

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

(DYNAMIC VISUALIZATION FOR COLLABORATIVE DOCUMENTS)

Sanjayan. C

(IT21375514)

BSc (Hons) degree 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**

(DYNAMIC VISUALIZATION FOR COLLABORATIVE DOCUMENTS)

Sanjayan. C

(IT21375514)

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
Sanjayan. C	IT21375514	

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 today's educational landscape, where digital tools are increasingly shaping how learners interact with content, there remains a significant gap in the way textual information is consumed, structured, and internalized. This research addresses that challenge through the development and deployment of an intelligent mind map generation system designed to enhance comprehension and engagement in academic settings. Built upon a fine-tuned transformer-based language model, the system converts structured or unstructured academic text into visually organized mind maps through three customizable modes: standard, extended, and simplified. These maps are further enriched with semantically matched images, offering a multimodal interface that bridges textual understanding with visual cognition. The core backend services are modularly designed and hosted on cloud infrastructure, supporting API-based integration and real-time inference. The system is embedded into a collaborative web application, WriteWizard, which includes a React-powered frontend and a Node.js backend that allows users to generate, edit, export, and share mind maps within a live document workspace. Semantic alignment, layout flexibility, and multi-user collaboration are central to the platform's design, enabling its application in a wide range of academic writing and study workflows. The research incorporates a complete methodology for prompt formatting, model interfacing, semantic image retrieval, and frontend visualization using D3.js. This thesis contributes to the field of educational technology by introducing a scalable and adaptable framework for AI-driven knowledge structuring, offering a blueprint for future systems that seek to support personalized, efficient, and collaborative academic learning environments.

Keywords: Mind Map Generation, Transformer Model, Semantic Visualization, Natural Language Processing, Collaborative Learning, Fine-Tuning, AI-Assisted Writing, D3.js Visualization

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 Software Engineering 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

DECLARATION	iii
ABSTRACT	iv
ACKNOWLEDGEMENT	v
TABLE OF CONTENTS	vi
LIST OF FIGURES	viii
LIST OF TABLES	xi
LIST OF ABBREVIATIONS	xii
1. INTRODUCTION	1
1.1 Background Study and Literature Review	1
1.1.1 Background study	1
1.1.2 Literature review	3
1.2 Research Gap	5
1.3 Research Problem	7
1.4 Research Objectives	9
1.4.1 Main objective	9
1.4.2 Specific objectives	9
1.4.3 Business objectives	10
2. METHODOLOGY	11
2.1 Methodology	11
2.1.1 Feasibility study / planning	13
2.1.2 Requirement gathering and analysis	24
2.1.3 Designing	30
2.1.4 Implementation	34
2.1.5 Testing	81
2.1.6 Deployment and maintenance	93
2.2 Commercialization	102
3. RESULTS & DISCUSSION	105
4. FUTURE SCOPE	109

5. CONCLUSION.....	111
REFERENCES	113

LIST OF FIGURES

Figure 1: Comparison of Mind Map Tools	6
Figure 2: Agile Project Management Cycle	11
Figure 3: Gantt Chart (Schedule Management)	16
Figure 4: User Stories	26
Figure 5: Use Case Diagram	26
Figure 6: Sequence Diagram	27
Figure 7: System Diagram	31
Figure 8: Flow of Automated Mind Map Generation	33
Figure 9: MS Planner Board	34
Figure 10: Work Breakdown Structure	35
Figure 11: Sample Dataset Collection for Mind Map Generation	37
Figure 12: Sample Dataset Collection for Summarized and Extended Text	37
Figure 13: Instruction-Based Prompt Formatting in Alpaca Style	38
Figure 14: Model Loading and LoRA Configuration	42
Figure 15: Training Configuration and SFTTrainer Setup	43
Figure 16: Training Loss Log	44
Figure 17: Saving Model and Tokenizer after Fine-Tuning	44
Figure 18: API Endpoint Implementations for Mind Map Generation	47
Figure 19: Fine-Tuned Model and Tokenizer Initialization with Preprocessing and Postprocessing Logic	48
Figure 20: Conditional Image-Node Matching and API Integration	50
Figure 21: WebSocket logic – User join, selection events, and structure updates.....	52
Figure 22: WebSocket logic – User exit handling and disconnection cleanup.....	53
Figure 23: REST API logic for mind map creation, and retrieval	55
Figure 24: REST API logic for mind map update, and user assignment	56
Figure 25: Class Diagram for Mind Map Data Model	57
Figure 26: fetchMindMap Function to Handle Mind Map Generation Request.....	60
Figure 27: traverseMindmap Function to Extract Hierarchical Node-Link Structure	61
Figure 28: fetchImageCatalog and preloadImages Functions to Prepare Image Data	62
Figure 29: fetchMatchedImages Function for Node Image Matching	63

Figure 30: getMatchingImage Function for Inline Image Rendering	64
Figure 31: generateMindmap Function to Initialize SVG and D3 Simulation	65
Figure 32: update Function to Refresh Visualization After Changes	67
Figure 33: reRenderNode Function to Redraw Nodes with Images and Interactive Text Editing.....	68
Figure 34: applyTreeLayout and buildHierarchy Functions to Build a Hierarchical Layout	69
Figure 35: ticked and calculateEdgePosition Functions for Dynamic Node Positioning and Edge Alignment	70
Figure 36: handleAddNode Function to Add New Node to the Mind Map	71
Figure 37: handleKeyDown Function to Manage Deletion of Nodes and Links.....	72
Figure 38: dragstarted, dragged, and dragended Functions for Node Repositioning.	73
Figure 39: handleSave Function to Serialize and Store Mind Map	74
Figure 40: downloadAsPNG Function to Export Mind Map as PNG Image	75
Figure 41: fetchPad and processSection Functions to Retrieve and Clean Document Content	77
Figure 42: buildFullMindmap Function to Generate Document-Level Mind Map ...	78
Figure 43: socketRef Function to Establish and Manage Real-Time Collaboration .	79
Figure 44: syncMindmapToDatabase Function to Persist and Broadcast Mind Map Updates.....	80
Figure 45: Unit and Integration Test Suite for Mind Map Generation Microservice	83
Figure 46: Pytest Execution Results for Unit and Integration Tests	84
Figure 47: Azure VM Configuration for Mind Map Generation Service	94
Figure 48: Nginx Configuration for Routing FastAPI Endpoints.....	95
Figure 49: FastAPI Application Running with LoRA Model on Azure VM.....	95
Figure 50: Postman Verification of Public Mind Map Generation Endpoint	95
Figure 51: Dockerfile Used for Hugging Face Image Matching Service	96
Figure 52: Docker Compose File for Frontend and Backend Service Orchestration	98
Figure 53: React Frontend of WriteWizard Running via Nginx Routing.....	99
Figure 54: Express.js Backend API Response After Nginx Proxy Configuration.....	99
Figure 55: GitHub Actions Workflow for CI/CD Automation	100

Figure 57: Map Modal Displaying Generated Mind Map with Customization	
Features	107
Figure 56: Mind Map Dashboard in WriteWizard	107
Figure 58: Document-Level Mind Map Generation Interface with Customization	
Features	108
Figure 59: Saved Mind Map Real-Time Collaborative Feature.....	108

LIST OF TABLES

Table 1: Cost Management	14
Table 2: Risk Management Plan	17
Table 3: Communication Management Plan.....	21
Table 4: Test Case for Standard Mind Map Generation	86
Table 5: Test Case for Extended Mind Map Generation	86
Table 6: Test Case for Simplified Mind Map Generation.....	87
Table 7: Test Case for Full Document Mind Map Generation	87
Table 8: Test Case for Image Matching with Nodes.....	88
Table 9: Test Case for Real-Time Collaboration	88
Table 10: Test Case for Node Addition and Deletion.....	89
Table 11: Test Case for Relation Addition and Deletion	90
Table 12: Test Case for Saving Mind Map	90
Table 13: Test Case for Exporting as PNG	91
Table 14: Test Case for Responsiveness on Mobile Devices	91
Table 15: Test Case for Invalid Input Handling.....	91

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
RAKE	Rapid Automatic Keyword Extraction
GCN/A-GCN	(Attentive) Graph Convolutional Network
LDA	Latent Dirichlet Allocation
BERT	Bidirectional Encoder Representations from Transformers
LoRA	Low-Rank Adaptation
PEFT	Parameter-Efficient Fine-Tuning
JSON	JavaScript Object Notation
SVG	Scalable Vector Graphics
CI/CD	Continuous Integration / Continuous Deployment
VM	Virtual Machine
UAT	User Acceptance Testing
D3.js	Data-Driven Documents (JavaScript Visualization Library)
GPU	Graphics Processing Unit
CPU	Central Processing Unit

1. INTRODUCTION

1.1 Background Study and Literature Review

1.1.1 Background study

The emergence of artificial intelligence has led to significant advancements in how academic content is created, managed, and shared. With increasing pressure on researchers to maintain high standards of structure, clarity, formatting, and citation accuracy, traditional document editing tools have proven insufficient in supporting these needs efficiently. Academic writing involves multiple interdependent stages, including literature exploration, content development, citation management, and peer collaboration. Most conventional tools are limited to basic text editing and formatting features, offering minimal assistance in handling the complexity of research writing. This has resulted in increased cognitive load, fragmented workflows, and a dependence on multiple external tools to complete a single research document.

To address these challenges, modern research environments are gradually integrating AI-driven components that introduce intelligent automation into the writing process. These components include automated citation generators, language refinement systems, and document formatting tools that work together to reduce manual effort while improving output quality. More recently, there has been a growing focus on incorporating visualization features that support knowledge management and enhance the collaborative writing experience. These improvements not only streamline content organization and academic compliance but also enable users to engage more deeply with the material through contextual suggestions, structural mapping, and enhanced clarity in document flow. Collaborative platforms that support such intelligent features are paving the way for a new standard in academic writing and knowledge sharing.

One such enhancement explored in this research is the automated generation of mind maps from textual content. Mind maps are visual tools that represent the relationships between key concepts in a structured, hierarchical manner. They support deeper cognitive engagement by allowing users to explore the organization of information beyond linear text. Originating from the work of Tony Buzan in the 1960s [1], mind

mapping has been widely adopted in educational, creative, and analytical fields as a method for idea generation and information analysis. In the context of academic work, mind maps provide a scaffolded structure that can simplify complex material and improve both comprehension and recall.

Scientific studies underline the relevance of such visual techniques: findings indicate that 65% of the population are visual learners [2], while the human brain processes images up to 60,000 times faster than text. Moreover, 90 percent of the information transmitted to the brain is visual in nature [3]. As Burton describes, “Visual aids are those sensory objects or images which initiate or stimulate and support learning,” while Kinder, S. James emphasizes that they are “devices which can be used to make the learning experience more real, more accurate and more active” [4]. Mind maps, as a form of visual aid, capitalize on these strengths by turning abstract concepts into structured, intuitive representations.

The use of mind maps in research is particularly valuable during early-stage ideation, literature synthesis, and conceptual analysis. However, traditional mind mapping tools often require manual input, which can be time-consuming and prone to structural inconsistencies. These tools also tend to offer limited customization, restricting how effectively users can tailor the visual representation to align with the complexity or specificity of academic content. Given the volume and depth of modern academic writing, there is a need for more intelligent systems that can interpret and transform text into well-organized visual formats that reflect content hierarchies and thematic relevance.

This research addresses that need by introducing an automated mind map generation component within a collaborative academic platform. By leveraging advanced natural language processing techniques, the component can extract relevant concepts and structure them into visual representations that are both flexible and academically meaningful. The aim is to support knowledge visualization as an integral part of the academic workflow, offering researchers a tool that enhances organization, improves understanding, and complements the broader AI-enhanced document editing environment.

1.1.2 Literature review

The Mind map generation has undergone significant transformation with the advent of advanced natural language processing (NLP) and machine learning techniques. These developments have aimed to overcome the limitations of manual diagramming, particularly in processing large-scale textual content and preserving the semantic structure of information. Earlier techniques such as Term Frequency–Inverse Document Frequency (TF-IDF) and TextRank emphasized statistical keyword extraction but often failed to capture deeper contextual relationships. These methods provided useful term salience but lacked the capacity to preserve thematic or hierarchical structure across paragraphs and sections.

To address these limitations, graph-based approaches emerged, offering more context-aware extraction methods. For example, the integration of Rapid Automatic Keyword Extraction (RAKE) with the Keyword Extraction using Collective Node Weight (KECNW) model improved performance by leveraging both statistical relevance and text structure to identify candidate concepts more accurately [5]. However, dependency parsing methods that aimed to extract syntactic relationships were found to be vulnerable to textual noise and sentence complexity, making them insufficient for modeling deeper semantic connections.

Recent research has proposed more dynamic models such as Attentive Graph Convolutional Networks (A-GCN), which assign contextual importance to dependency edges. These networks selectively filter relevant semantic links from syntactic clutter, thus enabling more robust concept hierarchies [6]. In parallel, developments in graph structuring have explored converting entire documents into relation graphs using sequence-to-graph transformation. These graphs are then refined through reinforcement learning techniques, which help eliminate redundant or incoherent nodes and edges, improving overall map clarity and interpretability [7].

Some systems, like English2MindMap, go further by generating multilevel mind maps that allow high-level nodes to expand into more detailed sub-maps. This functionality enhances both the readability and comprehension of complex documents, particularly

in academic or technical fields [8]. At the same time, a range of text mining algorithms has been developed to extract and organize concepts into coherent mind maps. However, ensuring semantic fidelity, especially when scaling across large datasets or multiple domains, continues to be a considerable challenge [9].

Other relevant techniques include Latent Dirichlet Allocation (LDA) for topic modeling and transformer-based models such as BERT for contextual summarization. While LDA is useful for discovering thematic clusters, it does not inherently support hierarchical structuring. On the other hand, BERT and similar transformers offer superior semantic understanding but are computationally demanding, especially for fine-tuning in domain-specific settings. These trade-offs limit their use in real-time applications or systems with constrained resources.

Collectively, these studies emphasize the importance of balancing contextual accuracy, computational efficiency, and output interpretability. The fusion of keyword extraction, semantic parsing, and graph-based modeling has enabled significant progress in automated mind map generation. Issues remain in terms of scalability, maintaining hierarchical outputs, and designing user-facing platforms that can dynamically adapt to different needs while preserving the underlying semantic structure of extended textual content. These shortfalls ultimately set the stage for further inquiry.

1.2 Research Gap

Mind map generation continues to be an area of active exploration in the field of natural language processing and knowledge visualization. However, despite recent advancements, several key gaps hinder its practical integration into real-time, collaborative environments. A primary limitation lies in the inability of existing systems to effectively retain semantic structure and hierarchy, especially when processing complex or large-scale documents. Many tools generate diagrams with flat or inconsistent arrangements, making it difficult to identify relationships between parent and child nodes or to follow the progression of interconnected ideas.

Another concern is the lack of support for collaborative interaction and customization. In most existing systems, users have limited control over modifying node relationships, adjusting the depth of information displayed, or visually reorganizing elements based on thematic relevance. These solutions are often designed for single-user operation, and real-time multi-user collaboration is either unsupported or minimal. Moreover, the rigidity in visual styling, interaction, and layout adjustment often prevents the creation of contextually relevant or user-specific representations, which are essential in academic or research workflows.

A further shortfall is the absence of intelligent image integration within the mind map generation process. While some tools allow for manual image embedding, they do not provide any context-aware suggestion or automated association of images with node content. This limits their potential for enhanced comprehension, particularly in visual learning or documentation tasks where images play a critical role in concept retention.

These limitations are illustrated in Figure 1, which provides a comparison of existing mind mapping tools across key criteria, including semantic preservation, collaboration, content interaction, visual customization, and image embedding capabilities.

Tool	Automatic Generation	Real-Time Multi-User Collaboration	Customization	Extended/Simplified Generation	Semantic Structure Preservation	Related Image Integration	Node/Relation Edit	Free or Paid
coggle	No	Yes	Partial (Basic)	No	Yes	No (Manual addition only)	Yes	Freemium (free tier, paid upgrade)
mind	Yes	Yes	Yes (Extensive)	Yes	Yes	No (Manual addition only)	Yes	Freemium (free for 3 maps, paid upgrade)
AYOA	Yes	Yes	Yes	Yes	Yes	No (Manual addition only)	Yes	Freemium (free basic, paid upgrade)
Mindnode	No	No (Single-user)	Yes	No	Yes	No (Manual addition only)	Yes	Paid (Trial available)
KMind	Yes	Yes	Yes	Yes	Yes	No (Manual addition only)	Yes	Freemium (free basic, premium upgrade)
Lucidchart	Yes	Yes	Yes	No	Partial (Optional layout)	No (Manual addition only)	Yes	Freemium
miro	Yes	Yes	Yes	Yes	Partial (Free-form or structured)	No (Manual addition only)	Yes	Freemium
Whimsical	Yes	Yes	Partial (Limited styling)	No	Yes	No (Manual addition only)	Yes	Freemium
Mindomo	Yes	Yes	Yes	Yes	Yes	No (Manual addition only)	Yes	Freemium
Notepad	No	Partial (via Google Drive)	Partial (Basic)	No	Yes	No (Manual addition only)	Yes	Freemium (free, paid upgrade available)
Noteshard	Yes	Yes	Yes (Extensive)	Yes	Yes	Yes	Yes	Free

Figure 1: Comparison of Mind Mapping Tools

In addition to these challenges, existing collaborative document editing tools such as Google Docs, Notion, and Etherpad also lack integrated mind map generation. These platforms focus primarily on real-time text editing, formatting, and version tracking, but do not support the transformation of document content into structured visual representations. Users must rely on external software to manually translate content into mind maps, which often results in fragmented workflows and a loss of contextual linkage between text and its visual structure.

Addressing these gaps requires the development of an integrated, intelligent system capable of generating accurate, hierarchical mind maps with support for collaborative interaction, contextual media integration, and user-driven customization, all embedded within a unified document editing environment.

1.3 Research Problem

The increasing volume and complexity of academic and technical documents demand more intelligent and structured ways of organizing information. While traditional word processors support linear document creation, they do little to help users understand how ideas are interconnected or how the overall structure of a document evolves across sections. This limitation becomes more apparent in collaborative environments, where multiple contributors work on different parts of a document, often with varying perspectives and levels of understanding. As a result, it becomes difficult to maintain coherence, thematic alignment, and structural clarity throughout the document lifecycle.

Although mind maps offer an effective solution for visualizing relationships and structuring content, most existing tools rely heavily on manual input, requiring users to extract concepts, organize nodes, and build hierarchies themselves. This manual approach is not only time-consuming but also subject to bias and inconsistency, particularly when working with lengthy, domain-specific documents that require deeper contextual analysis. Furthermore, these tools operate independently of the writing environment, meaning users must constantly shift between applications, disrupting workflow and weakening the connection between content and structure.

At the same time, advances in natural language processing and transformer-based models have opened new possibilities for automated content structuring. However, these models are not yet fully integrated into practical, user-oriented systems that support real-time collaboration, iterative content refinement, and interactive visualization. Most implementations of NLP-based mind map generation remain confined to academic prototypes or restricted environments that do not scale well to diverse document types or collaborative workflows. They also struggle with ensuring semantic accuracy, maintaining logical hierarchies, and adapting to user preferences.

Another layer of complexity arises when considering the need for customization and content enrichment. Users may wish to emphasize specific relationships, collapse or expand content based on audience needs, or embed visual elements such as images

that complement textual information. Current systems lack the adaptability to support these requirements in a seamless, real-time manner. Additionally, there is no cohesive framework that bridges textual content with a dynamic, editable visual representation within a unified environment.

Therefore, the research problem centers on the absence of an intelligent, collaborative system that can convert raw textual content into semantically rich, hierarchically structured, and visually adaptable mind maps. Such a system must operate within the user's existing workflow, respond to document-level changes, and support real-time collaboration and contextual enhancements. Without this integration, the potential of mind maps as a tool for research planning, academic writing, and knowledge dissemination remains largely untapped. Addressing this problem requires not just technological innovation but a thoughtful alignment of automation, usability, and collaboration within the digital writing experience.

1.4 Research Objectives

1.4.1 Main objective

To design and implement an intelligent system that enables the automatic generation of hierarchical mind maps from textual content, supporting enhanced content comprehension, organization, and visualization within a collaborative editing environment.

1.4.2 Specific objectives

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

- Provide real-time integration with collaborative document editing environments for simultaneous multi-user interaction.
- Support dynamic customization options, including node editing, layout adjustments, and visual styling.
- Enable interactive mind map visualization through an intuitive, user-friendly interface.
- Integrate relevant images into the mind map to enhance contextual understanding and visual clarity.
- Allow users to export mind maps in various formats.
- Offer functionality to adjust the depth or detail level of the mind map based on user preference.
- Ensure semantic structure preservation by accurately identifying relationships between concepts and organizing them hierarchically.
- Enable the generation of complete mind maps that reflect the full structure and flow of entire documents.

1.4.3 Business objectives

- Improve Knowledge Visualization and Retention: Enhance users' ability to understand and organize large volumes of content through structured, interactive mind maps that simplify complex information.
- Support Collaborative Workflows: Provide a platform that fosters real-time knowledge sharing and idea structuring among multiple users within a shared document editing environment.
- Enhance Productivity and Efficiency: Reduce the time and effort required to manually create and organize visual representations of content, improving the overall workflow for research, education, and documentation.
- Promote Educational and Research Innovation: Offer academic institutions and researchers a tool that supports effective planning, review, and presentation of research through visual means.
- Increase Platform Competitiveness: Add value to digital writing platforms by incorporating intelligent, user-centric mind mapping features that distinguish them from traditional document editors.
- Ensure Accessibility and Usability: Design the system to be lightweight, flexible, and adaptable across different usage scenarios, supporting a wide range of users with varying technical expertise.

2. METHODOLOGY

2.1 Methodology

Methodology in research describes the systematic approach and set of procedures used to collect, analyze, and interpret data. It establishes a clear roadmap for addressing research questions and achieving project objectives while ensuring that the results are reliable and reproducible. In essence, the methodology is the guiding framework that shapes every phase of the research process.

This study adopts an Agile approach as its core methodology. Agile provides a flexible and iterative framework that has been successfully adapted from software development to various research domains. By breaking the project into smaller, manageable phases, Agile facilitates continuous progress, rapid feedback, and prompt adjustments to evolving requirements. This approach is particularly valuable in complex projects such as the development of a system for automated mind map generation, where ongoing refinement and user feedback are crucial.

1. Agile Project Management for Research Development

In this research, Agile methodology serves as the foundation for managing the project's progress. The approach supports an incremental development strategy where each phase builds on the achievements of the previous one. This allows the research team to continuously reassess and refine both the research direction and the technical implementation. Regular reviews ensure alignment with objectives. The Agile project management cycle, shown in Figure 2, illustrates the iterative flow of planning, development, testing, and feedback.

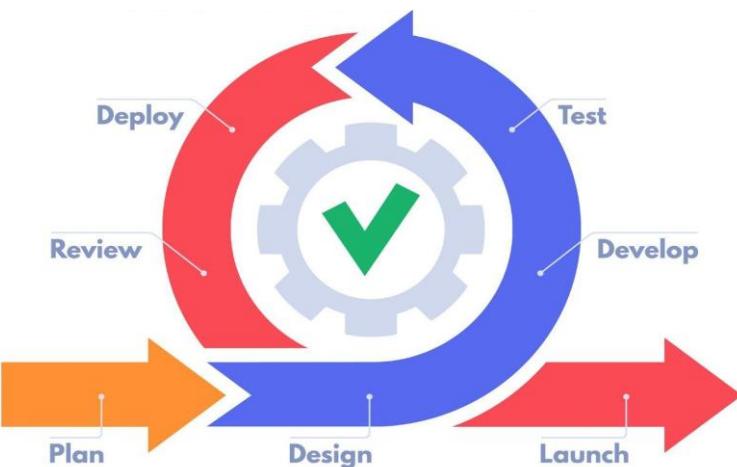


Figure 2: Agile Project Management Cycle

2. Multi-Phase Development Process

The research is organized into a multi-phase process that sequentially addresses critical aspects of the project. The phases include initial requirement identification, conceptual design, detailed system planning, development, testing, deployment, and post-deployment maintenance. Each phase is designed to capture essential insights and allow for iterative improvements based on stakeholder feedback and experimental results. This structured progression enables efficient management of project complexities and ensures that each component of the mind map generation system is thoroughly validated.

3. Iterative Progress and Team Collaboration

A key element of the chosen methodology is the emphasis on iterative progress. Rather than attempting to develop the entire system in one continuous effort, the project is divided into smaller increments, each focusing on a specific functionality such as text processing, semantic analysis, or interactive visualization. Regular team meetings and feedback sessions play a vital role in this iterative process, ensuring that insights gained during one phase inform the next. This collaborative environment enhances problem-solving and allows the research team to adjust quickly to any unforeseen challenges.

4. Flexibility and Adaptability

The dynamic nature of automated mind map generation, particularly in a collaborative setting, necessitates a flexible approach. The Agile methodology employed in this research supports rapid adaptation to changes in user requirements or technical challenges. By maintaining a flexible development process, the project can incorporate new insights and improve its design continuously, ensuring that the final system is robust and meets the evolving needs of its users.

2.1.1 Feasibility study / planning

This phase determines the feasibility of implementing an automated mind map generation system that converts document content into structured, hierarchical visual representations. The feasibility study for this research is organized into several key areas:

2.1.1.1. Technical feasibility:

Data Availability: The system will rely on the availability of large-scale textual content from academic documents and collaborative inputs. The possibility of manually collecting and annotating data from local contexts has been evaluated to ensure that sufficient data exists for analysis and testing.

Hardware and Software Requirements: The project requires hardware capable of processing extensive text data and rendering interactive visualizations. The use of modern computing environments, along with open-source libraries for natural language processing and visualization such as D3.js, ensures that the necessary computational resources are accessible. Additionally, the system is designed to integrate with existing collaborative document editing platforms, ensuring a seamless real-time user experience.

Integration Feasibility: The technical design has been assessed to confirm that automated mind map generation, semantic structure preservation, and image integration can be cohesively implemented. The system architecture is built to support real-time data processing and interactive customization, ensuring compatibility with both cloud-based and local deployment environments.

2.1.1.2. Economic feasibility:

Budgetary Considerations: The financial feasibility has been examined by estimating the costs associated with development, cloud hosting, and user testing. The use of open-source tools minimizes licensing expenses while cloud services provide scalable computing resources.

Resource Allocation: An evaluation of available research personnel indicates that the necessary expertise in natural language processing, web development, and data visualization is accessible. This ensures that both the development and testing phases are adequately supported.

Return on Investment: The potential benefits, including improved productivity, enhanced collaboration, and more effective visualization of complex documents, are expected to justify financial investments.

Table 1 summarizes the estimated expenses.

Table 1: Cost Management

Type	Cost
Cloud Hosting and Internet Usage	LKR 20000
Publication cost	LKR 45000
Stationary	LKR 8000
TOTAL	LKR 73000

2.1.1.3. Legal and ethical feasibility:

Data Privacy and Compliance: The system will process large volumes of textual and visual data obtained from academic documents and collaborative sources. Measures have been evaluated to ensure that data collection and processing comply with relevant copyright and data protection regulations. Informed consent procedures and anonymization methods are planned for any sensitive information.

Intellectual Property Considerations: The project will utilize open-source libraries and publicly available datasets, and all software components will adhere to proper licensing protocols. These practices mitigate potential legal risks and ensure ethical usage of all content and technologies integrated into the system.

2.1.1.4. Operational and user acceptance feasibility:

Data Collection and Processing: The system's ability to efficiently collect and process document content is a critical aspect of its operational feasibility. A robust data pipeline has been evaluated to ensure that textual inputs from various sources

are accurately extracted, preprocessed, and transformed into structured formats suitable for mind map generation.

Model Deployment: The deployment strategy for the system has been thoroughly evaluated to guarantee reliable implementation in diverse environments. The architecture is designed to support real-time processing and visualization of document content, with built-in provisions for continuous monitoring and maintenance. This approach ensures that the system remains robust and responsive while accommodating evolving user needs and operational demands.

System Integration: The solution is designed to seamlessly integrate with existing collaborative document editing platforms. An evaluation of the operational workflow confirms that the system can process document content in real time and render interactive mind maps without disrupting ongoing activities. This integration ensures that users can generate and interact with visual representations of their documents within a familiar editing environment.

User Interface and Experience: The interactive visualization component is engineered to provide an intuitive and user-friendly interface. It offers dynamic customization options such as node editing, layout adjustments, and visual styling, allowing users to tailor the mind map to their specific needs. Early feedback from pilot tests suggests that the interface meets the requirements of academic and research communities, enhancing the overall user experience and supporting efficient knowledge management.

2.1.1.5. Time and schedule feasibility:

Project Timeline: A comprehensive schedule has been developed outlining the key milestones for requirements analysis, system design, iterative development, testing, and deployment. This timeline ensures that each phase builds upon previous insights, facilitating continuous improvement throughout the project.

Monitoring and Contingencies: Regular progress reviews and agile management practices have been established to monitor project milestones. Contingency plans are

in place to address any unforeseen delays, ensuring that the project is completed within the allocated time frame.

Figure 3 illustrates the Schedule Management plan.



Figure 3: Gantt Chart (Schedule Management)

2.1.1.6. Social and cultural feasibility:

System Performance: The proposed system is designed to handle the processing of extensive textual data and render interactive visualizations in real time. The technical architecture has been evaluated to confirm that it can support high user loads and large datasets without compromising performance.

