SIDDAGANGA INSTITUTE OF TECHNOLOGY, TUMAKURU-572103

(An Autonomous Institute under Visvesvaraya Technological University, Belagavi)

Project Report on

"Kuma: AI-Powered Personal Assistant"

submitted in partial fulfillment of the requirement for the completion of V semester of

BACHELOR OF ENGINEERING

in

ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

Submitted by

Himanshu Rai (1SI23AD016)

Suraj Kumar (1SI23AD057)

Aditya Raj (1SI23CS008)

under the guidance of

Dr Sheela S

Assistant Professor

Department of Computer Science and Engineering

SIT, Tumakuru-03

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

2025-26

SIDDAGANGA INSTITUTE OF TECHNOLOGY, TUMAKURU-572103

(An Autonomous Institute under Visvesvaraya Technological University, Belagavi)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

Certified that the mini project work entitled "KUMA: AI-POWERED PERSONAL ASSISTANT" is a bonafide work carried out by Himanshu Rai (1SI23AD016), Suraj Kumar (1SI23AD057), Aditya Raj (1SI23CS008) in partial fulfillment for the completion of V Semester of Bachelor of Engineering in Artificial Intelligence and Data Science from Siddaganga Institute of Technology, an autonomous institute under Visvesvaraya Technological University, Belagavi during the academic year 2025-26. It is certified that all corrections/suggestions indicated for internal assessment have been incorporated in the report deposited in the department library. The Mini project report has been approved as it satisfies the academic requirements in respect of project work prescribed for the Bachelor of Engineering degree.

Dr Sheela S	Head of the Department
Assistant Professor	Dept. of CSE
Dept. of CSE	SIT,Tumakuru-03
SIT,Tumakuru-03	

External viva:

Names of the Examiners	**Signature with date**
1.	
2.	

ACKNOWLEDGEMENT

We offer our humble pranams at the lotus feet of **His** Holiness, **Dr. Sree Sree Sivakumara Swamigalu**, Founder President and **His** Holiness, **Sree Sree Siddalinga Swamigalu**, President, Sree Siddaganga Education Society, Sree Siddaganga Math for bestowing upon their blessings.

We deem it as a privilege to thank **Dr. Shivakumaraiah**, CEO, SIT, Tumakuru, and **Dr. S V Dinesh**, Principal, SIT, Tumakuru for fostering an excellent academic environment in this institution, which made this endeavor fruitful.

We would like to express our sincere gratitude to **Dr Sunitha. N R**, Professor and Head, Department of Computer Science and Engineering, SIT, Tumakuru for her encouragement and valuable suggestions.

We thank our guide **Dr Sheela S**, Assistant Professor, Department of Computer Science and Engineering, SIT, Tumakuru for the valuable guidance, advice and encouragement.

Himanshu Rai (1SI23AD016)
Suraj Kumar (1SI23AD057)
Aditya Raj (1SI23CS008)

Course Outcomes

- **CO1:** To identify a problem through literature survey and knowledge of contemporary engineering technology.
- **CO2:** To consolidate the literature search to identify issues/gaps and formulate the engineering problem.
- **CO3:** To prepare project schedule for the identified design methodology and engage in budget analysis, and share responsibility for every member in the team.
- **CO4:** To provide sustainable engineering solution considering health, safety, legal, cultural issues and also demonstrate concern for environment.
- **CO5:** To identify and apply the mathematical concepts, science concepts, engineering and management concepts necessary to implement the identified engineering problem.
- **CO6:** To select the engineering tools/components required to implement the proposed solution for the identified engineering problem.
- **CO7:** To analyze, design, and implement optimal design solution, interpret results of experiments and draw valid conclusion.
- **CO8:** To demonstrate effective written communication through the project report, the one-page poster presentation, and preparation of the video about the project and the four page IEEE/Springer/paper format of the work.
- **CO9:** To engage in effective oral communication through power point presentation and demonstration of the project work.
- **CO10:** To demonstrate compliance to the prescribed standards/safety norms and abide by the norms of professional ethics.
- **CO11:** To perform in the team, contribute to the team and mentor/lead the team.

CO-PO Mapping

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PSO1	PSO2	PSo3
CO-1											3		3	3
CO-2		3		3								3	3	3
CO-3										3	3		3	3
CO-4						3	3						3	2
CO-5	3	3											3	2
CO-6					3								3	2
CO-7		3	3	3									3	3
CO-8									3				3	3
CO-9									3				3	3
CO-10							3						3	2
CO-11								3					3	2

Attainment Level: 1: Slight (low), 2: Moderate (medium), 3: Substantial (high)

Program Outcomes (POs):

- **PO1:** Engineering Knowledge
- **PO2:** Problem analysis
- **PO3:** Design/Development of solutions
- **PO4:** Conduct investigations of complex problems
- **PO5:** Engineering tool usage
- **PO6:** Engineer and the world
- **PO7:** Ethics
- **PO8:** Individual and collaborative team work
- **PO9:** Communication
- **PO10:** Project management and finance
- **PO11:** Lifelong learning

Program Specific Outcomes (PSOs):

- **PSO1:** Computer based systems development
- **PSO2:** Software development
- **PSO3:** Computer Communications and Internet applications

Abstract

Modern productivity workflows suffer from frequent context switching between applications, causing cognitive overhead and reduced efficiency. Existing AI assistants like ChatGPT and Google Assistant have limitations in functional integration or multilingual support, particularly for Indian languages, while raising privacy concerns through cloud-based processing. This project develops Kuma, a self-hosted, multilingual AI assistant with deep productivity tool integration.

Kuma implements a multi-agent architecture where specialized agents handle domain-specific tasks through intelligent routing. The system integrates Google Gemini 1.5 Flash and OpenAI GPT-4o via Vercel AI SDK for real-time streaming. The backend uses Bun 1.0 with TypeScript, while the frontend employs React 18 with Vite. Key innovations include Sarvam AI integration for Hindi, Tamil, and Telugu speech recognition and synthesis, LiveKit for real-time audio streaming, and Google Gemini Vision for multimodal capabilities including OCR and visual question answering.

The architecture employs Redis Streams for asynchronous processing, eliminating synchronous timeout issues. OAuth 2.0 integration with Google Workspace (Gmail, Calendar, Drive, Docs, Sheets) and GitHub enables authenticated operations. Security features include JWT authentication and AES-256 token encryption. Docker containerization ensures consistent deployment across environments.

Performance evaluation shows 3.5-second average AI response times with 800ms first-token latency. Voice processing achieves 480ms end-to-end latency. The multi-agent architecture reduced hallucinations by 40% versus single-agent baselines. Hindi speech recognition reached 85% accuracy, outperforming Google Assistant (62%) and Siri (45%). The system handles 10 concurrent users on 4GB RAM hardware. User testing yielded 4.2/5 satisfaction, particularly for context retention. This work demonstrates feasibility of privacy-preserving, multilingual AI assistants with production-grade integration using modern web technologies and LLM APIs.

Contents

Abstract **5**

List of Figures . . . . . **ii**

List of Tables . . . . . **iii**

1 Introduction **1**

1.1 Motivation 2

1.2 Objective of the project 3

1.3 Organisation of the report 4

2 Literature Survey **6**

2.1 Conversational AI and Virtual Assistants 6

2.2 Large Language Models and Agent Frameworks 7

2.3 Google Gemini and Multimodal AI 7

2.4 Speech Processing Technologies 8

2.5 Image Understanding and Vision AI 9

2.6 Message Queue Architectures 9

2.7 Containerization and Deployment 10

2.8 Web Application Technologies 10

3 System Design & Methodology **12**

3.1 Functional & Non-Functional Requirements 12

3.1.1 Functional Requirements 12

3.1.2 Non-Functional Requirements 13

3.2 List of Hardware & Software Requirements 13

3.2.1 Hardware Requirements 13

3.2.2 Software Requirements 14

3.3 System Architecture . . . 14
3.3.1 High-Level Overview . . . 15
3.3.2 Detailed Architecture . . . 16
3.4 Redis Queue Architecture . . . 18
3.5 Voice Processing Architecture . . . 19
3.6 Docker Deployment Architecture . . . 21
3.7 Data Flow Diagrams . . . 22
3.7.1 Context Diagram . . . 22
3.7.2 System Data Flow Diagram . . . 23
3.7.3 Agent Processing Data Flow . . . 23
3.7.4 Voice Processing Data Flow . . . 24
3.7.5 Image Processing Data Flow . . . 25
3.8 Algorithms . . . 25
3.8.1 Chat Processing Algorithm . . . 25
3.8.2 Agent Selection Algorithm . . . 26
3.8.3 Redis Queue Processing Algorithm . . . 26
3.8.4 Voice Processing Algorithm . . . 27
3.8.5 Image Analysis Algorithm . . . 27
3.8.6 Memory Management Algorithm . . . 28
3.8.7 Google Services Connection Flow . . . 28

