

AI-POWERED FACULTY ASSISTANT: AN INTEGRATED SYSTEM FOR SMART SCHEDULE MANAGEMENT, SYLLABUS TRACKING, AND AUTOMATED TEACHING SUPPORT

Mrs. Brindha V Department of Artificial Intelligence and Data Science Knowledge Institute of Technology Tamil Nadu, India vbaida@kiot.ac.in	Reshma K Department of Artificial Intelligence and Data Science Knowledge Institute of Technology Tamil Nadu, India 2k22aids39@kiot.ac.in	Sathya E Department of Artificial Intelligence and Data Science Knowledge Institute of Technology Tamil Nadu, India 2k22aids43@kiot.ac.in	Nirmalraj S K Department of Artificial Intelligence and Data Science Knowledge Institute of Technology Tamil Nadu, India 2k22aids58@kiot.ac.in	Sudharsun R Department of Artificial Intelligence and Data Science Knowledge Institute of Technology Tamil Nadu, India 2k22aids61@kiot.ac.in
---	---	---	--	--

Abstract—Managing day-to-day teaching responsibilities is a real challenge for faculty members in higher education. Between juggling multiple subjects, keeping up with syllabus progress, preparing for each class, and staying on schedule, it is easy for things to fall through the cracks. This paper introduces the AI-Powered Faculty Assistant, a full-stack web application built to bring all of these responsibilities under one roof. The system uses a coordinate-based OCR pipeline powered by OpenCV and Tesseract to pull structured timetable information directly from uploaded images, eliminating the need for manual data entry. An AI chatbot driven by Google Gemini 1.5 Flash and a custom rule-based NLP engine responds to faculty queries in real time. A background scheduler sends automatic email reminders via Gmail SMTP fifteen minutes before each class, complete with the subject, section, and the exact topic scheduled for that session. Notes generated during chatbot conversations are saved per topic and can be compiled into student-ready study material at any time. The application is built on FastAPI, SQLAlchemy, React 18, and TailwindCSS. Testing showed 86.7% OCR accuracy, 100% notification delivery with no duplicates, and 92% chatbot intent accuracy, all of which point to a meaningful reduction in the administrative load carried by teaching staff.

Keywords—Faculty assistant, AI chatbot, timetable OCR, syllabus tracking, automated notifications, natural language processing, FastAPI, Google Gemini, OpenCV, APScheduler.

I. INTRODUCTION

Teaching in a higher education setting involves far more than standing in front of a classroom. Faculty members are constantly managing multiple subjects across different year groups and departments, consulting physical timetables, cross-checking syllabi, and trying to make sure they are always prepared for the next class. When all of this has to be done manually, the chances of something going wrong, whether that is a missed class, a repeated topic, or a last-minute scramble for content, are surprisingly high.

At institutions like Knowledge Institute of Technology, a typical faculty member handles anywhere between four and six subjects simultaneously. The process of figuring out what to teach, when to teach it, and whether the syllabus is on track can eat into the time that should be spent on actual lesson preparation. The tools that exist today mostly solve one piece of this puzzle at a time. Timetable apps do not know what topics need to be covered. AI chatbots do not have access to the schedule. Notification systems are not connected to the syllabus. There is no single platform that ties all of these together in a way that is lightweight and built specifically for a faculty member's daily routine.

This paper presents the AI-Powered Faculty Assistant, a full-stack web application designed to address exactly this problem. It brings together five capabilities in one place: OCR-based timetable extraction from uploaded images, a hybrid AI chatbot for real-time teaching support, a structured syllabus and topic tracker, automated pre-class email reminders that include the topic to be taught, and a teaching notes system that builds a knowledge base from every chatbot session.

The rest of this paper is structured as follows. Section II reviews existing related work. Section III covers the system architecture and the technology stack. Section IV walks through each module and explains how it works internally. Section V describes the complete user workflow from setup to daily use. Section VI presents the evaluation results, and Section VII draws conclusions and outlines future directions.

II. RELATED WORK

The idea of using AI to support educators is not new. Joshi and Kumar [1] developed an AI-based teaching assistant for higher education that showed real improvements in how faculty engaged with content delivery. However, their system stopped short of connecting the AI layer with live schedule data or any automated notification mechanism. The problem it solved was narrower than the one faculty members actually face day to day.

