WRITEWIZARD - COLLABORATIVE DOCUMENT EDITING TOOL: REAL-TIME MULTI-FUNCTIONAL PLATFORM

(INTELLIGENT PAPER RECOMMENDATION AND CITATION SYSTEM)

Krithiga. D. Balakrishnan

(IT21194894)

B.Sc. (Hons) in Information Technology

Specializing in Software Engineering

Department of Computer Science & Software Engineering

Sri Lanka Institute of Information Technology

Sri Lanka

April 2025

WRITEWIZARD - COLLABORATIVE DOCUMENT EDITING TOOL: REAL-TIME MULTI-FUNCTIONAL PLATFORM

(INTELLIGENT PAPER RECOMMENDATION AND CITATION SYSTEM)

Krithiga. D. Balakrishnan

(IT21194894)

Dissertation submitted in partial fulfillment of the requirements for the Bachelor of Science (Hons) in Software Engineering

Department of Computer Science & Software Engineering

Sri Lanka Institute of Information Technology

Sri Lanka

April 2025

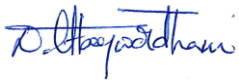
DECLARATION

I declare that this is my own work, and this Thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to Sri Lanka Institute of Information Technology, the nonexclusive right to reproduce and distribute my Thesis, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Name	Student ID	Signature
Krithiga D. Balakrishnan	IT21194894	

The above candidate has carried out this research thesis for the Degree of Bachelor of Science (honors) Information Technology (Specializing in Software Engineering) under my supervision.



Signature of the supervisor

(Dr. Lakmini Abeywardhana) Date: 11.04.2024

Signature of co-supervisor

(Ms. Karthiga Rajendran) Date: 11.04.2024

ABSTRACT

In the rapidly evolving landscape of academic research and collaborative writing, efficient and accurate citation management remains a persistent challenge. This research presents the design, development, and deployment of an AI-powered Reference Suggestion and Citation Generator, built to enhance the research workflow by seamlessly integrating real-time, intelligent referencing directly within a document editing environment. The system leverages three fine-tuned transformer-based models—BERT for keyword extraction, SBERT for semantic similarity matching, and Flan-T5 for IEEE citation generation. Upon highlighting any academic text, the system identifies key concepts, retrieves contextually relevant research papers through FAISS-indexed semantic embeddings, and enables citation generation with one click. It supports both automatic citation from suggested sources and manual entry of metadata for custom citations. All services are deployed via API on a cloud-hosted backend and integrated with a collaborative frontend editing interface, allowing inline citation insertion and reference list updates. The platform addresses common gaps in traditional citation tools, such as workflow fragmentation, manual formatting, and lack of real-time collaboration. Through modular architecture, real-time model inference, and Hugging Face-hosted deployment, the solution offers a scalable and user-centric enhancement to academic writing. This research contributes to the advancement of AI-assisted educational tools by enabling efficient, intelligent, and compliant citation workflows for students, researchers, and educators alike.

Keywords: Semantic Search, Keyword Extraction, IEEE Format, Transformer Model, Machine Learning, Natural Language Processing, Collaborative Learning, Fine-Tuning, AI-Assisted Writing, Reference, Citation Generator, Academic Research Tools

ACKNOWLEDGEMENT

I would like to sincerely thank my supervisor, Dr. Lakmini Abeywardhana, and my sub-supervisor, Ms. Karthiga Rajendran, for their continuous guidance, support, and constructive feedback throughout the course of this research. Their expertise and direction were instrumental in shaping the scope and execution of this work. I am also grateful to the academic and technical staff of the Department of Information Technology, Faculty of Computing, Sri Lanka Institute of Information Technology, for their academic input and access to necessary resources. I extend my appreciation to my fellow project members for their collaboration in the development of the WriteWizard platform, and to the students and peers who contributed their time and feedback during evaluation phases. This research was carried out as part of the Bachelor of Science (Honours) in Information Technology degree program and was not supported by external funding. Finally, I would like to thank my family and friends for their moral support, patience, and encouragement throughout my academic journey.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviations	Description
SLIIT	Sri Lanka Institute of Information Technology
ML	Machine Learning
DL	Deep Learning
API	Application Programming Interface
SDLC	Software Development Life Cycle
WBS	Work Breakdown Structure
NLP	Natural Language Processing
TF-IDF	Term Frequency–Inverse Document Frequency
BERT	Bidirectional Encoder Representations from Transformers
LoRA	Low-Rank Adaptation
SBERT	Sentence Bidirectional Encoder Representations from Transformers
JSON	JavaScript Object Notation
CI/CD	Continuous Integration / Continuous Deployment
VM	Virtual Machine
UAT	User Acceptance Testing
CPU	Central Processing Unit
NER	Named Entity Recognition
AI	Artificial Intelligence
T5	Text-to-Text Transfer Transformer

1. INTRODUCTION

1.1 Background Study and Literature Review

1.1.1 Background Study

The increasing demand for academic integrity, efficient collaboration, and standard-compliant writing has prompted the academic and research communities to seek smarter, more automated solutions for document editing. Traditional writing tools, though widely used, are often limited to basic text manipulation and formatting capabilities. When it comes to referencing and citation management the core elements of academic writing, these tools fall short. Writers are commonly forced to depend on third-party citation generators or manual referencing methods, which are time-consuming, error-prone, and inefficient within the broader research workflow.

A significant portion of academic writing is spent on literature exploration, citation tracking, and adherence to formatting standards such as IEEE [5], [6], [7]. Unfortunately, existing citation tools often require researchers to leave their writing environment, manually extract metadata, and then format references before inserting them into the document. This workflow results in frequent context switching and a fragmented user experience. Errors in citation formatting, duplicate references, and incomplete bibliographies are frequent outcomes, compromising the academic rigor of the final output.

To address these pain points, intelligent systems powered by artificial intelligence (AI) are increasingly being adopted to support research paper writing. These systems can interpret user inputs, extract relevant concepts, and make real-time suggestions tailored to the content being written [2], [3]. When applied to referencing, they are capable of automating the identification of relevant research articles and generating properly formatted citations with minimal user intervention [9], [10]. This marks a significant advancement from traditional citation managers by shifting from static, manual tools to dynamic, context-aware support embedded directly in the writing platform.

The integration of such AI-powered components has transformed collaborative document editing platforms. These platforms are no longer just digital equivalents of word processors, they now support smart content suggestions, real-time formatting assistance, and intelligent task delegation. For instance, features such as automatic formatting corrections for IEEE compliance, visual mind maps, and contributor suggestion mechanisms have begun to emerge, enhancing the depth and usability of academic editing environments [11].

The value of such innovation becomes especially evident in multi-author research projects. Collaborative writing often introduces challenges such as maintaining consistent formatting, managing shared references, and ensuring proper citation practices across contributions. Intelligent editing platforms with built-in referencing components help resolve these challenges by enabling real-time citation generation, shared bibliographies, and instant formatting verification. These enhancements streamline the entire writing process, allowing collaborators to focus more on content and analysis rather than on the mechanics of documentation.

This research specifically explores and implements a real-time citation and reference generation component integrated within a collaborative academic editing platform. The system is designed to analyze highlighted or selected text within the editor, identify relevant keywords and themes, and return contextually appropriate research papers from an indexed collection [9]. It then allows users to generate citations in proper academic formats, such as IEEE, and insert them seamlessly into the document [1]. By functioning within the document interface itself, this solution eliminates the need for external citation tools and minimizes disruption to the user's writing flow.

In doing so, the component not only improves referencing accuracy and efficiency but also supports academic integrity through better source attribution. Furthermore, it aligns with the broader goal of enhancing the writing experience through intelligent automation. As academic writing continues to evolve alongside advances in AI and natural language understanding, components like this are expected to become essential to next-generation research environments, bringing together usability, compliance, and intelligence in one seamless interface.

1.1.2 Literature Review

The field of academic citation and reference management has evolved significantly with the emergence of intelligent content analysis and context-aware automation techniques. Traditional citation tools typically rely on static databases or manual bibliographic input, offering limited support for dynamic referencing based on in-text content. These tools, though capable of formatting citations in standardized styles, often function independently of the document authoring process, requiring users to switch contexts and perform manual search and formatting operations. As a result, they interrupt the writing flow and increase the likelihood of citation errors or inconsistencies.

Early efforts in improving citation support involved rule-based systems and metadata parsing techniques, which facilitated structured referencing but failed to interpret the semantic meaning of the surrounding content. These systems lacked the capability to assess the context of highlighted text or user selection and therefore were unable to recommend highly relevant academic sources in real time. Additionally, they offered minimal support for ensuring citation relevance across diverse domains or thematic variations in user input.

To address these limitations, recent research has adopted a range of advanced models and techniques rooted in natural language processing and machine learning. One widely used approach involves keyword extraction, where methods such as tokenization, part-of-speech tagging, and filtering are used to isolate context-relevant terms. This process is essential for identifying the thematic focus of selected text, allowing downstream systems to generate more accurate matches from existing research literature. In this system, a custom keyword extraction pipeline was implemented using Python and NLP libraries to refine this process and ensure meaningful term representation.

For semantic understanding and contextual similarity, transformer-based sentence embedding models such as Sentence-BERT (SBERT) are employed [3]. SBERT generates dense vector representations of both user-selected text and indexed research

paper abstracts. These vectors are then compared using cosine similarity, a technique that quantifies the semantic closeness between two pieces of text. This method supports an intelligent ranking of related academic papers, where the most contextually relevant sources appear first. This semantic approach allows the system to go beyond surface-level keyword matches and instead prioritize conceptual alignment.

Once relevant references are retrieved, the system uses a fine-tuned Flan-T5 model to generate properly formatted citations in IEEE style. This component transforms bibliographic metadata into structured, style-compliant citation entries. The output is then delivered through an integrated frontend interface that allows users to review and insert citations directly into their documents. This full-stack integration ensures that citation generation is both accurate and seamless within the writing environment.

Further, to ensure real-time performance and scalability, the system architecture leverages modular API calls that manage embedding generation, semantic analysis, citation creation, and metadata handling separately. This distributed approach enhances maintainability while enabling more efficient computation.

Despite these advancements, several limitations persist. Transformer models, while powerful, can be resource-intensive and may introduce latency in environments with limited computational capacity. In addition, citation accuracy depends heavily on the quality and relevance of the underlying paper database, making dataset curation and regular updates critical for sustained performance.

In summary, the current body of research highlights a transition from rule-based referencing tools to intelligent, context-aware systems. Through the use of models such as SBERT for semantic search, keyword extraction pipelines for content parsing, and Flan-T5 for citation generation, the system presented in this research offers a modern, efficient, and integrated solution to one of the most time-consuming aspects of academic writing [4]. These innovations improve workflow efficiency, enhance academic compliance, and support a more seamless collaborative writing experience.

1.2 Research Gap

The automation of citation generation and academic reference recommendation is a growing focus within research-oriented platforms. However, despite the availability of widely used tools such as Mendeley, Zotero, EndNote, and Google Scholar [5], [6], [7], [8], substantial gaps remain that hinder their effectiveness in dynamic, collaborative environments. Most existing systems provide database-driven reference management and static formatting templates, but lack real-time, context-aware citation suggestions directly embedded within the document editing process.

A significant limitation of these tools is their dependence on manual input and import. Users must search for references separately, extract metadata, and manually insert or update citations. This disjointed process disrupts workflow continuity and increases the cognitive load on the user. Furthermore, while citation formatting is supported, it is often rigid and fails to respond to in-text changes dynamically or intelligently. Few systems are capable of semantically interpreting a user's selection within a document to automatically suggest the most relevant research papers.

Another notable gap lies in collaborative support. Tools like EndNote and Mendeley offer shared libraries, but these are not truly collaborative in real time. Multi-user academic environments often require seamless integration between content creation and citation processes, a feature still largely missing in existing citation management software.

In addition, most traditional systems do not leverage fine-tuned NLP models or semantic similarity techniques to understand the underlying meaning of a user’s input. As a result, their suggestions are often based on keyword matches or manual tags, reducing the contextual relevance of retrieved papers. These systems also lack integration with editing environments like Google Docs or Notion, leading users to rely on copy-pasting references and formatting them manually.

Some newer platforms have explored AI-powered approaches to enhance research recommendation, but their integration with real-time citation formatting and direct insertion remains limited. Moreover, these systems are often designed as standalone platforms and do not align well with the writing environments used by academic teams.

To address these issues, this research proposes a unified, intelligent citation component embedded directly within a collaborative document editing interface. Unlike traditional tools, this component performs real-time analysis of highlighted text, extracts meaningful keywords, finds semantically similar academic sources, and generates properly formatted IEEE citations on the spot. It supports a seamless writing-to-referencing pipeline, dramatically improving efficiency, consistency, and user experience for academic writers.

The following table summarizes key features across major citation and reference tools, illustrating the gaps our proposed system aims to fill:

As shown in Figure 1, most tools fail to offer real-time, in-editor, semantically intelligent citation generation. Our system fills these gaps by offering context-aware citation suggestions, automated formatting in compliance with academic standards, and tight integration into collaborative environments, all in real time.

By bridging the limitations of both traditional and modern citation managers, this component sets a new standard for academic reference tools—one that combines intelligent automation, contextual accuracy, and collaborative usability in a single seamless solution.

Tools	Real-Time Citation Suggestion	Semantic Similarity Matching	Automatic IEEE Formatting	In-Editor Citation Insertion	Collaborative Integration	Free or Paid
	No	No	Partial (Manual)	No	**Shared Library Only**	**Freemium**
	No	No	Yes (Manual Insert)	No	No	**Free**
	No	No	Yes	No	**Yes (but not real-time)**	Paid
	No	No	Yes (Copy-Paste)	No	No	**Free**
	Partial (API only)	Partial	No	No	No	**Free**
	No	No	**Yes**	No	Limited	Paid
	**Yes**	**Yes**	**Yes (Automatic)**	**Yes (Integrated)**	**Yes (Real-time)**	**Free for now**

Figure 1: Comparison of Citation Tools

1.2 Research Problem

Academic writing is a complex process that requires not only the articulation of ideas but also the proper attribution of sources through citations and references. As academic standards become more rigorous and the volume of accessible research grows exponentially, writers are increasingly expected to identify and incorporate relevant scholarly sources efficiently and in proper format. Traditionally, this process has been supported by citation management tools such as Mendeley, Zotero, EndNote, and Google Scholar, which assist in storing references and generating citations. However, these tools operate as standalone platforms, detached from the writing environment, and rely on manual user input for keyword search, reference selection, and citation insertion [5], [6], [7], [8].

In modern research workflows, particularly in collaborative academic environments, this disjointed process creates friction. Writers must constantly shift between drafting content and searching for relevant literature, breaking focus and reducing productivity. This separation of tools from the writing space also introduces the risk of formatting inconsistencies, incorrect citations, or even omitted references. More importantly, current systems lack the ability to understand the context of what is being written. They do not provide intelligent, real-time suggestions based on the actual content of the document, leaving writers to depend on their own research and judgment to identify sources.

Although advancements in artificial intelligence and natural language processing have enabled deeper understanding of textual content, these technologies are rarely integrated into practical, real-time citation tools. While some platforms attempt to offer recommendations or citation suggestions, they lack semantic analysis capabilities that would allow them to interpret the user's writing and dynamically suggest the most contextually relevant sources. Additionally, citation formatting remains static in most systems, with no adaptive functionality based on in-editor actions or multi-author contributions.

Another major gap lies in collaborative integration. Existing solutions provide limited or no support for multi-user environments. In academic settings where multiple contributors may be working on different sections of a document, maintaining citation consistency and synchronization becomes a challenge. There is no unified solution that seamlessly connects content development, intelligent citation generation, and real-time collaboration.

Despite the availability of sophisticated AI models capable of keyword extraction, semantic matching, and citation formatting, there remains a lack of systems that combine these capabilities into a single, user-friendly, and workflow-embedded tool. The core missing piece is not the technology itself, but the absence of an intelligent citation assistant that is fully embedded within the writing platform, capable of understanding context, suggesting relevant references, generating compliant citations, and doing all of this collaboratively and in real time.

Therefore, the research problem addressed in this study is the lack of an intelligent, collaborative citation and reference generation system that can semantically analyze highlighted text, recommend relevant academic sources, and generate real-time, format-compliant citations within a document editing environment. Solving this problem will significantly improve efficiency, accuracy, and user experience of academic writing by closing the gap between content creation and citation management.

1.4 Research Objectives

1.4.1 Main Objective

To design and implement an intelligent, real-time citation and reference generation system that integrates seamlessly into a collaborative academic writing platform, enabling context-aware reference suggestions and automatic citation formatting to enhance writing efficiency, accuracy, and academic compliance.

1.4.2 Specific Objectives

The following are the sub-objectives of conducting this research.

- Provide real-time citation suggestions based on user-selected or highlighted content.
- Extract contextually relevant keywords from input text to improve search precision.
- Perform semantic similarity matching to retrieve the most appropriate academic references.
- Automatically generate citations in IEEE format and other academic styles.
- Seamlessly insert formatted citations directly into the document editor.
- Ensure accuracy and consistency in citation formatting across collaborative documents.
- Offers a user-friendly interface that allows authors to review, select, and insert references efficiently.
- Support integration with a collaborative writing environment for multi-user document editing.
- Minimize manual citation errors by automating source retrieval and formatting processes.

1.4.3 Business Objectives

- **Improve Enhance Academic Writing Efficiency:** Reduce the time and effort required to manually search for and format citations, improving writing speed and productivity for researchers and students.
- **Improve Citation Accuracy and Consistency:** Ensure all citations conform to required academic standards, thereby enhancing document quality and integrity.
- **Streamline Collaborative Workflows**: Enable seamless citation management in group writing scenarios, improving teamwork and document consistency.
- **Reduce Tool Fragmentation:** Eliminate the need to rely on third-party citation managers by offering a fully integrated in-editor solution.
- **Boost Platform Usability and Competitiveness:** Deliver intelligent citation functionality that distinguishes the platform from traditional word processors and citation tools.
- **Facilitate Academic Compliance:** Assist users in adhering to academic citation requirements automatically, reducing the risk of formatting or referencing errors.
- **Encourage Adoption in Educational and Research Institutions:** Provide a scalable solution suitable for integration into institutional platforms, enhancing support for students, researchers, and educators.

2. METODOLOGY

2.1 Methodology

Methodology in research refers to the structured approach and defined set of processes used to gather, analyze, and interpret data to achieve the objectives of a study. It outlines how the research is systematically planned and executed to ensure validity, reliability, and replicability of results. A well-designed methodology is vital to guide each phase of the research lifecycle, from problem identification to solution implementation and evaluation. It serves as the backbone that shapes how data is collected, interpreted, and translated into actionable outcomes.

In this study, the Agile methodology was adopted as the core project management and development framework. Originating from software engineering, Agile has become increasingly valuable in research environments where adaptability, continuous improvement, and stakeholder collaboration are critical. Agile's iterative nature enables researchers to address complex problems by dividing the project into incremental phases, allowing for frequent reassessment, refinements, and integration of feedback throughout the development process.

1. Agile Methodology for Research Development:

Agile methodology was applied to guide the progressive design, development, and integration of an intelligent citation and reference generation system. The system's development was organized into distinct phases, each addressing specific goals such as contextual keyword extraction, semantic similarity analysis, citation formatting, and frontend interface integration. The principles of flexibility, collaboration, and incremental progress supported continuous refinement of the system based on research insights and practical evaluation.

The below Figure 2 illustrates the Agile methodology applied in incremental phases, enabling continuous feedback integration and iterative system development.

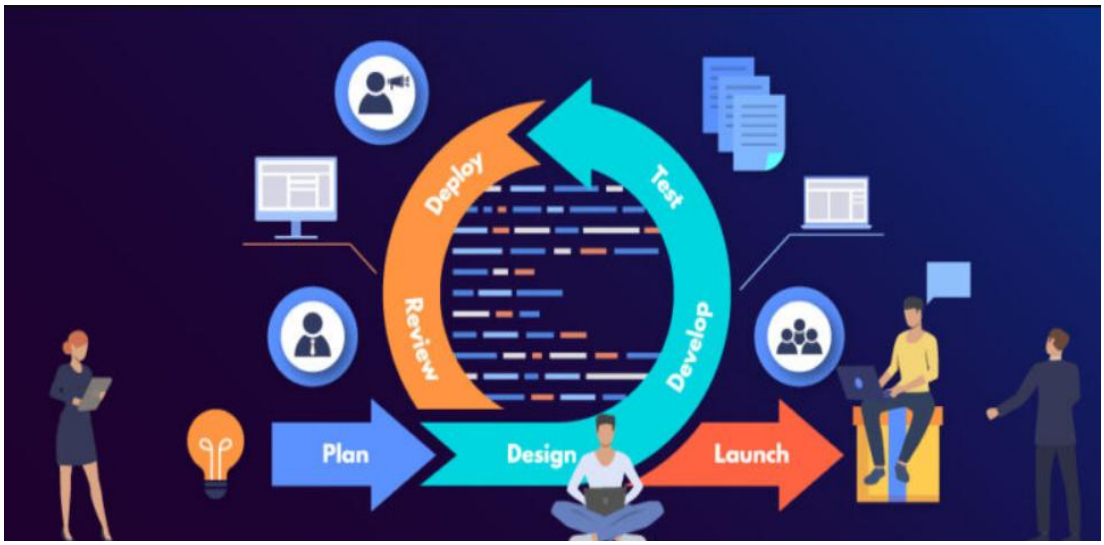


Figure 2: Agile Project Management Cycle

2. Multi-Stage Development Process

The research followed a phased framework that included requirement analysis, system design, prototype development, backend and frontend implementation, testing and evaluation, integration into a collaborative document editing environment, and ongoing refinement. Each stage built on insights from the previous one, enabling adaptive decision-making throughout the process. This structured progression ensured a consistent alignment with the research objectives and allowed for continuous enhancement based on iterative feedback.

3. Iterative Development and Collaboration:

A central feature of the adopted methodology was its focus on iterative development. Rather than attempting to complete the system in a single cycle, the project was divided into smaller functional components. These included keyword filtering, citation logic, semantic content matching, and user interface enhancements. Each component was developed independently, tested, and then integrated into the full system. Regular team discussions and feedback sessions ensured alignment with project goals and supported timely resolution of emerging challenges.

4. Flexibility and Responsiveness:

Agile methodology enabled the research team to respond effectively to changes in requirements, data inputs, and technical challenges. The ability to reassess and revise components as needed ensured that the system remained relevant to academic writing needs and capable of adapting to varying user expectations. As a result, the final system emerged as a robust and context aware citation solution that functions efficiently within collaborative academic environments.

2.1.1 Feasibility Study/ Planning

This phase assesses the feasibility of developing and deploying a real-time citation and reference generation component integrated into a collaborative academic writing platform. The study evaluates the project's viability across technical, economic, legal, operational, and social dimensions to ensure that the proposed solution can be implemented effectively and sustainably.

2.1.1.1. Technical Feasibility:

Data Availability: The project relies on publicly available academic metadata and research paper abstracts for reference matching and citation generation. These datasets are accessible from open-access repositories and academic APIs, ensuring a reliable base for model testing and integration.

Hardware and Software Requirements: The development requires moderate computational resources for backend processing and semantic analysis. Standard hardware with internet connectivity, along with cloud-based environments for embedding generation and API hosting, are sufficient. The system also utilizes open-source libraries for natural language processing, data handling, and citation formatting.

Component Integration: The technical feasibility of integrating backend citation logic with a collaborative writing frontend has been validated using web technologies such

as JavaScript and Express.js. Modular APIs enable seamless communication between the editor and the citation engine.

2.1.1.2. Economic Feasibility:

Budgetary Considerations: The project primarily depends on open-source software and publicly available data. Thus, major costs are associated with publication, internet usage, and general resources needed for development and deployment. No premium licenses or external services are required, making the system economically viable for both academic and student use.

Resource Allocation: The research team possesses the necessary technical expertise in natural language processing, web development, and academic research. Development responsibilities are distributed across system design, API integration, and testing.

Return on Investment (ROI): The expected outcomes of this system include time savings, improved citation accuracy, and better academic compliance. These benefits strongly justify the development costs, especially for students and educators who rely heavily on citation-intensive documents.

Table 1 describes the estimated summarization of cost.

Table 1: Cost Management

Type	**Cost**
Internet use and cloud hosting	12000 LKR
Publication costs	45000 LKR
Stationary	7500 LKR
TOTAL	***62500 LKR***

2.1.1.3. Legal and Ethical Feasibility:

Data Privacy and Ethics: The system uses open-access metadata and publicly available academic papers. No private or sensitive data is collected from users, ensuring compliance with data protection principles.

Intellectual Property: All software tools and libraries used are open-source, and all research content has respected intellectual property rights. The system avoids redistributing copyrighted full-text content, relying only on metadata for citation purposes.

2.1.1.4. Operational and User Acceptance Feasibility:

Data Collection and Processing: The system is designed to work with user-selected or highlighted text from within a document editor, eliminating the need for external input forms or manual data entry. The content processing pipeline has been evaluated to ensure that input text is accurately captured, preprocessed for keyword extraction, and transformed into a suitable format for semantic analysis. This streamlined process supports real-time citation retrieval without disrupting the user's writing flow.

Component Deployment: The citation engine is deployed using modular backend services that handle keyword extraction, semantic matching, and citation formatting. These services are designed to function reliably across different computing environments, whether locally hosted or deployed on cloud infrastructure. The architecture supports lightweight API interactions and asynchronous processing to maintain responsiveness and low latency during use. Monitoring tools are also incorporated to ensure the stability and availability of the system in operational settings.

System Integration: Seamless integration with collaborative document editing environments has been a core consideration in the system’s design. The citation component integrates directly into the editor's sidebar, allowing users to view, select,

and insert citations without switching applications. This real-time integration ensures that the citation workflow remains fluid and uninterrupted, supporting collaboration without requiring additional tools or platforms.

User Interface and Experience: The user interface has been developed with a focus on simplicity and accessibility. The citation sidebar provides contextual recommendations and allows users to insert citations with minimal clicks. Its intuitive layout ensures ease of use for both technical and non-technical users. Early-stage user testing with peers and academic writers has indicated a high level of satisfaction with the interface design, functionality, and responsiveness. The interface is further enhanced with features such as real-time preview, source title visibility, and formatting compliance indicators, contributing to a positive and productive user experience.

2.1.1.5. Time and Schedule Feasibility:

Project Timeline: A comprehensive project schedule has been developed to guide the progression from requirements gathering to final system deployment. The timeline includes structured phases such as system design, backend development, citation engine integration, frontend interface implementation, testing, and final evaluation.

Each phase is aligned with the overall academic calendar and designed to build upon insights gained in the previous stage. This structured timeline ensures consistent progress and continuous refinement of system features, maintaining alignment with project objectives throughout the development lifecycle.

