iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** V **Month of publication:** May 2026

DOI: https://doi.org/10.22214/ijraset.2026.81874

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue V May 2026- Available at www.ijraset.com

AI-Enhanced Real-Time Messaging Platform with Integrated Third-Party API Services for Intelligent Communication

David Shalom M[1], Kabilan K[2], N. Ananda Kumar[3]
[1, 2]*Computer Science and Engineering Arunai Engineering College Tiruvannamalai, India*
[3]*M.E., M.C.A., M.Phil., Assistant Professor Computer Science and Engineering, Arunai Engineering College Tiruvannamalai, India*

Abstract: ***This project came out of a simple frustration. Every chat application we use daily — WhatsApp, Telegram, Instagram DMs — does one thing well: deliver messages fast. But the moment you need to fix your grammar before sending, translate what someone said, or adjust your tone for a formal conversation, you leave the app. You open Google Translate in one tab, Grammarly in another, maybe even ChatGPT to rephrase something. The flow breaks every single time.***
We wanted to see if we could build a messaging platform where all of that intelligence lives inside the chat itself. Not as a separate bot you have to talk to, but as tools wired directly into the input box and message bubbles. You type a message, tap a button, and it gets rewritten in a professional tone. You receive a message in French, and you translate it without leaving the conversation. You want to send a custom image — just describe it in a text prompt.
We built the system using the MERN stack with Socket.IO handling real-time message delivery. For the AI features, we chose Google Gemini and Stability AI — both accessible through REST APIs, both with free tiers that fit a student project budget of exactly zero rupees. Media uploads go through Cloudinary. Location sharing uses the browser's built-in Geolocation API.
The system works. Messages arrive instantly. Grammar cor- rection, tone modification, translation, and summarization all function as expected through the Gemini API. Image generation through Stability takes around ten seconds but produces usable results. We also implemented typing indicators, read receipts with blue ticks, and emoji reactions — features that users expect from any modern chat app but that are surprisingly tricky to get right with Socket.IO room management. This paper covers the architecture, what we built, what broke along the way, and what we learned from it.
Index Terms: real-time messaging, Socket.IO, MERN stack, third-party AI APIs, Google Gemini, Cloudinary, chat applica- tion, WebSocket

I. INTRODUCTION

A. Motivation and Problem Statement

The idea for this project did not come from reading research papers. It came from watching ourselves use five different apps to do what should happen inside one. A typical scenario: you are chatting with a college senior about a job referral. You want your message to sound professional. So you type it in WhatsApp, copy it, open ChatGPT, ask it to rewrite the message in a formal tone, copy the output, go back to WhatsApp, paste it, and send. That is six steps for something that should take one [1].
Translation is the same story. A friend sends a message in Tamil or Hindi — not a problem if you speak the language. But if you need a translation, you leave the chat, open Google Translate, paste it, read the result, and come back. The context is gone by then [9]. We figured that if generative AI APIs are now good enough to handle grammar correction, tone modification, and transla- tion through simple HTTP calls [2], [3], the real engineering problem is not whether AI can help — it obviously can. The real problem is building the plumbing that connects these API services into a real-time messaging system where everything feels native and fast.
That is what this project attempts to solve.

B. Objectives and Scope

We set four clear goals at the start. First, build a functional real-time chat system with instant message delivery using WebSockets. Second, integrate third-party AI APIs (Google Gemini for text, Stability AI for images) so that intelligent features work directly inside the chat interface. Third, imple- ment the modern chat mechanics that users take for granted— typing indicators, read receipts, emoji reactions, media sharing, and location sharing. Fourth, keep all API keys and sensitive operations on the server side so nothing leaks to the browser.

©IJRASET: All Rights are Reserved

We are two final-year students working under a faculty guide. The scope reflects that reality. We built what we could build properly in the time we had, and we are straightforward in this paper about what works, what does not, and what we would change.

