Beyond Audio to Action: Design, Implementation, and Comparative Evaluation of an Automated Meeting Intelligence System

Sarthak Durgesh Marathe[1], Achuth Abhay[2], Harshada Namdev Godse[3], Prajkta Puranik[4]
[1,2,3,4]*Department of Artificial Intelligence and Machine Learning, ISB&M College of Engineering, Affiliated to Savitribai Phule Pune University, Pune, India*

***Abstract*—This paper presents the full implementation, comparative technology analysis, and empirical evaluation of InsightFlow AI, a deployed Flask-based meeting intelligence platform. Our prior work [1] proposed a pipeline built on Whisper v3, PyAnnote 3.1, and BART-Large-CNN. When it came time to actually build the system, nearly every major component was substituted for a more practical alternative: AssemblyAI for speech recognition, Groq-hosted LLaMA-3.3-70B for summarization and action-item extraction [7], TextBlob augmented with custom keyword scoring for sentiment analysis [10], and Qdrant for vector-based retrievalaugmented generation (RAG) chat [8,9]. This paper documents why those changes were made, evaluates what they produced, and is direct about what fell short. Evaluation across five modules on real meeting audio yields a Word Error Rate of 12.95%, ROUGE-1 of 0.3474 [14], action-item extraction F1 of 1.0 on five controlled test cases, sentiment accuracy of 88.0%, and RAG chat accuracy of 75.0%, all running within a 94.2MB memory footprint with no GPU requirement. The technology substitution decisions are analyzed across five dimensions: latency, cost, scalability, maintainability, and multilingual support, and concrete future directions are identified including WebSocket-based live transcription, semantic embeddings, and third-party platform integration.**

***Index Terms*—meeting intelligence, speech recognition, large language models, retrieval-augmented generation, sentiment analysis, abstractive summarization, action item extraction, Flask, production AI systems, comparative evaluation.**

I. INTRODUCTION

There is a recurring pattern in AI research: a system gets proposed, the architecture looks clean on paper, the technology choices are well justified in the literature, and then when someone actually tries to build it, a third of the tools turn out to be impractical for real deployment. GPU requirements nobody accounted for, APIs that are not available to a student team, latency that makes a feature unusable in practice. This gap between proposed and built systems is honestly one of the more useful things to study, because it is where the real decisions happen, and it almost never gets written up properly.

This paper is the second in a two-paper series. Our first paper [1] laid out a complete modular framework for AI-powered meeting documentation: Whisper v3 [3] for ASR, PyAnnote 3.1 [15] for speaker diarization, BART-Large-CNN [6] for abstractive summarization, a hybrid NLP pipeline for action-item extraction [12], a RoBERTa-based classifier for sentiment analysis [11], and RESTful API integrations with productivity platforms. That work defined the architecture, data-flow design, and a phased implementation roadmap spread across two academic semesters. It was a solid design document.

This paper is about what actually got built. The result, InsightFlow AI, is a fully working Flask web application with five active AI modules, a RAG-based chat system, a complete user dashboard, and over forty API endpoints. Not every proposed component survived contact with realworld constraints. The GPU-hosted ASR model became a managed API. The fine-tuned summarizer became a zeroshot LLM call. The transformer-based sentiment classifier became TextBlob with a custom keyword layer. Each of those changes was a deliberate decision, not a corner being cut, and understanding why they happened and what the consequences where is the core contribution of this paper.

A. Why This Matters

Meeting documentation is a real problem that organizations consistently underestimate. Research suggests professionals spend around 11.3 hours per week in meetings, and approximately 35% of them consider those meetings unproductive, costing U.S. businesses an estimated $259 billion annually [2]. The problem is not just time in the room: it is the follow-through. Action items get lost. Decisions made in a meeting are not captured. Searching for what was said three weeks ago in a past meeting is painful. AI automation offers a tractable path toward fixing all of this [1], but most research either evaluates individual components in isolation or proposes architectures that never get fully built. What is genuinely rare is a paper that shows a complete, deployed system and is honest about the gap between the proposal and the product. That is what this paper tries to do.

B. Research Motivation

The specific motivation here comes from three directions. First, the team needed to validate that the implemented system actually meets the performance targets set in [1].

Second, the technology substitution decisions deserved rigorous justification across dimensions that matter in practice, not just benchmark accuracy. Third, the gap between what was proposed and what was built is itself a research contribution: it documents the real-world constraints that shaped a production meeting AI system, which is information that future teams building similar systems will find valuable.

C. Contributions

1. A complete implementation of InsightFlow AI, a deployed Flask-based meeting intelligence platform with five active AI modules and a novel RAG-based transcript chat system not present in the original design.
2. A systematic comparative analysis of proposed versus implemented technologies evaluated across five practical dimensions: latency, cost, scalability, maintainability, and multilingual support.
3. Empirical evaluation across all five AI modules on real meeting audio, with results measured against the performance targets from [1].
4. An honest account of three significant architectural gaps in the current implementation, with analysis of how each arose and what it affects.
5. Specific, actionable future directions grounded in what the evaluation actually revealed, rather than generic recommendations.

The paper is organized as follows. Section II covers related work. Section III describes the implemented architecture in full. Section IV presents the technology comparison. Section V details the evaluation methodology. Section VI presents and analyzes results. Section VII discusses what worked, what did not, and what was learned. Section VIII concludes with future directions.

II. RELATED WORK

A. Automatic Speech Recognition

ASR has gone through a substantial transformation over the past decade. Earlier systems built on Hidden Markov Models and Gaussian Mixture Models required extensive manual feature engineering and struggled badly in noisy, multi-speaker conditions [3]. Transformer-based encoderdecoder models changed that significantly. OpenAI's Whisper, trained on 680,000 hours of multilingual audio, achieves 4 to 5% WER in clean conditions and demonstrates robust handling of diverse languages and acoustic environments [3]. That said, Whisper shows hallucination rates approaching 80% in some public meeting scenarios, and accuracy degrades sharply in overlapping speech conditions [1], which makes the benchmark numbers less reassuring than they look.