Adaptability to User Needs: The system's design includes dynamic customization features that allow users to adjust the visual presentation of mind maps according to their preferences. This ensures that the solution remains relevant across diverse academic and research contexts, making it a flexible tool for different user requirements.

Cloud and Local Deployment: The system is engineered to be scalable and adaptable to various operational settings. It can handle high user loads and large datasets without performance degradation. This flexibility ensures that the solution is accessible to a broad range of users, including academic and research communities, and aligns with diverse cultural and educational practices.

Other than these feasibility studies, the risk management plan and communication management plan have been done.

2.1.1.7. Risk management plan:

The following risk management plan identifies potential threats to the successful development and deployment of the automated mind map generation system. It outlines the triggers, assigns responsibility, and proposes contingency strategies to mitigate each risk. Table 2 summarizes the key risks, triggers, responsible owners, responses, and required resources.

Table 2: Risk Management Plan

Risk	Trigger	Owner	Response	Resource Required
Project Team Unavailability	Illness or sudden	Project Leader	Inform supervisors;	Project Schedule Plan;

	absence of team members		redistribute tasks among remaining team members; activate backup personnel plan	Backup Resources
Stakeholder Revision Requests	Panel or supervisor requests changes	Project Leader	Implement changes promptly; update documentation and project backlog; communicate revisions clearly	Meeting Log; Updated Product Backlog
Integration Challenges	Incompatibility between system components or platforms	Project Leader	Reassess integration strategy; conduct additional integration tests; consult with platform experts	Integration Testing Framework; Technical Support
Data Quality and Availability	Insufficient or low-quality data during collection	Project Leader	Identify supplementary data sources; employ data cleaning procedures; schedule further	Data Quality Assurance Plan

			collection sessions	
Performance and Scalability Issues	System slowdown under heavy data or user load	Project Leader	Optimize code and architecture; implement scalable cloud solutions; conduct stress testing	Cloud Services; Performance Testing Tools
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
Legal and Ethical Compliance Risks	Non-compliance with copyright or data protection regulations	Project Leader	Conduct regular legal reviews; ensure secure data handling; obtain necessary consents	Legal Consultation; Compliance Guidelines

2.1.1.8. Communication management plan:

The Communication Management Plan ensures that all team members, supervisor, and co-supervisor receive timely and relevant information to effectively fulfil their roles throughout the project. The plan outlines the communication objectives, the channels through which communication will occur, and the specific meeting arrangements designed to foster clarity and prompt decision-making.

Communication Objectives:

- Communication must be proactive, ensuring that all updates are provided in the right format and with appropriate content for the targeted audience.
- Communication should be sufficient, delivering all necessary information in a clear and concise manner while avoiding unnecessary repetition.
- Communications must be timely to address issues as they arise and maintain project momentum.

Communication Media:

1. Email – For official updates, documentation sharing, and formal communication with supervisors and stakeholders.
2. MS Teams – For real-time meetings, collaborative discussions, and file sharing within the team.
3. On-Campus Meetings – Face-to-face interactions are arranged as needed, particularly for meetings with the supervisor and co-supervisor. These sessions are used to discuss critical project milestones, resolve technical challenges, and provide formal feedback in a direct setting.
4. Phone Calls – For urgent matters and direct communication among team members and supervisors.
5. WhatsApp – For quick updates, informal discussions, and immediate coordination.
6. MS Planner – For task management, scheduling, and tracking progress across the different phases of the project.)

Table 3: Communication Management Plan

Meeting Type	Attendees	Purpose	Frequency	Agenda Items
Planning Kick-off Meeting	Supervisor, Co-supervisor, All Team Members	Formally launch the planning phase; define project scope, novelties, roles, and expectations.	Once at project inception.	Establish planning timetable, project scope, governance structure, roles, risk assessment, and review initial communication strategies.
Execution Kick-off Meeting	Supervisor, Co-supervisor, All Team Members	Initiate the execution phase; ensure all stakeholders understand their responsibilities.	At the start of major project phases	Present the detailed project work plan, review meeting logs, agree on dispute resolution and escalation methods, and outline quality assurance and change management.
Internal Status Meeting	All Team Members	Review ongoing progress, discuss developments, and address emerging issues.	Weekly	Provide progress updates, review accomplishments, discuss new risks or issues, confirm milestones, and address any

Meeting Type	Attendees	Purpose	Frequency	Agenda Items
				modifications needed to tasks.
Supervisor Review Meeting	Supervisor, Co-supervisor, All Team Members	Monitor project performance and assess major adjustments in scope or strategy.	Twice a week or as needed	Evaluate progress against the work plan, discuss scope or methodology changes, review budget and resource usage, and update risk management and quality control.
Project Steering Committee Meeting	All Team Members	Discuss overall project status and receive formal approvals at key milestones.	Once a month or at major milestones	Review project status, discuss unresolved issues, update risk and issue management, review budget and work history, and obtain official endorsements.
Pre-Milestone Review Meeting	Supervisor, Co-supervisor, All Team Members	Hold face-to-face discussions with supervisors prior to key milestones to clarify expectations and	Prior to each major milestone (Proposal, PP1, PP2, Research)	Review progress, discuss critical technical issues, refine project scope, and confirm

Meeting Type	Attendees	Purpose	Frequency	Agenda Items
		technical challenges.	Paper Submission)	upcoming deliverables.
Change Control Meeting	Supervisor, Co-supervisor, All Team Members	Prioritize and discuss change requests or panel inquiries.	As needed following major reviews (Proposal, PP1, PP2)	Review proposed changes, discuss panel feedback, agree on modifications to implement, update project documentation, and assign new tasks if required.
Project-End Review Meeting	Supervisor, Co-supervisor, All Team Members	Evaluate overall project performance and document lessons learned.	Once at project completion or major phases	Assess key accomplishments, review overall performance, discuss issues encountered and resolutions, summarize lessons learned, and propose future improvements.

2.1.2 Requirement gathering and analysis

The Requirement Gathering and Analysis phase was a pivotal step in defining the project's scope, objectives, and constraints. During this phase, both functional and non-functional requirements were identified to ensure that the automated mind map generation system aligns with user needs and adheres to established quality standards.

2.1.2.1. Functional requirements

Functional requirements specify the features and capabilities that the automated mind map generation system must possess to meet its intended goals. For this research, these requirements were derived from structured interviews with stakeholders such as educators, students, and experts in collaborative document editing. The aim is to ensure that the system effectively transforms document content into a structured, interactive, and customizable mind map. The key functional requirements for the project are as follows:

Hierarchical Mind Map Generation: The system is required to transform input text into structured, hierarchical mind maps that visually represent the semantic relationships between concepts. For this research, a custom dataset was prepared that reflects the contextual meaning and logical structure of academic content, capturing how information is organized across different levels of depth. This dataset serves as the basis for fine-tuning the model that converts document content into an organized mind map. The resulting system processes provided text and outputs a mind map that preserves the overall flow and logical connections within the document.

Real-Time Processing and Visualization: The system must support data processing to enable immediate generation and updating of mind maps as document content changes. A responsive user interface is essential so that users can view and interact with the generated mind maps without noticeable delay.

Interactive Customization: Users should be able to modify the generated mind maps through node editing, layout adjustments, and visual styling options. Customization features empower users to tailor the mind map to their specific requirements, ensuring that the visual representation is both informative and user-friendly.

Image Integration: The system should provide the option to automatically associate and integrate relevant images with corresponding nodes. This visual enrichment is intended to enhance contextual understanding and improve overall comprehension.

Export and Reporting Capabilities: The system must allow users to export the generated mind maps in various formats, such as PNG, SVG, and JSON. In addition, it should offer reporting features to capture user activity and system performance metrics.

Session Management: In a collaborative environment, the system needs to track individual user sessions and maintain consistency across simultaneous editing sessions. This ensures that changes are accurately reflected for all participants in real time.

Data Logging and Traceability: The system should log key actions and changes within the mind map generation process. This includes tracking user input, customization adjustments, and export operations to provide an audit trail for further analysis.

The gathered requirements were validated through iterative feedback sessions and follow-up interviews with stakeholders. Their inputs were analyzed to refine and prioritize functionalities, ensuring that essential features were clearly defined and any ambiguity eliminated. A priority matrix was created to classify each requirement as essential, desirable, or optional, guiding the development process. These functional requirements were subsequently mapped into user stories; Figure 4 illustrates the user stories derived from this process. In addition, Figure 5 presents a use case diagram that details the interactions between various user roles and the system, and Figure 6 outlines a sequence diagram showing the operational flow from data extraction to mind map generation.

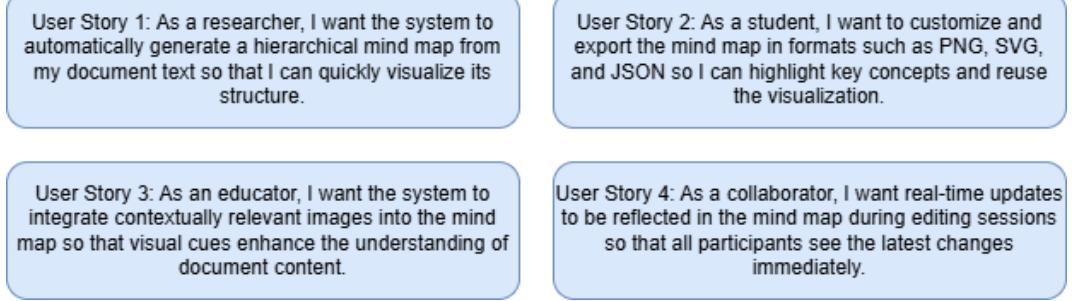


Figure 4: User Stories

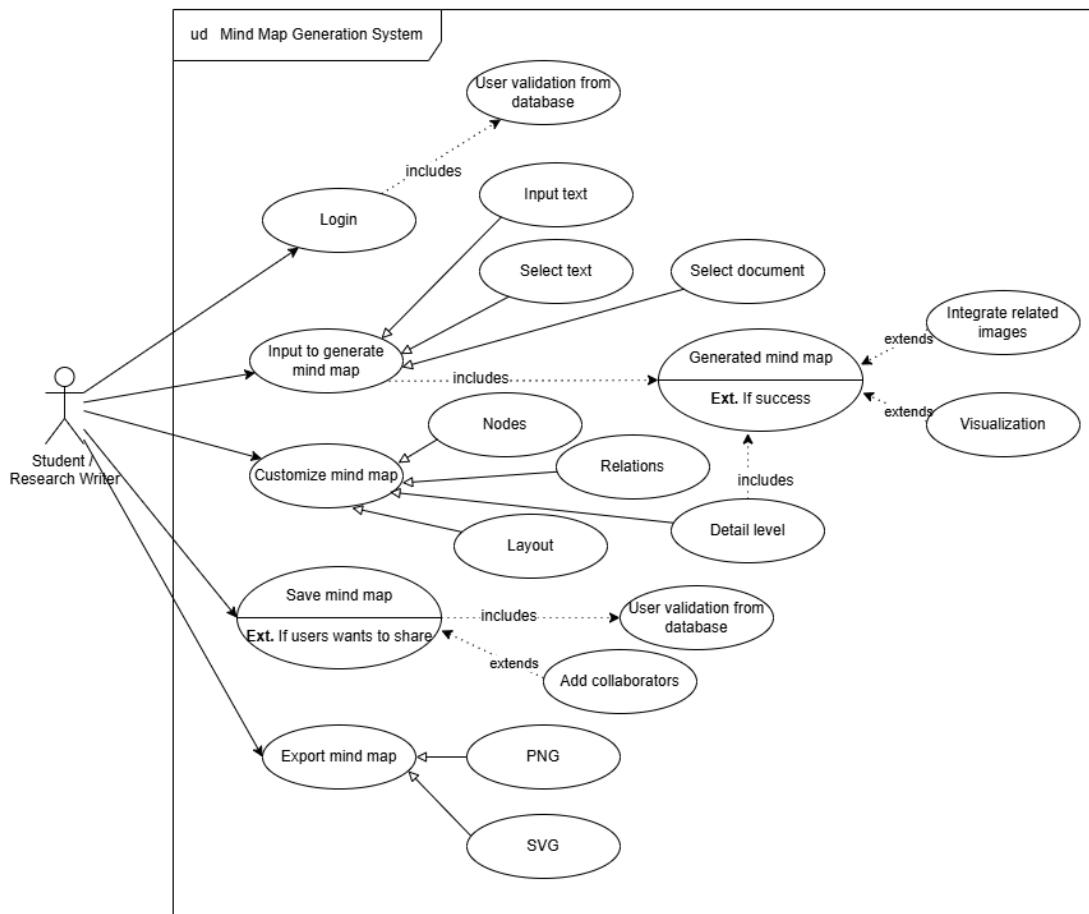


Figure 5: Use Case Diagram

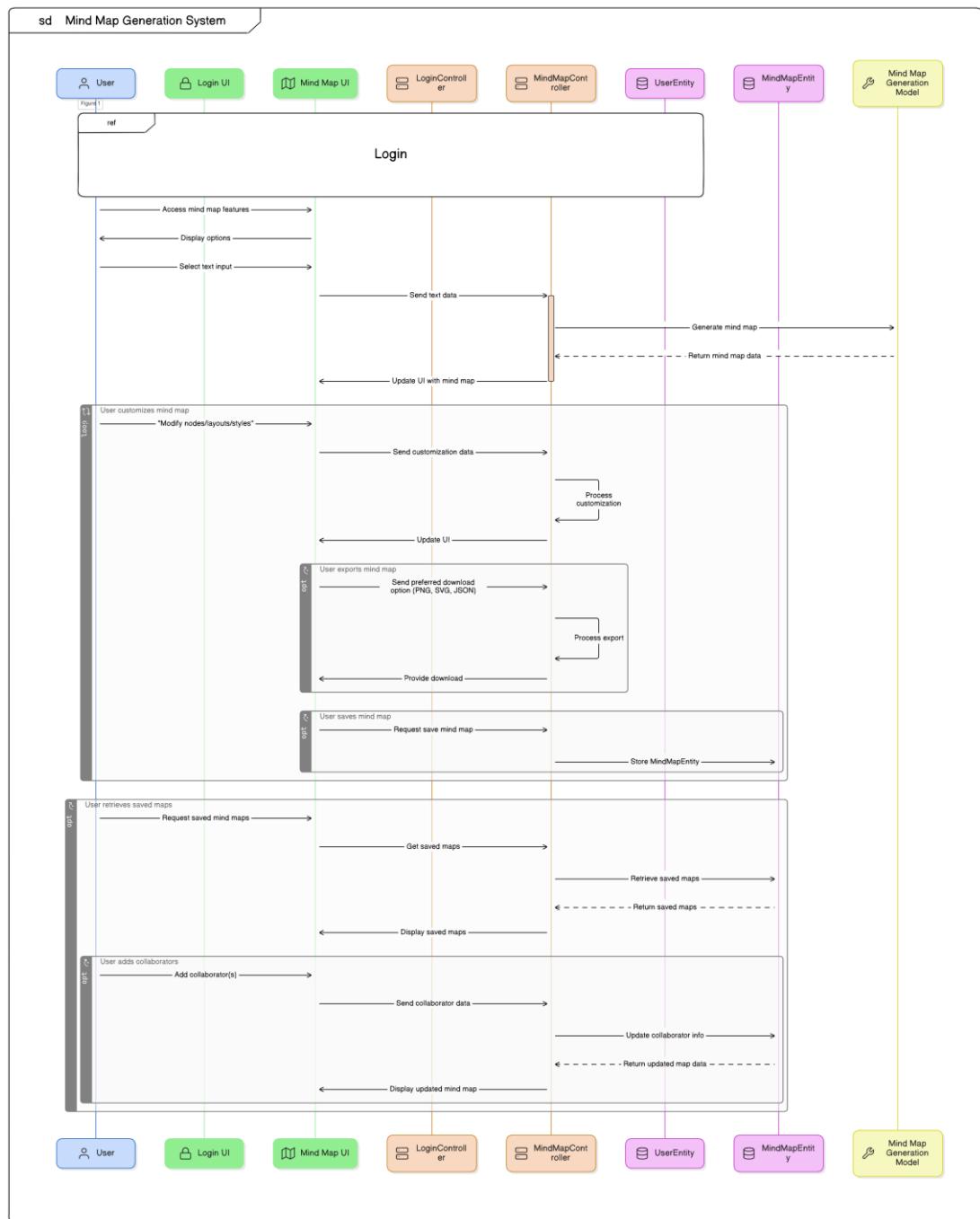


Figure 6: Sequence Diagram

2.1.2.2. Non-functional requirements

Non-functional requirements define the qualitative aspects of the automated mind map generation system. These requirements ensure that the system operates reliably, performs efficiently, and aligns with user expectations in terms of usability, scalability, and ethical compliance. These were identified through interviews with students, researchers, and academic professionals who provided insights on how the system should behave in real-world usage.

The performance of the system is crucial, particularly in real-time scenarios where users expect minimal delays during mind map generation, customization, or collaborative editing. The system must be capable of delivering consistently fast response times during user interactions, especially when processing large documents or handling multiple simultaneous users.

Usability is another essential quality. The user interface must be intuitive and easy to navigate for users of varying technical proficiency. This includes clear visual indicators, accessible interaction controls, and consistent layout structures that support seamless generation and editing of mind maps.

System availability must be maintained during typical usage hours, particularly in collaborative academic settings. The system should be accessible across devices and browsers with minimal downtime, and it should support consistent synchronization during collaborative sessions without data loss.

The accuracy of the generated mind maps must reflect the original semantic structure of the input text. The system should be capable of preserving contextual relationships while producing hierarchical representations that are clear and logically organized.

Reliability ensures that the system remains stable and functions correctly under different use cases and load conditions. This includes handling unexpected user actions, network interruptions, or large datasets without crashing or producing inconsistent outputs.

Scalability has been considered to support growing numbers of users and documents over time. The system architecture must allow for future improvements and increased traffic without affecting core functionalities.

Ethical considerations include the handling of user data, particularly in collaborative editing contexts. Users' content should be managed securely, and all features must comply with standard data handling and privacy practices.

These non-functional requirements provide the framework for designing a system that is not only efficient and effective but also user-friendly and ethically sound.

2.1.3 Designing

The Designing phase of the Software Development Life Cycle provides an opportunity to define a comprehensive architectural blueprint for the automated mind map generation component. At this stage, the focus is on creating a cohesive system that converts document content into interactive, hierarchical mind maps for enhanced knowledge visualization. Detailed planning ensures that all modules and features are integrated seamlessly into the collaborative document editing environment while keeping internal operations such as model training and database management hidden from the end user. This phase establishes the structural foundation necessary to support real-time processing, dynamic customization, and collaborative sharing of mind maps.

The design process emphasizes clarity, modularity, and scalability. The overall architecture is carefully crafted to ensure that each component, from text pre-processing to interactive visualization, is optimized for performance and usability. Through meticulous planning and iterative refinement, the design guarantees that the system meets user requirements and supports future enhancements without disruption.

System architecture diagram:

The system architecture diagram visually represents the interactions between the core components of the research system. It shows how the system processes text from documents, converts it into a structured JSON mind map, and displays the map through an interactive visualization interface. A Business Logic Layer coordinates the data flow among these modules to preserve the semantic structure of the content while supporting real-time collaboration features. Figure 7 illustrates the high-level system architecture.

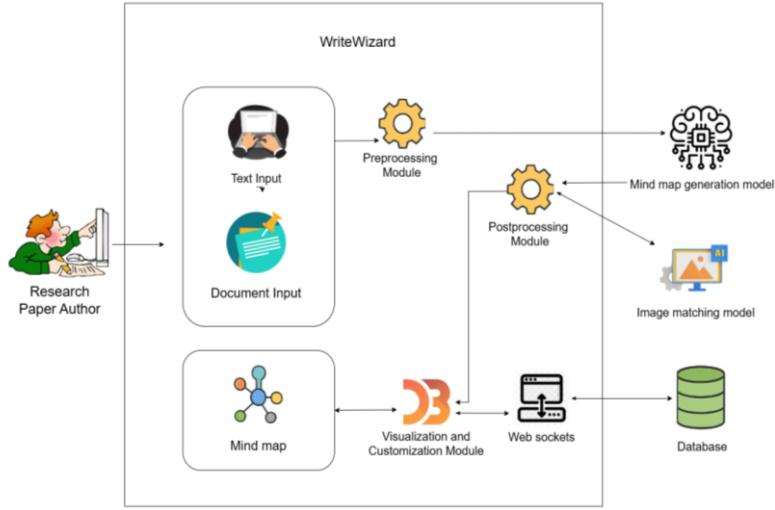


Figure 7: System Diagram

Design details:

The architectural design is organized into a three-tier structure.

The Presentation Layer is responsible for delivering an intuitive and interactive user interface. It enables users to log in, input text in the form of selected snippets or full documents and view the generated mind maps in real time. This layer includes interactive controls that allow users to edit nodes, rearrange layouts, and adjust visual styles. It is designed to hide underlying complexities, ensuring that users interact only with a polished and responsive interface.

The Business Logic Layer forms the core of the system's functionality. This layer processes the input text using a fine-tuned mind map generation model to convert the content into a structured JSON format that represents hierarchical relationships among concepts. It also manages user customization actions such as node editing, layout adjustments, and style modifications through dedicated controllers. In addition, it coordinates export operations by processing user preferences for formats such as PNG, SVG, or JSON and triggers the corresponding workflows to generate the desired output.

The Data Access Layer manages persistent storage and ensures data integrity across sessions. It handles user session management and securely stores generated mind maps

in the database, providing a reliable mechanism for saving and retrieving mind maps. This layer supports real-time collaboration by synchronizing updates across multiple users, ensuring that modifications are consistently reflected without data loss. Furthermore, it interfaces with external data sources as required while keeping these operations transparent to the user interface.

Flow of mind map generation:

The process begins when the user submits text input through the Presentation Layer. The input is transmitted to the Business Logic Layer, where the Mind Map Generation Module invokes the pre-trained model to produce a JSON representation of the mind map. The JSON output is then sent to the Visualization Module, which renders the mind map interactively using D3.js. At this point, the user can customize the mind map, export it in various formats, or share it with collaborators in real time. Figure 8 illustrates the operational flow from text submission to mind map visualization and subsequent user interactions.

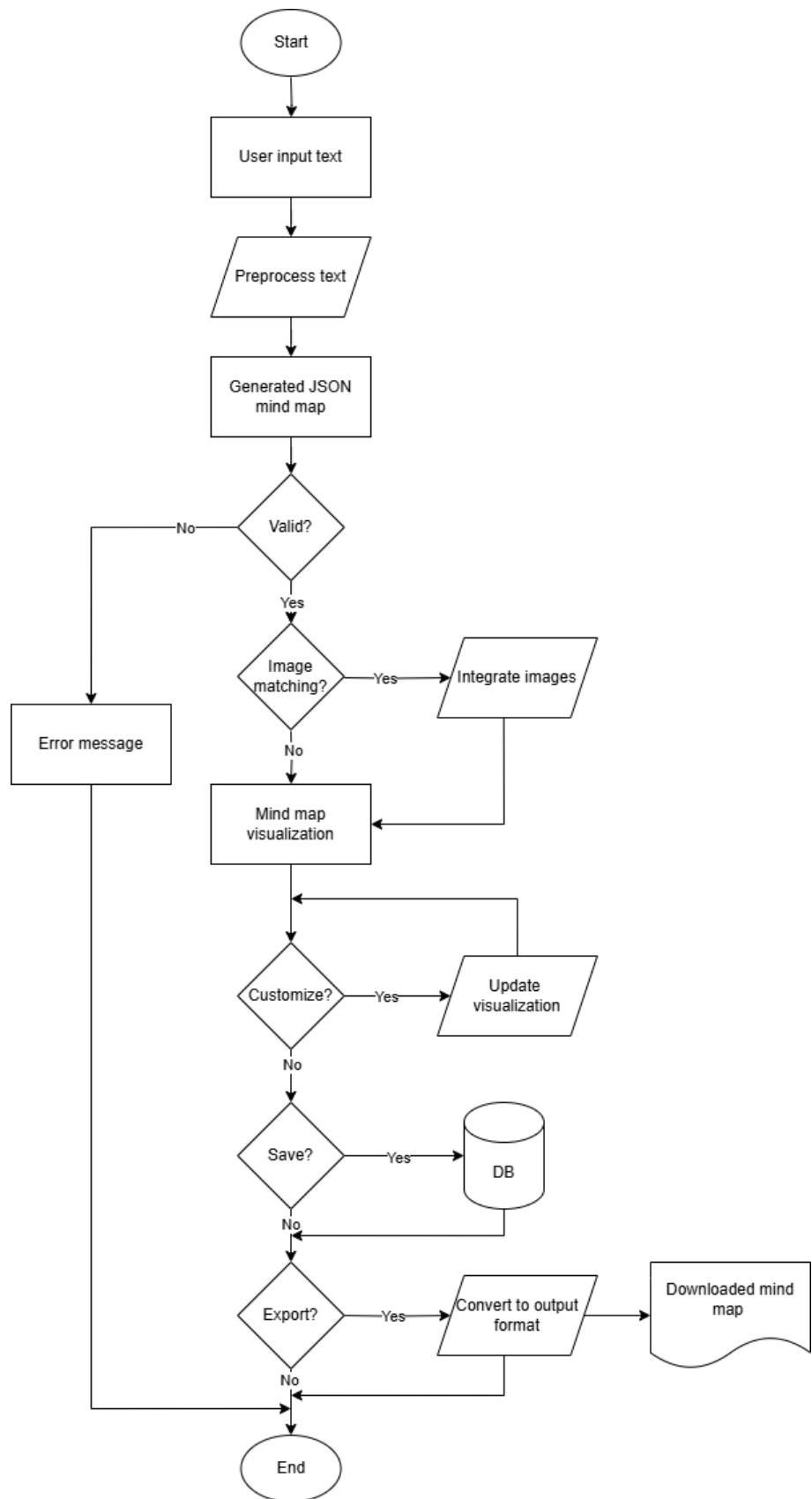


Figure 8: Flow of Automated Mind Map Generation

2.1.4 Implementation

The Implementation phase of this research represented the transition from architectural planning to practical realization of the automated mind map generation component. This phase focused on assembling the individual modules into a cohesive and functional system capable of processing user input, generating structured mind maps, and delivering an interactive visualization interface. Each feature from model integration to user interface development was implemented with emphasis on modularity, usability, and real-time responsiveness.

Task breakdown and project management:

Microsoft Planner, a versatile project management tool, was harnessed to meticulously delineate the project into well-defined subtasks and milestones. This structured approach enabled the team to maintain a clear and organized roadmap throughout the development process. Each task was scheduled and assigned with consideration of team roles and timelines, promoting efficient sprint-based collaboration and timely completion of deliverables. Figure 9 illustrates the Microsoft Planner board used during the implementation phase.

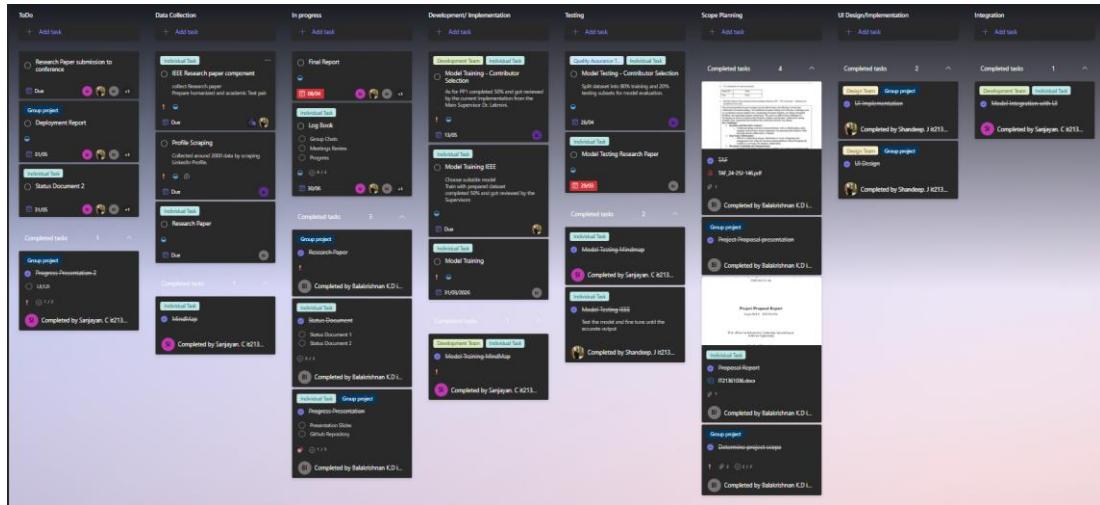


Figure 9: MS Planner Board

To complement task planning, a Work Breakdown Structure (WBS) was developed to provide a hierarchical representation of the entire project lifecycle, from initial planning through to final deployment and maintenance. This structure offered a clear overview of task dependencies and deliverables across each phase, enabling systematic progress tracking and effective coordination. Figure 10 presents the WBS designed for this project's implementation phase.