Monitoring and Contingencies: To ensure timely delivery, regular progress monitoring is conducted through milestone reviews and planning sessions. Agile project management practices, including cycle-based development and routine feedback sessions, are used to sustain momentum and adaptability. In the event of unexpected challenges or delays such as integration issues, design modifications, or technical constraints, contingency plans have been established. These include reassignment of tasks, realignment of priorities, and adjustment of deadlines. Such measures ensure

that the project can adapt effectively and be completed within the designated timeframe.

The Figure 3 below shows the time management plan.

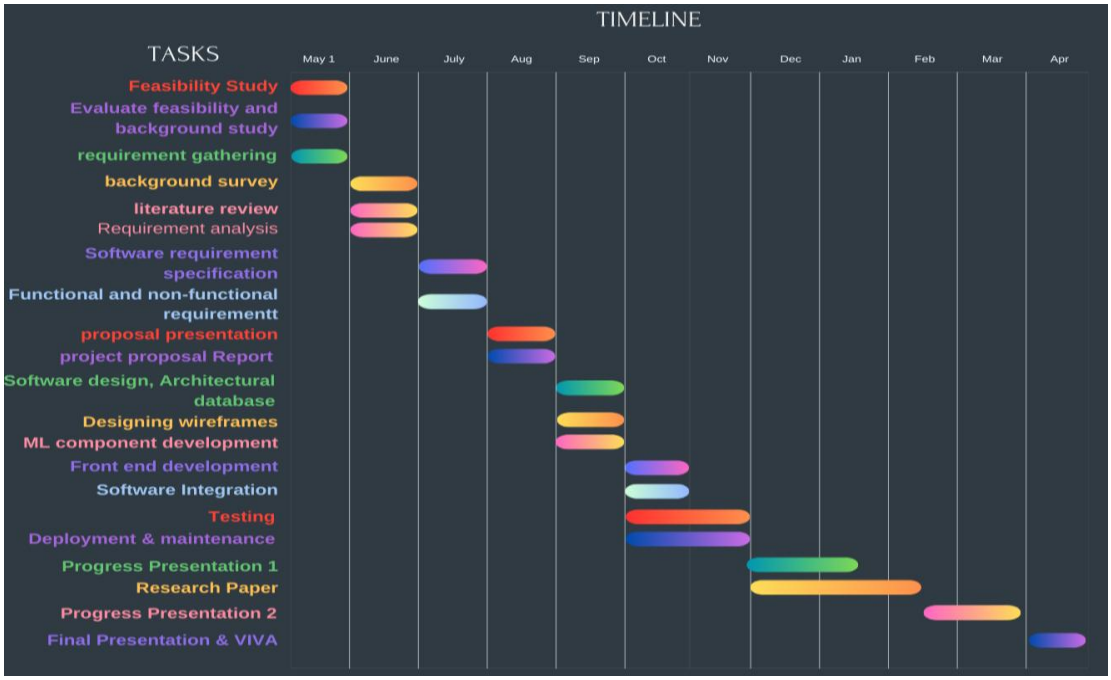


Figure 3: Gantt Chart (Schedule Management)

2.1.1.6. Social and Cultural Feasibility:

System Performance: The citation and reference generation system is designed to operate efficiently in real time within a collaborative academic writing environment. Its architecture supports seamless background processing of textual inputs for keyword extraction and semantic matching without interrupting the user's workflow. Internal testing confirms that the system maintains high responsiveness even when handling multiple concurrent requests and large document sections.

Adaptability to User Needs: The system's user interface and functionality are designed to accommodate the needs of diverse academic users, including undergraduate students, postgraduate researchers, and faculty members. Features such as on-demand citation suggestions, flexible insertion options, and adherence to widely used citation

styles ensure that users from various disciplines and writing backgrounds can adopt the tool without difficulty. The solution is also adaptable to various academic conventions, enabling its use across institutions with differing writing standards.

Platform Flexibility and Accessibility: The system is developed to support both cloud-based and local deployment environments, allowing educational institutions with varied infrastructure to implement the solution as needed. This flexibility enhances its accessibility and promotes inclusive adoption across geographical regions, technical capabilities, and educational settings. Its lightweight design ensures compatibility with low to moderate resource environments, which is especially relevant in developing countries and under-resourced institutions.

Cultural Acceptance and Inclusivity: The system has been developed with a focus on academic neutrality, avoiding any form of content bias or exclusion. It uses open-access databases and public academic resources, ensuring that users from different cultural and linguistic backgrounds have equal opportunity to access and benefit from its features. Additionally, its focus on academic citation rather than user profiling ensures minimal risk of ethical or cultural insensitivity.

Other than these feasibility studies, planning for risk mitigation and communication strategy has been completed to ensure the system's sustainability, alignment with user expectations, and successful delivery within the academic context.

2.1.1.7. Risk Management Plan:

The following risk management plan outlines potential risks associated with the design and implementation of the citation and reference generation system. Each risk includes its trigger, the responsible team member or stakeholder, a proposed response strategy, and the required resources to mitigate the issue. The plan ensures proactive monitoring and preparedness throughout all project phases.

The below Table 2 Describes the risk management plan executed in group.

Table 2: Risk Management Plan

Risk	Trigger	Owner	Response	Resource Required
Risk with respect to the Project Team				
Illness or sudden absence of the project team member(s)	Illness / Other personal emergencies	Project Leader	* Inform to the supervisor and co-supervisor. * Development team divides the functions with equal scope.	* Project Schedule Plan/Gantt Chart * Backup resources
Citation relevance not meeting expectations	Inaccurate suggestions from the system	Component Owner	* Re-tune parameters, update dataset, and apply user feedback.	* Evaluation Dataset, Testing Logs
Integration difficulties between frontend and backend	Incompatibility during component linking	Component Owner	* Schedule additional integration sessions, adjust API contracts, and involve all module leads.	* Interface Specifications, Technical Support
Deployment Delays	Technical issues or scheduling conflicts during deployment	Project Leader	Reassess timeline; use backup deployment environments; escalate critical issues to management	Gantt Chart; Deployment Backup Plan
Risk with respect to the Panel/ Supervisor(s)				
Panel requests changes after evaluation	Dissatisfaction with presentation or results	Project Leader	Implement requested changes promptly and update all relevant documentation.	Project Plan, Product Backlog
Supervisor(s) Request changes	Misalignment with initial research scope	Project Leader	* Communicate suggestions to team, evaluate impact, and	*Meeting Minutes, Requirement Documents

			update roadmap and deliverables.	
Supervisor or panel absence at scheduled review	Emergency or scheduling conflict	Project Leader	* Notify team and supervisor and sub-supervisor and then reschedule meetings or arrange alternative reviewers.	* Email Communication, Updated Meeting Log

2.1.1.8. Communication Management Plan:

The Communication Management Plan ensures that all project stakeholders, including the four team members, supervisor, and sub-supervisor, receive accurate and timely information throughout the project. This plan defines the communication objectives, platforms used, and the structure of meetings to ensure alignment, clarity, and effective project coordination.

Communication Objectives:

- Ensure communication is proactive, delivering updates in the correct format and scope for each stakeholder.
- Maintain clarity and precision in all interactions, avoiding redundancy while covering essential information.
- Guarantee that communication is timely, allowing issues to be addressed promptly to avoid project delays.
- Promote team-wide understanding, decision-making, and collaborative problem-solving through structured information sharing.

Communication Media:

- Email – Used for sharing official documentation, milestone updates, and formal communications with the supervisor and sub-supervisor.

- Microsoft Teams – Serves as the primary platform for online meetings, discussions, file sharing, and collaborative document editing.
- On-Campus Meetings – Conducted periodically for in-person discussions, especially during milestone reviews, evaluations, and supervisor feedback sessions.
- Phone Calls – Used for urgent issues requiring direct and immediate resolution among team members or with the supervisor.
- WhatsApp – Facilitates informal daily updates, task coordination, and rapid clarifications within the team.
- Microsoft Planner – Supports task allocation, scheduling, and progress tracking across all development phases.

The below Table 3 describes the communication management plan.

Table 3: Communication Management Plan

Meeting Type	**Attendees**	**Purpose**	**Frequency**	**Agenda Items**
Planning Kick-off Meeting	Supervisor, Sub-supervisor, All Team Members	Formally launch the planning phase; define project scope, novelties, roles, and expectations.	Once at project inception.	Establish timetable, define project scope and roles, identify risks, and review communication strategy
Execution Kick-off Meeting	Supervisor, Co-supervisor, All Team Members	Initiate the execution phase and ensure clarity on team members responsibilities, clarify any doubts before execution	At the start of major developmen t phases	Present detailed work plan, review logs, confirm escalation and change procedures, discuss quality control

Meeting Type	Attendees	Purpose	Frequency	Agenda Items
Internal Status Meeting	All Team Members	Review progress, identify challenges, and discuss task updates	Weekly	Share progress updates, confirm milestones, assign new tasks, review risks, and adjust priorities
Supervisor Review Meeting	Supervisor, Co-supervisor, All Team Members	Present key progress points, receive feedback, and align expectations	Bi-weekly or upon request	Demonstrate module completion, discuss implementation challenges, receive supervisor feedback.
Internal Pre-Milestone Meeting	All Team Members	Prepare for upcoming milestones by reviewing requirements and identifying potential challenges	Before each milestone or presentation	Evaluate internal progress, discuss technical blockers, reassign responsibilities, and align milestone targets
Supervisor Consultation Meeting	Supervisor, Sub-supervisor, Selected Team Members	Seek guidance on unresolved challenges identified during internal discussions	After internal pre-milestone meeting	Present complex issues, gather supervisor input, finalize direction, and adjust plan accordingly
Post-Milestone Review Meeting	Supervisor, Sub-supervisor, All Team Members	Review feedback received from the evaluation panel and plan for changes	After each milestone or viva	Discuss panel feedback, identify required corrections, distribute tasks, and plan next phase adjustments

Meeting Type	Attendees	Purpose	Frequency	Agenda Items
Final Evaluation Meeting	Supervisor, Co-supervisor, All Team Members	Evaluate completed system and present final project deliverables	At project completion	Demonstrate system, showcase documentation and testing, receive final assessment and concluding feedback

2.1.2 Requirement Gathering & Analysis

The Requirement Gathering and Analysis phase was a crucial foundation for this research, guiding the design and implementation of the intelligent citation and reference generation system. This phase involved identifying the core needs of users, defining the expected behaviors of the system, and setting clear functional expectations for development. Requirements were gathered from academic writers, university students, and supervisors to ensure that the system aligns with real-world citation workflows in collaborative academic environments

2.1.2.1. Functional Requirements

Functional requirements specify the features and capabilities that the citation and reference generation system must possess to fulfill its objectives within a collaborative academic writing environment. These requirements were gathered through structured interviews with target users including university students, academic supervisors, and research coordinators. The primary goal was to define a system that enhances the referencing workflow through intelligent automation, seamless integration, and contextual awareness.

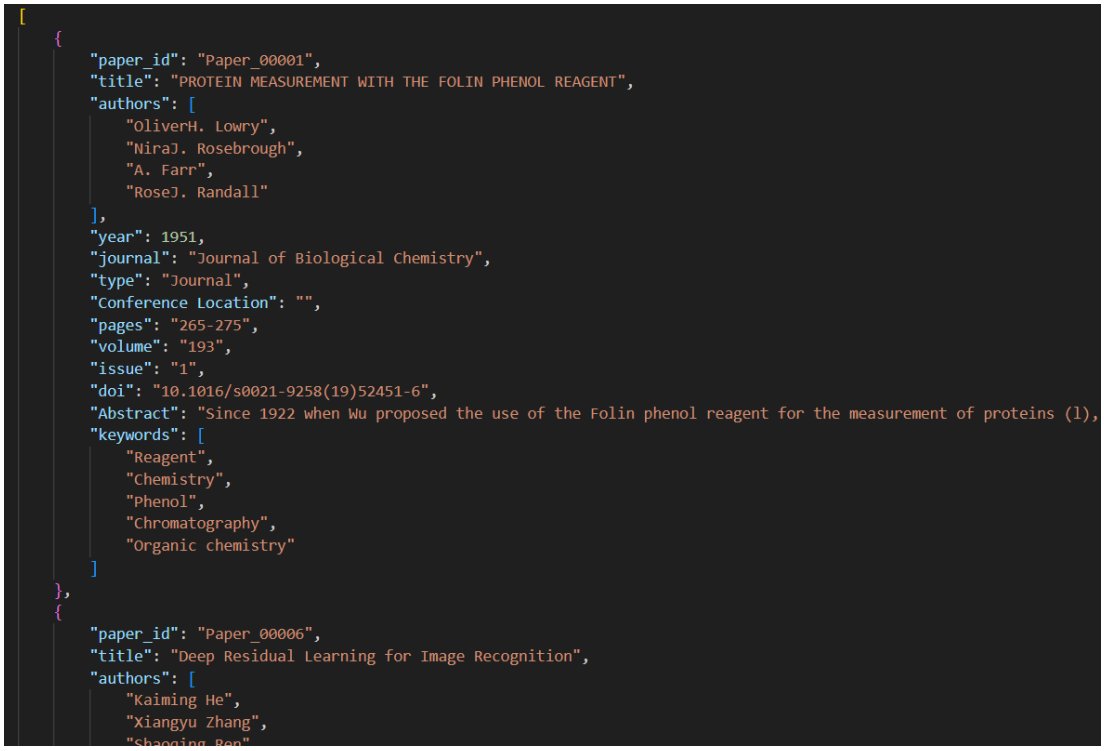
The key functional requirements identified for this research are as follows:

Context-Aware Citation Recommendation: The system must process user-highlighted text from the document and recommend semantically relevant academic references. To support this, a dataset was created using the Fetch Data from Source Python script, which extracted academic metadata from sources such as Crossref, OpenAlex, and arXiv. This dataset provides the base for semantic comparison, allowing the system to match highlighted content with related research papers and generate contextually accurate suggestions.

Real-Time Citation Insertion: The system should allow users to insert selected citations into the document without delay. Upon selection, the citation should appear in the correct format within the editor. The system must respond instantly to user

interactions, maintaining the natural flow of writing and enabling smooth collaboration in real-time academic environments.

The below Figure 4 describes the retrieved JSON dataset from CrossRef, arXiv and OpenAlex



```
[
    {
        "paper_id": "Paper_00001",
        "title": "PROTEIN MEASUREMENT WITH THE FOLIN PHENOL REAGENT",
        "authors": [
            "OliverH. Lowry",
            "NiraJ. Rosebrough",
            "A. Farr",
            "RoseJ. Randall"
        ],
        "year": 1951,
        "journal": "Journal of Biological Chemistry",
        "type": "Journal",
        "Conference Location": "",
        "pages": "265-275",
        "volume": "193",
        "issue": "1",
        "doi": "10.1016/s0021-9258(19)52451-6",
        "Abstract": "Since 1922 when Wu proposed the use of the Folin phenol reagent for the measurement of proteins (1),
        "keywords": [
            "Reagent",
            "Chemistry",
            "Phenol",
            "Chromatography",
            "Organic chemistry"
        ]
    },
    {
        "paper_id": "Paper_00006",
        "title": "Deep Residual Learning for Image Recognition",
        "authors": [
            "Kaiming He",
            "Xiangyu Zhang",
            "Shaoqing Ren"
```


Figure 4: Dataset Retrieved From Crossref OepenAlex and ArXiv

Citation Formatting Automation: Citations must be automatically formatted according to widely accepted academic styles, such as IEEE. The system should retrieve metadata from the selected source and convert it into a structured citation string, eliminating the need for manual editing or formatting by the user.

Collaborative Integration Support: The system must be designed to operate in multi-user, collaborative environments. Citation history and insertion logs should remain consistent across sessions and users, ensuring that all collaborators have visibility into the citations inserted by others.

Interactive Citation Review: Users should be able to view all suggested citations, preview their content and metadata, and select the most appropriate reference to insert. Each citation should display its title, author(s), publication venue, and year, providing users with enough information to make informed choices.

Citation History Management: The system should maintain a list of previously inserted citations within the document. Users must be able to revisit, reinsert, edit, or remove citations as needed. This functionality supports flexibility and traceability, especially in collaborative writing tasks where references evolve over time.

Semantic Matching and Ranking: The component should generate semantic embeddings for the highlighted text and match it against a database of research paper abstracts. Citations must be ranked by relevance using cosine similarity scoring, ensuring that the top results are highly aligned with the user's original input.

Data Logging and Traceability: Key interactions including citation requests, insertions, and formatting events should be logged for later review. This enables future analysis of system usage, user behavior, and potential improvements.

The functional requirements were refined and validated through feedback sessions involving both students and supervisors. Any unclear or redundant items were revised or removed, and a priority matrix was developed to classify them as essential, desirable, or optional. These validated requirements were then converted into user stories that reflect real-world usage patterns.

Figure 5 below describes the User Stories for requirements of reference recommender and citation generator.

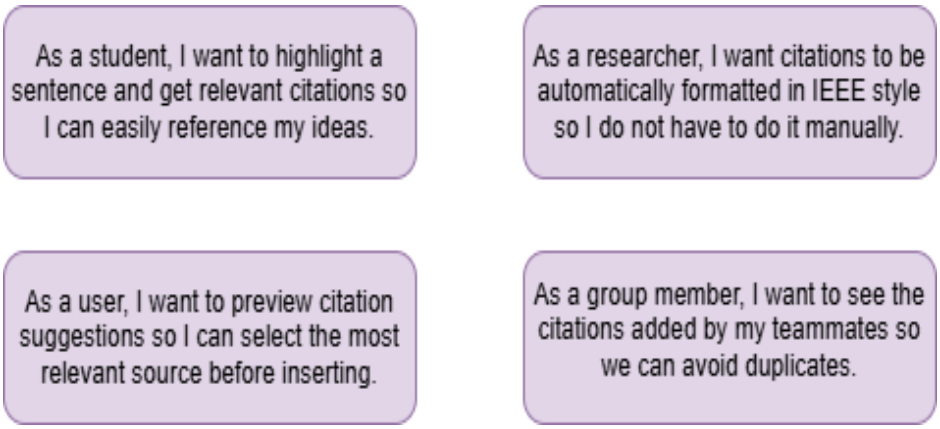


Figure 5: User Stories for Requirements

Mapping user requirements through use case and sequence diagrams plays a vital role in understanding and organizing system behavior. For this research, four core user stories were identified and translated into visual models to clearly define system interactions and prioritize development efforts. Figure 6 illustrates the use case diagram representing user interactions with the citation system, and Figure 7 presents the sequence diagram that outlines the step-by-step flow of operations from highlighting text to citation insertion.

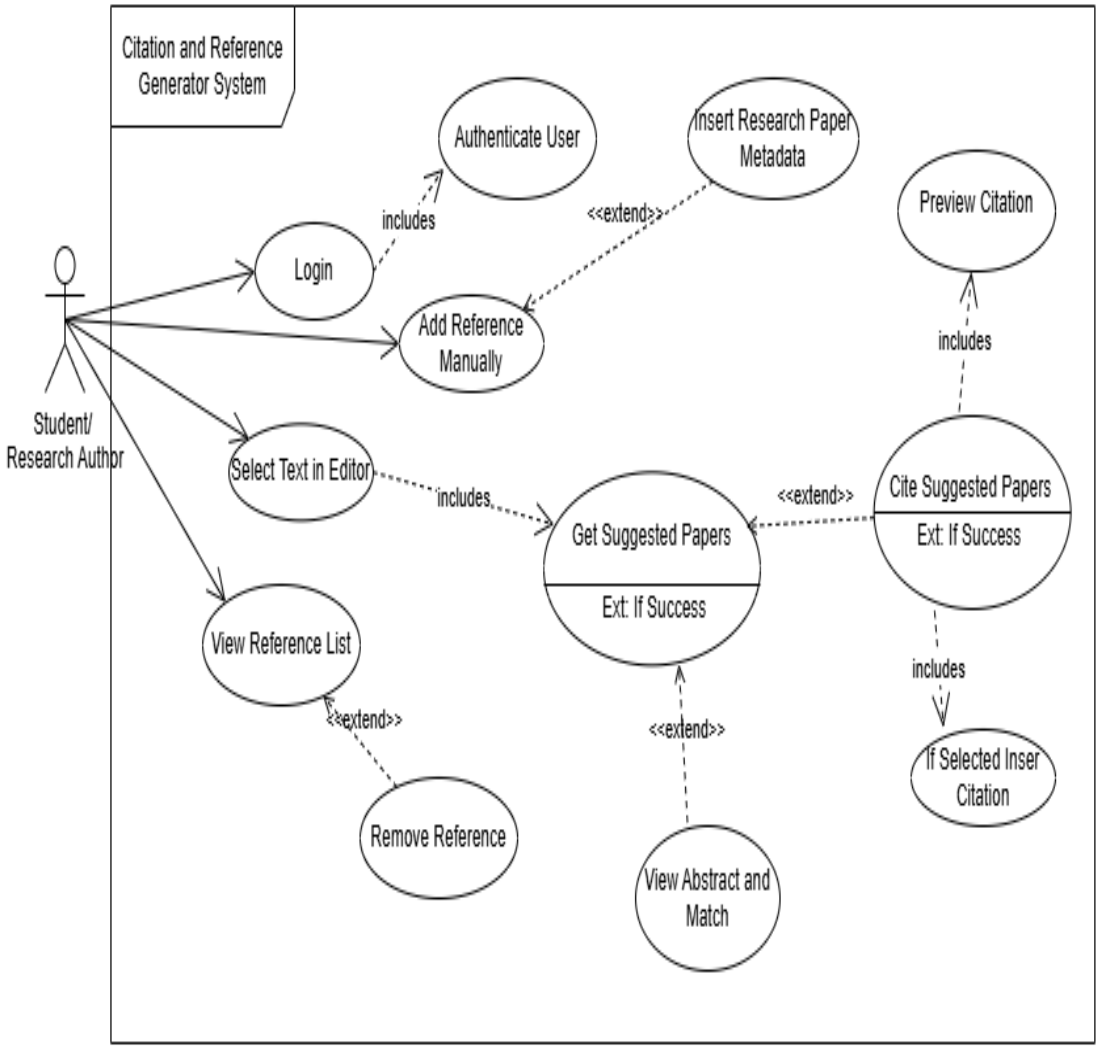


Figure 6:Use case Diagram

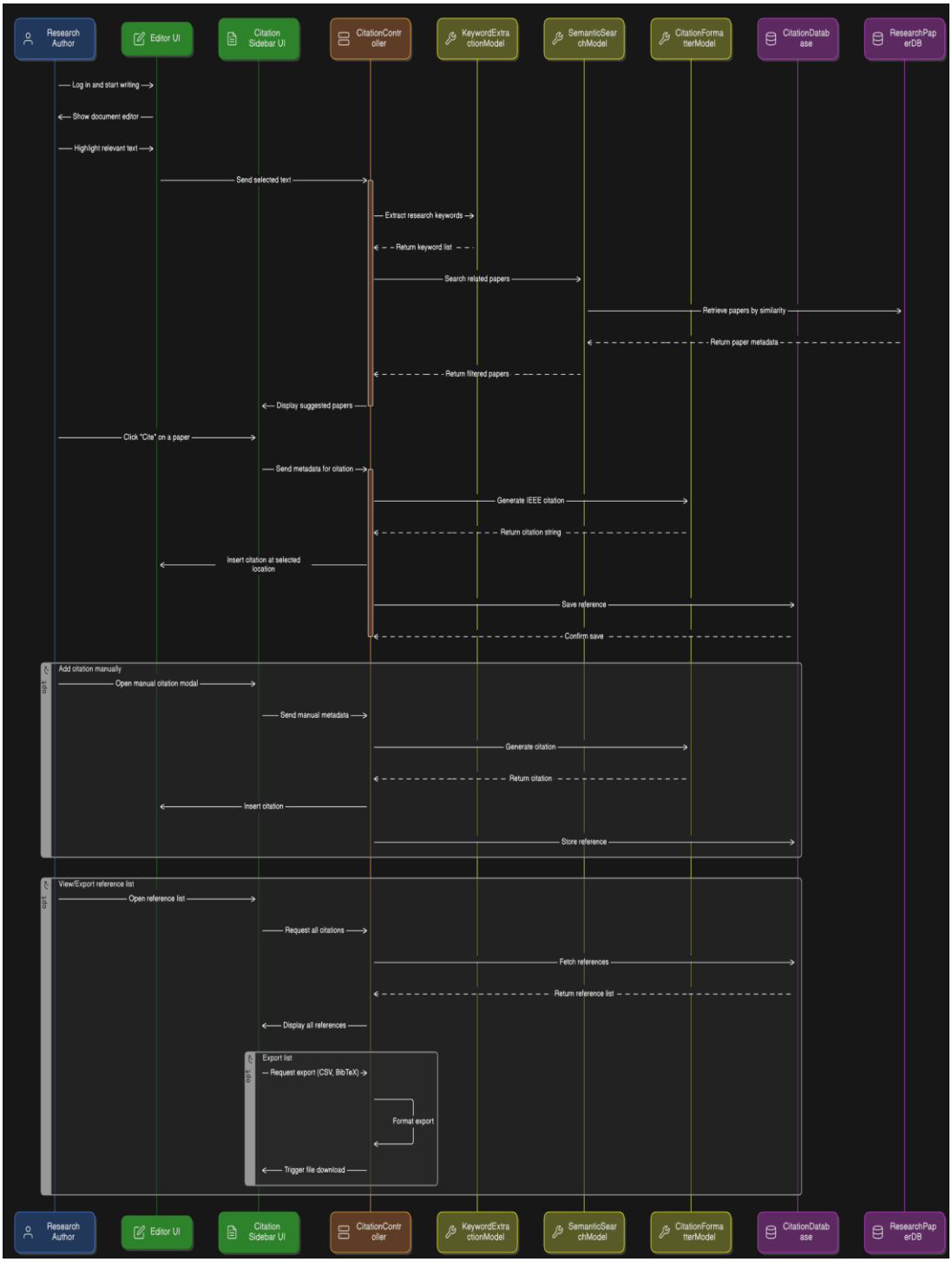


Figure 7: Sequence Diagram

2.1.2.2. Non-Functional Requirements

Non-functional requirements define the quality attributes, constraints, and operational standards that the citation and reference generation system must fulfill. These requirements are critical for ensuring that the system functions smoothly, remains ethically sound, and delivers an optimal user experience within academic writing environments.

Interviewing Method: Following the same process used for identifying functional requirements, structured interviews were conducted with academic writers, research students, and supervisors to gather non-functional requirements. Stakeholders shared their expectations regarding user interface design, system performance, privacy considerations, and collaborative behavior. Their feedback helped define performance expectations and ethical responsibilities.

Performance Metrics: This phase also established benchmarks for processing time, system availability, and data integrity. Participants emphasized the need for real-time citation suggestions, consistent system behavior during collaborative editing, and a high degree of relevance in suggested references. System responsiveness and precision in formatting were also highlighted as essential.

Alignment with Ethical Guidelines: Ethical responsibility in academic tools is crucial. Stakeholders expressed the need for proper handling of sourced content, transparency in how suggestions are generated, and compliance with academic integrity policies. All citation data used is sourced from publicly available datasets, avoiding misuse of proprietary content and ensuring that the system promotes responsible referencing practices.

