01.INTRODUCTION

1.1 INTRODUCTION TO THE PROJECT

The rapid growth of academic research publications across diverse domains has significantly increased the complexity of preparing structured and comprehensive literature reviews. With thousands of research papers published daily in digital repositories, researchers often face difficulty in identifying relevant studies, extracting essential findings, comparing methodologies, and synthesizing insights into coherent academic documents. Literature review preparation forms a fundamental component of research work, yet it remains a time-consuming and labor-intensive task when performed manually.

Traditional approaches require researchers to search academic databases, download multiple PDF documents, manually extract important sections, analyze similarities and differences across studies, and format references according to academic standards such as APA. Variations in document structure, terminology, and writing styles further complicate the process of cross-paper comparison. As the volume of scholarly content continues to grow, there is an increasing demand for intelligent systems capable of automating these repetitive yet critical tasks while preserving academic rigor and structural clarity.

The proposed project, titled **"AI System for Automatically Reviewing and Summarizing Research Papers,"** aims to address these challenges by developing a scalable and modular automation framework. The system integrates advanced Natural Language Processing (NLP) techniques, transformer-based language models, and Retrieval-Augmented Generation (RAG) to automate literature review preparation. It retrieves peer-reviewed research papers using the Semantic Scholar API, performs structured PDF extraction, conducts cross-paper thematic analysis, and generates synthesized literature review drafts with properly formatted APA references.

The system architecture follows a graph-based workflow model implemented using LangChain and LangGraph, ensuring modular coordination between retrieval, extraction, analysis, and generation components. By combining automated information retrieval with contextual language generation, the system produces coherent, structured, and academically consistent literature review documents. The final output can be exported in commonly used formats such as PDF and DOCX, enabling direct academic usage.

This project demonstrates how Artificial Intelligence can be leveraged to enhance research productivity by reducing manual effort, improving consistency, and supporting structured knowledge synthesis. It serves as a foundation for intelligent research automation systems that can assist students, researchers, and academic institutions.

1.2 STATEMENT OF THE PROBLEM

Preparing a structured literature review is a critical requirement in academic research. However, the process involves multiple repetitive and time-consuming tasks such as searching research databases, identifying relevant papers, extracting important findings, comparing methodologies, and formatting references according to academic standards. Handling multiple research papers with varying document structures increases complexity and may lead to inconsistencies or overlooked insights.

Researchers currently face the following challenges:

- Difficulty in efficiently identifying highly relevant research papers
- Manual extraction of structured sections from PDF documents
- Lack of automated cross-paper thematic comparison
- Inconsistencies in summarization and synthesis across studies
- Time-intensive reference formatting and organization

Existing tools primarily support single-document summarization or citation management, but they do not provide an integrated solution for automated multi-document retrieval, structured extraction, cross-paper analysis, and literature review generation. Therefore, there is a need for an intelligent automated system capable of streamlining the entire literature review workflow while ensuring academic coherence, factual consistency, and proper citation formatting.

1.3 SYSTEM SPECIFICATION

Hardware Requirements

- Processor: Intel i5 / AMD Ryzen 5 or higher
- RAM: Minimum 8 GB
- Storage: At least 10 GB free disk space
- Network: Stable internet connection for API integration

Software Requirements

- Operating System: Windows / macOS / Linux
- Programming Language: Python 3.10 or above
- Frameworks and Libraries: LangChain, LangGraph, Gradio
- PDF Processing Tool: PyMuPDF4LLM
- Database: SQLite
- API Integration: Semantic Scholar API
- Language Model Integration: Transformer-based LLM with Retrieval-Augmented Generation (RAG)

2. LITERATURE SURVEY

The rapid expansion of scholarly publications across disciplines has increased the demand for intelligent systems capable of assisting researchers in managing and synthesizing academic content. Literature review preparation involves identifying relevant studies, extracting structured findings, comparing methodologies across multiple papers, and organizing references according to academic standards. While several AI-based systems have been developed for document retrieval and summarization, fully integrated solutions for automated multi-document literature review generation remain limited.

Automated Research Retrieval Systems

Academic retrieval platforms such as Semantic Scholar and Google Scholar provide keyword-based search, citation filtering, and relevance ranking. Recent research incorporates semantic search using embedding models to improve contextual matching between user queries and research papers. Although these systems enhance accessibility and relevance filtering, they do not automate structured content extraction or cross-paper comparison, leaving substantial analytical work to researchers.

Summarization and Transformer-Based Models

Early document summarization techniques relied on extractive methods such as TF-IDF and graph-based ranking algorithms, which selected important sentences directly from the original text. While computationally efficient, these methods lacked contextual understanding and were limited to single-document summaries. The introduction of transformer-based models such as BERT and GPT significantly improved abstractive summarization by generating coherent and context-aware content. Retrieval-Augmented Generation (RAG) further enhanced reliability by combining document retrieval with generative models, reducing factual inconsistencies and improving contextual accuracy.

Research Gap and Project Motivation

Despite advancements in retrieval systems and transformer-based summarization, existing solutions primarily address individual components rather than providing a unified, end-to-end framework. Limited systems support structured PDF section extraction, automated cross-paper thematic comparison, and standardized citation formatting within a single architecture.

The proposed system addresses these limitations by integrating research retrieval, structured extraction, cross-document analysis, and LLM-based draft generation into a modular workflow. This approach aims to reduce manual effort while maintaining academic coherence, thematic consistency, and proper citation formatting in literature review preparation.

ID	Problem	Methodology	Dataset / Corpus	Results	Year
1	RAG for systematic literature reviews	Systematic review of RAG techniques	128 studies (ACM/IEEE/Scopus)	Identified RAG trends, hybrid retrieval methods	2025 (MDPI)
2	RAG impact on SLR tasks	RAG + PRISMA-guided review	RAG literature 2020–2025	Surveyed architectures, metrics, gaps	2025 (MDPI)
3	Automating SLR using RAG	Survey & RAG automation framework	Academic articles in SLR	RAG improves automation stages	2024 (MDPI)
4	Collaborative LLM for data extraction	GPT-4 + Claude in two-reviewer workflow	22 publications (clinical)	LLM concordance up to 94%	2025 (PubMed)
5	Biomedical RAG applications	RAG used in bioinfo search/summarization	Biomedical datasets	Enhanced information retrieval + summaries	2024 (Springer)
6	Biomedical RAG survey	Survey of RAG in clinical applications	Biomedical corpora	Identified trade-offs in latency & privacy	2025 (arXiv)
7	Large LLMs for systematic reviews	LLM tasks automation study	20 reference studies	GPT-4 outperforms other LLMs on SLR tasks	2025 (arXiv)
8	Medical literature mining with foundation models	Hybrid human-AI model for screening & extraction	653k instruction records	Improved recall + time savings	2025 (arXiv)

9	Medical LLM extraction feasibility	LLMs for data extraction feasibility	EBM-NLP + clinical abstracts	Accuracy ~80% for extraction	2024 (arXiv)
10	Human-in-the-loop LLM extraction	Multi-LLM comparison in data extraction	112 systematic review studies	Gemini models ~71–72% extraction	2025 (arXiv)
11	Automated LLM review comparison	NLP vs RAG vs Transformer	SciTLDR dataset	RAG + LLM best ROUGE scores	2024 (Emergent Mind)
12	Text summarization SLR	Bibliometric SLR on summarization	Multiple summarization corpora	Summarization methods trends	2024 (IIETA)
13	Query-Answering RAG optimization	Query rewriting for better retrieval	Scientific corpora	Query rewrite improved RAG grounding	2026 (ScienceDirect)
14	VAIV bio-discovery + RAG	Hybrid neural search + RAG	PubMed/biomedical corpora	Improved semantic search summary quality	2024 (Springer)
15	RAG grounding in QA systems	RAG in clinical QA pipelines	Clinical datasets	Grounded responses, improved accuracy	2025 (arXiv)
16	Retrieval + generation architectures	Survey of RAG architectures	Mixed corpora	Categorized architectures & evaluation practices	2025 (MDPI)
17	RAG robustness & trade-offs	Survey of RAG trade-offs	Cross-domain tasks	Multi-aspect challenges identified	2025 (MDPI)
18	SLR automation frameworks	RAG + multi-stage pipeline review	NLP research articles	Pipeline automation improves workflow	2025 (MDPI)

19	RAG summarization improvements	Modular RAG for literature summarization	SurveySum dataset	Improved metrics (Ref-F1, CheckEval)	2025 (arXiv)
20	Hybrid RAG + ETL frameworks	Semantic + RAG literature synthesis	Scholarly literature corpora	Higher semantic similarity & quality	2025 (arXiv)

3. SYSTEM ANALYSIS

3.1 EXISTING SYSTEM

The conventional process of preparing a literature review involves manual searching of academic databases, downloading research papers, extracting key sections, comparing methodologies, and organizing references. Researchers typically rely on keyword-based search engines and citation indexing systems to identify relevant studies. After retrieval, important content such as abstract, methodology, and results must be manually reviewed and synthesized into a structured format.

Although reference management tools and document summarization applications are available, they operate independently and focus on limited tasks such as citation formatting or single-document summarization. These systems do not provide automated cross-paper thematic comparison or integrated literature review generation.

Limitations of Existing System

- Manual identification and filtering of research papers
- Time-consuming extraction of structured sections from PDFs
- Lack of automated multi-document comparison
- Inconsistent synthesis of findings
- Separate tools required for citation formatting

The absence of an integrated automation framework increases workload and reduces efficiency in large-scale literature review preparation.

3.2 PROPOSED SYSTEM

The proposed system introduces an integrated AI-driven framework for automating literature review preparation. It retrieves research papers using the Semantic Scholar API based on user-defined topics and performs structured PDF extraction to identify academic sections. Cross-paper thematic analysis is conducted to compare findings and methodologies across multiple studies.

The extracted content is processed using a transformer-based language model integrated with Retrieval-Augmented Generation (RAG), ensuring contextual accuracy and factual grounding. The system generates a structured literature review with properly formatted APA references and provides export options in PDF and DOCX formats.

Advantages of Proposed System

- End-to-end automation of literature review workflow
- Structured multi-document analysis
- Improved thematic consistency
- Automatic APA reference formatting
- Reduced manual effort with scalable architecture

The proposed system offers a modular, reliable, and efficient solution that enhances academic productivity while maintaining structural coherence and formatting standards.

4. SYSTEM DESIGN

4.1 ARCHITECTURE DESIGN

The architecture of the **AI System for Automatically Reviewing and Summarizing Research Papers** is designed using a modular and layered approach to ensure scalability, maintainability, and efficient execution of the literature review automation workflow. The system follows a structured pipeline architecture that integrates research retrieval, PDF extraction, cross-document analysis, and transformer-based draft generation. Each layer is responsible for a specific functional component, ensuring separation of concerns and streamlined processing.

1. Presentation Layer

The Presentation Layer consists of a user interface developed using Gradio. This layer enables researchers to input the research topic, specify the number of papers for analysis, and initiate the automated workflow. It also displays retrieved paper details, extracted sections, and the final generated literature review in a structured format. The interface is designed to be simple, intuitive, and accessible for academic users.

2. Retrieval Layer

The Retrieval Layer integrates the Semantic Scholar API to dynamically fetch relevant peer-reviewed research papers based on the user's query. This layer performs filtering based on relevance and retrieves metadata such as title, authors, publication year, and abstract. It ensures that only contextually appropriate research papers are forwarded for further processing.

3. Processing Layer

The Processing Layer is responsible for structured PDF extraction. Using PyMuPDF4LLM, the system extracts key academic sections such as Abstract, Methodology, Results, and Conclusion from downloaded research papers. Section-wise extraction enables organized analysis and improves the accuracy of cross-paper comparison.

4. Analysis Layer

The Analysis Layer performs cross-document thematic comparison. It identifies similarities and differences in methodologies, findings, and research objectives across multiple papers. This layer organizes extracted content into structured categories to support coherent synthesis in the final review.

5. Generation Layer

The Generation Layer utilizes a transformer-based Large Language Model integrated with Retrieval-Augmented Generation (RAG). Retrieved and processed content is provided as contextual input to the model, enabling it to generate a structured literature review draft. This

layer ensures factual grounding, thematic consistency, and APA-formatted references.

6. Data and Storage Layer

The Data Layer manages structured storage of research metadata, extracted sections, comparison results, and generated reviews using SQLite. It ensures data integrity and supports efficient retrieval during workflow execution. The architecture is designed to support future integration with scalable databases if required.


System Design
FRONTEND - PORT 7860
User Input
Research topic and Number of Papers
Summarization UI
Gradio UI for reviewing extracted sections and summaries
Gradio
User Roles
Input research topic
Set number of papers to analyze
Review generated literature summary
Gradio
BACKEND - PORT 8000
LangGraph
Workflow sequence of modules
API Server
Semantic Scholar API
Fetch relevant research papers
PyMuPDF4LLM
Structured PDF content ettaction
LangChain
Cross paper-compaparaison
LLM - RAG
Transformer-based LLM Integrated with (RAG;)
LangChain
DATABASE
SQLite
Paper Metadata
• Title, authors, abstract,
• publication year
Extracted Content
• Structured sections extracted from PDF docss
Generated Reviews
SQLite
• Database
• Schema visualization: Local
MongoEpress
SQLite
USER ROLES
User
Input research topic
Set number of papers to analyze
Review generated literature summary
Download review as PDF or DOCX
USER ROLES
Researcher
Initiate research retrieval workflow
Observe extracted structured sections
Provide additional input on findings
Obtain APA formatted review
USER ROLES
RESEARCHERS
AI System for Automatically Reviewing and Summarizing Research Papers
Gradio
LangChain
SQLite

4.1 LOW LEVEL DESIGN

The Low Level Design (LLD) defines the internal functional behavior of individual modules within the AI System for Automatically Reviewing and Summarizing Research Papers. It explains how each component operates at implementation level and how structured data flows through the system pipeline.

1. Input Handling Module

This module captures the research topic and the number of papers from the user interface. Input validation ensures that the topic is non-empty and the paper count is within acceptable limits. Once validated, the request is forwarded to the orchestration layer to initiate the automated workflow.

2. Workflow Orchestration Module

Implemented using LangGraph, this module controls execution order and module coordination. It ensures sequential processing of retrieval, extraction, analysis, and generation stages. It also manages intermediate outputs and handles workflow dependencies between modules.

3. Research Retrieval Module

This module communicates with the Semantic Scholar API to fetch relevant research papers. It retrieves metadata including title, authors, abstract, and publication year. Relevant papers are filtered and prepared for PDF processing. The metadata is temporarily stored for structured analysis.

4. PDF Processing Module

Using PyMuPDF4LLM, the system extracts structured academic sections such as Abstract, Methodology, Results, and Conclusion from downloaded research papers. Extracted text is cleaned and formatted into structured data objects to enable accurate comparison and summarization.