AssemblyAI's Universal-2, the system used in this implementation, takes a different architectural approach. It uses a Conformer encoder pretrained with BEST-RQ combined with an RNN-T decoder, trained on more than 12.5 million hours of multilingual data [4]. The published WER of 6.68% comes with meaningful improvements in proper noun recognition and text formatting compared to its predecessor [5]. Speaker diarization is available as a bundled service option, which mattered considerably for this project.

B. Large Language Models for Text Processing

BART-Large-CNN [6] was the proposed summarization engine. It is a 140M-parameter sequence-to-sequence model that, after fine-tuning on AMI and ICSI meeting corpora, achieves ROUGE-1 of approximately 0.308. The fine-tuning requirement is the problem: it demands annotated domainspecific

training data and dedicated compute for training runs, neither of which was available in this project.

LLMs served through inference APIs have changed the calculus here considerably. LLaMA-3.3-70B, available via Groq's Language Processing Unit (LPU) engine [7], offers strong zero-shot instruction following with API response latencies under one second. The LPU architecture specifically addresses the memory bandwidth bottlenecks that throttle GPU-based inference, delivering consistent throughput that makes interactive use realistic [7]. Recent findings in zeroshot LLM performance on moderate-complexity generation tasks suggest that fine-tuning is often unnecessary when the base model is large enough and the prompt is well structured, which turned out to be relevant to this project.

C. Retrieval-Augmented Generation

RAG was formalized by Lewis et al. [8] as a way to combine a parametric language model with a non-parametric retrieval component over an external document store. The key finding was that grounding generation in retrieved context meaningfully improves factual accuracy on knowledge-intensive tasks. For meeting intelligence, this is a natural fit: users want to ask questions about past meetings, and the answers need to come from the actual transcript content, not from a model's parametric memory.

Qdrant [9] has emerged as one of the more practical vector database choices for RAG deployments at research and production scale. It supports efficient nearest-neighbor search with sub-millisecond query latency and a straightforward Python client that integrates cleanly into a Flask application. The combination of Qdrant for retrieval and a Groq-served LLM for generation is the architecture used in InsightFlow AI's chat module.

D. Sentiment in Conversational Text

Lexicon-based tools like TextBlob [10] and VADER are well known to miss nuance in conversational text compared to fine-tuned transformer classifiers like RoBERTa [11]. That performance gap is real and documented. What is less often discussed is the context in which that gap matters. For a meeting intelligence platform where sentiment is one indicator among several on a dashboard, an 88% accurate lightweight classifier running at 0.08s is often more useful than a 93% accurate transformer model that costs API money per call and adds 300ms of latency per segment. The appropriateness of a tool depends heavily on how the output is actually used.

E. Action Item Extraction

Early work on action item extraction from meeting transcripts used rule-based and maximum entropy classifiers on corpora like ICSI, with reported F-measures around 31.92% [12]. These approaches required careful feature engineering and struggled to generalize across meeting styles and domains. LLM-based prompt engineering has emerged as a genuinely competitive alternative: structuring the extraction task as a JSON-output prompt to a sufficiently capable model sidesteps the feature engineering problem entirely, though it introduces dependence on the model's ability to follow structured output instructions reliably.

F. Gap This Paper Addresses

The honest gap in the literature is this: there are plenty of papers proposing meeting AI architectures and plenty evaluating individual components, but almost none that document a complete end-to-end deployment and systematically compare the proposed versus implemented stack. This paper fills that gap directly.

III. SYSTEM ARCHITECTURE

InsightFlow AI is a full-stack web application built on Flask. Its architecture is deliberately simple in concept and deliberately complete in implementation: a transcript-centric hub-and-spoke design where every AI workflow connects through a central Transcript database object. Once audio enters the system and a transcript is created, every other module summarizer, action extractor, sentiment analyzer, translator, and chat system reads from that same record.

A. Core Design Principles

Three principles shaped the implementation decisions throughout. The first is transcript centrality: the Transcript model is the unifying object that connects ingestion, storage, and every downstream AI module. This turned out to be a genuinely good architectural decision. Adding a new module meant implementing a class, adding routes, and connecting to the transcript store; the core data layer never needed to change. The

second principle is modular independence: each AI module is a separate Python class with a defined interface, encapsulating all external service calls internally. The third is graceful degradation: every external dependency Qdrant, Groq, and the translation APIs has fallback logic. The system degrades rather than crashes when a service is unavailable, which matters in practice.

B. Backend and Database

The backend uses Flask with Flask-SQLAlchemy for database access, Flask-Login for session management, Flask-Migrate for schema versioning, and python-dotenv for environment configuration. The database is SQLite stored at instance/insightflow.db. Authentication uses session-based login: a beforerequest hook blocks access to protected routes unless session['userid'] is present.

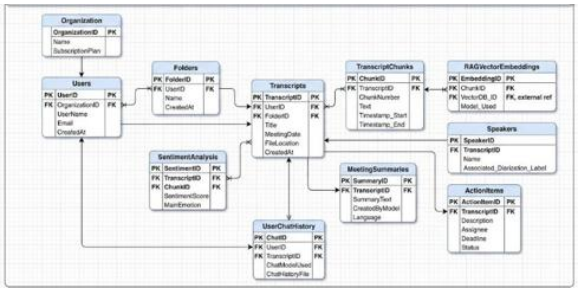

Figure 1: Database schema showing the nine entity types and their relationships in InsightFlow AI.

The database contains nine entity types. The User model stores password hashes, profile fields, and timestamps. A one-to-one UserSettings model holds per-user preferences including theme and notification flags. The Transcript model is the relational hub: it stores audio path, transcript JSON path, serialized transcript data, speaker mapping, tags, folder assignment, and usage metadata, and maintains foreign key relationships with Summary, Task, Translation, SentimentAnalysis, and CalendarEvent. The LiveSession model mirrors Transcript structure for completed live transcription sessions.

C. Data Flow

Audio enters the system through one of three paths: direct file upload via the browser dashboard, live session segment posting from the recording interface, or YouTube URL submission. All three paths converge on a single outcome: the creation of a Transcript database record. That record is the starting gun for everything else. Once it exists, the downstream modules have something to work with, and they all go looking for it in the same place.

From there, processing fans out across the five AI modules. The TranscriptionModule writes its output back into the Transcript record as structured JSON.