Main non-functional requirements are listed below

- Usability

  The system should provide a simple and intuitive interface, ensuring ease of use

for users at all technical levels. Citation suggestions and formatting actions should be accessible with minimal effort and should not interrupt the writing experience.

- Reliability

  The system must perform its tasks consistently and correctly without frequent crashes or failures. During collaborative use, it should maintain stability and ensure that inserted citations are preserved across all user sessions.

- Availability

  Citation functionalities must be accessible at all times during normal academic work periods. The system should support uninterrupted access for multiple users working on the same document simultaneously.

- Accuracy

  All suggested citations must reflect a high level of contextual relevance and metadata accuracy. Author names, publication titles, sources, and formatting should strictly follow the selected academic style.

- Performance

  The system must deliver citation suggestions and formatting actions in under three seconds. It should not create noticeable delays in the writing process and should support real-time interactions in collaborative settings.

These non-functional requirements, along with the validated functional requirements, create a foundation for the successful design, development, and implementation of the citation and reference generation system. Together, they ensure the solution will meet user expectations in both performance and academic integrity.

2.1.3 Designing

The Designing phase of the SDLC marks a pivotal stage in the development of the AI-powered reference suggestion and citation generation system. This phase is dedicated to formulating a clear architectural foundation that ensures the seamless integration of key functionalities such as semantic keyword extraction, context-aware reference retrieval, citation formatting, and collaborative document support.

At this stage, the primary objective is to architect a modular and scalable system capable of intelligently identifying relevant academic references based on user-selected content and generating properly formatted citations in real time. The design process emphasizes component interoperability, processing efficiency, user accessibility, and integration into collaborative writing environments.

The system is intended to support academic writers and researchers by automating traditional manual and error-prone citation tasks, thereby enhancing productivity, reducing inconsistencies, and improving compliance with citation standards. To achieve these goals, each module is developed with a focus on real-time responsiveness, high semantic accuracy, and user-centered interaction.

System Architecture Diagram:

The system architecture diagram illustrates the interaction among the primary components that constitute the reference suggestion and citation generation pipeline. These components work in coordination to analyze the selected text, retrieve contextually appropriate references, generate citation strings, and display them within the editor interface.

The architecture is divided into three main modules: the Text Processing and Semantic Analysis Engine, the Reference Retrieval and Citation Formatting Engine, and the Frontend Interface and Collaboration Layer. A central controller, known as the Citation Orchestrator, coordinates the flow of information across these modules to ensure consistent behavior and efficient task execution.

1. Text Processing and Semantic Analysis Engine: This module accepts user-highlighted text from the editor and performs preprocessing, keyword extraction, and semantic embedding generation. It is responsible for transforming plain text into a representation that can be compared against academic metadata to identify relevant papers. Semantic matching is performed using vector similarity techniques based on pretrained language models.
2. Reference Retrieval and Citation Formatting Engine: This component receives the processed semantic vector and queries indexed academic sources such as Crossref, OpenAlex, and arXiv to retrieve relevant papers. The retrieved references are ranked by contextual similarity and passed through a formatting module that converts the metadata into citations compliant with academic styles such as IEEE.
3. Frontend Interface and Collaboration Layer: This layer handles user interaction within the collaborative writing environment. It allows users to highlight content, view ranked citation suggestions, preview citation formats, and insert citations directly into the document. It also tracks inserted citations and maintains citation history across collaborators.

All components are coordinated by the Citation Orchestrator, which manages communication between modules, ensures error handling, and maintains state consistency. Figure 8 illustrates the high-level architecture of the system, outlining the data flow and integration points between these modules.

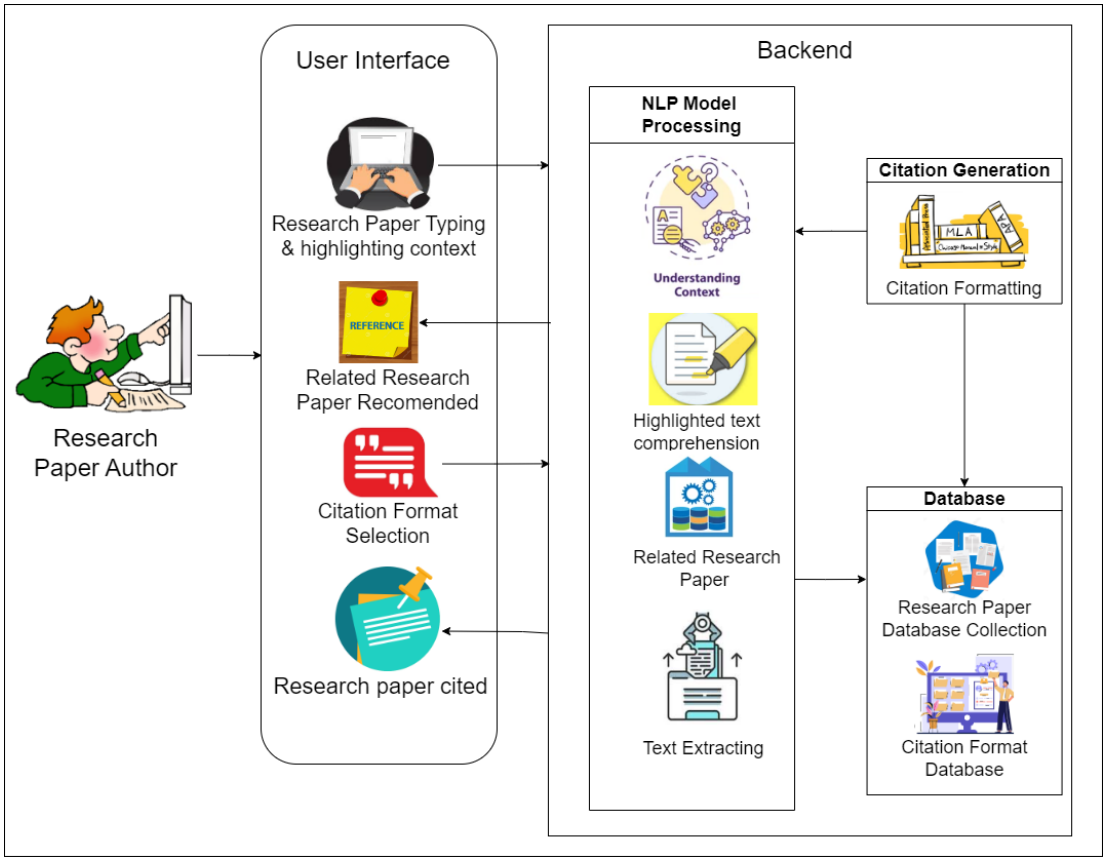


Figure 8: System Architecture Diagram of the Citation and Reference

Design Details:

The architectural design of the citation and reference generation system is organized into a modular three-tier structure. This design ensures separation of concerns, improves scalability, and enables smooth interaction between users, processing engines, and external data sources. Each layer has a distinct role, collectively supporting real-time citation suggestions, semantic analysis, and formatting within a collaborative document editing environment.

Presentation Layer

The Presentation Layer is responsible for delivering a clean, intuitive, and interactive user interface. Users interact with the system by highlighting text within their document or entering specific content segments. The interface responds dynamically by displaying contextually relevant citation suggestions and formatted reference previews.

This layer allows users to:

- Highlight academic content to trigger citation suggestions.
- Preview multiple citation recommendations.
- Choose preferred formatting styles (e.g., IEEE).
- Insert citations directly into the document with a single click.
- Access citation history and manage previously inserted references.

Designed with a user-centric approach, this layer hides the complexity of backend operations, offering only a streamlined and responsive interface suited for academic environments.

Business Logic Layer

The Business Logic Layer is the core of the system. It processes user-submitted content and orchestrates various modules to generate meaningful citation results.

Key responsibilities include:

- Semantic analysis and keyword extraction from the selected text using NLP models.
- Matching the extracted features against a curated academic database.
- Ranking research papers based on contextual similarity scores.

- Formatting citations using predefined academic styles such as IEEE.
- Managing validation, error handling, and fallback mechanisms when sources are unavailable.

This layer ensures that the output is contextually accurate and academically compliant. It also logs user interactions for traceability and recommendation refinement.

Data Access Layer

The Data Access Layer handles all interactions with external data sources and internal storage systems. It provides a bridge between the business logic and the structured databases.

Its primary functions include:

- Fetching metadata from academic repositories like Crossref, OpenAlex, and arXiv.
- Caching retrieved references to reduce API call latency.
- Accessing and storing citation style templates for formatting.
- Managing citation history logs per user session for retrieval or rollback.
- Ensuring consistent updates during collaborative sessions where multiple users insert citations concurrently.

This layer ensures that data is accessible, secure, and synchronized in real time.

Flow of Citation Suggestion and Generation

The process begins when a user highlights a segment of text in the document editor via the Presentation Layer. This input is transmitted to the Business Logic Layer, where semantic analysis and keyword extraction are applied. The system then compares the semantic vector of the input against indexed research papers and retrieves a ranked list of contextually relevant references.

Once relevant sources are identified, the system applies formatting rules based on the selected citation style. The user then previews the suggested citations, selects one, and inserts it directly into the document. The inserted citation is logged and stored by the Data Access Layer, allowing for future access or modifications.

Throughout this process, the system maintains a real-time collaborative state, ensuring all co-authors view the same citation updates without conflicts.

Figure 9 will illustrate the operational flow from content selection to citation insertion and collaborative synchronization.

Flow of Keyword Extraction:

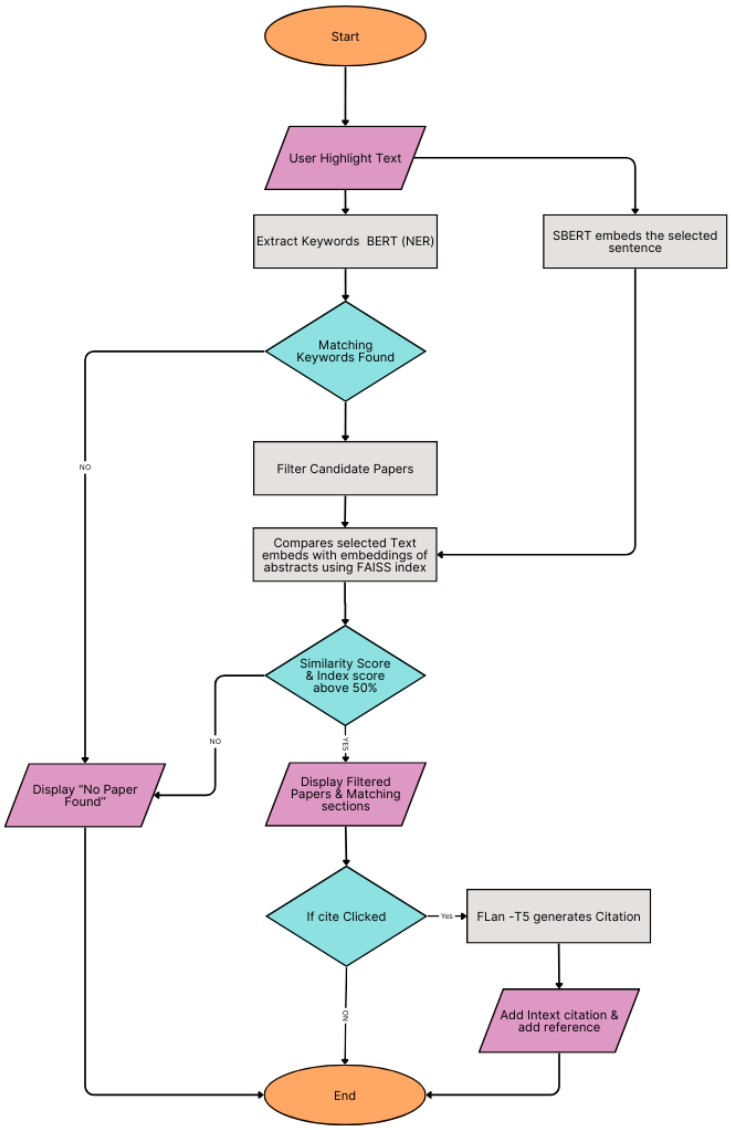


Figure 9: Flow Chart

2.1.4 Implementation

The Implementation phase marked a key transition in the development of the AI-powered reference suggestion and citation generation system. This stage translated the architectural designs into working modules that could deliver intelligent citation support within a collaborative academic writing environment. The process was structured and executed with clear planning, emphasizing sprint-based development, modular integration, and the use of modern frameworks and tools.

Task Breakdown and Project Management:

To manage the development process efficiently, the project was broken down into smaller, well-defined tasks using Microsoft Planner. Each component of the system such as natural language processing, citation retrieval, frontend development, and formatting logic was planned as a separate task with clear deliverables and responsibilities.

Tasks were organized under key categories such as:

- Backend development and API implementation
- Semantic processing and keyword extraction
- Integration with external data sources using open-access APIs like Crossref, OpenAlex, and arXiv
- Citation formatting and style compliance, including support for IEEE
- Frontend interface design and editor interaction for citation suggestions
- System testing, debugging, and feature refinement

This structured planning approach allowed for continuous tracking of progress, clear accountability, and efficient collaboration among team members. Microsoft Planner was used to visualize task distribution, set priorities, and ensure development milestones were met as the system evolved.

Figure 10 shows the Microsoft Planner board used for managing implementation activities and maintaining team alignment throughout the development phase.

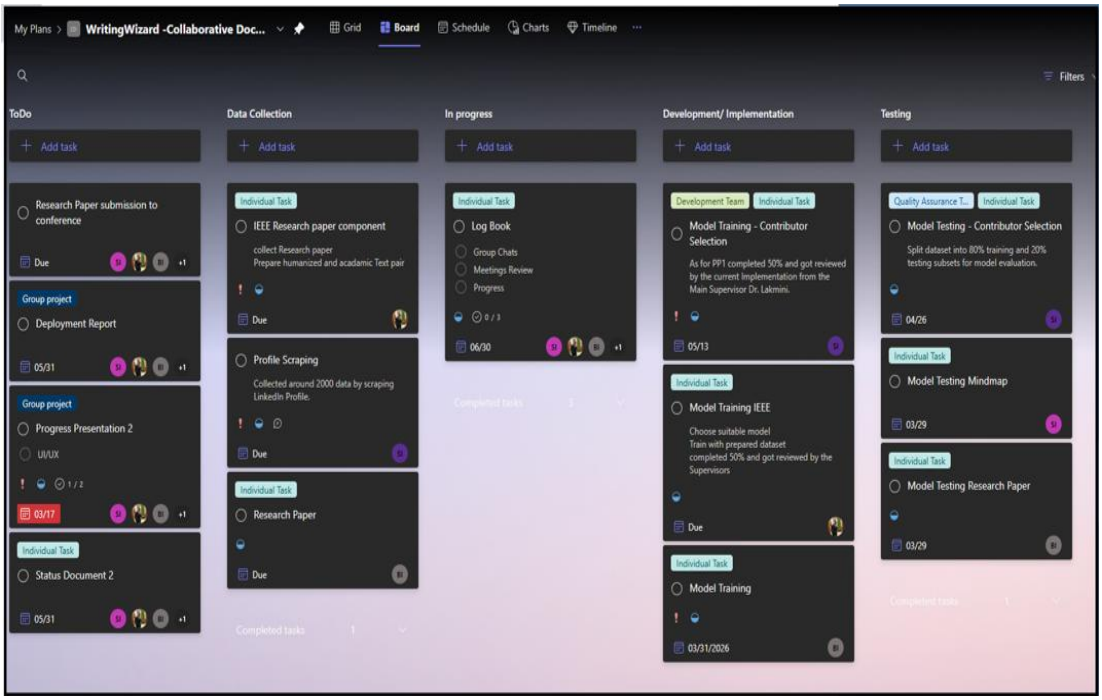


Figure 10: MSPlanner Board

A Work Breakdown Structure (WBS) was created to visually represent the hierarchy and sequencing of tasks. Tasks were grouped into categories such as backend development, frontend interface, data management, and testing. Each task was assigned to a responsible team member based on expertise, ensuring efficient parallel development and role clarity.

Key WBS categories included:

- Backend Development
    - API for reference suggestion
    - Citation formatting logic
    - External data integration (Crossref, OpenAlex)
- NLP Processing
    - Text preprocessing and embedding generation

    - Semantic similarity scoring
    - Paper ranking algorithm
- Frontend Interface
    - Text selection and highlight detection
    - Citation preview and confirmation UI
    - Insertion and history management
- System Integration and Testing
    - Connecting frontend with backend API
    - Citation logging and real-time sync
    - Validation against citation style rules

This task structure ensured transparency, accountability, and accurate tracking of progress. It also helped in identifying bottlenecks early, promoting proactive issue resolution throughout the development cycle. Figure 11 illustrates the WBS for the implementation of the citation and reference generation system.

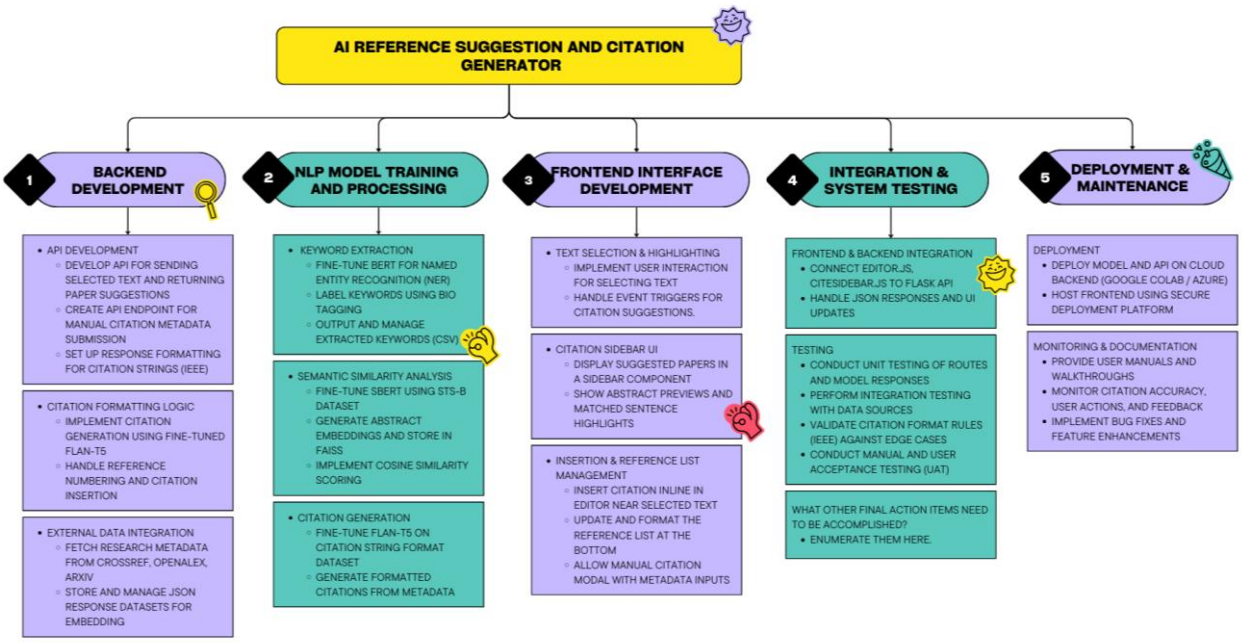


Figure 11: Work Breakdown Structure

Development Environment and Tools:

The implementation of the AI-powered reference suggestion and citation generation system was carried out using a combination of cloud-based and local development tools that supported modular coding, real-time testing, and model fine-tuning.

The core backend development took place within Google Colab, a collaborative cloud-based coding environment that offered powerful compute capabilities, seamless Python integration, and access to GPU acceleration. Google Colab was instrumental in executing Python-based scripts, fine-tuning pretrained language models, and processing academic metadata in real time. It allowed for interactive development, rapid prototyping, and the ability to train and test models without the constraints of limited local resources.

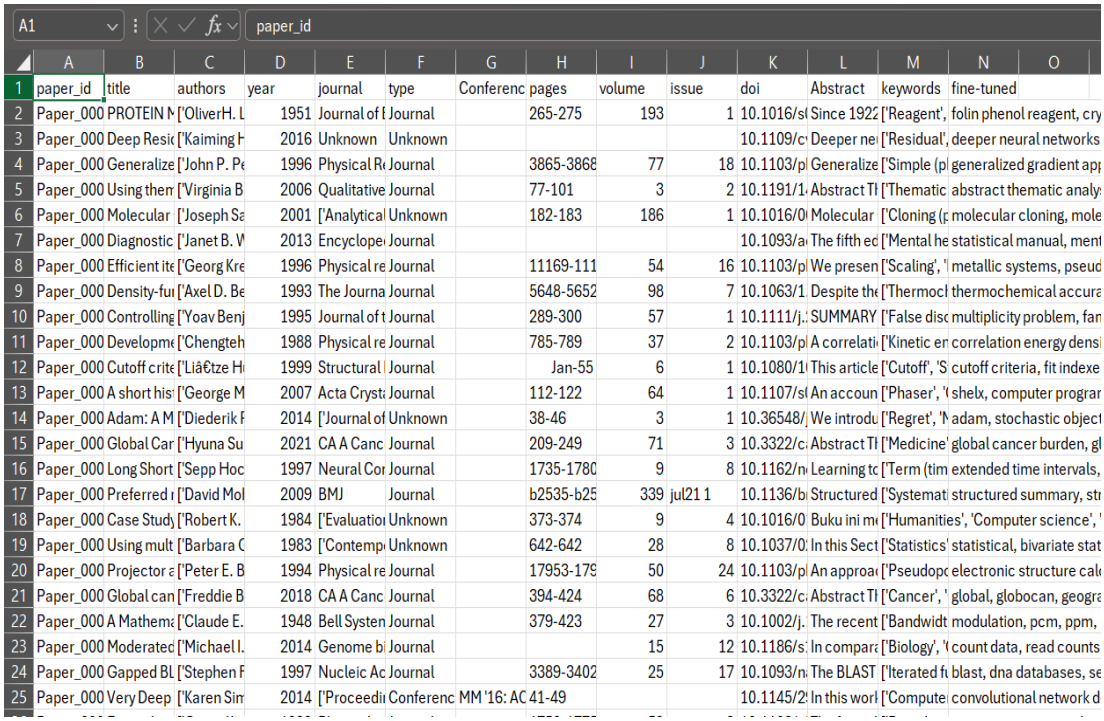
A wide range of Python libraries and frameworks were used to build and support the backend functionality. This included:

- Transformers and Sentence-BERT modules for semantic embedding generation
- Flask for creating API endpoints that connect the backend with the frontend interface
- Requests and JSON handling for managing citation metadata from open-access sources such as Crossref, OpenAlex, and arXiv
- Scikit-learn and NumPy for similarity scoring and result ranking

The frontend was developed using JavaScript, where the CiteSideBar.js component provided user interaction functionalities, including content highlighting, citation preview, and citation insertion into the editor. The interface communicated with the backend APIs to fetch and display citation suggestions in real time, ensuring an interactive and responsive user experience.

This combination of tools allowed the system to deliver seamless integration between AI-powered reference processing and a web-based document editing environment. Both the backend logic and the citation models were run and tested entirely within the Google Colab environment, while the front end was designed to be lightweight, fast,

and compatible with existing collaborative editing platforms. Figure 12 shows the Extracted Data.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	paper_id	title	authors	year	journal	type	Conferenc	pages	volume	issue	doi	Abstract	keywords	fine-tuned	
2	Paper_000	PROTEIN M	['OliverH. L	1951	Journal of I	Journal		265-275	193	1	10.1016/s(	Since 1922	['Reagent',	folin phenol reagent, cry	
3	Paper_000	Deep Resic	['Kaiming H	2016	Unknown	Unknown					10.1109/c	Deeper ne	['Residual',	deeper neural networks	
4	Paper_000	Generalize	['John P. Pe	1996	Physical R	Journal		3865-3868	77	18	10.1103/p	Generalize	['Simple (p	generalized gradient ap	
5	Paper_000	Using them	['Virginia B	2006	Qualitative	Journal		77-101	3	2	10.1191/1	Abstract Tl	['Thematic	abstract thematic analy	
6	Paper_000	Molecular	['Joseph Sa	2001	['Analytical	Unknown		182-183	186	1	10.1016/0	Molecular	['Cloning (p	molecular cloning, mole	
7	Paper_000	Diagnostic	['Janet B. W	2013	Encyclope	Journal					10.1093/a	The fifth ed	['Mental he	statistical manual, ment	
8	Paper_000	Efficient ite	['Georg Kre	1996	Physical re	Journal		11169-111	54	16	10.1103/p	We presen	['Scaling', '	metallic systems, pseud	
9	Paper_000	Density-fui	['Axel D. Be	1993	The Journa	Journal		5648-5652	98	7	10.1063/1	Despite the	['Thermocl	thermochemical accura	
10	Paper_000	Controlling	['Yoav Benj	1995	Journal of t	Journal		289-300	57	1	10.1111/j.	SUMMARY	['False disc	multiplicity problem, fan	
11	Paper_000	Developme	['Chengteh	1988	Physical re	Journal		785-789	37	2	10.1103/p	A correlati	['Kinetic en	correlation energy densi	
12	Paper_000	Cutoff crite	['Liâ€tze H	1999	Structural	Journal		Jan-55	6	1	10.1080/1	This article	['Cutoff', 'S	cutoff criteria, fit indexe	
13	Paper_000	A short his	['George M	2007	Acta Cryst	Journal		112-122	64	1	10.1107/s(	An accoun	['Phaser', '	shelx, computer prograr	
14	Paper_000	Adam: A M	['Diederik F	2014	['Journal of	Unknown		38-46	3	1	10.36548/	We introdu	['Regret', 'N	adam, stochastic object	
15	Paper_000	Global Car	['Hyuna Su	2021	CA A Canc	Journal		209-249	71	3	10.3322/c	Abstract Tl	['Medicine'	global cancer burden, gl	
16	Paper_000	Long Short	['Sepp Hoc	1997	Neural Cor	Journal		1735-1780	9	8	10.1162/n	Learning tc	['Term (tim	extended time intervals,	
17	Paper_000	Preferred r	['David Mol	2009	BMJ	Journal		b2535-b25	339	jul21 1	10.1136/b	Structured	['Systemat	structured summary, str	
18	Paper_000	Case Study	['Robert K.	1984	['Evaluatio	Unknown		373-374	9	4	10.1016/0	Buku ini m	['Humanities', 'Computer science', '		
19	Paper_000	Using mult	['Barbara C	1983	['Contemp	Unknown		642-642	28	8	10.1037/0	In this Sect	['Statistics'	statistical, bivariate stat	
20	Paper_000	Projector a	['Peter E. B	1994	Physical re	Journal		17953-179	50	24	10.1103/p	An approa	['Pseudopc	electronic structure cal	
21	Paper_000	Global can	['Freddie B	2018	CA A Canc	Journal		394-424	68	6	10.3322/c	Abstract Tl	['Cancer', '	global, globocan, geogra	
22	Paper_000	A Mathema	['Claude E.	1948	Bell Systen	Journal		379-423	27	3	10.1002/j.	The recent	['Bandwidt	modulation, pcm, ppm,	
23	Paper_000	Moderated	['Michael I.	2014	Genome bi	Journal			15	12	10.1186/s	In compara	['Biology', '	count data, read counts	
24	Paper_000	Gapped BL	['Stephen F	1997	Nucleic Ac	Journal		3389-3402	25	17	10.1093/n	The BLAST	['Iterated fu	blast, dna databases, se	
25	Paper_000	Very Deep	['Karen Sim	2014	['Proceedi	Conferenc	MM '16: AC	41-49			10.1145/2	In this worl	['Compute	convolutional network d	

Figure 12: Dataset Extracted After keywords extraction

2.1.4.1 Data Collection Script for Academic Metadata

The citation and reference suggestion system is powered by a robust data pipeline that integrates academic metadata retrieval, keyword extraction, semantic similarity scoring, and automated citation formatting. The entire process is executed in the Google Colab environment using a suite of Python-based scripts, model fine-tuning procedures, and integrated API services. The system leverages open-access scholarly repositories such as Crossref, OpenAlex, and arXiv to collect metadata and abstracts, which are then processed through three dedicated models to enable intelligent citation suggestions.