C. Contributions

1) A working real-time chat platform built with Node.js, Ex- press, React, MongoDB, and Socket.IO, supporting text, images, video, audio, files, and GPS location messages.
2) Server-side integration of Google Gemini API for gram- mar correction, tone modification, multi-language transla- tion, conversation summarization, and sentiment analysis with response caching and rate limiting.
3) Integration of Stability AI API for text-to-image gener- ation, with results uploaded to Cloudinary and delivered as chat messages.
4) Implementation of real-time chat features including typ- ing indicators, read receipts (grey and blue ticks), and per-message emoji reactions using Socket.IO room-based broadcasting.
5) An honest account of the integration problems we en- countered and how we resolved them.

D. Paper Organization

Section II reviews related work. Section III describes our technology choices. Section IV explains the system architec- ture. Section V covers the implementation details. Section VI describes the AI API integration pipeline. Section VII presents evaluation results. Section VIII concludes.

II. RELATED WORK

A. Real-Time Communication and WebSocket Protocols

Real-time messaging on the web has evolved significantly from the days of HTTP long-polling. The WebSocket proto- col established a full-duplex communication channel over a single TCP connection, which made true instant messaging possible in the browser [4]. Socket.IO built on top of this by adding automatic reconnection, room-based broadcasting, and graceful fallback to polling when WebSockets are unavailable. Most modern chat tutorials use Socket.IO, but they typically stop at basic message sending. Features like typing indicators and read receipts require careful management of socket rooms and event broadcasting patterns that are not covered well in most documentation.

B. AI-Assisted Communication Tools

The emergence of powerful language models accessible through REST APIs changed what small teams can build. Services like Google Gemini and OpenAI provide endpoints that accept natural language prompts and return structured responses [2], [3], [6]. Grammarly popularized the idea of AI- assisted writing, but it operates as a separate layer on top of existing applications. It does not integrate into the messaging flow itself. What we attempted is different — embedding the AI assistance directly into the send flow so the user never has to leave the conversation context.

C. Modern Chat Application Features

WhatsApp set the standard for what users expect from a messaging app: delivery confirmation (single tick), read confirmation (blue ticks), typing indicators, and media sharing [13]. Recreating these in a custom application is harder than it looks. Read receipts, for instance, require tracking which users have actually opened a thread and updating the sender's UI in real time. Typing indicators need debouncing to avoid flooding the socket server with events on every keystroke. These are engineering problems, not research problems, but they consume significant development time.

III. TECHNOLOGY STACK

We used the MERN stack — MongoDB, Express.js, React, and Node.js. React handles the frontend with a glassmorphism- inspired dark theme. Express and Node.js serve the REST API and the Socket.IO server from the same process. Mon- goDB stores users, threads, and messages through Mongoose schemas.

Socket.IO was chosen over raw WebSockets because it handles reconnection automatically and provides room abstrac- tions that map cleanly to chat threads. When a user opens a conversation, their socket joins that thread's room. Messages, typing events, and reaction updates are broadcast only to sockets in that room.

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

For media storage we used Cloudinary. The alternative was storing files as binary data in MongoDB, which we tried briefly and abandoned because it made database queries noticeably slower and complicated backups. Cloudinary accepts uploads through a simple API, returns a CDN URL, and handles format optimization automatically [8].

For AI we chose Google Gemini because it has a free API tier. That was honestly the deciding factor. We are students with no project budget. OpenAI charges from the first call. Gemini gave us enough free quota to build, test, and demo without worrying about costs [6]. Stability AI was added for image generation because Gemini does not generate images through its text API [7].

Passwords are hashed with bcrypt before storage [11]. Authentication uses JSON Web Tokens (JWT) issued on login and verified on every protected route [10].

IV. SYSTEM ARCHITECTURE

A. *High-Level Overview*

The system follows a client-server architecture with three external service integrations. The React frontend communi- cates with the Express backend through two channels: REST API calls for standard CRUD operations (authentication, fetch- ing messages, uploading files) and a persistent Socket.IO con- nection for real-time events (new messages, typing indicators, read receipts, reaction updates). The backend acts as a gateway to three external services. Gemini handles all text-processing AI tasks. Stability AI han- dles image generation. Cloudinary handles media file storage and delivery. All API keys live exclusively on the server. The browser never sees them.

