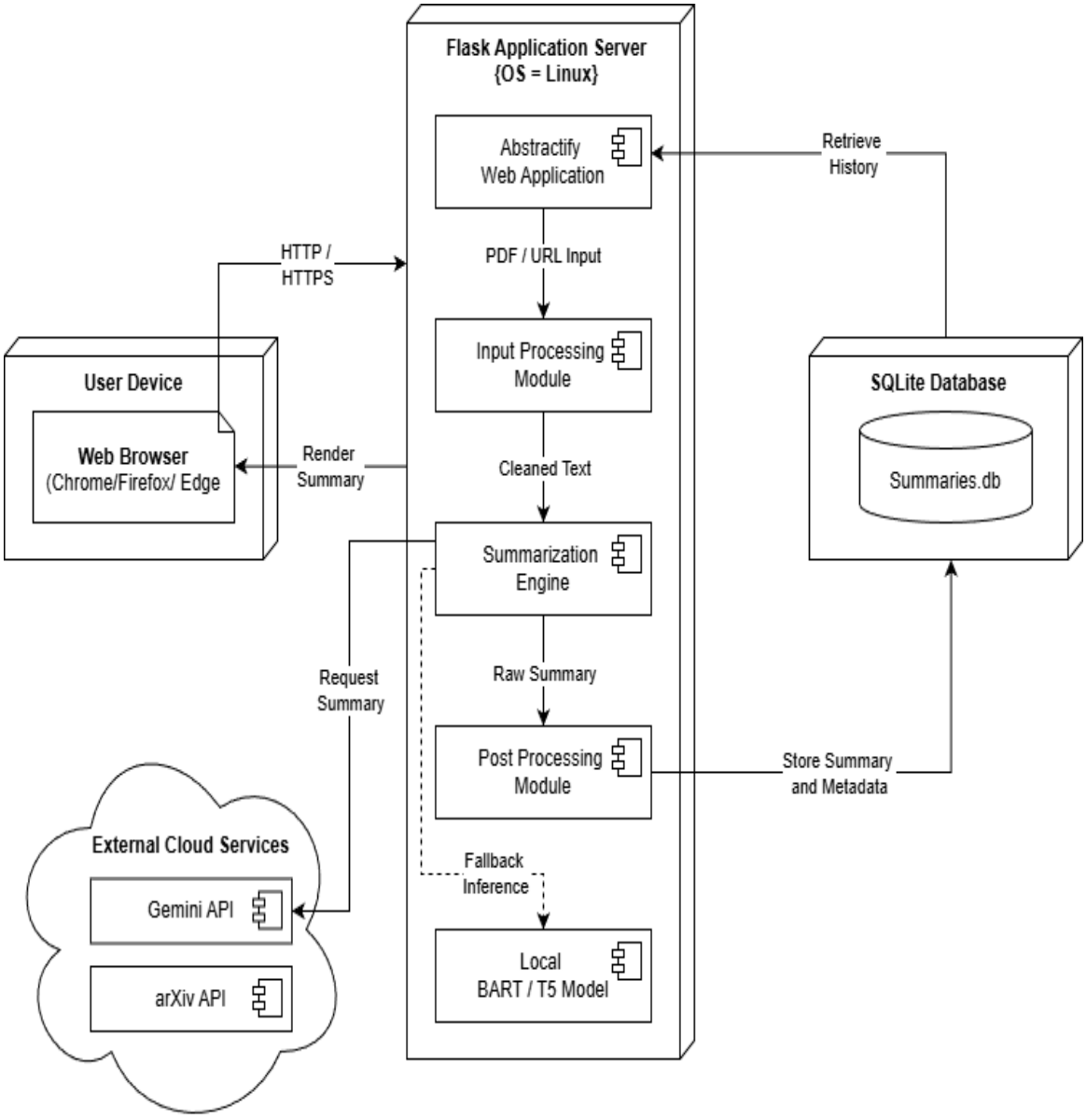

Figure 11: Deployment diagram of Abstractify

Description:

The following table describes the main nodes and components used in the Abstractify system deployment and their responsibilities within the overall architecture:

