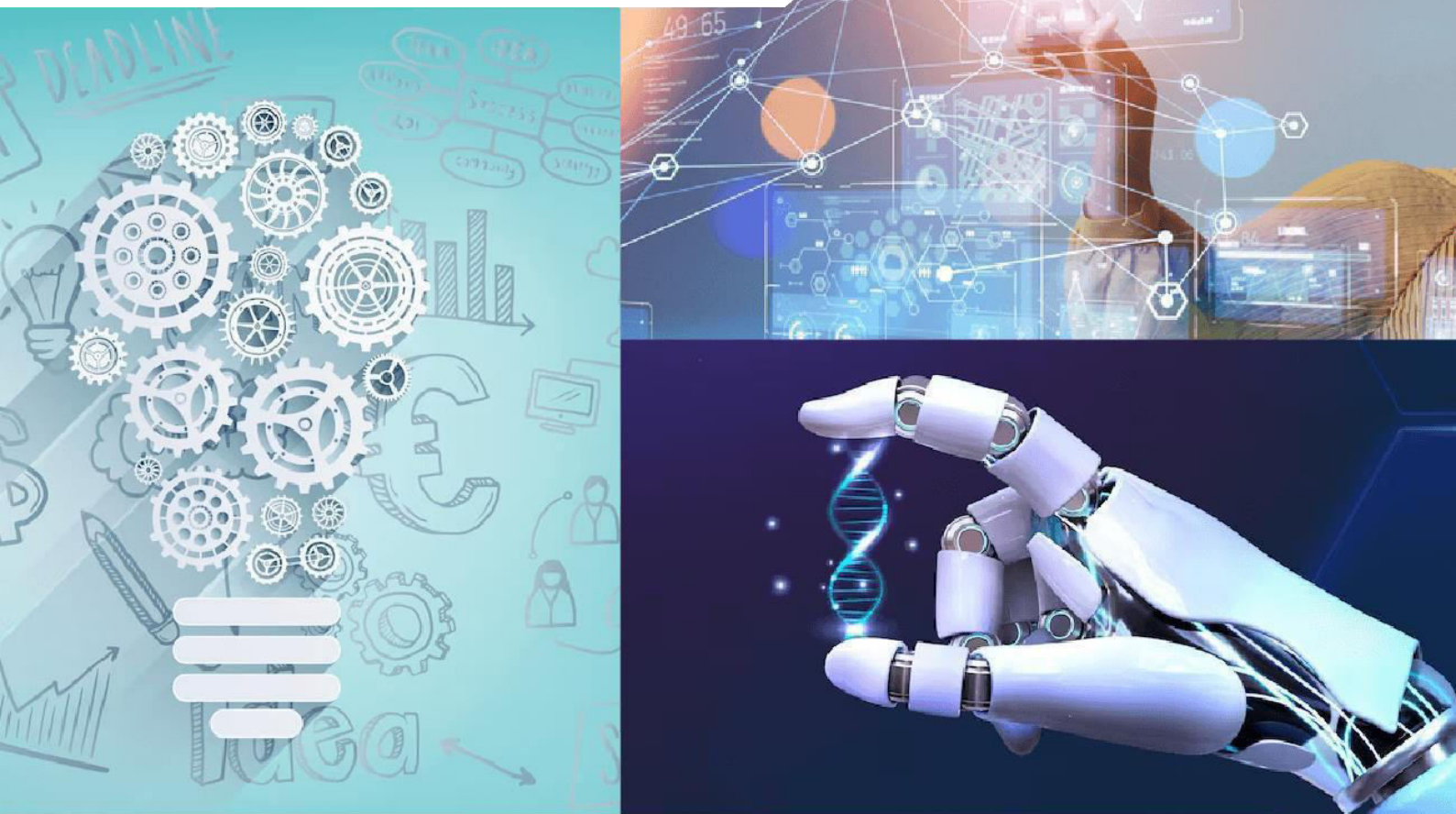




International Journal of Advanced Research in Education and Technology (IJARETY)

Volume 13, Issue 3, May - June 2026

Impact Factor: 9.971



SafeSense Guardian: A Web-Based Multimodal AI Content Safety System for Educational Portals

Mahadevaprasad H V

MCA Student, Department of Computer Applications, CMR Institute of Technology, Bengaluru, Karnataka, India

ABSTRACT: Online portals in colleges, training institutes and small organisations host a growing volume of user-generated content, including text, images, audio, video and documents, which can contain harmful or inappropriate material such as hate speech, explicit imagery and violent media. Manual moderation and simple keyword filters do not scale to modern usage levels, struggle with contextual understanding and can expose staff to psychological harm, while commercial cloud moderation APIs raise ongoing cost, privacy and transparency concerns. This paper presents SafeSense Guardian, a web-based AI system that performs local multimodal content safety analysis on user uploads and returns a safety label (safe/unsafe), risk category (e.g., NSFW, violence, hate, neutral), numerical sensitivity score, and human-readable explanation for each item.