This separation was not something we got right on the first attempt. Early in development, we had the Gemini API key hardcoded in a React component and were making API calls directly from the browser. It worked, but anyone opening the browser's developer tools could see the key in plain text in every network request. We moved everything server-side once we realized how exposed it was.

B. *Data Model*

We have three core Mongoose models. The User model stores username, phone number, hashed password, and online status. The Thread model links participants and tracks the last message for the sidebar preview. The Message model stores the content, sender reference, message type (text, image, video, audio, file, location), optional media URL, optional location coordinates, a reactions array (userId and emoji pairs), a readBy array (user IDs who have seen the message), and a timestamp.

The reactions and readBy arrays were added to the Message schema after the core chat was working. Adding them was straightforward in MongoDB since you do not need migrations you just update the schema and new documents include the fields. Older messages without these fields default to empty arrays in the application logic.

C. *Socket Event Architecture*

Socket.IO events are organized into five categories. Connec- tion events handle user registration and online status broadcast- ing. Message events handle sending and receiving messages within thread rooms. Typing events handle broadcasting typing and stop-typing signals with debouncing. Reaction events han- dle adding, changing, or removing emoji reactions on specific messages and broadcasting the updated reactions array. Read receipt events handle marking messages as read by a user and notifying the sender.

The distinction between socket.to() and io.to() turned out to be important. socket.to(threadId).emit() sends an event to everyone in the room except the sender — correct for typing indicators, because you do not want to see "You are typing" on your own screen. io.to(threadId).emit() sends to everyone including the sender — correct for reactions, because you want to see your own emoji appear immediately.

D. *Security Considerations*

Passwords go through bcrypt with a salt factor of 10 before storage. JWTs are signed with a server-side secret and expire after seven days. Every protected route runs through an authentication middleware that extracts and verifies the token from the Authorization header. We hit a real problem with JWT handling during devel- opment. Some routes were reading the token correctly, but the analytics and thread-fetching routes were returning 401 errors for users who were clearly logged in. It took us a full day to figure out that the issue was in how the frontend was attaching the token to different types of requests — some used the Authorization header, others forgot it entirely. Once we standardized the token attachment across all axios calls, the problem disappeared.

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue V May 2026- Available at www.ijraset.com

V. IMPLEMENTATION

A. Authentication Flow

Registration takes a username, phone number, and pass- word. The password gets hashed with bcrypt, the user record is saved to MongoDB, and a JWT is returned to the client. Login verifies credentials against the stored hash and returns a fresh token. The frontend stores this token in localStorage and attaches it to every subsequent request. We know localStorage is not the most secure storage for tokens — HttpOnly cookies would be better for production. But for a college project where the main threat model is "does it work correctly during the demo," localStorage was simpler and we went with it.

B. Real-Time Messaging Core

When a user sends a message, the frontend emits a send_message event through the socket connection with the thread ID, message content, sender ID, message type, and optional media URL or location data. The server receives this event, creates a new Message document in MongoDB, saves it, and broadcasts it to the thread room via receive_message. Every client in that room receives the event and appends the new message to their local state.

This happens fast enough that messages appear essentially instantly on local testing. The bottleneck is never the socket broadcast — it is the MongoDB write, which typically takes 20 to 50 milliseconds.

C. Typing Indicators

When the user types in the input box, the frontend emits a typing event. The server broadcasts display_typing to everyone else in the room. To avoid flooding the server with events on every keystroke, we implemented client-side debouncing: the typing event fires on the first keystroke, and a 2-second timeout automatically emits stop_typing if the user pauses. If the user sends the message before the timeout expires, a stop_typing is emitted immediately alongside the message. On the receiving end, the typing indicator shows bouncing dots with the username of whoever is typing. It disappears when the hide_typing event arrives.

D. Read Receipts

When a user opens a thread, the frontend fetches the message history and identifies messages that were not sent by the current user and do not already include the current user's ID in their readBy array. It emits a mark_read event with those message IDs. The server uses MongoDB's updateMany with $addToSet to add the user's ID to the readBy field of all specified messages in one database operation, then broadcasts a messages_read event to the room.