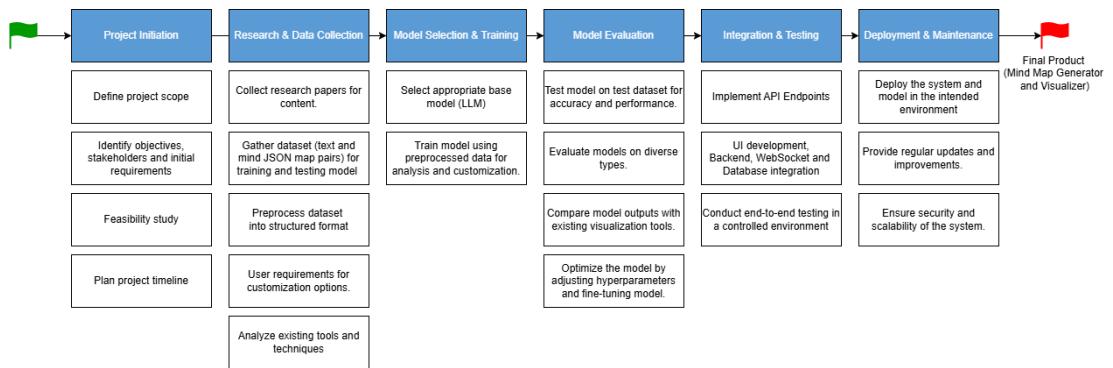


Figure 10: Work Breakdown Structure

Development environment and tools:

The core implementation was carried out using Visual Studio Code as the primary integrated development environment. This setup supported modular development across both frontend and backend components. Python was used on the backend with FastAPI for serving the mind map generation model and managing API endpoints, while MongoDB ensured persistent storage of user sessions and generated mind maps.

On the front end, React.js was employed to create a responsive interface, and D3.js enabled the rendering of JSON mind maps into interactive visual structures.

WebSockets supported real-time collaboration, while Axios handled API communication.

For model integration, Hugging Face’s transformers and datasets, along with PEFT and bitsandbytes, were used to streamline model fine-tuning and inference. Together, these tools provided a flexible and scalable foundation for building a real-time, collaborative mind map generation platform.

2.1.4.1 Dataset creation for mind map generation

The initial step in preparing for model fine-tuning was to identify an appropriate dataset capable of representing the transformation of academic content into mind map structures. However, despite an extensive review of publicly available repositories, no existing datasets were found that fulfilled the specific requirement of paired document text and structured mind map representations. This led to the decision to manually construct a custom dataset tailored to the objectives of this research.

Before finalizing the format for data preparation, multiple experimental approaches were evaluated. Early attempts included converting academic content into keyword-weight pairs and keyword hierarchies. In one method, text was broken down into sets of weighted keywords to reflect their importance or frequency. Another variation attempted to extract key phrase pairs to build relational links that would later map into a visual structure. However, these strategies proved inadequate in capturing the nuanced hierarchical and semantic relationships necessary for an effective and accurate mind map representation.

As a result, a more robust strategy was adopted: curating direct text-to-JSON mind map pairs. This approach involved manually annotating unstructured academic text with corresponding hierarchical JSON structures. Each JSON output followed a consistent schema, maintaining a single-root tree with nested child nodes that mirrored the logical organization of the content. This structure enabled the model to learn how to generate visualizable trees that accurately reflect the parent-child and sibling relationships embedded in educational texts.

The final dataset consists of 500 manually created input-output pairs. Most of the text samples were extracted from academic articles in the domain of machine learning, with additional examples drawn from general scientific and educational topics to provide a degree of diversity. Each entry was carefully crafted to maintain semantic integrity, ensuring that the JSON output accurately reflected the core ideas and their relationships within the original text.

This dataset serves as the foundation for fine-tuning the model and played a critical role in shaping its ability to produce structured, hierarchical mind maps directly from academic content. Figure 11 illustrates an example from the created dataset, showing how a passage of text is transformed into a structured JSON mind map representation.

Text	Mindmap
Animals are diverse and fascinating creatures. Tremendous amounts of data are generated by them. Big data can be defined in terms of 6Vs. IOT is today one of the main applications of big data. Big data has revolutionized the healthcare industry. Due to the existence of day-to-day technology, Big data has also made it possible to transform the transportation industry. Huge volumes of data are created by smart grids and IoT data. The authors referenced deep learning techniques and kernel-based learning. Another method called complex tasks. Think about a situation where we have a large amount of raw data, Agriculture plays a vital role in the economy. Machine learning algorithms are categorized into various categories. Number of researchers and scientist applying machine learning in agriculture. Support vector machine is a supervised learning algorithm. Artificial neural network (ANN) is neural network. K-nearest neighbor is similarity based learning. The lung is a key organ in human bodies. Random forest is an ensemble method for classification. The 'Naive Bayes' algorithm [3] is a supervised learning method. The dataset was obtained via Kaggle. Figure 4 outlines the steps for predicting diseases using machine learning. Malware remains a significant threat in the advancement of malware. This research compares different machine learning techniques. In times past, the malware was detected using traditional techniques. Shabtai et al. [19], by presenting various machine learning algorithms, detected malware. Mohit et al. [21] compare the categorization of malware. Feature Extraction & Pre-Processing. Machine learning is a branch of artificial intelligence.	{"mindmap": [{"name": "Animals", "subnodes": [{"name": "Diversity", "subnodes": []}], "parent": null}, {"name": "Big Data", "subnodes": [{"name": "Definition", "subnodes": []}, {"name": "6Vs", "subnodes": []}, {"name": "IOT", "subnodes": [{"name": "IoT Applications", "subnodes": []}], "parent": null}, {"name": "Healthcare", "subnodes": [{"name": "Revolutionizing Healthcare", "subnodes": []}], "parent": null}, {"name": "Social Media", "subnodes": [{"name": "Platforms", "subnodes": []}], "parent": null}, {"name": "Transportation", "subnodes": [{"name": "Transformation", "subnodes": []}], "parent": null}, {"name": "Smart Grids and IoT Data", "subnodes": [{"name": "Data Generation", "subnodes": []}], "parent": null}, {"name": "Deep Learning Techniques", "subnodes": [{"name": "Hot Learning Process", "subnodes": []}], "parent": null}, {"name": "Kernel-Based Learning", "subnodes": [{"name": "Engineering Applications", "subnodes": []}], "parent": null}, {"name": "Complex Tasks", "subnodes": [{"name": "Large Amount of Raw Data", "subnodes": []}], "parent": null}, {"name": "Agriculture", "subnodes": [{"name": "Serious Issues", "subnodes": []}], "parent": null}, {"name": "Machine Learning Algorithms", "subnodes": [{"name": "Categories", "subnodes": []}], "parent": null}, {"name": "Machine Learning in Agriculture", "subnodes": [{"name": "Computing Technology", "subnodes": []}], "parent": null}, {"name": "Support Vector Machine (SVM)", "subnodes": [{"name": "Supervised Machine Learning", "subnodes": []}], "parent": null}, {"name": "Artificial Neural Network (ANN)", "subnodes": [{"name": "Supervised Learning", "subnodes": []}], "parent": null}, {"name": "K-Nearest Neighbor (K-NN)", "subnodes": [{"name": "Similarity Based Learning", "subnodes": []}], "parent": null}, {"name": "Lung", "subnodes": [{"name": "Key Organ", "subnodes": []}], "parent": null}, {"name": "Random Forest", "subnodes": [{"name": "Ensemble Method for Classification", "subnodes": []}], "parent": null}, {"name": "Naive Bayes Algorithm", "subnodes": [{"name": "Supervised Learning", "subnodes": []}], "parent": null}, {"name": "K-Nearest Neighbor (KNN)", "subnodes": [{"name": "Supervised Learning", "subnodes": []}], "parent": null}, {"name": "Dataset", "subnodes": [{"name": "Source", "subnodes": []}], "parent": null}, {"name": "Predicting Diseases Using Machine Learning", "subnodes": [{"name": "Kaggle", "subnodes": []}], "parent": null}, {"name": "Malware", "subnodes": [{"name": "Significant Threat", "subnodes": []}], "parent": null}, {"name": "Advancement of Malware", "subnodes": [{"name": "Increased Internet Usage", "subnodes": []}], "parent": null}, {"name": "Machine Learning", "subnodes": [{"name": "Tasks", "subnodes": []}], "parent": null}, {"name": "Malware Detection Techniques", "subnodes": [{"name": "Traditional Techniques", "subnodes": []}], "parent": null}, {"name": "Machine Learning Algorithms", "subnodes": [{"name": "Malware Detection", "subnodes": []}], "parent": null}, {"name": "Traditional Machine Learning", "subnodes": [{"name": "Feature Extraction & Pre-Processing", "subnodes": []}], "parent": null}, {"name": "Feature Extraction & Pre-Processing", "subnodes": [{"name": "Feature Extraction", "subnodes": []}], "parent": null}, {"name": "Artificial Intelligence (AI) and Computer Science", "subnodes": [{"name": "Machine Learning", "subnodes": []}], "parent": null}], "parent": null}

Figure 11: Sample Dataset Collection for Mind Map Generation

In addition to the core dataset for text-to-mind map generation, an auxiliary dataset was also created to support future enhancements such as generating simplified or extended versions of mind maps. This dataset pairs academic text with its corresponding summarized and extended formats, enabling the possibility of transforming the same content into varying levels of detail while preserving its original context. This will serve as a foundation for generating adaptive mind maps that can be tailored to different comprehension levels. Figure 12 presents a sample entry from this dataset.

Text	Summarized Text	Extended Text
Animals are diverse and fascinating.	Animals exist in every environment and fall into groups.	Animals are remarkably diverse and endlessly fascinating, inhabiting Tremendous amounts of data are generated every day and accessed.
Tremendous amounts of data are generated every day and accessed.	Big data goes beyond what traditional tools can handle.	Big data can be defined in terms of 6 'V's: volume, variety, velocity.
Big data can be defined in terms of 6 'V's: volume, variety, velocity.	Big data is defined by six 'V's: volume, variety, velocity.	Big data can be comprehensively defined by six key dimensions, known as the '6 V's: volume, variety, velocity, value, veracity, and variability.
IoT is today one of the main applications of Big Data.	IoT is a major Big Data application that leverages machine learning and data analysis.	The Internet of Things (IoT) is now one of the most prominent applications of Big Data, transforming various industries through the collection and analysis of data from connected devices.
Big data has revolutionized the healthcare sector.	Big Data transforms healthcare by enabling real-time analysis and personalized treatment plans.	Big data has revolutionized the healthcare sector globally by fundamentally changing the way we approach diagnosis, treatment, and patient care.
Due to the existence of day-to-day technologies like social media.	Social media platforms like Facebook and Twitter, for example.	With the increasing prevalence of technologies and mobile phones, Big data has also made it possible to analyze transportation data.
Big data has also made it possible to analyze transportation data.	Big Data is revolutionizing transportation by analyzing traffic patterns and predicting travel times.	Big data has significantly transformed the transportation industry in numerous ways, from optimizing routes to improving public transit systems.

Figure 12: Sample Dataset Collection for Summarized and Extended Text

2.1.4.2 Preprocessing for model input

To ensure compatibility with the fine-tuning pipeline and improve model alignment for instruction-following behavior, a consistent and structured preprocessing approach was applied to the dataset prior to training. The preprocessing process began with basic text normalization procedures to maintain consistency across all input samples. This included removing non-standard characters, handling whitespace uniformly, and standardizing punctuation. Tokenization was performed using model-compatible tokenizers to prepare the text for input embedding, ensuring that the document structure remained intact throughout the transformation process.

To further enhance input clarity and coherence, sentence segmentation and basic coreference resolution were applied. These steps helped preserve contextual continuity within passages and supported the model's ability to maintain logical flow during mind map generation. Importantly, the preprocessing was designed to be non-destructive, with no aggressive filtering or summarization applied that might alter the intended meaning or structure of the original content.

Following these preprocessing operations, each sample was formatted using an instruction-response style schema similar to the Alpaca format. The instruction segment served as the prompt to the model, followed by the corresponding academic passage as input and its structured JSON mind map as the target response. This formatting strategy enabled seamless integration into instruction-tuned language models. Figure 13 illustrates an example of a preprocessed input-output pair in Alpaca-style format, showing how the prompt, academic text, and mind map structure were organized for fine-tuning purposes.

```
Formatted Prompt Output:
--- Prompt 1 ---
Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request. #3 Instruction: Convert the following text into a structured JSON mind map with parent node and logical nested subnodes: #3 Input: Trillions amounts of data are generated daily, and accessed all over the world. Earlier, big data was defined as data that could not be efficiently handled by traditional methods of its time. Today, the term is not limited only to volume. Below is organized, unstructured, and semi-organized types of information, big data is referred to as data that exceeds the capacity of common data set processing tools to store, capture, manage, share, and analyze. Despite textual data, big data has spread in the form of multimedia data like videos, audios, and images on different platforms like the Internet of Things (IoT), social media sites, machine-to-machine communication, sensor networks, and so on. Deriving appropriate information patterns from the massive data is quite challenging. For example, 8 Terabytes of data are daily generated by Twitter, thus processing 80 million tweets per day, and 2.5 Petabytes of data per year are generated by Walmart. #3 Response: {"mindmap": [{"name": "Big Data", "subnodes": [{"name": "Definition", "subnodes": [{"name": "Data exceeding traditional processing capacity", "subnodes": []}], "subnodes": [{"name": "Includes", "subnodes": []}], "subnodes": [{"name": "Volume", "subnodes": []}], "subnodes": [{"name": "Variety", "subnodes": [{"name": "Organized", "subnodes": []}], "subnodes": [{"name": "Unstructured", "subnodes": []}], "subnodes": [{"name": "Semi-organized", "subnodes": []}]}, {"name": "Types of Data", "subnodes": [{"name": "Textual Data", "subnodes": []}], "subnodes": [{"name": "Multimedia Data", "subnodes": [{"name": "Videos", "subnodes": []}, {"name": "Audios", "subnodes": []}, {"name": "Images", "subnodes": []}]}], "subnodes": [{"name": "Sources of Big Data", "subnodes": [{"name": "Internet of Things (IoT)", "subnodes": []}, {"name": "Social Media", "subnodes": []}, {"name": "Machine-to-Machine Communication", "subnodes": []}, {"name": "Sensor Networks", "subnodes": []}], "subnodes": [{"name": "Examples", "subnodes": [{"name": "Twitter", "subnodes": []}, {"name": "Walmart", "subnodes": []}]}]}]}], "subnodes": [{"name": "Deriving appropriate information patterns", "subnodes": []}]}}

--- Prompt 2 ---
Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request. #3 Instruction: Convert the following text into a structured JSON mind map with parent node and logical nested subnodes: #3 Input: Big data can be defined in terms of 6 Vs: volume, variety, velocity, veracity, value, and variability. Volume refers to the size or amount of the information, depicting the exponential growth of data in terms of Petabytes, Exabytes, or Terabytes. Variety refers to the various structures or varieties of information, such as organized data, semi-organized data, or unstructured data. Velocity refers to the speed or frequency at which information is created, produced, stored, analyzed, and accessed. Veracity refers to the reliability of data in terms of accuracy and quality, as not all gathered data is reliable. Value is an important factor because big data only gains significance if it is made valuable by its use in different organizations on different platforms. Variability refers to the constant change in the meaning of something. For example, if an ice cream parlor offers 10 different flavors of ice cream, but you get the same flavor with a different taste each day, that is variability. The same applies to data; if its significance changes continually, it can have a massive impact on your data homogenization. #3 Response: {"mindmap": [{"name": "Big Data", "subnodes": [{"name": "6 V's", "subnodes": [{"name": "Volume", "subnodes": [{"name": "Size of information", "subnodes": []}], "subnodes": [{"name": "Exponential growth", "subnodes": []}], "subnodes": [{"name": "Units", "subnodes": []}], "subnodes": [{"name": "Petabytes", "subnodes": []}, {"name": "Exabytes", "subnodes": []}, {"name": "Terabytes", "subnodes": []}]}], "subnodes": [{"name": "Variety", "subnodes": [{"name": "Varieties of Information", "subnodes": [{"name": "Organized Data", "subnodes": []}, {"name": "Semi-Organized Data", "subnodes": []}, {"name": "Unstructured Data", "subnodes": []}]}], "subnodes": [{"name": "Velocity", "subnodes": [{"name": "Speed of information", "subnodes": []}], "subnodes": [{"name": "Veracity", "subnodes": [{"name": "Reliability of Data", "subnodes": []}], "subnodes": [{"name": "Accuracy", "subnodes": []}, {"name": "Quality", "subnodes": []}], "subnodes": [{"name": "Value", "subnodes": [{"name": "Data Significance", "subnodes": []}], "subnodes": [{"name": "Variability", "subnodes": [{"name": "Constant Change in Meaning", "subnodes": []}]}]}]}]}]}]
```

Figure 13: Instruction-Based Prompt Formatting in Alpaca Style

Each processed input maintained a clear alignment with its corresponding JSON structure, ensuring that the hierarchical relationships defined in the output could be faithfully learned from the sequence of sentences and subtopics in the input. This consistency was crucial to enabling the model to associate textual structures with logical, nested outputs in the form of tree-like JSON representations.

2.1.4.3 Model selection and justification

The selection of appropriate model architecture was a pivotal step in this research, as it directly influenced the system's ability to generate accurate and semantically coherent mind maps. A series of preliminary experiments were conducted using state-of-the-art language models to evaluate their capability in transforming academic text into structured, hierarchical JSON mind maps.

Initial trials included models such as T5, SciBERT, and Phi-2, which offered strong performance in general natural language understanding and summarization tasks. However, these models demonstrated limitations in producing deeply nested outputs and maintaining hierarchical consistency, particularly in cases involving complex academic discourse. Their architectural constraints and token limitations further affected their ability to generate extensive, multi-level structures suitable for mind maps.

To address these challenges, larger and more expressive models were explored. LLaMA 2 and Mistral 7B v0.3 emerged as the most promising candidates. Comparative evaluation focused on criteria such as vocabulary range, instruction-following alignment, and JSON structural integrity. Mistral 7B v0.3 was ultimately selected for its extended vocabulary capacity of 32,768 tokens, which proved advantageous for handling long-form academic input while preserving context. Its transformer-based architecture demonstrated superior proficiency in capturing logical relationships and expressing them in well-organized JSON tree formats.

This model selection process was conducted prior to fine-tuning, using few-shot prompting and qualitative assessments to evaluate the structural reliability of generated mind maps. Based on this analysis, Mistral 7B v0.3 was identified as the optimal foundation for downstream training and integration.

2.1.4.4 Model fine-tuning for mind map generation

To specialize in the selected Mistral 7B v0.3 model for generating structured JSON mind maps from academic content, Supervised Instruction Fine-Tuning (SFT) was employed using a LoRA-based Parameter-Efficient Fine-Tuning (PEFT) strategy. The fine-tuning process was executed using the Unsloth framework, a performance-optimized wrapper over Hugging Face Transformers, enabling reduced memory consumption and accelerated training times, which is ideal for free-tier GPU environments.

All training was conducted on Google Colab (Free Tier) using a Tesla T4 GPU. Due to limited compute availability, the setup used 4-bit quantization, gradient checkpointing, and mixed precision training (automatically switching between bfloat16 and float16) to reduce VRAM usage while maintaining output quality.

Fine-tuning strategy and frameworks

The training pipeline was built on the Hugging Face Transformers, peft, and trl libraries. The SFTTrainer class from trl allowed integration of instruction-response training data and streamlined the supervised fine-tuning process. The base model, Mistral 7B v0.3, was loaded with support for 4-bit quantization (load_in_4bit=True) to conserve memory, while LoRA was used to update only selected transformer parameters rather than the full model, making the process feasible even on constrained hardware. To support long academic input passages and ensure structural alignment in outputs, a maximum sequence length (max_seq_length) of 4096 tokens was set, enabling the model to process complex, multi-level content effectively within a single pass.

LoRA configuration and target modules

To optimize adaptation while maintaining computational efficiency, LoRA was applied to the following key transformer modules:

- Target Modules: q_proj, k_proj, v_proj, o_proj, gate_proj, up_proj, down_proj
- LoRA Rank (r): 16 – balancing parameter reduction and representational power

- LoRA Alpha: 16 – maintaining learning scale
- Dropout: 0 – avoiding instability on low-resource hardware
- Gradient Checkpointing: Enabled via "unsloth" to reduce peak memory during training

This setup allowed effective fine-tuning while keeping GPU usage minimal. The careful selection of attention and MLP projection layers ensured that the model learned meaningful structural representations without requiring full weight updates. Figure 14 illustrates the model loading and LoRA configuration using Unsloth and Hugging Face Transformers.

```
max_seq_length = 4096
dtype = None
load_in_4bit = True

model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = "mistralai/Mistral-7B-v0.3",
    max_seq_length = max_seq_length,
    dtype = dtype,
    load_in_4bit = load_in_4bit,
)

model = FastLanguageModel.get_peft_model(
    model,
    r = 16,
    target_modules = ["q_proj", "k_proj", "v_proj", "o_proj",
                      "gate_proj", "up_proj", "down_proj"],
    lora_alpha = 16,
    lora_dropout = 0,
    bias = "none",
    use_gradient_checkpointing = "unsloth",
    random_state = 3407,
    use_rslora = False,
    loftq_config = None,
)
```

Figure 14: Model Loading and LoRA Configuration

Training configuration and parameters

A gradual fine-tuning strategy was employed, starting with conservative steps and scaling up based on observed GPU capacity and model stability. The final configuration used for training was:

- Batch Size: 2 (with 4 gradient accumulation steps → effective batch size = 8)
- Max Steps: 80 – used as a manual early-stopping mechanism to prevent overfitting and control runtime

- Warmup Steps: 5 – allowed stable learning rate ramp-up
- Learning Rate: 2e-4 – optimized for LoRA training and rapid convergence
- Weight Decay: 0.01 – regularization to mitigate overfitting
- Optimizer: 8-bit AdamW – reduced memory footprint
- Scheduler: Linear – combined with warmup for stable performance
- Logging: Every step – to monitor convergence in real time
- Precision: fp16 (fallback to bf16 when compatible)
- A fixed random seed (3407) was used across tokenizer and training to ensure deterministic behavior and reproducibility.

Figure 15 presents the training argument block and SFTTrainer configuration used to run the fine-tuning loop.

```

trainer = SFTTrainer(
    model = model,
    tokenizer = tokenizer,
    train_dataset = dataset,
    dataset_text_field = "text",
    max_seq_length = max_seq_length,
    dataset_num_proc = 2,
    packing = False,
    args = TrainingArguments(
        per_device_train_batch_size = 2,
        gradient_accumulation_steps = 4,
        warmup_steps = 5,
        max_steps = 80,
        learning_rate = 2e-4,
        fp16 = not is_bfloat16_supported(),
        bf16 = is_bfloat16_supported(),
        logging_steps = 1,
        optim = "adamw_8bit",
        weight_decay = 0.01,
        lr_scheduler_type = "linear",
        seed = 3407,
        output_dir = "outputs",
        report_to = "WandB",
    ),
)
trainer_stats = trainer.train()

```

Figure 15: Training Configuration and SFTTrainer Setup

Training progress and stability

The model exhibited consistent learning behavior throughout training. The loss graph showed a smooth, downward trajectory without significant spikes, suggesting that the

model effectively learned from the training samples without diverging. This can be attributed to the well-aligned dataset, modest training steps, and stable LoRA optimization.

A sample of the stepwise loss output from Unsloth is shown in Figure 16, indicating stable convergence over 80 steps across 4 epochs. The decreasing trend of training loss demonstrates the model's ability to generalize without signs of overfitting, even in a resource-constrained setup and the model preserved its generalization capabilities across samples from different academic domains. The effective use of LoRA and limited step training helped to retain foundational knowledge from the base model while aligning it toward the specialized task of hierarchical mind map generation.

[80/80 54:23, Epoch 3/4]			
Step	Training Loss		
1	1.083300	70	0.495100
2	1.112500	71	0.467200
3	1.013400	72	0.426100
4	1.067100	73	0.472200
5	0.916600	74	0.384000
6	1.004700	75	0.403100
7	1.066400	76	0.949400
8	0.901000	77	0.445200
9	0.964000	78	0.513000
10	0.946700	79	0.524800
		80	0.362900

Figure 16: Training Loss Log

After completion of training, the model and tokenizer were saved. This checkpoint allows future loading, evaluation, and integration into the backend inference pipeline. Figure 17 illustrates the final output section of the training notebook, showing the model and tokenizer save commands used at the end of the fine-tuning process. The fine-tuned model was now capable of transforming academic passages into structured mind maps with high semantic fidelity and was ready for backend integration and evaluation in real-world scenarios.

```
# Local saving
model.save_pretrained("lora_model")
tokenizer.save_pretrained("lora_model")

('lora_model/tokenizer_config.json',
 'lora_model/special_tokens_map.json',
 'lora_model/tokenizer.model',
 'lora_model/added_tokens.json',
 'lora_model/tokenizer.json')
```

Figure 17: Saving Model and Tokenizer after Fine-Tuning

2.1.4.5 Backend integration and API development

2.1.4.5.1 Backend integration for mind map generation using Python and FastAPI

To facilitate seamless communication between the user interface and the fine-tuned Mistral 7B v0.3 model, a modular microservice was developed using FastAPI. This service enabled stateless, on-demand mind map generation by exposing a set of lightweight RESTful endpoints. The design followed a decoupled microservice architecture to ensure scalability, efficient inference, and ease of integration into the broader document editing environment.

The microservice was implemented in Python using a carefully selected combination of frameworks and libraries that emphasized performance, validation, and model compatibility. FastAPI served as the backbone of the service, offering asynchronous capabilities and fast response handling. Quantized 4-bit inference was enabled through `load_in_4bit=True`, making it suitable for environments with limited hardware resources. Hugging Face's transformers and tokenizer utilities powered the inference mechanism, while Pydantic enabled schema validation and structure enforcement for incoming requests. Middleware configuration was managed through CORSMiddleware to allow frontend integration during development and testing.

Application Structure and Functionality

- The microservice was structured into three core logical components, each serving a distinct purpose in the processing pipeline:
- Preprocessing and Cleaning: Incoming academic text is initially passed through a preprocessing step that removes noise such as emojis, irregular whitespaces, and line breaks. Regex-based filters ensure the input is well-formatted for consistent downstream processing.
- Prompt Tokenization and Inference: Cleaned text is embedded into an Alpaca-style prompt that maintains consistency with the training schema. This formatted instruction is tokenized, checked for sequence length violations, and then passed to the fine-tuned Mistral model for inference. The tokenizer enforces length restrictions and

gracefully returns error responses if the input exceeds acceptable bounds. Upon successful inference, the raw output is returned as a structured string.

- Error Handling and Postprocessing: The model output is parsed into a JSON format. To address cases where minor structural inconsistencies arise (such as missing brackets or incomplete nesting), a custom repair function is invoked. This function uses a stack-based mechanism to correct bracket mismatches and ensure the output remains parseable. Once validated, the JSON is returned to the client.

Exposed API Endpoints

The microservice exposes three RESTful endpoints, each tailored for a specific mind map generation use case. All endpoints accept a POST request with a body defined by the shared `MindMapRequest` Pydantic schema:

- `/generate`

Accepts raw academic text and returns a structured JSON mind map. This is the core endpoint, used for converting unprocessed input into its visualizable tree representation.

- `/simplify`

First applies a simplification transformation to the input text to reduce complexity and verbosity. Then it processes the simplified content through the same generation pipeline. This is ideal for audiences that prefer an abstracted or high-level version of the content.

- `/extend`

Expands the original input by enriching it with additional context or detail and then produces a more elaborate mind map. This endpoint is suitable for deeper academic exploration or knowledge expansion.

Each endpoint internally invokes the `get_response()` function, which is responsible for generating a structured output string based on the model's response. This output is then passed to the `get_mindmap()` function, which handles parsing, validation, and formatting into the appropriate JSON structure.