Node / Component	Description
User Device	Represents the end user's device (laptop, desktop, or mobile) running a modern web browser (Chrome, Firefox, Edge). It enables users to access Abstractify via HTTP/HTTPS, upload PDFs, submit arXiv URLs, view summaries, and interact with history and export summaries.
Flask Application Server	The central deployment node (Linux-based server, local or cloud-hosted like Render/Heroku). It hosts the core Flask web application, Input Processing Module, Summarization Engine, Post-processing Module (MathJax rendering), and Local Model. It manages user requests, coordinates summarization, and communicates with external services and the database.
SQLite Database	A lightweight, file-based local database deployed on the same server as the Flask application. It persistently stores generated summaries along with metadata (source, timestamp, model used, processing time) for fast retrieval and history access.
Gemini 2.0 Flash API	An external cloud-based NLP service (Google AI). It serves as the preferred summarization engine, receiving cleaned text and returning high-quality abstractive summaries when internet connectivity is available.
arXiv API	An external public API from arXiv.org. It fetches full paper content when a user submits an arXiv URL instead of uploading a PDF.
Local BART / T5 Model	A local fallback inference component running on the Flask server (via Hugging Face Transformers). It generates summaries when the Gemini API is unavailable, ensuring system reliability and offline capability.

Table 14: Description of Deployment Nodes and Components in Abstractify

6 Implementation and Testing

6.1 Implementation

The Abstractify system is implemented as a Flask-based web application that seamlessly integrates document processing, natural language processing for summarization, and SQLite-based storage into an efficient, user-friendly workflow.

6.1.1 Implementation Tools

The project employs the following modern, open-source technologies and tools:

Category	**Tool / Technology**	**Purpose / Description**
Programming Language	Python	Used as the core programming language for backend development, text processing, and model integration due to its extensive NLP and web development support.
Web Framework	Flask	Handles routing, user requests, session management, and communication between frontend, summarization engine, and database.
NLP Models	BART, T5	Transformer-based abstractive summarization models used as local fallback options when cloud-based APIs are unavailable.
Cloud AI Service	Gemini 2.0 Flash API	Provides fast and high-quality abstractive summaries using a cloud-based large language model.
PDF Processing	pdfplumber	Extracts raw text from uploaded research paper PDFs for further preprocessing and summarization.
External Data Source	arXiv API	Fetches research paper content when users submit URLs instead of PDF files.
Frontend Technologies	HTML, CSS, JavaScript	Used to design a responsive and interactive web interface for user interaction.
Database	SQLite	Stores generated summaries along with metadata such as source, timestamp, and model used for traceability and history management.
Development Tools	VS Code, Git	VS Code is used for development, while Git is used for version control and project management.

Table 15: Implementation Tools Used in Abstractify

6.1.2 User Interface Demonstration

The Abstractify system has been implemented with a user-friendly web interface. The following screenshots illustrate the key features of the application:

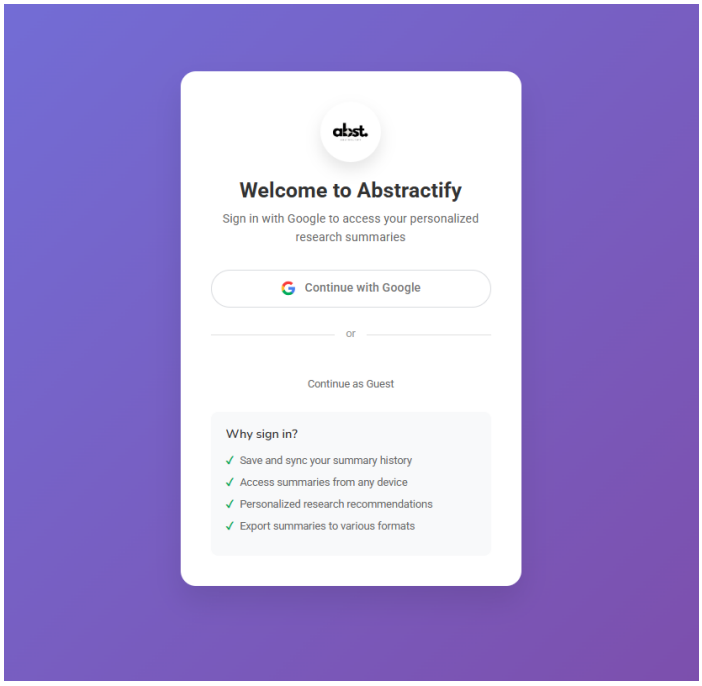

Figure 12: Abstractify login page

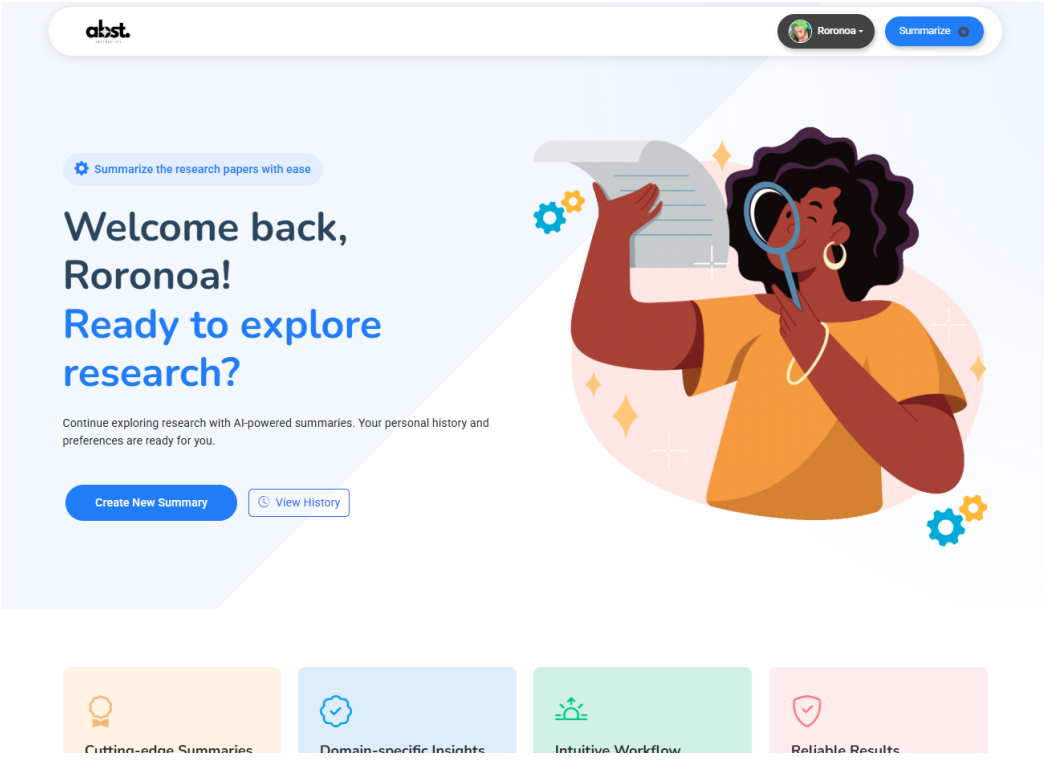

Figure 13: Abstractify homepage

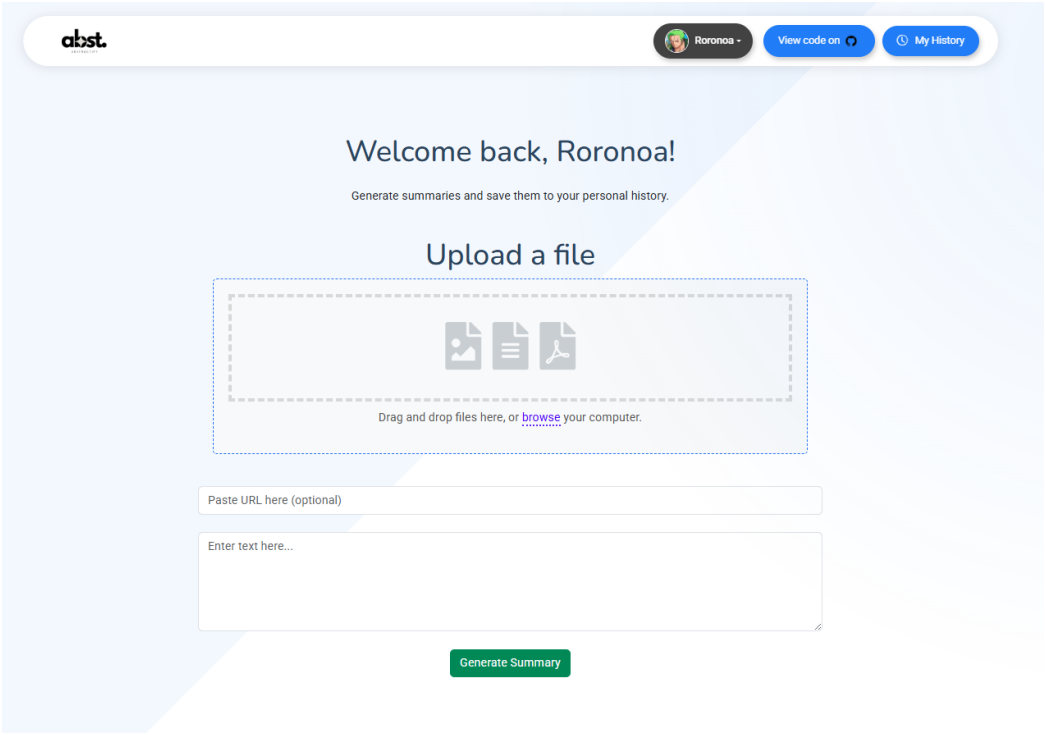

Figure 14: Summary input interface

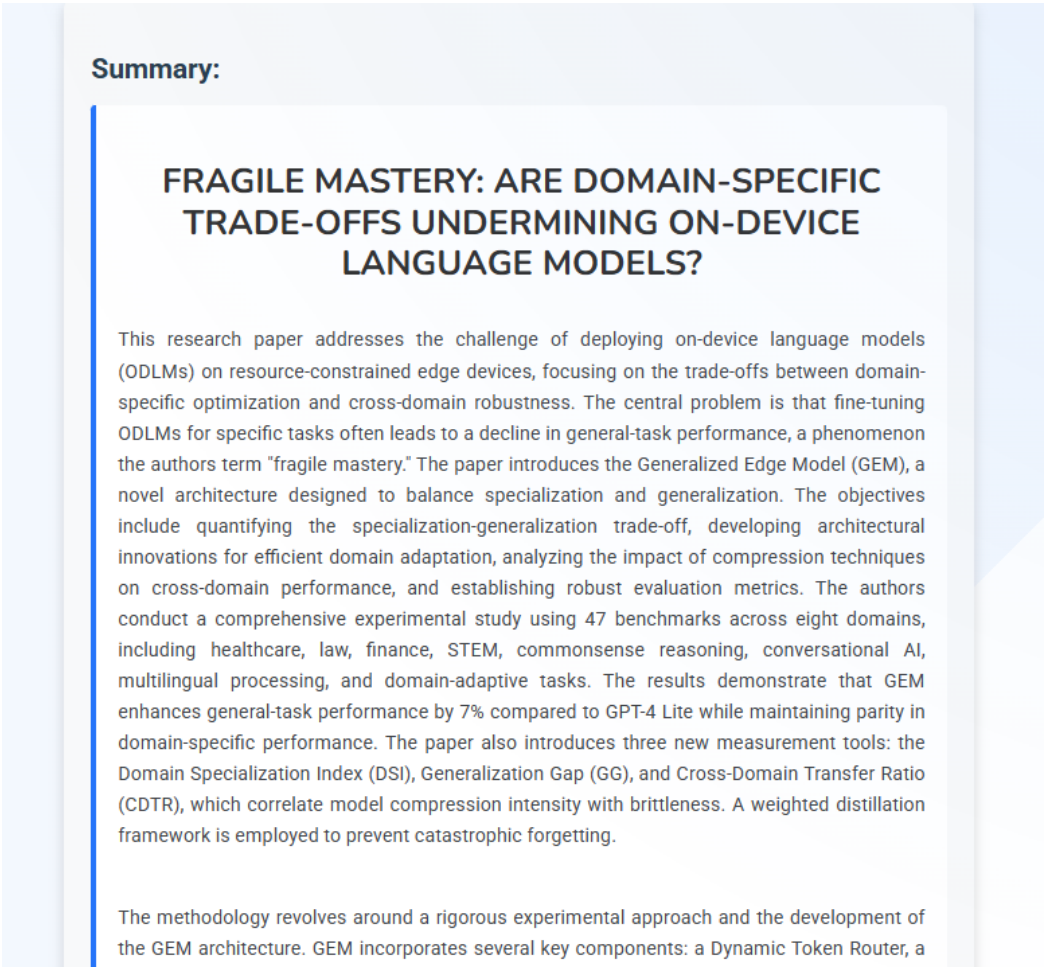

Figure 15: Generated summary output

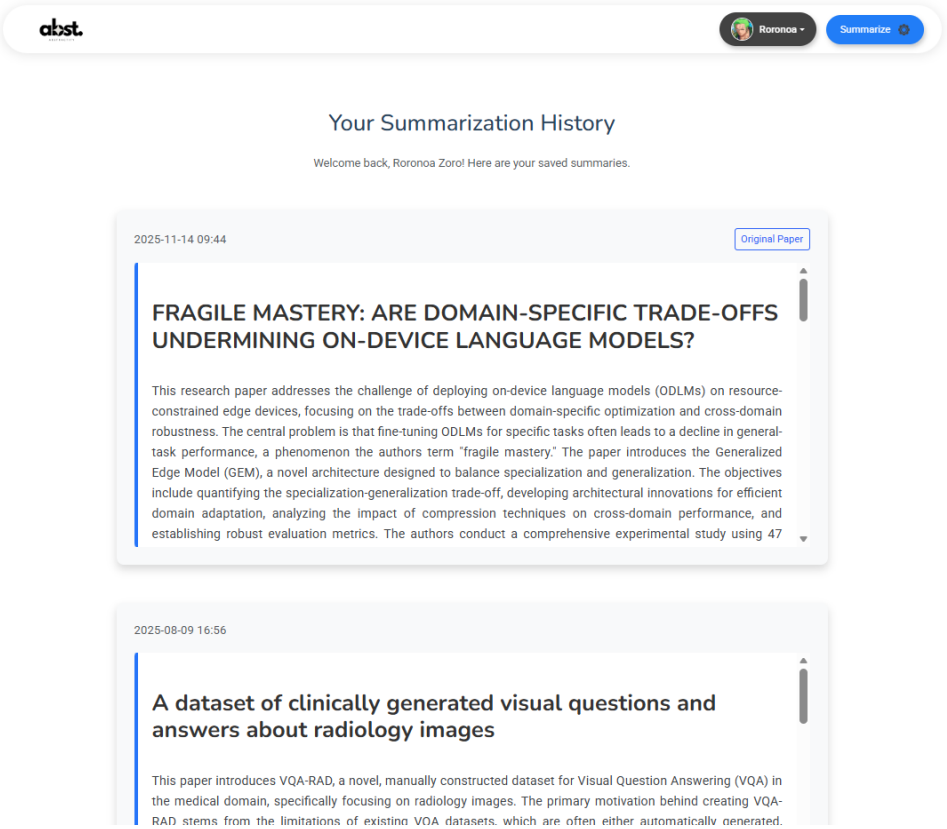

Figure 16: Summary history page

6.2 Testing

The Abstractify system has been thoroughly tested to ensure reliability, accuracy, and performance. The following tests have been performed:

6.2.1 System Testing

System testing ensures that all components of the Abstractify system work together as expected. The complete system was tested using both PDF uploads and arXiv URL inputs. The performance was evaluated based on successful text extraction, summary generation, formatting accuracy, and storage reliability.

Test Case ID	Input	Expected Output
TC-ST-01	Upload PDF research paper	Clean text extracted and summarized correctly
TC-ST-02	Submit valid arXiv URL	Summary generated and displayed with equations rendered

Table 16: System Testing Test Cases

6.2.2 Unit Testing

Each major module of the system, such as input validation, text extraction, summarization, and database storage, was individually tested using unit test cases in Python. This ensured that every component performed correctly in isolation.