On the sender's side, the message bubble shows grey double-ticks by default (delivered) and switches to blue double-ticks when the readBy array becomes non-empty. This matches the WhatsApp convention that users already understand.

E. Emoji Reactions

Each message has a reaction button that opens a small emoji picker (six preset emojis). Clicking one emits an add_reaction event. The server finds the message, checks if the user already reacted. If they tapped the same emoji, it removes the reaction (toggle off). If they tapped a different emoji, it updates their reaction. If they had no reaction, it adds one. The updated reactions array is saved and broadcast to the room.

Reactions display as small emoji badges at the bottom of the message bubble, with a count. The implementation is simple but getting the toggle logic right required careful handling of the array index operations.

F. Rich Media and Location Sharing

File uploads go through a standard multipart form sub- mission to the Express upload route. The server uses Multer middleware configured with Cloudinary storage. Cloudinary processes the file, stores it on their CDN, and returns a secure URL. The server then emits a message event with the URL and appropriate type flag (image, video, audio, or file).

Location sharing uses the browser's Geolocation API with enableHighAccuracy: true. When the user taps the location button, the browser requests GPS permission, retrieves coordinates, and the frontend emits a location-type message with latitude and longitude. The receiving client renders a styled map card with the coordinates and a link to Google Maps.

We found that desktop browsers often return approximate locations based on IP address rather than precise GPS co- ordinates. Mobile browsers are significantly more accurate. We added enableHighAccuracy and set maximumAge to zero to get the best possible result, but accuracy ultimately depends on the device hardware.

©IJRASET: All Rights are Reserved

VI. AI API INTEGRATION PIPELINE

A. Service Layer Architecture

All AI features route through a server-side service layer. The controller receives the request, validates input, calls the appropriate service function, and returns the result. The service layer handles the actual API communication, including prompt construction, response parsing, caching, and rate limiting.

We implemented an in-memory cache with a 30-minute TTL and a maximum of 500 entries. The cache key is computed from the task type and a prefix of the input text. This was added after we noticed during testing that the same translation or grammar correction was being requested multiple times in a single chat session — every duplicate request was wasting API quota for identical output. The cache eliminated that waste and also made responses feel faster on repeated use.

Rate limiting adds a minimum gap of one second between consecutive API calls, with the gap increasing after consec- utive failures to avoid hammering a service that might be temporarily overloaded.

B. AI Task Overview

Table I maps each AI feature to its external API provider and the format of the data returned to the frontend.

TABLE I
AI FEATURE TO API MAPPING

Feature	API Provider	Output Format
Grammar Fix	Google Gemini	JSON {fixed: text}
Tone Change	Google Gemini	Plain rewritten text
Translation	Google Gemini	Plain translated text
Summarization	Google Gemini	Bullet-point text
Sentiment	Google Gemini	JSON {sentiment, emoji}
Image Gen	Stability AI	Base64 → Cloudinary URL

C. Text Processing Features (Gemini API)

1) Grammar Correction. Sends the user's text with a struc- tured prompt asking for a JSON response containing the corrected text. The prompt includes an example input-output pair to guide the response format. The server parses the JSON and returns the corrected text to the frontend, which displays it in a modal where the user can accept or reject the suggestion. Gemini occasionally returns JSON wrapped in markdown code fences — we added a sanitization step that strips these fences before parsing.
2) Tone Modification. Accepts the user's text and a target tone (professional, casual, funny, or romantic). The prompt instructs Gemini to rewrite the text while preserving the core meaning. The response is plain text with no JSON parsing needed, making this the most reliable of all our AI features.
3) Multi-Language Translation. Takes the message content and a target language selected from a dropdown supporting ten languages (English, Hindi, Tamil, Spanish, French, German, Chinese, Japanese, Russian, Arabic). We take only the first line of the response to strip any explanations the model might add.
4) Conversation Summarization. Collects the last ten mes- sages in a thread, formats them as a conversation transcript, and asks Gemini for three bullet-point summaries. This runs on demand, not automatically.
5) Sentiment Analysis. Follows the same pattern as summa- rization but returns a JSON object with a sentiment label (Positive, Neutral, Negative, or Heated), an emoji, and a one- sentence explanation.