Figure 18 presents the implemented FastAPI routes endpoints along with their core handler logic. Figure 19 illustrates the initialization logic of the FastAPI microservice,

including tokenizer setup, model loading using quantized 4-bit precision, and It shows how incoming requests are validated, preprocessed, passed through the fine-tuned Mistral model, and finally post-processed into structured JSON mind maps using robust error handling and output validation mechanisms.

```

@app.post("/generate")
async def generate_mindmap(request: MindMapRequest):
    """
    Returns a mind map for the provided text.
    """
    try:
        # Preprocess the input text
        cleaned_text = preprocess_text(request.content)
        mindmap_obj = get_mindmap(cleaned_text)
        return mindmap_obj
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.post("/simplify")
async def simplify_text(request: MindMapRequest):
    """
    First simplifies the text and then returns the mind map for the simplified text.
    """
    try:
        # Preprocess the input text
        cleaned_text = preprocess_text(request.content)
        # Generate the simplified version of the text
        simplified_text = get_response("Summarize and shorten the following text : ", cleaned_text)
        # Then generate a mind map from the simplified text.
        mindmap_obj = get_mindmap(simplified_text)
        return mindmap_obj
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.post("/extend")
async def extend_text(request: MindMapRequest):
    """
    First extends the text and then returns the mind map for the extended text.
    """
    try:
        # Preprocess the input text
        cleaned_text = preprocess_text(request.content)
        # Generate the extended version of the text.
        extended_text = get_response("Extend, expand, and enrich the following text by providing deeper detail, relevant examples, and additional context, while preserving its original meaning and style: ", cleaned_text)
        # Then generate a mind map from the extended text.
        mindmap_obj = get_mindmap(extended_text)
        return mindmap_obj
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

# Fix event loop
nest_asyncio.apply()

# Run the server in a separate thread
def run_server():
    uvicorn.run(app, host="0.0.0.0", port=8000)

```

Figure 18: API Endpoint Implementations for Mind Map Generation

```

  app.py

import json
import importlib
from fastapi import FastAPI
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from fastapi.middleware.cors import CORSMiddleware
import torch
from transformers import PreTrainedTokenizerFast
import uvicorn
import threading
import nest_asyncio

# Load the model from Hugging Face
adapter_model_path = "Semjayan207/mistral-7b-v0.3-finetuned-for-mindmap-lora"
max_seq_length = 2048
dtype = None
load_in_4bit = True

model, tokenizer = FastLanguageModel.from_pretrained(
    model_name=adapter_model_path,
    max_seq_length=max_seq_length,
    dtype=dtype,
    load_in_4bit=load_in_4bit,
)

FastLanguageModel.for_inference(model)

# Alpaca-style prompt template
alpaca_prompt = """Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.

### Instruction:
{}

### Input:
{}

### Response:
{}"""

app = FastAPI()

# Configure CORS settings
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Defining a request model for input text (re-used for all endpoints)
class MindmapRequest(BaseModel):
    content: str = ""

    def remove_emoji(self, text: str) -> str:
        """Remove emoji from the text."""
        emoji_pattern = re.compile(
            "\ud83c\udc00-\ud83c\udcff" # emotions
            "\ud83d\udc00-\ud83d\udcff" # symbols & pictographs
            "\ud83e\udc00-\ud83e\udcff" # transport & map symbols
            "\ud83f\udc00-\ud83f\udcff" # flags (ios)
            "|", flags=re.UNICODE)
        return emoji_pattern.sub(r'', text)

    def preprocess_text(self, text: str) -> str:
        """Cleans the input text by:
        - Removing emojis/icons.
        - Replacing line breaks and multiple spaces with a single space.
        - Stripping leading/trailing whitespace.
        """
        text = self.remove_emoji(text)
        # Reduces unnecessary whitespace (including line breaks) with a single space
        text = re.sub(r"\s+", ' ', text)
        return text.strip()

    def tokenize_prompt(self, instruction: str, content: str) -> dict:
        """Tokenizes the prompt without truncation.
        If the total token count exceeds max_seq_length, raises an HTTPException.
        """
        prompt = alpaca_prompt.format(instruction, content, "")
        inputs = tokenizer(prompt, return_tensors="pt", truncation=False)
        if inputs.input_ids.shape[1] > max_seq_length:
            raise HTTPException(
                status_code=400,
                detail="Input text too long, got {inputs.input_ids.shape[1]} tokens."
            )
        return inputs.to("cuda")

    def get_response(self, instruction: str, content: str) -> str:
        """Generates a response from the model using the given instruction and content.
        Returns only the part after '### Response:'.
        """
        inputs = self.tokenize_prompt(instruction, content)
        outputs = model.generate(**inputs, max_new_tokens=2048, use_cache=True)
        full_response = tokenizer.batch_decode(outputs, skip_special_tokens=True)[0]
        if "### Response:" in full_response:
            response_text = full_response.split("### Response:")[1].strip()
        else:
            response_text = full_response.strip()
        return response_text

    def fix_incomplete_json(self, json_str: str) -> str:
        """Attempts to fix incomplete JSON by scanning through the string using a stack to track unmatched opening braces and brackets. It ignores brackets that occur inside string literals.
        Note: This is a heuristic and may not handle all edge cases.
        """
        stack = []
        in_string = False
        escape = False

        # Iterate through each character in the JSON string
        for char in json_str:
            if char == '"' and not escape:
                in_string = not in_string
            elif in_string:
                if char == '\\':
                    escape = True
                else:
                    escape = False
                    continue

            if char in ['{', '(', '[']:
                stack.append(char)
            elif char in ['}', ')', ']']:
                if stack:
                    last = stack[-1]
                    # Check if current closing matches the last opening
                    if (last == '{' and char == '}') or (last == '[' and char == ']'):
                        stack.pop()
                    else:
                        # If there's a mismatch, you might want to handle it differently.
                        # For now, we simply ignore and continue.
                        pass
                else:
                    # Define the mapping for corresponding closing characters.
                    closing_map = {'{': '}', '[': ']'}

                    # Append the required closing tokens in the reverse order of openings.
                    json_str += closing_map[stack.pop()]

            # Define the mapping for corresponding closing characters.
            closing_map = {'{': '}', '[': ']'}

            # Append the required closing tokens in the reverse order of openings.
            json_str += closing_map[stack.pop()]

        return json_str

```

Figure 19: Fine-Tuned Model and Tokenizer Initialization with Preprocessing and Postprocessing Logic

2.1.4.5.2 Backend integration for semantic image matching using Python and FastAPI

To enhance the contextual richness of the mind maps generated, a semantic image matching mechanism was developed and integrated into the backend pipeline. This feature identifies relevant images based on node-level semantics and matches them with the appropriate segments of the mind map structure.

The image matching service was implemented as a FastAPI-based microservice, leveraging transformer-based models to understand and align image descriptions with mind map node content. Specifically, the sentence-transformers/all-mpnet-base-v2 model was used to encode semantic similarities between node texts and image descriptions, while a roberta-large-mnli pipeline performed Natural Language Inference (NLI) to validate logical consistency.

The matching algorithm operates in two stages:

- **Cosine Similarity Scoring:** The model computes vector representations for node texts and image descriptions, followed by cosine similarity comparisons to determine semantic closeness.
- **NLI Validation:** If the similarity exceeds a predefined threshold, an NLI score is computed using a zero-shot classification pipeline to confirm that the image description logically aligns with the node content.

An image is accepted as a match only if the NLI score surpasses a configurable threshold (set to 0.7). Further, images with high similarity (above 0.8) are flagged as "high-priority matches" for more confident integration into the mind map. This dual-check approach ensures both semantic alignment and logical consistency between images and mind map nodes.

Figure 20 displays a snapshot of the Python implementation used for conditional image-node alignment, including the logic for similarity scoring, NLI threshold validation, and the API endpoint responsible for returning matched image-node pairs. This semantic matching framework not only improves the visual value of the mind

maps but also maintains contextual fidelity, allowing users to better grasp key ideas through aligned visuals.



```

from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
from typing import List
from sentence_transformers import SentenceTransformer, util
from transformers import pipeline
import torch

# Initialize FastAPI app
app = FastAPI()

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Define request models
class ImagePair(BaseModel):
    image_url: str
    image_description: str

class MatchRequest(BaseModel):
    image_pairs: List[ImagePair]
    node_texts: List[str]

# Load models once at startup for performance
sim_model = SentenceTransformer("sentence-transformers/all-mnli-base-v2")
nli_pipeline = pipeline("zero-shot-classification", model="roberta-large-mnli")

def match_image_to_node_conditional(image_description, image_url, node_texts, sim_threshold=0.8, nli_threshold=0.7):
    """
    Matches an image (via its description) to the best candidate node text.

    Rules:
    - Reject if NLI score < nli_threshold.
    - If NLI score >= nli_threshold and cosine similarity >= sim_threshold, mark as high-priority.

    Returns a tuple: (candidate_node, image_url, match_status)
    """
    # Ensure node_texts is not empty
    if not node_texts:
        raise ValueError("No node texts provided. Please include at least one node in your request.")

    # Compute embedding for the image description.
    desc_embedding = sim_model.encode(image_description, convert_to_tensor=True)

    # Compute embeddings for all node texts.
    node_embeddings = sim_model.encode(node_texts, convert_to_tensor=True)

    # If node_embeddings is empty (shouldn't happen after the check), return error.
    if node_embeddings.shape[0] == 0:
        raise ValueError("Node embeddings could not be computed. The node_texts list appears to be empty.")

    # Compute cosine similarities.
    similarities = util.pytorch_cos_sim(desc_embedding, node_embeddings)  # shape [1, N]
    best_sim, best_idx = torch.max(similarities, dim=1)
    candidate_node = node_texts[best_idx.item()]
    cosine_score = best_sim.item()

    # Construct hypothesis and run NLI.
    hypothesis = f"This image is about {candidate_node}."
    nli_result = nli_pipeline(image_description, candidate_labels=[candidate_node])
    nli_score = nli_result["scores"][0]

    print(f"Image: '{image_description}'")
    print(f" Candidate node: '{candidate_node}'")
    print(f" Cosine similarity: {cosine_score:.3f}, NLI score: {nli_score:.3f}")

    # Apply conditional logic.
    if nli_score < nli_threshold:
        return None, None, "Rejected: Low NLI score"
    else:
        if cosine_score >= sim_threshold:
            return candidate_node, image_url, "High-priority match"
        else:
            return candidate_node, image_url, "Accepted match (lower cosine similarity)"

@app.post("/match")
def match_images(request: MatchRequest):
    matched_results = []
    for pair in request.image_pairs:
        candidate, url, status = match_image_to_node_conditional(pair.image_description,
                                                                pair.image_url,
                                                                request.node_texts,
                                                                sim_threshold=0.8,
                                                                nli_threshold=0.7)

        if candidate:
            matched_results.append({"node": candidate, "image_url": url, "status": status})

    return {"matched_pairs": matched_results}

```

Figure 20: Conditional Image-Node Matching and API Integration

2.1.4.5.3 Backend integration for real-time document editing and service orchestration using Node.js

WriteWizard is a MERN stack-based collaborative document editing platform that supports academic writing and visual content structuring. The Express.js backend plays an essential role in enabling real-time communication, managing user sessions, and interacting with the database. It is responsible for handling the server-side logic specific to mind map collaboration and for ensuring smooth integration with related services.

This backend enables multiple users to simultaneously view and interact with shared mind map structures in a synchronized and consistent manner. All WebSocket-based communication, session tracking, and data persistence related to mind maps are handled through this layer.

Real-time collaboration through WebSocket

WebSocket communication is established using Socket.IO to allow multiple users to collaborate on the same mind map session in real time. Each session is identified using a unique mindmapId, and users joining a session are added to a room associated with that identifier. The backend maintains an in-memory object to track connected users, storing socket IDs and activity status for synchronization.

When a user joins a mind map, the join-mindmap event registers them in the room. As users interact, events such as node-selected and update-mindmap are triggered, broadcasting node selections or structural changes to all other users in the same session. When a user disconnects or manually leaves a session, the backend cleans up the session and notifies remaining users accordingly.

Figures 21 and 22 illustrate the backend WebSocket logic implemented using Socket.IO. These figures together showcase how user sessions are managed, how node selections and structural updates are broadcast in real time, and how disconnections and user exits are handled gracefully to maintain collaboration state.



```

const app = express();
connectDB();
app.use(cors());
app.use(express.json({ limit: "10mb" }));

app.get("/", (req, res) => {
  res.json({ success: true, message: "API is running successfully!" });
});

const server = http.createServer(app);

const io = new Server(server, {
  cors: {
    origin: "*",
    methods: ["GET", "POST"],
  },
});

// In-memory structure for tracking connected users per pad
let mindmaps = {};

io.on("connection", (socket) => {
  console.log("✓ User connected:", socket.id);

  // 1. User joins a specific mindmap
  socket.on("join-mindmap", async ({ mindmapId, userId, userName }) => {
    socket.join(mindmapId);

    if (!mindmaps[mindmapId]) {
      mindmaps[mindmapId] = { users: {} };
    }
    // Set active flag to true on join.
    mindmaps[mindmapId].users[userId] = {
      userId,
      userName,
      socketId: socket.id,
      active: true,
    };

    // Broadcast the updated user list to all clients in this mindmap.
    io.to(mindmapId).emit(
      "mindmap-users",
      Object.values(mindmaps[mindmapId].users)
    );
  });

  socket.on("node-selected", ({ mindmapId, nodeId, userName }) => {
    // Broadcast to everyone else in that mindmap room
    socket.to(mindmapId).emit("node-selected", { nodeId, userName });
  });

  // 2. Update mindmap changes
  socket.on("update-mindmap", async ({ mindmapId, nodes, links, userId }) => {
    try {
      // Then broadcast the changes to everyone else
      io.to(mindmapId).emit("receive-mindmap-changes", {
        nodes,
        links,
        userId,
      });
    } catch (err) {
      console.error("✗ Error updating mindmap:", err);
    }
  });
});

```

Figure 21: WebSocket logic – User join, selection events, and structure updates

```

// 3. If user manually leaves the mindmap
socket.on("leave-mindmap", ({ mindmapId, userId }) => {
  console.log(`❌ User ${userId} left mindmap ${mindmapId}`);
  if (mindmaps[mindmapId] && mindmaps[mindmapId].users[userId]) {
    delete mindmaps[mindmapId].users[userId];
    io.to(mindmapId).emit(
      "mindmap-users",
      Object.values(mindmaps[mindmapId].users)
    );
  }
});

// 4. Handle user disconnect
socket.on("disconnect", () => {
  console.log("❌ User disconnected:", socket.id);

  for (const mindmapId in mindmaps) {
    if (!mindmaps[mindmapId]) continue;
    let userToRemove = null;
    let userData = null;

    // Use the actual key from your mindmaps users object.
    for (const uId in mindmaps[mindmapId].users) {
      if (mindmaps[mindmapId].users[uId].socketId === socket.id) {
        userToRemove = uId;
        userData = mindmaps[mindmapId].users[uId];
        break;
      }
    }

    if (userToRemove) {
      console.log(
        `✖️ Removing user: ${userToRemove} from mindmap ${mindmapId}`
      );
      delete mindmaps[mindmapId].users[userToRemove];

      // Emit update for the mindmap users list
      io.to(mindmapId).emit(
        "mindmap-users",
        Object.values(mindmaps[mindmapId].users)
      );
    }
  }
};

// Emit the "node-selected" event with null nodeId and the user's name
const userName = (userData && userData.username) || userToRemove;
io.to(mindmapId).emit("node-selected", { nodeId: null, userName });
}

});

app.use("/api/auth", authRoutes);
app.use("/api/mindmaps", mindmapRoutes);

// Use when hosted
const port = process.env.PORT || 4000;
server.listen(port, "0.0.0.0", () => {
  console.log(`✅ Server running on port ${port}`);
});

```

Figure 22: WebSocket logic – User exit handling and disconnection cleanup

Middleware for service integration

In addition to handling real-time communication, the backend acts as the central orchestrator that manages all service-level logic related to mind maps. It exposes a set of RESTful API endpoints using Express Router, allowing users to create, retrieve, update, and share mind maps. Each route is implemented with structured input validation and response messaging.

Key route functionalities include:

- POST /api/mindmaps allows new mind maps to be created from user sessions.
- GET /api/mindmaps/:id retrieves a specific mind map using its identifier.
- PUT /api/mindmaps/:id allows structural changes such as updated nodes or links.
- PUT /api/mindmaps/:id/addUser adds new users and assigns their roles.

Figures 23 and 24 present the backend implementation of RESTful APIs used for mind map management. These routes handle various operations including creation, retrieval, updating, and user assignment. The figures collectively demonstrate request validation, route handling, and MongoDB integration for storing structured mind map data.



```

● ● ● JS mindmapRoutes.js

const express = require("express");
const router = express.Router();
const Mindmap = require("../models/Mindmap");

// POST /api/mindmaps - Create a new mindmap
router.post("/", async (req, res) => {
  try {
    const { users, nodes, links, image, downloadDate } = req.body;

    if (!users || !Array.isArray(users) || users.length === 0) {
      return res
        .status(400)
        .json({ message: "At least one user is required." });
    }

    const newMindmap = new Mindmap({
      users,
      nodes,
      links,
      image,
      downloadDate: downloadDate ? new Date(downloadDate) : new Date(),
    });

    const savedMindmap = await newMindmap.save();
    res.status(201).json({
      message: "Mindmap saved successfully",
      mindmap: savedMindmap,
    });
  } catch (error) {
    console.error("Error saving mindmap:", error);
    res.status(500).json({
      message: "Error saving mindmap",
      error: error.message,
    });
  }
});

// GET /api/mindmaps - Get all mindmaps
router.get("/", async (req, res) => {
  try {
    const mindmaps = await Mindmap.find();
    res.status(200).json(mindmaps);
  } catch (error) {
    console.error("Error fetching mindmaps:", error);
    res.status(500).json({
      message: "Error fetching mindmaps",
      error: error.message,
    });
  }
});

// GET /api/mindmaps/:id - Get a mindmap by its ID
router.get("/:id", async (req, res) => {
  try {
    const mindmap = await Mindmap.findById(req.params.id);
    if (!mindmap) {
      return res.status(404).json({ message: "Mindmap not found" });
    }
    res.status(200).json(mindmap);
  } catch (error) {
    console.error("Error fetching mindmap by ID:", error);
    res.status(500).json({
      message: "Error fetching mindmap",
      error: error.message,
    });
  }
});

// GET /api/mindmaps/user/:userId - Get all mindmaps for a specific user
router.get("/user/:userId", async (req, res) => {
  try {
    const mindmaps = await Mindmap.find({ "users.userId": req.params.userId });
    res.status(200).json(mindmaps);
  } catch (error) {
    console.error("Error fetching mindmaps by userId:", error);
    res.status(500).json({
      message: "Error fetching mindmaps",
      error: error.message,
    });
  }
});

```

Figure 23: REST API logic for mind map creation, and retrieval

```

// PUT /api/mindmaps/:id - Update an existing mindmap by its ID
router.put("/:id", async (req, res) => {
  try {
    const { nodes, links, image } = req.body;

    if (!nodes || !links) {
      return res
        .status(400)
        .json({ message: "Nodes and links data are required." });
    }

    const updateData = { nodes, links };
    if (image) {
      updateData.image = image;
    }

    const updatedMindmap = await Mindmap.findByIdAndUpdate(
      req.params.id,
      updateData,
      { new: true }
    );

    if (!updatedMindmap) {
      return res.status(404).json({ message: "Mindmap not found" });
    }

    res.status(200).json({
      message: "Mindmap updated successfully",
      mindmap: updatedMindmap,
    });
  } catch (error) {
    console.error("Error updating mindmap:", error);
    res.status(500).json({
      message: "Error updating mindmap",
      error: error.message,
    });
  }
});

// PUT /api/mindmaps/:id/addUser - Add a user to a mindmap
router.put("/:id/addUser", async (req, res) => {
  try {
    const { userId, role } = req.body;
    if (!userId) {
      return res.status(400).json({ message: "userId is required." });
    }

    const mindmap = await Mindmap.findById(req.params.id);
    if (!mindmap) {
      return res.status(404).json({ message: "Mindmap not found." });
    }

    const userExists = mindmap.users.some(
      (u) => u.userId.toString() === userId
    );
    if (userExists) {
      return res
        .status(400)
        .json({ message: "User already exists in this mindmap." });
    }

    mindmap.users.push({
      userId,
      role: role || "editor",
    });

    const updatedMindmap = await mindmap.save();
    res.status(200).json({
      message: "User added successfully",
      mindmap: updatedMindmap,
    });
  } catch (error) {
    console.error("Error adding user:", error);
    res.status(500).json({
      message: "Error adding user",
      error: error.message,
    });
  }
});

module.exports = router;

```

Figure 24: REST API logic for mind map update, and user assignment

Mind map data schema and storage

The backend uses Mongoose schema to store mind map data in MongoDB. The schema contains fields for node and link structures, image data, associated users, and download timestamps. Each user is stored with an assigned role (editor or owner), supporting flexible access management.

Figure 25 presents a simple class diagram showing the key attributes of the Mindmap model used for storage.

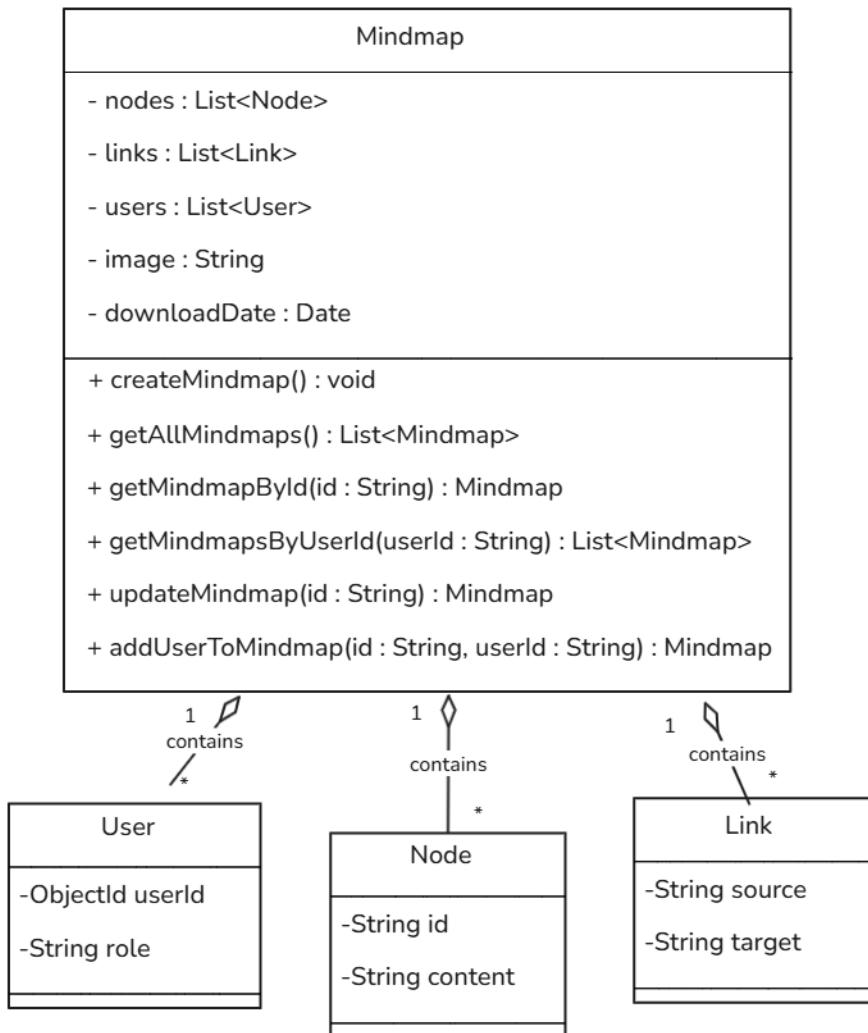


Figure 25: Class Diagram for Mind Map Data Model

2.1.4.6 Frontend development and visualization

The frontend of the WriteWizard system was developed using React, with the objective of providing a highly interactive and intuitive interface for mind map generation, editing, and visualization. It communicates primarily with the Node.js backend, which handles document editing functionalities, manages real-time collaboration, and facilitates database operations as part of the MERN-based architecture. In addition, it connects two Python-based backend services: one for mind map generation and another for semantic image matching.

Through these integrations, the frontend retrieves mind map data, dynamically renders it using D3.js, and enables users to interact with and manipulate the structure. Additional functionality such as image integration, live layout adjustments, and export capabilities are embedded into the interface, enhancing user control and visual clarity.

The following are the core frontend functions, grouped by category for clarity.

Fetching and processing data from mind map microservice

The front end retrieves mind map data from a backend microservice powered by a fine-tuned language model. This service returns a structured hierarchical JSON based on the input text or document. The following functions manage the retrieval and processing of this data:

- `fetchMindMap()` is triggered when the user wants to generate a standard mind map. It sends an asynchronous API call to the backend `/generate` endpoint, which is handled by the mind map generation microservice. The response returns a structured JSON containing the base-level hierarchy of the mind map, which is then parsed and rendered within the interface.
- `fetchExtendedMindMap()` is used when the user chooses to generate an extended version of the mind map. It follows the same structure as the standard fetch function but calls the `/extend` endpoint. This endpoint provides a more comprehensive hierarchy, expanding on the source content with additional layers of depth and subtopics.

- `fetchSimplifiedMindMap()` allows users to retrieve a simplified mind map. It sends a request to the `/simplify` endpoint, which returns a condensed version of the original input. The result includes fewer, broader concepts and is particularly useful for users seeking an overview or summary representation.

Figure 26 illustrates the implementation of the `fetchMindMap()` function, which demonstrates how the front end connects to the backend microservice, sends the request, processes the JSON response, and prepares the data for rendering. The other functions follow a similar structure and logic



```

// Function to fetch mind map data from the "generate" endpoint
async function fetchMindMap() {
  try {
    hasFetchedDatabaseRef.current = true;
    console.log("fetchMindMap");

    const endpoint = "generate";

    const response = await fetch(`/${baseApiUrl}/${endpoint}`, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ content: selectedText }),
    });

    // Check for server response errors
    if (!response.ok) {
      throw new Error(`Server error: ${response.status}`);
    }

    const data = await response.json();

    // If the API returned an error, log it and exit
    if (data.error) {
      console.error("API returned error: " + data.error);
      return;
    }

    // Verify that data.mindmap is an array; otherwise, throw an error
    if (!data || !Array.isArray(data.mindmap)) {
      throw new Error("Invalid data structure: " + JSON.stringify(data));
    }

    // Store the returned data
    const mindmapRoot = data;

    // Clear previous node and link data
    nodes = [];
    links = [];

    // Traverse the response structure to build nodes and links
    mindmapRoot.mindmap.forEach((mindmapNode) => {
      traverseMindmap(mindmapNode, null);
    });

    // Build hierarchical structure and apply tree layout to the nodes
    const hierarchyData = buildHierarchy(nodes, links);
    applyTreeLayout(hierarchyData);

    // Fetch matched images and then update visualization and loading state
    await fetchMatchedImages(nodes, imageCatalog)
      .then(() => {
        update();
      })
      .then(() => {
        setLoading(false);
      })
      .catch((err) =>
        console.error("Error fetching matched images after adding node:", err));
  } catch (error) {
    console.error("Error fetching data from mock data:", error);
    setLoading(false);
  }
}

```

Figure 26: fetchMindMap Function to Handle Mind Map Generation Request

- `traverseMindmap()` is responsible for processing the retrieved JSON. It recursively walks through the hierarchical structure, identifies relationships between nodes, and converts the structure into two arrays: one for nodes and one for links. These arrays are essential for D3 to accurately render the graph layout.

Figure 27 illustrates how the hierarchical structure is traversed to build the node-link arrays.



```

● ● ● JS MindmapModal.js

let uniqueNodeCounter = 1;

function traverseMindmap(nodeData, parentData) {
  if (!nodeData || !nodeData.name) return;

  // Generate a unique ID and create the node
  const uniqueId = `${nodeData.name}-${uniqueNodeCounter++}`;
  nodes.push({
    id: uniqueId,
    text: nodeData.name,
  });

  // Create a link to the parent node if applicable
  if (parentData && parentData.id) {
    links.push({
      source: parentData.id,
      target: uniqueId,
      type: "HAS_SUBNODE",
    });
  }

  // Recursively process subnodes
  if (Array.isArray(nodeData.subnodes)) {
    nodeData.subnodes.forEach((subnode) => {
      traverseMindmap(subnode, { id: uniqueId });
    });
  }
}

```

Figure 27: `traverseMindmap` Function to Extract Hierarchical Node-Link Structure

Image preloading and matching for nodes

A core enhancement in the system is the ability to enrich nodes with semantically relevant images. This is accomplished by first retrieving an image catalog from the database, which is then preloaded and prepared for visualization. The titles of the mind map nodes are sent along with this image set to a backend microservice responsible for semantic image matching. This service analyzes the textual content of each node and returns the most relevant image, which is then mapped to the corresponding node and rendered directly within the visualization.

- `fetchImageCatalog()` sends a request to the backend to retrieve available image resources.
- `preloadImages()` fetches each image and converts it to a base64 data URL, ensuring the images are fully loaded and immediately available for rendering.

Figure 28 highlights the process of loading and preparing the image catalog for visualization.



```
// Fetch image catalog from the backend
async function fetchImageCatalog() {
  const token = localStorage.getItem("token");
  if (!token) return;

  try {
    const endpointUrl = `${process.env.REACT_APP_BACKEND_API_URL}/api/pads/${padId}/images`;
    const response = await fetch(endpointUrl, {
      method: "GET",
      headers: {
        "Content-Type": "application/json",
        "Authorization": `Bearer ${token}`,
      },
    });
    if (!response.ok) {
      throw new Error(`Failed to fetch image catalog: ${response.status}`);
    }

    const data = await response.json();

    if (data.imagePairs && Array.isArray(data.imagePairs)) {
      const mappedCatalog = data.imagePairs.map(pair => ({
        url: pair.image_url,
        description: pair.image_description,
      }));
      imageCatalogRef.current = mappedCatalog;
      return mappedCatalog;
    } else {
      throw new Error("No imagePairs found in response");
    }
  } catch (error) {
    console.error("Error fetching image catalog:", error);
    return [];
  }
}

// Preload images by converting them to base64 data URLs
async function preloadImages(imageCatalog) {
  await Promise.all(imageCatalog.map(async (item) => {
    try {
      const response = await fetch(item.url, { mode: "cors" });
      if (!response.ok) return;
      const blob = await response.blob();
      await new Promise((resolve) => {
        const reader = new FileReader();
        reader.onload = () => {
          item.dataUrl = reader.result;
        };
        reader.onerror = () => resolve();
        reader.readAsDataURL(blob);
      });
    } catch (error) {
      console.error(`Skipping image ${item.url} due to error:`, error);
    }
  }));
  return imageCatalog;
}
```

Figure 28: `fetchImageCatalog` and `preloadImages`
Functions to Prepare Image Data

- `fetchMatchedImages()` sends node titles to a semantic image matching microservice, which identifies the most relevant image based on text similarity.
- `getMatchingImage()` accesses the matching result and associates it with each node in the visualization so that it can be rendered inline.