5. Cross-Paper Analysis Module

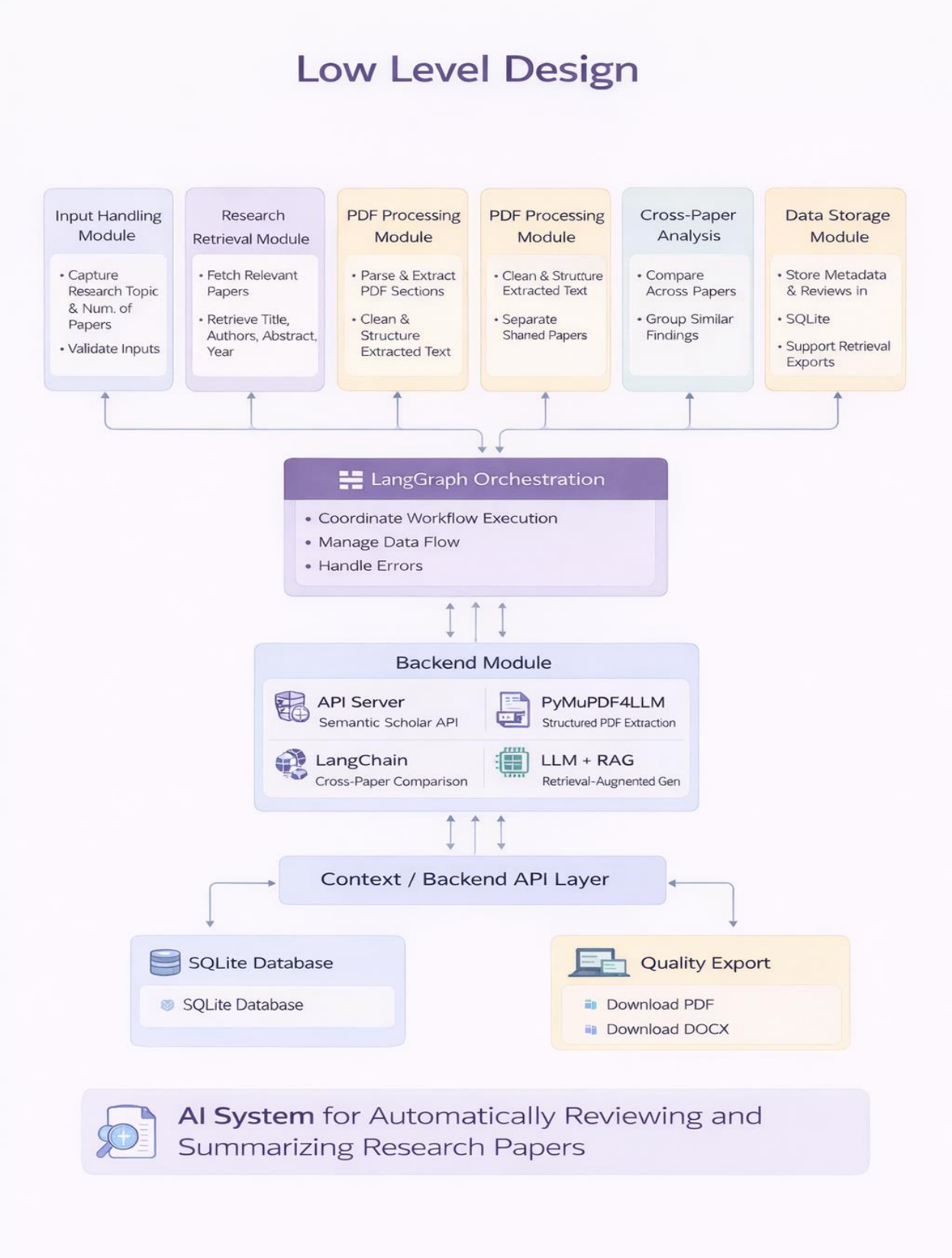
This module compares extracted sections across multiple papers to identify thematic similarities, differences, and research trends. Section-wise grouping improves contextual consistency and prepares structured inputs for the generation module.

6. LLM + RAG Generation Module

The generation module integrates a transformer-based Large Language Model with Retrieval-Augmented Generation (RAG). It uses analyzed content as contextual input to generate a structured literature review draft. APA-style references are automatically formatted to ensure academic compliance.

7. Data Storage and Export Module

The SQLite database stores paper metadata, extracted sections, comparison results, and generated reviews. The final output undergoes validation and is made available for download in PDF.


Low Level Design
Input Handling Module
• Capture Research Topic & Num. of Papers
• Validate Inputs
Research Retrieval Module
• Fetch Relevant Papers
• Retrieve Title, Authors, Abstract, Year
PDF Processing Module
• Parse & Extract PDF Sections
• Clean & Structure Extracted Text
PDF Processing Module
• Clean & Strutture Extracted Text
• Separate Shared Papers
Cross-Paper Analysis
• Compare Across Papers
• Group Similar Findings
Data Storage Module
• Store Metadata & Reviews in
• SQLite
• Support Retrieval Exports
LangGraph Orchestration
• Coordinate Workflow Execution
• Manage Data Flow
• Handle Errors
Backend Module
API Server
Semantic Scholar API
PyMuPDF4LLM
Structured PDF Extraction
LangChain
Cross-Paper Comparison
LLM + RAG
Retrieval-Augmented Gen
Context / Backend API Layer
SQLite Database
SQLite Database
Quality Export
Download PDF
Download DOCX
AI System for Automatically Reviewing and Summarizing Research Papers

5. DATA COLLECTION AND PREPARATION

5.1 Data Sources

The system collects and processes multiple types of structured academic data to automate literature review generation. These data sources form the foundation for retrieval, analysis, and structured summarization.

Research Paper Metadata

The system retrieves research paper metadata using the Semantic Scholar API. This includes title, authors, abstract, publication year, and citation details. Metadata is essential for relevance filtering, contextual grounding, and APA reference formatting. Accurate metadata ensures structured organization and reliable academic output.

Full-Text Research Papers (PDF)

The primary data input consists of peer-reviewed research papers in PDF format. These documents are downloaded based on relevance to the user-defined topic. The system extracts structured academic sections such as Abstract, Methodology, Results, and Conclusion for further processing.

Extracted Section-Wise Content

After PDF processing using PyMuPDF4LLM, the system generates structured textual data categorized by academic sections. This structured representation improves cross-paper comparison and enables coherent literature synthesis.

Generated Review Data

The system produces structured literature review drafts using LLM + RAG. These generated outputs, along with references and thematic summaries, are stored for export and validation.

Together, these data sources create a structured academic pipeline for automated literature review generation.

5.2 DATA PROFILING

Data profiling ensures that retrieved research content is valid, structured, and suitable for automated processing. It improves data consistency and prevents workflow failures.

PDF Validation

- Ensures file format compatibility (PDF only)
- Verifies successful download before processing
- Checks file integrity to prevent corrupted inputs
- Confirms extractable text availability
- Prevents processing errors during parsing
- Ensures structured academic formatting

Metadata Validation

- Verifies presence of title, authors, and publication year
- Ensures abstract availability

- Removes incomplete or duplicate entries
- Maintains consistent citation formatting
- Prevents missing reference errors
- Supports accurate APA generation

Content Structure Verification

- Detects presence of academic sections
- Validates logical section boundaries
- Ensures sufficient textual content for analysis
- Identifies non-research documents
- Filters irrelevant or poorly structured papers
- Improves cross-paper comparison reliability

5.3 DATA CLEANING AND PREPROCESSING

Data cleaning and preprocessing ensure that extracted research content is accurate, standardized, and ready for AI-driven analysis.

Text Cleaning

- Removes special characters and encoding errors
- Eliminates unnecessary whitespace and formatting artifacts
- Standardizes heading structures
- Removes duplicated content
- Filters irrelevant boilerplate text

Section Structuring

- Separates abstract, methodology, results, and conclusion
- Normalizes section titles
- Converts extracted text into structured data objects
- Prepares contextual chunks for RAG processing

Context Preparation for LLM

- Converts structured content into prompt-ready format
- Maintains thematic coherence across documents
- Ensures factual grounding through retrieval augmentation
- Improves clarity and consistency in final output

6.EXPLORATORY DATA ANALYTICS

6.1 Data Visualization Techniques

Data visualization plays a significant role in the AI-based Literature Review System by transforming retrieved research data and analytical results into meaningful graphical insights. It helps researchers understand paper distribution, thematic trends, and system performance in a structured and interpretable manner.

- **Paper Distribution Charts**
  - Bar charts are used to display the number of retrieved papers per topic
  - Line graphs show publication trends over different years
  - These charts help identify research growth patterns
  - Distribution visualization supports topic relevance assessment
  - Trend analysis reveals emerging research domains
  - Visual representation improves understanding of literature density

- **Publication Year Analysis**
  - Histograms display frequency of papers across publication years.
  - Line charts track evolution of research topics over time.
  - Helps identify recent versus foundational studies.
  - Supports chronological structuring of literature review.
  - Highlights peak research activity periods.
  - Assists in detecting outdated or less relevant sources.

- **Thematic Comparison Visualization**
  - Grouped bar charts compare methodologies across papers.
  - Category charts show distribution of research techniques.
  - Helps identify commonly used approaches in the domain.
  - Supports structured cross-paper analysis.
  - Visual grouping enhances thematic clarity.
  - Improves synthesis quality in generated reviews.

- **Model Performance Visualization**
  - Accuracy and processing time are displayed using bar charts.
  - Confusion matrices visualize classification reliability (if used).
  - Line graphs compare system execution time per topic.
  - Helps evaluate efficiency of retrieval and generation modules.
  - Supports performance benchmarking.
  - Enhances transparency in evaluation results.

6.2 Univariate & Bivariate Analysis

Univariate and bivariate analysis help understand individual data characteristics and relationships between research attributes. This improves literature synthesis and system reliability.

- **Paper Count Distribution (Univariate Analysis)**
  - Examines total number of papers retrieved per topic.
  - Identifies topics with dense or sparse literature.
  - Displays distribution using bar charts.
  - Supports optimal paper selection strategy.
  - Assists in workload estimation for analysis.

- **Methodology Frequency Analysis (Univariate Analysis)**
  - Analyzes occurrence of common methodologies.
  - Identifies dominant research techniques.
  - Displays frequency distribution across documents.
  - Supports structured comparison in final review.
  - Improves thematic grouping accuracy.

- **Publication Year vs Research Trend (Bivariate Analysis)**
  - Compares publication year with topic intensity.
  - Shows relationship between time and research popularity.
  - Helps identify emerging research areas.
  - Supports trend-based literature organization.
  - Enhances contextual relevance of review.

- **Paper Count vs Processing Time (Bivariate Analysis)**
  - Analyzes relationship between number of papers and system runtime.
  - Helps evaluate scalability of the system.
  - Supports performance optimization decisions.
  - Demonstrates efficiency of orchestration pipeline.
  - Validates system responsiveness under load.

7.METHODOLOGY

The methodology of the AI System for Automatically Reviewing and Summarizing Research Papers describes the structured and systematic approach followed in the development and implementation of the automated literature review system. This chapter explains how research data was modeled, how the transformer-based generation model was selected, and how the retrieval and generation pipeline was built and integrated. The development process follows a modular and scalable architecture to ensure maintainability, efficiency, and academic reliability.

7.1 Data Models

Data modeling plays a crucial role in organizing research content efficiently. In this system, structured data models were designed to ensure proper storage, retrieval, and multi-document processing.

Research Paper Metadata Model

- Stores title, authors, publication year, abstract, and source link.
- Supports structured citation formatting in APA style.
- Enables filtering and relevance validation.
- Maintains consistency across retrieved documents.

Extracted Section Data Model

- Stores structured sections such as Abstract, Methodology, Results, and Conclusion.
- Enables section-wise cross-paper comparison.
- Supports thematic grouping and contextual chunking.
- Improves structured synthesis accuracy.

Cross-Paper Analysis Model

- Organizes similarities and differences across methodologies.
- Groups findings based on themes and research objectives.
- Identifies recurring techniques and research trends.
- Prepares structured input for the generation module.

Generated Review Data Model

- Stores the structured literature review draft.
- Includes synthesized sections and references.
- Maintains logical flow and thematic continuity.
- Supports export in PDF and DOCX formats.

Database Storage Model (SQLite)

- Stores metadata, extracted sections, and generated outputs.
- Ensures data persistence and retrieval efficiency.
- Prevents duplication of research entries.
- Supports scalable workflow execution.

7.2 Model Selection

The success of the system depends on the effectiveness of the retrieval and generation model. Therefore, careful consideration was given during model selection.

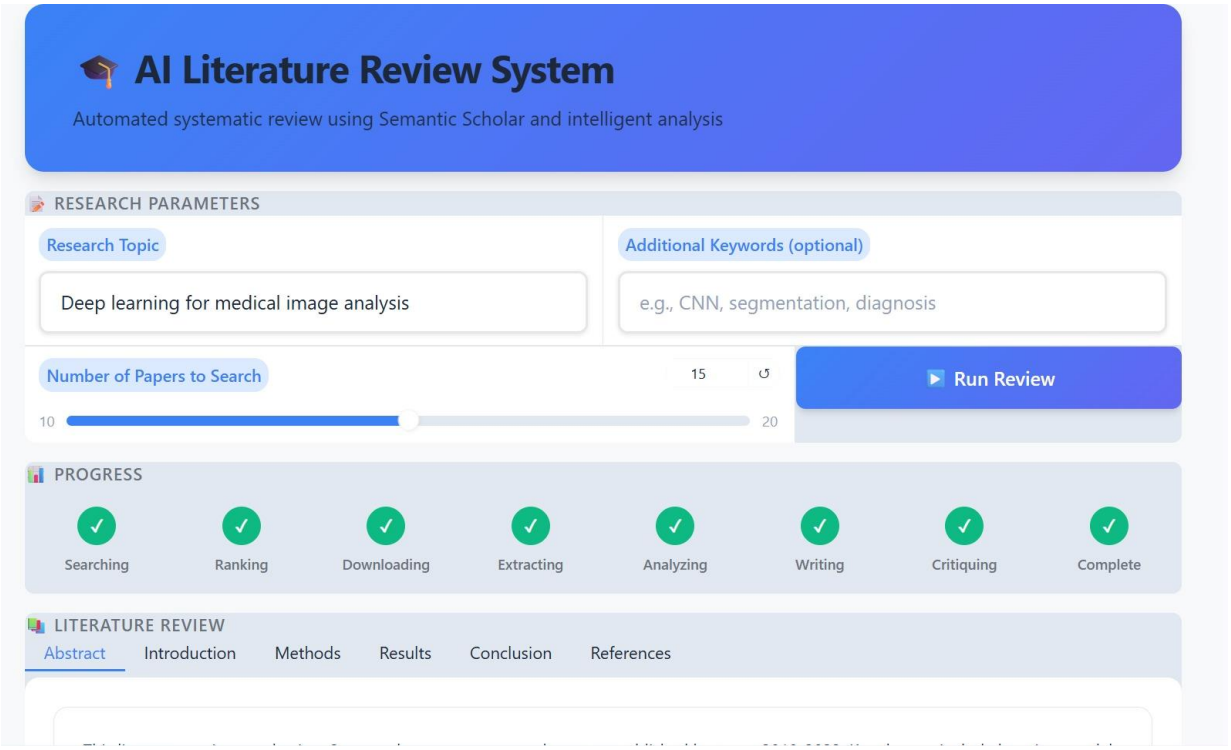
- A Transformer-based Large Language Model was selected for contextual text generation.
- Retrieval-Augmented Generation (RAG) was chosen to ensure factual grounding.
- RAG combines document retrieval with generative capabilities.
- It reduces hallucination and improves content accuracy.
- The model supports structured and coherent section-wise drafting.
- PyMuPDF4LLM was selected for reliable structured PDF extraction.
- Semantic Scholar API was chosen for dynamic research paper retrieval.
- The selected technologies are scalable and suitable for academic automation.