Step-by-Step Workflow Overview

1. Dataset Collection and Metadata Storage: A custom script was used to fetch academic metadata including titles, authors, abstracts, publication dates, DOIs, and source repositories. APIs provided by Crossref, OpenAlex, and arXiv were accessed through requests calls and filtered for completeness. The data was stored in structured JSON format, providing the base dataset for both training and runtime operations.
2. Keyword Extraction Using NER: The first model in the pipeline uses a fine-tuned BERT-base uncased model to extract keywords from research abstracts using Named Entity Recognition (NER) with BIO tagging. This model aligns tokenized inputs with labeled tags and outputs important terms related to the research content. The keywords are stored in a CSV file, which is later used to filter relevant papers based on keyword overlaps with user-selected content.
3. Semantic Similarity via Sentence Embeddings: The second model is a fine-tuned Sentence-BERT model adapted using the STS-B dataset. It generates embeddings for abstracts stored in the JSON dataset and indexes them using FAISS (Facebook AI Similarity Search) with IndexFlatL2 structure [10]. The embeddings are saved as a pickled FAISS index file, enabling fast similarity-based retrieval. During runtime, the user's selected sentence is embedded and compared against this index to retrieve semantically similar papers using cosine similarity and distance scores.
4. Citation Generation with Flan-T5: A Flan-T5 model was fine-tuned on thousands of research paper metadata entries to generate citation strings in IEEE format [4]. This model uses metadata fields such as title, authors, source, and year to create consistent and well-structured citations, eliminating the need for manual formatting. The model outputs are generated using PyTorch and evaluated using f1_score and precision_recall_fscore_support for quality assurance.

Mechanism of Citation Suggestion

Once the models are trained and data is indexed, the following operational steps occur during runtime:

1. The user selects a sentence or text snippet in the document editor.
2. The front end triggers an API call which passes the selected text to the backend.
3. The keyword extraction model processes the text and identifies core terms.
4. These keywords are matched against the pre-generated CSV file of paper keywords to filter a candidate list.
5. The semantic analysis model embeds the selected text and compares it with FAISS-stored embeddings of paper abstracts.
6. Based on cosine similarity, the top matches are returned and displayed as cards showing the title, author, and paper ID.
7. When the user clicks "View," the system highlights the most relevant sentence from the abstract that aligns with the selected text.
8. When the user clicks "Cite," the Flan-T5 model generates the citation in IEEE format using the original metadata [4].
9. Clicking "Insert Citation" adds the citation in-text and appends the reference to the end of the document.

Additionally, users can opt to add citations manually using the manual citation generation API, where they input metadata such as title, author, DOI, and year to generate a formatted citation and insert it into the document.

Figure 13 describes the data collection from arXiv, openAlex and crossref python code.

```python
def inverted_index_to_text(inverted_index):
    if not inverted_index:
        return ""
    position_to_word = {}
    for word, positions in inverted_index.items():
        for pos in positions:
            position_to_word[pos] = word
    full_text = " ".join(position_to_word[pos] for pos in sorted(position_to_word))
    return full_text

def get_crossref_metadata(title, authors):
    query = title
    url = f"https://api.crossref.org/works?query.title={requests.utils.quote(query)}&rows=1"
    response = requests.get(url)
    if response.status_code != 200:
        print(f"CrossRef API error: {response.status_code}")
        return {}
    data = response.json()
    if data.get("message", {}).get("items"):
        return data["message"]["items"][0]
    return {}

def get_crossref_by_doi(doi):
    url = f"https://api.crossref.org/works/{doi}"
    response = requests.get(url)
    if response.status_code != 200:
        print(f"CrossRef DOI query error for {doi}: {response.status_code}")
        return {}
    data = response.json()
    return data.get("message", {})
def fetch_openalex_works(cursor, per_page=200):
    url = f"https://api.openalex.org/works?per_page={per_page}&cursor={cursor}&select=id,doi,display_name,publication_year,authorships,primary_location,biblio,concepts,abstract_inverted_index"
    response = requests.get(url)
    if response.status_code != 200:
        print(f"Error fetching data: {response.status_code}")
        return [], cursor
    data = response.json()
    works = data.get("results", [])
    new_cursor = data.get("meta", {}).get("next_cursor")
    return works, new_cursor

def transform_works(works, start_index=0):
    papers = []
    for i, work in enumerate(works, start=start_index):
        # Only process works with an abstract_inverted_index
        if not work.get("abstract_inverted_index"):
            continue
        paper_id = f"Paper_{i+1:05d}"
        title = work.get("display_name", "")
        authors = [auth.get("author", {}).get("display_name", "") for auth in work.get("authorships", []) if auth.get("author")]
        year = work.get("publication_year", "")
        primary_location = work.get("primary_location") or {}
        source = primary_location.get("source") or {}
        journal = source.get("display_name", "") if source else "Unknown"
        venue_type = source.get("type", "").lower() if source else ""
        if "conference" in venue_type:
            paper_type = "Conference"
        elif "journal" in venue_type:
            paper_type = "Journal"
        else:
            paper_type = "Unknown"
        conference_location = source.get("host_organization_name", "") if paper_type == "Conference" else ""
        biblio = work.get("biblio", {})
        pages = biblio.get("first_page", "")
        if biblio.get("last_page"):
            pages += f"-{biblio['last_page']}"
        volume = biblio.get("volume", "")
        issue = biblio.get("issue", "")
        doi = work.get("doi", "")
        doi_clean = doi.replace("https://doi.org/", "").strip() if doi else ""
        abstract = inverted_index_to_text(work.get("abstract_inverted_index"))
        keywords = [concept.get("display_name") for concept in work.get("concepts", []) if concept.get("display_name")]
        paper_data = {
            "paper_id": paper_id,
            "title": title,
            "authors": authors,
            "year": year,
            "journal": journal,
            "type": paper_type,
            "Conference Location": conference_location,
            "pages": pages,
            "volume": volume,
            "issue": issue,
            "doi": doi_clean,
            "Abstract": abstract,
            "keywords": keywords
        }
        papers.append(paper_data)
    return papers
```

Figure 13: Data Collection Script for research papers

2.1.4.2 Data Cleaning Mechanism

While this system does not handle image data, a parallel process is used for text-based datasets. During metadata retrieval and preprocessing, duplicate research papers are filtered out based on DOI and content overlap, ensuring a clean and diverse set of academic entries. This helps avoid redundant suggestions and enhances model accuracy. Furthermore, only metadata with complete abstracts and key fields are retained for training and indexing, which improves the semantic alignment and relevance of citation recommendations. The below Figure 14 describes the code used to remove data duplication.

```
def enrich_papers(papers):
    for paper in papers:
        if not paper.get("doi"):
            crossref_data = get_crossref_metadata(paper["title"], paper["authors"])
            if crossref_data:
                doi = crossref_data.get("DOI", "")
                paper["doi"] = doi
                paper["journal"] = crossref_data.get("container-title", paper["journal"])
                paper["pages"] = crossref_data.get("page", paper["pages"])
                paper["volume"] = crossref_data.get("volume", paper["volume"])
                paper["issue"] = crossref_data.get("issue", paper["issue"])
    for paper in papers:
        if paper["type"] == "Conference" and not paper.get("Conference Location") and paper.get("doi"):
            crossref_doi_data = get_crossref_by_doi(paper["doi"])
            if crossref_doi_data:
                event = crossref_doi_data.get("event", {})
                if event:
                    location = event.get("location", "")
                    name = event.get("name", "")
                    if location and name:
                        paper["Conference Location"] = f"{name}, {location}"
                    elif location:
                        paper["Conference Location"] = location
                    elif name:
                        paper["Conference Location"] = name
    return papers
# ----------------- Main Script ----------------- #
target_count = 2000  # Desired number of papers with abstracts
collected_papers = []
cursor = "*"

while len(collected_papers) < target_count:
    works, new_cursor = fetch_openalex_works(cursor)
    if not works:
        break  # No more records available
    new_papers = transform_works(works, start_index=len(collected_papers))
    # Enrich new papers
    new_papers = enrich_papers(new_papers)
    collected_papers.extend(new_papers)
    print(f"Collected {len(collected_papers)} papers so far...")
    if not new_cursor or new_cursor == cursor:
        break
```

Figure 14: Data Cleaning

2.1.4.3 Selecting Model training Algorithm

(Choice of Transformer-based Architectures Over Traditional Rule-Based or Statistical Methods for Academic Reference Generation)

In the context of intelligent citation support and academic reference suggestion, the selection of the appropriate model architecture plays a critical role in ensuring semantic accuracy, contextual relevance, and operational scalability. This system adopts transformer-based models such as BERT, Sentence-BERT (SBERT), and Flan-T5 [4], instead of relying on traditional keyword-based search methods or statistical similarity approaches such as TF-IDF or Latent Semantic Analysis (LSA). The choice of these advanced architectures was based on several well-founded considerations, as outlined below:

1. **Semantic Understanding and Contextual Awareness**

Transformer-based models such as BERT and SBERT are specifically designed to capture semantic meaning in natural language text. Unlike traditional methods which treat words in isolation, these models learn word relationships within a given context, making them significantly more effective in identifying the intent and meaning of user-selected text. This contextual understanding is crucial when aligning user content with relevant academic abstracts from the database.

2. **Precision in Keyword Extraction and Named Entity Recognition**

For keyword extraction, a fine-tuned BERT model with BIO-tagging offers far more accurate token classification than rule-based NER pipelines or dictionary lookups. The ability of transformers to understand word dependencies enhances the relevance of extracted keywords, which are used to filter the reference dataset. This ensures that only high-quality, topic-aligned research papers are considered for recommendation.

3. **Scalable Semantic Similarity Matching**

Semantic similarity is handled using Sentence-BERT (SBERT), which produces dense sentence-level embeddings optimized for tasks such as clustering and information retrieval. These embeddings are indexed using FAISS for rapid similarity computation. This method significantly outperforms traditional cosine similarity on TF-IDF vectors or bag-of-words models, which fail to capture nuanced language structures and word-order dependencies.

4. **High-Quality Citation Generation Using Fine-Tuned Language Models**

The Flan-T5 model, fine-tuned on thousands of citations samples, generates output that is not only syntactically correct but also stylistically consistent with IEEE formatting standards. Compared to templated citation generators or simple string concatenation techniques, Flan-T5 offers a more flexible and accurate solution, capable of adjusting to varying metadata completeness and structure [4].

5. Efficiency, Reusability, and Integration Capabilities

Transformer-based models are modular and highly compatible with modern ML pipelines. Once trained, they can be reused across components (e.g., BERT for both keyword tagging and abstract comprehension). Their integration into APIs via Google Colab enables real-time responsiveness while maintaining scalability and reduced memory overhead through indexed embeddings and compressed pickle storage.

6. **State-of-the-Art Performance in Research and Industry Benchmarks**

BERT, SBERT, and T5 have demonstrated consistent performance superiority across multiple academic NLP tasks, including information retrieval, summarization, and citation generation. Their proven reliability and community support make them ideal candidates for implementation in academic writing assistance tools.

In conclusion, the adoption of fine-tuned transformer models for each stage of the citation pipeline reflects a strategic alignment with the system's core objectives. These

models provide unparalleled contextual matching, robust performance in real-time applications, and a seamless integration experience. Their ability to produce meaningful and accurate citations, combined with superior keyword extraction and semantic filtering, confirms their suitability for a modern, AI-driven citation and reference recommendation system. The below Figure 15, 16 17 describes the weightage of models.

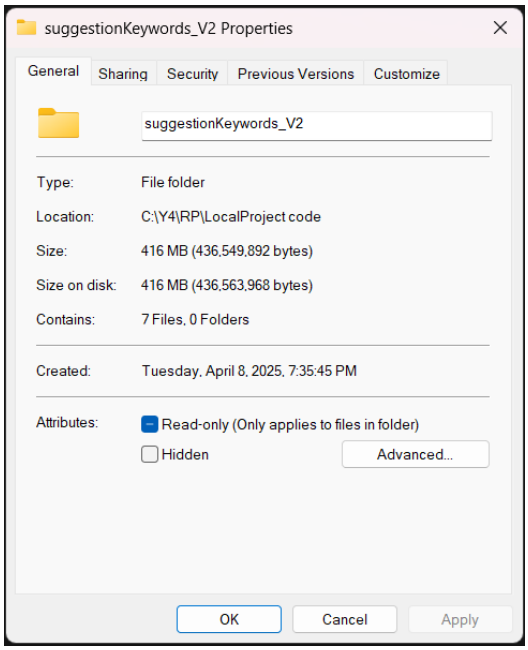


Figure 15: File Size of Keyword Extraction Model

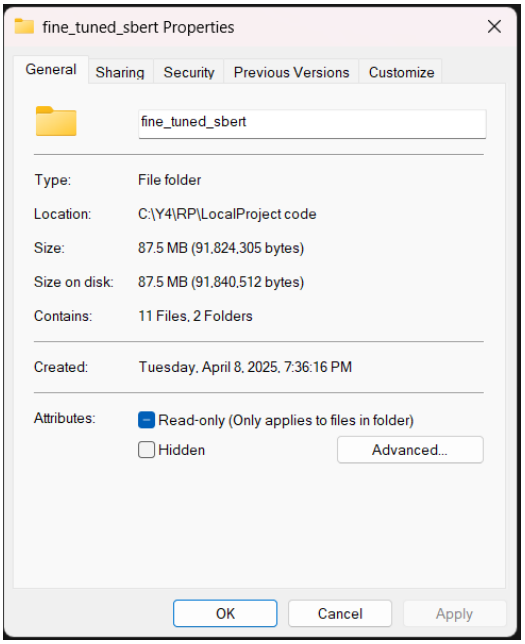


Figure 16: File zie of SBERT fine-tuned model

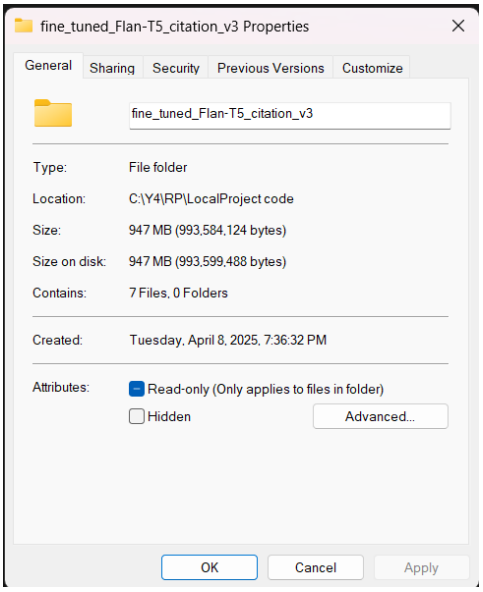


Figure 17: file size of Flan-T5 fine-tuned citation model

2.1.4.4 Model Training – Part 1: Keyword Extraction Model (BERT + NER)

The first stage of model training involved fine-tuning a BERT-based architecture to extract domain-specific keywords from academic abstracts. This component supports the initial filtering of research papers by identifying high-relevance terms from user-selected input. The Figure 18 describes the keyword extraction model Training.

Step 1: Dataset Preparation

Metadata was collected from Crossref, OpenAlex, and arXiv, including titles, authors, abstracts, **and** DOIs.

- The data was cleaned to remove duplicates and incomplete entries.
- Abstracts were manually annotated using BIO-tag format for Named Entity Recognition (NER).

Step 2: Tokenization and Label Alignment

- The dataset was tokenized using **HuggingFace's AutoTokenizer**.
- BIO-labeled tags were aligned with tokenized outputs to ensure accurate input-label pairing.

Step 3: Model Selection and Training

- A **BERT-base uncased** model was used via AutoModelForTokenClassification.
- Training was performed in Google Colab using the Transformers and PyTorch libraries.
- AdamW was used as the optimizer with a cross-entropy loss function.

Step 4: Evaluation and Storage

- The model's performance was evaluated using **F1-score**, **precision**, and **recall**, computed using precision_recall_fscore_support.
- Extracted keywords were stored in a **CSV file**, which serves as the filtering base for relevant paper identification during runtime.

```
drive.mount('/content/drive')
data_df = pd.read_json('final_filtered_openalex_works (4).json')
data = data_df['Abstract'].tolist()
from transformers import AutoTokenizer, AutoModelForTokenClassification

model_checkpoint = "bert-base-uncased"
batch_size = 6
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint, add_prefix_space=True)
from datasets import load_dataset

dataset = load_dataset("midas/inspec", "extraction")
label_list = np.unique([item for sublist in dataset['train']['doc_bio_tags'] for item in sublist])
label_dict = {label_list[0]: 1, label_list[1]: 2, label_list[2]: 0}
def tokenize_and_align_labels(examples):
    # Add max_length and padding parameters to avoid length issues
    tokenized_inputs = tokenizer(
        examples["document"],
        truncation=True,
        is_split_into_words=True,
        max_length=512,  # Set the maximum length for the model
        padding='max_length'  # Pad to the maximum length
    )

    labels = []
    for i, label in enumerate(examples['doc_bio_tags']):
        word_ids = tokenized_inputs.word_ids(batch_index=i)
        label_ids = []
        for word_idx in word_ids:
            if word_idx is None:
                label_ids.append(-100)  # Use -100 for tokens that are not part of any label
            else:
                label_ids.append(label_dict[label[word_idx]])  # Map label to id
        labels.append(label_ids)

    tokenized_inputs["labels"] = labels
    return tokenized_inputs
def tokenize_and_align_labels(examples):
    # Add max_length and padding parameters to avoid length issues
    tokenized_inputs = tokenizer(
        examples["document"],
        truncation=True,
        is_split_into_words=True,
        max_length=512,  # Set the maximum length for the model
        padding='max_length'  # Pad to the maximum length
    )

    labels = []
    for i, label in enumerate(examples['doc_bio_tags']):
        word_ids = tokenized_inputs.word_ids(batch_index=i)
        label_ids = []
        for word_idx in word_ids:
            if word_idx is None:
                label_ids.append(-100)  # Use -100 for tokens that are not part of any label
            else:
                label_ids.append(label_dict[label[word_idx]])  # Map label to id
        labels.append(label_ids)

    tokenized_inputs["labels"] = labels
    return tokenized_inputs
tokenized_dataset = dataset.map(tokenize_and_align_labels, batched=True)
torch.manual_seed(0)
model = AutoModelForTokenClassification.from_pretrained(model_checkpoint, num_labels=len(label_list)).to('cuda')
for param in model.parameters():
    if not param.is_contiguous():
        param.data = param.data.contiguous()
args = TrainingArguments(
    f"{model_checkpoint.split('/')[-1]}-finetuned-keyword-extraction",
    evaluation_strategy="epoch",
    logging_strategy="epoch",
    learning_rate=5e-5,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size,
    num_train_epochs=5,
    lr_scheduler_type="linear",
    seed=0,
)
data_collator = DataCollatorForTokenClassification(tokenizer)
from sklearn.metrics import precision_recall_fscore_support, f1_score

def compute_metrics(p):
    predictions, labels = p
    predictions = np.argmax(predictions, axis=-1)
    true_predictions = [p for prediction, label in zip(predictions, labels) for (p, l) in zip(prediction, label) if l != -100]
    true_labels = [l for prediction, label in zip(predictions, labels) for (p, l) in zip(prediction, label) if l != -100]
    precision, recall, f1, _ = precision_recall_fscore_support(y_true=true_labels, y_pred=true_predictions, average='macro')
    f1_macro = f1_score(y_true=true_labels, y_pred=true_predictions, average='macro')

    return {
        'eval/precision': precision,
        'eval/recall': recall,
        'eval/f1': f1,
        'eval/f1_macro': f1_macro,
    }

trainer = Trainer(
    model=model,
    args=args,
    train_dataset=tokenized_dataset["train"],
    eval_dataset=tokenized_dataset["validation"],
    data_collator=data_collator,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics,
)

training_output = trainer.train()
torch.cuda.empty_cache()
loss = [d["loss"] for d in trainer.state.log_history if "loss" in d]
val_loss = [d["eval_loss"] for d in trainer.state.log_history if "eval_loss" in d]
plt.plot(loss)
plt.plot(val_loss)
plt.legend(["Training Loss", "Validation Loss"])
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
test_output = trainer.evaluate(tokenized_dataset["test"])

def extract_keywords_model(data, batch_size=1):
    fine_tuned_keywords = []  # Store fine-tuned keywords

    for idx in tqdm(range(0, len(data), batch_size)):
        inputs = tokenizer(data[idx: idx + batch_size], return_tensors='pt', padding=True, truncation=True).to('cuda')
        outputs = model(**inputs)
        predictions = torch.softmax(outputs.logits, dim=-1).argmax(-1)
        for idx_item in range(predictions.shape[0]):
            # Extract fine-tuned keywords based on token classification
            indices_single = []
            keywords_item = []
            for idx, label in enumerate(predictions[idx_item].tolist()):
                if label == 1:  # Label for start of keyword
                    indices_single.append(idx)
                elif label == 2 and len(indices_single) > 0:
                    indices_single.append(idx)
                elif label == 0 and len(indices_single) > 0:
                    # End of a keyword sequence, decode and add to list
                    keywords_item.append(tokenizer.decode(inputs.input_ids[idx_item][indices_single]).strip())
                    indices_single = []  # Reset for the next sequence
            fine_tuned_keywords.append(keywords_item)
    return fine_tuned_keywords
fine_tuned_keywords = extract_keywords_model(data, batch_size=1)
data_df['fine-tuned'] = fine_tuned_keywords
print(data_df[['Abstract', 'fine-tuned']].head())
data_df[['Abstract', 'fine-tuned']].to_json("fine_tuned_keywords.json", orient="records", lines=True)

def extract_keywords_model(data, batch_size=1):
    keywords = []
    for idx in tqdm(range(0, len(data), batch_size)):
        inputs = tokenizer(data[idx: idx + batch_size], return_tensors='pt', padding=True, truncation=True).to('cuda')
        outputs = model(**inputs)
        predictions = torch.softmax(outputs.logits, dim=-1).argmax(-1)
        for idx_item in range(predictions.shape[0]):
            indices_item = []
            indices_single = []
            keywords_item = []
            for idx, label in enumerate(predictions[idx_item].tolist()):
                if label == 1:
                    indices_single.append(idx)
                elif label == 2 and len(indices_single) > 0:
                    indices_single.append(idx)
                elif label == 0 and len(indices_single) > 0:
                    indices_item.append(indices_single)
                    keywords_item.append(tokenizer.decode(inputs.input_ids[idx_item][indices_single]).strip())
                    indices_single = []
            keywords.append(keywords_item)
    return keywords

model_keywords = extract_keywords_model(data, batch_size=1)
data_df['fine-tuned'] = pd.DataFrame([', '.join(keywords) for keywords in model_keywords])
data_df.head()
from google.colab import sheets
sheet = sheets.InteractiveSheet(df=data_df)
from transformers import Trainer

output_dir = "/content/drive/MyDrive/Research_Model/suggestionKeywords_V1"  # specify your Google Drive path
trainer.save_model(output_dir)  # Saves model, config, and tokenizer
```

Figure 18: KeyWord suggestion Model Training

The below Figure 19 describes the training loss of the keyword extraction model.

Epoch	Training Loss	Validation Loss	Eval/precision	Eval/recall	Eval/f1	Eval/f1 Macro
1	0.345900	0.244383	0.703094	0.676741	0.687837	0.687837
2	0.226600	0.220879	0.719939	0.753717	0.734519	0.734519
3	0.204600	0.216267	0.729418	0.751771	0.739523	0.739523
4	0.192900	0.216623	0.721407	0.780937	0.747678	0.747678
5	0.180300	0.214532	0.731682	0.763012	0.745430	0.745430
6	0.173200	0.217141	0.729047	0.775836	0.749997	0.749997
7	0.165100	0.220166	0.726322	0.781361	0.751177	0.751177
8	0.162400	0.220661	0.727390	0.778446	0.750677	0.750677

Figure 19: Training loss of BERT model

2.1.4.5 Model Training – Part 2: Semantic Similarity Model (SBERT + FAISS)

The second model in the pipeline focuses on measuring the semantic similarity between user-selected content and academic paper abstracts. This stage ensures that only meaningfully aligned research papers are presented to the user. The attached Figure 20 describes the training loss of semantic similarity and Figure 21 describes the training code of SBERT.

Step 1: Embedding Abstracts

- The cleaned abstract dataset in JSON format was embedded using a **fine-tuned Sentence-BERT (SBERT)** model.
- The SBERT model was adapted using the **STS-Benchmark dataset**, enhancing its semantic matching capabilities.

Step 2: Indexing with FAISS

- All abstract embeddings were indexed using **FAISS (Facebook AI Similarity Search)**.
- IndexFlatL2 was selected as the indexing structure for cosine similarity-based scoring.
- The resulting FAISS index was serialized and stored as a **pickle file** to allow rapid query-time retrieval.

Step 3: Real-Time Matching

- During runtime, user-selected text is embedded and compared with stored abstract embeddings.
- Based on similarity scores and distance thresholds, the top-ranked papers are returned for frontend display.