Figure 29 illustrates how matching images are selected based on node content, while Figure 30 shows how these matched images are embedded inline in the visualization.



```

● ● ● JS MindmapModal.js

// Global cache to store API-matched images keyed by node text
let matchedImageMap = {};

// Fetch matched images from the external API
async function fetchMatchedImages(nodes, imageCatalog) {
  if (!nodes || nodes.length === 0) {
    console.warn("No nodes provided; skipping API call.");
    return;
  }

  const endpoint = "https://sanjayan201-my-image-matching-app.hf.space/match";
  const payload = {
    node_texts: nodes.map(n => n.text),
    image_pairs: imageCatalogRef.current.map(item => ({
      image_url: item.url,
      image_description: item.description,
    })),
  };
  console.log("Payload:", payload);
  const response = await fetch(endpoint, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(payload),
  });

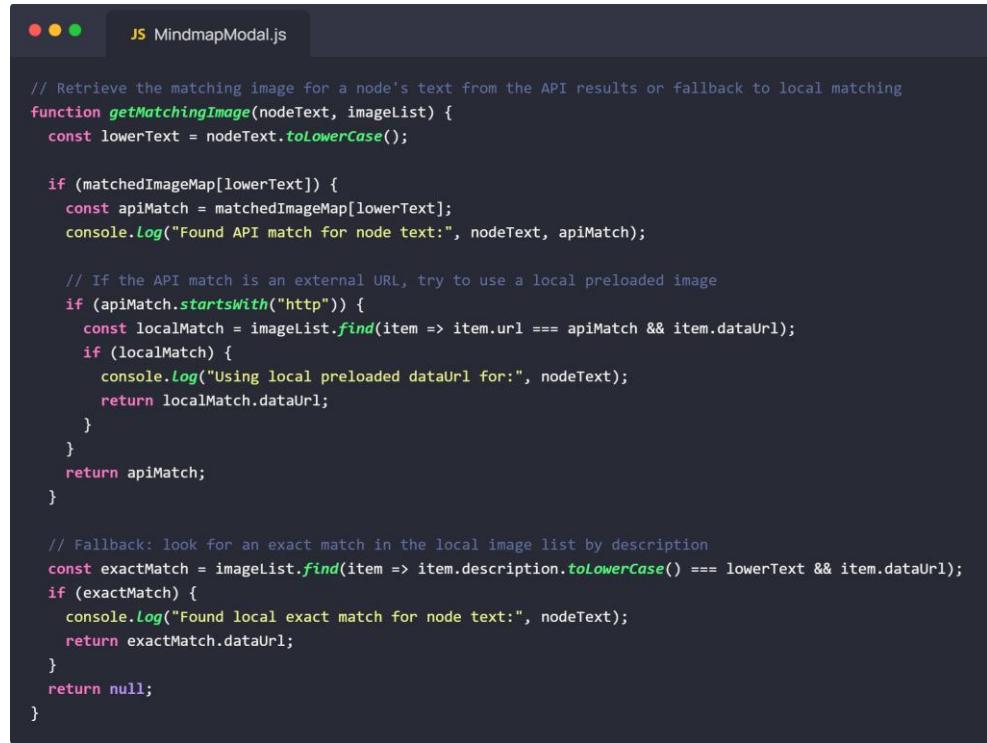
  if (!response.ok) {
    throw new Error(`Failed to fetch matches: ${response.status}`);
  }

  const data = await response.json();
  console.log("API Response:", data);

  // Update matchedImageMap with the new data
  matchedImageMap = {};
  if (data.matched_pairs && Array.isArray(data.matched_pairs)) {
    data.matched_pairs.forEach(pair => {
      matchedImageMap[pair.node.toLowerCase()] = pair.image_url;
    });
  }
  console.log("Updated matchedImageMap:", matchedImageMap);
}

```

Figure 29: `fetchMatchedImages` Function for Node Image Matching



```
// Retrieve the matching image for a node's text from the API results or fallback to local matching
function getMatchingImage(nodeText, imageList) {
  const lowerText = nodeText.toLowerCase();

  if (matchedImageMap[lowerText]) {
    const apiMatch = matchedImageMap[lowerText];
    console.log("Found API match for node text:", nodeText, apiMatch);

    // If the API match is an external URL, try to use a local preloaded image
    if (apiMatch.startsWith("http")) {
      const localMatch = imageList.find(item => item.url === apiMatch && item.dataUrl);
      if (localMatch) {
        console.log("Using local preloaded dataUrl for:", nodeText);
        return localMatch.dataUrl;
      }
    }
    return apiMatch;
  }

  // Fallback: look for an exact match in the local image list by description
  const exactMatch = imageList.find(item => item.description.toLowerCase() === lowerText && item.dataUrl);
  if (exactMatch) {
    console.log("Found local exact match for node text:", nodeText);
    return exactMatch.dataUrl;
  }
  return null;
}
```

Figure 30: `getMatchingImage` Function for Inline Image Rendering

Visualization and rendering with D3

The front end utilizes D3.js to render and continuously update the mind map in real time. Several key functions control the canvas behavior, layout computation, and redraw logic.

- `generateMindmap()` initializes the visualization environment. It creates the SVG container, appends the groups for nodes and links, sets up zooming and panning capabilities, and starts the D3 force simulation engine that dynamically positions the elements.

Figure 31 shows the setup process of the SVG layout and simulation initialization.



```
const generateMindmap = () => {
  // Remove existing SVG
  d3.select(graphRef.current).select("svg").remove();

  // Set dimensions
  width = mindmapContainer.clientWidth || 800;
  height = mindmapContainer.clientHeight || 600;

  // Create SVG container and groups
  svg = d3.select(graphRef.current)
    .append("svg")
    .attr("width", "100%")
    .attr("height", "100%")
    .attr("viewBox", `0 0 ${width} ${height}`)
    .attr("preserveAspectRatio", "xMidYMid meet")
    .style("cursor", "pointer");

  zoomGroup = svg.append("g");
  linkGroup = zoomGroup.append("g").attr("class", "links");
  nodeGroup = zoomGroup.append("g").attr("class", "nodes");

  // Setup zoom behavior
  zoomBehavior = d3.zoom()
    .scaleExtent([0.05, 5])
    .on("zoom", event => zoomGroup.attr("transform", event.transform));
  svg.call(zoomBehavior);

  // Initialize force simulation
  simulation = d3.forceSimulation(nodes)
    .force("link", d3.forceLink(links).id(d => d.id).distance(150))
    .force("charge", d3.forceManyBody().strength(-300))
    .force("collide", d3.forceCollide().radius(d => Math.max(30, d.text.length * 5) + 10))
    .force("center", d3.forceCenter(width/2, height/2))
    .on("tick", ticked);

  if (!hasFetchedDatabaseRef.current) {
    fetchFromDatabase().catch(err => console.error("Error fetching mind map data:", err));
  }
};
```

Figure 31: `generateMindmap` Function to Initialize SVG and D3 Simulation

- `update()` is responsible for refreshing the canvas after structural changes such as node additions, deletions, or drag movements. It re-binds the updated data arrays to SVG elements and recalculates the node and edge rendering.

Figure 32 shows the core logic for updating the visual layout of the mind map, ensuring that nodes and links reflect the current state.

- `reRenderNode()` manages the appearance of each node by drawing its shape, embedding associated images, and enabling interactive text editing. It determines layout dimensions based on whether an image is matched and renders content accordingly.

Figure 33 illustrates how individual nodes are redrawn with images and labels when visual updates occur.

```

JS MindmapModal.js

function update() {
  if (!window.nodeColorMap) window.nodeColorMap = new Map();
  const nodePosMap = new Map(nodes.map(n => [n.id, { x: n.x, y: n.y }])); 
  const hierarchyData = buildHierarchy(nodes, links);

  // Update depths from hierarchy data
  hierarchyData.each(node => {
    const n = nodes.find(n => n.id === node.data.id);
    if (n && n.depth === null) {
      n.depth = node.depth;
      if (!colorMap[n.depth]) colorMap[n.depth] = defaultColorScale(node.depth);
    }
  });

  const rootColor = colorMap[0] || defaultColorScale(0);
  nodes.forEach(n => {
    const isStandalone = links.some(link => link.source === n.id || link.target === n.id);
    if (isStandalone && n.depth === undefined) n.depth = 0;
    if (!window.nodeColorMap.has(n.id)) {
      window.nodeColorMap.set(n.id, isStandalone ? rootColor : (colorMap[n.depth] || defaultColorScale(n.depth)));
    }
  });

  // Ensure proper depth and update color mapping
  nodes.forEach(n => {
    if (links.some(link => link.source === n.id || link.target === n.id) && (!n.depth || n.depth === 0)) {
      const found = hierarchyData.find(h => h.data.id === n.id);
      n.depth = found ? found.depth : 0;
    }
    window.nodeColorMap.set(n.id, colorMap[n.depth] || defaultColorScale(n.depth));
  });

  // Update links
  linkGroup.selectAll("line")
    .data(links, d => `${d.source}-${d.target}`)
    .join(
      enter => enter.append("line")
        .attr("stroke", "#999")
        .attr("stroke-width", 2)
        .on("click", (e, d) => {
          selectedLink = d;
          selectedNode = null;
          updateBlinking();
          e.stopPropagation();
        })
    );
  update => update;
  exit => exit.remove();
}

// Update nodes
nodeGroup.selectAll("g")
  .data(nodes, d => d.id)
  .join(
    enter => {
      const g = enter.append("g")
        .call(d3.drag()
          .on("start", dragstarted)
          .on("drag", dragged)
          .on("end", dragended))
        .on("click", (e, d) => {
          if (interactiveRelationMode) {
            if (selectedSourceForRelation && selectedSourceForRelation.id === d.id) {
              alert("Source and target nodes cannot be the same. Please select a different target node.");
              e.stopPropagation();
              return;
            }
            links.push({ source: selectedSourceForRelation.id, target: d.id, type: "HAS_SUBNODE" });
            d.depth = selectedSourceForRelation.depth + 1;
            updateDepthRecursively(d, new Map(nodes.map(n => [n.id, n])), links);
            nodes.forEach(n => {
              if (!colorMap[n.depth]) colorMap[n.depth] = defaultColorScale(n.depth);
              window.nodeColorMap.set(n.id, colorMap[n.depth]);
            });
            interactiveRelationMode = false;
            selectedSourceForRelation = null;
            if (addRelationInteractiveSpan) {
              addRelationInteractiveSpan.style.display = "inline-block";
            }
            if (selectRelationFeedbackRef.current) {
              selectRelationFeedbackRef.current.style.display = "none";
              update();
              e.stopPropagation();
              return;
            }
            selectedLevel = d.depth;
            if (colorPickerContainerRef.current) {
              colorPickerContainerRef.current.style.display = "block";
            }
            colorPickerRef.current.value = colorMap[selectedLevel] || defaultColorScale(d.depth || 0);
            selectedNode = d;
            selectedLink = null;
            updateBlinking();
            e.stopPropagation();
            const matchedUrl = d.imageDeleted ? null : getMatchingImage(d.text, imageCatalog);
            if (deleteImageBtnRef.current) {
              deleteImageBtnRef.current.style.display = matchedUrl ? "inline-block" : "none";
            }
          }
        })
        .on("dblclick", e => {});
      // Re-render each node using reRenderNode()
      nodeGroup.selectAll("g").each(function(d) {
        reRenderNode(d, d3.select(this));
      });
      return g;
    },
    update => update.attr("transform", d => {
      const pos = nodePosMap.get(d.id);
      return pos ? `translate(${pos.x},${pos.y})` : `translate(${d.x},${d.y})`;
    })
    .select("ellipse")
    .attr("fill", d => window.nodeColorMap.get(d.id)),
    exit => exit.remove();
  );

  simulation.nodes(nodes);
  simulation.force("link").links(links);
  simulation.alpha(1).restart();
  window.requestAnimationFrame(fitToScreen);
}

```

Figure 32: update Function to Refresh Visualization After Changes

```

function reRenderNode(d, group) {
  group.selectAll("*").remove();
  const matchedUrl = d.imageDeleted ? null : getMatchingImage(d.text, imageCatalogRef.current);

  // Text click handler for editing node text
  const onTextClick = event => {
    event.stopPropagation();
    const input = document.createElement("input");
    input.type = "text";
    input.value = d.text;
    input.style = `position: absolute; left:${event.pageX}px; top:${event.pageY}px; z-index:1000;`;
    document.body.appendChild(input);
    input.focus();
    input.addEventListener("blur", () => {
      const updatedText = input.value.trim();
      if (updatedText && updatedText !== d.text) {
        d.text = updatedText;
        setLoading(true);
        const nodeObj = nodes.find(n => n.id === d.id);
        if (nodeObj) nodeObj.text = updatedText;
        fetchMatchedImages(nodes, imageCatalog)
          .then(update)
          .then(() => setLoading(false))
          .catch(err => {
            console.error("Error fetching matched images after editing node:", err);
            setLoading(false);
          });
      }
      document.body.removeChild(input);
      handleDocumentClick();
      d.imageDeleted = false;
    });
  };

  if (matchedUrl) {
    const imageWidth = 100, imageHeight = 100, spacing = 10, estTextHeight = 20;
    const compHeight = imageHeight + spacing + estTextHeight,
      compTop = -compHeight / 2,
      textY = compTop + imageHeight + spacing,
      ellipseRx = Math.max(50, d.text.length * 6) + 40,
      ellipseRy = compHeight / 2 + 20;

    group.append("ellipse")
      .attr("rx", ellipseRx)
      .attr("ry", ellipseRy)
      .attr("fill", window.nodeColorMap.get(d.id))
      .attr("stroke", "#333")
      .attr("stroke-width", 2);

    group.append("image")
      .attr("xlink:href", matchedUrl)
      .attr("width", imageWidth)
      .attr("height", imageHeight)
      .attr("x", -imageWidth / 2)
      .attr("y", compTop)
      .on("click", event => {
        event.stopPropagation();
        setFullScreenImageUrl(matchedUrl);
        handleDocumentClick();
      });

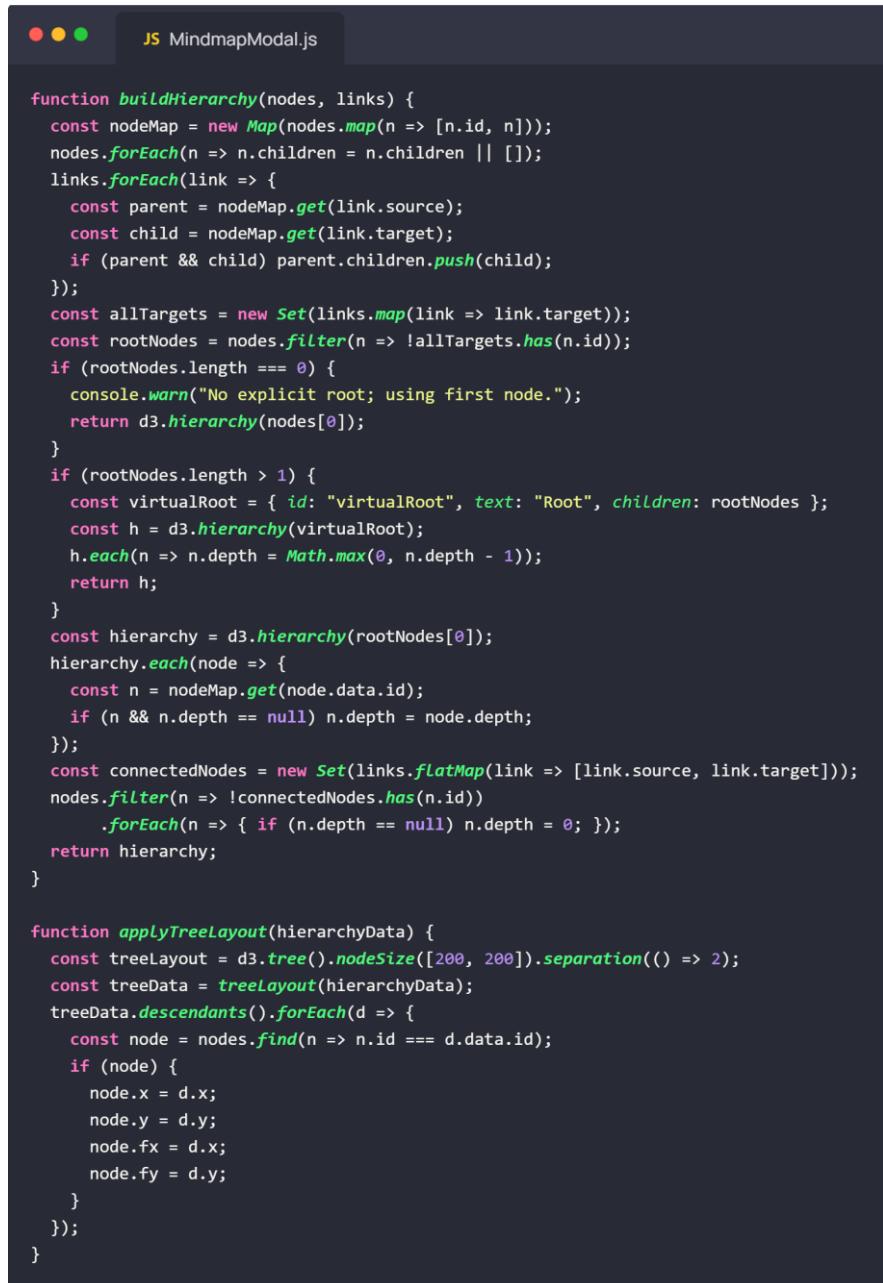
    group.append("text")
      .attr("text-anchor", "middle")
      .attr("alignment-baseline", "hanging")
      .attr("y", textY)
      .text(d.text)
      .on("click", onTextClick);
  } else {
    const ellipseRx = Math.max(40, d.text.length * 5),
      ellipseRy = 40;
    group.append("ellipse")
      .attr("rx", ellipseRx)
      .attr("ry", ellipseRy)
      .attr("fill", window.nodeColorMap.get(d.id))
      .attr("stroke", "#333")
      .attr("stroke-width", 2);
    group.append("text")
      .attr("text-anchor", "middle")
      .attr("alignment-baseline", "middle")
      .text(d.text)
      .on("click", onTextClick);
  }
}

```

Figure 33: `reRenderNode` Function to Redraw Nodes with Images and Interactive Text Editing

- `applyTreeLayout()` takes flat node/link data and constructs a hierarchical tree structure by establishing parent-child relationships. This is what ensures semantic depth and topic hierarchy are defined.
- `buildHierarchy()` takes that already-built hierarchy and computes the x/y positions of each node using D3's tree layout, effectively converting the abstract hierarchy into spatial coordinates for rendering.

Figure 34 demonstrates the transformation from flat structures into a hierarchical layout.



```

● ● ● JS MindmapModal.js

function buildHierarchy(nodes, links) {
  const nodeMap = new Map(nodes.map(n => [n.id, n]));
  nodes.forEach(n => n.children = n.children || []);
  links.forEach(link => {
    const parent = nodeMap.get(link.source);
    const child = nodeMap.get(link.target);
    if (parent && child) parent.children.push(child);
  });
  const allTargets = new Set(links.map(link => link.target));
  const rootNodes = nodes.filter(n => !allTargets.has(n.id));
  if (rootNodes.length === 0) {
    console.warn("No explicit root; using first node.");
    return d3.hierarchy(nodes[0]);
  }
  if (rootNodes.length > 1) {
    const virtualRoot = { id: "virtualRoot", text: "Root", children: rootNodes };
    const h = d3.hierarchy(virtualRoot);
    h.each(n => n.depth = Math.max(0, n.depth - 1));
    return h;
  }
  const hierarchy = d3.hierarchy(rootNodes[0]);
  hierarchy.each(node => {
    const n = nodeMap.get(node.data.id);
    if (n && n.depth == null) n.depth = node.depth;
  });
  const connectedNodes = new Set(links.flatMap(link => [link.source, link.target]));
  nodes.filter(n => !connectedNodes.has(n.id))
    .forEach(n => { if (n.depth == null) n.depth = 0; });
  return hierarchy;
}

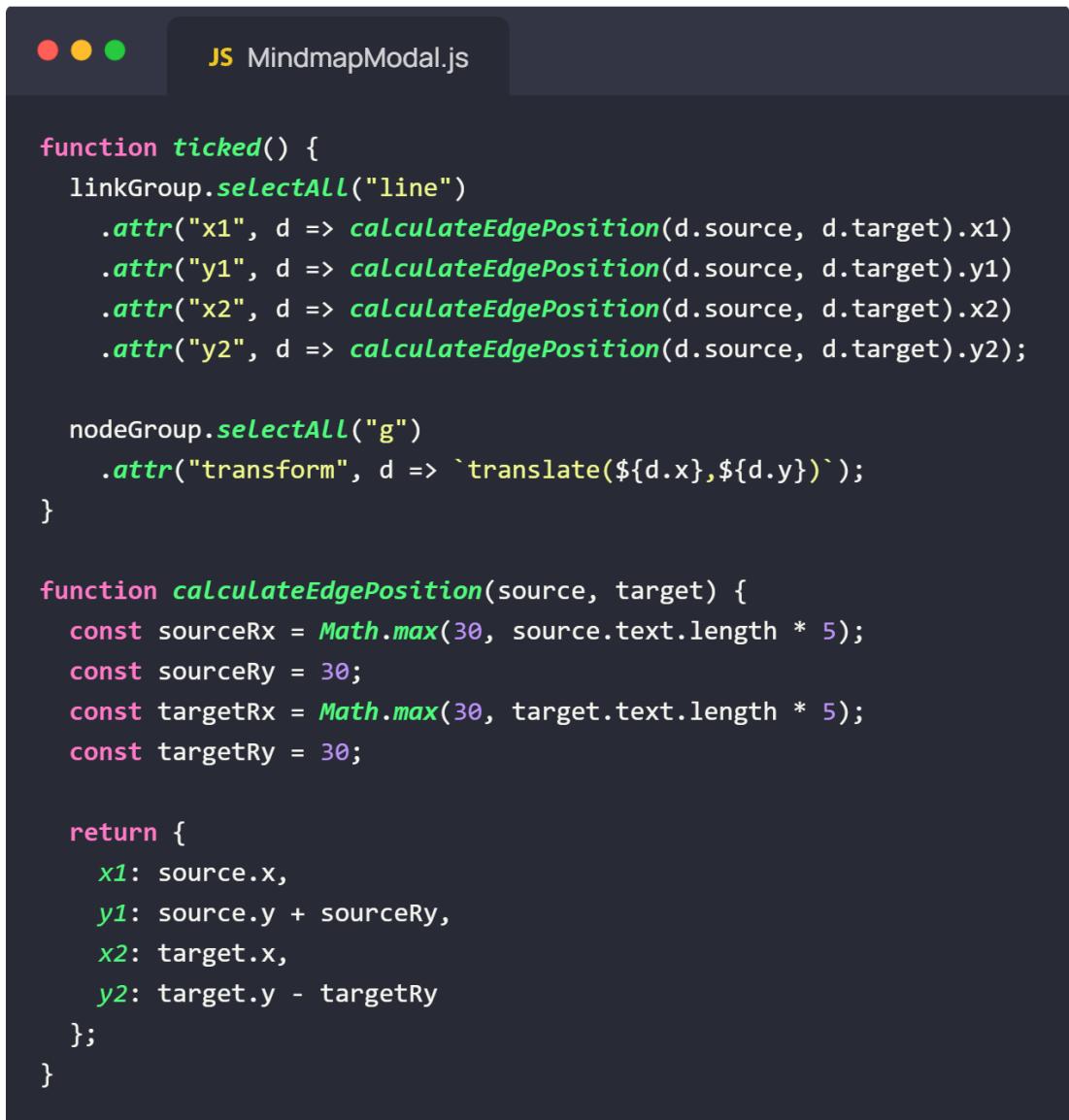
function applyTreeLayout(hierarchyData) {
  const treeLayout = d3.tree().nodeSize([200, 200]).separation(() => 2);
  const treeData = treeLayout(hierarchyData);
  treeData.descendants().forEach(d => {
    const node = nodes.find(n => n.id === d.data.id);
    if (node) {
      node.x = d.x;
      node.y = d.y;
      node.fx = d.x;
      node.fy = d.y;
    }
  });
}

```

Figure 34: `applyTreeLayout` and `buildHierarchy` Functions to Build a Hierarchical Layout

- `ticked()` is the core D3 tick function that runs continuously during simulation. It updates the positions of nodes and links on each frame, ensuring that the layout reacts naturally to user interaction and simulated physics.
- `calculateEdgePosition()` determines the geometric properties of edges. It calculates precise curvature and anchor points for link rendering, ensuring that all node connections appear smooth and visually consistent.

Figure 35 illustrates the simulation loop and geometry calculations used for dynamic positioning and smooth rendering.



```

function ticked() {
  linkGroup.selectAll("line")
    .attr("x1", d => calculateEdgePosition(d.source, d.target).x1)
    .attr("y1", d => calculateEdgePosition(d.source, d.target).y1)
    .attr("x2", d => calculateEdgePosition(d.source, d.target).x2)
    .attr("y2", d => calculateEdgePosition(d.source, d.target).y2);

  nodeGroup.selectAll("g")
    .attr("transform", d => `translate(${d.x}, ${d.y})`);
}

function calculateEdgePosition(source, target) {
  const sourceRx = Math.max(30, source.text.length * 5);
  const sourceRy = 30;
  const targetRx = Math.max(30, target.text.length * 5);
  const targetRy = 30;

  return {
    x1: source.x,
    y1: source.y + sourceRy,
    x2: target.x,
    y2: target.y - targetRy
  };
}

```

Figure 35: `ticked` and `calculateEdgePosition` Functions for Dynamic Node Positioning and Edge Alignment

User interaction and event handling

The system allows users to build and modify the mind map directly within the interface through intuitive mouse and keyboard interactions.

- `handleAddNode()` adds a new node to the current structure. It generates a unique identifier, sets the node's initial position relative to its parent, and updates the visualization accordingly.

Figure 36 shows the implementation of the logic that enables dynamic node addition.



The screenshot shows a code editor window with a dark theme. The title bar says "JS MindmapModal.js". The code is a JavaScript function named `handleAddNode`. It first checks if a new node name is provided. If not, it shows an alert and returns. It then sets the `addNodeBtn.disabled` property to `true`, sets a loading state, and clears the `newNodeNameRef.current.value`. It determines the root node (either the first node or the first node with depth 0). It calculates a random position for the new node relative to the root node or the container. It creates a new node object with a unique ID, the new node name, and the calculated position. It pushes this new node into the `nodes` array. Finally, it fetches images and refreshes the visualization.

```
const handleAddNode = () => {
  const newNodeName = newNodeNameRef.current.value.trim();
  if (!newNodeName) {
    alert("Please enter a valid node name.");
    return;
  }

  addNodeBtn.disabled = true;
  setLoading(true);
  newNodeNameRef.current.value = "";

  // Determine root node or fallback to the first available node
  const rootNode = nodes.find(node => node.depth === 0) || nodes[0];

  // Calculate new node position
  let x, y;
  if (rootNode) {
    const distance = 100 + Math.random() * 300;
    const angle = Math.random() * 2 * Math.PI;
    x = rootNode.x + distance * Math.cos(angle);
    y = rootNode.y + distance * Math.sin(angle);
  } else {
    const container = graphRef.current;
    x = container.clientWidth / 2;
    y = container.clientHeight / 2;
  }

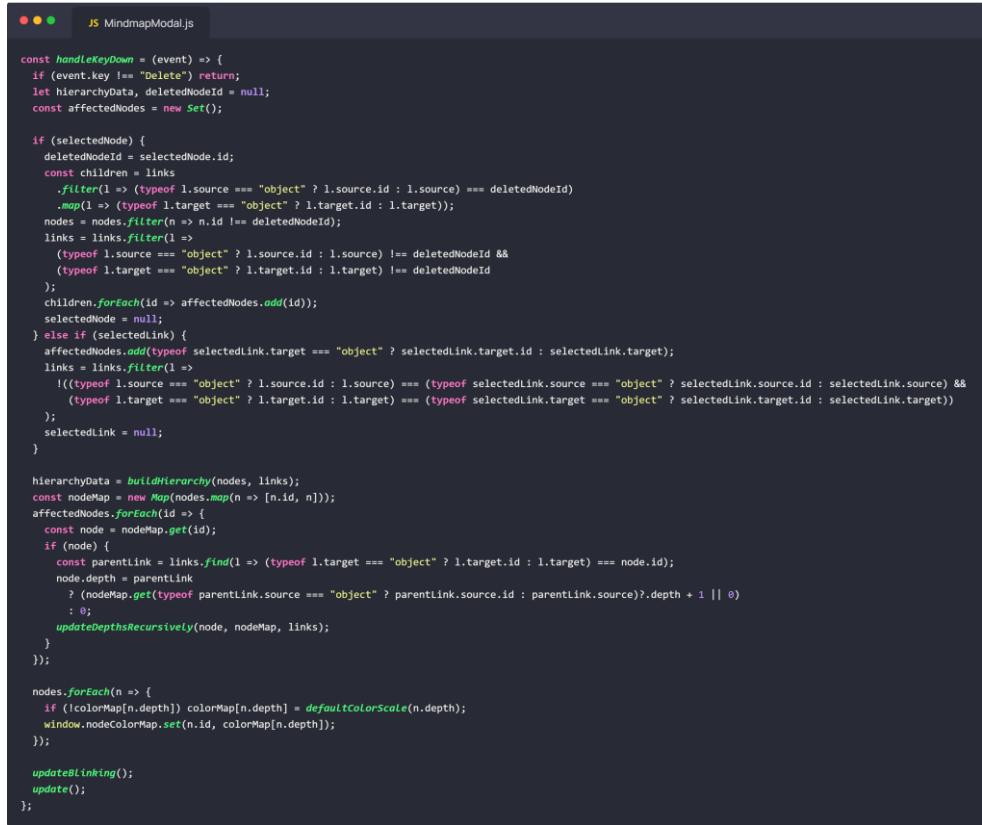
  // Create and add the new node
  const newNode = {
    id: `node-${uniqueNodeCounter++}`,
    text: newNodeName,
    x,
    fx: x,
    fy: y,
    depth: 0,
  };
  nodes.push(newNode);

  // Fetch images and refresh visualization
  fetchMatchedImages(nodes, imageCatalog)
    .then(() => {
      update();
      setLoading(false);
      addNodeBtn.disabled = false;
    })
    .catch(err => {
      console.error("Error fetching matched images after adding node:", err);
      setLoading(false);
      addNodeBtn.disabled = false;
    });
};
```

Figure 36: `handleAddNode` Function to Add New Node to the Mind Map

- `handleKeyDown()` listens for keyboard events and handles the deletion of selected nodes or links when specific keys are pressed. This supports fast cleanup and structural changes without needing mouse input.