D. Image Generation (Stability AI API)

Image generation uses the Stability AI API rather than Gemini. The user enters a text prompt, the server sends it to Stability's endpoint, receives a base64-encoded image, uploads it to Cloudinary, and returns the CDN URL. The frontend then sends this URL as an image-type message in the chat. The whole process takes around 10 to 15 seconds, mostly due to diffusion sampling on Stability's servers [7].

©IJRASET: All Rights are Reserved

E. Generation Algorithm

Algorithm 1 describes the cache-first pattern used for every generation request.

Algorithm 1: Cache-first generation with rate limiting

1) **Input:** task label t, prompt text p, JSON mode flag j.
2) Compute cache key k = hash(t, prefix(p)).
3) If a valid entry exists for k in cache, return it immedi- ately.
4) Apply rate limiting — wait if minimum gap not met.
5) Call GeminiAPI(p, j) to get output y.
6) If j is true, sanitize and parse y as JSON.
7) Store y in cache with 30-minute expiry.
8) Return y to the controller.

VII. RESULTS AND EVALUATION

A. Evaluation Approach

We evaluated the system on three dimensions: whether the core features work end-to-end, what the response times look like for different operation types, and how people reacted when they used it. We did not measure communication effectiveness with formal instruments — that would require a controlled study beyond our scope.

B. Functional Validation

Table II lists every feature tested and its validation status. We tested with both of us logged into different accounts simultaneously.

TABLE II
END-TO-END FUNCTIONAL VALIDATION

Feature	Status	Issues Found
User Registration / Login	Pass	0
Thread Creation	Pass	0
Text Messaging (Socket.IO)	Pass	0
Image / Video Upload	Pass	0
Audio Recording & Upload	Pass	0
File Sharing (PDF, Doc)	Pass	0
Location Sharing	Pass	0
Typing Indicators	Pass	1*
Read Receipts (Blue Ticks)	Pass	1*
Emoji Reactions	Pass	1*
AI Grammar Correction	Pass	0
AI Tone Modification	Pass	0
AI Translation	Pass	0
AI Summarization	Pass	0
AI Sentiment Analysis	Pass	0

*Fixed during development. See Section VII-B.

Three issues came up during testing that we fixed before finalizing. First, the typing indicator was not showing because the user was emitting the typing event before their socket had joined the thread room. Second, read receipts only updated after a page refresh because the frontend was missing the messages_read socket listener. Third, the reaction toggle had an off-by-one error in the array splice operation.

C. Response Time Observations

Table III shows the mean response times across approxi- mately 20 runs per category.

Database and socket operations stay well under one second. Gemini API calls land between 1.5 and 3 seconds depending on input complexity. Image generation averages around 12.5 seconds — slow, but acceptable when the UI shows a progress indicator. Users tolerated all AI response times when visual feedback was present [13].

TABLE III
MEAN RESPONSE TIMES BY OPERATION TYPE

Operation	Mean Latency
Authentication (Login/Register)	~120 ms
Fetch Messages (MongoDB)	~80 ms
Socket.IO Message Delivery	<50 ms
Media Upload (Cloudinary)	~1.2 s
Grammar Correction (Gemini)	~1.8 s
Tone Modification (Gemini)	~1.5 s
Translation (Gemini)	~2.0 s
Summarization (Gemini)	~3.2 s
Sentiment Analysis (Gemini)	~2.8 s
Image Generation (Stability)	~12.5 s

D. Qualitative Observations

During informal walkthroughs with classmates, three things stood out. The tone changer got the strongest reaction — peo- ple immediately started typing casual messages and converting them to professional tone, laughing at the results. Translation was used heavily by students who communicate in a mix of Tamil and English. The read receipts and typing indicators made the app feel "real" in a way that the AI features alone did not — people said it felt like an actual chat app rather than a project demo.

E. Comparison with Existing Platforms

Table IV positions our system against widely used messag- ing platforms.