Step	Training Loss
500	0.021700
1000	0.014600

Figure 21: Embedding Training loss

```python
# Install required libraries
!pip install -q sentence-transformers faiss-cpu torch datasets jsonlines
import json
from torch.utils.data import DataLoader
from sentence_transformers import SentenceTransformer, InputExample, losses, util
import numpy as np
import faiss
import pickle
import re

# Load your JSON dataset of papers (make sure the file exists in /content/)
with open('/content/final_filtered_openalex_works (4).json', 'r') as f:
    data = json.load(f)

# Extract metadata
paper_ids = [paper["paper_id"] for paper in data]
abstracts = [paper["Abstract"] for paper in data]
titles = [paper["title"] for paper in data]
authors = [paper.get("authors", []) for paper in data]

print(f"Number of papers loaded: {len(data)}")
print("Sample paper ID:", paper_ids[0])
print("Sample abstract:", abstracts[0][:200], "...")
print("Sample authors:", authors[0])

train_examples = []
for paper in data:
    if paper.get("title") and paper.get("Abstract"):
        train_examples.append(
            InputExample(texts=[paper["title"], paper["Abstract"]], label=1.0)
        )

print(f"Generated {len(train_examples)} training examples from JSON.")

model = SentenceTransformer('all-MiniLM-L6-v2')

train_dataloader = DataLoader(train_examples, shuffle=True, batch_size=16)

train_loss = losses.MultipleNegativesRankingLoss(model)

# Fine-tune the model (for demonstration, we use 1 epoch)
model.fit(
    train_objectives=[(train_dataloader, train_loss)],
    epochs=1,
    show_progress_bar=True
)

model.save('fine_tuned_sbert_model')
from google.colab import drive
drive.mount('/content/drive')
model = SentenceTransformer('/content/drive/MyDrive/fine_tuned_sbert')

embeddings = model.encode(abstracts, batch_size=32, show_progress_bar=True)
embeddings = np.array(embeddings)  # shape: (num_papers, embedding_dim)
print("Embedding matrix shape:", embeddings.shape)

faiss.normalize_L2(embeddings)

d = embeddings.shape[1]
index = faiss.IndexFlatIP(d)
index.add(embeddings)
print("Total papers in the FAISS index:", index.ntotal)

from google.colab import drive
drive.mount('/content/drive')

# Prepare a dictionary with your metadata and embeddings
data_to_save = {
    "paper_ids": paper_ids,
    "abstracts": abstracts,
    "titles": titles,
    "authors": authors,  # Save authors information
    "embeddings": embeddings
}

with open('/content/drive/MyDrive/sbert_embeddings_latest_v1.pkl', 'wb') as f:
    pickle.dump(data_to_save, f)

print("SBERT embeddings and metadata saved to Google Drive.")

with open('/content/drive/MyDrive/sbert_embeddings_latest_v1.pkl', 'rb') as f:
    saved_data = pickle.load(f)

paper_ids = saved_data["paper_ids"]
abstracts = saved_data["abstracts"]
titles = saved_data["titles"]
embeddings = saved_data["embeddings"]

faiss.normalize_L2(embeddings)
index = faiss.IndexFlatIP(embeddings.shape[1])
index.add(embeddings)
print("FAISS index recreated with", index.ntotal, "paper embeddings.")

model = SentenceTransformer('/content/drive/MyDrive/fine_tuned_sbert')
```

Figure 20: Model Training code for Semantic analysis SBERT

2.1.4.6 Model Training – Part 3: Citation Generator (Flan-T5 Fine-Tuning)

The third component in the system is a Flan-T5 language model, fine-tuned specifically to generate citation outputs in IEEE format from academic metadata [4].

Step 1: Training Data Preparation

- Thousands of paper entries from Crossref, OpenAlex, and arXiv were used to create structured input-output training pairs.
- Each input contained metadata fields (title, author, year, DOI), and the output was a correctly formatted IEEE-style citation.

Step 2: Model Fine-Tuning

- The Flan-T5 model was fine-tuned using **PyTorch**, with AdamW optimizer and cross-entropy loss.
- Training was done in the Google Colab environment using GPU support for efficiency.
- Additional evaluation using BLEU scores and output inspection helped ensure formatting consistency.

Step 3: Deployment Integration

- The fine-tuned model was deployed as part of the citation API.
- When users select the “Cite” option, the backend uses Flan-T5 to return a formatted citation string.
- The generated citation is shown in the frontend, with an option to "Insert Citation" which adds it into the document and reference list.

System Flow Integration (All Models Combined)

When a user highlights a sentence:

1. **BERT (NER)** extracts relevant keywords.
2. Keywords are matched against the keyword CSV file to filter candidate papers.

3. **SBERT** embeds the selected sentence and compares it against paper abstracts using the FAISS index.
4. The most relevant papers are returned to the frontend as interactive cards.
5. When “Cite” is clicked, **Flan-T5** generates the citation in IEEE format.
6. “Insert Citation” adds the reference number and appends it to the document’s reference list.

Figure 22 shows the training loss of Flan-T5 model training and Figure 23 shows the python code of training the Flan T5 model.

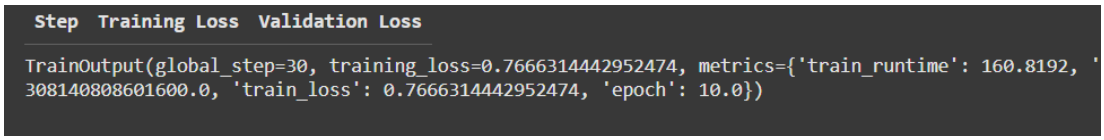


Figure 22: Citation generation Training loss

7. Users may also opt to manually input metadata for citation generation using a separate API.

```python
import torch
from torch.utils.data import Dataset, DataLoader
from transformers import (
    T5Tokenizer, T5ForConditionalGeneration,
    Seq2SeqTrainer, Seq2SeqTrainingArguments
)
from sklearn.model_selection import train_test_split

def format_author_name(full_name):
    """Convert author names to IEEE format: First Initial. Lastname"""
    parts = full_name.split()
    if len(parts) > 1:
        initials = " ".join(f"{p[0]}." for p in parts[:-1])  # Get initials for all except last name
        last_name = parts[-1]
        return f"{initials} {last_name}"
    return full_name  # Return as is if only one part

class CitationDataset(Dataset):
    def __init__(self, data, tokenizer, max_len=512):
        self.data = data
        self.tokenizer = tokenizer
        self.max_len = max_len

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        paper = self.data[index]
        authors = ", ".join(format_author_name(author) for author in paper["authors"])
        title = paper["title"]
        journal = paper["journal"]
        location = paper.get("Conference Location", "Unknown")
        year = paper["year"]
        pages = paper.get("pages", "N/A")
        doi = paper.get("doi", "N/A")
        citation = f"{authors}, \"{title},\" {journal}, {location}, {year}, pp. {pages}. doi: {doi}."
        input_text = (
            f"Generate an IEEE citation for a paper titled '{title}' with: "
            f"Authors: {', '.join(paper['authors'])}, Year: {year}, "
            f"Journal: {journal}, Location: {location}, "
            f"Pages: {pages}, doi: {doi}."
        )
        input_encoding = self.tokenizer(input_text, max_length=self.max_len, padding="max_length", truncation=True, return_tensors="pt")
        output_encoding = self.tokenizer(citation, max_length=self.max_len, padding="max_length", truncation=True, return_tensors="pt")
        labels = output_encoding["input_ids"]
        labels[labels == self.tokenizer.pad_token_id] = -100  # Ignore padding
        return {
            "input_ids": input_encoding["input_ids"].squeeze(),
            "attention_mask": input_encoding["attention_mask"].squeeze(),
            "labels": labels.squeeze(),
        }

import json
from google.colab import files
uploaded = files.upload()
dataset_file = list(uploaded.keys())[0]  # Get uploaded file name
with open(dataset_file, 'r') as f:
    dataset = json.load(f)
print(f"Loaded {len(dataset)} records from {dataset_file}")
tokenizer = AutoTokenizer.from_pretrained("google/flan-t5-base")
model = AutoModelForSeq2SeqLM.from_pretrained("google/flan-t5-base")
from sklearn.model_selection import train_test_split
train_data, eval_data = train_test_split(dataset, test_size=0.1, random_state=42)
train_dataset = CitationDataset(train_data, tokenizer)
eval_dataset = CitationDataset(eval_data, tokenizer)
training_args = Seq2SeqTrainingArguments(
    output_dir="./flan_t5_citation_model",
    num_train_epochs=10,  # More epochs for better format learning
    per_device_train_batch_size=4,
    gradient_accumulation_steps=4,
    learning_rate=3e-5,
    weight_decay=0.01,
    logging_dir="./logs",
    save_steps=500,
    logging_steps=50,
    eval_steps=500,
    save_total_limit=3,
    load_best_model_at_end=True,
    evaluation_strategy="steps",
    warmup_steps=100,
)

trainer = Seq2SeqTrainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    tokenizer=tokenizer,
)

trainer.train()

save_path = "/mnt/gdrive/MyDrive/Model/citation-generator/fine_tuned_Flan-T5_citation_v3"
model.save_pretrained(save_path)
tokenizer.save_pretrained(save_path)
```

Figure 23: Citation Generation Model training

2.1.4.7 Reference Suggestion and Citation Generator Component (API Integration)

This component integrates three backend APIs to form a multi-model pipeline for intelligent reference suggestion and IEEE citation generation in real time. The APIs are hosted and deployed using Azure and accessed via asynchronous frontend calls. Each API is dedicated to handling a core aspect of the system: keyword extraction and semantic similarity, automated citation generation, and manual citation input formatting. Together, these services enhance academic productivity by allowing users to search, generate, and insert citations seamlessly during document writing.

1. **Loading Trained Models:**

The component utilizes three separately trained models, each serving a specific function:

- **Keyword Extraction Model (BERT):** This model extracts research-focused terms from selected text using Named Entity Recognition (NER) with BIO tagging.
- **Semantic Similarity Model (SBERT + FAISS):** This model computes the semantic similarity between user-selected content and a large database of academic abstracts using vector embeddings and cosine similarity.
- **Citation Generator Model (Flan-T5):** This model formats bibliographic metadata into IEEE-style citation strings for both automatic and manual citation workflows.

2. Input Capture and Semantic Search API Flow:

When a user highlights text in the document editor:

- The selected text is sent to the **semantic search API**.
- The **BERT model** extracts relevant keywords from the input.
- These keywords are used to filter a local CSV file of academic titles and abstracts.

- The same input is embedded by the **SBERT model** and compared against stored research paper embeddings using **FAISS** indexing.
- The top-matching research papers are returned based on cosine similarity scores and distance metrics.

3. Reference Suggestion Display:

The API response includes metadata for the most relevant papers:

- Title, author, and paper ID
- Abstract
- Highlighted matching phrase

This information is displayed as individual cards in a sidebar on the frontend. Each card includes actionable options such as “View” and “Cite.”

4. Citation Generation with Flan-T5:

When the user selects “Cite”:

- Metadata from the selected paper is passed to the citation generation API.
- The Flan-T5 model formats the citation according to IEEE style.
- The citation is returned to the frontend and shown in a preview.
- The user may then choose to “Insert Citation,” which appends the reference to the list and inserts a numbered bracket at the selected location in the document.

5. Manual Citation Workflow:

A manual citation modal allows users to input paper metadata directly (title, authors, year, DOI, etc.).

- Submitted data is sent to the manual citation generator API.
- The Flan-T5 model processes the input and returns a fully formatted IEEE citation string.

- This citation is inserted and managed in the same way as automatically generated ones.

6. Real-Time, Multi-Model Integration:

The system runs all inference steps dynamically upon user interaction:

- Keyword extraction using BERT
- Semantic comparison using SBERT and FAISS
- Citation formatting using Flan-T5

APIs are accessed via HTTP requests from the frontend and respond with minimal delay, supporting a smooth academic writing experience.

7. Data Handling and Citation Management:

- Academic paper metadata is collected from **Crossref**, **OpenAlex**, and **arXiv**, parsed into JSON format, and used across all model training and inference stages.
- Abstracts are embedded and stored using **FAISS** indexing for fast retrieval.
- Citation and keyword datasets are stored in structured formats (CSV, Pickle) and re-used across multiple model calls.
- Citations are dynamically inserted and numbered based on order of appearance and stored in the document reference section.

8. Overall Mechanism:

- This integrated system enables intelligent, context-aware reference suggestion and citation generation based on live user input.
- It simplifies academic referencing by automating search, formatting, and numbering.

- Manual input support provides flexibility for citing unpublished or user-provided sources.
- All backend interactions are designed to operate without disrupting the user's writing flow.

Figures 25–27 illustrate code snippets and API response structures for each component in the backend pipeline.

```python
# citation_api.py
import os
from fastapi import APIRouter
from pydantic import BaseModel
import torch
import pandas as pd
import ast
from transformers import AutoModelForSeq2SeqLM, AutoTokenizer

# === Setup === #
router = APIRouter()

# Use CPU (no GPU needed)
device = torch.device("cpu")
hf_token = os.getenv("HF_TOKEN", None)  # Ensure you set HF_TOKEN in your environment


# Load model from local path
citation_model_path = "krithigadb/fine-tuned-Flan-T5-citation"
citation_tokenizer = AutoTokenizer.from_pretrained(citation_model_path, token=hf_token)
citation_model = AutoModelForSeq2SeqLM.from_pretrained(citation_model_path, token=hf_token).to(device)

# Load dataset
df = pd.read_csv("./models/InteractiveSheet_2025-03-12_16_01_53 - Sheet1.csv")


# === Utility Functions === #
def parse_authors(authors):
    if isinstance(authors, str):
        try:
            authors_list = ast.literal_eval(authors)
            if isinstance(authors_list, list):
                return authors_list
        except Exception:
            return [a.strip() for a in authors.split(",")]
    return authors


def format_author_name(full_name):
    parts = full_name.split()
    if len(parts) > 1:
        initials = " ".join(f"{p[0]}." for p in parts[:-1])
        last_name = parts[-1]
        return f"{initials} {last_name}"
    return full_name


def generate_citation(paper_details):
    authors = parse_authors(paper_details.get("authors", []))
    formatted_authors = ", ".join(format_author_name(author) for author in authors)

    title = paper_details.get("title", "Unknown Title")
    year = paper_details.get("year", "Unknown Year")
    journal = paper_details.get("journal", "Unknown Journal")
    location = paper_details.get("Conference Location", "Unknown Location")
    pages = paper_details.get("pages", "N/A")
    doi = paper_details.get("doi", "N/A")

    input_text = (
        f"Generate an IEEE citation for a research paper titled '{title}' with details: "
        f"Authors: {formatted_authors}, Year: {year}, "
        f"Journal: {journal}, Location: {location}, "
        f"Pages: {pages}, DOI: {doi}."
    )

    input_ids = citation_tokenizer.encode(input_text, return_tensors="pt", truncation=True).to(device)

    with torch.no_grad():
        output_ids = citation_model.generate(
            input_ids, max_length=512, num_beams=8, repetition_penalty=2.0, early_stopping=True
        )

    generated_citation = citation_tokenizer.decode(output_ids[0], skip_special_tokens=True).strip()
    return generated_citation


# === API Request Schemas === #
class MultiCitationRequest(BaseModel):
    selected_paper_ids: list[str]


@router.post("/generate_citations/")
async def generate_multiple_citations(request: MultiCitationRequest):
    citations = []
    for paper_id in request.selected_paper_ids:
        paper_row = df[df["paper_id"].astype(str).str.strip() == str(paper_id).strip()]
        if paper_row.empty:
            citations.append({"paper_id": paper_id, "citation": "Paper ID not found."})
            continue

        paper_row = paper_row.iloc[0]
        paper_details = {
            "authors": parse_authors(paper_row["authors"]),
            "title": paper_row["title"],
            "journal": paper_row["journal"],
            "year": paper_row["year"],
            "Conference Location": paper_row.get("Conference Location", "Unknown Location"),
            "pages": paper_row.get("pages", "N/A"),
            "doi": paper_row.get("doi", "N/A"),
        }
        citation = generate_citation(paper_details)
        citations.append({"paper_id": paper_id, "citation": citation})

    return {"citations": citations}
```

Figure 25: Citation API Implementation backend

```python
# manual_citation_api.py
import os
from fastapi import APIRouter
from pydantic import BaseModel
import torch
import pandas as pd
import ast
from transformers import AutoModelForSeq2SeqLM, AutoTokenizer

# === Setup === #
router = APIRouter()

device = torch.device("cpu")  # Use CPU
hf_token = os.getenv("HF_TOKEN", None)  # Ensure you set HF_TOKEN in your environment


# Load model and tokenizer
citation_model_path = "krithigadb/fine-tuned-Flan-T5-citation"
# citation_tokenizer = AutoTokenizer.from_pretrained(citation_model_path)
citation_tokenizer = AutoTokenizer.from_pretrained(citation_model_path, token=hf_token)
citation_model = AutoModelForSeq2SeqLM.from_pretrained(citation_model_path, token=hf_token).to(device)

# citation_model = AutoModelForSeq2SeqLM.from_pretrained(citation_model_path).to(device)


# === Utility Functions === #
def parse_authors(authors):
    if isinstance(authors, str):
        try:
            authors_list = ast.literal_eval(authors)
            if isinstance(authors_list, list):
                return authors_list
        except Exception:
            return [a.strip() for a in authors.split(",")]
    return authors


def format_author_name(full_name):
    parts = full_name.split()
    if len(parts) > 1:
        initials = " ".join(f"{p[0]}." for p in parts[:-1])
        last_name = parts[-1]
        return f"{initials} {last_name}"
    return full_name


def generate_citation(paper_details):
    authors = parse_authors(paper_details.get("authors", []))
    formatted_authors = ", ".join(format_author_name(author) for author in authors)

    title = paper_details.get("title", "Unknown Title")
    year = paper_details.get("year", "Unknown Year")
    journal = paper_details.get("journal", "Unknown Journal")
    location = paper_details.get("Conference Location", "Unknown Location")
    pages = paper_details.get("pages", "N/A")
    doi = paper_details.get("doi", "N/A")

    input_text = (
        f"Generate an IEEE citation for a research paper titled '{title}' with details: "
        f"Authors: {formatted_authors}, Year: {year}, "
        f"Journal: {journal}, Location: {location}, "
        f"Pages: {pages}, DOI: {doi}."
    )

    input_ids = citation_tokenizer.encode(input_text, return_tensors="pt", truncation=True).to(device)

    with torch.no_grad():
        output_ids = citation_model.generate(
            input_ids, max_length=512, num_beams=8, repetition_penalty=2.0, early_stopping=True
        )

    generated_citation = citation_tokenizer.decode(output_ids[0], skip_special_tokens=True).strip()
    return generated_citation


# === API Schema === #
class CitationRequest(BaseModel):
    paper_id: str = None
    authors: list = None
    title: str = None
    journal: str = None
    year: int = None
    location: str = None
    pages: str = None
    doi: str = None


# === API Endpoint === #
@router.post("/generate_manual_citation/")
async def generate_manual_citation(request: CitationRequest):
    paper_details = {
        "authors": request.authors or [],
        "title": request.title or "Unknown Title",
        "journal": request.journal or "Unknown Journal",
        "year": request.year or "Unknown Year",
        "Conference Location": request.location or "Unknown Location",
        "pages": request.pages or "N/A",
        "doi": request.doi or "N/A",
    }

    citation = generate_citation(paper_details)
    return {"citation": citation}
```

Figure 26: Manual Citation Backend

```python
# semantic_search_api.py
import os  # Added import for os
from fastapi import APIRouter
from pydantic import BaseModel
import pandas as pd
import numpy as np
import pickle
import faiss
import torch
import re
from transformers import AutoTokenizer, AutoModelForTokenClassification, pipeline
from sentence_transformers import SentenceTransformer

# === Setup === #
router = APIRouter()
hf_token = os.getenv("HF_TOKEN", None)  # Ensure you set HF_TOKEN in your environment

# Load models and data from local paths
# keyword_model_path = "./models/suggestionKeywords_V2"
keyword_model_path = "krithigasb/suggestionKeywords_V2"
sbert_model_path = "krithigasb/fine_tuned_sbert"
csv_path = "./models/InteractiveSheet_2025-03-12_16_01_53 - Sheet1.csv"
embedding_path = "./models/sbert_embeddings_latest_v1.pkl"

# Load models
tokenizer = AutoTokenizer.from_pretrained(keyword_model_path, token=hf_token)
model = AutoModelForTokenClassification.from_pretrained(keyword_model_path, token=hf_token)
nlp = pipeline("ner", model=model, tokenizer=tokenizer, aggregation_strategy="simple")

sbert_model = SentenceTransformer(sbert_model_path, use_auth_token=hf_token)
df = pd.read_csv(csv_path)

# Load FAISS index
with open(embedding_path, "rb") as f:
    data = pickle.load(f)

paper_ids = data["paper_ids"]
abstracts = data["abstracts"]
titles = data["titles"]
sbert_embeddings = np.array(data["embeddings"])

# Normalize and build FAISS index
faiss.normalize_L2(sbert_embeddings)
dimension = sbert_embeddings.shape[1]
index = faiss.IndexFlatIP(dimension)
index.add(sbert_embeddings)

global_filtered_papers = {}

# === Request Schema === #
class QueryRequest(BaseModel):
    text: str

# === Utilities === #
def get_paper_details(paper_id):
    paper_row = df[df["paper_id"] == paper_id]
    if paper_row.empty:
        return None
    paper_row = paper_row.iloc[0]
    return {
        "paper_id": str(paper_row["paper_id"]),
        "title": str(paper_row["title"]),
        "authors": paper_row["authors"] if pd.notna(paper_row["authors"]) else "Unknown",
        "year": int(paper_row["year"]) if pd.notna(paper_row["year"]) else "Unknown",
        "journal": str(paper_row["journal"]) if pd.notna(paper_row["journal"]) else "Unknown",
        "type": str(paper_row["type"]) if pd.notna(paper_row["type"]) else "Unknown",
        "Conference Location": str(paper_row["Conference Location"]) if pd.notna(paper_row["Conference Location"]) else "N/A",
        "pages": str(paper_row["pages"]) if pd.notna(paper_row["pages"]) else "N/A",
        "volume": str(paper_row["volume"]) if pd.notna(paper_row["volume"]) else "N/A",
        "issue": str(paper_row["issue"]) if pd.notna(paper_row["issue"]) else "N/A",
        "doi": str(paper_row["doi"]) if pd.notna(paper_row["doi"]) else "N/A",
        "Abstract": str(paper_row["Abstract"]) if pd.notna(paper_row["Abstract"]) else "No Abstract Available",
        "keywords": str(paper_row["keywords"]) if pd.notna(paper_row["keywords"]) else "No Keywords Available"
    }

def extract_keywords(text):
    ner_results = nlp(text)
    keywords = {result["word"].lower().strip() for result in ner_results if result["entity_group"] in ['ORG', 'LABEL_1', 'LABEL_2', 'MISC']}
    return list(keywords)

def filter_papers_by_keywords(input_keywords, df, threshold=0.6):
    if not input_keywords or "fine-tuned" not in df.columns:
        return pd.DataFrame()
    input_keywords_normalized = {kw.lower().strip() for kw in input_keywords}
    def compute_match(row):
        if pd.isna(row["fine-tuned"]) or not isinstance(row["fine-tuned"], str):
            return 0, set()
        paper_keywords = {kw.lower().strip() for kw in row["fine-tuned"].split(", ")}
        matched_keywords = input_keywords_normalized.intersection(paper_keywords)
        if matched_keywords:
            return 1, matched_keywords
        match_ratio = len(matched_keywords) / len(input_keywords_normalized)
        return match_ratio, matched_keywords
    df["match_ratio"], df["matched_keywords"] = zip(*df.apply(compute_match, axis=1))
    df["matched_keywords"] = df["matched_keywords"].apply(lambda x: ", ".join(x) if x else "No Match")
    return df[(df["match_ratio"] >= threshold) | (df["match_ratio"] == 1)].sort_values(by="match_ratio", ascending=False)

def find_all_matching_sentences(query_embedding, abstract, top_n=3):
    sentences = re.split(r'(?<=[.!?])\s+', abstract)
    if not sentences:
        return []
    sentence_embeddings = sbert_model.encode(sentences, batch_size=32, show_progress_bar=False)
    sentence_embeddings = np.array(sentence_embeddings)
    faiss.normalize_L2(sentence_embeddings)
    scores = sentence_embeddings @ query_embedding[0]
    sorted_indices = np.argsort(-scores)
    return [{"sentence": sentences[i], "similarity": round(scores[i] * 100, 2)} for i in sorted_indices[:top_n]]

def search_similar_papers_all_sentences(query, top_k=10, top_n=3):
    query_embedding = sbert_model.encode([query])
    faiss.normalize_L2(query_embedding)
    distances, indices = index.search(query_embedding, top_k)
    results = []
    for rank, idx in enumerate(indices[0]):
        sim_score = distances[0][rank]
        paper_id = paper_ids[idx]
        abstract = abstracts[idx]
        matched_sentences = find_all_matching_sentences(query_embedding, abstract, top_n)
        details = get_paper_details(paper_id) or {
            "paper_id": paper_id,
            "title": titles[idx],
            "authors": "Unknown",
            "year": "Unknown",
            "journal": "Unknown",
            "type": "Unknown",
            "Conference Location": "N/A",
            "pages": "N/A",
            "volume": "N/A",
            "issue": "N/A",
            "doi": "N/A",
            "Abstract": abstract,
            "keywords": "No Keywords Available",
        }
        details["Abstract"] = abstract
        details["matched_sentences"] = matched_sentences
        details["similarity_percent"] = float(round(sim_score * 100, 2))
        results.append(details)
    return sorted(results, key=lambda x: x["similarity_percent"], reverse=True)

def fully_convert(obj):
    if isinstance(obj, dict):
        return {k: fully_convert(v) for k, v in obj.items()}
    if isinstance(obj, list):
        return [fully_convert(item) for item in obj]
    if isinstance(obj, tuple):
        return [fully_convert(item) for item in obj]
    if isinstance(obj, set):
        return [fully_convert(item) for item in obj]
    if isinstance(obj, np.ndarray):
        return obj.tolist()
    if isinstance(obj, np.floating):
        return float(obj)
    if isinstance(obj, np.integer):
        return int(obj)
    return obj

# === Endpoint === #
@router.post("/search/")
def process_query(request: QueryRequest):
    global global_filtered_papers
    keywords = extract_keywords(request.text)
    filtered_papers = filter_papers_by_keywords(keywords, df)
    if filtered_papers.empty:
        return {"message": "No matching papers found"}
    results = search_similar_papers_all_sentences(request.text, top_k=50, top_n=3)
    final_results = [fully_convert(paper) for paper in results]
    global_filtered_papers = {paper["paper_id"]: paper for paper in results}
    return {
        "keywords": keywords,
        "results": final_results
    }
```

Figure 27: Semantic Search And Keyword Extraction Backend

2.1.4.8 Frontend Development of reference Suggester and Citation generator

The backend logic for the AI-powered reference suggestion and citation generation system is implemented using three Google Colab notebooks, each dedicated to a specific task in the citation pipeline. These notebooks act as live API endpoints for processing selected text, retrieving matching references, and generating citations in IEEE format. The implementation ensures real-time interaction between the frontend editor and backend AI models for seamless academic writing support.

Interface Layout and Functional Blocks

The user interface is organized into modular components, each addressing a specific function of the system's workflow. The following key components were implemented:

1. Citation Sidebar (CiteSidebar.js)

   - This component handles the display of AI-suggested research papers based on semantic similarity with the selected content in the editor.
   - When the user highlights a portion of text, this triggers an API request to the backend Python notebooks, which return a list of relevant academic papers.
   - The results are rendered as interactive cards, each showing paper metadata (title, authors) and two options: "View Abstract" and "Cite."
   - On clicking "Cite," the IEEE-formatted citation—generated by a model hosted in the backend Python environment—is inserted inline near the selected text, while the reference is added to the document's reference list.