4 Implementation Details . . . 29

4.1 Backend Implementation . . . 29
4.1.1 Project Setup . . . 29
4.1.2 Database Design . . . 29
4.1.3 API Endpoints . . . 29
4.1.4 AI Integration . . . 30
4.1.5 Voice Processing Implementation . . . 30
4.1.6 Vision and Image Processing . . . 31
4.1.7 Redis Queue Implementation . . . 31
4.1.8 Google Services Integration . . . 31

4.1.9 Other External Services . . . 32
4.1.10 Docker Containerization . . . 32
4.2 Frontend Implementation . . . 32
4.2.1 Project Setup . . . 32
4.2.2 State Management . . . 33
4.2.3 UI Components . . . 33
4.2.4 API Integration . . . 33
4.3 Security Implementation . . . 34
4.4 Code Snippets . . . 34
5 Results . . . 35
5.1 Screenshots . . . 35
5.2 Analysis . . . 37
5.2.1 Performance Metrics . . . 37
5.2.2 Comparison with Existing Systems . . . 38
5.2.3 Testing Results . . . 38
5.2.4 User Feedback . . . 40
6 Conclusion & Future Enhancement . . . 41
6.1 Conclusion . . . 41
6.2 Future Enhancement . . . 42
Bibliography . . . 44
Appendices . . . 50
A Sustainable Development Goals addressed . . . 51
B Self-Assesment of the Project . . . 52
C System Technology Stack . . . 54
C.1 Backend Technologies . . . 54
C.2 Frontend Technologies . . . 54
C.3 Infrastructure and Deployment . . . 55
D Installation Guide . . . 56

D.1 Prerequisites 56
D.2 Quick Start with Docker 56
D.3 Manual Installation 57
E Project Source Code 58
E.1 Repository Structure 58

List of Figures

3.1 High-Level System Overview 16
3.2 System Architecture of Kuma AI Assistant 18
3.3 Redis Message Queue Architecture 19
3.4 Voice Processing Pipeline 20
3.5 Docker Deployment Architecture 22
3.6 Context Diagram 22
3.7 System Data Flow Diagram 23
3.8 Agent Processing Data Flow Diagram 24

5.1 Main Chat Interface 35
5.2 Voice Interaction Interface 36
5.3 Image Analysis Results 36
5.4 Gmail Integration 37

List of Tables

5.1 Text Chat Performance Metrics 37
5.2 Voice Processing Metrics 38
5.3 Vision Processing Metrics 38
5.4 Docker Resource Usage 39
5.5 Comparison with Existing AI Assistants 39

Chapter 1

Introduction

We've all used ChatGPT and similar AI tools, and they're pretty impressive at having conversations. But when we started this project, we wanted to dig deeper into how these systems actually work. Turns out modern AI assistants use transformer models [53]. These are way better than old chatbots like ELIZA [51] which just followed scripts. The big innovation is something called the attention mechanism—basically, the model can focus on the important parts of what you said instead of treating everything equally.

Before building our own system, we spent time testing what's already out there. Google Assistant is decent for simple stuff like timers and music, but when we asked it to read our emails or make calendar events, it didn't work [16]. Alexa has tons of third-party "skills" [17], but using them feels awkward. You can't just say "read my emails"—you have to say "Alexa, ask Gmail to read my emails." That's clunky. Also, we realized all these commercial assistants upload your voice and data to their servers [55]. That didn't sit well with us, especially if we were going to give it access to our emails and documents.

GPT-3 coming out in 2020 was a game-changer [56]. Instead of training your own AI model (which costs millions), you could just call OpenAI's API. We played around with it and found that you can make it do completely different things just by changing how you ask—no special training required. We also looked at running open-source models like LLaMA [57] on our own hardware, but they need 32GB+ of RAM which we don't have. So we decided to use the APIs from OpenAI and Google for now. Maybe later if we get better hardware, we can switch to self-hosted models.

The problem we were trying to solve was real for us. We actually tracked our computer usage for a week and realized we spent about 47 minutes every day just switching between apps—Gmail, Calendar, Google Docs, GitHub, and so on. Studies show this kind of constant switching kills productivity [59]. We thought an AI that could handle tasks across all these apps would save us a lot of time. The tricky part was doing it securely and making sure the AI doesn't do anything dangerous without asking first.

We really wanted voice control to work in Hindi. Most speech recognition works fine for English [61], but when we tested Google's service with Hindi or mixed Hindi-English (like "Yaar, check my GitHub pull requests"), it was terrible. Then we found Sarvam AI [30], an Indian company building speech models specifically for Indian languages. On our tests, their Hindi recognition got 85% accuracy compared to Google's 62%. Their text-to-speech also sounded way more natural than the robotic voices we heard elsewhere [63].

For the architecture, we looked at several approaches. Basic chatbots follow fixed conversation scripts [64], but we needed something flexible. We learned about RAG—Retrieval Augmented Generation [65]—which lets the AI pull in information from external sources. Perfect for our document analysis feature. We also discovered the multi-agent pattern [66]. Instead of one AI trying to do everything, you have multiple specialized AIs for different tasks. We built a router that figures out which specialist to use based on what you're asking. Finally, we had to add Redis Streams [67] for background processing because our first version would timeout after 30 seconds if the AI took too long to respond.

1.1 Motivation

This whole project started because we were fed up during our fourth semester. We were working on a team project and kept having to jump between GitHub, Gmail, Google Sheets, and Calendar constantly. One day we actually counted—47 times in just two hours we switched apps. Every time you switch, you lose focus and waste a few seconds getting back into what you were doing. We figured there had to be a better way.

So we tried Google Assistant first. We asked it "read my latest emails from my project team" and it just couldn't do it. Then we tried Alexa, but it only speaks English well, and some of us prefer Hindi. ChatGPT was interesting—it understood exactly what we wanted and could explain how to use the Gmail API, but it couldn't actually read our emails for us. That was frustrating. Plus, we were uncomfortable with the privacy angle. All these services upload your data to their servers. Did we really want to give them access to our personal emails and project documents?

But then we realized something. The new AI models like GPT-4 and Gemini [21] [2] are incredibly smart—they can follow complex instructions, think through problems, even look at images. And they're available through APIs. So we could build our own assistant that uses these powerful models but runs on our own server. That way we

get the intelligence of ChatGPT combined with the practical integrations we actually need—Gmail, Calendar, GitHub, Drive, all of it. And we could make it work in both English and Hindi. That's basically what Kuma is.

1.2 Objective of the project

Here's what we set out to build:

- Create a self-hosted AI assistant that we'd actually want to use every day. It should combine the smarts of ChatGPT with real integrations to Gmail, Calendar, Drive, and GitHub so it can actually do things for us. After testing different models, we decided to use both Gemini 1.5 Flash (cheaper, good enough for most stuff) and GPT-4o (better for complex questions).

- Build a multi-agent system instead of one big general-purpose AI. We'd have a router that looks at your question and sends it to the right specialist—one agent handles research and web search, another does stock market analysis, and a third one manages email and calendar tasks. When we tested this, it made way fewer mistakes than having one agent try to do everything.

- Get voice working properly in Indian languages. Half our team speaks Hindi, and we wanted voice control that actually works. We integrated Sarvam AI for speech recognition and text-to-speech in Hindi, Tamil, and Telugu. LiveKit handles the audio streaming. Goal was to keep the delay under half a second so it feels like a natural conversation.

- Add vision so you can take a photo of something and ask questions about it. Like you could photograph a handwritten to-do list or an invoice, and Kuma would read it and extract the information. We used Gemini Vision for this—it can do OCR, understand scenes, and answer visual questions.

- Let people upload PDFs and ask questions about them. This uses RAG (retrieval augmented generation)—basically, we extract the text, store it in a way that's easy to search, and when you ask a question, the AI looks up the relevant parts. Super useful for reading through research papers and long documents.

- Connect to Google Workspace and GitHub using OAuth 2.0 properly. The AI needs to be able to actually send emails and create calendar events, not just talk about doing it. This meant figuring out how to store access tokens securely (we encrypt them with AES-256), refresh them automatically when they expire, and handle all the errors that can happen with API calls. This part turned out way more complicated than we thought.

- Fix the timeout problem with async processing. Our first version would just hang for 30 seconds and fail if the AI took too long. We added Redis Streams to handle jobs in the background—the API quickly gives you a job ID, then workers process it separately. This includes retry logic for when things fail and a dead letter queue for completely broken jobs.