Figure 37 presents the handler for deleting selected elements using keyboard events.



```

JS MindmapModal.js

const handleKeyDown = (event) => {
  if (event.key !== "Delete") return;
  let hierarchyData, deletedNodeId = null;
  const affectedNodes = new Set();

  if (selectedNode) {
    deletedNodeId = selectedNode.id;
    const children = links
      .filter(l => (typeof l.source === "object" ? l.source.id : l.source) === deletedNodeId)
      .map(l => (typeof l.target === "object" ? l.target.id : l.target));
    nodes = nodes.filter(n => n.id !== deletedNodeId);
    links = links.filter(l =>
      (typeof l.source === "object" ? l.source.id : l.source) !== deletedNodeId &&
      (typeof l.target === "object" ? l.target.id : l.target) !== deletedNodeId
    );
    children.forEach(id => affectedNodes.add(id));
    selectedNode = null;
  } else if (selectedLink) {
    affectedNodes.add(typeof selectedLink.target === "object" ? selectedLink.target.id : selectedLink.target);
    links = links.filter(l =>
      !(typeof l.source === "object" ? l.source.id : l.source) === (typeof selectedLink.source === "object" ? selectedLink.source.id : selectedLink.source) &&
      (typeof l.target === "object" ? l.target.id : l.target) === (typeof selectedLink.target === "object" ? selectedLink.target.id : selectedLink.target)
    );
    selectedLink = null;
  }

  hierarchyData = buildHierarchy(nodes, links);
  const nodeMap = new Map(nodes.map(n => [n.id, n]));
  affectedNodes.forEach(id => {
    const node = nodeMap.get(id);
    if (node) {
      const parentLink = links.find(l => (typeof l.target === "object" ? l.target.id : l.target) === node.id);
      node.depth = parentLink
        ? (nodeMap.get(typeof parentLink.source === "object" ? parentLink.source.id : parentLink.source).depth + 1 || 0)
        : 0;
      updateDepthsRecursively(node, nodeMap, links);
    }
  });

  nodes.forEach(n => {
    if (!colorMap[n.depth]) colorMap[n.depth] = defaultColorScale(n.depth);
    window.nodeColorMap.set(n.id, colorMap[n.depth]);
  });

  updateBlinking();
  update();
};

```

Figure 37: `handleKeyDown` Function to Manage Deletion of Nodes and Links

- `dragstarted()`, `dragged()`, and `dragended()` control the interactive movement of nodes. These functions update the internal positions and force simulation parameters, enabling real-time layout reshaping as users drag nodes around.

Figure 38 displays the drag-and-drop events used to update node positions interactively.



```

JS MindmapModal.js

function dragstarted(event, d) {
  simulation.alpha(0).stop();
  d.fx = d.x;
  d.fy = d.y;
}

function dragged(event, d) {
  d.fx = event.x;
  d.fy = event.y;

  // Update node position
  const node = nodes.find(n => n.id === d.id);
  if (node) {
    node.x = d.fx;
    node.y = d.fy;
  }

  nodeGroup.selectAll("g")
    .filter(n => n.id === d.id)
    .attr("transform", `translate(${d.fx},${d.fy})`);

  // Update connected links
  linkGroup.selectAll("line")
    .filter(l => l.source.id === d.id || l.target.id === d.id)
    .attr("x1", l => l.source.id === d.id ? d.fx : l.source.x)
    .attr("y1", l => l.source.id === d.id ? d.fy : l.source.y)
    .attr("x2", l => l.target.id === d.id ? d.fx : l.target.x)
    .attr("y2", l => l.target.id === d.id ? d.fy : l.target.y);
}

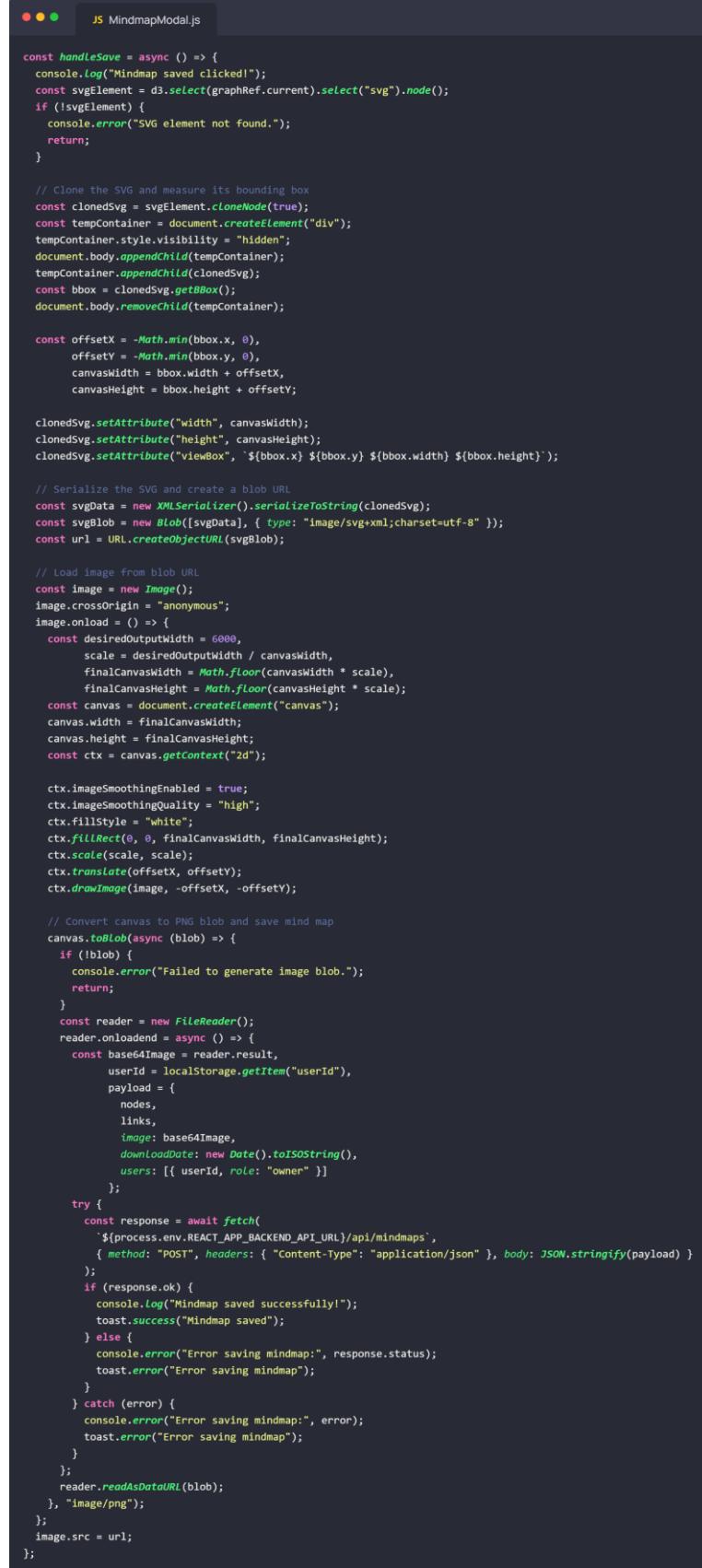
function dragended(event, d) {
  d.fx = null;
  d.fy = null;
}

```

Figure 38: `dragstarted`, `dragged`, and `dragended` Functions for Node Repositioning

- `handleSave()` captures the current state of the mind map, including nodes, links, and any embedded images. It serializes this data and the rendered SVG image, and sends them to the backend for storage.

Figure 39 demonstrates the function that prepares and sends the final mind map for persistent storage.



```

JS MindmapModal.js

const handleSave = async () => {
  console.log("Mindmap saved clicked!");
  const svgElement = d3.select(graphRef.current).select("svg").node();
  if (!svgElement) {
    console.error("SVG element not found.");
    return;
  }

  // Clone the SVG and measure its bounding box
  const clonedSvg = svgElement.cloneNode(true);
  const tempContainer = document.createElement("div");
  tempContainer.style.visibility = "hidden";
  document.body.appendChild(tempContainer);
  tempContainer.appendChild(clonedSvg);
  const bbox = clonedSvg.getBBox();
  document.body.removeChild(tempContainer);

  const offsetX = -Math.min(bbox.x, 0),
        offsetY = -Math.min(bbox.y, 0),
        canvasWidth = bbox.width + offsetX,
        canvasHeight = bbox.height + offsetY;

  clonedSvg.setAttribute("width", canvasWidth);
  clonedSvg.setAttribute("height", canvasHeight);
  clonedSvg.setAttribute("viewBox", `${bbox.x} ${bbox.y} ${bbox.width} ${bbox.height}`);

  // Serialize the SVG and create a blob URL
  const svgData = new XMLSerializer().serializeToString(clonedSvg);
  const svgBlob = new Blob([svgData], { type: "image/svg+xml;charset=utf-8" });
  const url = URL.createObjectURL(svgBlob);

  // Load image from blob URL
  const image = new Image();
  image.crossOrigin = "anonymous";
  image.onload = () => {
    const desiredOutputWidth = 6000,
          scale = desiredOutputWidth / canvasWidth,
          finalCanvasWidth = Math.floor(canvasWidth * scale),
          finalCanvasHeight = Math.floor(canvasHeight * scale);
    const canvas = document.createElement("canvas");
    canvas.width = finalCanvasWidth;
    canvas.height = finalCanvasHeight;
    const ctx = canvas.getContext("2d");

    ctx.imageSmoothingEnabled = true;
    ctx.imageSmoothingQuality = "high";
    ctx.fillStyle = "white";
    ctx.fillRect(0, 0, finalCanvasWidth, finalCanvasHeight);
    ctx.scale(scale, scale);
    ctx.translate(offsetX, offsetY);
    ctx.drawImage(image, -offsetX, -offsetY);
  }

  // Convert canvas to PNG blob and save mind map
  canvas.toBlob(async (blob) => {
    if (!blob) {
      console.error("Failed to generate image blob.");
      return;
    }
    const reader = new FileReader();
    reader.onloadend = async () => {
      const base64Image = reader.result,
            userId = localStorage.getItem("userId"),
            payload = {
              nodes,
              links,
              image: base64Image,
              downloadDate: new Date().toISOString(),
              users: [{ userId, role: "owner" }]
            };
      try {
        const response = await fetch(
          `${process.env.REACT_APP_BACKEND_API_URL}/api/mindmaps`,
          { method: "POST", headers: { "Content-Type": "application/json" }, body: JSON.stringify(payload) }
        );
        if (response.ok) {
          console.log("Mindmap saved successfully!");
          toast.success("Mindmap saved");
        } else {
          console.error("Error saving mindmap:", response.status);
          toast.error("Error saving mindmap");
        }
      } catch (error) {
        console.error("Error saving mindmap:", error);
        toast.error("Error saving mindmap");
      }
    };
    reader.readAsDataURL(blob);
  }, "image/png");
  image.src = url;
};

```

Figure 39: handleSave Function to Serialize and Store Mind Map

Exporting the mind map

Export functionality allows users to save the visual structure of the mind map as downloadable files in two formats.

- `downloadAsPNG()` converts the current SVG into a PNG image. This is useful for embedding the mind map into reports or presentations.

Figure 40 illustrates the logic used to render and download the mind map in PNG format.



```
JS MindmapModal.js

const downloadAsPNG = () => {
  const svgElement = d3.select(graphRef.current).select("svg").node();
  if (!svgElement) {
    console.error("SVG element not found.");
    return;
  }

  // Clone SVG and measure its bounding box
  const clonedSvg = svgElement.cloneNode(true);
  const tempContainer = document.createElement("div");
  tempContainer.style.visibility = "hidden";
  document.body.appendChild(tempContainer);
  tempContainer.appendChild(clonedSvg);
  const bbox = clonedSvg.getBBox();
  document.body.removeChild(tempContainer);

  const offsetX = -Math.min(bbox.x, 0),
        offsetY = -Math.min(bbox.y, 0),
        canvasWidth = bbox.width + offsetX,
        canvasHeight = bbox.height + offsetY;

  clonedSvg.setAttribute("width", canvasWidth);
  clonedSvg.setAttribute("height", canvasHeight);
  clonedSvg.setAttribute("viewBox", `${bbox.x} ${bbox.y} ${bbox.width} ${bbox.height}`);

  // Serialize SVG and create a blob URL
  const svgData = new XMLSerializer().serializeToString(clonedSvg);
  const svgBlob = new Blob([svgData], { type: "image/svg+xml;charset=utf-8" });
  const url = URL.createObjectURL(svgBlob);

  // Compute scale for desired output resolution
  const desiredOutputWidth = 24000,
        scale = desiredOutputWidth / canvasWidth;

  const image = new Image();
  image.crossOrigin = "anonymous";
  image.onload = () => {
    const finalCanvasWidth = Math.floor(canvasWidth * scale),
          finalCanvasHeight = Math.floor(canvasHeight * scale);
    const canvas = document.createElement("canvas");
    canvas.width = finalCanvasWidth;
    canvas.height = finalCanvasHeight;
    const ctx = canvas.getContext("2d");

    ctx.imageSmoothingEnabled = true;
    ctx.imageSmoothingQuality = "high";
    ctx.fillStyle = "white";
    ctx.fillRect(0, 0, finalCanvasWidth, finalCanvasHeight);

    ctx.scale(scale, scale);
    ctx.translate(offsetX, offsetY);
    ctx.drawImage(image, -offsetX, -offsetY);

    canvas.toBlob(blob => {
      if (blob) {
        const link = document.createElement("a");
        link.href = URL.createObjectURL(blob);
        link.download = "mindmap.png";
        link.click();
        URL.revokeObjectURL(link.href);
        toast.success("Mindmap downloaded");
      } else {
        const dataUrl = canvas.toDataURL("image/png");
        const link = document.createElement("a");
        link.href = dataUrl;
        link.download = "mindmap.png";
        link.click();
        toast.success("Mindmap downloaded (fallback method)");
      }
    }, "image/png");
  };
  image.src = url;
};
```

Figure 40: `downloadAsPNG` Function to Export Mind Map as PNG Image

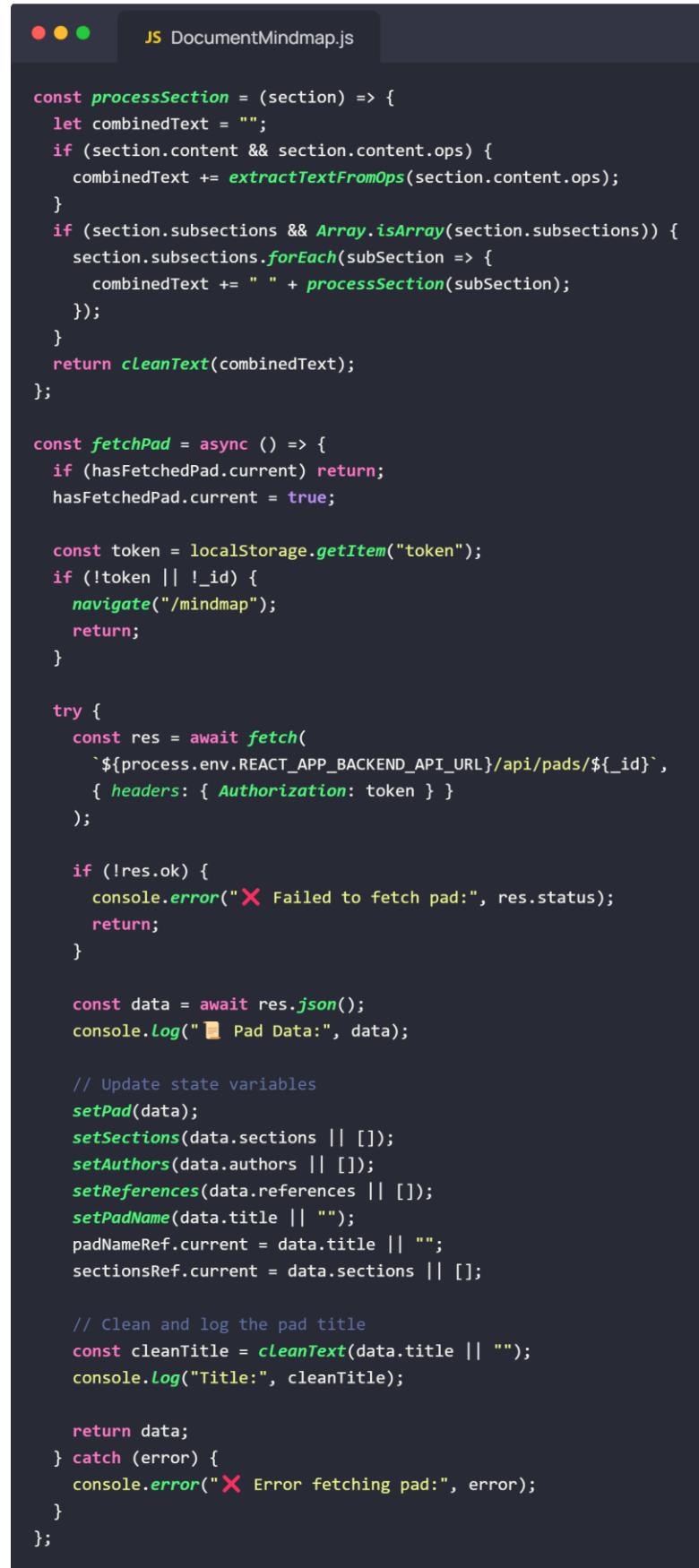
- `downloadAsSVG()` allows users to download the raw SVG markup of the current visualization for use in vector graphic editors or further customization.

Document-level mind map generation

To support mind map generation at the scale of an entire document, the frontend integrates a set of additional functions that process document metadata, extract clean section-level content, and build a unified mind map structure. These features allow users to visualize a complete overview of their entire document.

- `fetchPad()` retrieves the full document (referred to as a "pad") from the backend document management service. This includes fetching the title, authors, references, and all section data. The function also updates frontend state variables, ensuring that all relevant content is available before mind map generation begins.
- `processSection()` takes each section and recursively processes its content and subsections. It converts the rich text format (Quill delta) into plain text and concatenates the content into a clean, flattened structure. This cleaned text becomes the input for the mind map generation service.
- `buildFullMindmap()` coordinates the entire flow. It builds the root node using the pad title and then loops through all sections. For each section, it calls `processSection()` to clean the text and then uses `fetchMindMap()` to retrieve a sub-mind map from the backend generation service. These are assembled under the root node and transformed into a hierarchical structure using existing utility functions such as `traverseMindmap()`, `buildHierarchy()`, and `applyTreeLayout()`. The final mind map is enriched with matched images and rendered for user interaction.

Figure 41 illustrates the combined logic of the `fetchPad` and `processSection` functions, showing how the document is retrieved, and each section's content is recursively cleaned and prepared for mind map construction. Figure 42 displays the implementation logic of `buildFullMindmap()`, demonstrating how section-level maps are aggregated into a comprehensive document-level mind map.



```

const processSection = (section) => {
  let combinedText = "";
  if (section.content && section.content.ops) {
    combinedText += extractTextFromOps(section.content.ops);
  }
  if (section.subsections && Array.isArray(section.subsections)) {
    section.subsections.forEach(subSection => {
      combinedText += " " + processSection(subSection);
    });
  }
  return cleanText(combinedText);
};

const fetchPad = async () => {
  if (hasFetchedPad.current) return;
  hasFetchedPad.current = true;

  const token = localStorage.getItem("token");
  if (!token || !_id) {
    navigate("/mindmap");
    return;
  }

  try {
    const res = await fetch(
      `${process.env.REACT_APP_BACKEND_API_URL}/api/pads/${_id}`,
      { headers: { Authorization: token } }
    );

    if (!res.ok) {
      console.error("✖ Failed to fetch pad:", res.status);
      return;
    }

    const data = await res.json();
    console.log("█ Pad Data:", data);

    // Update state variables
    setPad(data);
    setSections(data.sections || []);
    setAuthors(data.authors || []);
    setReferences(data.references || []);
    setPadName(data.title || "");
    padNameRef.current = data.title || "";
    sectionsRef.current = data.sections || [];

    // Clean and log the pad title
    const cleanTitle = cleanText(data.title || "");
    console.log("Title:", cleanTitle);

    return data;
  } catch (error) {
    console.error("✖ Error fetching pad:", error);
  }
};

```

Figure 41: fetchPad and processSection Functions to Retrieve and Clean Document Content



The image shows a screenshot of a code editor with a dark theme. The title bar says "JS DocumentMindmap.js". The code in the editor is a JavaScript function named `buildFullMindmap`. The function starts with logging the current state of `padNameRef.current` and `sectionsRef.current.length`. It then initializes a `rootNode` with a name of `padNameRef.current` or `"Untitled Pad"` and an empty `subnodes` array. The function then loops through each section in `sectionsRef.current`, processes the section text, and creates a `sectionNode` with a name of the section title and an empty `subnodes` array. It then tries to fetch a mindmap for the section using `fetchMindMap`. If successful, it adds the fetched mindmap to the `sectionNode.subnodes`. If there is an error, it logs the error and adds the `sectionNode` to the `rootNode.subnodes`. After processing all sections, the function converts the full mindmap structure into nodes and links using `traverseMindmap`. It then builds a hierarchy and applies a tree layout. Finally, it waits for `fetchMatchedImages` to complete, updating the state and logging that the full mindmap is built. If there is an error fetching images, it logs the error. The code uses several utility functions like `processSection`, `buildHierarchy`, `applyTreeLayout`, and `setLoading`.

```
async function buildFullMindmap() {
  console.log("Full Mind map building....");
  console.log("padNameRef.current:", padNameRef.current);
  console.log("sectionsRef.current.length:", sectionsRef.current.length);

  hasFetchedDatabaseRef.current = true;

  // Build the root node using the pad title
  const rootNode = {
    name: padNameRef.current || "Untitled Pad",
    subnodes: []
  };

  // Loop through each section to build its node
  for (const section of sectionsRef.current) {
    const sectionText = processSection(section);
    const sectionNode = {
      name: section.title || sectionText.slice(0, 50),
      subnodes: []
    };

    try {
      const mindmapResponse = await fetchMindMap(sectionText);
      if (mindmapResponse && mindmapResponse.mindmap) {
        sectionNode.subnodes = mindmapResponse.mindmap;
      }
    } catch (error) {
      console.error("Error fetching mindmap for section:", section.title, error);
    }
    rootNode.subnodes.push(sectionNode);
  }

  // Convert the full mindmap structure into nodes and links
  nodes = [];
  links = [];
  traverseMindmap(rootNode, null);

  const hierarchyData = buildHierarchy(nodes, links);
  applyTreeLayout(hierarchyData);

  await fetchMatchedImages(nodes, imageCatalog)
    .then(() => update())
    .then(() => {
      console.log("Full Mind map built.");
      setLoading(false);
    })
    .catch(err => console.error("Error fetching matched images while building full mindmap:", err));
  }
}
```

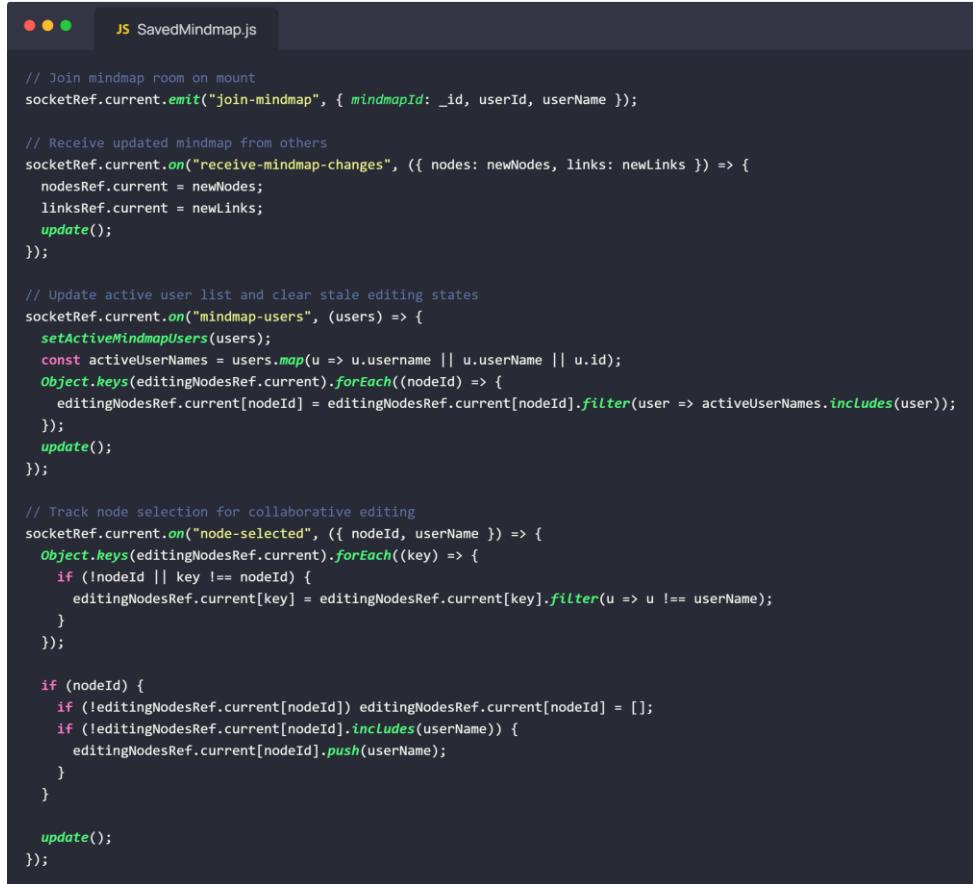
Figure 42: `buildFullMindmap` Function to Generate Document-Level Mind Map

Collaborative synchronization and real-time updates

To support collaborative editing of mind maps in real time, the frontend includes features that communicate directly with the Node.js backend via WebSockets and synchronize updates across users.

- `socketRef` maintains a persistent WebSocket connection with the backend server. It is initialized using `useRef()` and set up within a `useEffect()` hook to ensure that it connects when the component mounts. The socket is used to emit and listen for events such as `join-mindmap`, `node-selected`, and `update-mindmap`. When a user joins a session, they are added to a shared room identified by a unique mind map ID. All interactions, including structure changes and node selections, are broadcasted to other users in the same session. This enables seamless, real-time collaboration with visible user activity indicators.