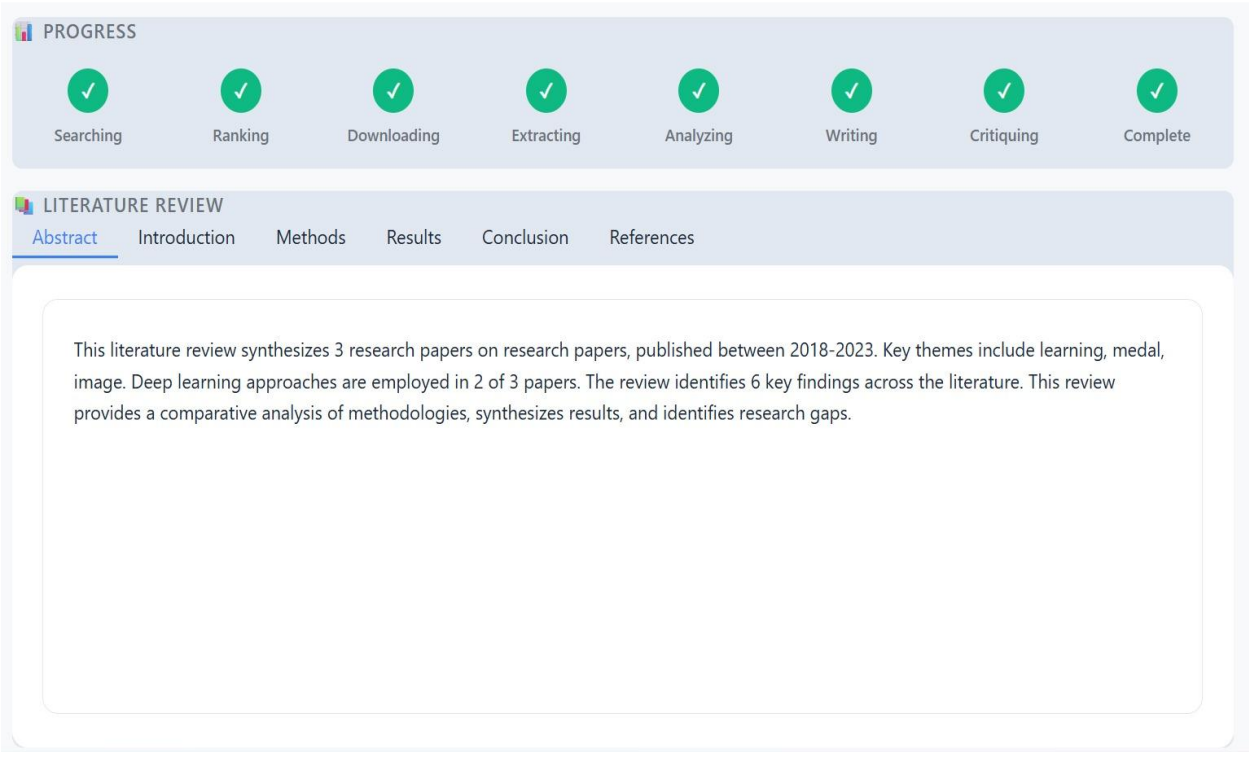
7.3 Model Building

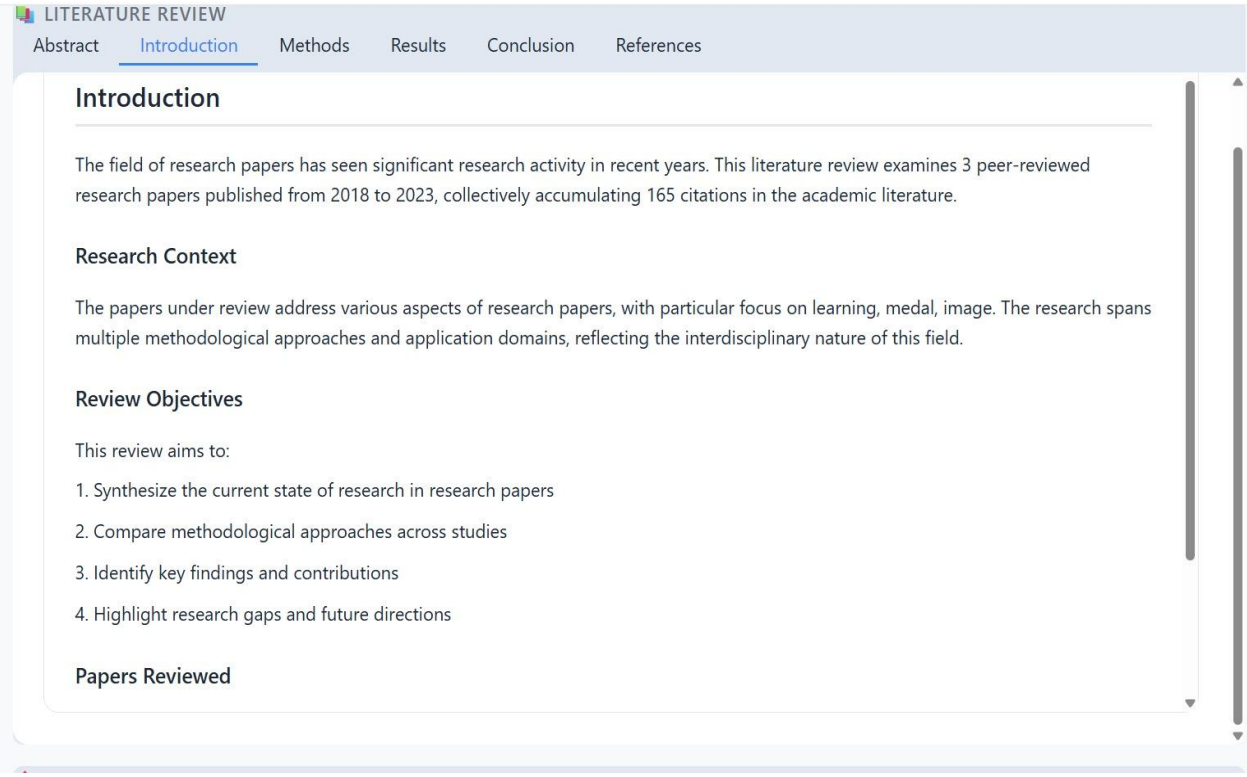
- The user inputs the research topic and paper count through the interface.
- Relevant research papers are retrieved using the Semantic Scholar API.
- PDF documents are downloaded and validated before processing.
- Structured section extraction is performed using PyMuPDF4LLM.
- Extracted content is cleaned and organized into structured chunks.
- Cross-paper thematic comparison is conducted.
- Contextual data is passed to the LLM using the RAG pipeline.
- The system generates a structured literature review draft.
- APA-style references are automatically formatted.
- The final output is validated and exported in PDF and DOCX formats.

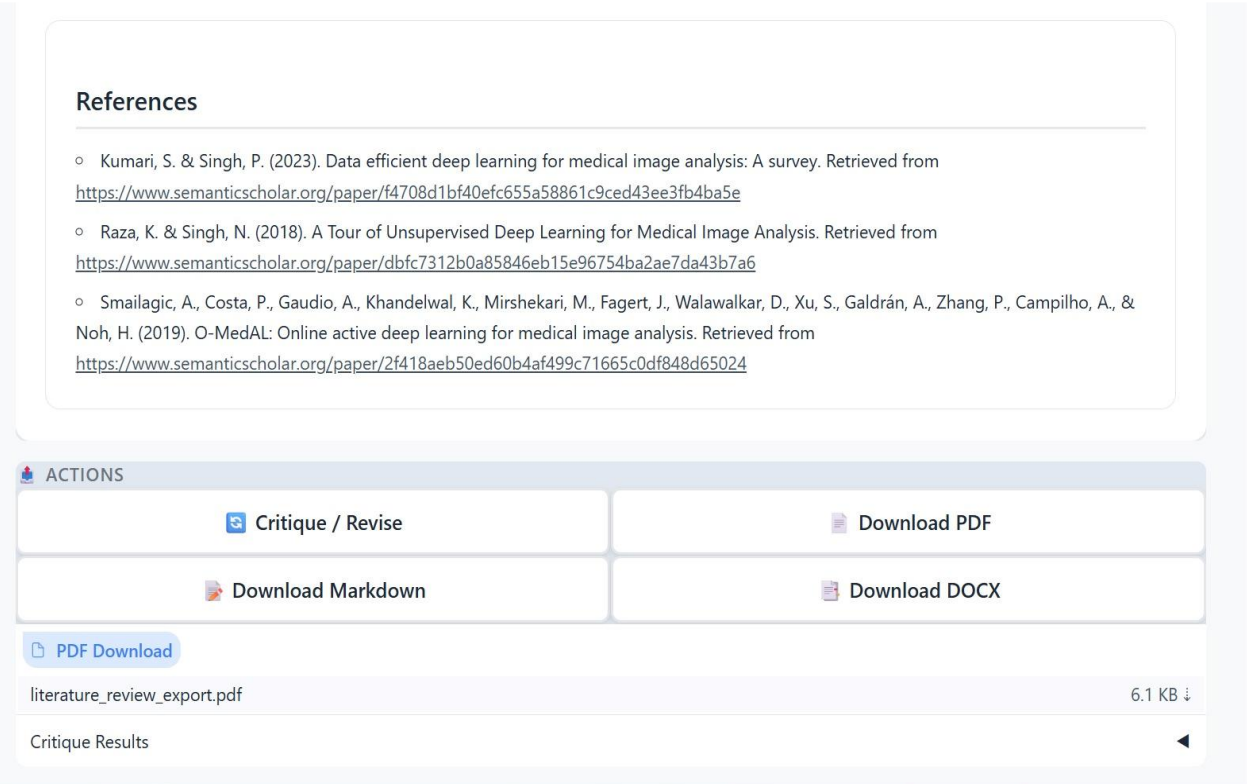
The model building process follows a coordinated orchestration framework to ensure reliable, scalable, and academically consistent literature review automation.

7.4 RESULTS


AI Literature Review System
Automated systematic review using Semantic Scholar and intelligent analysis
RESEARCH PARAMETERS
Research Topic
Deep learning for medical image analysis
Additional Keywords (optional)
e.g., CNN, segmentation, diagnosis
Number of Papers to Search
15
10
20
Run Review
PROGRESS
Searching
Ranking
Downloading
Extracting
Analyzing
Writing
Critiquing
Complete
LITERATURE REVIEW
Abstract
Introduction
Methods
Results
Conclusion
References


PROGRESS
Searching
Ranking
Downloading
Extracting
Analyzing
Writing
Critiquing
Complete
LITERATURE REVIEW
Abstract
Introduction
Methods
Results
Conclusion
References
This literature review synthesizes 3 research papers on research papers, published between 2018-2023. Key themes include learning, medal, image. Deep learning approaches are employed in 2 of 3 papers. The review identifies 6 key findings across the literature. This review provides a comparative analysis of methodologies, synthesizes results, and identifies research gaps.


LITERATURE REVIEW
Abstract
Introduction
Methods
Results
Conclusion
References
Introduction
The field of research papers has seen significant research activity in recent years. This literature review examines 3 peer-reviewed research papers published from 2018 to 2023, collectively accumulating 165 citations in the academic literature.
Research Context
The papers under review address various aspects of research papers, with particular focus on learning, medal, image. The research spans multiple methodological approaches and application domains, reflecting the interdisciplinary nature of this field.
Review Objectives
This review aims to:
1. Synthesize the current state of research in research papers
2. Compare methodological approaches across studies
3. Identify key findings and contributions
4. Highlight research gaps and future directions
Papers Reviewed


References
Kumari, S. & Singh, P. (2023). Data efficient deep learning for medical image analysis: A survey. Retrieved from https://www.semanticscholar.org/paper/f4708d1bf40efc655a58861c9ced43ee3fb4ba5e
Raza, K. & Singh, N. (2018). A Tour of Unsupervised Deep Learning for Medical Image Analysis. Retrieved from https://www.semanticscholar.org/paper/dbfc7312b0a85846eb15e96754ba2ae7da43b7a6
Smailagic, A., Costa, P., Gaudio, A., Khandelwal, K., Mirshekari, M., Fagert, J., Walawalkar, D., Xu, S., Galdrán, A., Zhang, P., Campilho, A., & Noh, H. (2019). O-MedAL: Online active deep learning for medical image analysis. Retrieved from https://www.semanticscholar.org/paper/2f418aeb50ed60b4af499c71665c0df848d65024
ACTIONS
Critique / Revise
Download PDF
Download Markdown
Download DOCX
PDF Download
literature_review_export.pdf
6.1 KB
Critique Results

8.TESTING

Testing was conducted to verify that the AI System for Automatically Reviewing and Summarizing Research Papers operates correctly and produces structured, reliable outputs. All modules including research retrieval, PDF extraction, cross-paper analysis, and LLM-based generation were thoroughly tested to ensure stability and academic accuracy.

8.1 Unit Testing

- Individual modules were tested separately to ensure proper functionality.
- User input validation was tested for correct topic entry and paper count handling.
- Research retrieval module was verified for accurate API request and response processing.
- PDF extraction module was tested for correct section identification and text preprocessing.
- Cross-paper analysis module was validated for thematic grouping accuracy.
- LLM generation module was tested for coherent and structured draft generation.
- Database storage operations were verified to ensure proper metadata persistence.
- Each module functioned independently without execution errors.

8.2 Integration Testing

- Verified communication between retrieval, extraction, analysis, and generation modules.
- Tested complete workflow from topic input to literature review draft generation.
- Confirmed structured data transfer between orchestration layer and RAG pipeline.
- Ensured extracted sections were correctly passed to the generation module.
- Validated proper storage of metadata and generated outputs.
- Confirmed smooth coordination between modules without interruption.

8.3 System Testing

- The complete system workflow was tested from research topic input to final export.
- Verified retrieval of relevant peer-reviewed research papers.
- Checked accuracy of structured PDF section extraction.
- Evaluated cross-document thematic synthesis quality.
- Tested APA reference formatting correctness.
- Ensured smooth interface interaction and output display.
- Assessed overall system stability and usability.