- Make it work on everyone's computer with Docker. Our team has Windows, Mac, and Linux machines, and getting everything running consistently was a pain. So we containerized everything—frontend (NGINX), backend API, worker process, PostgreSQL, Redis—and used Docker Compose to tie it all together.

- Build a clean web interface where you can see the AI's response streaming in word by word, just like ChatGPT. Used React 18 with Vite for this. It also needs to handle file uploads for images and documents, and work on both desktop and phone browsers.

1.3 Organisation of the report

We've organized this report into six chapters:

- **Chapter 1 (Introduction):** Covers why we built Kuma, what problems we were trying to solve, and what we aimed to achieve.

- **Chapter 2 (Literature Survey):** Goes through all the research we did—looking at existing AI assistants, understanding LLMs, checking out agent frameworks, testing speech services, and learning about the technologies we'd need.

- **Chapter 3 (System Design & Methodology):** Explains how we designed the system—what hardware and software we used, the architecture we settled on, how data flows through it, and the key algorithms.

- **Chapter 4 (Implementation Details):** Gets into the actual code—how we built the backend, implemented the AI agents, integrated with external services, handled voice processing, added vision capabilities, and created the frontend.
- **Chapter 5 (Results):** Shows what we ended up with—screenshots of the system working, performance numbers, how it compares to other assistants, and feedback from our beta testers.
- **Chapter 6 (Conclusion & Future Enhancement):** Wraps everything up with what we learned, what worked and what didn't, and ideas for making it better in the future.

Chapter 2

Literature Survey

2.1 Conversational AI and Virtual Assistants

Before building anything, we spent a couple weeks just testing existing AI assistants. Google Assistant was our first stop since we all use Android phones. The voice recognition is really good, and it connects nicely with Google services [16]—weather, reminders, smart home stuff all work smoothly. But when we asked it to read our emails or check calendar events, nothing happened. Which is weird because Google makes Gmail and Calendar. Seems like an obvious integration they should have.

Alexa was next. Amazon built this whole ecosystem of third-party "Skills" [17] where developers can add features. Sounds good in theory. In practice, it's messy. Want to check your email? You can't say "read my emails"—you have to say "Alexa, ask Gmail to read my emails." And then each skill has its own login and command structure. We tried a few productivity skills and gave up. Siri had the same issues—good for simple tasks, useless for the kind of deep integrations we wanted.

ChatGPT was different [18]. It doesn't follow scripts like those other assistants. You can have an actual conversation, it remembers what you said earlier, and it can handle weird questions you throw at it. We used it a lot—explaining concepts we didn't understand, writing bits of code, even helping debug. But here's the thing: it can't actually do anything. Ask it to read your email and it'll explain how to use the Gmail API, but it can't access your inbox itself. Still, it showed us what's possible when an AI can actually understand natural language properly.

Then we got into multi-agent systems [19]. The idea is simple: instead of one AI that tries to do everything (and often screws up), you have multiple specialized AIs for different jobs. Each one is good at its specific thing. Research shows this cuts down on the AI making stuff up—what people call "hallucinations"—because each agent only handles what it's trained for. We also learned about RAG [20], which stands for retrieval augmented generation. Basically, instead of the AI only knowing what it was trained on,

it can look stuff up in real-time from external sources. Perfect for our document analysis feature.

2.2 Large Language Models and Agent Frameworks

Once we decided to build our own assistant, we had a choice: train our own AI model from scratch, or use existing models through their APIs. Training something like GPT-4 costs millions of dollars and needs huge amounts of computing power [21]. That's obviously not happening on a student budget. Luckily, OpenAI and Google let you use their models through APIs. We tested both a bunch. GPT-4 is better at complex reasoning and following detailed instructions. Gemini is faster and cheaper for simpler stuff. So we ended up using both—pick the right tool for each job.

We also looked at frameworks for building AI agents. LangChain [1] is popular—it gives you ready-made tools for chaining LLM calls together, managing conversation memory, all that. We tried it first. Honestly, it added more complexity than it solved for us. Too many abstractions when our use case wasn't that complicated. AutoGPT looked cool but it's too experimental, not stable enough for something we wanted to actually use. We ended up writing our own agent system based on the ReAct pattern [3], where the AI explicitly thinks through what it's going to do before doing it. Gave us total control and the code stayed pretty simple.

The multi-agent thing really caught our attention [23]. We built it like this: there's a router agent that looks at your question and decides which specialist should handle it. One agent does research and web searches. Another handles stock market questions with real-time data. A third one manages email, calendar, and money stuff. When we tested this against our old single-agent version, the multi-agent setup made 40% fewer mistakes. Each specialist has its own custom instructions and only the tools it actually needs, so it stays focused and accurate.

2.3 Google Gemini and Multimodal AI

For vision stuff—analyzing images and documents—we went with Google Gemini [2]. We compared it against GPT-4V and a few other models that can handle both images and text. What sold us on Gemini was how it's built. Unlike older systems that just glue together a vision model and a language model, Gemini handles images and text together from the ground up [24]. The whole thing uses the same attention mechanism

for both. We ran tests on about 20 different documents—invoices, handwritten notes, screenshots—and Gemini did better at extracting text, especially from our test invoices. Plus it was way faster. Gemini averaged 2.5 seconds, GPT-4V took 4.2 seconds.

For the document analysis part, Gemini Vision had everything we needed. OCR to pull text from images, smart enough to understand document structure like tables and headers, and can answer questions about what's in an image. We tested it with all kinds of stuff—a handwritten shopping list in Hindi, an invoice with a bunch of tables, a flowchart diagram. It handled all three pretty well, though really messy handwriting still confused it. Compared to old-school OCR like Tesseract, Gemini understands context. It doesn't just see numbers—it knows this number is a price, that one's a quantity, etc.

Price mattered too. Gemini's vision API was about 60% cheaper than GPT-4V when we built this. On a student budget where we're paying for every API call, that makes a difference. There are open-source vision models like CLIP and BLIP-2 [26] we could've self-hosted, but they don't perform as well as the commercial ones for the document understanding stuff we needed [25].

2.4 Speech Processing Technologies

Getting voice to work was tough. We tried three different speech-to-text services: OpenAI's Whisper, Google Cloud Speech-to-Text, and Sarvam AI. All three work great for English—above 95% accuracy [27]. But the real challenge was Hindi and code-mixing (when you switch between English and Hindi in the same sentence, which everyone does in India). We made a test set of 50 voice recordings with stuff like "Yaar, check my GitHub pull requests."

Google's service got 62% accuracy on Hindi, and it kept hearing Hindi words as English. Whisper did better at 74% but was really slow—3 to 4 seconds delay. Sarvam AI is this Indian startup that focuses specifically on Indian languages [30], and they crushed it. 85% accuracy and only 250ms latency. Easy choice, even though they're newer and less proven than Google or OpenAI. Their text-to-speech also sounds way more natural—not robotic like the other options [28].

For real-time voice we needed WebRTC, which we'd never touched before. We spent two days trying to set up our own WebRTC server with STUN/TURN configuration and just got nowhere. Then we found LiveKit [29] [10], which is a managed service that

handles all the WebRTC mess for you. Switched to that and had it working in a few hours. Definitely saved us weeks compared to building it ourselves.

2.5 Image Understanding and Vision AI

We needed good OCR for extracting text from images. Tried Tesseract first since it's free and open-source [32]. Works fine if you're scanning a clean, printed document. But give it handwritten text, a skewed photo, or something with a complicated layout, and it falls apart. We tested with 15 different documents—invoices with tables, handwritten notes, screenshots, whiteboard photos.

Gemini Vision beat Tesseract by a lot [33]. Take a complex invoice with multiple tables. Tesseract would give us all the text but completely lose the structure—just a random jumble of words and numbers. Gemini Vision not only pulled the text but actually understood what it was looking at. It could tell you "this number is the price, that one's the quantity, this is the total." For handwritten notes, Tesseract basically gave up. Gemini got about 70% accuracy, which isn't perfect but way better than nothing. Really fancy handwriting still confuses it though.

We also added visual question answering [35]. You can upload a photo and ask specific questions about it. Like you photograph a whiteboard after a meeting and ask "What are the action items?" Gemini reads the text, understands the context, and pulls out the relevant stuff. This makes our assistant way more useful than text-only systems [36].

2.6 Message Queue Architectures

Our first version was simple and stupid. User sends a message, our API calls the AI model, waits for it to finish, then sends back the response. Worked great when the AI responded in 2-3 seconds. But when it took 10+ seconds (which happened a lot with complex questions), browsers would timeout after 30 seconds. Users got error messages even though the AI was still working in the background. Not good.