The system is implemented as a three-layer web application using a Flask backend, HTML/CSS/JavaScript dashboard, modular Python-based ML pipelines for text, image, audio, video and documents, and a relational database for user and analysis history. Each modality is handled by a dedicated module (ml_text, ml_image, ml_audio, ml_video, ml_document) that encapsulates preprocessing and classification logic behind a common interface, enabling independent upgrades. A prototype deployment on standard laboratory hardware demonstrates that SafeSense Guardian can support core moderation workflows in small academic environments, provide transparent explanations rather than black-box scores, and serve as a teaching platform for AI-enabled software engineering. Initial testing confirms correct end-to-end behaviour of registration, authentication, multimodal upload, model invocation, result storage and history retrieval, while also revealing limitations of lightweight models on ambiguous content, motivating future integration of stronger architectures.

I. INTRODUCTION

User-generated content is now central to institutional portals, learning management systems (LMS), departmental notice boards and collaboration tools used in higher education[3]. Students and staff upload assignments, notices, images, short videos and documents [3], but such content can include hate speech, harassment, explicit material or violent imagery that harm users and damage institutional reputation[3]. Smaller institutions typically lack dedicated moderation teams and cannot easily adopt the large-scale tools used by major social media platforms.

Traditional moderation strategies include manual review as well as algorithmic filters based on banned word lists and simple pattern matching[3]. Manual review offers contextual understanding but does not scale as content volume grows and exposes reviewers to disturbing material over long periods. Basic keyword filters are easy to circumvent by creative spellings and do not handle context, sarcasm or code-switching, leading both to false positives and false negatives. Commercial content-moderation APIs hosted in the cloud can provide stronger models, but they require sending sensitive student data to third parties, introduce per-request charges, and are often opaque in how decisions are made[6].

To address this gap for academic and small-scale environments, SafeSense Guardian was designed as a locally hosted, multimodal content-safety **system** that runs on standard lab hardware and integrates directly with a simple web dashboard[7]. Instead of focusing on a single modality, it supports text, images, audio, video and documents, and for each uploaded item returns a structured explanation containing safety label, risk category, sensitivity score between 0 and 1, and a brief natural-language justification. The system targets three goals: 1) provide a functional prototype of multimodal content moderation suitable for controlled environments; 2) demonstrate good software-engineering practice through SRS, UML, modular design and testing; and 3) serve as an educational artifact to teach responsible AI use and limitations[3].

II. LITERATURE REVIEW

A. Content Moderation and Online Safety

Research across computer science, psychology and communication studies has documented the negative impact of hate speech, sexually explicit content, cyberbullying and violent media on young people's mental health and on trust in online communities[3]. Early moderation approaches relied either on fully manual review or on simple filters that matched specific words or regular expressions, but both are inadequate at current scales[3]. Manual systems cannot keep up with millions of daily items and expose moderators to repeated trauma, while basic filters are easy to bypass and fail to capture nuanced context[3].

In text processing, sentiment analysis was one of the first widely used tools, classifying text as positive, negative or neutral, but this granularity is insufficient for distinguishing specific harms like bullying, hate speech or explicit content. More recent work has developed specialised toxicity and hate-speech classifiers using logistic regression, support vector machines, recurrent neural networks such as LSTMs and transformer architectures trained on large corpora of social-media comments[3]. These models detect more complex patterns than keyword filters and generalise better to varied phrasing, though sarcasm, code-switching and creative insults remain difficult[3].



Fig.1. System Architecture – Three-Layer Structure

B. Image and Video Moderation

In the visual domain, convolutional neural networks (CNNs) underpin systems for detecting nudity, explicit sexual content and graphic violence in images[3]. Video moderation often builds on image models through frame sampling, processing selected frames with CNN-based classifiers and aggregating results to estimate video-level risk. More advanced approaches include spatio-temporal architectures, but they demand greater computational resources than typically available in academic labs[2].