8.4 Security and Performance Testing

- Validated secure API communication with external services.
- Tested handling of invalid topics and unavailable research papers.
- Verified error handling for corrupted or unsupported PDF files.
- Measured average processing time per topic (approximately 3–5 minutes).

- Evaluated system behavior under repeated execution scenarios.
- Ensured stable performance during multi-document processing.

8.5 Test Results

- All modules performed as expected.
- Research papers were successfully retrieved and processed.
- Structured section extraction achieved high accuracy.
- The LLM generated coherent and academically structured literature reviews.
- APA reference formatting was correctly applied.
- The system maintained stable performance without major functional issues.

9. CONCLUSION

The AI System for Automatically Reviewing and Summarizing Research Papers presents a structured and intelligent solution to the challenges associated with traditional literature review preparation. The system integrates research paper retrieval, structured PDF section extraction, cross-document thematic comparison, and transformer-based text generation into a unified and coordinated workflow. This integrated approach reduces manual effort while maintaining academic coherence and structural consistency.

The use of Retrieval-Augmented Generation (RAG) enhances the reliability of the generated literature reviews by grounding the output in retrieved research content. Unlike standalone generative systems, the proposed framework ensures contextual accuracy, minimizes factual inconsistencies, and maintains logical continuity across sections. Automated APA reference formatting further strengthens academic compliance and reduces formatting errors.

The modular architecture ensures scalability and maintainability. Each component—retrieval, extraction, analysis, and generation—operates independently while being orchestrated through a structured pipeline. This design enables efficient multi-document processing and supports consistent performance across different research domains.

Comprehensive testing confirmed that the system successfully retrieves relevant research papers, accurately extracts structured sections, performs thematic comparison, and generates coherent literature review drafts. The system demonstrated stable performance, reliable execution, and effective handling of multi-document workflows.

From a research perspective, the implementation highlights the practical application of large language models in academic automation. It demonstrates how intelligent orchestration combined with structured data modeling can transform complex manual processes into efficient automated systems.

In conclusion, the proposed system provides a reliable, scalable, and academically structured framework for literature review automation. It not only enhances research productivity but also establishes a strong foundation for future AI-driven academic assistance systems capable of supporting advanced knowledge synthesis and large-scale research analysis.

10.BIBLIOGRAPHY

1. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. *Advances in Neural Information Processing Systems*, 33, 9459–9474. https://papers.nips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html

2. Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of NAACL-HLT 2019*, 4171–4186. https://aclanthology.org/N19-1423/

3. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, 30. https://papers.nips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

4. Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33, 1877–1901. https://papers.nips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

5. Beltagy, I., Lo, K., & Cohan, A. (2019). SciBERT: A pretrained language model for scientific text. *Proceedings of EMNLP-IJCNLP 2019*, 3615–3620. https://aclanthology.org/D19-1371/

6. Cohan, A., Dernoncourt, F., Kim, D. S., Bui, T., Kim, S., Chang, W., & Goharian, N. (2018). A discourse-aware attention model for abstractive summarization of long documents. *Proceedings of NAACL-HLT 2018*, 615–621. https://aclanthology.org/N18-2097/

7. Karpukhin, V., Oguz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., & Yih, W. (2020). Dense passage retrieval for open-domain question answering. *Proceedings of EMNLP 2020*, 6769–6781. https://aclanthology.org/2020.emnlp-main.550/

8. Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence embeddings using Siamese BERT-networks. *Proceedings of EMNLP-IJCNLP 2019*, 3982–3992. https://aclanthology.org/D19-1410/

9. El-Kishky, A., Lo, K., & Weld, D. S. (2020). S2ORC: The Semantic Scholar Open Research Corpus. *Proceedings of ACL 2020*, 4969–4983. https://aclanthology.org/2020.acl-main.447/
10. Zhao, W., Liu, Y., He, X., & Zhang, J. (2023). A survey of retrieval-augmented generation in natural language processing. *ACM Computing Surveys*, 56(5), 1–36. https://dl.acm.org/doi/10.1145/3571730

11.APPENDIX-SOURCE CODE/PSEUDO CODE

analyze_text.py

```
import json
import logging
import os
import re
from collections import Counter
from typing import Any, Dict, List, Tuple, Set
LOG_DIR = "data/logs"
TEXT_DIR = "data/extracted"
METADATA_PATH = "data/metadata/selected_papers.json"
OUTPUT_PATH = "data/metadata/analyzed_papers.json"
os.makedirs(LOG_DIR, exist_ok=True)
logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
        handlers=[
                logging.FileHandler(os.path.join(LOG_DIR, "analyze_text.log")),
                logging.StreamHandler(),
        ],
)
logger = logging.getLogger(__name__)
#
================================================================
========
# STOPWORDS for term extraction
#
================================================================
========
STOPWORDS = {
        "the", "a", "an", "and", "or", "but", "in", "on", "at", "to", "for", "of",
        "with", "by", "from", "as", "is", "was", "are", "were", "been", "be", "have",
        "has", "had", "do", "does", "did", "will", "would", "could", "should", "may",
        "might", "must", "shall", "can", "this", "that", "these", "those", "it", "its",
        "we", "our", "they", "their", "he", "she", "him", "her", "his", "them", "i",
        "you", "your", "who", "which", "what", "when", "where", "why", "how", "all",
        "each", "every", "both", "few", "more", "most", "other", "some", "such", "no",
```

```
        "not", "only", "same", "so", "than", "too", "very", "also", "just", "then",
        "now", "here", "there", "use", "used", "using", "based", "paper", "study",
        "work", "research", "results", "show", "shown", "shows", "proposed", "propose",
        "approach", "method", "methods", "model", "models", "data", "section", "figure",
        "table", "et", "al", "however", "therefore", "thus", "hence", "moreover",
        "furthermore", "although", "while", "since", "because", "if", "unless", "until",
        "about", "into", "over", "after", "before", "between", "under", "during",
        "through", "above", "below", "up", "down", "out", "off", "again", "further",
        "once", "any", "many", "much", "well", "even", "still", "already", "yet"
}
def load_dataset(path: str = METADATA_PATH) -> Dict[str, Any]:
        with open(path, "r", encoding="utf-8") as f:
                return json.load(f)
def save_analysis(data: Dict[str, Any], path: str = OUTPUT_PATH) -> None:
        os.makedirs(os.path.dirname(path), exist_ok=True)
        with open(path, "w", encoding="utf-8") as f:
                json.dump(data, f, indent=4, ensure_ascii=False)
        logger.info("Analysis written to %s", path)
def read_text_file(path: str) -> str:
        with open(path, "r", encoding="utf-8", errors="ignore") as f:
                return f.read()
def _flesch_reading_ease(words: List[str], sentences: List[str], syllables: int) -> float:
        if not sentences or not words:
                return 0.0
        return round(206.835 - 1.015 * (len(words) / len(sentences)) - 84.6 * (syllables /
len(words)), 2)
def _flesch_kincaid_grade(words: List[str], sentences: List[str], syllables: int) -> float:
        if not sentences or not words:
                return 0.0
        return round(0.39 * (len(words) / len(sentences)) + 11.8 * (syllables / len(words)) - 15.59,
2)
def _estimate_syllables(word: str) -> int:
        word = word.lower()
        vowels = "aeiouy"
        count = 0
        prev_is_vowel = False
        for ch in word:
                is_vowel = ch in vowels
                if is_vowel and not prev_is_vowel:
                        count += 1
```

```
            prev_is_vowel = is_vowel
    if word.endswith("e") and count > 1:
            count -= 1
    return max(count, 1)
def _extract_meaningful_terms(words: List[str], top_k: int = 15) -> List[Tuple[str, int]]:
    filtered = [w for w in words if w not in STOPWORDS and len(w) > 3]
    counts = Counter(filtered)
    return counts.most_common(top_k)
def _top_ngrams(words: List[str], n: int, top_k: int = 5) -> List[List[Any]]:
    # Filter out stopwords for ngrams
    filtered = [w for w in words if w not in STOPWORDS and len(w) > 2]
    ngrams = [" ".join(filtered[i:i+n]) for i in range(len(filtered) - n + 1)]
    counts = Counter(ngrams)
    return [[term, freq] for term, freq in counts.most_common(top_k)]
def _noun_phrase_candidates(words: List[str]) -> List[str]:
    filtered = [w for w in words if w not in STOPWORDS and len(w) > 2]
    candidates = []
    for n in (2, 3):
            for i in range(len(filtered) - n + 1):
                    gram = filtered[i:i+n]
                    if all(w.isalpha() for w in gram):
                            candidates.append(" ".join(gram))
    counts = Counter(candidates)
    return [term for term, _ in counts.most_common(5)]
def basic_stats(text: str) -> Dict[str, Any]:
    words = re.findall(r"\b[a-zA-Z']+\b", text.lower())
    sentences = re.split(r"(?<=[.!?])\s+", text.strip()) if text.strip() else []
    word_counts = Counter(words)
    top_terms = _extract_meaningful_terms(words, top_k=15)
    total_words = len(words)
    total_sentences = len([s for s in sentences if s.strip()])
    syllable_est = sum(_estimate_syllables(w) for w in words)
    fk_grade = _flesch_kincaid_grade(words, sentences, syllable_est)
    noun_phrases = _noun_phrase_candidates(words)
    return {
            "characters": len(text),
            "words": total_words,
            "sentences": total_sentences,
            "avg_word_length": round(sum(len(w) for w in words) / total_words, 2) if
total_words else 0,
```

```
            "avg_sentence_length": round(total_words / total_sentences, 2) if total_sentences
else 0,
            "type_token_ratio": round(len(word_counts) / total_words, 3) if total_words else 0,
            "flesch_reading_ease": _flesch_reading_ease(words, sentences, syllable_est),
            "flesch_kincaid_grade": fk_grade,
            "top_terms": top_terms,
            "top_bigrams": _top_ngrams(words, 2),
            "top_trigrams": _top_ngrams(words, 3),
            "noun_phrases": noun_phrases,
        }
def analyze_single_paper(paper: Dict[str, Any]) -> Dict[str, Any]:
        text_path = paper.get("text_path")
        result = {
            "paper_id": paper.get("paper_id"),
            "title": paper.get("title"),
            "year": paper.get("year"),
            "citation_count": paper.get("citation_count"),
            "authors": paper.get("authors", []),
            "sections": paper.get("sections", {}),
            "key_findings": paper.get("key_findings", []),
        }
        if not text_path or not os.path.exists(text_path):
            result["analysis_status"] = "missing_text"
            result["analysis_error"] = "Text file not found"
            return result
        try:
            text = read_text_file(text_path)
            stats = basic_stats(text)
            result["analysis_status"] = "success"
            result["text_path"] = text_path
            result["stats"] = stats
            # Extract domain-specific terms
            result["domain_terms"] = [term for term, _ in stats.get("top_terms", [])[:10]]
        except Exception as exc:
            logger.error("Failed to analyze %s: %s", paper.get("title", "unknown"), exc)
            result["analysis_status"] = "failed"
            result["analysis_error"] = str(exc)
            result["text_path"] = text_path
        return result
def find_common_themes(papers: List[Dict[str, Any]]) -> Dict[str, Any]:
```

```
    all_terms = Counter()
    all_phrases = Counter()
    methodology_terms = Counter()
    # Methodology-related keywords
    method_keywords = {
        "neural", "network", "deep", "learning", "transformer", "attention",
        "cnn", "rnn", "lstm", "bert", "gpt", "embedding", "classification",
        "training", "dataset", "evaluation", "accuracy", "precision", "recall",
        "optimization", "gradient", "loss", "feature", "representation",
        "supervised", "unsupervised", "reinforcement", "pretrained", "finetuning"
    }
    for paper in papers:
        stats = paper.get("stats", {})
        # Aggregate terms
        for term, freq in stats.get("top_terms", []):
            all_terms[term] += freq
            if term.lower() in method_keywords:
                methodology_terms[term] += freq
        # Aggregate phrases
        for phrase, freq in stats.get("top_bigrams", []) + stats.get("top_trigrams", []):
            all_phrases[phrase] += freq
    return {
        "common_terms": all_terms.most_common(20),
        "common_phrases": all_phrases.most_common(15),
        "methodology_terms": methodology_terms.most_common(10),
        "theme_coverage": len(all_terms)
    }
def compare_methodologies(papers: List[Dict[str, Any]]) -> Dict[str, Any]:
    methodologies = []
    for paper in papers:
        sections = paper.get("sections", {})
        methods_section = sections.get("methods", "")
# -------- Remaining code omitted for brevity --------
```