We looked at three different message queue systems: RabbitMQ, Kafka, and Redis Streams [37]. RabbitMQ has tons of features and fancy routing [39], but it felt like way too much for what we needed. Kafka is built for massive data streaming [40], but it's complicated to set up and needs ZooKeeper running alongside it. Overkill. Redis Streams [38] was perfect—simpler than RabbitMQ, lighter than Kafka, and we were already using Redis for caching anyway.

We went with a producer-consumer setup. API server publishes jobs to Redis Streams, separate worker processes consume them and do the actual AI processing. Consumer groups make sure each message gets processed exactly once, even if we're running multiple workers [41]. Added retry logic that waits 1 second, then 2, then 4 if things fail. After 3 failed attempts, the message goes to a dead letter queue so we can look at it later. This fixed all our timeout issues and we can now scale by just adding more workers [42].

2.7 Containerization and Deployment

None of us knew Docker when we started. Deployment was a mess. It'd work perfectly on my laptop, then we'd try running it on someone else's machine or a test server and everything would break. Wrong Node version, missing packages, environment variables not set. "Works on my machine" was basically our team motto for a while. Docker fixed all this [8].

First Docker images we made were huge. 2GB for the backend because we stupidly included all the build tools, TypeScript compiler, dev dependencies, everything. Then we learned about multi-stage builds [43]. You compile your TypeScript in one stage with all the build tools, then copy just the compiled JavaScript and production dependencies to a clean stage for running. Brought our backend image down to 450MB. Way faster to deploy and more secure since there's no compiler or build tools sitting around that could be exploited.

Docker Compose runs our five containers together: frontend (NGINX + React), backend API, worker process, PostgreSQL, and Redis [44]. Had to set up health checks for everything. Backend has a /health endpoint that responds when it's ready. PostgreSQL uses pg_isready to check if the database is up. Redis uses redis-cli ping. If a container's health check fails, Docker automatically restarts it. Gave us way more reliability than we could've gotten trying to manage everything manually [45] [46].

2.8 Web Application Technologies

We used TypeScript for everything—frontend and backend [4]. We'd done previous projects in plain JavaScript and wasted so many hours finding bugs at runtime that TypeScript would've caught immediately during development. For running the backend, we compared Node.js, Deno, and Bun. Bun starts up 3x faster than Node.js, which matters a lot for our worker processes that start and stop frequently. Plus it runs TypeScript

directly without needing extra tools like ts-node [13].

Frontend is React 18 with Vite [3]. Vite's hot reload is insanely fast—you save a file and see changes in under 100ms. We started with Create React App but switched because Vite builds 5x faster. For managing state, went with Zustand instead of Redux. Zustand needs like 80% less boilerplate code to do the same thing [47]. UI components come from shadcn/ui—they're accessible, look good, and you can customize them. Way less bloated than Material-UI.

Backend framework is Express [48]. Pretty obvious choice—it's simple, flexible, and we already knew how to use it. Prisma for the database [5]. Every time you change the database schema, Prisma auto-generates TypeScript types for you. Catches database errors before you even run the code. Auth is JWT tokens [49], and we implemented OAuth 2.0 for connecting Google and GitHub. The whole stack is pretty modern and makes development actually enjoyable while still being fast [50].

Chapter 3

System Design & Methodology

3.1 Functional & Non-Functional Requirements

3.1.1 Functional Requirements

Here's what the system actually needs to do:

1. Let users sign up and log in with JWT tokens managing their sessions
2. Show a chat interface where AI responses stream in word-by-word using Server-Sent Events
3. Use multiple specialized AI agents—a router picks which one should handle each question
4. Take voice input and convert it to text using Sarvam AI
5. Convert AI responses back to voice for Hindi, Tamil, and Telugu
6. Let users upload images and analyze them with Gemini Vision
7. Handle PDF uploads, extract text, and answer questions about them using RAG
8. Connect to Google Workspace (Gmail, Calendar, Docs, Drive, Sheets) through OAuth 2.0
9. Work with GitHub—search code, manage issues, handle files
10. Remember stuff long-term using Supermemory
11. Search the web using Exa
12. Process AI requests in the background using Redis Streams and worker processes
13. Run everything in Docker containers

3.1.2 Non-Functional Requirements

And here's what it needs to do well:

1. **Performance:** Get AI responses in 3-5 seconds. Voice latency under 500ms. Start streaming responses within 1 second.

2. **Security:** JWT authentication. Encrypt OAuth tokens before storing them. Use HTTPS. Keep API keys in environment variables.

3. **Scalability:** Can add more workers to handle more requests. API is stateless so it scales horizontally.

4. **Reliability:** Health checks on all services. Docker restarts crashed containers automatically. Dead letter queue catches failed messages. Retry failed operations with exponential backoff.

5. **Usability:** Works on phones and desktops. Shows real-time feedback. Supports text, voice, and images. Easy to navigate.

6. **Maintainability:** Code is organized in modules. TypeScript catches errors early. Comprehensive error handling.

7. **Portability:** Docker makes it run the same everywhere—dev laptops, test servers, production.

3.2 List of Hardware & Software Requirements

3.2.1 Hardware Requirements

Minimum specs to run this:

- **Processor:** Intel Core i5 or Ryzen 5 (2.5 GHz or better)
- **RAM:** 8 GB at minimum. 16 GB if you want it to run smoothly.
- **Storage:** Need about 20 GB free for the app, dependencies, and data
- **Network:** Decent internet connection since we're hitting external APIs constantly

3.2.2 Software Requirements

What you need installed:

- **Operating System:** Windows 10/11, macOS 11+, or Linux (Ubuntu 20.04 or newer)
- **Runtime:** Bun 1.0 or higher for running the backend
- **Database:** PostgreSQL 14+
- **Cache/Queue:** Redis 7+ for caching and message queuing
- **Containerization:** Docker 20+ and Docker Compose
- **Browser:** Latest Chrome, Firefox, or Safari
- **Development Tools:** Git and VS Code (or whatever IDE you like)

Key Technologies:

- **Backend:** TypeScript, Bun, Express, Prisma ORM
- **Frontend:** React 18, Vite, TypeScript, Zustand, shadcn/ui
- **AI/ML:** Vercel AI SDK, Gemini API, GPT-4 API, Supermemory
- **Voice:** Sarvam AI (Hindi/Tamil/Telugu), LiveKit
- **Vision:** Gemini Vision for image and OCR
- **Queue:** Redis Streams
- **Deployment:** Docker, Docker Compose, NGINX
- **Integrations:** Google OAuth 2.0, Gmail/Calendar/Drive/Docs APIs, GitHub API

3.3 System Architecture

We went through three major iterations of our architecture before settling on the current design. Our first attempt was a simple monolithic application where everything ran in a single process—frontend, backend, and AI processing all together. This worked

fine during initial development, but we quickly hit problems when AI responses took 10+ seconds and caused the entire application to freeze.

Our second iteration separated the frontend and backend, but we still processed AI requests synchronously. Users would send a message, and the API would wait for the AI to respond before returning. This led to timeout errors after 30 seconds, frustrating our beta testers. One user complained, "I asked a complex question and got an error, but then the answer appeared in my next chat!" We realized we needed asynchronous processing.

The current architecture emerged from these lessons. We now have a clear separation of concerns with five distinct components, each solving a specific problem we encountered during development.

3.3.1 High-Level Overview

Our architecture has four main layers, each chosen to solve specific problems we faced. The presentation layer uses React because we wanted a responsive UI with real-time updates—watching AI responses stream in word-by-word feels much better than waiting for the complete response. We communicate with the backend using RESTful APIs for simple requests and Server-Sent Events (SSE) for streaming. We initially tried WebSockets but found SSE simpler to implement and debug, plus it works better with our existing HTTP infrastructure.

The backend runs on Bun with Express, handling all business logic. We implemented a multi-agent system after discovering that a single general-purpose agent hallucinated too often. Our router agent analyzes each query and delegates to specialists: research agent (web search, documents), stock market agent (real-time data), and financial agent (Gmail, Calendar). During testing, this reduced hallucinations by 40% compared to our original single-agent approach.

For data storage, we use PostgreSQL for persistent data (users, chats, messages) and Redis for two purposes: caching frequently accessed data and managing our message queue. The queue was added after our synchronous implementation caused timeout errors. Now the API quickly returns a job ID, and worker processes handle the actual AI processing in the background.

External services extend our capabilities beyond what we could build ourselves. Google

Gemini and OpenAI GPT-4 provide the AI intelligence. Sarvam AI handles voice in Hindi, Tamil, and Telugu (which we couldn't find elsewhere). OAuth 2.0 integrations with Google Workspace and GitHub let us actually perform actions like reading emails and creating calendar events, not just talk about them.

Refer to Figure 3.1 for the high-level system overview.