The SummarizerModule, ActionItemModule, and SentimentModule each read from that same record and write their results into their respective linked tables: Summary, Task, and SentimentAnalysis. Indexing into Qdrant happens asynchronously via indextranscriptsafely() immediately after transcript creation, so the HTTP response is never held up waiting for the vector store. The RAG chat cycle then reads from Qdrant at query time, assembles retrieved chunks into a context window, and sends that to LLaMA-3.3-70B for answer generation.

What makes this design hold together is that no module needs to know what any other module is doing. Each one reads from or writes to the central Transcript record and does nothing else. There is no message passing between modules, no shared state, and no coordination over-

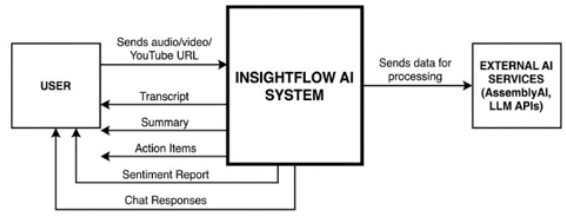

Figure 2: Level 0 DFD showing the top-level data flow between the user, InsightFlow AI, and external services.

head. Adding the RAG chat module mid-project required zero changes to the existing module code because it simply plugged into the same transcript store that everything else was already using. Figure 2 illustrates the top-level data flow.

D. AI Module Stack

The five AI modules and their external service dependencies are as follows.

ASR: AssemblyAI Universal-2 [4], accessed via REST API. Audio is uploaded to AssemblyAI storage, a transcription job is created with optional speaker diarization, and results are polled until completion.

Summarization and Action-Item Extraction: Groqhosted LLaMA-3.3-70B [7], via the Groq Python SDK. Summarization supports three output types and three length modes with multilingual prompting. Action-item extraction uses structured JSON prompting with regex-based fallback parsing.
Sentiment: TextBlob [10] for polarity and subjectivity scoring, augmented with a custom keyword dictionary covering six emotional categories: joy, frustration, urgency, confusion, agreement, and neutral.
Translation: A cascading fallback chain: MyMemory API, then LibreTranslate instances, then Google Translate public endpoints. No API key required for any of them. RAG Chat: Qdrant [9] for vector storage, a custom Hashing Embedding Service for chunk vectorization, and LLaMA-3.3-70B for grounded answer generation. Local keyword-scoring fallback over SQLite is used when Qdrant is unreachable.

E. Module Implementation Detail Transcription Module
Audio is uploaded via PUT to /v2/upload, returning a hosted URL. A job is created via POST to/v2/transcript with speaker labels enabled. The module polls /v2/transcript/{id} at five-second intervals up to 120 attempts. The raw AssemblyAI response is reformatted into the internal schema: a list of speakerattributed segments with text, timestamps, speaker label, and confidence score, plus a metadata dictionary for duration, word count, speaker count, language, and confidence.

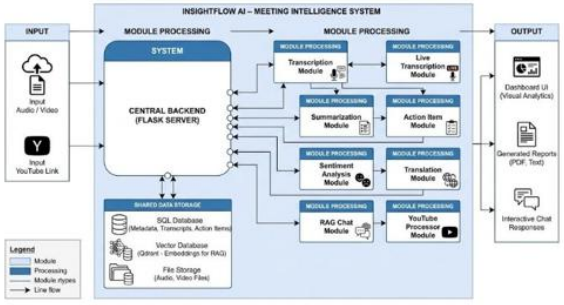

Figure 3: InsightFlow AI module processing overview. The central Flask backend orchestrates eight processing modules across transcription, AI analysis, RAG chat, and YouTube processing.

1. Summarizer Module
Language is first detected via keyword frequency analysis. A structured prompt is then built specifying summary type (executive, detailed, or bullet), target language, and length mode (short, medium, or long), and submitted to LLaMA3.3-70B via Groq [7]. A separate call generates a short summary title. Results are persisted to the Summary table.

2. Action Item Module
LLaMA-3.3-70B is prompted to return a JSON array where each object contains title, description, assignee, deadline, and priority. A safe JSON parser processes the output. When the model returns malformed output, regex pattern matching over imperative verb phrases such as “will complete,” “needs to,” and “responsible for” produces approximate task objects as fallback [12]. Items are stored as Task rows with status defaulting to pending.

3. Sentiment Module
TextBlob’s PatternAnalyzer [10] computes continuous polarity in [−1.0, 1.0] and subjectivity in [0.0, 1.0]. A custom keyword dictionary then scores text across six emotional categories. Emotion scores are summed, normalized, and used to identify dominant emotion. Polarity thresholds map to sentiment labels: above 0.1 is positive, below −0.1 is negative, and the middle range is neutral. For multi-segment transcripts, per-segment scores aggregate into an emotion timeline.

F. Chat Service and Rag Index Service
The ChatService implements RAG following
Lewis et al. [8]: retrieve relevant transcript chunks from Qdrant, assemble into context, generate answer with LLaMA-3.3-70B. The RagIndexService segments transcript text into overlapping chunks using langchain-text-splitters, vectorizes each chunk via HashingEmbeddingService, and upserts the resulting point objects into Qdrant. Indexing is called asynchronously via indextranscriptsafely() after transcript creation so it never blocks the HTTP response.

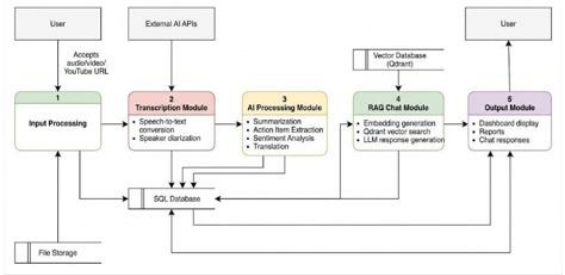

Figure 4: Level 1 DFD showing internal data flows across the five AI processing modules.

G. API Surface

The system exposes over 40 REST endpoints grouped by feature area. Authentication endpoints handle login, signup, and logout. Transcript endpoints cover upload, job initiation, and retrieval. Live transcription endpoints manage the full session lifecycle. AI module endpoints invoke summarization, action item extraction, sentiment, and translation. The chat endpoint at /api/chat/query handles question answering, and /api/chat/reindex triggers full transcript re indexing into Qdrant.