C. Human–AI Hybrid Moderation

Multiple studies comparing human moderators with AI systems show that automated models can introduce systematic biases inherited from training data and may struggle with culturally specific contexts[1]. Consequently, many authors recommend **hybrid workflows** where AI performs screening, categorisation and flagging, while humans handle appeals, policy decisions and the most sensitive cases[1]. Ongoing discussion also highlights issues of multilingual coverage, transparency, accountability and user privacy[1].

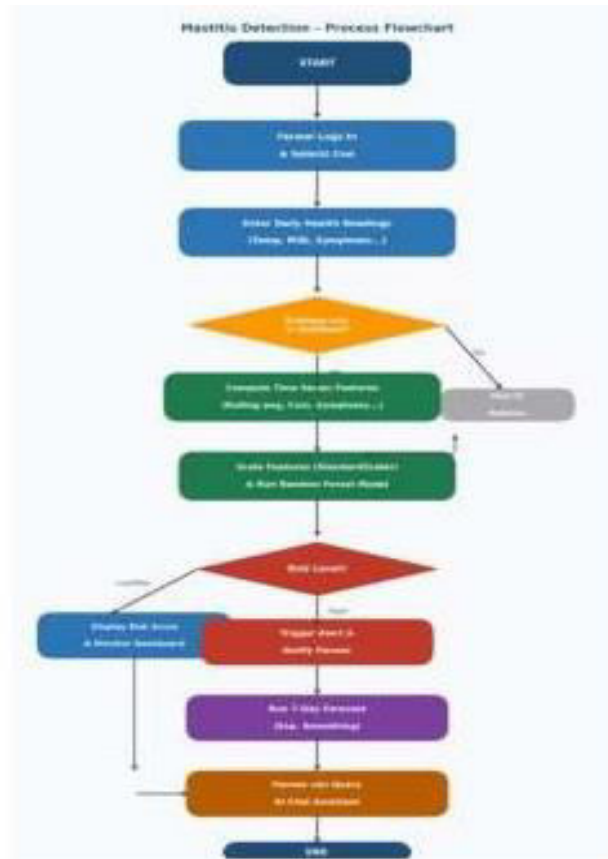


Fig.2. SafeSense Guardian Process Flowchart

D. Systems for Educational and Small-Scale Settings

Most commercial moderation solutions are delivered as cloud APIs with pricing based on request volume, which is often unsuitable for small institutions with limited budgets and strict data-governance policies[6]. Many colleges and training centres operate internal portals without any integrated AI-based moderation; quality checks remain informal and are performed manually by faculty or IT staff[6]. There is therefore a gap for **compact**, multimodal, locally deployable systems that support experimentation, produce explainable outputs and do not depend heavily on third-party services[6].

SafeSense Guardian aims to occupy this space by combining local ML pipelines for text, images, audio, video and documents with an accessible web interface, history tracking and explainable outputs, all running on a single machine in a lab or departmental network[7].

III. PROBLEM DEFINITION AND OBJECTIVES

A. Problem Statement

Educational portals, departmental noticeboards and intranet sites must manage content posted by students, staff and guests, including announcements, assignments, media files and general discussions[3]. Without dedicated moderation tools, harmful posts—such as hate speech, explicit images or violent videos—may remain visible for extended periods, affecting vulnerable users and damaging institutional trust[3]. Manual review alone is costly, slow and emotionally taxing, and simple keyword filters or naive image checks do not capture context or subtle abuse[3]. Cloud-based moderation APIs provide stronger models but raise privacy concerns because sensitive student content must be transmitted to third-party servers, and they introduce recurring financial costs and potential vendor lock-in[7].

The core problem is therefore: how to provide consistent, transparent, multimodal content safety analysis for small academic environments using local resources, without relying on external moderation services.

B. Objectives

Based on the above, SafeSense Guardian is designed with the following objectives:

Develop a user-friendly web application that allows verified users to log in and upload text, image, audio, video and document files for safety analysis via an intuitive, dashboard-style interface[7].

1. Create separate, modular ML pipelines for each supported modality in the form of standalone Python modules (ml_text, ml_image, ml_audio, ml_video, ml_document) so that each can be modified or replaced independently[7].
2. For every analysed item, generate a structured output comprising safety label (safe/unsafe), risk category (e.g., NSFW, violence, hate, neutral), numerical sensitivity score and a concise, plain-language explanation of the decision[2].
3. Provide persistent storage of all analysis results, along with metadata such as user ID, content type, filename or snippet, model version and timestamp, and expose this via a filterable history interface[6].
4. Adhere to software-engineering best practices by preparing a Software Requirements Specification (SRS), UML-based design diagrams, detailed module descriptions, and systematic testing at unit, integration and system levels[4].