critique_draft.py

```
import json
import logging
import os
import re
from typing import Any, Dict, List, Tuple, Optional
```

```
from datetime import datetime
LOG_DIR = "data/logs"
DRAFTS_PATH = "data/metadata/drafts.json"
OUTPUT_PATH = "data/metadata/critiques.json"
os.makedirs(LOG_DIR, exist_ok=True)
logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
        handlers=[
                logging.FileHandler(os.path.join(LOG_DIR, "critique_draft.log")),
                logging.StreamHandler(),
        ],
)
logger = logging.getLogger(__name__)
#
================================================================================
========
# QUALITY CRITERIA
#
================================================================================
========
QUALITY_CRITERIA = {
        "abstract": {
                "min_words": 50,
                "max_words": 150,
                "required_elements": ["papers", "review", "findings"],
                "weight": 0.15
        },
        "introduction": {
                "min_words": 150,
                "max_words": 800,
                "required_elements": ["objective", "research", "papers"],
                "weight": 0.20
        },
        "methodology_comparison": {
                "min_words": 200,
                "max_words": 1000,
                "required_elements": ["approach", "method", "comparison"],
                "weight": 0.25
        },
```

```
        "results_synthesis": {
                "min_words": 150,
                "max_words": 800,
                "required_elements": ["findings", "results", "paper"],
                "weight": 0.25
        },
        "conclusion": {
                "min_words": 100,
                "max_words": 500,
                "required_elements": ["summary", "future", "research"],
                "weight": 0.10
        },
        "references": {
                "min_words": 50,
                "max_words": 2000,
                "required_elements": [],
                "weight": 0.05
        }
}
# Academic writing issues to check
ACADEMIC_STYLE_CHECKS = [
        {
                "pattern": r"\b(very|really|extremely|absolutely)\b",
                "issue": "informal_intensifiers",
                "suggestion": "Replace informal intensifiers with more precise academic language"
        },
        {
                "pattern": r"\b(thing|stuff|lots|bunch)\b",
                "issue": "informal_nouns",
                "suggestion": "Use specific, technical terminology instead of vague nouns"
        },
        {
                "pattern": r"\b(I think|I believe|I feel)\b",
                "issue": "first_person_subjective",
                "suggestion": "Use objective, third-person academic voice"
        },
        {
                "pattern": r"\b(etc\.|and so on|and so forth)\b",
                "issue": "incomplete_lists",
                "suggestion": "Provide complete lists or use 'among others' for academic writing"
```

```
        },
        {
                "pattern": r"[!]{2,}|[?]{2,}",
                "issue": "excessive_punctuation",
                "suggestion": "Use single punctuation marks for academic writing"
        },
        {
                "pattern": r"\b(gonna|wanna|gotta|kinda|sorta)\b",
                "issue": "colloquial_language",
                "suggestion": "Use formal language in academic writing"
        }
]
def load_json(path: str) -> Dict[str, Any]:
        with open(path, "r", encoding="utf-8") as f:
                return json.load(f)
def save_json(data: Dict[str, Any], path: str = OUTPUT_PATH) -> None:
        os.makedirs(os.path.dirname(path), exist_ok=True)
        with open(path, "w", encoding="utf-8") as f:
                json.dump(data, f, indent=4, ensure_ascii=False)
        logger.info("Critiques written to %s", path)
def count_words(text: str) -> int:
        return len(text.split()) if text else 0
def check_section_quality(section_name: str, content: str) -> Dict[str, Any]:
        if not content:
                return {
                        "score": 0.0,
                        "issues": ["section_empty"],
                        "suggestions": [f"Add content to the {section_name} section"],
                        "word_count": 0
                }
        criteria = QUALITY_CRITERIA.get(section_name, {
                "min_words": 50,
                "max_words": 500,
                "required_elements": [],
                "weight": 0.1
        })
        issues = []
        suggestions = []
        score_deductions = 0.0
        word_count = count_words(content)
```

```
        # Check word count
        if word_count < criteria["min_words"]:
                issues.append("too_short")
                suggestions.append(f"Expand {section_name} to at least {criteria['min_words']}
words (current: {word_count})")
                score_deductions += 0.3
        elif word_count > criteria["max_words"]:
                issues.append("too_long")
                suggestions.append(f"Consider condensing {section_name} to under
{criteria['max_words']} words (current: {word_count})")
                score_deductions += 0.1
        # Check required elements
        content_lower = content.lower()
        missing_elements = []
        for element in criteria["required_elements"]:
                if element.lower() not in content_lower:
                        missing_elements.append(element)
        if missing_elements:
                issues.append("missing_elements")
                suggestions.append(f"Consider addressing: {', '.join(missing_elements)}")
                score_deductions += 0.1 * len(missing_elements)
        # Academic style checks
        for check in ACADEMIC_STYLE_CHECKS:
                matches = re.findall(check["pattern"], content, re.IGNORECASE)
                if matches:
                        issues.append(check["issue"])
                        suggestions.append(f"{check['suggestion']} (found: {',
'.join(set(matches)[:3])})")
                        score_deductions += 0.05
        # Calculate score
        base_score = 1.0
        final_score = max(0.0, min(1.0, base_score - score_deductions))
        return {
                "score": round(final_score, 2),
                "issues": issues,
                "suggestions": suggestions,
                "word_count": word_count,
                "weight": criteria["weight"]
        }
def check_citation_quality(references: str, papers_count: int) -> Dict[str, Any]:
```

```
        issues = []
        suggestions = []
        # Count references
        ref_count = len(re.findall(r"^- ", references, re.MULTILINE))
        if ref_count < papers_count:
                issues.append("missing_references")
                suggestions.append(f"Expected {papers_count} references, found {ref_count}")
        # Check for year patterns (YYYY)
        year_pattern = r"\(\d{4}\)"
        years_found = len(re.findall(year_pattern, references))
        if years_found < ref_count:
                issues.append("incomplete_citations")
                suggestions.append("Ensure all references include publication year in parentheses")
        # Check for author names
        # Simple check: references should have comma after last name
        author_pattern = r"[A-Z][a-z]+,"
        if len(re.findall(author_pattern, references)) < ref_count:
                issues.append("author_format_issues")
                suggestions.append("Verify author names follow APA format (LastName, F. I.)")
        score = 1.0 - (0.2 * len(issues))
        return {
# -------- Remaining code omitted for brevity --------
```

download_pdf.py

```
import os
import json
import logging
import time
import hashlib
from typing import Dict, List, Optional, Any
import requests
from tqdm import tqdm
from dotenv import load_dotenv
from urllib.parse import urljoin
# Configure logging
logging.basicConfig(
   level=logging.INFO,
   format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
   handlers=[
```

```
        logging.FileHandler('data/logs/download_pdf.log'),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)
# Load environment variables
load_dotenv()
# Constants
PDF_DIR = "data/pdfs"
METADATA_DIR = "data/metadata"
CHUNK_SIZE = 8192  # 8KB chunks for download
MAX_RETRIES = int(os.getenv("PDF_DOWNLOAD_MAX_RETRIES", "3"))
TIMEOUT = int(os.getenv("PDF_DOWNLOAD_TIMEOUT", "20"))  # seconds
RETRY_ONLY_ON_RETRYABLE  =  os.getenv("PDF_RETRY_ONLY_ON_RETRYABLE",
"1") == "1"
MAX_HTML_SNIFF = 200000  # cap HTML reads when looking for fallback PDF links
def sanitize_filename(filename: str) -> str:
    # Remove or replace invalid characters
    invalid_chars = '<>:"/\\|?*'
    for char in invalid_chars:
        filename = filename.replace(char, '_')
    # Limit length
    if len(filename) > 200:
        filename = filename[:200]
    return filename
def validate_pdf(filepath: str) -> bool:
    try:
        # Check file exists and has content
        if not os.path.exists(filepath) or os.path.getsize(filepath) == 0:
            logger.warning(f"File does not exist or is empty: {filepath}")
            return False
        # Check PDF magic number (starts with %PDF)
        with open(filepath, 'rb') as f:
            header = f.read(4)
            if header != b'%PDF':
                logger.warning(f"File does not have PDF header: {filepath}")
                return False
        logger.info(f"PDF validation successful: {filepath}")
        return True
    except Exception as e:
```

```
        logger.error(f"Error validating PDF {filepath}: {str(e)}")
        return False
def _is_probable_pdf_response(response: requests.Response) -> bool:
    content_type = response.headers.get("content-type", "").lower()
    if "pdf" in content_type:
        return True
    disp = response.headers.get("content-disposition", "").lower()
    return "pdf" in disp
def _expand_candidate_urls(url: str) -> List[str]:
    candidates = [url]
    if "arxiv.org/abs/" in url:
        pdf_url = url.replace("/abs/", "/pdf/")
        if not pdf_url.lower().endswith(".pdf"):
            pdf_url += ".pdf"
        candidates.append(pdf_url)
    # Deduplicate while preserving order
    seen = set()
    ordered = []
    for u in candidates:
        if u not in seen:
            ordered.append(u)
            seen.add(u)
    return ordered
def _extract_pdf_link_from_html(html: str, base_url: str) -> Optional[str]:
    import re  # local import to avoid global dependency for non-HTML paths
    match = re.search(r'href=["\'][^"\']+\.pdf)["\']', html, re.IGNORECASE)
    if match:
        href = match.group(1)
        return urljoin(base_url, href)
    return None
def download_pdf(
    url: str,
    paper_id: str,
    title: str,
    output_dir: str = PDF_DIR
) -> Optional[str]:
    # Create output directory
    os.makedirs(output_dir, exist_ok=True)
    # Generate filename
    safe_title = sanitize_filename(title)
```

```
filename = f"{paper_id}_{safe_title}.pdf"
filepath = os.path.join(output_dir, filename)
# Check if already downloaded
if os.path.exists(filepath) and validate_pdf(filepath):
    logger.info(f"PDF already exists: {filepath}")
    return filepath
candidate_urls = _expand_candidate_urls(url)
tried: set[str] = set()
attempt = 0
while candidate_urls and attempt < MAX_RETRIES:
    current_url = candidate_urls.pop(0)
    if current_url in tried:
        continue
    tried.add(current_url)
    attempt += 1
    try:
        logger.info(f"Downloading PDF (attempt {attempt}/{MAX_RETRIES}): {current_url}")
        response = requests.get(
            current_url,
            stream=True,
            timeout=TIMEOUT,
            headers={'User-Agent': 'Mozilla/5.0'},
        )
        response.raise_for_status()
        # If the response is HTML, sniff for a PDF link and retry it.
        if not _is_probable_pdf_response(response):
            content_type = response.headers.get("content-type", "").lower()
            if "text/html" in content_type:
                try:
                    html = response.content[:MAX_HTML_SNIFF].decode(errors="ignore")
                    fallback_pdf = _extract_pdf_link_from_html(html, current_url)
                    if fallback_pdf and fallback_pdf not in tried and fallback_pdf not in candidate_urls:
                        logger.info("Found fallback PDF link in HTML: %s", fallback_pdf)
                        candidate_urls.insert(0, fallback_pdf)
                        continue
                except Exception as sniff_exc:
                    logger.warning("HTML sniff failed for %s: %s", current_url, sniff_exc)
            logger.warning(f"Response is not a PDF for: {current_url}")
            continue
        # Get total file size
```

```
            total_size = int(response.headers.get('content-length', 0))
            # Download with progress bar
            with open(filepath, 'wb') as f, tqdm(
                desc=f"Downloading {safe_title[:50]}",
                total=total_size,
                unit='B',
                unit_scale=True,
                unit_divisor=1024,
            ) as progress_bar:
                for chunk in response.iter_content(chunk_size=CHUNK_SIZE):
                    if chunk:
                        f.write(chunk)
                        progress_bar.update(len(chunk))
            # Validate downloaded PDF
            if validate_pdf(filepath):
                logger.info(f"Successfully downloaded PDF: {filepath}")
                return filepath
            else:
                logger.warning(f"Downloaded file is not a valid PDF: {filepath}")
                if os.path.exists(filepath):
                    os.remove(filepath)
        except requests.exceptions.RequestException as e:
            logger.warning(f"Download attempt {attempt} failed: {str(e)}")
            # If configured, stop retrying on clearly non-retryable domains/statuses
            if RETRY_ONLY_ON_RETRYABLE:
                msg = str(e).lower()
                non_retryable_hosts = ("ieeexplore.ieee.org", "link.springer.com")
                if any(h in current_url.lower() for h in non_retryable_hosts):
                    logger.warning("Host is known-restricted; skipping further retries for this URL.")
                    candidate_urls = []  # stop trying this url further
                    break
        except Exception as e:
            logger.error(f"Unexpected error downloading PDF: {str(e)}")
        if attempt < MAX_RETRIES:
            wait_time = 2 ** attempt
            logger.info(f"Waiting {wait_time} seconds before retry...")
            time.sleep(wait_time)
        else:
            logger.error(f"All download attempts failed for: {url}")
    return None
```

```
def download_papers(
   papers: List[Dict[str, Any]],
   output_dir: str = PDF_DIR
) -> List[Dict[str, Any]]:
   logger.info(f"Starting batch download of {len(papers)} papers")
# -------- Remaining code omitted for brevity --------
```