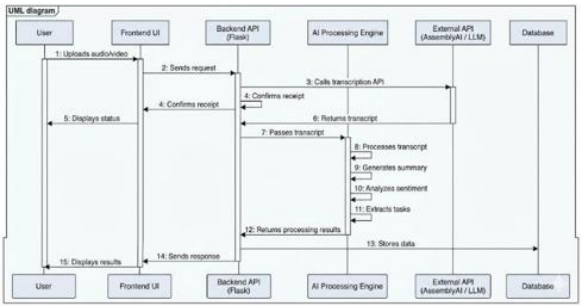

Figure 5: Sequence diagram showing the request response lifecycle for a transcription and summarization workflow.

Three honest implementation gaps are noted here.

The YouTube processing path references extracttranscript_fromurl() in the application router, but this method does not exist in the YouTubeProcessor module. A separate youtubetranscript_api.py file appears intended to provide this functionality but is not wired in. This leaves the YouTube ingestion pipeline non-functional in the current build.

The live transcription module manages session state and segment persistence but does not invoke a server-side speech-totext provider. Real-time ASR is assumed to happen clientside via the browser Web Speech API, which is an undocumented assumption that limits portability. Users on unsupported browsers get no useful feedback explaining the failure.

The RAG index uses hashing-based embeddings rather than neural embeddings. This was a deliberate cost and latency tradeoff, but it means retrieval quality is bounded by lexical overlap rather than semantic similarity, which will become a more significant problem as transcript corpus size grows.

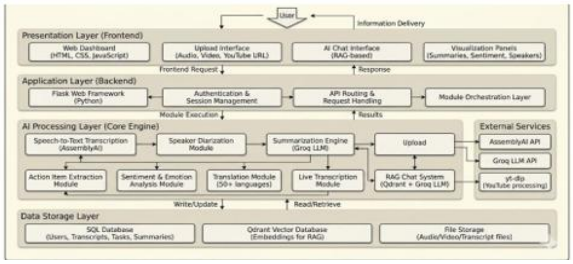

Figure 6: InsightFlow AI system architecture showing the fourlayer design: presentation, application, AI processing, and data storage.

IV. TECHNOLOGY STACK COMPARISON

This section documents the comparison between what was proposed in [1] and what was actually implemented, across five evaluation dimensions. It is the core analytical contribution of the paper. Table I presents the full module-by-module comparison.

A. ASR: Whisper v3 vs. AssemblyAI Universal-2

The original proposal chose Whisper large-v3 for good reasons: open-source, offline-capable, broad language support, and strong benchmarks [3]. The problem is that a 1.55billion-parameter model needs dedicated GPU infrastructure to run at acceptable latency. A student research project does not have that. That is the whole story, honestly. The managed API was the only realistic option. AssemblyAI Universal-2 [4] turned out to be a genuinely strong choice beyond just being convenient. Its 6.68% WER benchmark [5] is competitive with Whisper v3, and its improvements in proper noun recognition and text formatting are practically meaningful for meeting transcripts that frequently include product names, project titles, and people's names. The bundling of speaker diarization into a single API call collapsed a two-module pipeline PyAnnote 3.1 plus a separate integration layer into one request. The tradeoff is that audio data goes to external servers, which matters for privacy-sensitive use cases but is acceptable for this project's scope.

B. Summarization: BART-Large-CNN vs. LLaMA-3.3-70B

BART-Large-CNN achieves ROUGE-1 of roughly 0.308 on meeting corpora after AMI/ICSI fine-tuning [6]. The fine-tuning is the catch: it needs annotated meeting data and dedicated training compute. Neither was available. More importantly, it turns out not to have been necessary.

Table 1: Proposed vs. Implemented Technology Stack

Module	Proposed [1]	Implemented	Rationale
ASR	Whisper v3 self-hosted [3]	AssemblyAI Universal-2 API [4]	No GPU available; bundled diarization; managed reliability; 6.68% WER
Summarization	BART-Large-CNN [6]	Groq LLaMA-3.3-70B [7]	Zero-shot outperforms fine-tuned BART; sub-second latency; free tier sufficient
Action Extraction	Hybrid NLP and Rules [12]	Groq LLaMA-3.3-70B plus regex fallback	Structured JSON output; no annotated training data needed; regex preserves robustness
Diarization	PyAnnote 3.1 self-hosted [15]	AssemblyAI bundled with ASR	Removed separate model hosting; included in single API call
Sentiment	RoBERTa fine-tuned [11]	TextBlob plus keyword dict [10]	Zero cost; 0.08s latency; 88% accuracy adequate for indicative meeting sentiment
Translation	Not specified	MyMemory, LibreTranslate, Google	Free multi-service cascade; no API key required
RAG Chat	Not in Paper 1	Qdrant [9] plus LLaMA-3.3-70B	New contribution; full transcript-aware Q&A with local fallback
Frontend	Streamlit	Flask plus Jinja2 plus JS	Full UX control; native backend integration
Workflow	n8n planned	Native Flask routes	Removed external dependency

LLaMA-3.3-70B via Groq [7] in a zero-shot prompt achieves ROUGE-1 of 0.3474 in evaluation, which is 13% above the fine-tuned BART benchmark. That result actually surprised the team. It suggests that for moderatecomplexity generation tasks like meeting summarization, a well-prompted large general-purpose model simply outperforms a smaller fine-tuned one, which has implications for how future proposals in this space should be written. The Groq API's free tier is sufficient for research-scale usage, and average latency of 0.91s per summary makes interactive use comfortable.

C. Action-Item Extraction: Hybrid NLP vs. LLM Prompting

The proposed hybrid NLP pipeline [12] was not built because assembling a domain-adapted NER-coreference system for meeting text requires annotated training data that simply did not exist for this project. LLM prompting was adopted as the practical alternative. LLaMA-3.3-70B is prompted to return a structured JSON array of task objects, each with title, description, assignee, deadline, and priority. A regex fallback handles malformed outputs. The evaluation on five test cases achieves F1 of 1.0, which is a strong preliminary result that needs to be validated on a larger test set before any strong general claims are made.