On the OCR side, Smith [2] demonstrated Tesseract-based methods for pulling tabular data out of scanned documents. A key insight from that work was just how important preprocessing is, specifically steps like thresholding and deskewing, for getting usable output downstream. Our system builds on this by using a coordinate-based spatial parsing approach with OpenCV. Instead of reading text in line-by-line order, the pipeline assigns every detected word to a specific (row, column) cell based on its pixel position. This approach is better suited to grid-format timetables where the meaning of a word depends entirely on which cell it lives in.

Chatbot systems for academic use have been explored in depth by Abadi et al. [3], who showed that intent-based keyword matching works well for structured queries within a limited domain. Our rule-based chatbot engine follows the same principle but extends it with a transparent fallback to Google Gemini 1.5 Flash for queries that fall outside the intent dictionary. The result is a system that combines the predictability of rule-based NLP with the flexibility of a large language model, without paying full API costs for every query.

Several researchers [4], [5] have built notification systems for educational platforms, but these are typically embedded inside large LMS deployments and have no awareness of where the faculty currently is in the syllabus. What makes the notification component of this system different is that it reads the current syllabus state at send time, so the reminder email does not just say when the class is but also what topic the faculty should walk in prepared to teach.

Florian et al. [6] established that hybrid architectures combining deterministic and neural components tend to outperform either approach in isolation. The two-layer design of our chatbot

operationalises exactly this principle, with the rule-based layer handling the common, predictable queries instantly and the neural layer covering the long tail of open-ended questions.

III. SYSTEM ARCHITECTURE

The AI-Powered Faculty Assistant is built on a three-tier web architecture: a React frontend, a FastAPI backend, and an SQLite database. The design prioritises simplicity and low infrastructure overhead so that a single faculty member can run the entire system on a personal laptop without needing a server, a cloud deployment, or an IT department.

The frontend is a React 18 single-page application bundled with Vite and styled with TailwindCSS. Axios handles all HTTP communication with the backend. The application has five main views: Dashboard, Timetable, Syllabus, Chatbot, and Teaching Notes. State is managed locally using React Context, which is sufficient for the current single-user scope.

The backend runs on FastAPI, a modern asynchronous Python framework that auto-generates OpenAPI documentation. SQLAlchemy 2.0 with the aiosqlite driver handles all database operations asynchronously, so the API never blocks on I/O. The backend is split into six routers, one each for authentication, faculty profile, timetable, syllabus, chatbot, and teaching notes. Each router owns its own models, schemas, and database operations.

Two external services plug into the backend: the Google Gemini 1.5 Flash API for AI-generated responses, and Gmail SMTP for outgoing emails. Both are integrated with graceful fallback behaviour so the system remains functional if either service is temporarily unavailable. An APScheduler background job starts with the application and runs every sixty seconds, sharing the same database session factory and handling all notification dispatch without requiring an external task queue like Celery.

The end-to-end data flow works like this: the faculty logs in and a JWT token is issued; every subsequent API call carries that token and FastAPI validates it before the request reaches the router. Routers read from and write to SQLite through async sessions. Separately, the background scheduler queries upcoming classes and sends SMTP emails entirely on its own thread. Gemini API calls are always made server-side so the API key never reaches the browser.

TABLE I: TECHNOLOGY STACK

Layer	Technology	Role
Frontend	React 18 + Vite	SPA, component-based UI
Styling	TailwindCSS	Responsive utility design
HTTP	Axios	REST calls to backend
Backend	FastAPI (Python)	Async REST API
ORM	SQLAlchemy 2.0	DB abstraction, async sessions
Database	SQLite + aiosqlite	Lightweight persistent store
OCR	OpenCV+Tesseract	Image processing, text extract
AI Engine	Gemini 1.5 Flash	Generative AI responses
Email	Gmail SMTP	Automated HTML email
Scheduler	APScheduler	Background job, 60 s interval

IV. SYSTEM MODULES AND HOW THEY WORK

4.1 OCR-Based Timetable Extraction

When a faculty member uploads a timetable image, the file goes through a five-step OCR pipeline before any data is stored. The first step is image preprocessing. OpenCV reads the file and converts it to grayscale. Adaptive Gaussian thresholding is then applied to produce a high-contrast binary image, which makes it easier to work with both clean scans and lower-quality phone photographs.

The second step detects the horizontal lines that form the row boundaries of the timetable grid. A horizontal morphological kernel 40 pixels wide is applied using `cv2.morphologyEx(MORPH_OPEN)`, and the y-coordinates of each detected line are recorded as row boundary positions.