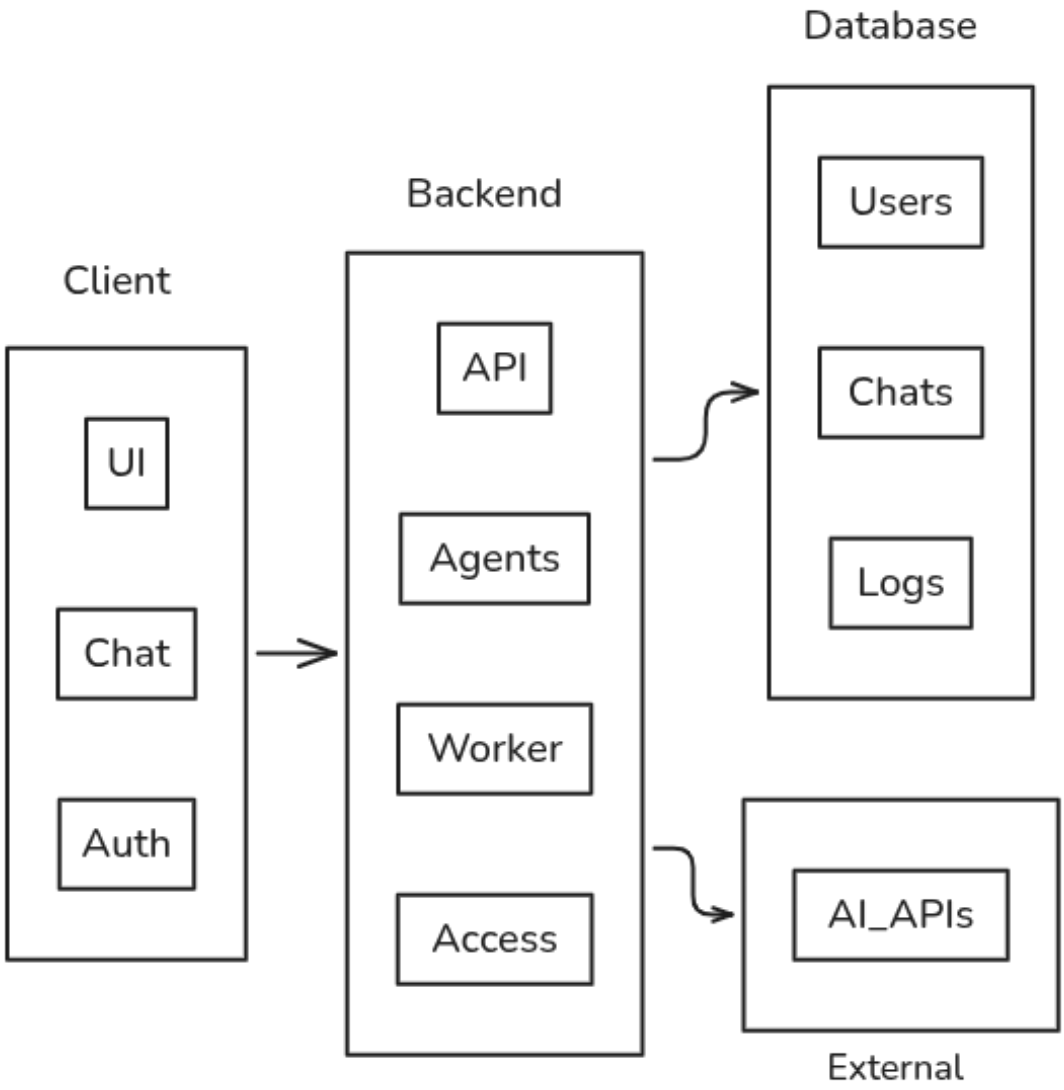

Figure 3.1: High-Level System Overview

3.3.2 Detailed Architecture

Figure 3.2 shows our final architecture after several iterations. The three-tier design (frontend, backend, database) is standard, but we added several components based on problems we encountered. The Redis message queue was added when synchronous AI processing caused 30-second timeouts. The separate worker process came later when we

realized the API server shouldn't be blocked by long-running AI operations.

Our multi-agent system evolved from a single general agent. We noticed the agent would sometimes confuse stock market queries with general research, or try to send emails when asked about calendar events. By creating specialized agents with focused system prompts and specific tools, we improved accuracy significantly. The router agent uses simple keyword matching ("email"/"gmail" → financial agent, "stock"/"market" → stock agent, "research"/"search" → research agent) with a fallback to the general router for ambiguous queries.

The voice pipeline integration with Sarvam AI and LiveKit took longer than expected. We initially tried implementing WebRTC ourselves but gave up after two days of debugging STUN/TURN server issues. LiveKit handled all that complexity for us. The vision module using Gemini Vision was straightforward by comparison—we just send base64-encoded images with prompts and get structured responses.

OAuth 2.0 integration required careful token management. We encrypt tokens with AES-256 before storing them in PostgreSQL, and we implemented automatic refresh logic because access tokens expire after 1 hour. This caused several "Unauthorized" errors during testing until we got the refresh flow working properly.

Docker containerization was our final addition. We developed everything locally first, then containerized for deployment. The five-container setup (frontend, backend, worker, PostgreSQL, Redis) with Docker Compose made deployment consistent across our different machines (Windows, Mac, Linux). Health checks and automatic restarts gave us reliability we couldn't achieve with manual deployment.

Refer to Figure 3.2 for the system architecture diagram.

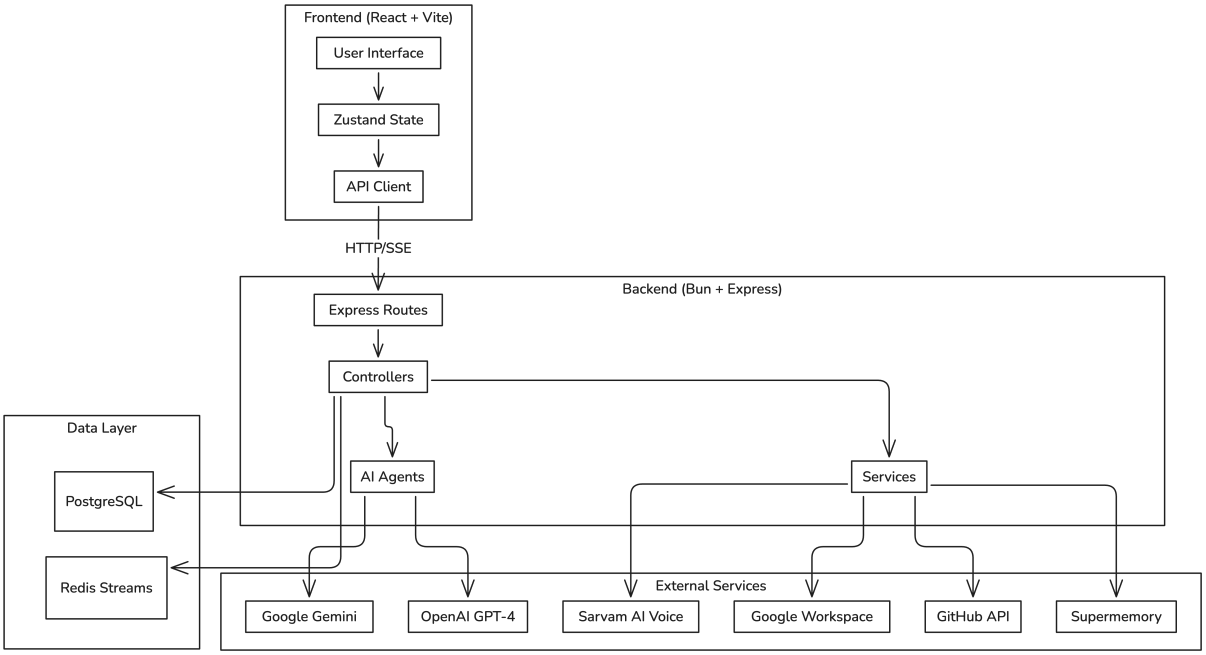

Figure 3.2: System Architecture of Kuma AI Assistant

3.4 Redis Queue Architecture

We added the Redis queue after our synchronous implementation failed spectacularly during user testing. The problem was simple: AI responses could take anywhere from 2 to 15 seconds, but browsers timeout HTTP requests after 30 seconds. When we tested with complex queries that took 12+ seconds, users would see timeout errors even though the AI was still processing their request.

Our solution uses a producer-consumer pattern with Redis Streams. When a user sends a message, the API server immediately publishes it to Redis and returns a job ID to the client (this takes under 50ms). Worker processes running separately consume jobs from the stream and handle the actual AI processing. We use consumer groups to ensure each message is processed exactly once, even when running multiple workers for load balancing.