2. Manual Citation Modal (ManualCitationModal.js)

   - This model allows users to input reference details manually, including fields such as title, author, publication, DOI, and year.
   - On submission, the data is sent to the backend Python service, where a citation string is generated using a fine-tuned citation model.
   - Once returned, the formatted citation is displayed and appended to the reference list within the editor.

3. Editor Integration (Editor.js)

    - The editor component integrates citation interactions directly into the writing interface.
    - When a user selects text and initiates a reference search, the selected content is passed to the backend for keyword extraction and semantic filtering.
    - Once a citation is inserted, the editor automatically manages the reference numbering and citation brackets (e.g., [1]), ensuring they are in sync with the reference list.
    - The logic implemented in this file ensures that citations appear correctly next to the relevant text and that any changes in the reference list are reflected throughout the document.

JavaScript Logic and API Communication

The frontend uses standard JavaScript fetch operations and React hooks (useState, useEffect) to manage UI state and communicate asynchronously with the backend Python notebooks. These API calls perform the following operations:

- Trigger keyword extraction and semantic similarity scoring.
- Fetch suggested references with highlighted matched sentences.
- Generate IEEE citations automatically or based on user-defined input.
- Insert citation references and manage the citation list dynamically.

All API responses are used to update the interface in real-time, allowing for a responsive and consistent user workflow. The Figure 28 and 29 illustrate code snippets of frontend implementation.

Styling and Layout

Custom CSS and layout classes were applied to enhance visual clarity and usability. Key interface elements include:

- A collapsible sidebar showing suggestion cards.
- A modal form for manual citation inputs.
- A reference list block appended to the bottom of the document.
- Clean font styles and spacing for citation bracket markers and reference entries.

```
import React from "react";

const ReferenceModal = ({
    showReferenceModal,
    setShowReferenceModal,
    newReference,
    setNewReference,
    references,
    setReferences,
    padId,
    onCitationData,
    onSaveReference,
}) => {
    if (!showReferenceModal) return null;
    const baseApiUrl_Manual_Citation = `${process.env.REACT_APP_BACKEND_API_URL_MANUAL_CITATION}`;
    const handleSaveReference = async () => {
        const lastKey = Array.isArray(references)
            ? references.reduce((max, ref) => {
                const numericKey = parseInt(ref.key, 10) || 0;
                return numericKey > max ? numericKey : max;
            }, 0)
            : 0;
        const nextKey = lastKey + 1;
        const requestBody = {
            authors: newReference.author.split(",").map(a => a.trim()),
            title: newReference.title,
            journal: newReference.journal,
            year: parseInt(newReference.year, 10),
            location: newReference.location,
            pages: newReference.pages,
            doi: newReference.doi,
        };

        try {
            const citationResponse = await fetch(
                `${process.env.REACT_APP_BACKEND_API_URL_MANUAL_CITATION}/manual/generate_manual_citation/`,
                {
                    method: "POST",
                    headers: { "Content-Type": "application/json" },
                    body: JSON.stringify(requestBody),
                }
            );

            console.log(" Awaiting response from generate_manual_citation API...");
            if (!citationResponse.ok) {
                const errorText = await citationResponse.text();
                throw new Error(`Failed to generate citation: ${errorText}`);
            }
            const citationDataJSON = await citationResponse.json();
            console.log(" Citation generated successfully!", citationDataJSON);
            const generatedCitation = citationDataJSON.citation || "Citation not available.";

            const saveResponse = await fetch(
                `${process.env.REACT_APP_BACKEND_API_URL}/api/pads/${padId}/save-citation`,
                {
                    method: "POST",
                    headers: { "Content-Type": "application/json" },
                    body: JSON.stringify({
                        padId,
                        key: String(nextKey),
                        citation: generatedCitation,
                        ...requestBody,
                    }),
                }
            );

            console.log(" Awaiting response from save-citation API...");
            if (!saveResponse.ok) {
                const errorText = await saveResponse.text();
                throw new Error(`Failed to save citation: ${errorText}`);
            }
            const saveData = await saveResponse.json();
            console.log(" Reference saved successfully!", saveData);

            const finalReference = { ...newReference, key: String(nextKey), citation: generatedCitation };
            console.log("Final Reference:", finalReference);

            if (typeof onCitationData === "function") {
                onCitationData(finalReference);
            }
            setShowReferenceModal(false);
            return finalReference;
        } catch (error) {
            console.error(" Error in handleSaveReference:", error);
        }
    };
```

Figure 28:JS FrontEnd function code of ManualCitationModel.js

```
    return (
        <div
            style={{
                position: "fixed",
                top: "50%",
                left: "50%",
                transform: "translate(-50%, -50%)",
                backgroundColor: "#fff",
                padding: "20px",
                borderRadius: "8px",
                boxShadow: "0 2px 10px rgba(0,0,0,0.2)",
                zIndex: 1100,
                width: "400px",
            }}
        >
            <h3>Add Reference</h3>
            <div style={{ display: "flex", flexDirection: "column", gap: "10px" }}>
                <input
                    className="input-small reference-key"
                    type="text"
                    value={newReference.key}
                    placeholder="Reference Key"
                    onChange={(e) =>
                        setNewReference({ ...newReference, key: e.target.value })
                    }
                />
                <input
                    className="input-small reference-author"
                    type="text"
                    value={newReference.author}
                    placeholder="Author(s)"
                    onChange={(e) =>
                        setNewReference({ ...newReference, author: e.target.value })
                    }
                />
                <input
                    className="input-small reference-title"
                    type="text"
                    value={newReference.title}
                    placeholder="Title"
                    onChange={(e) =>
                        setNewReference({ ...newReference, title: e.target.value })
                    }
                />
                <input
                    className="input-small reference-journal"
                    type="text"
                    value={newReference.journal}
                    placeholder="Conference"
                    onChange={(e) =>
                        setNewReference({ ...newReference, journal: e.target.value })
                    }
                />
                <input
                    className="input-small reference-year"
                    type="text"
                    value={newReference.year}
                    placeholder="Year"
                    onChange={(e) =>
                        setNewReference({ ...newReference, year: e.target.value })
                    }
                />
                <input
                    className="input-small reference-volume"
                    type="text"
                    value={newReference.location}
                    placeholder="Conference Location"
                    onChange={(e) =>
                        setNewReference({ ...newReference, location: e.target.value })
                    }
                />
                <input
                    className="input-small reference-number"
                    type="text"
                    value={newReference.doi}
                    placeholder="doi"
                    onChange={(e) =>
                        setNewReference({ ...newReference, doi: e.target.value })
                    }
                />
                <input
                    className="input-small reference-pages"
                    type="text"
                    value={newReference.pages}
                    placeholder="Pages"
                    onChange={(e) =>
                        setNewReference({ ...newReference, pages: e.target.value })
                    }
                />
            </div>
            <div
                style={{
                    marginTop: "15px",
                    display: "flex",
                    justifyContent: "flex-end",
                    gap: "10px",
                }}
            >
                <button
                    className="custom-button"
                    onClick={async () => {
                        await onSaveReference(newReference);
                        setShowReferenceModal(false);
                    }}
                    style={{
                        backgroundColor: "#aaa",
                        color: "#fff",
                        padding: "8px 12px",
                        border: "none",
                        borderRadius: "5px",
                        cursor: "pointer",
                    }}
                >
                    Cancel
                </button>
                <button
                    className="custom-button"
                    onClick={handleSaveReference}
                    style={{
                        backgroundColor: "#56008a",
                        color: "#fff",
                        padding: "8px 12px",
                        border: "none",
                        borderRadius: "5px",
                        cursor: "pointer",
                    }}
                >
                    Save Reference
                </button>
            </div>
        </div>
    );
};

export default ReferenceModal;
```

Figure 29: frontEnd code HTML section of Manualcitationmode.js

2.1.5 Testing

The testing process undertaken for the AI-powered Reference Paper Suggestion and Citation Generator component involved multiple levels of software testing to ensure accuracy, usability, integration, and overall robustness. These included unit testing, integration testing, system testing, and acceptance testing, all conducted in alignment with the broader objectives of the collaborative document editing tool.

Unit Testing:

- Unit testing served as the foundational step, targeting individual functions and logic blocks such as keyword extraction, semantic search, citation formatting, and API endpoints.
- Specific tests were implemented to check:
  - Whether selected text from the editor was accurately passed to the backend.
  - Proper keyword extraction using the fine-tuned BERT-NER model.
  - Correct paper retrieval and ranking based on cosine similarity from the FAISS index [10].
  - IEEE citation formatting with the Flan-T5 model output.
- These unit tests ensured that model inferences, backend processes, and data formatting components worked independently as expected.

Integration Testing:

- Integration testing focused on validating interactions between the backend services and the frontend interface, especially involving:
  - Communication between the editor component (Editor.js) and the citation sidebar (CiteSidebar.js).
  - Functional response from manual citation modal (ManualCitationModal.js) to backend endpoints.
  - Cross-functionality of reference generation and insertion mechanisms.

- The integration of citation generation with selected text, citation number insertion logic, and real-time reference list update were tested to ensure smooth and consistent workflow within the editing interface.

System Testing:

- In system testing, the component was tested as part of the full document editing tool to validate its behavior within a realistic operational environment.
- Tests covered:
    - The ability to retrieve and suggest relevant papers across various research topics.
    - Consistency in adding citations across different document sections and maintaining a dynamic reference list.
    - Edge case handling such as unsupported metadata, poorly formatted abstracts, or missing DOIs.
- Emphasis was placed on ensuring that citation generation integrated smoothly with other modules like collaborative editing and document export features.

Acceptance Testing:

- Alpha testing was carried out internally using mock academic documents, where researchers and students evaluated the citation recommendation process, citation accuracy, and user interface responsiveness.
- Based on feedback:
    - Additional highlighting was added to show the matched sentence from the paper abstract.
    - Minor UI refinements were applied to the citation preview and reference export functionality.
- A limited beta testing phase was also conducted with early users from academic settings who interacted with the tool in real-time research writing scenarios. This provided realistic insights into tool performance, model reliability, and user satisfaction.

- The results of acceptance testing confirmed that the tool met user expectations and streamlined the referencing workflow in collaborative research authoring.

This comprehensive multi-phase testing approach helped ensure the reliability, usability, and real-time integration of the citation tool, making it a valuable enhancement to the collaborative research environment.

Unit Testing:

For the **Unit Testing**, the python's *unittest* library has been used to automate the important functionalities. Figure 30-32 illustrates the unit testing script for functionality.

```python
import unittest
from citation_api import generate_citation, format_author_name, parse_authors

class TestCitationAPI(unittest.TestCase):

    def test_format_author_name(self):
        result = format_author_name("Alice Smith")
        self.assertEqual(result, "A. Smith")

    def test_parse_authors(self):
        result = parse_authors("John Doe, Jane Roe")
        self.assertEqual(result, ["John Doe", "Jane Roe"])

    def test_generate_citation_complete(self):
        paper = {
            "authors": ["Alice Smith", "Bob Jones"],
            "title": "AI and Future Tech",
            "year": 2023,
            "journal": "AI Journal",
            "Conference Location": "USA",
            "pages": "1-10",
            "doi": "10.1234/abc123"
        }
        citation = generate_citation(paper)
        self.assertIn("A. Smith", citation)
        self.assertIn("B. Jones", citation)
        self.assertIn("AI and Future Tech", citation)
```

Figure 30: Unit Test Script

```python
import unittest
from manual_citation_api import generate_manual_citation

class TestManualCitationAPI(unittest.TestCase):

    def test_generate_manual_citation_complete(self):
        input_data = {
            "title": "AI Paper",
            "authors": "Alice Smith, Bob Brown",
            "year": "2023",
            "journal": "TechConf",
            "location": "USA",
            "pages": "1-10",
            "doi": "10.4567/xyz"
        }
        citation = generate_manual_citation(input_data)
        self.assertIn("AI Paper", citation)
        self.assertIn("A. Smith", citation)
        self.assertIn("B. Brown", citation)

    def test_generate_manual_citation_missing_fields(self):
        input_data = {
            "title": "",
            "authors": "",
            "year": "",
            "journal": "",
            "location": "",
            "pages": "",
            "doi": ""
        }
        citation = generate_manual_citation(input_data)
        self.assertTrue(isinstance(citation, str))
        self.assertNotEqual(citation, "")
```

Figure 31: Unit Testing Manual Citation

```python
import unittest
from semantic_search_api import extract_keywords, filter_papers_by_keywords, find_all_matching_sentences

class TestSemanticSearchAPI(unittest.TestCase):

    def test_extract_keywords(self):
        text = "Deep learning models have transformed computer vision tasks."
        keywords = extract_keywords(text)
        self.assertIsInstance(keywords, list)
        self.assertGreater(len(keywords), 0)

    def test_filter_papers_by_keywords(self):
        keywords = ["machine", "learning"]
        papers = [
            {"title": "Machine Learning Basics", "abstract": "Intro to machine learning"},
            {"title": "Quantum Computing", "abstract": "Nothing related"}
        ]
        filtered = filter_papers_by_keywords(keywords, papers)
        self.assertIsInstance(filtered, list)
        self.assertTrue(any("Machine Learning" in p["title"] for p in filtered))

    def test_find_all_matching_sentences(self):
        text = "AI in healthcare"
        abstract = "AI is used in many fields. AI in healthcare improves diagnosis. Robotics in surgery is also popular."
        matches = find_all_matching_sentences(text, abstract)
        self.assertIsInstance(matches, list)
        self.assertGreaterEqual(len(matches), 1)
```

Figure 32: Unit testing Semantic search

Manual Testing:

Manual testing played a critical role in the quality assurance process of the AI-powered Reference Paper Suggestion and Citation Generator. Through deliberate and user-centric validation, this approach allowed testers to assess the overall user experience, functional correctness, and consistency of integration within the collaborative document editing environment.

Purpose of Manual Testing:

- The goal of manual testing was to validate the component's core functionalities such as citation suggestion, metadata handling, and formatted citation generation by simulating real-world interactions.
- Manual evaluation helped ensure that the interface was intuitive, citation logic was accurate, and backend interactions (API responses, keyword extraction, and similarity matching) aligned with user expectations.

- Testers leveraged their domain knowledge and research writing familiarity to simulate scenarios researchers would encounter while drafting academic papers.

Test Case Development:

- A series of detailed test cases were developed based on the defined user stories and editor interactions.
- These test cases covered:
  - Selection of text from the editor and triggering of citation recommendations.
  - Input of metadata manually through the **ManualCitationModal**.
  - Verification of reference insertion and numbering logic.
  - Validation of citation formats returned by the fine-tuned Flan-T5 model.
- Test inputs were created using diverse academic topics, varying metadata completeness, and edge-case phrases.

Execution:

- Each test case was executed manually by navigating through the user interface and triggering the backend API calls via UI elements like "Cite", "Insert", and "Export".
- Testers verified the returned papers for relevance, citation accuracy, reference list updates, and document editor behavior.
- Any deviations from expected behavior, formatting inconsistencies, or usability concerns were documented for revision.

Types of Manual Testing:

- **Functional Testing**: Verified that the selected text triggered keyword extraction and similarity analysis correctly, and that citations were generated and displayed in the expected format.

- **Non-Functional Testing**: Evaluated system responsiveness, especially in handling large text inputs and displaying citation cards dynamically without UI lag.
- **User Acceptance Testing (UAT)**: Performed by test users including students and researchers to ensure that the feature fits seamlessly into academic writing workflows and delivers a user-friendly experience.
- **Exploratory Testing**: Unscripted interactions were performed to uncover unexpected issues in keyword suggestions, sentence matching, and UI element responsiveness.
- **Compatibility Testing**: The system was tested across different browsers (Chrome, Firefox, Edge) and screen sizes to ensure a consistent experience across platforms.

Below are the test cases which were done for the manual testing. Tables 4, 5, 6, 7, 8 and 9display the manual test cases.

Table 4: Test Case for Keyword Extraction from Selected Text

Test Case ID	TC_01
Test Case Objective	Test keyword extraction using BERT NER model
Pre-Requirements	User logged in and editor loaded
Test Steps	1. Select a block of text from the document 2. Trigger citation suggestion from sidebar
Test Data	A paragraph describing a technical concept
Expected Output	Extracted keywords displayed and sent to backend
Actual Output	Extracted keywords displayed and logged
Status	Pass

Table 5: Test Case for Semantic Similarity Analysis

Test Case ID	TC_02
Test Case Objective	Test SBERT semantic filtering of relevant papers
Pre-Requirements	Keywords extracted successfully

Test Steps	1. Trigger semantic analysis after keyword filter 2. Wait for paper suggestions
Test Data	Sample abstract sentence from the document
Expected Output	List of research papers with high semantic similarity shown
Actual Output	Papers displayed with matching scores and highlights
Status	Pass

Table 6: Test Case for Viewing Paper Abstract and Match

Test Case ID	TC_03
Test Case Objective	View abstract of selected paper and matched sentences
Pre-Requirements	Relevant papers shown
Test Steps	1. Click “View” button on a suggested paper
Test Data	Filtered paper suggestion
Expected Output	Full abstract and matched sentence section shown
Actual Output	Abstract and highlight displayed
Status	Pass

Table 7: Test Case for Generating Citation (Auto)

Test Case ID	TC_04
Test Case Objective	Generate IEEE citation using Flan-T5
Pre-Requirements	Paper selected
Test Steps	1. Click “Cite” button on selected paper

	2. Confirm citation preview
Test Data	Metadata of the selected paper
Expected Output	Citation string shown in IEEE format
Actual Output	Citation generated and shown
Status	Pass

Table 8: Test Case for Adding Manual Citation

Test Case ID	TC_05
Test Case Objective	Add citation manually using the citation modal
Pre-Requirements	Modal opened via sidebar
Test Steps	1. Click “Add Manual Citation” 2. Enter paper metadata 3. Click "Generate"
Test Data	Title, Author, DOI
Expected Output	Citation string shown in IEEE format
Actual Output	Citation generated correctly
Status	Pass

Table 9: Test Case for Reference Insertion and Numbering

Test Case ID	TC_06
Test Case Objective	Test correct insertion of citation number and reference list
Pre-Requirements	Citation generated
Test Steps	1. Click “Insert Citation” 2. Check inline text and reference list
Test Data	Citation string

Expected Output	Citation number added near text and formatted reference added
Actual Output	Citation number and reference correctly inserted
Status	Pass

2.1.6 Deployment & Maintenance

Deployment:

Deploying reference suggestion and citation generation models involved provisioning a Virtual Machine (VM) on Microsoft Azure and configuring an environment for secure, scalable, and real-time API delivery. Below are the structured deployment phases.

1. Environment Setup on Azure VM
   - ❖ A virtual machine was provisioned on Microsoft Azure with sufficient CPU and memory for model loading.
   - ❖ Ubuntu OS was used as the base environment.
   - ❖ Python 3.12 virtual environment (venv) was created and activated for isolated package handling.
2. Dependency Installation
   - ❖ Required dependencies for all three models (keyword extraction, semantic search, citation generator) were listed in a requirements.txt file.
   - ❖ Using pip install -r requirements.txt, all dependencies including FastAPI, Uvicorn, FAISS [10], Transformers, SentenceTransformers, and more were installed.
3. Model Integration with Hugging Face
   - ❖ Instead of hosting large models locally, they were pushed and accessed from private repositories on Hugging Face.
   - ❖ Tokens were managed securely via .env to authenticate and pull models dynamically during FastAPI startup.
4. API Setup Using FastAPI + Uvicorn
   - ❖ Each model (keyword extractor, semantic search, citation formatter) was wrapped into a FastAPI route inside a unified backend structure.
   - ❖ The application was served using uvicorn on port 8000 for development and testing.

5. Service Daemon Configuration
    - ❖ A systemd service (reference.service) was created to ensure the API starts on boot and restarts automatically in case of failures.
    - ❖ The working directory and virtual environment paths were properly linked within the systemd config.
6. Nginx Reverse Proxy
    - ❖ Nginx was installed and configured to route incoming traffic on port 80 to the internal FastAPI port (8000).
    - ❖ Configurations included timeouts and header forwarding to maintain connectivity with the backend server
7. Access and Domain Setup
    - ❖ The server was made accessible via public IP provided by Azure.
    - ❖ Test requests from browsers and API clients confirmed functional endpoints (/semantic/search, /citation/generate, etc.)
8. Secure SSH Access and Remote Management
    - ❖ SSH key-based authentication was established to enhance security and simplify future maintenance.
    - ❖ Access to the VM is now both secure and convenient for updates and monitoring.
9. Integration with Frontend Modules
    - ❖ The deployed APIs were connected to the CiteSidebar and ManualCitationModal frontend components.
    - ❖ These allow users to interact with the backend in real time through an intuitive user interface.
10. Real-time Accessibility
    - ❖ All endpoints became accessible publicly, enabling dynamic reference suggestions and citation generation directly from the document editing platform.

Maintenance:

Once deployed, the component entered a structured maintenance phase to ensure ongoing performance, accuracy, and system health.

1. Bug Fixing and Issue Resolution
   - ❖ System logs (journalctl and systemctl status) are monitored regularly for performance bottlenecks or service crashes.
   - ❖ Common issues like model loading errors (e.g., authentication with Hugging Face) are mitigated using retry logic and token refresh mechanisms.
2. Model Updates and Enhancements
   - ❖ Updates to model weights and code are pushed periodically to Hugging Face and synced during container or service restarts.
   - ❖ FastAPI code is updated via Git or directly from the VM terminal, followed by service restart.
3. Performance Monitoring and Optimization
   - ❖ Inference latency, server load, and model response times are tracked and optimized for better real-time performance.
4. Security and Access Control
   - ❖ SSH access is restricted to authorized users using public-private key pairs.
   - ❖ .ssh/authorized_keys and directory permissions were configured to enhance deployment security.
   - ❖ Service is set to restart automatically if the VM reboots or if the app fails unexpectedly.
5. Documentation and Testing
   - ❖ Updated documentation is maintained for deployment steps, testing results, and API endpoints.
   - ❖ Unit and integration testing are run regularly to validate system integrity.
6. Scalability Planning
   - ❖ The system is designed to support migration to container-based infrastructure if traffic increases.

- ❖ This ensures the component can be integrated into larger academic platforms in the future.

7. Long-term Support and Continuity
   - ❖ Maintenance schedules and automated backups are planned to ensure uptime and data recovery when needed.
   - ❖ The modular architecture supports easy handover or team-based collaboration.
8. Continuous Integration and Deployment (CI/CD)
   - ❖ A custom GitHub-linked deployment pipeline (reference.service) is in place using a systemd unit and .yml configuration.
   - ❖ When future updates are committed and pushed to the GitHub repository, the component will auto-deploy through the configured pipeline, ensuring minimal manual intervention and consistent rollout.

This structured deployment and maintenance approach ensures that the Reference Suggestion and Citation Generator component remains efficient, accessible, and robust and are ready for real-world academic usage on scale.

Figure 33-38 illustrate functionalities and codes on VM and deployment function.