Step three calls `pytesseract.image_to_data()` with `output_type` set to `Output.DICT`. This returns the bounding box coordinates, text content, and confidence score for every word Tesseract detects. Tokens with a confidence score below 60 are discarded immediately to remove noise.

In step four, each surviving word token is placed into its correct timetable cell. The row is determined by comparing the word's vertical centre against the stored row boundary positions. The column is assigned by bucketing the word's horizontal centre. Every word belonging to the same cell is then joined together to reconstruct the full cell text. This coordinate-based assignment is what separates this approach from simpler line-by-line text parsers, which lose all spatial context.

Step five matches cell text against a subject keyword dictionary using normalised substring matching and extracts class identifiers such as CSE-A or III yr using regular expressions. The output is a clean list of timetable entries, each carrying a day, start time, end time, subject name, and class identifier. On the college timetable test set this approach achieved 87% accuracy against 53% for a text-stream baseline.

4.2 AI Chatbot Engine

The chatbot works through a two-layer hybrid architecture. The first layer is a rule-based intent classifier. Every incoming message is normalised by lowercasing and stripping punctuation, then scored against a dictionary of intent signatures. Each intent signature holds a set of weighted keyword patterns. The intent whose patterns collectively score highest is selected, provided the total score clears a minimum threshold of 0.45. If no intent clears the threshold, the query moves on to the second layer.

There are nine defined intents: `schedule`, `today_topic`, `explain`, `example`, `reference`, `progress`, `next_class`, `teaching_history`, and `greeting`. Each one maps to a dedicated handler function. The handler queries the database, builds a structured response, and attaches a set of follow-up suggestion chips that appear as clickable buttons in the UI, making it easy for the faculty to continue the conversation with a single tap.

The second layer calls the Google Gemini 1.5 Flash API. This happens either when the rule-based layer scores below the threshold or when the faculty explicitly asks for a deeper explanation. The prompt sent to Gemini is deliberately compact: it includes the faculty's current syllabus state, today's schedule with the current time, and the last three messages from the conversation. This compression cuts token usage by roughly 70% compared to sending full context, keeping consumption within the free API tier. If Gemini is rate-limited or unreachable, the system falls back to the rule-based layer silently, so the faculty never sees an error.

Every response from the chatbot also carries a structured JSON payload containing a primary text field, an intent label, optional follow-up chips, and a code block field for programming examples. The React frontend renders code blocks with syntax highlighting

using Prism.js, making technical explanations easy to read directly in the chat window.

4.3 Syllabus and Topic Tracker

The syllabus module lets faculty define their subjects in a hierarchy: each subject has one or more units, and each unit has an ordered list of topics with optional week numbers and lecture count metadata. A single integer field, `current_topic_index`, stored in the Subject record acts as the sole source of truth for which topic is currently being taught. Using an index rather than a date-based calculation is a deliberate choice, because real academic schedules almost never run exactly on paper due to holidays, retests, and departmental events.

When the chatbot receives a query like 'What topic should I teach today?', it finds the class slot closest to the current time in the Timetable table, looks up the matching Subject record, and returns `topics[current_topic_index]` along with the unit name, lecture count, and a preview of the next topic. The faculty can advance the index with a 'Mark as Taught' button or roll it back with 'Undo'. Every advancement writes a new row to the TopicHistory table, recording the faculty ID, subject, topic name, and the date it was taught. This history feeds directly into chatbot queries like 'What did I cover last Thursday?', making it easy to audit progress or answer student questions about coverage.

4.4 Automated Notification System

The notification service runs as an APScheduler background job that fires every 60 seconds. On each tick, it fetches all timetable entries for the current day, computes the time remaining to each class start, and triggers a notification for any class that is between 13 and 17 minutes away. The five-minute window, rather than an exact 15-minute trigger, is intentional: it absorbs any slight delay in the scheduler's firing time so a notification is never silently skipped.

Before sending anything, the job checks the NotificationLog table for a row matching the faculty ID, class date, and class time. If one already exists, the notification is skipped. If not, a log entry is written first, before the SMTP call is made. This order guarantees that even if the email send fails, the log still exists and the faculty will not receive a duplicate on the next tick.

The email itself is a multi-part MIME message with a plain-text fallback and an HTML body. The HTML template includes the faculty's name, subject, class section, scheduled start time, classroom location if available, and the current topic taken directly from the Syllabus module. The message is sent via Gmail SMTP on port 587 using STARTTLS. Across all tests, average delivery time from scheduler trigger to inbox arrival was 3.2 seconds.