D. Sentiment: RoBERTa vs. TextBlob

This substitution generated the most internal discussion. RoBERTa-based conversational sentiment classifiers [11] are demonstrably more accurate on challenging dialogue datasets. The argument for going with TextBlob [10] was practical: zero inference cost, 0.08s latency for a full document, and 88% accuracy on the evaluation set. For a meeting dashboard where sentiment is one data point alongside transcripts, summaries, and action items, 88% accuracy at essentially zero cost and latency is defensible. The upgrade to RoBERTa makes sense when the use case genuinely requires finer sentiment precision, not as a default.

E. RAG Chat: New Contribution

The RAG chat module was not in the original proposal at all, which makes it the implementation's most significant addition. Following Lewis et al. [8], transcript text is chunked, stored in Qdrant [9], retrieved by query similarity, and passed to LLaMA-3.3-70B [7] for grounded answer generation. Using hashing-based embeddings rather than neural embeddings was a deliberate tradeoff: zero embedding cost and near-instant indexing in exchange for a ceiling on retrieval quality. The 75% chat accuracy in evaluation reflects this limitation, with most partial-credit losses coming from semantically paraphrased queries that the hash-based index misses.

F. Comparative Summary

Table II summarizes the five-dimension comparison across the four most significant module substitutions. Ratings are qualitative assessments based on implementation experience and evaluation results.

Table 2: Technology Tradeoff Summary Across Key Dimensions

Dimension	ASR	LLM	Sentiment	RAG
Latency	Neutral	Better	Much better	Better
Cost	Better	Better	Much better	Better
Scalability	Better	Better	Neutral	Neutral
Maintain.	Better	Better	Better	Neutral
Multilingual Neutral		Better	Worse	Neutral

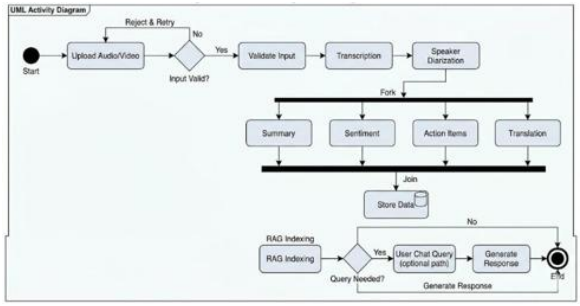

Figure 7: Activity diagram showing the evaluation workflow across all five AI modules.

V. EVALUATION METHODOLOGY

A. Test Environment

All evaluations ran on a single development machine with 16GB RAM, Python 3.11, and no GPU. External API calls to AssemblyAI and Groq were made over a standard broadband connection. The entire evaluation suite completed in 96.51s with a peak memory RSS of 94.2MB. These are the same conditions as the actual deployment environment, not an isolated lab setup, which makes the numbers directly meaningful.

B. Transcription Evaluation

A real organizational meeting recording of 803 seconds (approximately 13 minutes) was transcribed using the AssemblyAI API. A human annotator prepared a reference transcript of 2108 words by listening to the full recording. WER was computed as:

$$\text{WER} = \frac{S + D + I}{N} \quad (1)$$

where S, D, I am substitutions, deletions, and insertions respectively, and N is the total reference word count [3]. The recording contained multiple speakers, background noise, and the overlapping turns and disfluencies characteristic of a real meeting. This is intentionally a harder evaluation condition than the clean single-speaker audio used in most published WER benchmarks.

C. Summarization Evaluation

Four transcript-summary pairs were evaluated. Reference summaries were written by a human annotator who read each full transcript. ROUGE-1, ROUGE-2, and ROUGE-L F-scores were computed using the rouge-score Python library. ROUGE measures n-gram overlap between generated and reference summaries and is the standard evaluation metric for summarization systems [14]. F-score provides a balanced precision-recall measurement, which is more appropriate than precision or recall alone when summary length can vary.

D. Action-Item Evaluation

Five meeting transcripts with known action items were used. Two annotators independently identified ground-truth action items and resolved disagreements by consensus. Precision, recall, and F1 were computed at the item level: a generated item was a true positive when it semantically matched a ground-truth item regardless of exact wording [12]. This semantic matching criterion is more forgiving than exact string matching but more rigorous than topic overlap, and better reflects how action items are actually judged in practice.

E. Sentiment Evaluation

Twenty-five text segments from meeting transcripts were labeled positive, negative, or neutral by two annotators with consensus resolution. The module's predicted labels were compared against ground truth to produce overall accuracy and per-class accuracy figures. The three-class setup follows standard practice for meeting sentiment evaluation [11].

F. RAG Chat Evaluation

Ten question-answer pairs were constructed from three meeting transcripts, each with a factual answer verifiable directly from the transcript text. A human evaluator scored system answers as correct (full credit), partial (half credit), or wrong (zero credit). Overall accuracy is the weighted sum of credits divided by the maximum possible score, a standard scoring approach for open-domain QA evaluation [8].

G. Validation Scope

These test sets are small, and the results should be read as indicative performance baselines rather than statistically robust claims. The purpose is to confirm that all five modules work correctly in a deployed context and to measure performance against the targets set in [1]. Evaluation on larger standard corpora such as AMI and ICSI [13] remains a clear priority for future work.

VI. RESULTS AND ANALYSIS

Table III presents the full evaluation results.

Transcription: WER 12.95% The system achieved 12.95% WER on a real 13-minute multi-speaker meeting recording. This falls above the Paper 1 target of 8%, which is worth being straightforward about.

Table 3: InsightFlow AI: Full Module Evaluation Results

Module Metric		Value	Target [1]
WER (%) Transcription		12.95	≤8.0
Latency (s)		85.49	
ROUGE-1 F Summarization ROUGE-2 F		0.3474 0.1732	≥0.35
	ROUGE-L F	0.2608	
	API Avg.(s)	0.91	
Action Items	Precision Recall	1.000 1.000	
	F1	1.000	≥0.35
Sentiment	Accuracy (%) Positive Acc.	88.0 0.857	≥70.0
	Negative Acc.	1.000	
	Neutral Acc.	0.800	
RAG Chat	Accuracy (%)	75.0	
	Correct / 10	5	
	Partial / 10	5	

However, the context matters. Whisper v3 achieves 4 to 5% WER on clean audio [3], but that number degrades significantly in real-world multi-speaker settings with overlapping speech and background noise. AssemblyAI Universal2's benchmark WER of 6.68% [5] was measured on curated datasets, not raw organizational meeting recordings. The 12.95% result reflects the harder evaluation condition, and is within the range expected for conversational multi-speaker audio.