Test Case ID	Function / Input	Expected Output
TC-UT-01	validate_input() with valid PDF	Input accepted and forwarded for processing
TC-UT-02	extract_text() with research PDF	Correct raw text extracted from document
TC-UT-03	generate_summary() using model	Concise abstractive summary produced

Table 17: Unit Testing Test Cases

6.2.3 Integration Testing

Integration testing verified the correct interaction between modules. The input handling module was integrated with the summarization engine, and the generated summaries were successfully stored and retrieved from the SQLite database through the Flask backend.

Test Case ID	Scenario / Input	Expected Output
TC-IT-01	PDF upload through web interface	Text extracted, summarized, and stored in database
TC-IT-02	Summary generation request	Summary displayed on UI with timestamp saved

Table 18: Integration Testing Test Cases

6.2.4 User Acceptance Testing

User acceptance testing was conducted with students and researchers using the Abstractify web interface.The system was evaluated based on usability, response time, and clarity of generated summaries.

Test Case ID	Scenario	Expected Output
TC-UAT-01	User uploads research paper	Summary generated within acceptable time ($<$10 s)
TC-UAT-02	Non-technical user uses system	User easily navigates interface and understands output

Table 19: User Acceptance Testing Test Cases

7 Conclusion and Future Enhancements

7.1 Conclusion

The Abstractify system was successfully designed and implemented to provide efficient and accurate abstractive summarization of academic research papers. The system allows users to upload PDF documents or submit research paper URL and generates concise summaries using modern NLP techniques. By integrating transformer-based models such as Gemini 2.0 Flash along with locally deployed BART/T5 models, the system effectively captures the core concepts of lengthy academic texts while preserving contextual meaning.

The application follows a modular architecture that includes input validation, text extraction, preprocessing, summarization, and persistent storage. A web-based interface enables users to interact seamlessly with the system, submit documents, view generated summaries, and retrieve previously summarized content. The fallback mechanism enhances reliability by automatically switching to local models when the cloud-based API is unavailable.

Overall, Abstractify fulfills its intended objectives by offering a fast, reliable, and user-friendly solution for academic paper summarization. The project demonstrates the practical application of NLP techniques, machine learning models, backend web development, and database management within a cohesive software system. Although the current version is fully functional, it also provides a strong foundation for future improvements and extensions.

7.2 Future Enhancements

The Abstractify system can be further improved in future versions through the following enhancements:

1. **Advanced User Management**
   Implementation of role-based authentication, personalized user profiles, and enhanced summary history management to improve usability and security.

2. **Support for Additional Document Formats**
   Extension of input support to include formats such as DOCX, PPT, and scanned PDFs using OCR techniques.

3. **Model Optimization and Fine-Tuning**
   Fine-tuning summarization models on domain-specific datasets such as medical, legal, or technical research papers to improve accuracy and relevance.

4. **Multilingual Summarization**
   Addition of support for summarizing research papers written in multiple languages to broaden system accessibility.

5. **Cloud Deployment and Scalability**
   Deployment of the system on cloud platforms to ensure scalability, high availability, and improved performance under heavy workloads.

References

[1] P. Grinberg, *Flask Web Development: Developing Web Applications with Python*, 2nd ed. O'Reilly Media, 2020.

[2] T. Wolf *et al.*, "HuggingFace's Transformers: State-of-the-art Natural Language Processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Oct. 2020, pp. 38–45. [Online]. Available: https://arxiv.org/abs/1910.03771

[3] M. Lewis *et al.*, "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Jul. 2020, pp. 7871–7880. [Online]. Available: https://arxiv.org/abs/1910.13461

[4] Google AI, "Gemini 2.0 Flash: Technical Report and Developer Documentation," Google AI Blog, 2024. [Online]. Available: https://ai.google.dev/gemini-api/docs

[5] J. Rose, "pdfplumber: Plumb a PDF for detailed information about each char, rectangle, and line," GitHub Repository, 2023. [Online]. Available: https://github.com/jsvine/pdfplumber

[6] arXiv Team, "arXiv API User's Manual," Cornell University, 2024. [Online]. Available: https://arxiv.org/help/api/user-manual

[7] MathJax Consortium, "MathJax Documentation v3.2," 2024. [Online]. Available: https://docs.mathjax.org/en/latest/

[8] SQLite Consortium, "SQLite Documentation," 2024. [Online]. Available: https://www.sqlite.org/docs.html