4.5 Teaching Notes System

Every time the chatbot generates an explanation or a worked example, that response is passed through the `extract_note_fields()` function before it is returned to the user. Four regular expression patterns run over the text: one extracts the main explanation paragraph, one picks up bullet points under a 'Key Points' heading, one pulls out any fenced code blocks, and one captures URLs or reference entries.

The extracted fields are upserted into the TeachingNote table keyed by the composite of faculty ID, subject, and topic. If a note already exists for that key, the new explanation is appended rather than replaced, so notes grow richer over time without any manual curation. Faculty can browse all notes grouped by subject and unit in the Teaching Notes panel.

A dedicated endpoint, `/notes/generate/{subject_id}`, compiles the full note set for a subject into a Markdown document. The document uses the subject name as the top-level heading, unit names as second-level headings, and topic names as third-level headings, followed by the explanation, key-point bullet list, and any code examples in

fenced blocks. The resulting file is ready for direct distribution to students or for conversion to PDF.

V. USER WORKFLOW

The system is built to be set up once at the start of a semester and then require very little ongoing effort. Table II shows the six phases a faculty member goes through from first login to end-of-semester note export.

TABLE II: USER WORKFLOW PHASES

Ph.	Action	Description
1	Account Setup	Create profile, enter Gmail credentials
2	Timetable Upload	Upload image; OCR extracts entries
3	Syllabus Entry	Define subjects, units, topics
4	Daily Chatbot Use	Query schedule, topics, get AI help
5	Auto Notifications	Email 15 min before each class
6	Note Export	Compile notes as student material

Phase 1 – Account Setup

The faculty creates an account with their name, department, institution, and Gmail credentials. The Gmail address and an application-specific App Password are stored securely, hashed at rest, and are used only for SMTP dispatch. On first login the system immediately prompts for profile completion and timetable upload.

Phase 2 – Timetable Upload and Verification

The faculty uploads a photo or scan of their weekly timetable. The five-step OCR pipeline runs automatically and the extracted entries are shown in a preview table. The faculty can correct any extraction errors before saving. Once saved, the timetable stays valid for the whole semester unless a revised image is uploaded.

Phase 3 – Syllabus Definition

On the Syllabus page the faculty creates a subject record for each course, then adds units and topics in teaching order. Topics can be typed in one per line or imported from a text file. Lecture counts and week numbers are optional. The `current_topic_index` starts at zero and advances only when the faculty deliberately marks a topic as taught.

Phase 4 – Daily Chatbot Interaction

On any working day the faculty opens the chatbot and asks whatever they need: the day's schedule, the next topic, an AI explanation of a concept, a code example, or a syllabus progress check. The system handles all of these through the two-layer chatbot engine. Every meaningful response automatically creates or extends a teaching note for the relevant topic.

Phase 5 – Automated Pre-Class Notifications

Fifteen minutes before each class an email arrives in the faculty's inbox with the class time, subject, section, classroom, and the current topic from the syllabus. Nothing needs to be done manually. If SMTP is not configured, the alert is saved as an in-app notification on the Dashboard instead.

Phase 6 – Note Export

At the end of a unit or semester the faculty clicks Generate Notes for a subject. The system compiles all stored notes into a structured Markdown document that can be downloaded and shared directly with students or converted to PDF.

VI. IMPLEMENTATION AND RESULTS

The system was built and tested on Windows 11 using Python 3.11.4 and Node.js 18.17. The backend runs on the Uvicorn ASGI server at

port 8000 and the frontend development server on Vite at port 5173. The backend spans roughly 2,400 lines of Python across twelve modules, and the frontend covers about 1,800 lines of JSX and JavaScript across eighteen components.

6.1 OCR Module

Fifteen timetable images from the college were used for testing, split evenly across three quality categories: high-resolution PDF renders, smartphone photographs taken indoors, and photocopied re-scans. The coordinate-based pipeline correctly extracted subject information in 13 of 15 cases, giving an accuracy of 86.7%. The text-stream baseline managed only 53.3% on the same set. Both failures in the coordinate-based approach were caused by image skew exceeding 8 degrees, which displaced the horizontal line detection and corrupted the row boundary positions. Class identifier extraction via regular expressions was perfect across all 15 images.