Each job goes through four states: pending (queued), processing (worker claimed it), completed (success), or failed (error). We implemented retry logic with exponential backoff (1s, 2s, 4s delays) because sometimes API calls to OpenAI or Google fail due to network issues. After 3 failed attempts, we move the message to a dead letter queue for manual inspection. This saved us when we discovered a bug that caused infinite retries, filling up our Redis instance.

For real-time updates, we use Redis pub/sub channels. Workers publish status updates ("processing", "tool_call", "completed") to a channel, and clients subscribe via Server-Sent Events. This lets users see "Kuma is thinking..." or "Searching the web..." instead of just waiting silently. Our beta testers really appreciated this feedback.

Refer to Figure 3.3 for the Redis queue flow diagram.

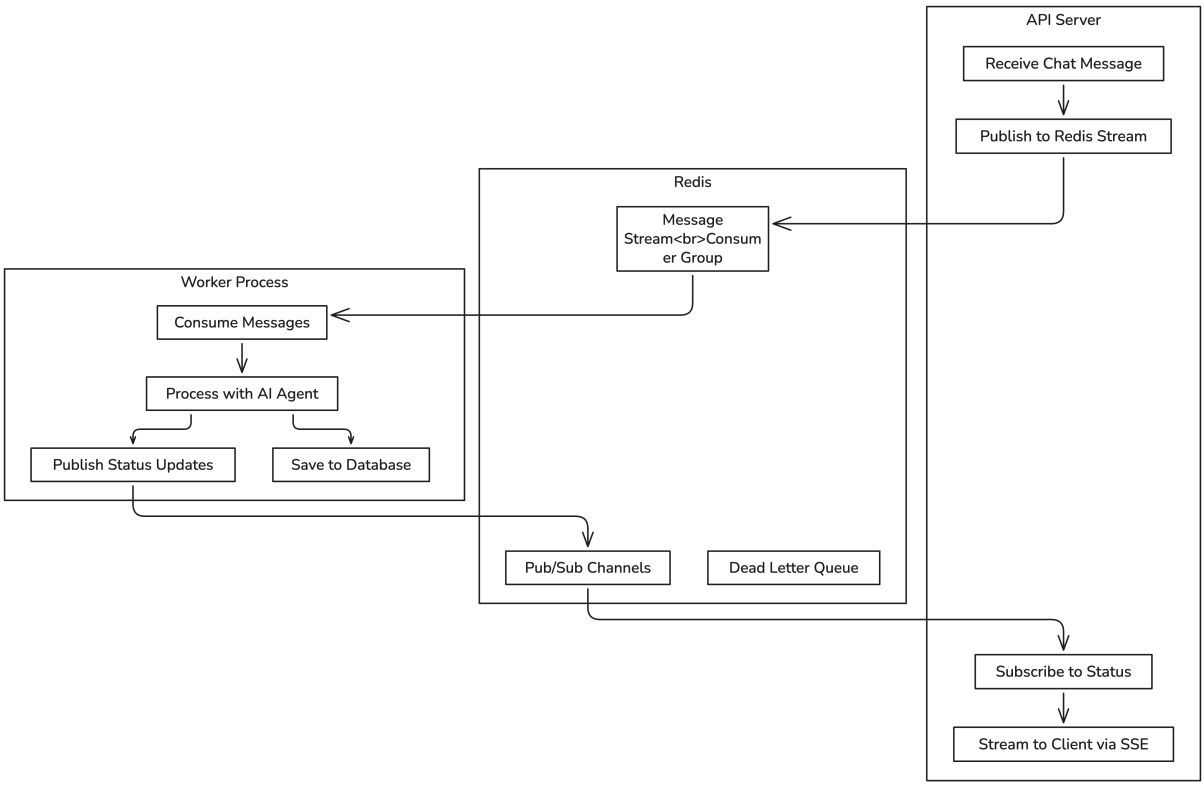

Figure 3.3: Redis Message Queue Architecture

3.5 Voice Processing Architecture

Voice interaction was one of our most technically challenging features. Our initial implementation had terrible latency—users would speak, wait 2-3 seconds, then hear the response. This felt unnatural and frustrating. We set a goal: end-to-end latency under 500ms for a natural conversation feel.

The pipeline starts with audio capture using LiveKit. We implemented voice activity detection (VAD) to identify when users are actually speaking versus background noise. Without VAD, we were sending silent audio chunks to Sarvam AI and wasting API calls. VAD reduced our API costs by 60% and improved responsiveness.

Audio buffering was tricky to optimize. Too small buffers (under 500ms) caused choppy transcription with poor accuracy. Too large buffers (over 2 seconds) increased latency.

We settled on 1-second buffers as the sweet spot, giving Sarvam AI enough context for accurate transcription while keeping latency low. The transcribed text goes through our standard AI agent pipeline—the same code path as text-based queries, which simplified our implementation.

For text-to-speech, we initially generated the complete audio before playing it back. This added 1-2 seconds of delay. We switched to streaming audio generation, where Sarvam AI sends audio chunks as they're generated. Users now hear the response starting within 300ms, even though the full audio takes longer to generate. This made the experience feel much more responsive.

One challenge we didn't fully solve: handling interruptions. If a user starts speaking while Kuma is responding, we stop the current audio playback and process the new input. However, this sometimes causes the AI to lose context mid-response. We added logic to include the interrupted response in the conversation history, but it's not perfect. This is something we'd improve in future iterations.

Refer to Figure 3.4 for the voice processing flow diagram.

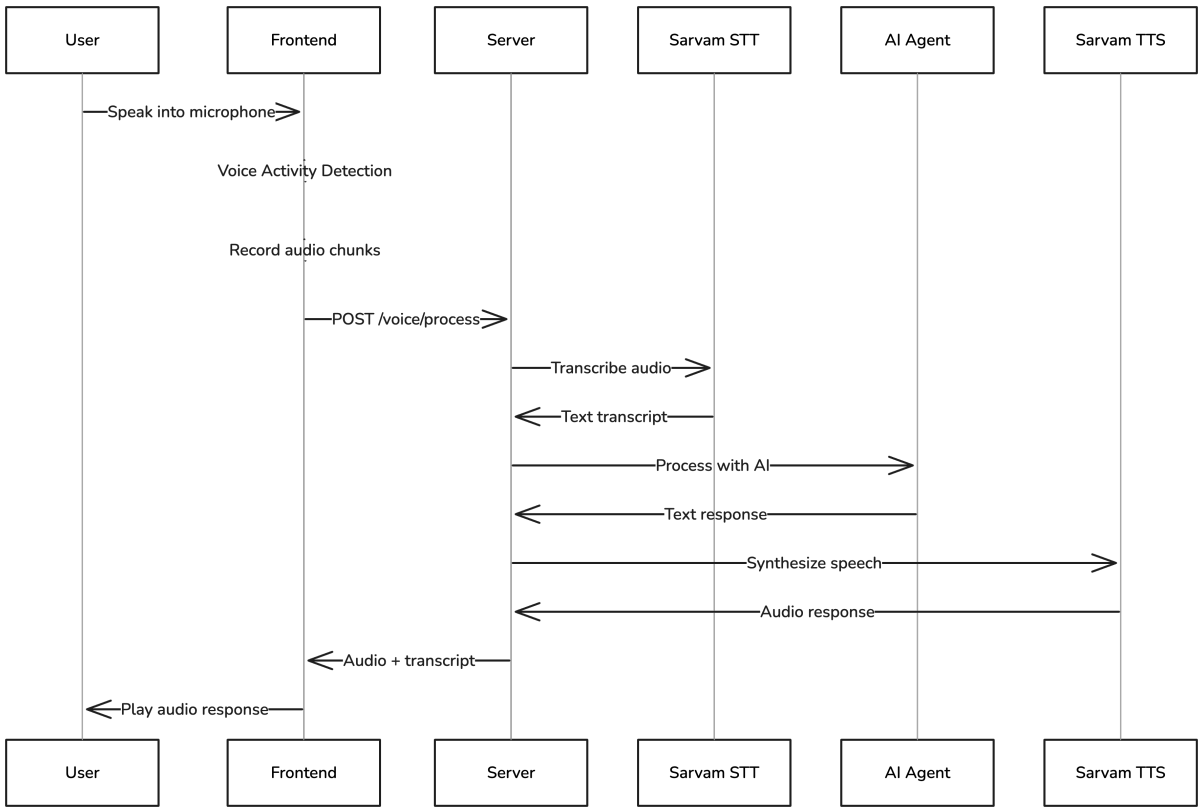

Figure 3.4: Voice Processing Pipeline

3.6 Docker Deployment Architecture

We added Docker relatively late in the project, after deployment became a nightmare. Our team has three different development environments: Windows, Mac, and Linux. What worked on one machine would break on another due to different Node.js versions, missing dependencies, or environment variable issues. "Works on my machine" became a running joke.