TABLE IV
FEATURE COMPARISON WITH EXISTING PLATFORMS

Feature	Ours	WhatsApp	Telegram	Discord
Real-time Messaging	✓	✓	✓	✓
Typing Indicators	✓	✓	✓	✓
Read Receipts	✓	✓	✓	–
Emoji Reactions	✓	✓	✓	✓
Media Sharing	✓	✓	✓	✓
Location Sharing	✓	✓	✓	–
Built-in Grammar Fix	✓	–	–	–
Built-in Tone Change	✓	–	–	–
Built-in Translation	✓	–	✓	–
AI Summarization	✓	–	–	–
AI Image Gen	✓	–	–	–
Sentiment Analysis	✓	–	–	–

The key differentiator is not any single feature — it is that all AI capabilities are embedded natively within the messaging interface rather than requiring users to switch to external tools.

VIII. CONCLUSION

We set out to build a chat application where AI assistance is not a separate experience but something woven into the messaging flow. The result is a working platform that handles real-time communication through Socket.IO, integrates Google Gemini for text intelligence and Stability AI for image gen- eration, and implements the chat mechanics that users expect from any modern messaging app. The hardest parts were not the AI integration — calling an API is straightforward once you have the plumbing right. The hardest parts were the real-time features. Getting typing indicators to feel responsive without flooding the socket server. Making read receipts update correctly across multiple con- current sessions. Handling the edge cases in emoji reaction toggling. These are small problems individually, but they consumed more development time than the entire AI pipeline. We learned more building this than we expected. Not just about the technologies, but about how many small decisions go into making software feel polished rather than just functional.