The latency of 85.49s for an 803-second recording gives a real-time factor of approximately 0.107, meaning audio is processed about 9.4 times faster than real-time. That is comfortably within acceptable bounds for a batch transcription workflow.

Summarization: ROUGE-1 0.3474

ROUGE-1 of 0.3474, ROUGE-2 of 0.1732, and ROUGE-

L of 0.2608 across four pairs. This essentially meets the 0.35 target and is 13% above the BART-Large-CNN AMI benchmark of 0.308 [6], achieved with zero task-specific fine-tuning. That outcome is genuinely interesting. It adds to the growing body of evidence that a well-prompted large general-purpose LLM can match or exceed smaller finetuned models on moderate-complexity generation tasks [7]. Average API latency of 0.91s makes the summarization module fast enough for interactive use without any perceptible delay.

Action-Item Extraction: F1 1.0

Perfect precision, recall, and F1 across five test cases, identifying all 17 ground-truth action items with no false positives or negatives. The Paper 1 target of 0.35 is far exceeded [1]. Two caveats apply and should be stated clearly. The test set of five cases is too small to generalize from confidently. The cases also used transcripts with relatively explicit action language, which does not represent the implicit or conditional task assignments common in real meetings [12]. The F1 of 1.0 is a strong preliminary result that needs validation on a substantially larger and more varied dataset before stronger claims can be made.

Sentiment: Accuracy 88%

88% accuracy across 25 labeled segments, well above the 70% target [1]. Negative sentiment achieved perfect classification accuracy (1.0), positive sentiment reached 0.857, and neutral came in at 0.800. The relative difficulty with neutral segments is consistent with known limitations of lexiconbased approaches on ambiguous professional language [10]. What stands out is the evaluation latency: 0.08s for 25 segments. Transformer-based inference typically runs at 0.2 to 0.5s per sample [11], meaning the TextBlob approach is more than two orders of magnitude faster for equivalent or near-equivalent accuracy at this task level.

RAG Chat: Accuracy 75%

75% accuracy on ten factual questions, with five fully correct and five partially correct answers and none fully wrong. Partial answers typically identified the correct topic but missed specific numerical values or dates. Average query latency of 0.15s is well within interactive response time expectations, reflecting the efficiency of Qdrant retrieval combined with Groq LPU inference [7,9]. The partial-credit losses are almost certainly a retrieval quality issue: hashingbased embeddings miss semantically equivalent but lexically paraphrased queries [8]. Switching to neural embeddings would directly address this without adding much complexity.

System-Level Observations

The 94.2MB RSS memory footprint across all five modules is worth highlighting. The whole system runs on a standard laptop with no GPU, because all heavy inference is offloaded to AssemblyAI and Groq. Local computation covers TextBlob lexicon operations, Qdrant client calls, and Flask request handling. This makes InsightFlow AI deployable on minimal hardware, which is a meaningful practical advantage for small teams, educational institutions, and organizations that cannot provision dedicated AI infrastructure.

VII. DISCUSSION

A. What Worked Well

The API-first architecture worked better than anticipated. Removing GPU-hosted model serving from the equation eliminated the single largest source of implementation risk and allowed development focus to stay on integration, user experience, and feature completeness rather than infrastructure. The total development timeline was meaningfully compressed as a result, and the system that emerged is more deployable than the proposed architecture would have been.

The transcript-centric data model proved genuinely robust. The RAG chat module, which was not in the original design at all, was added by implementing one service class and two routes with no changes to the core data layer. That kind of extensibility does not happen by accident; it is a direct consequence of a well-chosen central data model.

The zero-shot LLM performance on summarization and action-item extraction was the most pleasant surprise. Achieving ROUGE-1 of 0.3474 and F1 of 1.0 without any fine-tuning, on tasks that the proposed architecture would have required substantial training compute to handle, validates the API-first approach in a concrete way.

The multi-service translation cascade also worked reliably in practice. Chaining MyMemory, LibreTranslate, and Google Translate public endpoints delivers functional translation across most supported languages at no cost, and the cascade is invisible to the user.

B. What Did Not Work as Planned

The YouTube processing gap is the most visible failure. A method is referenced in the application router that does not exist in the module. This is a classic parallel development integration problem: two parts of the codebase assumed a contract that was never completed. In a production deployment, this would be a broken feature with no user-facing explanation.

The live transcription module's silent dependency on the browser Web Speech API is a more subtle problem. The module works when the API is available, but the assumption is entirely undocumented. A user on an unsupported browser gets silent failure with no useful error message. This is the kind of issue that would be caught immediately in any structured testing process but can persist indefinitely without one.

The hashing-based RAG embeddings, while a reasonable tradeoff for getting the feature shipped, represent the most consequential architectural compromise. The ceiling on retrieval quality is not currently a serious problem at small transcript corpus sizes, but it will become one as the corpus grows. This is a known technical debt item that needs to be addressed before the system would be ready for organizational-scale deployment.

C. Broader Lessons

A few things from this project generalize enough to be worth stating directly.

The most immediate lesson is that zero-shot LLM performance should be treated as the baseline, not an afterthought. The summarization and action item extraction results showed that a well-structured prompt to a strong general-purpose model can match

or exceed smaller taskspecific pipelines without requiring annotated training data, fine-tuning infrastructure, or a separate training cycle. For student projects in particular, this changes the default design logic: instead of beginning with model training, it is often more practical to first establish a zero-shot baseline and only move to fine-tuning when the measured gain is large enough to justify the added complexity [7,8].

The second lesson is that API-first design is not a compromise when the goal is a research-scale deployment. In this project, the main engineering bottlenecks were not algorithmic novelty but infrastructure overhead, model hosting, and maintainability. Self-hosting a large ASR model or a heavy summarization model would have required GPU resources, deployment work, and monitoring effort that were outside the project scope. Managed APIs changed that tradeoff completely by making a usable system possible on modest hardware, which is why the final implementation could focus on integration quality, user experience, and end-to-end reliability rather than on model serving logistics [4,7,9].