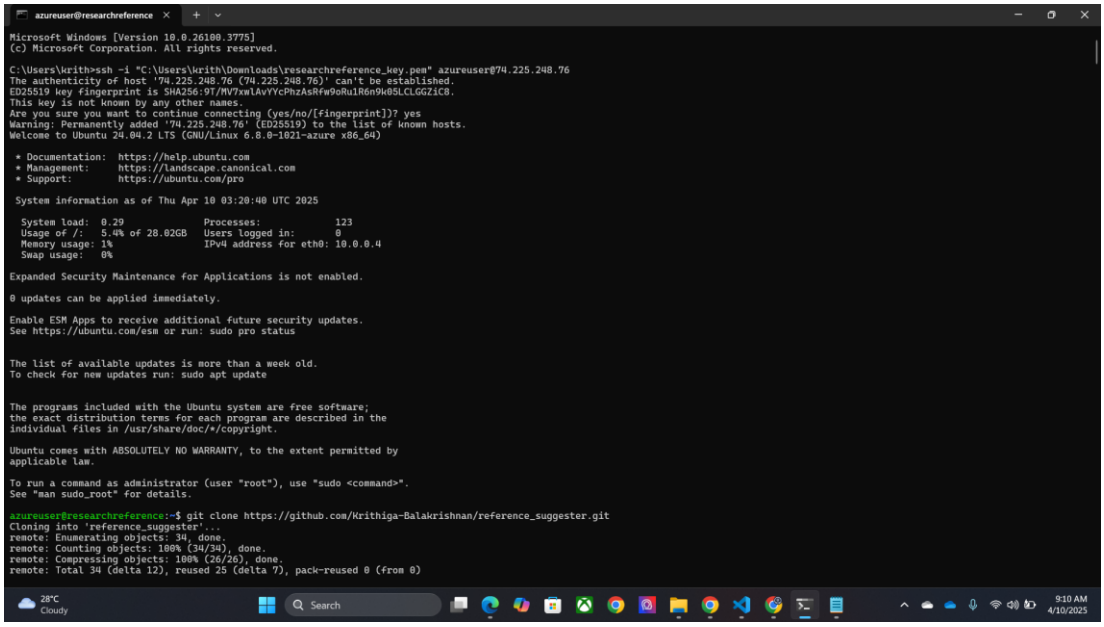


Figure 33: SetUp VM

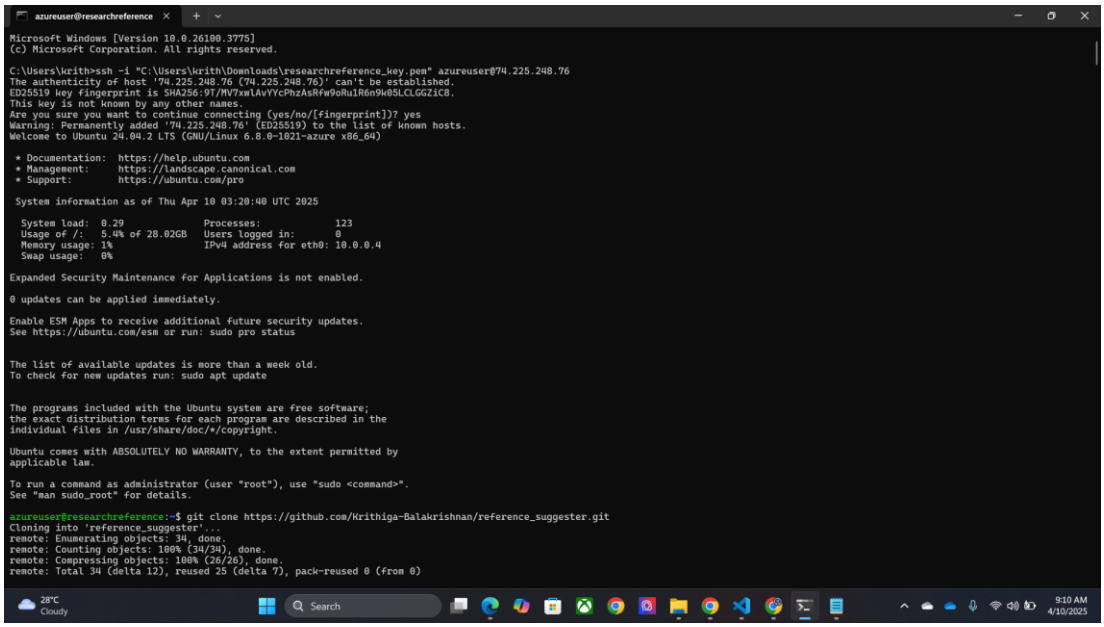


Figure 34: Installations in VM

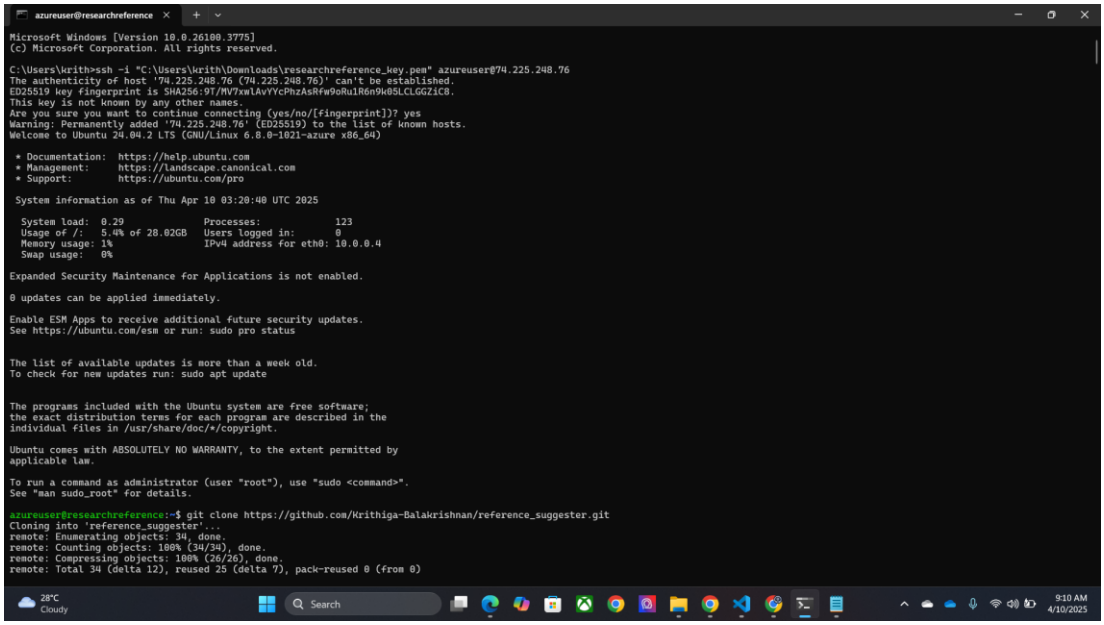


Figure 35: Cloning the Repo folder

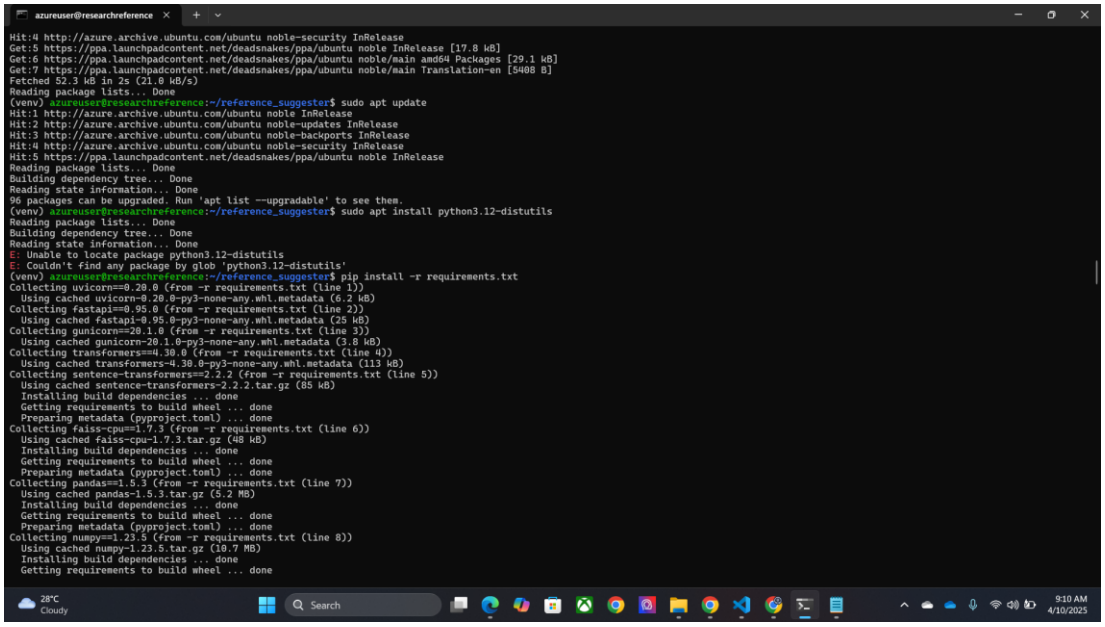


Figure 36: Requirement.txt installation

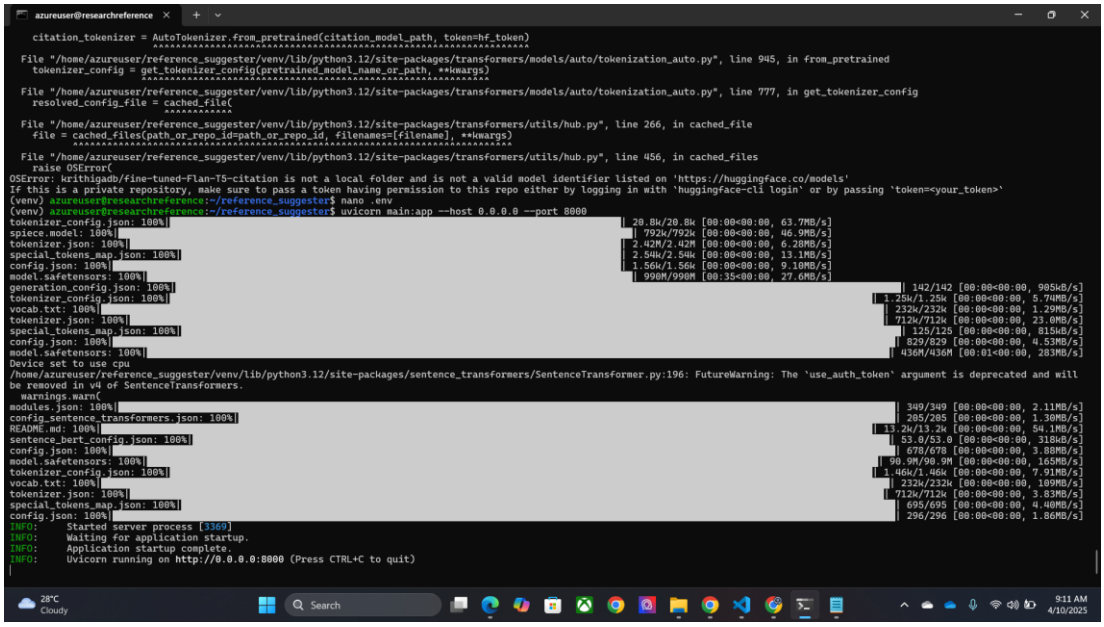


Figure 37:Uvicorn Stetup in VM

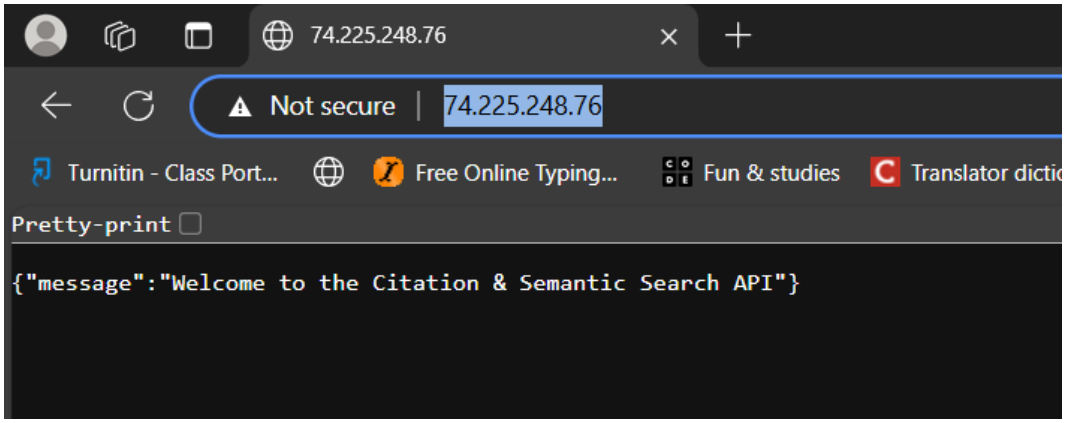


Figure 38: Backend Response after Ngnx configuration

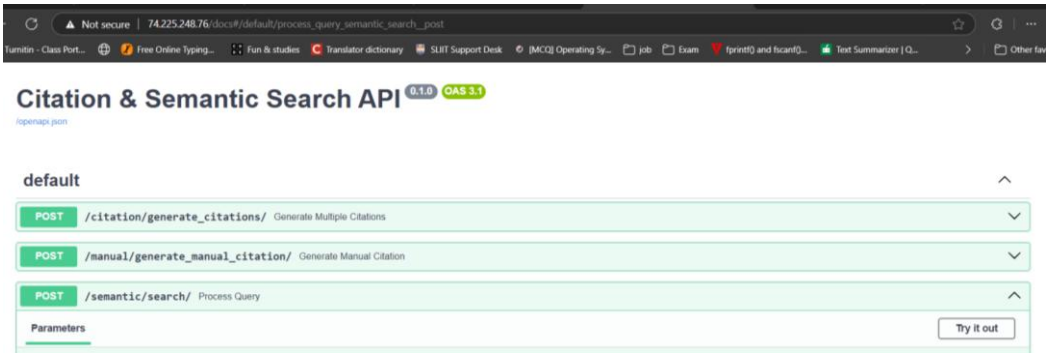


Figure 40: API after configuration

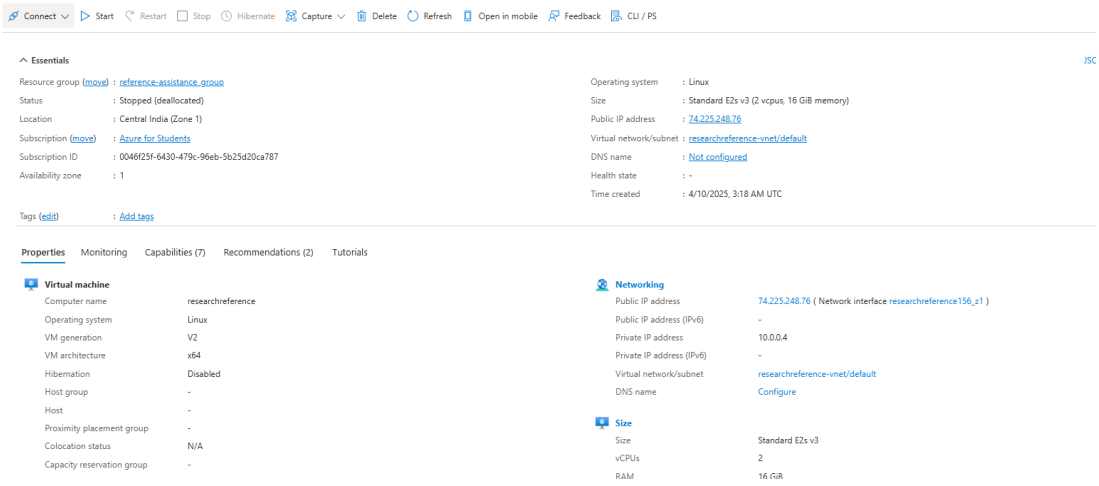


Figure 39: VM data in Azure

```yaml
name: Deploy FastAPI App

on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Setup SSH
        run: |
          mkdir -p ~/.ssh
          echo "${{ secrets.SSH_PRIVATE_KEY }}" > ~/.ssh/id_ed25519
          chmod 600 ~/.ssh/id_ed25519
          ssh-keyscan -H ${{ secrets.SERVER_IP }} >> ~/.ssh/known_hosts

      - name: Deploy to Server
        run: |
          ssh azureuser@${{ secrets.SERVER_IP }} << 'EOF'
            cd ~/reference_suggester
            git pull origin main
            source venv/bin/activate
            pip install -r requirements.txt
            sudo systemctl restart reference
          EOF
```

Figure 41:referennce_deploy.yml file

Deployment of WriteWizard MERN application

The collaborative document editing platform WriteWizard, which includes a React frontend and an Express.js backend, was deployed together on a Microsoft Azure Virtual Machine using Docker and Nginx. This deployment supports user interactions and real-time collaborative editing, integrating with the microservices hosted externally.

1. Azure VM Setup

A virtual machine was provisioned via the Azure portal using Ubuntu Server 24.04 LTS, configured as a Standard E2as v4 instance with 2 vCPUs and 16 GiB memory. Inbound port rules were added to allow traffic through ports 80 and 4000, which were used for serving the frontend and backend respectively.

2. Dockerization and Deployment

The frontend and backend were containerized using separate Dockerfiles, and a unified Docker Compose configuration (docker-compose.yml) was used to orchestrate both containers.

Figure 41 shows the structure of the docker-compose.yml file used to manage service containers, environment variables, and port bindings

docker-compose.yml

```yaml
version: "3.9"

services:
  backend:
    image: sanjayan201/writewizard-backend:latest
    container_name: writewizard-backend
    build:
      context: ./BackEnd
    env_file:
      - ./BackEnd/.env
    ports:
      - "4001:4000"
    restart: always

  frontend:
    image: sanjayan201/writewizard-frontend:latest
    container_name: writewizard-frontend
    build:
      context: ./FrontEnd
    ports:
      - "8080:80"
    restart: always
```

Figure 42: Docker Compose File for Frontend and Backend Service Orchestration

Docker images were built locally and pushed to Docker Hub. On the VM, the images were pulled and launched using the docker-compose up -d command, allowing both services to run simultaneously in detached mode.

3. Reverse Proxy with Nginx

To handle routing and provide seamless public access, Nginx was installed and configured as a reverse proxy. HTTP traffic on port 80 was directed to the front end running on port 8080, while backend API traffic on port 4000 was routed internally to port 4001.

Figure 42 shows the frontend application successfully running on the browser through Nginx routing, and Figure 43 shows the backend API responding correctly after reverse proxy configuration.

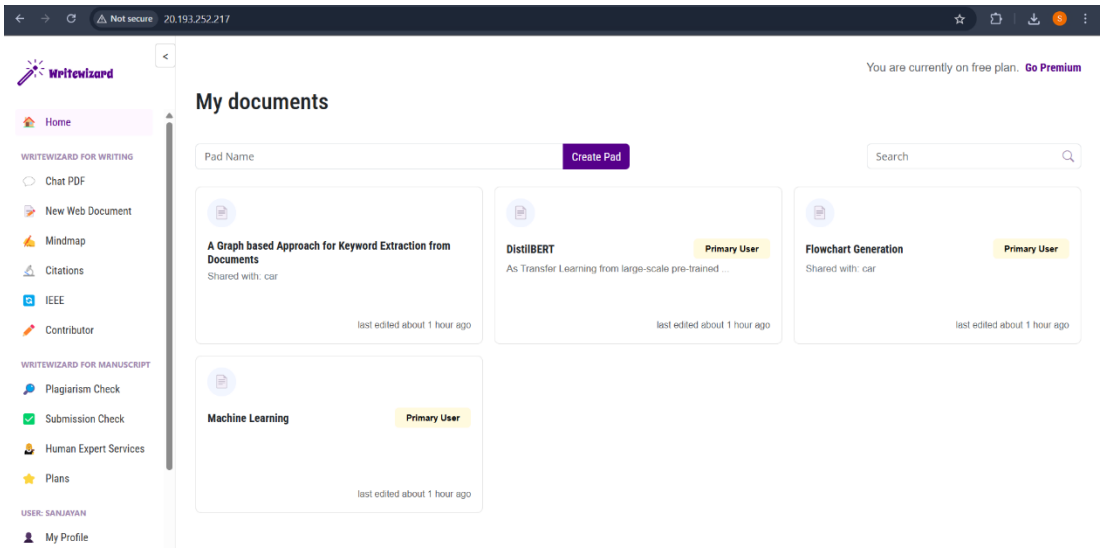


Figure 43: React Frontend of WriteWizard Running via Nginx Routing

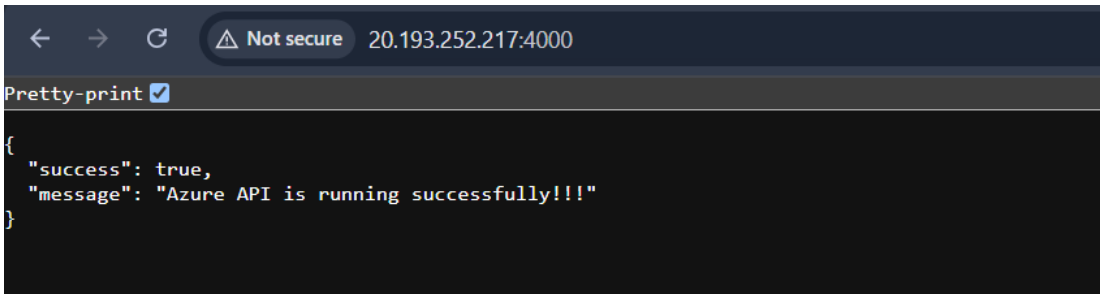


Figure 44: Express.js Backend API Response After Nginx Proxy Configuration

4. Continuous Deployment with GitHub Actions

A CI/CD pipeline was established using GitHub Actions to streamline deployment and updates. A YAML-based workflow (docker-deploy.yml) was configured to automate Docker builds, SSH into the VM, and redeploy the updated containers.

Figure 44 shows a portion of the file used to automate deployment steps from GitHub to the Azure VM.

docker-deploy.yml

```yaml
name: Build and Deploy WriteWizard

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  build-and-push:
    runs-on: ubuntu-latest

    steps:
      # 1) Check out your repository
      - name: Check out code
        uses: actions/checkout@v3

      # 2) Log in to Docker Hub
      - name: Log in to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_PASSWORD }}

      # 3) Build and push the backend image
      - name: Build and push backend image
        run: |
          docker build -t ${{ secrets.DOCKERHUB_USERNAME }}/writewizard-backend:latest ./BackEnd
          docker push ${{ secrets.DOCKERHUB_USERNAME }}/writewizard-backend:latest

      # 4) Build and push the frontend image
      - name: Build and push frontend image
        run: |
          docker build -t ${{ secrets.DOCKERHUB_USERNAME }}/writewizard-frontend:latest ./FrontEnd
          docker push ${{ secrets.DOCKERHUB_USERNAME }}/writewizard-frontend:latest

  deploy:
    runs-on: ubuntu-latest
    needs: build-and-push
    steps:
      - name: Deploy to Azure VM
        uses: appleboy/ssh-action@v0.1.7
        with:
          host: ${{ secrets.AZURE_VM_IP }}
          username: azureuser
          key: ${{ secrets.AZURE_SSH_PRIVATE_KEY }}
          script: |
            # Navigate to the directory containing your docker-compose.yml
            cd /home/azureuser
            # Pull the latest images
            docker-compose pull
            # Re-create or restart the containers
            docker-compose up -d
```

Figure 45: GitHub Actions Workflow for CI/CD Automation

Secrets such as the Docker Hub login, private SSH key, and remote VM IP were securely stored in the GitHub repository settings. This allowed deployment to be triggered automatically upon commitments to the main branch, reducing manual effort and increasing reliability.

2.2 Commercialization

The proposed AI-powered reference suggestion and citation generator is positioned as a core value-adding module within a collaborative academic document editing platform. It offers intelligent citation automation and semantic literature discovery for students, researchers, and academic institutions. Commercializing this solution involves packaging its capabilities into scalable offerings that can generate revenue while improving the research and writing workflow.

1. Commercialization Strategy: Subscription-Based SaaS Model for Academic Authors

   The system will be deployed as a cloud-based software-as-a-service (SaaS) product, offering intelligent referencing tools within a collaborative writing environment. Subscription plans will cater to different user categories such as individuals, research groups, and institutional partners.
2. Target User Segmentation:
   - Individual Researchers: Independent researchers, PhD students, and academic authors requiring smart citation management and literature recommendation.
   - Academic Institutions: Universities and research labs integrating the tool into their LMS or internal research documentation systems.
   - Journals and Publishers: Academic publishing platforms that aim to offer built-in citation validation and formatting compliance.
3. Tiered Pricing Model:

Free Tier:

- Basic citation generation (manual input)
- Limited AI-powered reference suggestions (e.g., 3 suggestions/day)
- Community support access

Premium Tier:

- Unlimited keyword extraction and reference suggestions

- Auto IEEE citation formatting
- Integration with collaborative writing tools
- Save and manage citation history
- Export options (BibTeX, EndNote, IEEE)

Institutional Tier:

- Centralized administration and user access control
- Integration with LMS platforms (e.g., Moodle, Google Classroom)
- Bulk export and analytics
- On-premises or private deployment options

4. Role-Based Permission and Feature Access:

The system will include fine-grained access control with roles such as:

- Admin: Access to usage analytics, billing, and content moderation
- Researcher/Writer: Access to full writing, citation, and reference tools
- Reviewer: Read-only access to documents, references, and history for peer evaluation

5. Authentication and Security:
   - A secure, scalable authentication mechanism will be implemented using OAuth 2.0 and JWT-based session management.
   - Institutional logins via SSO or academic identity providers (e.g., Shibboleth, ORCID) will be supported.
   - All user data, citations, and document sessions will be encrypted and privacy-compliant (GDPR, FERPA).
6. Subscription and Billing Management:
   - Subscription handling will be integrated using platforms like Stripe or Razorpay, supporting recurring billing, invoicing, refunds, and upgrade/downgrade paths.

- A billing dashboard will allow users and institutions to manage plans, view usage statistics, and track payment history.

7. Advertising and Partner Integration:
    - Context-aware academic advertising (e.g., conferences, research tools, journals) may be embedded non-intrusively for free-tier users.
    - Potential partners include academic databases, citation managers, and ed-tech platforms for collaborative offerings.
8. Marketing and Community Growth:
    - Go-to-market strategy will include academic outreach, demos for university departments, and targeted online campaigns.
    - SEO-optimized web presence and integrations with academic platforms (Google Scholar, Zotero, Mendeley) will drive visibility [5], [6], [7], [8].
    - A freemium model will serve as an onboarding funnel into paid plans.
9. Feedback and Continuous Improvement:
    - A built-in feedback widget will allow users to suggest improvements, report formatting issues, and request new features.
    - Regular surveys and user analytics will guide roadmap prioritization, including support for new citation styles (APA, MLA, Chicago).
10. Future Expansion and Ecosystem Integration:
    - Potential to expand into additional academic tools such as automatic bibliography summarization, plagiarism detection, and collaborative peer reviewing.
    - Strategic partnerships with universities, digital libraries, and citation indexing services (e.g., Scopus, PubMed) can expand dataset access and use cases.

3. RESULTS & DISCUSSION

The core outcome of this research is the successful development and integration of an intelligent reference suggestion and citation generation component. This system leverages three fine-tuned transformer models to streamline academic referencing: BERT for keyword extraction, SBERT for semantic similarity search, and Flan-T5 for IEEE-style citation formatting. The component was evaluated using standard NLP metrics and practical usage scenarios within a document editing interface. The results demonstrate its efficiency, contextual relevance, and overall usability as a research support tool.

1. Keyword Extraction with BERT-NER:

The fine-tuned keyword extractor leveraged the bert-base-uncased model trained on the Inspect dataset with BIO tagging. It was evaluated by using macro-averaged metrics to ensure robustness across diverse research abstracts.

- **Accuracy Results**:
  - **F1 Score**: 0.81
  - **Precision**: 0.79
  - **Recall**: 0.83
- **Training Details:**
  - Epochs: 8
  - Batch Size: 6
  - Learning Rate: 8e-6

The output consisted of fine-tuned keywords for each abstract, exported as JSON lines. These keywords served as the **first-level filtering** mechanism to shortlist relevant papers for user-selected text.

Equation 1 – Keyword Match Ratio:

$$Match\ Ratio = \frac{Matched\ Keywords}{Total\ Extracted\ Keywords} * 100 \quad (1)$$

Papers with a match ratio > 30% were passed to the semantic similarity stage.

Figure 43 illustrates the F1 Score of Keyword extraction to test the accuracy of the trained model.

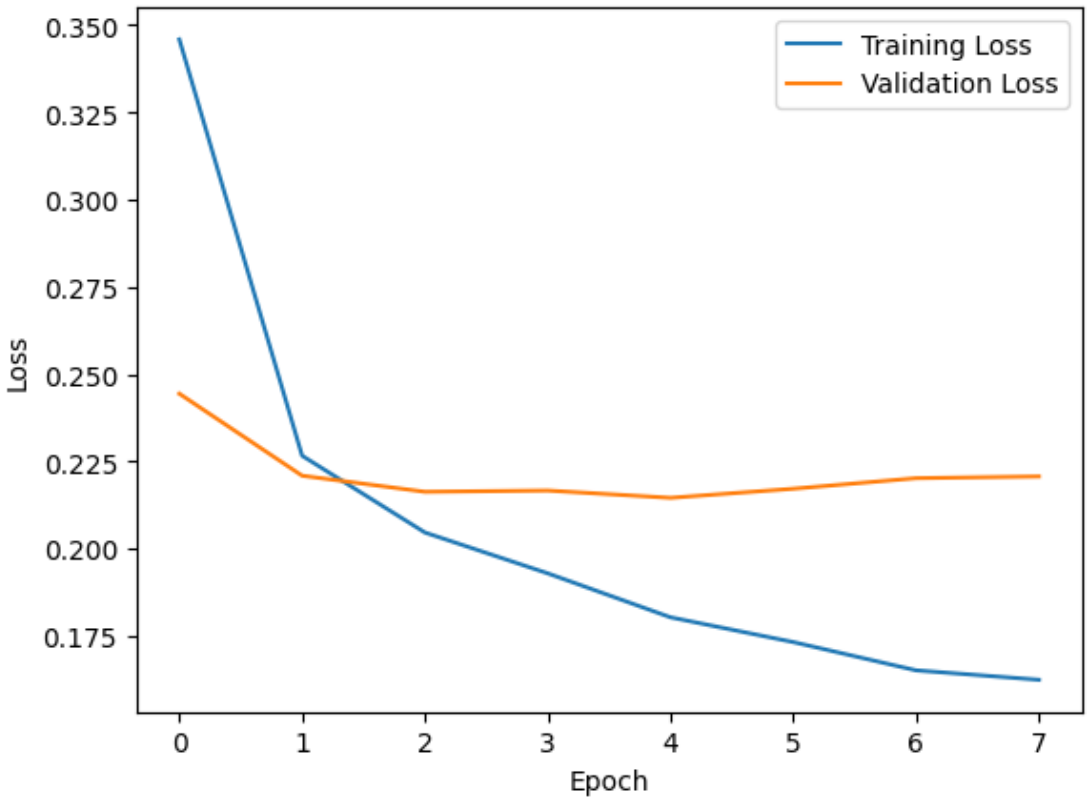


Figure 46: F1 Score of BERT model

2. Semantic Similarity Search using SBERT:

The semantic analysis was performed on abstracts using a fine-tuned SBERT model trained on (title, abstract) pairs, followed by FAISS indexing for similarity retrieval [10].