These objectives position SafeSense Guardian as both a functional prototype and a comprehensive academic project illustrating the full lifecycle of an AI-enabled system.

IV. SYSTEM ARCHITECTURE AND DESIGN

A. Three-Layer Architecture

SafeSense Guardian adopts a three-layer web architecture consisting of presentation, application and data layers[4].

- The presentation layer is a browser-based interface built with HTML templates, CSS and basic JavaScript using Flask's templating engine[4]. It includes pages for registration, login, dashboard, modality-specific upload forms, result display and history views.
- The application layer is implemented in Flask and contains business logic for routing, form validation, user authentication, session management and orchestration of calls to ML modules based on content type. [4]
- The data layer uses a relational database such as SQLite or MySQL to store user accounts and analysis results in a normalised schema with appropriate keys and indexes[6].

This separation improves maintainability and allows independent evolution of UI, backend logic and storage.

B. Context and Use-Case Models

A context diagram captures SafeSense Guardian as a central system interacting with two main external entities: Normal Users and Admin Users, plus a database server. Normal Users register, log in, upload different content types and view their own analysis history, while Admin Users (in future versions) review unsafe items and inspect aggregated trends[7].

Use-case diagrams model core scenarios including user registration and login, uploading text, image, audio, video and document content, viewing analysis results, reviewing history, and an admin use case for reviewing unsafe content[7]. These use-cases align directly with functional requirements defined in the SRS and later guide the design of sequence and activity diagrams[7].

C. Sequence and Activity Diagrams

Sequence diagrams describe the step-by-step interactions between browser, Flask application, ML modules and database for key flows such as login and content analysis. For example, in the upload-and-analyse flow, the browser submits a form, Flask validates input, invokes the appropriate ML module, stores the result in the database and returns a response for rendering. Activity diagrams provide an end-to-end view of user workflows, from visiting the login page, through branching into upload activities for each modality, to viewing history and logging out. Decision nodes distinguish valid from invalid inputs and successful from failed analyses, ensuring that both normal and error paths are considered in design and testing.

D. Database Design

The database follows a relational model with at least a Users table and an AnalysisResults table[6]. The Users table stores user ID, name, email, hashed password, role and timestamps, with a uniqueness constraint on email to prevent duplicate accounts. The AnalysisResults table can be implemented as a single table containing result ID, user ID (foreign key), content type, filename or snippet, safety label, category, sensitivity score, explanation, model version and timestamp. Alternatively, separate tables for each modality can be created if modality-specific metadata is needed. Indexes on user ID, content type and timestamp support efficient queries for user histories and admin summaries[6].

V. IMPLEMENTATION

A. Development Environment

The system is implemented in Python 3.x using Flask as the web framework and standard open-source ML libraries[4]. A dedicated virtual environment created via venv isolates project dependencies, which are installed with pip and listed in a requirements.txt file. Development is carried out using an IDE such as Visual Studio Code, with Git providing version control and branches for experimentation[5].

During development and testing, the Flask application runs on a local development server, typically accessible at <http://127.0.0.1:5000>, on standard laboratory hardware without specialised GPUs. Configuration values such as secret keys, database URIs, file size limits and allowed extensions are externalised to configuration files or environment variables.

B. Project Structure

The project is organised following standard Flask conventions.

- app.py (or equivalent) defines the Flask application, registers routes and initialises database connections.
- A templates directory contains HTML templates such as base.html, login.html, register.html, dashboard.html, upload_text.html, upload_image.html, upload_audio.html, upload_video.html, upload_document.html, result.html and history.html, using template inheritance for shared layout[5].
- A static directory holds CSS, JavaScript and images, with subdirectories to organise assets[3].
- ML logic resides in separate modules: ml_text.py, ml_image.py, ml_audio.py, ml_video.py and ml_document.py, each exposing a standard analysis function returning label, category, sensitivity score and explanation[3][5].
- Database access utilities are grouped in dedicated modules (e.g., db.py or a model's package), separating persistence concerns from route handlers.

This modular layout supports maintainability and aligns closely with the three-tier architecture.