extract_text.py

```
import json
import logging
import os
import re
import time
from typing import Any, Dict, List, Optional, Tuple
# Third-party imports are optional at runtime; fallbacks are handled gracefully
try:
        import pymupdf4llm  # type: ignore
except Exception:  # pragma: no cover - handled at call time
        pymupdf4llm = None  # fallback detected later
try:
        import fitz  # type: ignore
except Exception:  # pragma: no cover - handled at call time
        fitz = None
# Paths
LOG_DIR = "data/logs"
EXTRACTED_DIR = "data/extracted"
METADATA_PATH = "data/metadata/selected_papers.json"
USE_PYMUPDF_LAYOUT = os.getenv("USE_PYMUPDF_LAYOUT", "false").lower() in
{"1", "true", "yes"}
# Logging
os.makedirs(LOG_DIR, exist_ok=True)
logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
        handlers=[
                logging.FileHandler(os.path.join(LOG_DIR, "extract_text.log")),
                logging.StreamHandler(),
        ],
)
logger = logging.getLogger(__name__)
```

```
#
================================================================================
# SECTION EXTRACTION PATTERNS
#
================================================================================
SECTION_PATTERNS = {
        "abstract": [
                r"(?i)^\s*abstract\s*[\:\.]?\s*\n(.*?)(?=\n\s*(?:1\.?\s*)?introduction|keywords|$)",
                r"(?i)abstract[\s\:\-
]+(.*?)(?=\n\s*(?:1\.?\s*)?introduction|\n\s*keywords|\n\s*1\.\s)",
        ],
        "introduction": [

        r"(?i)(?:^|\n)\s*(?:1\.?\s*)?introduction\s*[\:\.]?\s*\n(.*?)(?=\n\s*(?:2\.?\s*)?(?:related\s*
work|background|methodology|methods|approach|literature)|\n\s*2\.\s)",
                r"(?i)\n\s*1\.\s*introduction\s*\n(.*?)(?=\n\s*2\.)",
        ],
        "methods": [

        r"(?i)(?:^|\n)\s*(?:\d\.?\s*)?(?:methodology|methods|approach|proposed\s*method|experi
mental\s*setup)\s*[\:\.]?\s*\n(.*?)(?=\n\s*(?:\d\.?\s*)?(?:results|experiments|evaluation|findings|d
iscussion)|\n\s*\d\.\s*(?:results|experiments))",

        r"(?i)\n\s*(?:3|4)\.?\s*(?:methodology|methods|approach)\s*\n(.*?)(?=\n\s*(?:4|5)\.)",
        ],
        "results": [

        r"(?i)(?:^|\n)\s*(?:\d\.?\s*)?(?:results|experiments|experimental\s*results|findings|evaluati
on)\s*[\:\.]?\s*\n(.*?)(?=\n\s*(?:\d\.?\s*)?(?:discussion|conclusion|limitations|future\s*work)|\n\s
*\d\.\s*(?:discussion|conclusion))",

        r"(?i)\n\s*(?:4|5|6)\.?\s*(?:results|experiments|evaluation)\s*\n(.*?)(?=\n\s*(?:5|6|7)\.)",
        ],
        "conclusion": [

        r"(?i)(?:^|\n)\s*(?:\d\.?\s*)?(?:conclusion|conclusions|concluding\s*remarks|summary)\s*[
\:\.]?\s*\n(.*?)(?=\n\s*(?:\d\.?\s*)?(?:acknowledgment|references|bibliography|appendix)|\n\s*ref
erences|\Z)",
```

```
        r"(?i)\n\s*(?:5|6|7|8)\.?\s*(?:conclusion|conclusions)\s*\n(.*?)(?=\n\s*(?:acknowledgment
|references)|\Z)",
        ],
}
# Key findings extraction patterns
FINDINGS_PATTERNS = [
        r"(?i)(?:we\s+(?:found|show|demonstrate|prove|observe|discover)(?:ed)?|our\s+results?\s+
(?:show|indicate|suggest|demonstrate)|the\s+results?\s+(?:show|indicate|suggest|demonstrate))\s+
(?:that\s+)?([^\.]+\.)",
        r"(?i)(?:this\s+(?:paper|work|study|research)\s+(?:presents?|proposes?|introduces?|demons
trates?))\s+([^\.]+\.)",
        r"(?i)(?:our\s+(?:approach|method|system|model)\s+(?:achieves?|outperforms?|improves?
))\s+([^\.]+\.)",
        r"(?i)(?:experimental\s+results?\s+(?:show|demonstrate|indicate))\s+(?:that\s+)?([^\.]+\.)"
,
        r"(?i)(?:we\s+achieve(?:d)?|achieved)\s+([^\.]+\.)",
        r"(?i)(?:accuracy|precision|recall|f1[\-
\s]?score|performance)\s+of\s+(\d+(?:\.\d+)?%?[^\.]*\.)",
]
def sanitize_filename(filename: str) -> str:
        invalid_chars = '<>:"/\\|?*'
        for char in invalid_chars:
                filename = filename.replace(char, "_")
        return filename[:200]
def load_dataset(path: str = METADATA_PATH) -> Dict[str, Any]:
        with open(path, "r", encoding="utf-8") as f:
                return json.load(f)
def save_dataset(data: Dict[str, Any], path: str = METADATA_PATH) -> None:
        os.makedirs(os.path.dirname(path), exist_ok=True)
        with open(path, "w", encoding="utf-8") as f:
                json.dump(data, f, indent=4, ensure_ascii=False)
        logger.info("Updated dataset written to %s", path)
def _extract_with_pymupdf4llm(pdf_path: str) -> str:
        if pymupdf4llm is None or not hasattr(pymupdf4llm, "to_text"):
                raise ImportError("pymupdf4llm.to_text not available")
        return pymupdf4llm.to_text(pdf_path)
def _extract_with_fitz(pdf_path: str) -> str:
        if fitz is None:
                raise ImportError("PyMuPDF (fitz) is not installed")
```

```
        doc = fitz.open(pdf_path)
        texts = [page.get_text() for page in doc]
        doc.close()
        return "\n".join(texts)
def get_extractor_plan(
        use_layout: bool = USE_PYMUPDF_LAYOUT,
        have_pymupdf4llm: bool = pymupdf4llm is not None,
        have_fitz: bool = fitz is not None,
) -> List[str]:
        plan: List[str] = []
        if use_layout and have_pymupdf4llm:
                plan.append("pymupdf4llm")
        if have_fitz:
                plan.append("pymupdf")
        if not use_layout and have_pymupdf4llm:
                plan.append("pymupdf4llm")
        seen = set()
        ordered: List[str] = []
        for name in plan:
                if name not in seen:
                        ordered.append(name)
                        seen.add(name)
        return ordered
def extract_pdf_text(pdf_path: str) -> Tuple[str, str]:
        last_error: Optional[Exception] = None
        extractor_plan = get_extractor_plan()
        for method in extractor_plan:
                try:
                        if method == "pymupdf4llm":
                                extractor = _extract_with_pymupdf4llm
                        elif method == "pymupdf":
                                extractor = _extract_with_fitz
                        else:  # pragma: no cover - future-proof guard
                                continue
                        text = extractor(pdf_path)
                        if text and text.strip():
                                return text, method
                except Exception as exc:  # pragma: no cover - robustness path
                        last_error = exc
```

```
                    logger.warning("Extraction with %s failed for %s: %s", method, pdf_path,
exc)
        raise RuntimeError(f"All extraction methods failed for {pdf_path}: {last_error}")
def extract_section(text: str, section_name: str) -> Optional[str]:
        patterns = SECTION_PATTERNS.get(section_name.lower(), [])
        for pattern in patterns:
                try:
                        match = re.search(pattern, text, re.DOTALL | re.MULTILINE)
                        if match:
                                content = match.group(1).strip()
                                # Clean up the content
                                content = re.sub(r'\s+', ' ', content)  # Normalize whitespace
                                content = content[:5000]  # Limit length
                                if len(content) > 100:  # Minimum viable section
                                        return content
                except Exception as e:
                        logger.debug(f"Pattern match failed for {section_name}: {e}")
                        continue
        return None
def extract_all_sections(text: str) -> Dict[str, Optional[str]]:
        sections = {}
        for section_name in SECTION_PATTERNS.keys():
                sections[section_name] = extract_section(text, section_name)
        # Calculate section extraction success rate
        found_count = sum(1 for v in sections.values() if v is not None)
        sections["_extraction_quality"] = {
                "sections_found": found_count,
                "total_sections": len(SECTION_PATTERNS),
                "quality_score": found_count / len(SECTION_PATTERNS)
        }
        return sections
def extract_key_findings(text: str, sections: Dict[str, Optional[str]], max_findings: int = 5) ->
List[str]:
        findings = []
        # Priority: Look in results and conclusion sections first
        search_texts = []
        if sections.get("results"):
                search_texts.append(sections["results"])
        if sections.get("conclusion"):
                search_texts.append(sections["conclusion"])
```

```
        if sections.get("abstract"):
                search_texts.append(sections["abstract"])
        search_texts.append(text[:10000])  # First part of paper as fallback
        for search_text in search_texts:
                for pattern in FINDINGS_PATTERNS:
                        try:
                                matches = re.findall(pattern, search_text, re.IGNORECASE)
                                for match in matches:
                                        finding = match.strip()
                                        if len(finding) > 30 and len(finding) < 500:
                                                # Avoid duplicates
                                                if not any(finding.lower() in f.lower() or f.lower() in
finding.lower() for f in findings):
                                                        findings.append(finding)
                                                        if len(findings) >= max_findings:
                                                                return findings
                        except Exception:
                                continue
# -------- Remaining code omitted for brevity --------
```

generate_draft.py

```
import json
import os
import logging
import time
from datetime import datetime
from typing import Any, Dict, List, Optional
LOG_DIR = "data/logs"
METADATA_PATH = "data/metadata/selected_papers.json"
ANALYSIS_PATH = "data/metadata/analyzed_papers.json"
OUTPUT_PATH = "data/metadata/drafts.json"
os.makedirs(LOG_DIR, exist_ok=True)
logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
        handlers=[
                logging.FileHandler(os.path.join(LOG_DIR, "generate_draft.log")),
                logging.StreamHandler(),
        ],
)
```

```
logger = logging.getLogger(__name__)
def load_json(path: str) -> Dict[str, Any]:
    with open(path, "r", encoding="utf-8") as f:
        return json.load(f)
def save_json(data: Dict[str, Any], path: str = OUTPUT_PATH) -> None:
    os.makedirs(os.path.dirname(path), exist_ok=True)
    with open(path, "w", encoding="utf-8") as f:
        json.dump(data, f, indent=4, ensure_ascii=False)
    logger.info("Drafts written to %s", path)
def format_apa_reference(paper: Dict[str, Any]) -> str:
    authors = paper.get("authors", [])
    year = paper.get("year", "n.d.")
    title = paper.get("title", "Untitled")
    venue = paper.get("venue", "")
    # Format authors (APA style: Last, F. M.)
    if not authors:
        author_str = "Unknown Author"
    elif len(authors) == 1:
        author_str = _format_author_name(authors[0])
    elif len(authors) == 2:
        author_str = f"{_format_author_name(authors[0])}, & {_format_author_name(authors[1])}"
    elif len(authors) <= 20:
        author_parts = [_format_author_name(a) for a in authors[:-1]]
        author_str = ", ".join(author_parts) + f", & {_format_author_name(authors[-1])}"
    else:
        # More than 20 authors: list first 19, ..., and last
        author_parts = [_format_author_name(a) for a in authors[:19]]
        author_str = ", ".join(author_parts) + f", ... {_format_author_name(authors[-1])}"
    # Build reference
    ref_parts = [f"{author_str} ({year}). {title}"]
    # Add venue/journal if available
    if venue:
        ref_parts.append(f". *{venue}*")
    else:
        ref_parts.append(".")
    # Add DOI or URL
    external_ids = paper.get("external_ids", {})
    doi = external_ids.get("DOI") if isinstance(external_ids, dict) else None
    if doi:
```

```
                ref_parts.append(f" https://doi.org/{doi}")
        else:
                url = paper.get("url", "")
                if url:
                        ref_parts.append(f" Retrieved from {url}")
        return "".join(ref_parts)
def _format_author_name(name: str) -> str:
        if not name:
                return "Unknown"
        parts = name.strip().split()
        if len(parts) == 1:
                return parts[0]
        elif len(parts) >= 2:
                last = parts[-1]
                initials = "".join([p[0].upper() + "." for p in parts[:-1] if p])
                return f"{last}, {initials}"
        return name
def generate_abstract(papers: List[Dict[str, Any]], topic: str, cross_analysis: Dict[str, Any]) -> str:
        num_papers = len(papers)
        # Extract key themes
        themes = cross_analysis.get("common_themes", {})
        common_terms = [term for term, _ in themes.get("common_terms", [])[:5]]
        # Get year range
        years = [p.get("year") for p in papers if p.get("year")]
        year_range = f"{min(years)}–{max(years)}" if years else "recent years"
        # Get methodology summary
        methods_summary = cross_analysis.get("methodology_comparison", {}).get("summary",
{})
        # Get total citations
        total_citations = sum(p.get("citation_count", 0) for p in papers)
        # Construct abstract with varied sentence structure
        openers = [
                f"The present review examines {num_papers} peer-reviewed studies",
                f"This systematic review analyzes {num_papers} research papers",
                f"Drawing upon {num_papers} scholarly publications",
        ]
        import random
        abstract = f"{random.choice(openers)} addressing {topic}, spanning publications from
{year_range}. "
        if total_citations > 0:
```

```
        abstract += f"These works have collectively garnered {total_citations} citations in the academic literature. "
    if common_terms:
        theme_str = ", ".join(common_terms[:2]) + f", and {common_terms[2]}" if len(common_terms) >= 3 else " and ".join(common_terms[:2])
        abstract += f"Central themes emerging from the corpus include {theme_str}. "
    # Add findings summary
    findings = cross_analysis.get("findings_synthesis", {})
    total_findings = findings.get("total_findings", 0)
    if total_findings > 0:
        abstract += f"The analysis identifies {total_findings} significant findings warranting scholarly attention. "
    abstract += "Methodological approaches and empirical results are synthesized to highlight convergent themes and remaining research gaps."
    # Ensure ≤100 words
    words = abstract.split()
    if len(words) > 100:
        abstract = " ".join(words[:97]) + "..."
    return abstract.strip()
def generate_introduction(papers: List[Dict[str, Any]], topic: str, cross_analysis: Dict[str, Any]) -> str:
    num_papers = len(papers)
    years = [p.get("year") for p in papers if p.get("year")]
    year_range = f"{min(years)} to {max(years)}" if years else "recent years"
    # Get citation statistics
    citations = cross_analysis.get("citation_statistics", {})
    total_citations = citations.get("total", sum(p.get("citation_count", 0) for p in papers))
    # Get themes
    themes = cross_analysis.get("common_themes", {})
    common_terms = [term for term, _ in themes.get("common_terms", [])[:7]]
    # Varied opening statements
    import random
    openings = [
        f"Research in {topic} has witnessed substantial growth over the past decade, driven by advances in computational methods and data availability.",
        f"The domain of {topic} continues to evolve rapidly, with researchers proposing increasingly sophisticated approaches.",
        f"Scholarly interest in {topic} has intensified in recent years, yielding a rich body of literature worthy of systematic examination.",
    ]
```

```
    intro = f
    if common_terms:
        theme_list = ", ".join(common_terms[:4]) if len(common_terms) > 4 else ", ".join(common_terms[:-1]) + f", and {common_terms[-1]}" if len(common_terms) > 1 else common_terms[0]
        intro += f"The reviewed studies address multifaceted aspects of {topic}, with recurring emphasis on {theme_list}. "
    else:
        intro += f"The reviewed studies address various dimensions of {topic}. "
    intro +=  + topic +
    # Add paper list with in-text citations
    for idx, paper in enumerate(papers, 1):
        authors = paper.get("authors", ["Unknown"])
        if len(authors) >= 2:
            first_author = authors[0].split()[-1] if authors[0] else "Unknown"
            author_cite = f"{first_author} et al."
        else:
            author_cite = authors[0].split()[-1] if authors else "Unknown"
        year = paper.get("year", "n.d.")
        title = paper.get("title", "Untitled")[:80]
        citations = paper.get("citation_count", 0)
        intro += f"\n{idx}. **{author_cite} ({year})**: _{title}_ [{citations} citations]"
    return intro
def generate_methodology_comparison(papers: List[Dict[str, Any]], cross_analysis: Dict[str, Any]) -> str:
    methods_data = cross_analysis.get("methodology_comparison", {})
    papers_methods = methods_data.get("papers", [])
    summary = methods_data.get("summary", {})
    import random
    num_papers = len(papers)
    openers = [
        f"The {num_papers} papers reviewed herein employ a range of methodological strategies, reflecting the evolving landscape of computational approaches in this domain.",
        f"Methodologically, the surveyed literature exhibits considerable diversity, with researchers drawing upon both established and emerging techniques.",
        f"A comparative assessment of the {num_papers} studies reveals varied methodological orientations, each tailored to specific problem characteristics.",
    ]
    section = f
    total = summary.get("total_papers", len(papers))
```

```
        # Build narrative paragraphs instead of bullet lists
        method_observations = []
        # Deep Learning usage
        dl_count = summary.get("using_deep_learning", 0)
        if dl_count > 0:
            pct = round(dl_count/total*100)
            method_observations.append(f"Deep learning constitutes the predominant paradigm, with {dl_count} of {total} studies ({pct}%) incorporating neural network architectures")
        # Transformer usage
        transformer_count = summary.get("using_transformer", 0)
        if transformer_count > 0:
            method_observations.append(f"Transformer-based models and attention mechanisms appear in {transformer_count} publications, underscoring the growing influence of self-attention architectures")
        # CNN usage
        cnn_count = summary.get("using_cnn", 0)
        if cnn_count > 0:
            method_observations.append(f"Convolutional neural networks remain relevant, appearing in {cnn_count} studies for feature extraction and pattern recognition")
        # RNN usage
        rnn_count = summary.get("using_rnn", 0)
        if rnn_count > 0:
# -------- Remaining code omitted for brevity --------
```