Figure 43 displays the WebSocket initialization and event handling logic implemented using `socketRef`, demonstrating how the frontend connects to and listens for collaboration-related updates.



```
JS SavedMindmap.js

// Join mindmap room on mount
socketRef.current.emit("join-mindmap", { mindmapId: _id, userId, userName });

// Receive updated mindmap from others
socketRef.current.on("receive-mindmap-changes", ({ nodes: newNodes, links: newLinks }) => {
  nodesRef.current = newNodes;
  linksRef.current = newLinks;
  update();
});

// Update active user list and clear stale editing states
socketRef.current.on("mindmap-users", (users) => {
  setActiveMindmapUsers(users);
  const activeUserNames = users.map(u => u.username || u.userName || u.id);
  Object.keys(editingNodesRef.current).forEach((nodeId) => {
    editingNodesRef.current[nodeId] = editingNodesRef.current[nodeId].filter(user => activeUserNames.includes(user));
  });
  update();
});

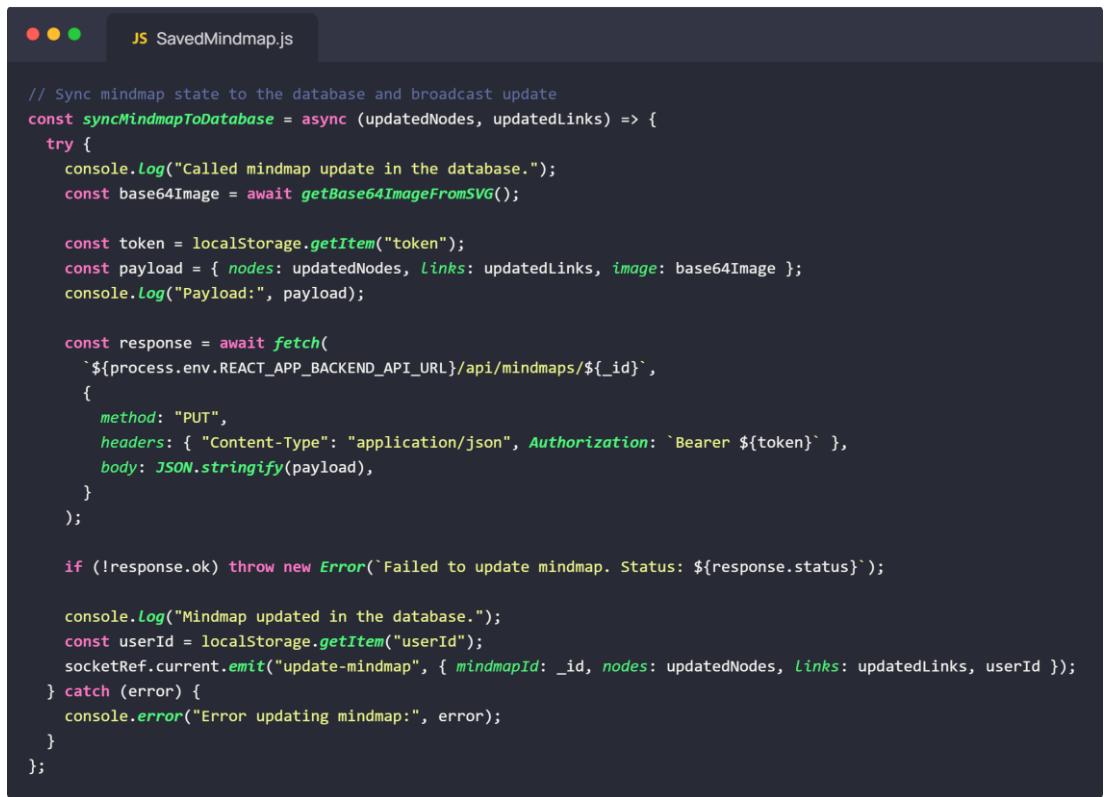
// Track node selection for collaborative editing
socketRef.current.on("node-selected", ({ nodeId, userName }) => {
  Object.keys(editingNodesRef.current).forEach((key) => {
    if (!nodeId || key !== nodeId) {
      editingNodesRef.current[key] = editingNodesRef.current[key].filter(u => u !== userName);
    }
  });

  if (nodeId) {
    if (!editingNodesRef.current[nodeId]) editingNodesRef.current[nodeId] = [];
    if (!editingNodesRef.current[nodeId].includes(userName)) {
      editingNodesRef.current[nodeId].push(userName);
    }
  }
  update();
});
```

Figure 43: `socketRef` Function to Establish and Manage Real-Time Collaboration

- `syncMindmapToDatabase()` is responsible for updating the current state of the mind map in the database after any major interaction such as adding, deleting, or modifying nodes and links. It extracts the latest structure, prepares the update payload including node and link arrays, and sends an authenticated HTTP PUT request to the backend. After a successful update, it emits an `update-mindmap` event using the socket connection, which broadcasts the changes to all other users connected to the same mind map session, ensuring synchronized collaboration across clients.

Figure 44 illustrates the implementation of the `syncMindmapToDatabase()` function, which processes structural updates and coordinates real-time broadcasting via sockets.



```

● ● ● JS SavedMindmap.js

// Sync mindmap state to the database and broadcast update
const syncMindmapToDatabase = async (updatedNodes, updatedLinks) => {
  try {
    console.log("Called mindmap update in the database.");
    const base64Image = await getBase64ImageFromSVG();

    const token = localStorage.getItem("token");
    const payload = { nodes: updatedNodes, links: updatedLinks, image: base64Image };
    console.log("Payload:", payload);

    const response = await fetch(
      `${process.env.REACT_APP_BACKEND_API_URL}/api/mindmaps/${_id}`,
      {
        method: "PUT",
        headers: { "Content-Type": "application/json", Authorization: `Bearer ${token}` },
        body: JSON.stringify(payload),
      }
    );

    if (!response.ok) throw new Error(`Failed to update mindmap. Status: ${response.status}`);

    console.log("Mindmap updated in the database.");
    const userId = localStorage.getItem("userId");
    socketRef.current.emit("update-mindmap", { mindmapId: _id, nodes: updatedNodes, links: updatedLinks, userId });
  } catch (error) {
    console.error("Error updating mindmap:", error);
  }
};

```

Figure 44: `syncMindmapToDatabase` Function to Persist and Broadcast Mind Map Updates

2.1.5 Testing

Testing is a vital part of ensuring the functionality, stability, and reliability of any system. For this research component, which centers around intelligent mind map generation and real-time collaboration, I have followed a structured testing approach including unit testing, integration testing, system testing, and acceptance testing. These phases have been carried out to assess not only the correctness of individual functionalities but also the smooth interoperability between microservices, frontend logic, and collaborative workflows. The testing process supports the robustness of the mind map generation flow within the broader WriteWizard platform.

Unit testing

- Unit testing forms the foundation of the validation process by verifying individual logic units in isolation. This included both shared utilities and core functions of the mind map generation microservice.
- Specific unit tests were written for backend logic such as emoji removal, text preprocessing, incomplete JSON correction, and token overflow handling to ensure input integrity before model inference.
- In addition to mind map-specific logic, API route behavior and expected data structures were independently verified using FastAPI test clients and mock data inputs.
- These unit tests help ensure that core logic remains reliable across updates and that inputs and outputs conform to expected formats throughout the platform.

Integration testing

- Integration testing validated communication and data exchange between different backend services used by the WriteWizard application.
- Key focus was placed on the mind map generation microservice, where endpoints /generate, /extend, and /simplify were tested using structured payloads to confirm consistent, hierarchical JSON responses.
- These tests confirmed that the fine-tuned model responded accurately under realistic academic inputs and integrated properly with other frontend components.

- Additional tests covered API interoperability with the Node.js backend and image matching service to ensure end-to-end content and metadata synchronization.
- Postman was used to manually test all major backend endpoints, validating request handling, response status, and error management for typical and edge-case scenarios.

System testing

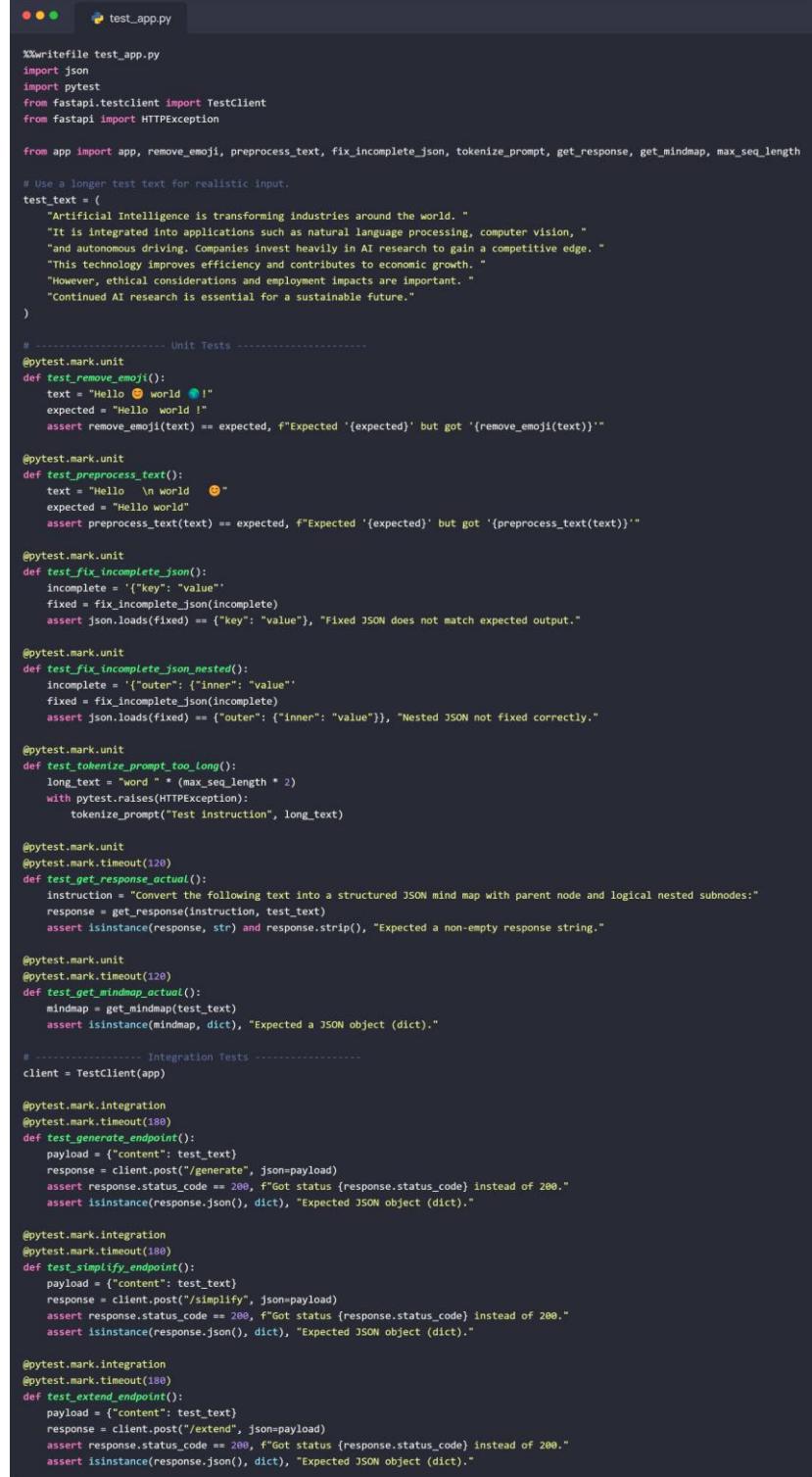
- System testing evaluated the full behavior of the platform in simulated real-world usage scenarios, ensuring that all components performed seamlessly together.
- From frontend request initiation to backend response handling and collaborative editing, this phase assessed the flow of mind map generation, editing, and export.
- Visual alignment, layout responsiveness, semantic accuracy of mind map content, and real-time updates through socket communication were evaluated across different browsers and screen resolutions.
- This stage ensured mind map functionality integrated well into the wider application, maintaining performance, usability, and feature completeness across modules.

Acceptance testing

- Acceptance testing was conducted to confirm that the developed functionality aligned with user expectations and academic use cases.
- Alpha testing within a controlled peer environment allowed for early-stage feedback on usability, layout design, and mind map relevance.
- Beta testing extended to users in their own environments helped validate real-world applicability, visual clarity, and platform performance.
- Overall, acceptance testing refined the mind map interface and validated WriteWizard's ability to serve its academic audience effectively.

By following a comprehensive testing process, key steps were taken to validate the correctness, compatibility, and user readiness of the mind map generation and collaboration module. The microservice was tested using Pytest and FastAPI's

TestClient, with unit tests covering preprocessing logic such as emoji removal, JSON correction, and token handling. Integration tests verified the /generate, /extend, and /simplify endpoints using academic inputs. Tests included assertions and timeouts to simulate real-world conditions. Figure 45 shows the test suite validating model interaction and structure, while Figure 46 presents successful execution results.



```

%%writefile test_app.py
import json
import pytest
from fastapi.testclient import TestClient
from fastapi import HTTPException

from app import app, remove_emoji, preprocess_text, fix_incomplete_json, tokenize_prompt, get_response, get_mindmap, max_seq_length

# Use a longer test text for realistic input.
test_text = (
    "Artificial Intelligence is transforming industries around the world. "
    "It is integrated into applications such as natural language processing, computer vision, "
    "and autonomous driving. Companies invest heavily in AI research to gain a competitive edge. "
    "This technology improves efficiency and contributes to economic growth. "
    "However, ethical considerations and employment impacts are important. "
    "Continued AI research is essential for a sustainable future."
)

# ----- Unit Tests -----
@pytest.mark.unit
def test_remove_emoji():
    text = "Hello 😊 world 😊 !"
    expected = "Hello world !"
    assert remove_emoji(text) == expected, f"Expected '{expected}' but got '{remove_emoji(text)}'"

@pytest.mark.unit
def test_preprocess_text():
    text = "Hello \n world 😊"
    expected = "Hello world"
    assert preprocess_text(text) == expected, f"Expected '{expected}' but got '{preprocess_text(text)}'"

@pytest.mark.unit
def test_fix_incomplete_json():
    incomplete = {"key": "value"}
    fixed = fix_incomplete_json(incomplete)
    assert json.loads(fixed) == {"key": "value"}, "Fixed JSON does not match expected output."

@pytest.mark.unit
def test_fix_incomplete_json_nested():
    incomplete = {"outer": {"inner": "value"}}
    fixed = fix_incomplete_json(incomplete)
    assert json.loads(fixed) == {"outer": {"inner": "value"}}, "Nested JSON not fixed correctly."

@pytest.mark.unit
def test_tokenize_prompt_too_long():
    long_text = "word " * (max_seq_length * 2)
    with pytest.raises(HTTPException):
        tokenize_prompt("Test instruction", long_text)

@pytest.mark.unit
@pytest.mark.timeout(120)
def test_get_response_actual():
    instruction = "Convert the following text into a structured JSON mind map with parent node and logical nested subnodes:"
    response = get_response(instruction, test_text)
    assert isinstance(response, str) and response.strip(), "Expected a non-empty response string."

@pytest.mark.unit
@pytest.mark.timeout(120)
def test_get_mindmap_actual():
    mindmap = get_mindmap(test_text)
    assert isinstance(mindmap, dict), "Expected a JSON object (dict)."

# ----- Integration Tests -----
client = TestClient(app)

@pytest.mark.integration
@pytest.mark.timeout(180)
def test_generate_endpoint():
    payload = {"content": test_text}
    response = client.post("/generate", json=payload)
    assert response.status_code == 200, f"Got status {response.status_code} instead of 200."
    assert isinstance(response.json(), dict), "Expected JSON object (dict)."

@pytest.mark.integration
@pytest.mark.timeout(180)
def test_simplify_endpoint():
    payload = {"content": test_text}
    response = client.post("/simplify", json=payload)
    assert response.status_code == 200, f"Got status {response.status_code} instead of 200."
    assert isinstance(response.json(), dict), "Expected JSON object (dict)."

@pytest.mark.integration
@pytest.mark.timeout(180)
def test_extend_endpoint():
    payload = {"content": test_text}
    response = client.post("/extend", json=payload)
    assert response.status_code == 200, f"Got status {response.status_code} instead of 200."
    assert isinstance(response.json(), dict), "Expected JSON object (dict)."

```

Figure 45: Unit and Integration Test Suite for Mind Map Generation Microservice

```
!pytest test_app.py -m unit --maxfail=1 --disable-warnings
=====
platform linux -- Python 3.11.11, pytest-8.3.5, pluggy-1.5.0
rootdir: /content
plugins: anyio-4.9.0, typeguard-4.4.2, langsmith-0.3.23
collected 10 items / 3 deselected / 7 selected

test_app.py ..... [100%]

===== 7 passed, 3 deselected, 16 warnings in 95.54s (0:01:35) =====

!pytest test_app.py -m integration --maxfail=1 --disable-warnings
=====
platform linux -- Python 3.11.11, pytest-8.3.5, pluggy-1.5.0
rootdir: /content
plugins: anyio-4.9.0, typeguard-4.4.2, langsmith-0.3.23
collected 10 items / 7 deselected / 3 selected

test_app.py ... [100%]

===== 3 passed, 7 deselected, 15 warnings in 123.29s (0:02:03) =====
```

Figure 46: Pytest Execution Results for Unit and Integration Tests

Manual testing:

Manual testing is an essential process in validating the application's usability and core functionalities from a user's perspective. Unlike automated testing, it relies on human interaction to assess system behavior, identify issues, and ensure features work as intended. For this research, manual testing focused on validating the mind map generation workflow, image integration, collaboration interface, and export functionalities within the WriteWizard platform.

Objectives and Strategy

- Manual testing was used to ensure proper system behavior across typical user flows such as generating, modifying, exporting, and saving mind maps.
- It allowed for real-time feedback based on observed output, performance, and edge case handling across user-triggered operations.

Test Case Development

- Test cases were designed based on system specifications and common user scenarios, particularly those involving the React frontend and backend microservices.
- Each test case included inputs, steps, expected outcomes, and observed results to maintain consistency and traceability.

Execution and Validation

- Tests were carried out across various devices and browsers to evaluate layout responsiveness, functionality, and image alignment.
- Actions were manually performed, including node addition, triggering backend requests, socket-based updates, and verifying data persistence.

Types of Manual Testing

- Functional Testing: Verified the expected outcome of actions like generating mind maps, saving visualizations, or interacting with nodes.
- Non-Functional Testing: Assessed responsiveness, image rendering, and the fluidity of real-time collaboration.

- User Acceptance Testing (UAT): Conducted with academic peers and instructors to validate the usability and accuracy of the mind map outputs.
- Exploratory Testing: Performed to assess how the component behaved with varied user inputs, long documents, and unexpected interactions.
- Compatibility Testing: Verified correct operation across major browsers including Chrome, Firefox, and Edge at different screen sizes.

Tables 4 through 15 summarize the manual test cases that were executed to ensure system functionality and user satisfaction.

Table 4: Test Case for Standard Mind Map Generation

Test Case ID	Tc_01
Test Case Objective	Test Standard Mind Map Generation from A Document Section
Pre-Requirements	User Must Be Logged into Writewizard
Test Steps	<ol style="list-style-type: none"> 1. Navigate To a Pad 2. Select a Document Section 3. Click “Generate Mind Map”
Test Data	Paragraph of Academic Content
Expected Output	Mind Map Rendered with A Structured Hierarchy of Nodes
Actual Output	Mind Map Displayed with Correct Structure
Status	Pass

Table 5: Test Case for Extended Mind Map Generation

Test Case Id	Tc_02
Test Case Objective	Test Extended Mind Map Generation from Input Text
Pre-Requirements	User Must Be Logged into Writewizard
Test Steps	<ol style="list-style-type: none"> 1. Navigate To Mind Map Dashboard 2. Input Text

	3. Click “Generate Mind Map” 4. Click “Extend”
Test Data	Paragraph Of Academic Content
Expected Output	Mind Map Generated with Additional Subtopics And Depth Based on the Input
Actual Output	Hierarchical Mind Map with Expanded Structure Rendered
Status	Pass

Table 6: Test Case for Simplified Mind Map Generation

Test Case ID	Tc_03
Test Case Objective	Test Simplified Mind Map Generation From Selected Or Input Text
Pre-Requirements	User Must Be Logged into Writewizard
Test Steps	1. Navigate to Mind Map Dashboard 2. Input Text 3. Click “Generate Mind Map” 4. Click “Simplify”
Test Data	Paragraph Of Academic Content
Expected Output	Mind Map Generated with Fewer, Broader Nodes Summarizing the Input
Actual Output	Simplified Mind Map Rendered Correctly
Status	Pass

Table 7: Test Case for Full Document Mind Map Generation

Test Case ID	TC_04
Test Case Objective	Test Mind Map Generation for a Full Document (Pad)
Pre-Requirements	User Must Be Logged into Writewizard

Test Steps	1. Navigate To Mind Map Dashboard 2. Select a Pad 3. Click “Generate”
Test Data	Complete Academic Document Stored in the Selected Pad
Expected Output	Multi-Level Mind Map Generated Using All Section Texts
Actual Output	Full Document Mind Map Created with Accurate Hierarchy
Status	Pass

Table 8: Test Case for Image Matching with Nodes

Test Case ID	TC_05
Test Case Objective	Test Automatic Image Matching For Nodes
Pre-Requirements	User Must Be Logged into Writewizard and Pad Must Contain Images with Descriptions
Test Steps	1. Navigate to a Pad 2. Select a Document Section 2. Click “Generate Mind Map”
Test Data	Paragraph of Academic Content
Expected Output	Relevant Images Automatically Assigned to Corresponding Nodes
Actual Output	Semantic Image Matching Successful
Status	Pass

Table 9: Test Case for Real-Time Collaboration

Test Case ID	TC_06
--------------	-------

Test Case Objective	Test Real-Time Collaboration for Joining Session and Node Selection Feedback
Pre-Requirements	Two Users Logged into Writewizard and Have Access to the Same Mind Map
Test Steps	<ol style="list-style-type: none"> 1. User A And B Navigate to Mind Map Dashboard 2. User A And B Open the Same Mind Map 3. User A Selects a Node
Test Data	Shared Mind Map ID
Expected Output	Both Users View Real-Time Updates and See Node Selection Highlights
Actual Output	Collaborative Session Established and Node Highlights Visible
Status	Pass

Table 10: Test Case for Node Addition and Deletion

Test Case ID	TC_07
Test Case Objective	Test Addition and Deletion of Nodes Within a Mind Map
Pre-Requirements	User Must Be Logged into Writewizard and Inside an Editable Mind Map
Test Steps	<ol style="list-style-type: none"> 1. Click “Add Node” 2. Enter Node Name 3. Select And Delete a Node
Test Data	Custom Node Text Input
Expected Output	Node Added and Deleted with Updates Reflected in the Layout
Actual Output	Node Added and Deleted Successfully
Status	Pass

Table 11: Test Case for Relation Addition and Deletion

Test Case ID	TC_08
Test Case Objective	Test Addition and Deletion of Relations (Links) between Nodes
Pre-Requirements	User Must Be Logged into Writewizard and Inside an Editable Mind Map With At least Two Nodes
Test Steps	<ol style="list-style-type: none"> 1. Select Source Node 2. Click “Add Relation” 3. Select Target Node 4. Delete relation
Test Data	IDs of Source and Target Nodes
Expected Output	Relation Added between Nodes and Removed on Delete with Updates Reflected in the Layout
Actual Output	Relation Added and Deleted as Expected
Status	Pass

Table 12: Test Case for Saving Mind Map

Test Case ID	TC_09
Test Case Objective	Test Saving of the Mind Map to the Database
Pre-Requirements	User Must Be Logged into Writewizard with an Editable Mind Map
Test Steps	<ol style="list-style-type: none"> 1. Click “Save Map” Button
Test Data	Node and Link Data Structure
Expected Output	Mind Map Saved in the Backend Database
Actual Output	Data Saved Successfully with Confirmation
Status	Pass

Table 13: Test Case for Exporting as PNG

Test Case ID	TC_10
Test Case Objective	Test export functionality of mind map as PNG image
Pre-Requirements	User Must Be Logged into Writewizard with an Editable Mind Map
Test Steps	<ol style="list-style-type: none"> 1. Click “Download” 2. Select PNG Format
Test Data	Current Mind Map SVG Rendering
Expected Output	PNG Image Downloaded to User's Device
Actual Output	PNG Exported and Downloaded Successfully
Status	Pass

Table 14: Test Case for Responsiveness on Mobile Devices

Test Case ID	TC_11
Test Case Objective	Test Application Responsiveness and Usability on Mobile Devices
Pre-Requirements	Access WriteWizard on a Mobile or Tablet Device
Test Steps	<ol style="list-style-type: none"> 1. Open Application in Mobile Browser 2. Interact with Mind Map
Test Data	Touch-Based Interactions on Mobile Interface
Expected Output	Layout Adjusts Correctly, All Actions are Accessible and Functional
Actual Output	Mobile Layout Responsive with Complete Functionality
Status	Pass

Table 15: Test Case for Invalid Input Handling

Test Case ID	TC_12
--------------	-------

Test Case Objective	Test the System's Response to Invalid or Empty Input During Mind Map Generation
Pre-Requirements	User Must Be Logged into Writewizard
Test Steps	<ul style="list-style-type: none"> 1. Navigate to the Mind Map Dashboard 2. Leave the input field empty or enter special characters only 3. Click “Generate Mind Map”
Test Data	Empty String or Special Characters (e.g., ###, !!, @)
Expected Output	Warning Message Displayed Indicating Invalid Input; No API Call Made
Actual Output	Warning Message Appeared and Generation was Prevented
Status	Pass

The manual testing process effectively validated both the functional and non-functional aspects of the mind map generation and collaboration component. All test cases passed successfully, confirming the reliability, usability, and cross-platform compatibility of the solution. The results demonstrate that the system performs as intended across a variety of user scenarios. Although testing outcomes were positive overall, minor latency was observed when rendering large mind maps on older mobile devices, and certain limitations related to resource handling were identified. These observations have been documented for future optimization and system enhancement.

2.1.6 Deployment and maintenance

This section outlines the deployment procedures followed for the core services of the system, including the mind map generation microservice, image matching service, and the collaborative document editor application. These components were deployed on virtual cloud infrastructure with a focus on performance, scalability, and integration. While deeper architectural details are covered in the team report, this section highlights the deployment steps relevant to the core research contribution.

Deployment of mind map generation microservice

The mind map generation service, central to this research, was deployed using a Microsoft Azure Virtual Machine configured with 4 vCPUs and 32 GiB RAM (Standard E4as v4). Due to Azure for Students plan limitations, GPU resources were unavailable; hence, a CPU-based configuration was selected and optimized for lightweight transformer-based inference.

The service was developed using FastAPI and integrates a LoRA-finetuned Mistral 7B v0.3 model hosted privately on Hugging Face. It processes academic text input, formats prompt, calls the transformer model, and returns structured mind map data in JSON format.

1. Azure Account and VM Setup

An Azure for Students account was used to provision virtual infrastructure. A virtual machine running Ubuntu 24.04 was created from the Azure portal, configured with open inbound traffic on port 8000 to allow external access to the API.

Figure 47 shows the basic VM configuration settings used during deployment.

Basics	
Subscription	Azure for Students
Resource group	(new) Mindmap_Model
Virtual machine name	MistralModelVM
Region	Central India
Availability options	Availability zone
Zone options	Self-selected zone
Availability zone	1
Security type	Trusted launch virtual machines
Enable secure boot	Yes
Enable vTPM	Yes
Integrity monitoring	No
Image	Ubuntu Server 24.04 LTS - Gen2
VM architecture	x64
Size	Standard E4ds v4 (4 vcpus, 32 GiB memory)
Enable Hibernation	No
Authentication type	SSH public key
Username	azureuser
SSH Key format	RSA
Key pair name	MistralModelVM_key
Public inbound ports	SSH
Azure Spot	No

Figure 47: Azure VM Configuration for Mind Map Generation Service

2. Environment Configuration

After SSH access was established, the system was updated, and Python 3.12 was installed. A virtual environment was created, and essential packages such as transformers, torch, fastapi, and uvicorn were installed to support model loading and API functionality.

3. Hugging Face Model Integration

The LoRA-finetuned model was stored in a private Hugging Face repository. It was dynamically loaded during FastAPI startup using the Transformers library. Access was authorized using a private token securely stored in a .env file.

4. Service Exposure and Testing

The FastAPI service was started using Uvicorn on port 8000. To make it publicly accessible, Nginx was configured as a reverse proxy to route HTTP traffic on port 80 to the FastAPI backend. API endpoints such as /generate, /extend, and /simplify were tested using Postman to verify their functionality.

Figure 48 presents the Nginx configuration used to route traffic to the local application, Figure 49 shows the FastAPI application successfully running on the Azure VM, loading the LoRA-finetuned model and serving the Uvicorn server on port 8000, and Figure 50 displays the Postman output confirming a successful API response from the deployed service.

```
server {
    listen 80;
    server_name 20.193.154.174;

    location / {
        proxy_pass http://127.0.0.1:8000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;

        proxy_connect_timeout 3600;
        proxy_send_timeout 3600;
        proxy_read_timeout 3600;
        send_timeout 3600;
    }
}
```

Figure 48: Nginx Configuration for Routing FastAPI Endpoints

```
(venv) azuser@nvidiaModelVM:~$ uvicorn main:app --host 127.0.0.1 --port 8000
/home/azuser/venv/lib/python3.12/site-packages/transformers/utils/hub.py:105: FutureWarning: Using 'TRANSFORMERS_CACHE' is deprecated and will be removed in v5 of Transformers. Use 'HF_HOME' instead.
  warnings.warn(
Loading base model on CPU (Float32)...
/home/azuser/venv/lib/python3.12/site-packages/transformers/models/auto/auto_factory.py:476: FutureWarning: The 'use_auth_token' argument is deprecated and will be removed in v5 of Transformers. Please use 'token' instead.
  warnings.warn(
Loading checkpoint shards: 100%|██████████| 3/3 [00:57<00:00, 19.32s/it]
Loading tokenizer...
/home/azuser/venv/lib/python3.12/site-packages/transformers/models/auto/tokenization_auto.py:897: FutureWarning: The 'use_auth_token' argument is deprecated and will be removed in v5 of Transformers. Please use 'token' instead.
  warnings.warn(
Attaching LoRA adapter...
Merging LoRA adapter into the model...
Model loaded on CPU.
INFO:     Started server process [20655]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     112.134.140.243:0 - "GET / HTTP/1.0" 404 Not Found
INFO:     112.134.140.243:0 - "GET /favicon.ico HTTP/1.0" 404 Not Found
INFO:     Setting `pad_token_id` to `eos_token_id` for open-end generation.
INFO:     112.134.140.243:0 - "POST /generate HTTP/1.0" 200 OK
```

Figure 49: FastAPI Application Running with LoRA Model on Azure VM

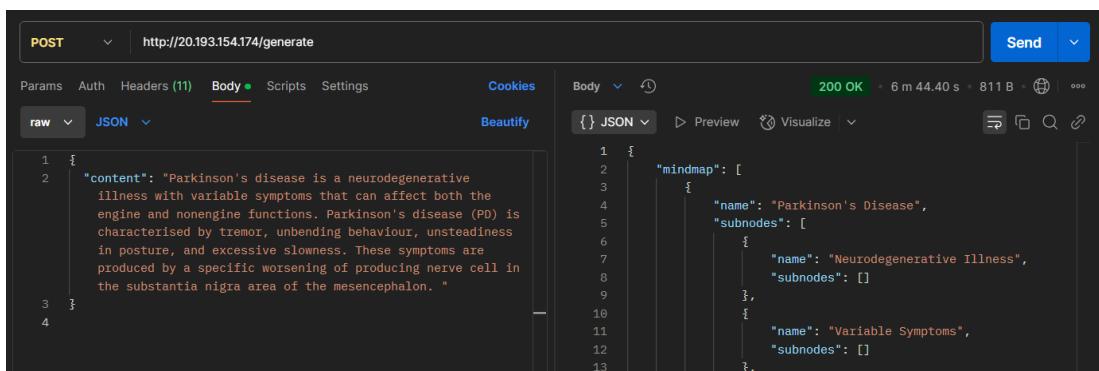


Figure 50: Postman Verification of Public Mind Map Generation Endpoint

5. Security and Monitoring

Sensitive environment variables were isolated from code. Nginx configurations were optimized to handle model processing time, and tools like htop were used to monitor CPU and memory usage, ensuring stable operation even under resource constraints.