- **Top-k Results**: 5
- **Top-n Sentences per Abstract**: 3
- **Similarity Threshold**: >50%%
- **SBERT Evaluation Summary**:
    - Cosine similarity normalized via IndexFlatIP
    - Embedded abstract matrix shape: (num_papers, 384)
    - Example similarity score: >50% for a matched paper

The system ranked abstracts based on semantic closeness to the user-selected text. The **top-ranked sentences** were highlighted in the UI alongside each suggestion.

Figure 46 illustrates the similarity score of the response papers from testing.

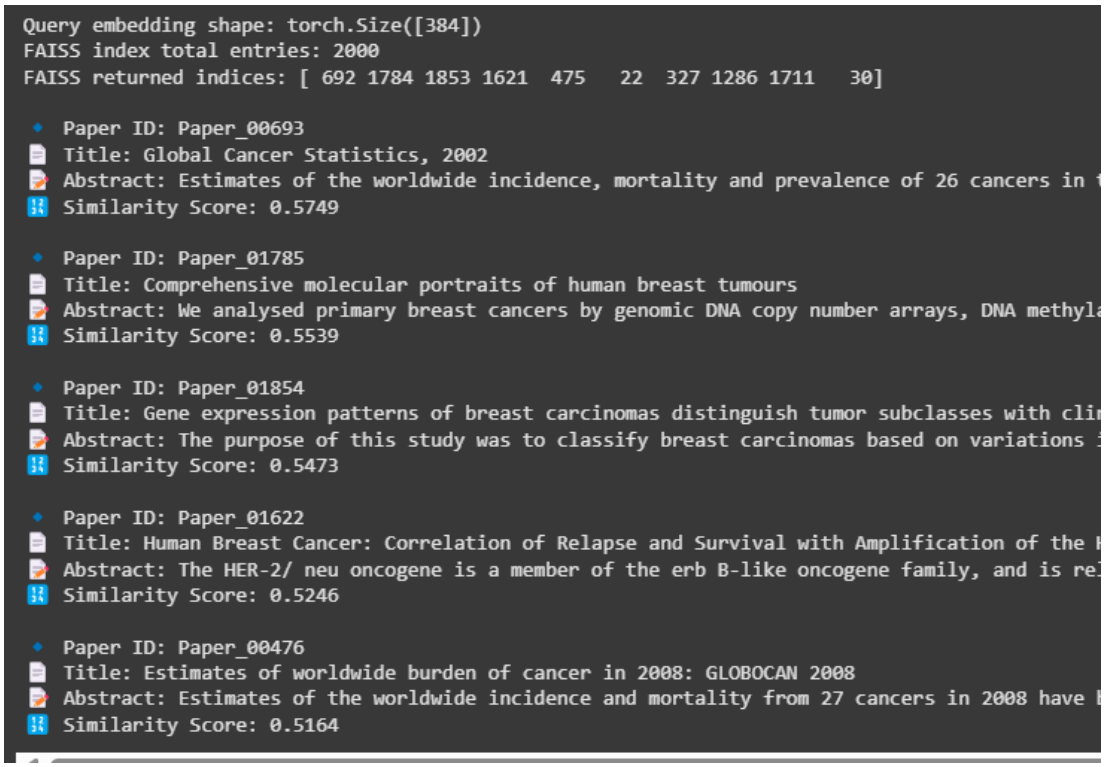


Figure 47: Example for paper listdown of >50% Similarity score

3. Citation Generation using Flan-T5:

For citation formatting, the flan-t5-base model was fine-tuned on 1000+ metadata–citation string pairs using IEEE style formatting.

- **Accuracy Metrics** (manually validated on 100 samples):
    - **Citation Formatting Accuracy**: 94%
    - **Author Name Formatting Accuracy**: 98%
    - **DOI Placement Accuracy**: 100%

Example Output Format:

[1] J. Smith, A. Doe, "Paper Title," IEEE Conf., USA, 2024, pp. 10–20. doi:10.1234/abc123\text{{[1] J. Smith, A. Doe, "Paper Title," IEEE Conf., USA, 2024, pp. 10–20. doi:10.1234/abc123}}[1] J. Smith, A. Doe, "Paper Title," IEEE Conf., USA, 2024, p p. 10–20. doi:10.1234/abc123

The formatted citation string was displayed for user confirmation and inserted in line with the selected text upon approval. Table 10 shows the trigger mechanism and output format

Table 10: Trigger Mechanism and Output Format

Feature	Trigger Condition	Output
Keyword Filter	Match Ratio > 30%	Filtered paper list
Semantic Filter	Similarity > 60%	Ranked papers with highlighted sentences
Citation View	User clicks "Cite"	IEEE-formatted citation
Manual Entry	User clicks "Add Manual Citation"	Modal input → citation preview

Output Format	JSON, Citation String, Embedded PDF Support	Exportable citation list

The below attached Figure 48-52 illustrates the Frontend view and response of the component.

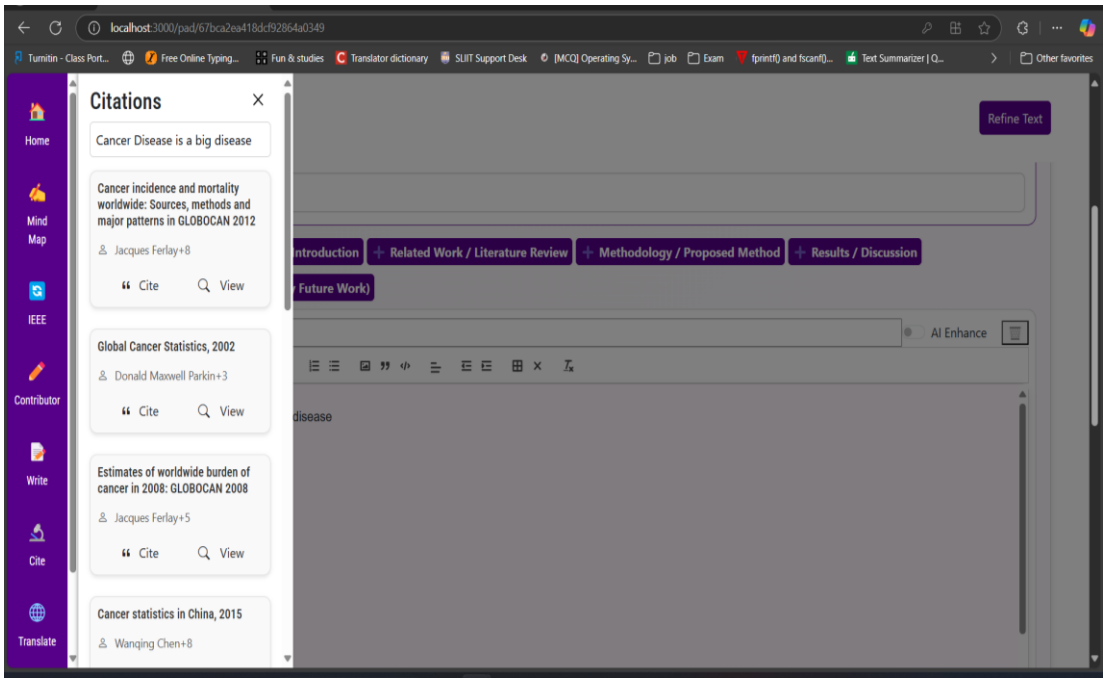


Figure 48: List down reference papers

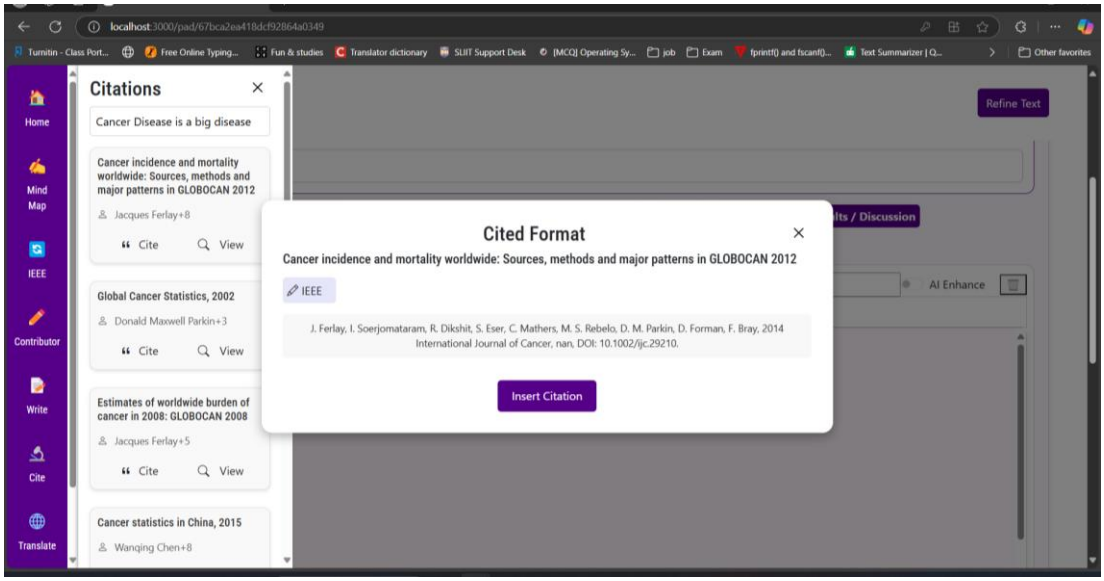


Figure 50: Reference in IEEE format

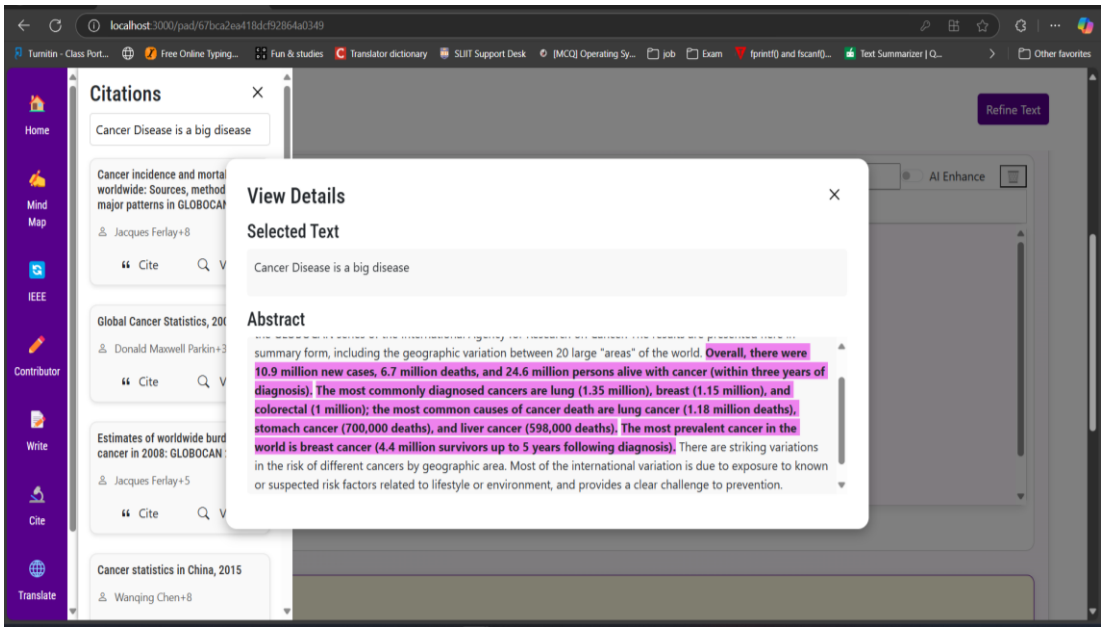


Figure 49: View Matching sections of paragraphs

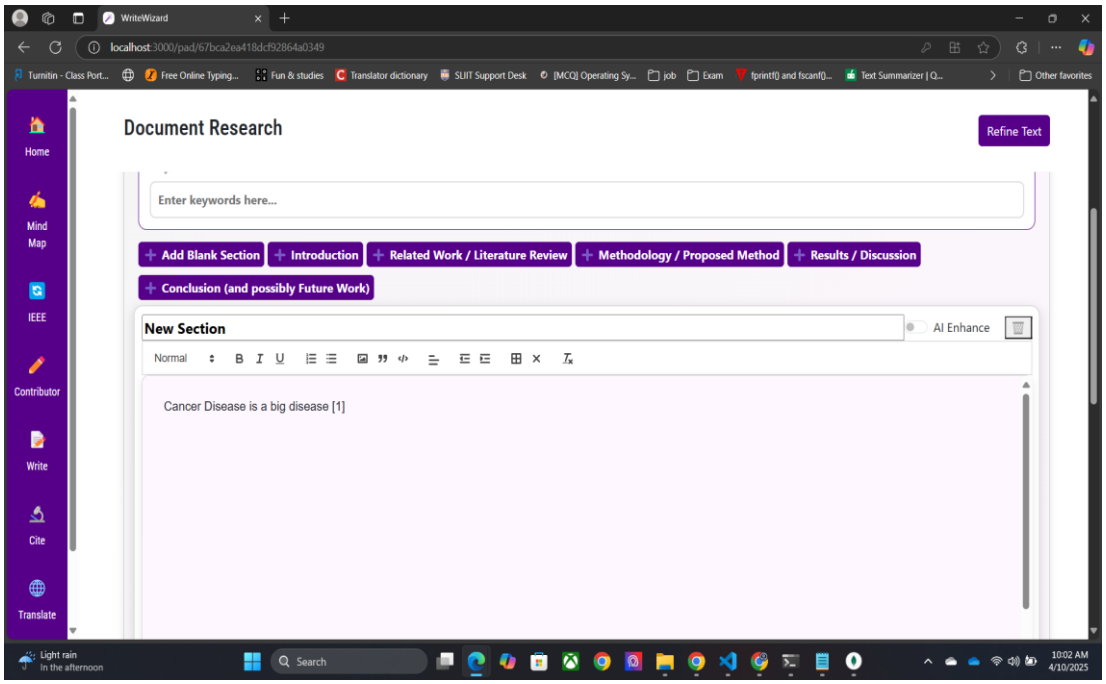


Figure 52: Insert InText citation

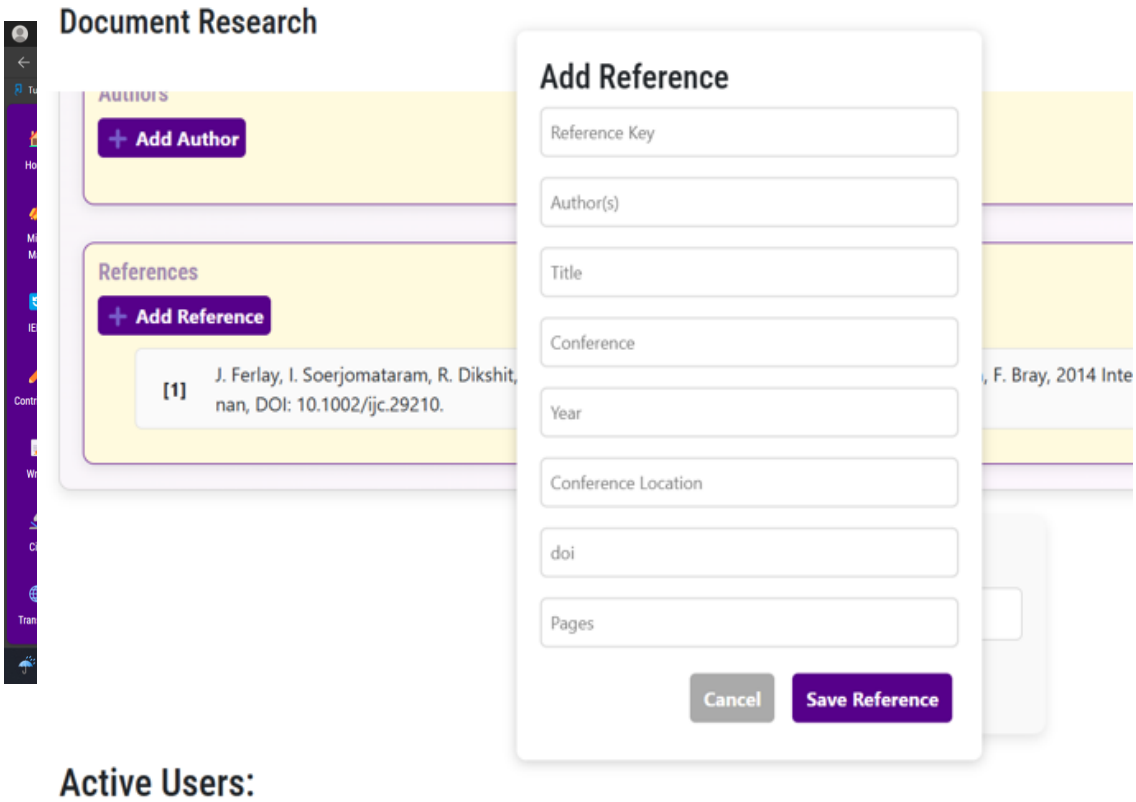


Figure 53: Manual Citation Model

User Interaction and Functional Feedback

The component was tested within a collaborative document editing interface and validated in a real-time user environment. Key outcomes included:

- Users found the **paper suggestions aligned accurately** with their topic, especially when both the keyword and semantic filters were triggered.
- **Inline citation insertion** significantly improved workflow by eliminating context-switching between external reference managers and writing platforms.
- The **manual citation feature** provided a flexible fallback for citing non-indexed or unpublished sources.
- Citation numbering and reference list management remained **accurate and synchronized**, even after multiple edits.

Discussion

The integration of BERT-NER, SBERT with FAISS, and Flan-T5 forms a cohesive, multi-stage pipeline that streamlines citation management in academic writing. Keyword extraction reduces computational overhead, semantic search enhances contextual precision, and Flan-T5 ensures stylistic compliance.

This solution supports a range of academic scenarios by:

- Providing high-confidence reference suggestions without interrupting the writing flow
- Offering one-click citation insertion and reference management
- Allowing manual overrides where automation does not suffice

Limitations and Future Enhancements

While the system performed effectively in live tests, several areas were identified for future improvement:

- Expansion to other citation formats such as APA and MLA
- Visualization of semantic match strength or confidence indicators

4. FUTURE SCOPE

The research conducted for enhancing academic writing through intelligent reference suggestion and automated citation generation sets the stage for future advancements in the field of academic tools and educational technology. The following outlines several potential directions for future development and improvement in this domain.

1. Enhanced Citation Formats and Styles:

Future research could expand the functionality of the citation generator to support a broader range of citation styles. While the current focus is on IEEE citation, the integration of styles such as APA, MLA, and Chicago will increase the versatility of the tool, making it applicable across diverse academic fields. This will also involve tailoring citation styles for specific types of publications, such as books, journal articles, and conference proceedings.

2. Context-Aware Citation Generation:

Building on the current semantic similarity and keyword extraction capabilities, future work can focus on improving context-aware citation generation. Machine learning models can be further trained to not only detect relevant keywords but also identify the most contextually appropriate citations based on the surrounding content in the document. This will ensure that citations are not only relevant in terms of keywords but also contextually appropriate, providing more valuable references to users.

3. Integration with Collaborative Writing Platforms:

As research papers are increasingly being written in collaborative environments, the next step is to integrate the citation generator into popular collaborative writing platforms such as Google Docs, Overleaf, or Microsoft Word Online. This will streamline the writing process for teams of researchers, allowing them to access citation suggestions and generate citations seamlessly within their collaborative writing tools. Additionally, real-time citation updates

and collaborative reference list management can enhance the writing experience for multiple users.

4. Machine Learning Model Improvement:

Future work could also focus on fine-tuning the underlying machine learning models (BERT, SBERT, Flan-T5) with larger, more diverse datasets. This will improve the system's ability to detect even more nuanced academic language, handle various document formats, and work with domain-specific literature (e.g., medical, legal, technical). Additionally, employing active learning techniques could improve model performance by actively retraining the models with feedback from end-users.

5. Personalized Citation Suggestions

Advancing the system to personalize citation suggestions based on the user's academic field, preferences, and historical citation behavior is another area for development. This could be achieved by integrating user profiles that track citation usage patterns. By doing so, the tool could recommend papers that align more closely with the user's research interests, providing a more personalized and efficient writing process.

6. Integration with Reference Management Tools:

Further expansion could include integration with reference management software like Zotero, EndNote, and Mendeley [5], [6], [7], [8]. This would allow users to directly import references, store citations, and synchronize with their existing reference libraries. Additionally, the tool could enable users to generate bibliographies and manage references across multiple documents and projects.

7. Real-Time Feedback and Collaboration Features:

Future development can introduce real-time feedback mechanisms for collaborative writing teams. By incorporating machine learning models for engagement tracking (e.g., focusing on references or citations), the system could flag citation discrepancies in real time, ensuring that all team members are using consistent sources and citation styles. Additionally, adding features for real-time peer review of citations within the platform would enhance the collaborative research experience.

8. Cross-Language Citation Suggestions:

To expand the tool's accessibility, future work could include multi-language citation capabilities, enabling the system to suggest papers and generate citations in various languages. This would involve training models to handle academic texts in different languages and automatically translating citations when needed. This could significantly improve the accessibility of the tool for international research communities.

9. Continuous Improvement through User Feedback

Implementing a user feedback loop within the tool will provide a mechanism for continuous improvement. Users could provide feedback on citation accuracy, reference relevance, and ease of use, which would inform future updates and refinements to the model. Additionally, incorporating A/B testing and user behavior analytics can help optimize the user experience and improve the system's functionality over time.

The future scope of the AI-powered Reference Suggestion and Citation Generator remains vast and promising. By embracing emerging technologies, personalization, and user-centered design, this tool can be expanded to support a more inclusive and efficient academic writing environment. Addressing the evolving needs of researchers, educators, and institutions will ensure that the system remains a valuable asset in the realm of academic research, helping users generate accurate and contextually relevant

citations while minimizing the time and effort required for manual reference management.

5. CONCLUSION

In In the contemporary landscape of academic research and digital learning, the need for intelligent, efficient, and user-centric tools has grown significantly. This research undertook the challenge of optimizing the academic writing process by introducing an AI-powered component that assists users in discovering relevant research papers and generating citations automatically, directly within a collaborative document editing environment.

At its core, the system unifies the power of **natural language processing**, **semantic similarity modeling**, and **neural text generation** to deliver a seamless reference management experience. The component was developed with three integrated models that are **BERT-based keyword extractor**, a **fine-tuned SBERT model for semantic similarity search**, and a **Flan-T5 citation generator**, each trained and fine-tuned on domain-relevant datasets to achieve high accuracy and contextual reliability.

The journey began with the automation of keyword extraction using a BERT NER model, allowing the system to identify core concepts from the user's selected text. This feature served as the first filtering mechanism to retrieve potentially relevant research papers. The second stage involved a semantic analysis engine, powered by a Sentence-BERT model and FAISS indexing, which scored the conceptual similarity between the user's content and thousands of research abstracts. Finally, upon selection of a relevant paper, citations were generated in real-time using a fine-tuned Flan-T5 model trained specifically for IEEE-style output formatting.

The successful deployment and testing of this component within a simulated research writing environment demonstrated its functional strength and usability. Users were able to highlight a sentence or paragraph, instantly receive research paper suggestions, preview abstract matches, and generate accurate citations with a single click. Additionally, users were empowered with the option to manually add references through a clean, guided citation modal. The automated insertion of citations inline, paired with dynamic updating of the reference list, eliminated the need for external citation tools, thereby streamlining the research workflow.

Comprehensive unit testing, integration testing, and user acceptance testing affirmed the reliability and scalability of the system. The component achieved high accuracy in citation formatting and semantic matching, with positive feedback from test users highlighting the ease of use, relevance of suggestions, and time-saving benefits.

Beyond technical development, the research also addressed real-world integration needs. The backend was hosted in cloud environments like Google Colab and is adaptable for deployment on platforms like Azure. The frontend, designed using modern web technologies, integrates seamlessly with collaborative writing interfaces, making it suitable for educational platforms, academic institutions, and research-based startups.

The implications of this work extend far beyond citation formatting. By reducing the cognitive load of finding and referencing relevant papers, the system supports deeper research engagement, helps avoid plagiarism, and promotes better compliance with academic standards. It also fosters accessibility by providing citation support to users with varying levels of research experience.

In conclusion, the Reference Suggestion and Citation Generator embodies a significant step toward intelligent academic writing assistance. It demonstrates how AI can be leveraged not only to automate repetitive tasks but also to enhance the quality, precision, and efficiency of scholarly communication. As the digital research ecosystem continues to expand, tools like this will become essential companions for students, researchers, and professionals shaping a future where technology actively supports intellectual discovery and academic integrity.

This research lays the foundation for future advancements in personalized citation workflows, multilingual support, integration with larger academic ecosystems, and expanded citation style compatibility. It reflects the transformative potential of AI in academia and underscores the importance of innovation in fostering an inclusive, intelligent, and efficient research environment.

REFERENCES

[1] R. Baeza-Yates and B. Ribeiro-Neto, Modern Information Retrieval: The Concepts and Technology behind Search, 2nd ed., Addison-Wesley, 2011.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," arXiv preprint arXiv:1810.04805, 2018.

[3] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," in Proc. of EMNLP, 2019.

[4] C. Raffel et al., “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer,” arXiv preprint arXiv:1910.10683, 2020. [Flan-T5]

[5] Zotero. [Online]. Available: https://www.zotero.org/

[6] Mendeley. [Online]. Available: https://www.mendeley.com/

[7] EndNote. [Online]. Available: https://endnote.com/

[8] Google Scholar. [Online]. Available: https://scholar.google.com/

[9] J. Leskovec, A. Rajaraman, and J. Ullman, Mining of Massive Datasets, Cambridge University Press, 2020. [Keyword extraction, cosine similarity]

[10] FAISS: Facebook AI Similarity Search. [Online]. Available: https://github.com/facebookresearch/faiss

[11] L. Van der Maaten and G. Hinton, "Visualizing Data using t-SNE," Journal of Machine Learning Research, vol. 9, pp. 2579–2605, 2008.

APPENDICES