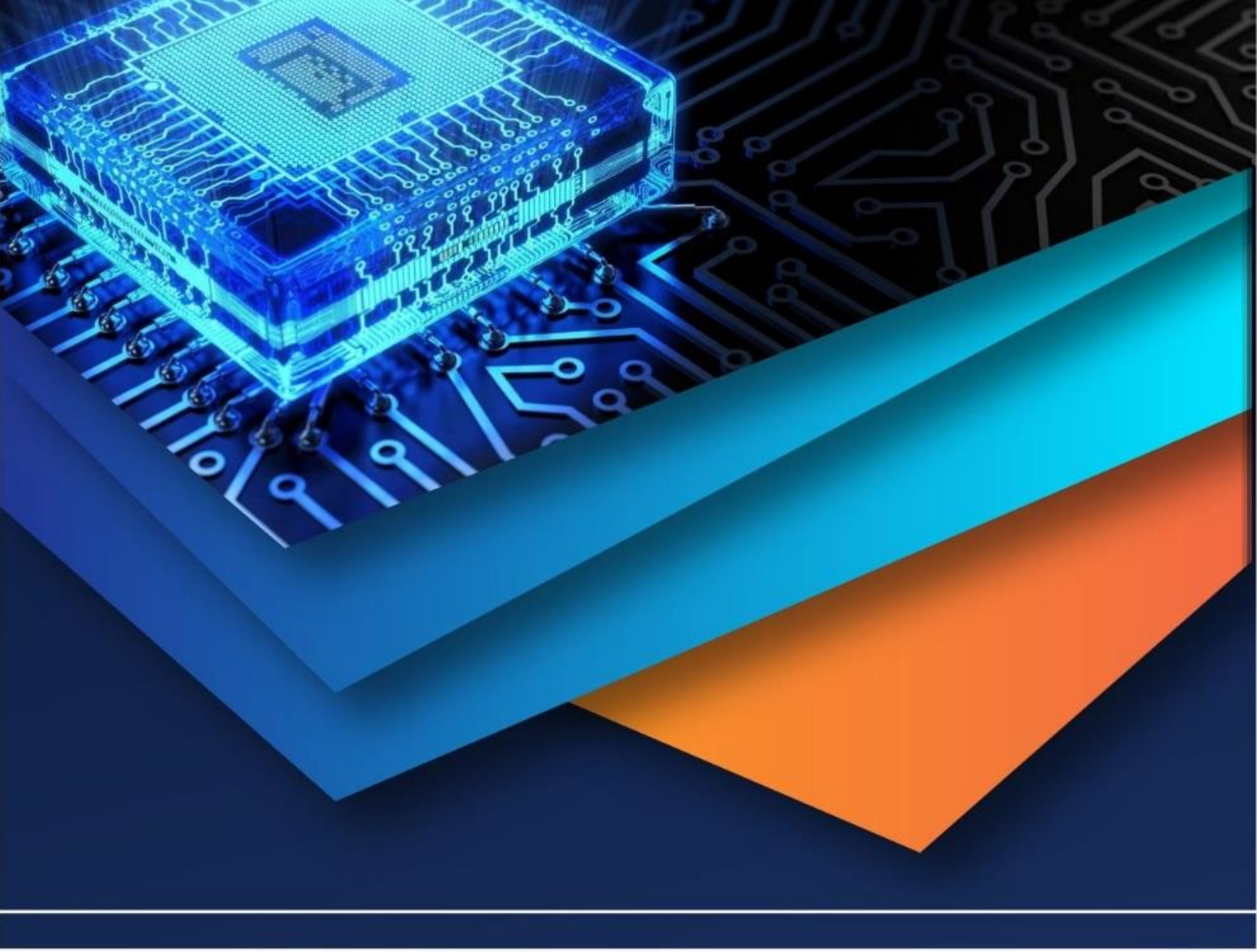
A. *Future Work*

1) Stream Gemini API responses to reduce perceived la- tency on grammar and translation features.
2) Add end-to-end encryption for message privacy using a protocol like Signal.
3) Implement push notifications for mobile browser users.
4) Migrate JWT storage from localStorage to HttpOnly cookies for improved security.
5) Add group chat support with multiple participants per thread.
6) Conduct a formal usability study with structured feed- back instruments.
7) Add rate limiting and cost monitoring for the AI APIs before any public deployment.

REFERENCES

[1] S. Hrastinski, “Asynchronous and synchronous e-learning,” *Educ. Quar- terly*, vol. 31, no. 1, pp. 51–55, 2008.
[2] J. Wei et al., “Emergent abilities of large language models,” Trans. Mach. Learn. Res., 2022.
[3] T. Brown et al., “Language models are few-shot learners,” in Proc. Adv. Neural Inf. Process. Syst. (NeurIPS), vol. 33, 2020, pp. 1877–1901.
[4] V. Pimentel and B. G. Nickerson, “Communicating and displaying real- time data with WebSocket,” *IEEE Internet Comput.*, vol. 16, no. 4, pp. 45–53, Jul./Aug. 2012.
[5] MongoDB Inc., “Mongoose Documentation,” 2025. [Online]. Available: https://mongoosejs.com/docs/guide.html
[6] Google, “Gemini API Documentation,” 2025. [Online]. Available: https://ai.google.dev/gemini-api/docs
[7] Stability AI, “Platform API Reference,” 2025. [Online]. Available: https://platform.stability.ai/docs/api-reference
[8] Cloudinary, “Cloudinary Documentation,” 2025. [Online]. Available: https://cloudinary.com/documentation
[9] D. M. West, “Education technology can help address inequality in a high inequality world,” Brookings Institution, TechTank blog, 2022.
[10] M. Jones, J. Bradley, and N. Sakimura, “JSON Web Token (JWT),” RFC 7519, IETF, May 2015. [Online]. Available: https://datatracker.ietf.org/ doc/html/rfc7519
[11] N. Provos and D. Mazie`res, “A future-adaptable password scheme,” in Proc. USENIX Annu. Tech. Conf. (ATEC), 1999, pp. 81–91.
[12] R. T. Fielding and R. N. Taylor, “Principled design of the modern web architecture,” ACM Trans. Internet Technol., vol. 2, no. 2, pp. 115–150, 2002.
[13] J. Nielsen, “Response times: The 3 important limits,” Nielsen Norman Group, 1993. [Online]. Available: https://www.nngroup.com/articles/ response-times-3-important-limits/
[14] Socket.IO, “Socket.IO Documentation,” 2025. [Online]. Available: https://socket.io/docs/v4/
[15] Meta Open Source, “React Documentation,” 2025. [Online]. Available: https://react.dev

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)