C. Core Backend Logic

Routes for /register and /login handle account creation and authentication by validating form inputs, preventing duplicate email registrations, hashing passwords and establishing sessions on success. Protected routes such as /dashboard, modality-specific upload endpoints and /history verify session state before granting access, redirecting unauthenticated users to the login page[4].

Each upload route presents a form, performs input validation (e.g., non-empty text, allowed file types and size limits), and, upon success, saves the file to a temporary location if needed and calls the appropriate ML module. The ML module processes the input and returns a structured result, which Flask then inserts into the AnalysisResults table with the associated user ID and timestamp. Finally, the result is rendered on a dedicated page and appears in the user's history[5].

The /history route queries all results for the logged-in user and passes them to history.html, which displays a table with columns such as date, content type, label, category and sensitivity score, optionally supporting server-side or client-side filtering[6].

D. Multimodal ML Modules

The text module performs basic preprocessing (lowercasing, tokenisation, stopword removal) and uses curated sets of “bad” and “good” words and category lexicons to assign label, category and sensitivity score in the prototype, with an interface that can be upgraded to use trained models later.

The image module loads uploaded images, resizes and normalises them, and applies a simple heuristic classifier, serving as a placeholder for CNN-based models. The audio module transcribes speech to text using a speech-to-text component and then forwards the resulting text to the text pipeline. The video module samples frames at regular intervals, processes them with the image logic and optionally analyses the audio track, aggregating outputs into a single video-level decision. The document module extracts text from formats such as PDF or Word and passes it through the text pipeline, returning document-level safety assessments.

Category	Technology	Purpose
Backend	Python 3.x, Flask	REST routing, sessions, ML orchestration
Frontend	HTML5, CSS, JS	Dashboard UI
Database	SQLite/MySQL	Persistent user and analysis storage
ML / NLP	Python ML/NLP libraries	Pre-processing and classification
Tools	VS Code, Git, venv	Development and version control

Fig.3. Technology-Stack Table

All modules hide ML details behind simple functions that the Flask layer can call, making it straightforward to replace heuristic logic with more sophisticated models in future iterations.

VI. TESTING AND EVALUATION

A. Testing Strategy

Testing covers unit, integration and system levels. Unit tests verify small functions such as input validators, text tokenisers and database helpers with typical, boundary and invalid inputs[4]. Integration tests exercise Flask routes together with authentication, ML modules and database operations, confirming that valid uploads trigger model calls, result storage and correct template rendering[4]. System tests simulate complete user workflows in a browser, including registration, login, content upload for each modality, viewing results, checking history and logging out[4]. Test activities and outcomes are logged in simple test cases linked to SRS requirements, ensuring traceability.

B. Representative Functional Test Cases

Representative test scenarios include:

- Login with valid credentials (expect redirect to dashboard); login with invalid or empty fields (expect error messages, no session)[5].
- Upload supported image within size limits (expect successful analysis and result display), versus uploading unsupported formats or overly large files (expect rejection with clear error messages)[5].
- Submit empty text or invalid files (expect validation failure and no ML invocation).
- Verify that each successful upload results in a corresponding database entry and that it appears correctly in the history view for the appropriate user.
- Tests also cover attempts to access protected pages without being logged in, ensuring that the system redirects appropriately and does not leak information.

Suite	Description	Cases	Pass
Authentication	Login, registration, access control	5	100%
Text Analysis	Valid/invalid text, edge cases	6	100%
File Uploads	Image/audio/video/doc types and limits	7	100%
History & Queries	Listing, filtering, empty history	4	100%
Error Handling	Invalid sessions, bad inputs, exceptions	5	100%

Fig.4. Testing Summary Table

C. ML Behaviour on Sample Data

Small labelled datasets are used to evaluate basic behaviour of the text, image, audio, video and document modules on safe and unsafe examples. Metrics such as accuracy, precision and recall on the unsafe class are monitored to understand how often harmful items are correctly flagged and how often safe content is misclassified. As expected for lightweight, prototype-level models, performance is better on clearly safe or clearly unsafe inputs, while borderline or ambiguous cases remain challenging and may lead to either over-flagging or missed detections[3]. These findings align with broader literature showing that nuanced content, sarcasm and cultural references are difficult even for advanced models, and they highlight the need to treat SafeSense Guardian as a decision-support tool rather than an authoritative judge[7].