Deployment of image matching service

To handle automatic semantic image integration in mind maps, a lightweight image matching microservice was deployed using Hugging Face Spaces. The service runs in a free-tier containerized environment with 2 vCPUs and 16 GiB RAM, sufficient for fast and responsive text-to-image matching.

The application, developed in FastAPI, accepts node titles as input and returns matched image URLs from a pre-indexed image catalog.

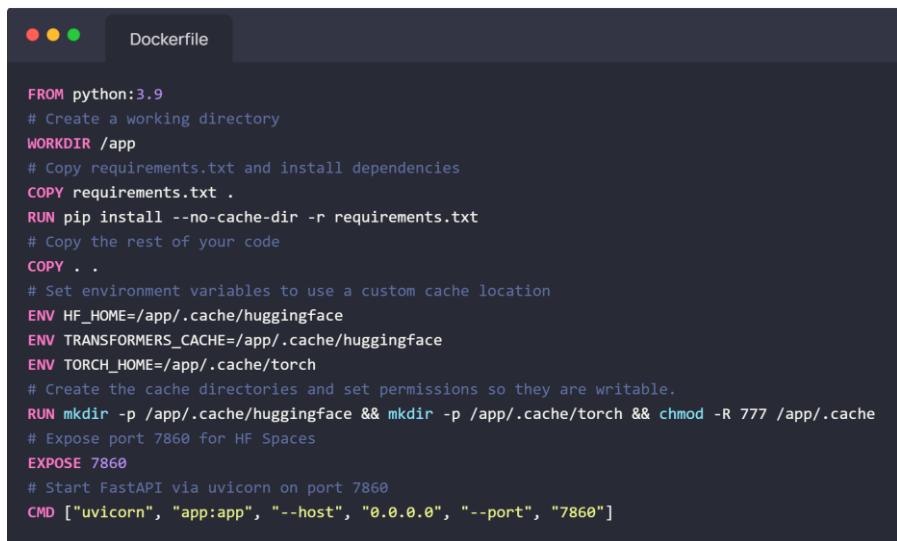
1. Platform Setup

A Hugging Face Space named Sanjayan201/my-image-matching-app was created using Docker as the backend configuration. This allowed the deployment of a full Python web application within a controlled environment managed by Hugging Face.

2. Containerization

A custom Dockerfile was written to specify the base image, install necessary dependencies, and run the FastAPI application.

Figure 51 displays the Dockerfile used for container creation.



```
FROM python:3.9
# Create a working directory
WORKDIR /app
# Copy requirements.txt and install dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
# Copy the rest of your code
COPY .
# Set environment variables to use a custom cache location
ENV HF_HOME=/app/.cache/huggingface
ENV TRANSFORMERS_CACHE=/app/.cache/huggingface
ENV TORCH_HOME=/app/.cache/torch
# Create the cache directories and set permissions so they are writable.
RUN mkdir -p /app/.cache/huggingface && mkdir -p /app/.cache/torch && chmod -R 777 /app/.cache
# Expose port 7860 for HF Spaces
EXPOSE 7860
# Start FastAPI via uvicorn on port 7860
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "7860"]
```

Figure 51: Dockerfile Used for Hugging Face Image Matching Service

3. Service Launch and Endpoint Exposure

Once deployed, the FastAPI server was automatically launched using Uvicorn. The service's main route /match-image became publicly accessible, allowing frontend components to send requests and receive matched image responses.

4. Testing and Performance

Functionality was verified using Postman and the integrated Hugging Face interface. The service handled requests efficiently, returning relevant matches based on input node text.

5. Updates and Logging

Logs were managed by the Hugging Face platform and monitored for request activity and error handling. Updates to the service could be made by modifying the Dockerfile and pushing changes to the space repository.

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.

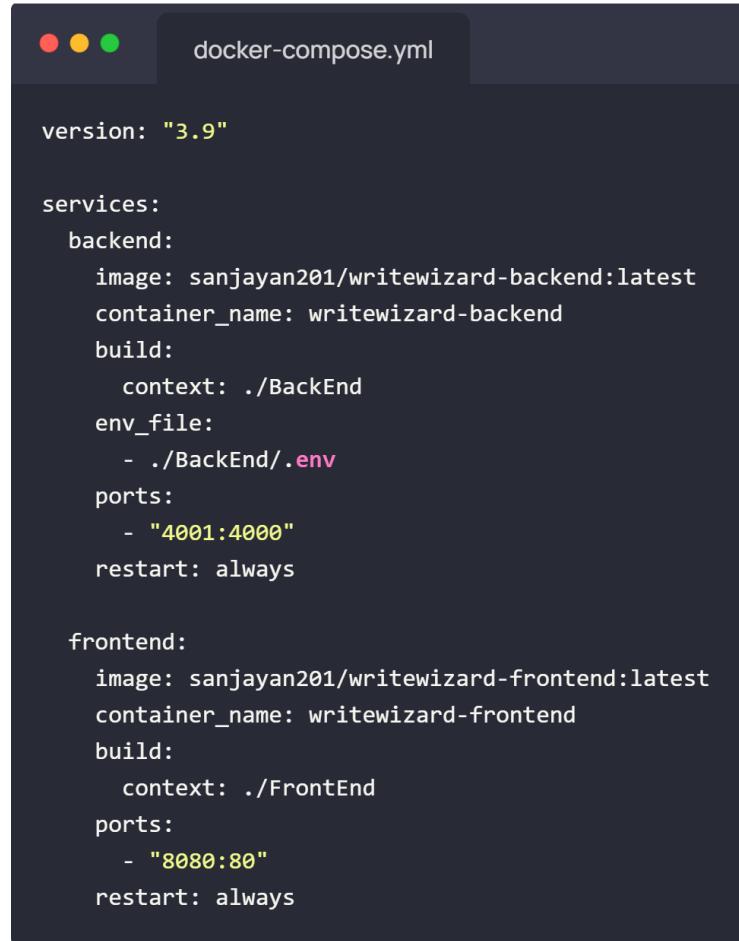
1. Azure VM Setup

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



```
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 52: 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 53 shows the frontend application successfully running on the browser through Nginx routing, and Figure 54 shows the backend API responding correctly after reverse proxy configuration.

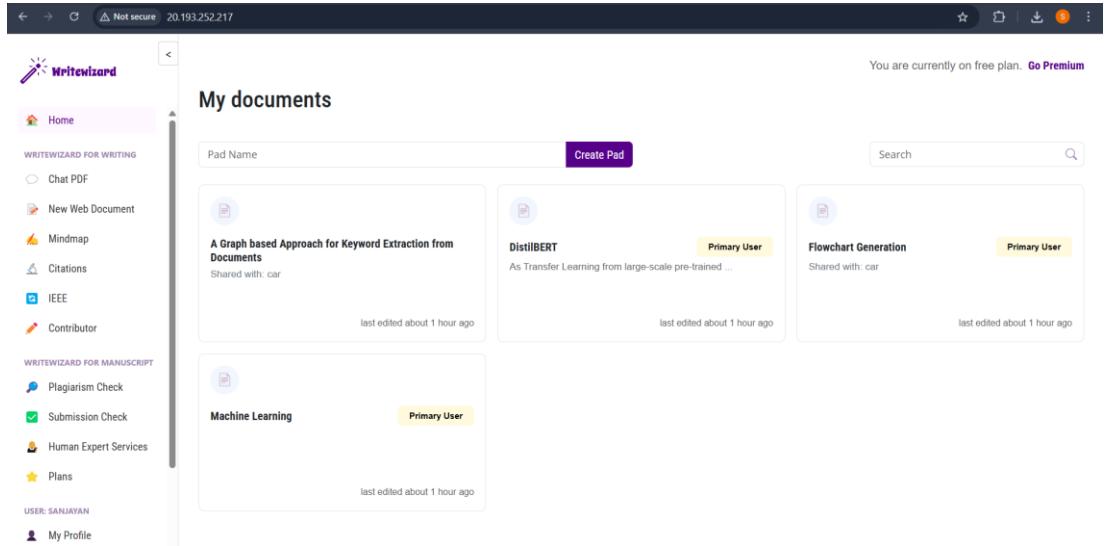


Figure 53: React Frontend of WriteWizard Running via Nginx Routing

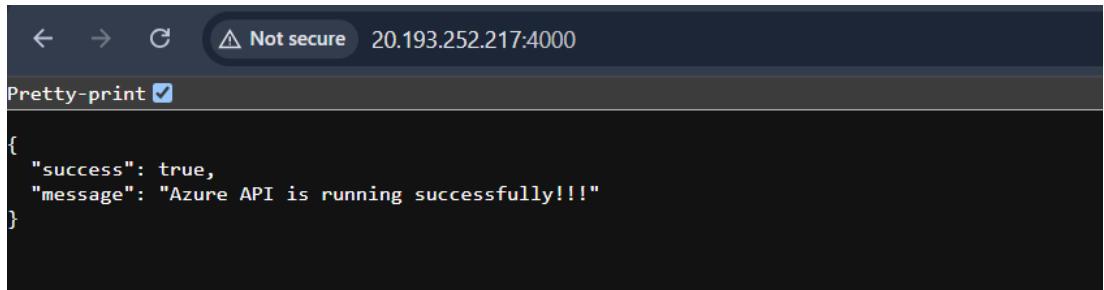


Figure 54: Express.js Backend API Response After Nginx Proxy Configuration

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



```

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 55: 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 commits to the main branch, reducing manual effort and increasing reliability.

Maintenance

Following deployment, the platform underwent continuous maintenance to ensure stable and efficient operation. The focus was on monitoring system behavior, resolving issues, and keeping services up to date. Key maintenance activities included:

- **Log Monitoring:** Logs from Unicorn, Nginx, and FastAPI were routinely monitored to detect unusual behavior or request failures.
- **Environment Updates:** Package dependencies, Docker containers, and system tools were updated regularly to patch vulnerabilities and ensure compatibility.
- **Bug Fixes:** Minor functional issues reported during testing and early usage were addressed, such as socket event synchronization and request-response handling bugs.
- **Uptime Monitoring:** While external monitoring services were not enabled, manual uptime checks were conducted for all endpoints using terminal tools and Postman and resource metrics were manually tracked using htop, terminal logs, and Hugging Face dashboards.
- **Update Pipeline:** Codebase changes triggered Docker rebuilds and redeployments through GitHub Actions, enabling seamless updates with minimal downtime or disruption.
- **Security Patching and Environment Isolation:** Environment variables and access tokens were managed securely across services. System-level patches were applied as released.

This structured approach to maintenance ensured that all services in the WriteWizard ecosystem remained operational, responsive, and aligned with evolving user needs during development and testing phases.

2.2 Commercialization

The WriteWizard collaborative document editing platform, with its AI-powered mind map generation capabilities and real-time collaboration features, presents several pathways for commercialization within educational and productivity markets. This section outlines a viable commercialization strategy built around a tiered subscription model, modular service offerings, and integration potential with academic and enterprise platforms.

1. Value Proposition

At the core of this platform lies the mind map generation microservice, developed using a fine-tuned transformer model optimized with LoRA. This component is capable of producing hierarchical mind maps from academic content through three distinct modes: standard, extended, and simplified. These outputs are further enriched with semantically matched images, delivering a dynamic and visually coherent learning tool.

This feature positions the platform as a valuable resource for:

- Students: Enhancing understanding by converting dense information into visual formats
- Educators: Simplifying the creation of teaching materials and visual lesson planning
- Institutions: Supporting digital content delivery, revision workflows, and curriculum planning

2. Revenue Model

A freemium subscription model is proposed, offering accessibility for casual users and scalability for institutional adoption.

Free Tier

- Limited access to standard mind map generation
- View-only collaboration features
- Exporting with watermark or resolution limitations

Premium Tier (Monthly or Yearly)

- Full access to all mind map generation modes (standard, extended, simplified)
- Unlimited uploads and editing privileges
- Access to image matching services
- Collaborative features including live editing
- High-resolution export options and API priority access

3. Microservice API Licensing

The mind map generation service, being modular by design, can be offered as a standalone API for external integration. Its outputs are structured in JSON, enabling compatibility with other document editors, LMS platforms, or research tools.

- API Monetization based on usage volume (e.g., per request or monthly quotas)
- Developer Access Plans for startups or software tools seeking visualization components
- White-label integration with educational software, content platforms, or AI productivity tools

This model promotes wide-scale adoption of the research component beyond the WriteWizard platform.

4. Payment Gateway and Subscription Management

To manage subscriptions and billing, integration with payment processors such as Stripe or PayPal will support:

- Onboarding, plan upgrades, and renewals
- Usage tracking and account-level billing
- Invoice generation and student discounts through institutional verification

5. Educational Advertising and Partnerships

To support ongoing development and subsidize free-tier access, the platform may host:

- Educational partner content such as university ads or course promotions
- Sponsored templates or study material from publishers and academic brands
- Pilots or licensing agreements with institutions seeking localized deployment

All integrations will comply with academic data privacy and remain non-intrusive to the user experience.

6. Marketing and Outreach Strategy

A focused marketing plan will help build adoption through:

- Search engine optimization for keywords like “AI mind map generator” and “automated learning visualization”
- Social media engagement, sharing use cases, before-and-after mind maps, and student testimonials
- Demo access and onboarding guides for new users or instructors
- Academic outreach through tech fairs, workshops, and campus ambassadorships

7. Feedback Loop and Iteration

To maintain alignment with academic trends and user needs:

- Analytics tools will track common usage patterns and user retention
- Surveys and feedback prompts will identify pain points and missing features
- Iterative updates to mind map modes, image matching accuracy, and export features will follow each cycle

8. Scalability and Future Expansion

Post-launch, the platform and its mind map generation service can be scaled or extended to new verticals:

- Development of mobile apps for quick note-to-map conversion
- AI tutoring integration, where the mind map structure supports personalized question generation
- Multilingual support for global academic audiences
- Enterprise licensing for knowledge mapping, meeting summaries, and corporate training materials

3. RESULTS & DISCUSSION

The core outcome of this research is the development and successful integration of a fine-tuned, lightweight transformer-based microservice capable of generating structured mind maps from academic text. This system was evaluated through both functional validation and subjective user feedback, with particular attention given to its role within the collaborative WriteWizard platform. The results demonstrate the system's potential as a practical academic tool for enhancing content comprehension and visual learning.

Upon successful deployment of the mind map generation microservice, various testing scenarios confirmed that the service could process academic text inputs and generate structured hierarchies across all three generation modes: standard, extended, and simplified. Each mode delivered consistent JSON outputs, enabling smooth rendering on the frontend interface. The standard mode focused on core concepts, the extended mode provided deeper semantic depth, while the simplified mode condensed complex content into broader thematic clusters. This offered flexibility for different learner needs and academic levels. The generated outputs-maintained consistency in structure and label quality, with clear parent-child relationships and relevant node naming across multiple tests.

A significant component of the evaluation involved validating the relevance and interpretability of the generated mind maps. Due to the brainstorming-oriented nature of mind mapping, traditional accuracy metrics were unsuitable. Instead, a subjective evaluation was conducted to assess how well the generated outputs aligned with user expectations and perceived topic relevance. This evaluation followed a forum-based approach where ten unique mind maps generated by the model were presented to a group of student volunteers. Participants rated the clarity, structural quality, and topical accuracy of each mind map. The survey recorded an approval rating exceeding 75 percent across all categories, indicating strong alignment between the generated structures and the participants' understanding of the source content.

Further validation occurred within the WriteWizard environment where the mind map outputs were embedded into a live document collaboration interface. Users were able

to edit, extend, and export the generated visualizations in real time. The integration of semantically matched images into nodes contributed positively to visual clarity and user engagement. Feedback indicated that the enriched visual content helped users identify key ideas more quickly and retain structural relationships better. The frontend visualization rendered complex trees effectively with minimal delay, even for extended outputs, confirming the efficiency of the JSON-to-D3 rendering pipeline.

During exploratory usage, it was observed that users interacted more frequently with extended mind maps for in-depth subjects and preferred simplified ones when revising or reviewing material. This insight supports the inclusion of multiple generation modes and highlights the versatility of the system in adapting to different academic workflows. For shorter inputs, users leaned toward standard or simplified modes, while full document inputs benefited from extended generation, which allowed deeper concept mapping without overwhelming the interface. Additionally, participants reported that having the ability to generate visual summaries directly from document sections reduced the time they typically spent on manual notetaking.

System responsiveness remained stable under practical loads, and the service maintained consistent performance across sessions. While resource constraints imposed some limitations on concurrent processing due to the absence of GPU acceleration, the deployed setup effectively supported the research objectives within its intended scope. All functionalities were validated in a live deployment context and were met with positive user responses during formal and informal walkthroughs. The modular nature of microservice also presents opportunities for reuse or integration into other document editing tools or academic platforms beyond WriteWizard.

In summary, the results affirm the feasibility and educational value of automated mind map generation powered by a fine-tuned language model. The system translates academic content into structured, editable visual formats, enhancing knowledge retention, organization, and collaboration. Forum-based evaluations and user feedback confirm that the component meets its objectives and offers strong potential for educational technology. To illustrate its impact, Figure 56 shows the Mind Map Dashboard with input functionality and saved map access. Figure 57 displays the

generated mind map with interactive options like Extend and Simplify. Figure 58 highlights full document mind map generation with image integration. Figure 59 demonstrates real-time collaborative editing, showing another user actively editing a node within a shared mind map.

Figure 57: Mind Map Dashboard in WriteWizard

Figure 56: Map Modal Displaying Generated Mind Map with Customization Features

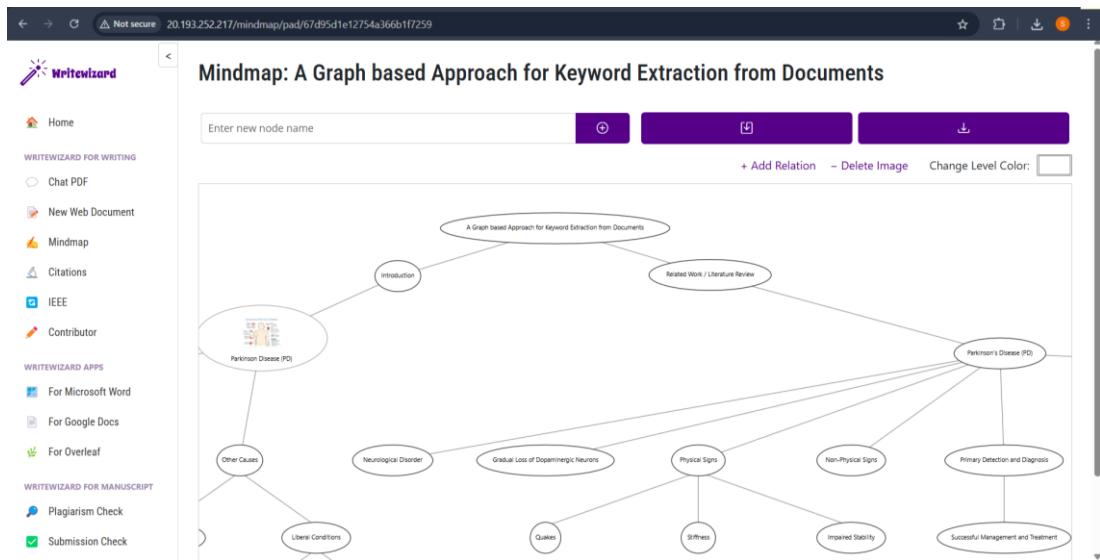


Figure 58: Document-Level Mind Map Generation Interface with Customization Features

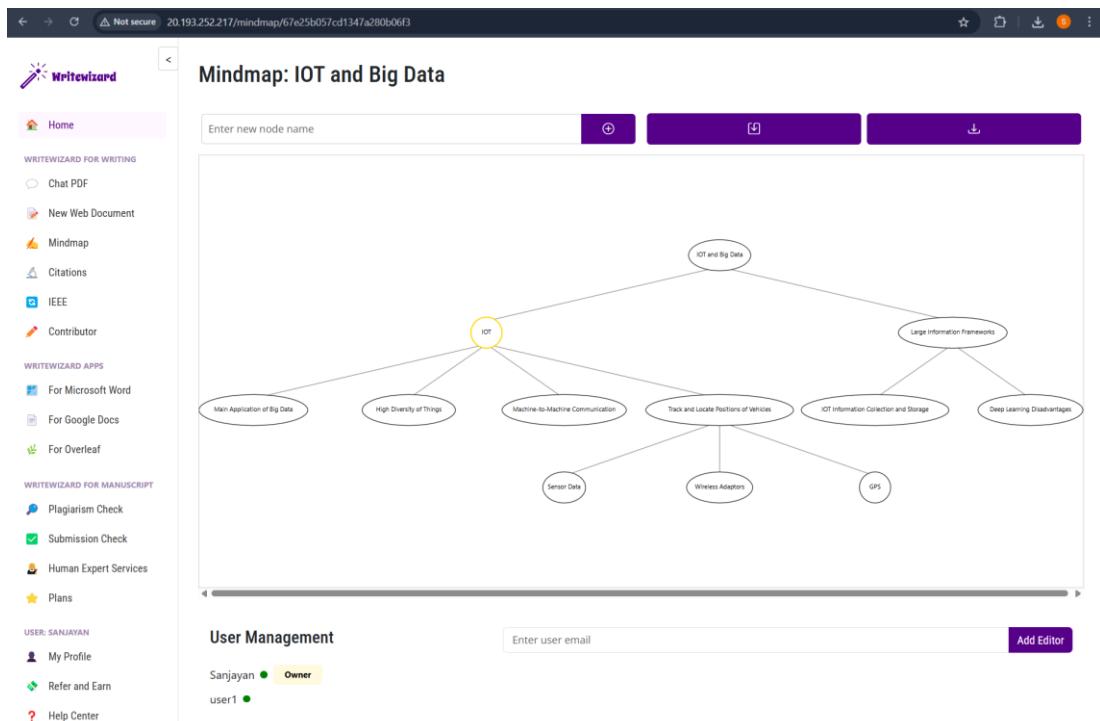


Figure 59: Saved Mind Map Real-Time Collaborative Feature

4. FUTURE SCOPE

This research presents a practical and scalable solution for automatic mind map generation using a fine-tuned transformer model, enhanced with image integration and collaborative editing capabilities. While the current implementation validates its effectiveness in academic settings, several avenues remain open for further exploration, enhancement, and innovation.

One of the most promising directions is the integration of ontology-driven or knowledge-based visualization techniques. By combining natural language processing with structured domain knowledge, the system can evolve to produce even more semantically aligned and pedagogically accurate mind maps. This would be particularly beneficial in subject-specific domains like medicine, law, or engineering, where hierarchical relationships follow formalized structures.

The current model leverages a generic academic dataset, and although it performs well across diverse inputs, domain-specific fine-tuning could lead to improved contextual relevance and terminology accuracy. In future iterations, training the system on curated datasets tailored for different disciplines could help produce context-aware outputs with deeper understanding.

Another key area for development is real-time image generation using advanced multimodal models. While the current implementation matches pre-existing images based on semantic similarity, future versions could generate illustrative images dynamically using generative AI, improving personalization and adaptability across subjects.

In terms of user experience, expanding the customization options for map rendering, such as layout styles (circular, radial, tree), node styling, and annotation features, would support broader use cases. Enhancing mobile responsiveness and developing a dedicated mobile application would also help users interact with their mind maps more conveniently across platforms.

To further improve collaboration, integrating live annotation features and commenting systems within the mind map editor could replicate the classroom discussion environment, encouraging peer-to-peer learning. Integration with widely used academic tools like Google Docs, Notion, or LMS platforms could streamline user workflows and boost adoption in institutional settings.

From a backend perspective, serverless infrastructure and cloud-native scalability options such as container auto-scaling could enhance performance under varying workloads. This would allow the system to support more concurrent users, especially if extended to public-facing applications or institutional licenses.

Finally, continued focus on evaluation frameworks is essential. Future studies could adopt hybrid metrics combining subjective feedback with structured scoring based on semantic similarity, edge consistency, and hierarchy depth. This would allow more objective benchmarking of mind map quality in both academic and research contexts.

In conclusion, the groundwork laid by this research enables future development across model sophistication, user interface design, and cross-platform accessibility. With growing interest in visual learning tools and AI-assisted education, this system has the potential to evolve into a modular, intelligent educational assistant capable of transforming how knowledge is consumed, structured, and retained.

5. CONCLUSION

This research successfully introduced an intelligent, fine-tuned transformer-based system for automated mind map generation within a collaborative academic environment. The solution addresses long-standing limitations in knowledge visualization by transforming academic text into semantically structured, hierarchical visual representations that can be viewed, edited, and shared in real time. Integrated into the WriteWizard platform, the system supports dynamic interaction and contextual enrichment through semantic image matching and multi-user collaboration.

The core microservice, trained using a custom instruction-style dataset, demonstrated strong performance across standard, extended, and simplified generation modes. Evaluations through structured testing and subjective feedback from academic users affirmed its utility, accuracy, and usability. The collaborative frontend built with React and D3.js enabled users to explore and manipulate these visual structures effectively, while the modular backend architecture ensured service decoupling, maintainability, and scalable deployment across cloud infrastructure.

By embedding this functionality directly into a real-time editing environment, the research eliminated workflow fragmentation often caused by external mind mapping tools. The results confirmed not only the technical feasibility of automated visual structuring but also its value in supporting deeper comprehension, faster information recall, and improved academic communication.

Beyond its technical achievements, this research represents a broader shift toward intelligent educational tools that prioritize learner engagement, adaptability, and accessibility. The system demonstrated how artificial intelligence can play a transformative role in academic productivity, allowing learners to process and organize information in ways that align with individual preferences and cognitive strengths.

The integration of this solution into a collaborative academic ecosystem underscored its relevance in modern educational contexts. Students benefited from the ability to generate and customize visual summaries directly from documents, while educators gained a tool that simplifies the creation of learning aids and visual teaching content.

The modularity of the system also opens opportunities for integration into external platforms such as note-taking applications, learning management systems, or domain-specific tools.

In conclusion, this research lays a strong foundation for the future of AI-assisted learning tools. It bridges the gap between textual content and visual cognition, introducing a system that is not only technically robust but also pedagogically impactful. Through further refinement and broader deployment, this approach to automated mind map generation has the potential to evolve into a cornerstone of intelligent academic writing and visualization platforms.

REFERENCES

- [1] T. Buzan and B. Buzan, The Mind Map Book. 1993. [Online]. Available: <http://ci.nii.ac.jp/ncid/BA83072771>
- [2] S. Jawed, H. U. Amin, A. S. Malik, and I. Faye, "Classification of visual and non-visual learners using electroencephalographic alpha and gamma activities," *Frontiers in Behavioral Neuroscience*, vol. 13, May 2019, doi: 10.3389/fnbeh.2019.00086.
- [3] N. Kosmyna, J. T. Lindgren, and A. Lécuyer, "Attending to Visual Stimuli versus Performing Visual Imagery as a Control Strategy for EEG-based Brain-Computer Interfaces," *Scientific Reports*, vol. 8, no. 1, Sep. 2018, doi: 10.1038/s41598-018-31472-9.
- [4] G. Shabiralyani, K. S. Hasan, N. Hamad, and N. Iqbal, "Impact of visual aids in enhancing the learning process Case research: District Dera Ghazi Khan.," *Journal of Education and Practice*, vol. 6, no. 19, pp. 226–233, Jan. 2015, [Online]. Available: <http://files.eric.ed.gov/fulltext/EJ1079541.pdf>
- [5] S. Anjali, N. M. Meera and M. G. Thushara, "A Graph based Approach for Keyword Extraction from Documents," 2019 Second International Conference on Advanced Computational and Communication Paradigms (ICACCP), Gangtok, India, 2019, pp. 1-4, doi: 10.1109/ICACCP.2019.8882946.
- [6] Y. Tian, G. Chen, Y. Song, and X. Wan, "Dependency-driven Relation Extraction with Attentive Graph Convolutional Networks," *ACLWeb*, Aug. 01, 2021. <https://aclanthology.org/2021.acl-long.344/>
- [7] M. Hu, H. Guo, S. Zhao, H. Gao, and Z. Su, "Efficient Mind-Map Generation via Sequence-to-Graph and Reinforced Graph Refinement," *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Jan. 2021, doi: <https://doi.org/10.18653/v1/2021.emnlp-main.641>.
- [8] M. Elhoseiny and A. Elgammal, "English2MindMap: An Automated System for MindMap Generation from English Text," Dec. 2012, doi: <https://doi.org/10.1109/ism.2012.103>.
- [9] R. Kudelic, M. Konecki, and Mirko Malekovic, "Mind map generator software model with text mining algorithm," *Information Technology Interfaces*, pp. 487–494, Jun. 2011.