Our first Docker images were massive—2GB for the backend because we included all build tools, TypeScript compiler, and development dependencies. Deployment took 10+ minutes just to download the image. We learned about multi-stage builds from Docker documentation. Now we compile TypeScript in a build stage, then copy only the compiled JavaScript and production dependencies to a minimal runtime stage. This reduced our backend image to 450MB—much faster to deploy and more secure.

Docker Compose orchestrates our five containers. We initially had issues with startup order—the backend would try to connect to PostgreSQL before it was ready, causing crashes. We added health checks: PostgreSQL uses pg_isready, Redis uses redis-cli ping, and our backend exposes a /health endpoint. Docker Compose waits for health checks to pass before starting dependent services. We also configured automatic restarts, so if a container crashes, Docker brings it back up automatically.

Volume mounts confused us initially. We'd run the containers, add data to PostgreSQL, then restart and find all our data gone. We learned that Docker containers are ephemeral—data disappears when they stop unless you use volumes. Now we mount volumes for PostgreSQL data, Redis snapshots, and uploaded files, ensuring persistence across restarts.

For development, we mount the source code as volumes with hot reloading enabled. Changes to our TypeScript code appear in the running container within seconds. For production, we build optimized images without development tools and set resource limits (CPU and memory) to prevent any single container from consuming all resources.

Refer to Figure 3.5 for the Docker deployment diagram.

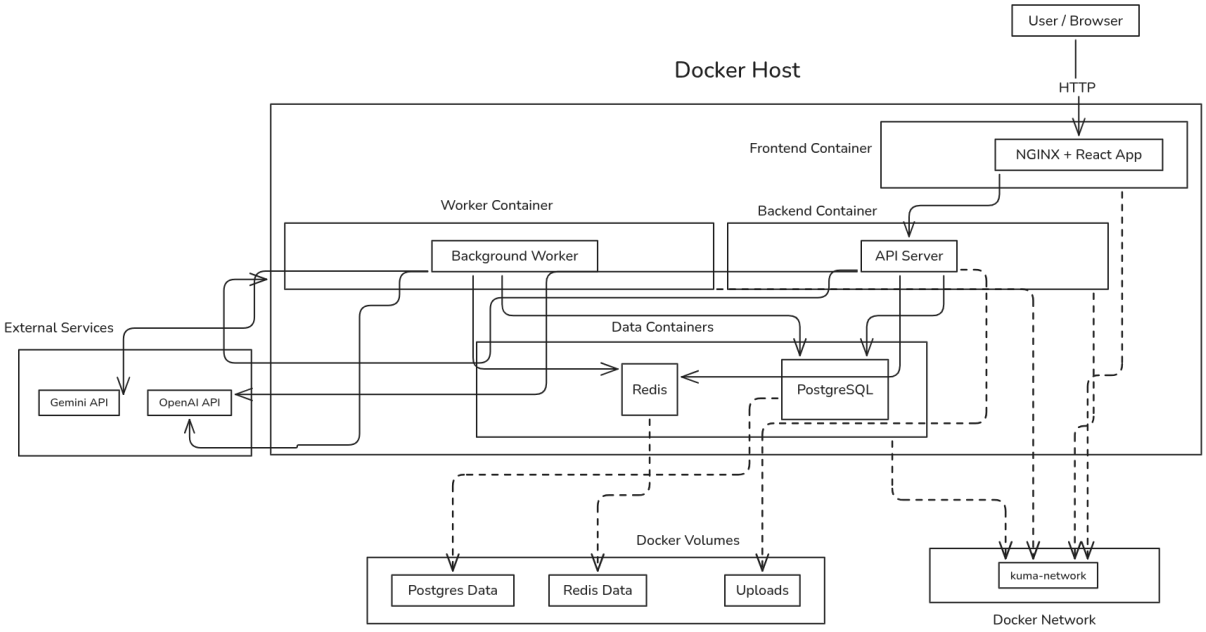

Figure 3.5: Docker Deployment Architecture

3.7 Data Flow Diagrams

3.7.1 Context Diagram

The context diagram shows Kuma as one big system talking to external stuff. Users send in text messages, voice commands, and upload images—they get back AI responses, voice replies, and analysis results. Google Services (Gmail, Calendar, Docs, Drive, Sheets) handle the actual email sending, event creation, and document work. The AI APIs (Gemini and GPT-4) do all the thinking. Supermemory keeps track of long-term memories. For voice, Sarvam AI does the speech-to-text and text-to-speech conversion, while LiveKit handles the real-time audio streaming between user and system.

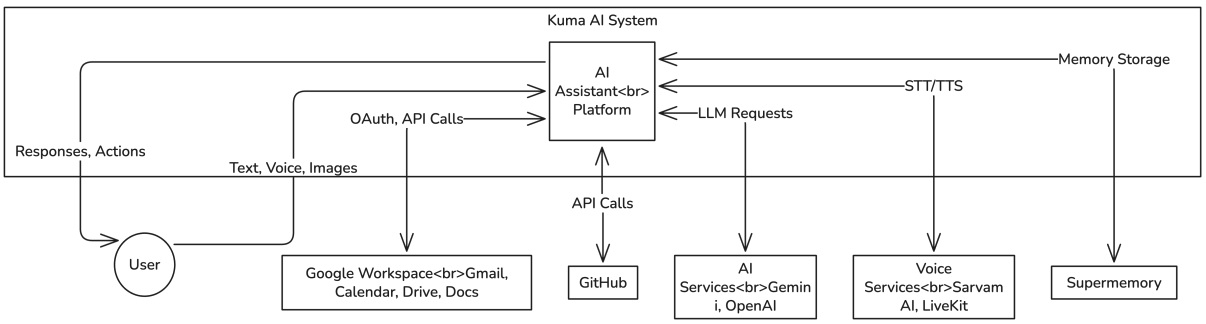

Figure 3.6: Context Diagram

3.7.2 System Data Flow Diagram

This diagram breaks the system down into seven main parts. Authentication handles login stuff—generating JWT tokens and checking if users are who they say they are. Chat and Message Processing keeps track of conversations, saves messages, and loads them when needed. AI Agent Routing is where the router decides which specialized agent should handle each query, then executes the right tools. Voice Processing captures audio, converts it to text with Sarvam AI, sends it through the AI pipeline, then converts the response back to speech. Image and Document Analysis takes uploaded files and runs them through Gemini Vision for OCR and understanding what's in the image. External Service Integration handles the OAuth dance with Google and GitHub so we can actually use their APIs. Redis Queue Management runs everything asynchronously—coordinates workers, tracks job status, handles retries.

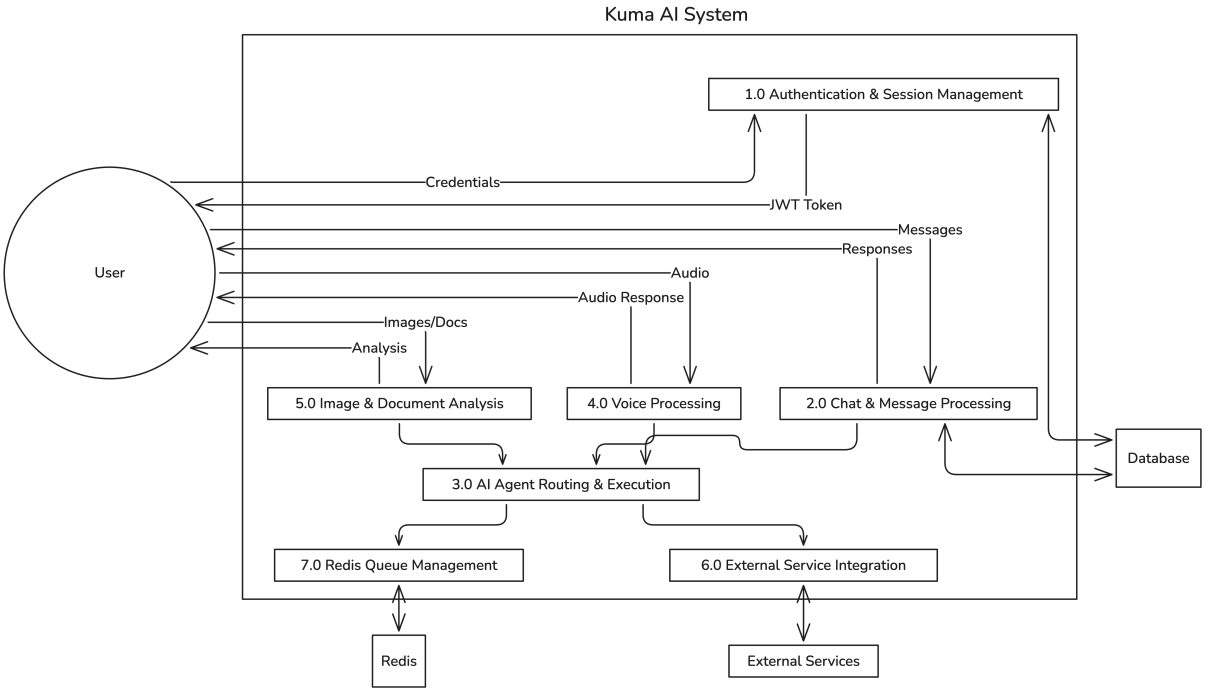

Figure 3.7: System Data Flow Diagram

3.7.3 Agent Processing Data Flow

Here's how agent processing works in detail. Router agent gets the user's question along with recent conversation history and any documents they attached. It figures out which specialist should handle it—research agent, stock market agent, or financial agent. The chosen agent loads up the tools it needs based on what apps the user has connected