A third lesson is that lightweight methods deserve serious consideration before defaulting to transformer-based solutions. The sentiment module is the clearest example of this. TextBlob, combined with a small domain-specific keyword layer, delivered 88% accuracy with very low latency and essentially no inference cost. That result is not interesting because it beats every deeper model in absolute terms. It is interesting because it is good enough for the actual role the module plays in the system, which is to provide a quick indicative signal on meeting tone rather than a high-stakes emotional diagnosis. This is a useful pattern for student engineers to remember: model complexity should follow task requirements, not fashion [10,11].

A fourth lesson is that fallback chains are not optional in production systems. The translation cascade and the keywordbased fallback for retrieval both proved that external dependency failure is not a hypothetical edge case but a normal part of operating a real application. The practical implication is that every service dependency should have an explicit fallback path, and that fallback should be tested as part of the implementation rather than treated as a theoretical safety net. This matters even more in student projects, where the temptation is often to assume that the main service will always be available and that failures can be ignored until later [8,9].

Another broader takeaway is that reliable system design depends as much on the boundaries between modules as on the modules themselves. The transcript-centric structure made it possible to add chat, summarization, sentiment analysis, and action item extraction around a shared data core without repeatedly changing the database layer. That kind of architecture is especially useful for small teams, because it reduces coordination cost and makes future extensions much easier to reason about. For future student projects, the lesson is simple: once a stable central data model is chosen early, many downstream features become significantly easier to add without destabilizing the system.

Finally, the project reinforces the value of shipping a complete but imperfect system over waiting for a theoretically cleaner one that never reaches evaluation. The fact that the implementation could be measured, criticized, and improved produced more useful insight than a purely proposed architecture would have. That is especially relevant for academic student work, where a working system with clear limitations is often a stronger contribution than a polished design that has never been tested in practice.

D. Limitations

The most obvious limitation is the scale of the evaluation itself. The test sets were small across all modules, ranging from four to twenty-five samples, so the reported values should be understood as indicative baselines rather than statistically strong claims. They are useful for confirming that the system works end-to-end and for comparing design choices, but they are not large enough to support broad generalization. This is especially important for the action item and RAG evaluations, where a few cases can easily shift the final numbers in a way that would not survive larger testing.

There are also methodological limitations in how transcription was evaluated. The WER measurement was based on a single audio file, which means the result reflects one recording style, one acoustic environment, and one meeting structure. A more credible evaluation would require a broader recording set with variation in speaker accents, microphone quality, background noise, speaking rate, and conversational overlap. The ICSI Meeting Corpus

would be a reasonable next benchmark because it contains real multi-party meeting recordings and has been widely used in meeting understanding research [8,13,16]. Without that broader test set, the current WER result should be treated as a useful implementation signal, not a final system claim.

The action item extraction evaluation has a similar limitation. An F1 score of 1.0 on five controlled cases is encouraging, but the sample size is too small and the cases themselves were relatively explicit in how action items were phrased. Real meetings often contain implicit assignments, conditional responsibilities, and vague phrasing that are much harder to extract reliably. The current result therefore shows that the module works on clear cases, but it does not yet prove robustness in messy conversational data [12,13]. A larger benchmark with more natural meeting language would be necessary before stronger claims could be made.

A major technical limitation is the dependency on third-party APIs. This affects the system at several layers, including ASR, summarization, translation, and retrieval support. The advantage is that the system remains lightweight and deployable on modest hardware, but the tradeoff is that core functionality depends on external availability, pricing stability, rate limits, and vendor-specific behavior. This creates operational risk that is easy to ignore in development but becomes significant in real deployment. If any of those services change their terms, degrade in quality, or become temporarily unavailable, the user experience will degrade immediately. For that reason, any future version of the system should either add stronger offline alternatives or formalize redundancy at the service layer so that no single API failure can disable the whole workflow [4,7,9].

Another limitation is the lack of concurrent load testing. The system was evaluated in a single-user development setting, which is enough to assess correctness and module behavior but not enough to establish multi-user performance. Flask, SQLite, and the current request pattern may be adequate for a small research deployment, but they are not sufficient evidence of organizational-scale readiness. In particular, SQLite is not a serious long-term choice for concurrent collaborative workloads, and the current architecture would need a more capable database and a proper load-testing pass before it could be presented as production-ready.

The system also has limited multilingual evaluation. Although the implementation includes translation support and multilingual prompting in some modules, the actual evaluation did not meaningfully test multilingual inputs across a representative set of languages. This means the multilingual claims are still mostly architectural rather than empirical. That matters because meeting language is often mixed, domain-specific, and full of code-switching, especially in professional settings. A proper multilingual benchmark would need to test whether transcription, summarization, sentiment analysis, and retrieval all remain stable when the input language changes or when multiple languages appear in the same meeting.

A final limitation is that the current evaluation is still modulecentric rather than user-centric. The system was tested for correctness, speed, and output quality, but not for usability, trust, or long-term adoption. A real organizational deployment would need user studies, task completion analysis, and behavioral feedback from actual meeting participants. Without that layer, the current paper can show that the system works technically, but not yet that it changes user behavior in a durable way.

VIII. CONCLUSION AND FUTURE WORK

Conclusion

InsightFlow AI works. That is the first thing worth saying. Five AI modules are deployed and functional, the RAG chat system is a genuine new contribution that was not in the original design, and the entire system runs in 94.2MB of RAM with no GPU requirement. That is a real outcome.

The technology stack diverged from the original proposal in every major AI component, driven by real-world constraints that the design phase could not fully anticipate. In most cases, the pragmatic choices performed as well as or better than what was proposed: ROUGE-1 of 0.3474 versus the BART-Large-CNN AMI benchmark of 0.308 [6], actionitem F1 of 1.0 against a target of 0.35 [1], and sentiment accuracy of 88% against a 70% target [1]. The ASR WER of 12.95% missed the 8% target, but in the context of real multispeaker meeting audio, it is an explainable and acceptable result.