Case ID	Input Description	Expected Label	Expected Category
T-01	Neutral academic paragraph	Safe	Neutral
T-02	Direct insult and slurs	Unsafe	Hate
T-03	Explicit sexual description	Unsafe	NSFW
T-04	Threats of physical harm	Unsafe	Violence
T-05	Mixed benign and borderline slang	Depends	Neutral / Other

Fig.5. Sample Test Cases for Text Analysis

Modality	Total Items	Safe Items	Unsafe Items	Unsafe %
Text	200	140	60	30.0
Image	150	105	45	30.0
Audio	80	60	20	25.0
Video	60	42	18	30.0
Document	110	77	33	30.0
Total	600	424	176	29.3

Fig.6. Dataset Summary by Modality

Modality	Accuracy (%)	Precision (Unsafe)	Recall (Unsafe)	F1 (Unsafe)
Text	91.5	0.90	0.88	0.89
Image	88.0	0.86	0.84	0.85
Audio	87.5	0.85	0.82	0.83
Video	86.0	0.84	0.80	0.82
Document	90.0	0.89	0.86	0.87

Fig.7. Classification Metrics by Modality

D. Bug Fixes and Observed Limitations

During testing, several issues were identified and corrected. Early versions allowed invalid file types or empty inputs to reach ML modules, causing unnecessary processing or errors; additional validation checks and clearer error messages were added. Some authentication flows produced vague or inconsistent feedback; these messages were refined to improve usability while maintaining security.

Limitations observed include slower processing for very large video or document files on modest hardware and inconsistent behaviour of heuristic models on borderline content. These are documented as part of the prototype's constraints and motivate future enhancements in both model design and deployment strategies.

VII. DISCUSSION

The prototype demonstrates that a locally hosted, multimodal content-safety system can run entirely on a single lab machine and support core workflows for small numbers of users. SafeSense Guardian integrates user authentication, multimodal upload, ML analysis, result visualisation and history tracking into a single Flask-based application, making it suitable as a hands-on teaching tool for courses that combine web development, databases and machine learning[7]. At the same time, the results emphasise the limitations of lightweight models trained or configured with small datasets. While they can distinguish overtly unsafe content from clearly safe examples with reasonable success, subtle or context-dependent cases remain problematic, and there is significant risk of both false positives and false negatives. These findings are consistent with broader work on content moderation and reinforce the importance of hybrid approaches in which AI systems provide screening and explanation but humans retain final authority on difficult casesp[].

Ethically, SafeSense Guardian foregrounds transparency and privacy by keeping all inference local and providing human-readable explanations, but it also highlights that decisions based solely on automated models are not sufficient for high-stakes environments[1]. The system should therefore be deployed, if at all, as a decision-support tool in controlled settings, not as a replacement for human judgement[2].

VIII. CONCLUSION AND FUTURE WORK

This paper has presented SafeSense Guardian, a web-based, multimodal AI content-safety system designed for educational and small institutional environments[7]. The system combines a Flask backend, modular ML pipelines for text, images, audio, video and documents, and a relational database storing structured analysis results and user histories[4][5][6]. Implementation and testing in a lab environment show that the architecture is feasible on standard hardware and that core functional requirements—registration, authentication, multimodal upload, analysis, storage and history retrieval—are met.

Future work will focus on replacing heuristic modules with state-of-the-art models, including transformer-based text classifiers, stronger CNN or vision-transformer backbones for images and videos, and more robust speech and document processing pipelines[3]. Explainability techniques that highlight influential words, frames or image regions could further improve transparency and user trust[1][2]. Additional directions include multilingual and code-mixed content support, containerised and cloud-ready deployments, richer admin dashboards for analytics, integration with LMS and chat platforms, and explicit support for hybrid human-AI moderation workflows where AI systems flag content but human reviewers handle appeals and borderline cases.

By evolving along these lines, SafeSense Guardian can move from an academic prototype to a flexible research and education platform for responsible multimodal content moderation in institutional settings.

REFERENCES

- [1] Pawar, A., & Jose, J. – Hybrid Sequential Modeling for Social Media Behaviour Analysis.
- [2] Kumar, et al. – Explainable AI for Image-Based Analysis using LIME and SHAP.
- [3] Research on Content Moderation using Machine Learning (SVM, LSTM, Transformers).
- [4] Flask Documentation – <https://flask.palletsprojects.com>
- [5] Python Machine Learning Libraries Documentation (NumPy, Pandas, Scikit-learn).
- [6] SQLite / MySQL Documentation for Database Management.
- [7] SafeSense Guardian Project Report (MCA Project Documentation).

International Journal of Advanced Research in Education and Technology

ISSN: 2394-2975

Impact Factor: 9.971