search_papers.py

```
import json
import os
import logging
import time
import math
from datetime import datetime
from typing import List, Dict, Optional, Any, Tuple
from semanticscholar import SemanticScholar
from dotenv import load_dotenv
# Configure logging
os.makedirs("data/logs", exist_ok=True)
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
```

```
    handlers=[
        logging.FileHandler('data/logs/search_papers.log'),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)
# Load environment variables
load_dotenv()
API_KEY = os.getenv("SEMANTIC_SCHOLAR_API_KEY")
# Tunables via environment
SEM_SCH_TIMEOUT = int(os.getenv("SEMANTIC_SCHOLAR_TIMEOUT", "12"))
SEM_SCH_MAX_RETRIES = int(os.getenv("SEMANTIC_SCHOLAR_MAX_RETRIES",
"2"))
USE_CACHED_ON_FAILURE = os.getenv("USE_CACHED_METADATA_ON_FAILURE",
"1") == "1"
# Initialize Semantic Scholar client with configurable timeout
sch = SemanticScholar(api_key=API_KEY, timeout=SEM_SCH_TIMEOUT)
#
================================================================
========
# RANKING WEIGHTS - Configurable scoring parameters
#
================================================================
========
RANKING_WEIGHTS = {
    "relevance": 0.30,      # w1: Topical relevance (title + abstract matching)
    "recency": 0.20,        # w2: Publication recency (newer = better)
    "citations": 0.25,      # w3: Citation count (normalized)
    "author_score": 0.10,   # w4: Author reputation (h-index proxy)
    "pdf_available": 0.15   # w5: PDF availability bonus
}
# Normalization constants
CURRENT_YEAR = datetime.now().year
MAX_RECENCY_YEARS = 10  # Papers older than this get minimum recency score
MAX_CITATIONS_NORM = 1000  # Citation count normalization ceiling
def calculate_relevance_score(paper: Dict[str, Any], query_terms: List[str]) -> float:
    title = (paper.get("title") or "").lower()
    abstract = (paper.get("abstract") or "").lower()
    if not query_terms:
        return 0.5  # Neutral score if no query terms
```

```
    title_matches = sum(1 for term in query_terms if term in title)
    abstract_matches = sum(1 for term in query_terms if term in abstract)
    # Title matches weighted higher (2x)
    title_score = min(1.0, (title_matches * 2) / len(query_terms))
    abstract_score = min(1.0, abstract_matches / len(query_terms))
    # Combined relevance (60% title, 40% abstract)
    relevance = 0.6 * title_score + 0.4 * abstract_score
    return round(relevance, 4)
def calculate_recency_score(year: Optional[int]) -> float:
    if not year:
        return 0.3  # Low score for unknown year
    years_old = CURRENT_YEAR - year
    if years_old <= 0:
        return 1.0  # Current or future year (preprints)
    elif years_old >= MAX_RECENCY_YEARS:
        return 0.1  # Minimum score for old papers
    else:
        # Linear decay with floor
        score = 1.0 - (years_old / MAX_RECENCY_YEARS) * 0.9
        return round(max(0.1, score), 4)
def calculate_citation_score(citation_count: Optional[int]) -> float:
    if not citation_count or citation_count <= 0:
        return 0.0
    # Logarithmic normalization to handle wide citation ranges
    log_citations = math.log10(citation_count + 1)
    log_max = math.log10(MAX_CITATIONS_NORM + 1)
    score = min(1.0, log_citations / log_max)
    return round(score, 4)
def calculate_author_score(authors: List[Any], influential_citations: int = 0) -> float:
    if not authors:
        return 0.2  # Low score for missing author info
    # Base score from author count (collaborative research is valued)
    author_count = len(authors)
    if author_count >= 3:
        collab_score = 0.6
    elif author_count >= 2:
        collab_score = 0.5
    else:
        collab_score = 0.4
    # Bonus for influential citations (proxy for author impact)
```

```
    if influential_citations:
        influence_bonus = min(0.4, influential_citations * 0.05)
    else:
        influence_bonus = 0.0
    return round(min(1.0, collab_score + influence_bonus), 4)
def calculate_paper_score(paper: Dict[str, Any], query_terms: List[str]) -> Tuple[float, Dict[str, float]]:
    w = RANKING_WEIGHTS
    # Calculate individual component scores
    relevance = calculate_relevance_score(paper, query_terms)
    recency = calculate_recency_score(paper.get("year"))
    citations = calculate_citation_score(paper.get("citation_count"))
    author_score = calculate_author_score(
        paper.get("authors", []),
        paper.get("influential_citation_count", 0)
    )
    pdf_score = 1.0 if paper.get("pdf_available") else 0.0
    # Weighted sum
    total_score = (
        w["relevance"] * relevance +
        w["recency"] * recency +
        w["citations"] * citations +
        w["author_score"] * author_score +
        w["pdf_available"] * pdf_score
    )
    component_scores = {
        "relevance": relevance,
        "recency": recency,
        "citations": citations,
        "author_score": author_score,
        "pdf_available": pdf_score,
        "total": round(total_score, 4)
    }
    return round(total_score, 4), component_scores
def rank_papers(papers: List[Dict[str, Any]], query: str, top_n: int = 3) -> List[Dict[str, Any]]:
    if not papers:
        return []
    # Extract query terms for relevance scoring
    query_terms = [term.lower().strip() for term in query.split() if len(term) > 2]
    # Calculate scores for all papers
```

```
    scored_papers = []
    for paper in papers:
        total_score, components = calculate_paper_score(paper, query_terms)
        paper_with_score = paper.copy()
        paper_with_score["ranking_score"] = total_score
        paper_with_score["score_breakdown"] = components
        scored_papers.append(paper_with_score)
    # Sort by total score (descending)
    scored_papers.sort(key=lambda x: x["ranking_score"], reverse=True)
    # Assign ranks
    for idx, paper in enumerate(scored_papers, 1):
        paper["rank"] = idx
    # Log ranking results
    logger.info("=" * 60)
    logger.info("PAPER RANKING RESULTS")
    logger.info("=" * 60)
    logger.info(f"Query: {query}")
    logger.info(f"Total papers scored: {len(scored_papers)}")
    logger.info(f"Top {top_n} papers:")
    for paper in scored_papers[:top_n]:
        logger.info(f"\n  Rank #{paper['rank']}: {paper.get('title', 'Unknown')[:60]}...")
        logger.info(f"    Total Score: {paper['ranking_score']:.4f}")
        breakdown = paper['score_breakdown']
        logger.info(f"    Breakdown - Relevance: {breakdown['relevance']:.3f}, "
                    f"Recency: {breakdown['recency']:.3f}, "
                    f"Citations: {breakdown['citations']:.3f}, "
                    f"Author: {breakdown['author_score']:.3f}, "
                    f"PDF: {breakdown['pdf_available']:.1f}")
    logger.info("=" * 60)
    # Return top N papers
    return scored_papers[:top_n] if top_n else scored_papers
def search_papers(
    topic: str,
    limit: int = 10,
    year_min: Optional[int] = None,
    year_max: Optional[int] = None,
    min_citations: Optional[int] = None,
    author: Optional[str] = None,
    fields: Optional[List[str]] = None
) -> Dict[str, Any]:
```

```
    logger.info(f"Starting paper search for topic: '{topic}' (limit={limit})")
    # Default fields to retrieve
    if fields is None:
        fields = [
            'title', 'abstract', 'year', 'authors', 'citationCount',
            'paperId', 'url', 'openAccessPdf', 'publicationDate',
            'venue', 'publicationTypes', 'externalIds', 'influentialCitationCount'
        ]
    # Build search query with filters
    query = topic
    if author:
        query += f" author:{author}"
# -------- Remaining code omitted for brevity --------
```

security.py

```
import logging
import os
import re
import time
import hashlib
from collections import defaultdict
from datetime import datetime, timedelta
from functools import wraps
from typing import Any, Callable, Dict, List, Optional, Union
from dataclasses import dataclass, field
# Setup logging for audit trail
LOG_DIR = "data/logs"
os.makedirs(LOG_DIR, exist_ok=True)
# Security audit log
audit_handler = logging.FileHandler(os.path.join(LOG_DIR, "security_audit.log"))
audit_handler.setFormatter(logging.Formatter(
    "%(asctime)s - SECURITY - %(levelname)s - %(message)s"
))
audit_logger = logging.getLogger("security.audit")
audit_logger.setLevel(logging.INFO)
audit_logger.addHandler(audit_handler)
audit_logger.addHandler(logging.StreamHandler())
#
================================================================
========
```

```
# PYDANTIC VALIDATION SCHEMAS
#
================================================================================
========
try:
    from pydantic import BaseModel, Field, field_validator, ConfigDict
    PYDANTIC_AVAILABLE = True
except ImportError:
    PYDANTIC_AVAILABLE = False
    audit_logger.warning("Pydantic not installed. Using fallback validation.")
if PYDANTIC_AVAILABLE:
    class SearchRequest(BaseModel):
        model_config = ConfigDict(strict=True, extra="forbid")
        topic: str = Field(
            ...,
            min_length=3,
            max_length=200,
            description="Research topic to search"
        )
        limit: int = Field(
            default=10,
            ge=1,
            le=20,
            description="Maximum papers to retrieve"
        )
        year_min: Optional[int] = Field(
            default=None,
            ge=1990,
            le=2030,
            description="Minimum publication year"
        )
        min_citations: Optional[int] = Field(
            default=None,
            ge=0,
            le=100000,
            description="Minimum citation count"
        )
        top_n: int = Field(
            default=3,
            ge=1,
```

```
            le=10,
            description="Number of top papers to select"
        )
        @field_validator("topic")
        @classmethod
        def sanitize_topic(cls, v: str) -> str:
            # Remove potentially dangerous characters
            sanitized = re.sub(r'[<>"\';\\`${}|]', '', v)
            # Normalize whitespace
            sanitized = ' '.join(sanitized.split())
            return sanitized.strip()
    class PaperMetadata(BaseModel):
        model_config = ConfigDict(extra="ignore")
        paper_id: str = Field(..., max_length=100)
        title: str = Field(..., max_length=500)
        authors: List[str] = Field(default_factory=list, max_length=50)
        abstract: Optional[str] = Field(default=None, max_length=10000)
        year: Optional[int] = Field(default=None, ge=1900, le=2030)
        citation_count: int = Field(default=0, ge=0)
        pdf_url: Optional[str] = Field(default=None, max_length=500)
        pdf_available: bool = Field(default=False)
    class CritiqueRequest(BaseModel):
        model_config = ConfigDict(extra="forbid")
        draft_id: Optional[str] = Field(default=None, max_length=100)
        sections: Optional[List[str]] = Field(default=None)
        @field_validator("sections")
        @classmethod
        def validate_sections(cls, v):
            if v is None:
                return v
            allowed = {"abstract", "introduction", "methodology_comparison",
                       "results_synthesis", "conclusion", "references"}
            for section in v:
                if section.lower() not in allowed:
                    raise ValueError(f"Invalid section: {section}")
            return v
#
================================================================
========
# RATE LIMITING
```

```
#
================================================================================
@dataclass
class RateLimitConfig:
    requests_per_minute: int = 30
    requests_per_hour: int = 300
    burst_limit: int = 10
    cooldown_seconds: int = 60
class RateLimiter:
    def __init__(self, config: RateLimitConfig = None):
        self.config = config or RateLimitConfig()
        self._buckets: Dict[str, List[float]] = defaultdict(list)
        self._cooldowns: Dict[str, float] = {}
    def _get_client_id(self, ip: str = None, user_id: str = None) -> str:
        identifier = f"{ip or 'unknown'}:{user_id or 'anonymous'}"
        return hashlib.sha256(identifier.encode()).hexdigest()[:16]
    def _cleanup_old_requests(self, client_id: str) -> None:
        now = time.time()
        hour_ago = now - 3600
        self._buckets[client_id] = [
            ts for ts in self._buckets[client_id] if ts > hour_ago
        ]
    def check_rate_limit(self, ip: str = None, user_id: str = None) -> Dict[str, Any]:
        client_id = self._get_client_id(ip, user_id)
        now = time.time()
        # Check cooldown
        if client_id in self._cooldowns:
            cooldown_end = self._cooldowns[client_id]
            if now < cooldown_end:
                retry_after = int(cooldown_end - now)
                audit_logger.warning(f"Rate limit: Client {client_id} in cooldown for {retry_after}s")
                return {
                    "allowed": False,
                    "reason": "rate_limit_exceeded",
                    "remaining": 0,
                    "reset_at": datetime.fromtimestamp(cooldown_end).isoformat(),
                    "retry_after": retry_after
                }
            else:
```

```
            del self._cooldowns[client_id]
        self._cleanup_old_requests(client_id)
        requests = self._buckets[client_id]
        # Check per-minute limit
        minute_ago = now - 60
        requests_last_minute = len([ts for ts in requests if ts > minute_ago])
        if requests_last_minute >= self.config.requests_per_minute:
            self._cooldowns[client_id] = now + self.config.cooldown_seconds
            audit_logger.warning(f"Rate limit exceeded: Client {client_id} -
{requests_last_minute}/min")
            return {
                "allowed": False,
                "reason": "minute_limit_exceeded",
                "remaining": 0,
                "reset_at": datetime.fromtimestamp(now + 60).isoformat(),
                "retry_after": 60
            }
        # Check per-hour limit
        if len(requests) >= self.config.requests_per_hour:
            audit_logger.warning(f"Rate limit exceeded: Client {client_id} - {len(requests)}/hour")
            return {
                "allowed": False,
                "reason": "hour_limit_exceeded",
                "remaining": 0,
                "reset_at": datetime.fromtimestamp(requests[0] + 3600).isoformat(),
                "retry_after": int(requests[0] + 3600 - now)
            }
        # Check burst limit (last 10 seconds)
        ten_seconds_ago = now - 10
        requests_burst = len([ts for ts in requests if ts > ten_seconds_ago])
        if requests_burst >= self.config.burst_limit:
            return {
                "allowed": False,
                "reason": "burst_limit_exceeded",
                "remaining": 0,
                "reset_at": datetime.fromtimestamp(now + 10).isoformat(),
                "retry_after": 10
            }
        # Request allowed - record it
        self._buckets[client_id].append(now)
```

```
        remaining = self.config.requests_per_minute - requests_last_minute - 1
        return {
            "allowed": True,
# -------- Remaining code omitted for brevity --------
```