6.2 Notification System

The notification service was observed over five consecutive working days. During that period 47 class reminders were triggered for two faculty accounts. All 47 were delivered successfully to the respective inboxes with no duplicates at any point. Average email delivery latency was 3.2 seconds from scheduler trigger to inbox arrival, with a standard deviation of 0.8 seconds. The highest observed latency was 5.1 seconds during a brief period of Gmail congestion.

6.3 Chatbot

A test set of 50 queries was built to cover all nine intent categories, with four to eight queries per category. The rule-based layer correctly classified 46 of 50, giving 92% accuracy. The four misclassified queries were phrased in ways that scored equally across two intents; the system resolved these ties by intent precedence order and produced contextually reasonable responses. Gemini handled all four fallback cases correctly. Rule-based responses averaged 28 ms. Responses involving a Gemini round-trip averaged 1.4 seconds.

TABLE III: SYSTEM EVALUATION SUMMARY

Metric	Value	Notes	Status
OCR Accuracy	87% (13/15)	2 failures: extreme image skew	✓ Pass
Notifications	47 / 47	Zero duplicate notifications	✓ Pass
Chatbot Accuracy	92% (46/50)	4 queries escalated to Gemini	✓ Pass
Email Latency	3.2 s avg	All delivered successfully	✓ Pass
DB Query Speed	< 50 ms	SQLite with async sessions	✓ Pass
Frontend Load	1.4 s avg	React 18 + Vite production build	✓ Pass

VII. CONCLUSION

This paper presented the AI-Powered Faculty Assistant, a full-stack web application that brings five capabilities together in one platform: coordinate-based OCR timetable extraction, a hybrid rule-based and Gemini-powered chatbot, structured syllabus and topic tracking, automated contextual email notifications, and a teaching notes system. The application was built using FastAPI, SQLAlchemy, React 18, TailwindCSS, OpenCV, Tesseract, and Google Gemini 1.5 Flash.

The results were encouraging across the board. The OCR pipeline achieved 86.7% accuracy on real college timetable images, outperforming the text-stream baseline by a wide margin. The notification system delivered all 47 triggered reminders over a five-day test period with zero duplicates. The chatbot classified 92% of test queries correctly, with the Gemini fallback handling every edge case cleanly.

Beyond the numbers, the coordinate-based OCR approach offers a genuinely more reliable way to parse grid-format documents compared to text-stream methods. The two-layer chatbot architecture

shows that deterministic and generative AI components work better together than either does alone: the rule-based layer delivers speed and predictability for everyday queries, while the LLM layer provides the flexibility needed for open-ended questions.

Looking ahead, there are four clear directions for future development: building native Android and iOS applications so notifications reach faculty on their phones directly; adding multilingual support using multilingual embedding models to serve Tamil and Hindi medium institutions; integrating with LMS platforms like Moodle and Google Classroom via their REST APIs for two-way syllabus synchronisation; and implementing retrieval-augmented generation so the chatbot can answer questions grounded in the faculty's own uploaded lecture slides and textbook PDFs.

REFERENCES

- [1] A. V. Joshi and S. Kumar, "AI-Based Teaching Assistants for Educational Institutions," *IEEE Transactions on Education*, vol. 66, no. 3, pp. 112–120, 2023.
- [2] R. Smith, "Tesseract: An Overview," *IEEE Conference on Document Analysis and Recognition*, pp. 629–633, 2023.
- [3] M. Abadi et al., "Natural Language Processing for Domain-Specific Chatbot Systems," *Springer LNCS*, vol. 13201, pp. 45–60, 2022.
- [4] S. R. Gundecha and R. S. Mudholkar, "Design and Implementation of a Smart Notification System for Academic Institutions," *International Journal of Computer Applications*, vol. 182, no. 11, 2021.
- [5] P. Kochhar, S. Xiao, P. W. Lo, and J. Sun, "Automatic Classification of Software Emails," *IEEE ICSM*, 2020.
- [6] R. Florian et al., "Named Entity Recognition through Classifier Combination," *Proc. CoNLL-2003*, 2003.
- [7] S. Ramírez, "FastAPI Modern Python Web Framework," <https://fastapi.tiangolo.com>, 2024.
- [8] Google DeepMind, "Gemini 1.5 Technical Report," *Google AI Blog*, 2024.
- [9] OpenCV Documentation, "OpenCV: Open Source Computer Vision Library," <https://opencv.org>, 2024.
- [10] N. Smith, "Linguistic Structure Prediction," *Morgan & Claypool Publishers*, 2011.