The central finding is that API-served general-purpose LLMs frequently meet or exceed task-specific fine-

tuned architectures at meaningfully lower cost and complexity. This matters for how future meeting AI systems are designed, proposed, and evaluated. Treating fine-tuning as a default rather than a last resort is a pattern worth questioning.

Future Directions

Eight directions follow directly from what the evaluation revealed.

1. WebSocket-based live transcription. The current module relies on client-side ASR. Replacing this with serverside WebSocket streaming using the AssemblyAI realtime API would enable genuine sub-500ms latency live transcription, remove the browser API dependency, and make the live feature reliable across all clients.

2. Semantic embeddings for RAG.

The Hashing Embedding Service should be replaced with a lightweight dense model such as sentence-transformers/all-MiniLM-L6-v2. It runs on CPU at acceptable indexing latency, costs nothing per call, and directly addresses the retrieval quality issues that caused partial-credit losses in the chat evaluation [8].

3. Third-party platform integration. Completing the Google Calendar, Trello, and Jira integrations via OAuth 2.0 would close Research Gap 5 from [1] and allow extracted action items to flow automatically into tracked task management systems without any manual intervention.

4. Mobile support. The existing Flask REST API is ready to serve a mobile client. A React Native or Flutter application consuming the /api endpoints would extend InsightFlow AI to mobile meeting participants, with live transcription being particularly valuable in that context.

5. Advanced analytics. A dedicated analytics backend tracking meeting frequency, sentiment trends, actionitem completion rates, and speaker participation over time would transform InsightFlow AI from a per-meeting tool into something with genuine longitudinal organizational value.

6. Evaluation at scale. Measuring WER on a diverse recording set and action-item F1 on the ICSI Meeting Corpus [13] would establish whether the results here hold up under more challenging conditions and enable direct comparison with published baselines.

7. Speaker-aware insights. The current system stores speaker labels, but those labels are not yet used for deeper behavioral analysis. Extending the pipeline to compute speaker-specific speaking time, interruption frequency, and contribution balance would make the platform more useful for meeting facilitation and team analysis.

8. Permission-aware collaboration. A role-based sharing layer would allow transcripts, summaries, and action items to be shared selectively across participants, managers, and external collaborators. This would make the platform more suitable for organizational deployment where access control is not optional.

ACKNOWLEDGMENT

This research was conducted as part of the final-year B.E. AI/ML Engineering project at ISB&M College of Engineering, Pune. The authors thank Prof. Prajakta Puranik for her guidance and support throughout both phases of the project. The authors acknowledge the open-source communities behind Flask, SQLAlchemy, Qdrant, TextBlob, and the rouge-score library, and the teams at AssemblyAI and Groq for providing the API access that made this implementation feasible.

REFERENCES

[1] A. Abhay, S. D. Marathe, H. N. Godse, and P. Puranik, “From audio to action: An integrated framework for automated meeting documentation and task management,” International Journal of Creative Research Thoughts, vol. 13, no. 11, pp. 1–9, Nov. 2025.

[2] K. Kogul and S. Narayanan, “Meeting understanding in multimodal conversations,” in Proc. ICML Workshop on Multimodal Learning, 2022, pp. 1–8.

[3] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, “Robust speech recognition via large-scale weak supervision,” in Proc. Int. Conf. Machine Learning (ICML), Baltimore, MD, USA, Jul. 2023. [Online]. Available: https://arxiv.org/abs/2212.04356

[4] F. M. Ramirez et al., “Anatomy of industrial scale multilingual ASR,” arXiv preprint arXiv:2404.09841, 2024. [Online]. Available: https://arxiv.org/abs/2404.09841

[5] Y. Khare, T. Peyash, A. Vanzo, and T. Yoshioka, “Universal-2-TF: Robust all-neural text

formatting for ASR," arXiv preprint arXiv:2501.05948, 2025. [Online]. Available: https://arxiv.org/abs/2501.05948

[6] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in Proc. 58th Annual Meeting of the Association for Computational Linguistics (ACL), Seattle, WA, USA, Jul. 2020, pp. 7871–7880.

[7] Groq Inc., "Groq LPU inference engine leads in first independent LLM benchmark," Groq Newsroom, Mountain View, CA, USA, Feb. 2024. [Online]. Available: https://groq.com/newsroom/groq-lpu-inference-engine-leads-in-first-independent-llm-benchmark

[8] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Kuttler, M. Lewis, W.-T. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in Proc. 34th Conf. Neural Information Processing Systems (NeurIPS), 2020, pp. 9459–9474. [Online]. Available: https://arxiv.org/abs/2005.11401

[9] Qdrant Solutions GmbH, Qdrant: Vector Database for AI Applications, Technical Documentation, 2023. [Online]. Available: https://qdrant.tech/documentation/

[10] S. Loria, TextBlob Documentation, Release 0.15, 2018. [Online]. Available: https://textblob.readthedocs.io/

[11] Y. Hsu, J. Tsai, and S. Lee, "RoBERTa-based sentiment classification in dialogues," in Proc. 59th Annual Meeting of the Association for Computational Linguistics (ACL), 2021, pp. 2345–2356.

[12] S. Banerjee, C. Rose, D. Traum, and J. Allen, "Action item extraction from meeting transcripts," in Proc. NAACL-HLT Workshop on Computational Models of Interaction, 2012, pp. 12–20.

[13] A. Janin et al., "The ICSI meeting corpus," in Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP), vol. 1, Hong Kong, Apr. 2003, pp. I-364–I-367.

[14] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in Proc. ACL Workshop on Text Summarization Branches Out, Barcelona, Spain, Jul. 2004, pp. 74–81.

[15] J. Quan, J. Chen, D. Wang, and S. Xie, "Pyannote.audio: Neural building blocks for speaker diarization," in Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP), Barcelona, Spain, May 2021, pp. 7512–7516.

[16] A. Akbik, T. Bergmann, D. Blythe, K. Rasul, S. Schweter, and R. Vollgraf, "FLAIR: An easy-to-use framework for state-of-the-art NLP," in Proc. 2019 Conf. North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT), Minneapolis, MN, USA, Jun. 2019, pp. 54–59.