workflow.py

```
import logging
import os
import json
from datetime import datetime
from typing import Any, Dict, List, Optional, TypedDict
from dataclasses import dataclass, field
# Setup logging
LOG_DIR = "data/logs"
os.makedirs(LOG_DIR, exist_ok=True)
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
    handlers=[
        logging.FileHandler(os.path.join(LOG_DIR, "workflow.log")),
        logging.StreamHandler(),
    ],
)
logger = logging.getLogger(__name__)
#
================================================================
========
# STATE DEFINITIONS
#
================================================================
========
class PipelineState(TypedDict, total=False):
    # Input
    topic: str
    limit: int
    year_min: Optional[int]
    min_citations: Optional[int]
    top_n: int
    # Search phase
    search_results: List[Dict[str, Any]]
```

```
    ranked_papers: List[Dict[str, Any]]
    selected_papers: List[Dict[str, Any]]
    # Download phase
    downloaded_papers: List[Dict[str, Any]]
    papers_with_text: List[Dict[str, Any]]
    # Analysis phase
    analysis_results: Dict[str, Any]
    cross_analysis: Dict[str, Any]
    # Writing phase
    sections: Dict[str, str]
    literature_review: Dict[str, Any]
    # Review phase
    critique: Dict[str, Any]
    revised_review: Dict[str, Any]
    final_draft: Dict[str, Any]
    # Meta
    errors: List[str]
    current_step: str
    status: str
    execution_log: List[Dict[str, Any]]
def create_initial_state(
    topic: str,
    limit: int = 10,
    year_min: Optional[int] = None,
    min_citations: Optional[int] = None,
    top_n: int = 3
) -> PipelineState:
    return PipelineState(
        topic=topic,
        limit=limit,
        year_min=year_min,
        min_citations=min_citations,
        top_n=top_n,
        search_results=[],
        ranked_papers=[],
        selected_papers=[],
        downloaded_papers=[],
        papers_with_text=[],
        analysis_results={},
        cross_analysis={},
```

```
        sections={},
        literature_review={},
        critique={},
        revised_review={},
        final_draft={},
        errors=[],
        current_step="start",
        status="initialized",
        execution_log=[]
    )
def log_step(state: PipelineState, step_name: str, status: str, details: Dict[str, Any] = None) ->
None:
    entry = {
        "step": step_name,
        "status": status,
        "timestamp": datetime.now().isoformat(),
        "details": details or {}
    }
    state["execution_log"].append(entry)
    logger.info(f"[{step_name}] {status} - {details}")
#
==========================================================================
========
# WORKFLOW NODES (per PDF architecture)
#
==========================================================================
========
def process_input(state: PipelineState) -> PipelineState:
    log_step(state, "process_input", "started")
    # Validate topic
    topic = state.get("topic", "").strip()
    if not topic or len(topic) < 3:
        state["errors"].append("Topic must be at least 3 characters")
        state["status"] = "error"
        return state
    # Normalize parameters
    state["limit"] = min(max(int(state.get("limit", 10)), 5), 20)
    state["top_n"] = min(max(int(state.get("top_n", 3)), 1), 10)
    if state.get("year_min"):
        state["year_min"] = max(int(state["year_min"]), 2000)
```

```
    if state.get("min_citations"):
        state["min_citations"] = max(int(state["min_citations"]), 0)
    state["current_step"] = "process_input"
    log_step(state, "process_input", "completed", {"topic": topic, "limit": state["limit"]})
    return state
def planner(state: PipelineState) -> PipelineState:
    log_step(state, "planner", "started")
    # Define execution plan based on input
    plan = {
        "search_strategy": "semantic_scholar",
        "ranking_criteria": ["pdf_available", "relevance", "recency", "citations", "author_score"],
        "sections_to_generate": ["abstract", "introduction", "methodology_comparison",
"results_synthesis", "conclusion", "references"],
        "enable_critique": True,
        "enable_revision": True
    }
    state["execution_plan"] = plan
    state["current_step"] = "planner"
    log_step(state, "planner", "completed", plan)
    return state
def researcher(state: PipelineState) -> PipelineState:
    log_step(state, "researcher", "started")
    try:
        from modules.search_papers import search_papers
        results = search_papers(
            topic=state["topic"],
            limit=state["limit"],
            year_min=state.get("year_min"),
            min_citations=state.get("min_citations")
        )
        state["search_results"] = results.get("papers", [])
        state["current_step"] = "researcher"
        log_step(state, "researcher", "completed", {"papers_found": len(state["search_results"])})
    except Exception as e:
        state["errors"].append(f"Search failed: {str(e)}")
        state["status"] = "error"
        log_step(state, "researcher", "failed", {"error": str(e)})
    return state
def article_decisions(state: PipelineState) -> PipelineState:
    log_step(state, "article_decisions", "started")
```

```
    try:
        from modules.search_papers import rank_papers
        papers = state.get("search_results", [])
        if not papers:
            state["errors"].append("No papers to rank")
            return state
        # Rank all papers
        ranked = rank_papers(papers, state["topic"], top_n=len(papers))
        state["ranked_papers"] = ranked
        # Select top N papers WITH PDF availability (mandatory per PDF spec)
        papers_with_pdf = [p for p in ranked if p.get("pdf_available")]
        state["selected_papers"] = papers_with_pdf[:state["top_n"]]
        state["current_step"] = "article_decisions"
        log_step(state, "article_decisions", "completed", {
            "total_ranked": len(ranked),
            "with_pdf": len(papers_with_pdf),
            "selected": len(state["selected_papers"])
        })
    except Exception as e:
        state["errors"].append(f"Ranking failed: {str(e)}")
        log_step(state, "article_decisions", "failed", {"error": str(e)})
    return state
def download_articles(state: PipelineState) -> PipelineState:
    log_step(state, "download_articles", "started")
    try:
        from modules.download_pdf import download_papers, save_download_metadata
        papers = state.get("selected_papers", [])
        if not papers:
            state["errors"].append("No papers to download")
            return state
        downloaded = download_papers(papers)
        save_download_metadata(downloaded, "selected_papers.json")
        state["downloaded_papers"] = downloaded
        state["current_step"] = "download_articles"
        log_step(state, "download_articles", "completed", {"downloaded": len(downloaded)})
# -------- Remaining code omitted for brevity --------
```

run.py

```
import os
import sys
```

```
import argparse
# Add project root to path
ROOT = os.path.dirname(os.path.abspath(__file__))
if ROOT not in sys.path:
    sys.path.insert(0, ROOT)
# Change to project root
os.chdir(ROOT)
def run_check_imports():
    print("\n" + "="*60)
    print("Checking Dependencies...")
    print("="*60)
    from scripts.check_imports import main
    main()
def run_workflow_test():
    print("\n" + "="*60)
    print("Running Workflow Test...")
    print("="*60)
    import test_workflow
    test_workflow.main()
def run_full_pipeline(topic, limit, min_year, min_citations):
    print("\n" + "="*60)
    print("Running Full Pipeline...")
    print("="*60)
    from scripts.prepare_dataset import create_dataset, validate_dataset
    stats = create_dataset(
        topic=topic,
        limit=limit,
        year_min=min_year,
        min_citations=min_citations,
        download_pdfs=True
    )
    validate_dataset()
    return stats
def run_flask_ui():
    print("\n" + "="*60)
    print("Launching Web UI...")
    print("="*60)
    from ui.app_flask import app
    print("\nOpen http://127.0.0.1:5000 in your browser")
    print("Press Ctrl+C to stop\n")
```

```
    app.run(host='127.0.0.1', port=5000, debug=False)
def run_gradio_ui():
    print("\n" + "="*60)
    print("Launching Gradio UI...")
    print("="*60)
    print("\nNote: Gradio may have compatibility issues with Python 3.13")
    print("If you encounter errors, use the Flask UI instead (--flask)\n")
    from ui.app import demo
    demo.launch(server_name="127.0.0.1", server_port=7860, share=False)
def main():
    parser = argparse.ArgumentParser(
        description="AI Paper Review System - Main Entry Point",
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog=
    )
    parser.add_argument('--check', action='store_true',
                        help='Check if all dependencies are installed')
    parser.add_argument('--test', action='store_true',
                        help='Run workflow test to verify modules')
    parser.add_argument('--ui', '--flask', action='store_true',
                        help='Launch Flask web UI')
    parser.add_argument('--gradio', action='store_true',
                        help='Launch Gradio web UI')
    parser.add_argument('--run', action='store_true',
                        help='Run the full pipeline')
    # Pipeline arguments
    parser.add_argument('--topic', type=str,
                        default='deep learning natural language processing',
                        help='Research topic to search')
    parser.add_argument('--limit', type=int, default=10,
                        help='Maximum number of papers (default: 10)')
    parser.add_argument('--min-year', type=int, default=2020,
                        help='Minimum publication year (default: 2020)')
    parser.add_argument('--min-citations', type=int, default=50,
                        help='Minimum citation count (default: 50)')
    args = parser.parse_args()
    # If no arguments, show help
    if len(sys.argv) == 1:
        print("\n" + "="*60)
        print("AI Paper Review System")
```

```
        print("="*60)
        print("\nAvailable commands:")
        print("  --check     : Check dependencies")
        print("  --test      : Run workflow test")
        print("  --ui        : Launch Flask web UI")
        print("  --gradio    : Launch Gradio web UI")
        print("  --run       : Run full pipeline")
        print("\nFor more options, run: python run.py --help")
        return
    # Execute requested action
    if args.check:
        run_check_imports()
    elif args.test:
        run_workflow_test()
    elif args.ui:
        run_flask_ui()
    elif args.gradio:
        run_gradio_ui()
    elif args.run:
        run_full_pipeline(args.topic, args.limit, args.min_year, args.min_citations)
    else:
        parser.print_help()
if __name__ == '__main__':
    main()
```

app_modern.py

```
import os
import sys
import json
import time
import logging
import tempfile
import re
from datetime import datetime
from typing import Dict, Any, List, Optional, Tuple
from pathlib import Path
# Ensure project root on path
ROOT = Path(__file__).parent.parent
if str(ROOT) not in sys.path:
```

```
    sys.path.insert(0, str(ROOT))
os.chdir(ROOT)
# Setup logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)
import gradio as gr
# Import modules
from modules.search_papers import search_papers, rank_papers, save_metadata
from modules.download_pdf import download_papers, save_download_metadata
from modules.extract_text import extract_text_for_papers
from modules.analyze_text import process_analysis
from modules.generate_draft import generate_drafts
from modules.critique_draft import critique_drafts
from urllib.parse import urlparse
#
================================================================================
========
# CUSTOM CSS — Dark Academic Theme
#
================================================================================
========
CUSTOM_CSS =
#
================================================================================
========
# PROGRESS STATE
#
================================================================================
========
PIPELINE_STEPS = [
    {"id": "search",   "label": "Search",   "icon": ""},
    {"id": "rank",     "label": "Rank",     "icon": ""},
    {"id": "download", "label": "Download", "icon": "↓"},
    {"id": "extract",  "label": "Extract",  "icon": ""},
    {"id": "analyze",  "label": "Analyze",  "icon": ""},
    {"id": "write",    "label": "Write",    "icon": "✍"},
    {"id": "critique", "label": "Critique", "icon": ""},
    {"id": "finalize", "label": "Complete", "icon": "✔"},
]
def create_progress_html(current_step: int, status: str = "running") -> str:
```

```
    html = '<div class="progress-stepper">'
    for idx, step in enumerate(PIPELINE_STEPS):
        if idx < current_step:
            state_class = "step-complete"
            icon = "✓"
        elif idx == current_step:
            if status in ("running", "pending"):
                state_class = "step-active"
                icon = "●"
            elif status == "error":
                state_class = "step-error"
                icon = "✕"
            else:
                state_class = "step-complete"
                icon = "✓"
        else:
            state_class = "step-pending"
            icon = str(idx + 1)
        html += f
    html += '</div>'
    return html
#
================================================================================
========
# PIPELINE EXECUTION
#
================================================================================
========
def run_review_pipeline(
    topic: str,
    keywords: str,
    num_papers: int,
    progress=gr.Progress()
) -> Tuple[Dict, str, str, str, str, str, str, str]:
    results = {
        "status": "running",
        "current_step": 0,
        "papers_found": 0,
        "papers_selected": 0,
        "errors": []
```

```
}
search_query = topic
if keywords and keywords.strip():
    search_query += " " + keywords.replace(",", " ")
PREFERRED_HOSTS = (
    "arxiv.org", "biorxiv.org", "medrxiv.org",
    "pdfs.semanticscholar.org", "s2-", "semanticscholar.org",
    "openreview.net", "core.ac.uk", "openaccess.thecvf.com",
    "aclanthology.org", "ceur-ws.org", "hal.science",
    "eprints", "researchgate.net",
)
BLOCKED_HOSTS = (
    "ieeexplore.ieee.org", "dl.acm.org", "link.springer.com",
    "wiley.com", "sciencedirect.com", "onlinelibrary.wiley.com",
    "tandfonline.com", "emerald.com",
)
def get_host(url: str) -> str:
    try:
        return urlparse(url).netloc.lower()
    except Exception:
        return ""
def is_preferred_host(url: str) -> bool:
    host = get_host(url)
    return any(ph in host for ph in PREFERRED_HOSTS)
def is_blocked_host(url: str) -> bool:
    host = get_host(url)
    return any(bh in host for bh in BLOCKED_HOSTS)
def is_acceptable_pdf(p: Dict[str, Any]) -> bool:
    if not p.get("pdf_available"):
        return False
    url = p.get("pdf_url") or ""
    return bool(url) and not is_blocked_host(url)
def pdf_priority(p: Dict[str, Any]) -> int:
    url = p.get("pdf_url") or ""
    if is_preferred_host(url) or "arxiv" in url.lower():
        return 0
    elif is_blocked_host(url):
        return 99
    return 1
try:
```

```
# Step 1: Search
progress(0.1, desc="🔍 Searching for papers...")
results["current_step"] = 0
all_papers: List[Dict[str, Any]] = []
seen_ids: set = set()
search_results = search_papers(
    topic=f"{search_query} open access",
    limit=30, year_min=2018, min_citations=3
)
for p in search_results.get("papers", []):
    pid = p.get("paper_id")
    if pid and pid not in seen_ids:
        all_papers.append(p); seen_ids.add(pid)
progress(0.12, desc="🔍 Searching arxiv...")
arxiv_results = search_papers(
    topic=f"{search_query} arxiv",
    limit=20, year_min=2018, min_citations=0
)
for p in arxiv_results.get("papers", []):
    pid = p.get("paper_id")
    if pid and pid not in seen_ids:
        all_papers.append(p); seen_ids.add(pid)
if len(all_papers) < 15:
    progress(0.14, desc="🔍 Fallback search...")
    fallback_results = search_papers(
        topic=search_query, limit=25, year_min=2015, min_citations=3
    )
    for p in fallback_results.get("papers", []):
        pid = p.get("paper_id")
        if pid and pid not in seen_ids:
            all_papers.append(p); seen_ids.add(pid)
results["papers_found"] = len(all_papers)
if not all_papers:
    return (
        {"status": "error", "message": "No papers found"},
        create_progress_html(0, "error"),
        "No papers found for this topic.", "", "", "", "", ""
    )
# Step 2: Rank
progress(0.2, desc="📊 Ranking papers...")
```

```
        results["current_step"] = 1
        papers_with_pdf = [p for p in all_papers if is_acceptable_pdf(p)]
        papers_with_pdf.sort(key=pdf_priority)
        ranked_papers = rank_papers(papers_with_pdf, search_query, top_n=min(20,
len(papers_with_pdf)))
        ranked_papers.sort(key=lambda p: (pdf_priority(p), -(p.get("rank", 0))))
        pdf_candidates = ranked_papers[:15] if len(ranked_papers) > 5 else ranked_papers
        if not pdf_candidates:
            return (
                {"status": "error", "message": "No papers with PDFs available"},
                create_progress_html(1, "error"),
                "No papers with downloadable PDFs found.", "", "", "", "", ""
            )
        save_metadata({
            "topic": search_query,
            "timestamp": datetime.now().isoformat(),
            "papers_found": len(all_papers),
            "papers_with_pdf": len(papers_with_pdf),
            "papers": ranked_papers
# -------- Remaining code omitted for brevity --------
```