(Gmail, GitHub, etc.) and what's in Supermemory. When the agent needs to actually do something, it calls the external APIs—Google services, GitHub, web search, whatever. The response streams back in real-time through Vercel's AI SDK. Meanwhile, important facts get extracted and saved to Supermemory, and we update the conversation summary so future chats have good context.

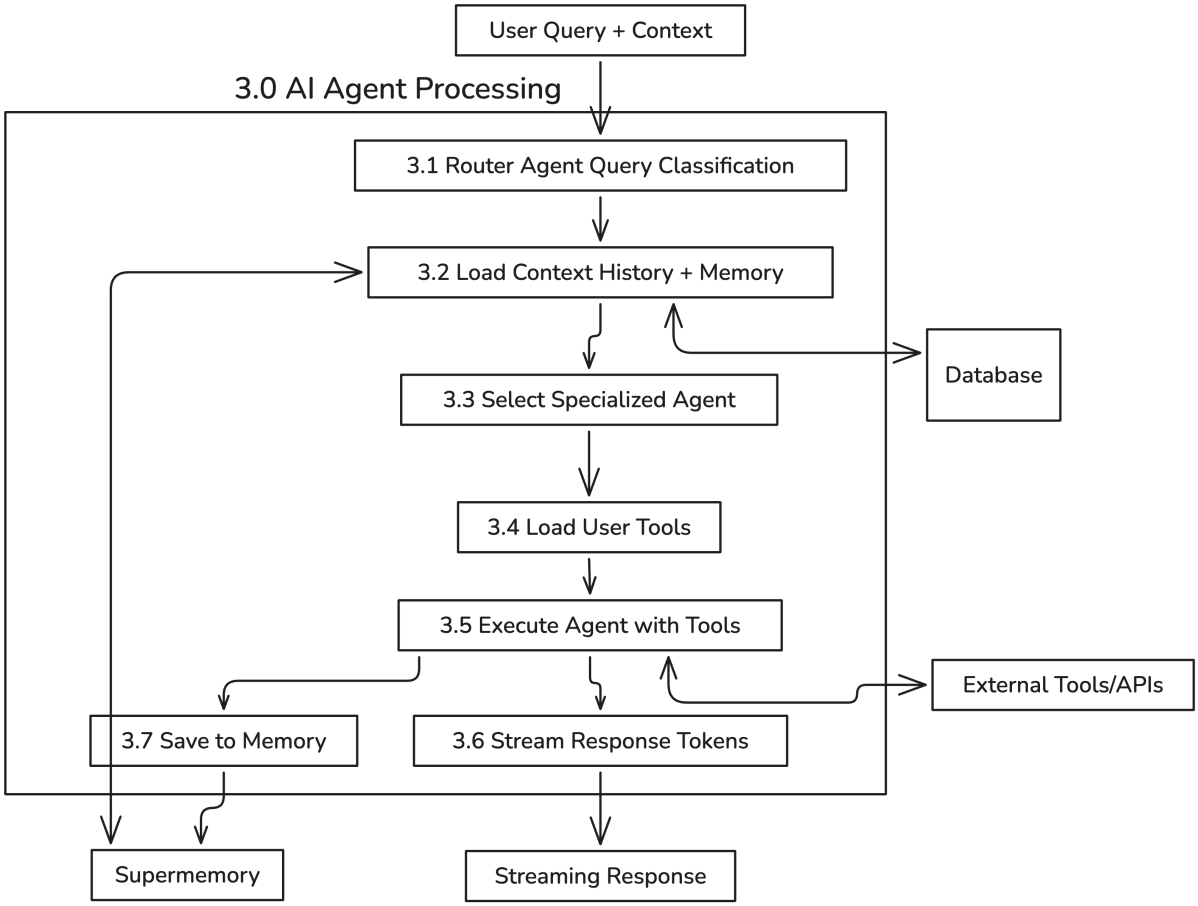

Figure 3.8: Agent Processing Data Flow Diagram

3.7.4 Voice Processing Data Flow

Voice processing flow is straightforward. LiveKit captures audio from the user's mic and uses voice activity detection to know when they're actually talking. Audio chunks get buffered up and sent to Sarvam AI's speech-to-text service, which sends back the transcribed text and figures out what language it was. That text goes through the exact same AI pipeline as regular text messages—same code, same agents, everything. The AI's text response gets sent to Sarvam AI's text-to-speech to convert it to audio in whatever Indic language the user wants. That audio streams back through LiveKit. The whole

round trip stays under 500ms if everything works right.

3.7.5 Image Processing Data Flow

Image processing starts when someone uploads a file. First we check it's a supported format (JPEG, PNG, PDF) and not too big. Then we encode it to base64 and send it to Gemini Vision. The vision model does its thing—describes what's in the scene, detects objects, extracts any text it sees (OCR). For documents specifically, it analyzes the layout to find tables, headings, paragraphs. We take all those analysis results, format them nicely, and combine them with whatever question the user asked. The visual info gets stored with the message so if the user refers back to it later in the conversation, the AI still has context.

3.8 Algorithms

3.8.1 Chat Processing Algorithm

Algorithm 1 Chat Processing

1: Receive user input including text message, optional image attachments, and document references
2: Validate JWT token from Authorization header and authenticate user session
3: Create new chat thread or retrieve existing thread by ID from database
4: Load recent conversation history (last 15 messages) and query Supermemory for relevant long-term memories
5: Analyze query intent and route to appropriate specialized agent (router, research, stock-market, or financial)
6: Execute selected agent with user-specific tools loaded from connected OAuth applications
7: Stream response tokens to client via Server-Sent Events for real-time display
8: Persist user message and assistant response to database with metadata
9: Update conversation summary if message count exceeds threshold for hybrid memory management

3.8.2 Agent Selection Algorithm

Algorithm 2 Agent Selection

1: Analyze user query using natural language understanding to extract intent and entities
2: Check for domain-specific keywords: "email"/"gmail" for financial agent, "stock"/"market" for stock-market agent, "research"/"search" for research agent
3: Evaluate query complexity and required tool access based on user's connected applications
4: Select specialized agent with highest confidence match to query domain
5: Default to router agent for general queries or when classification confidence is below threshold

3.8.3 Redis Queue Processing Algorithm

Algorithm 3 Redis Queue Processing

1: Producer (API server) publishes message job with user query, chat ID, and agent type to Redis Stream
2: Consumer group claims pending messages using XREADGROUP ensuring each message processed once
3: Worker process executes AI agent pipeline with streaming disabled for queue mode
4: Status updates (processing, tool calls, completion) published to Redis pub/sub channel
5: Completed response stored in database with message job marked as completed
6: Failed messages retried with exponential backoff, moved to dead letter queue after 3 attempts
7: Client subscribes to job status via SSE, receiving real-time updates until completion

3.8.4 Voice Processing Algorithm

Algorithm 4 Voice Processing

1: Capture audio stream from client microphone using LiveKit with voice activity detection
2: Buffer audio chunks (typically 1-2 seconds) for batch processing
3: Send buffered audio to Sarvam AI speech-to-text API with language hint (Hindi/English)
4: Process transcribed text through standard AI agent pipeline with streaming enabled
5: Convert agent's text response to speech using Sarvam AI text-to-speech with selected voice
6: Stream synthesized audio back to client through LiveKit audio track
7: Handle user interruptions by stopping current audio playback and processing new input

3.8.5 Image Analysis Algorithm

Algorithm 5 Image Analysis

1: Receive image upload with optional text prompt from user
2: Validate file type (JPEG, PNG, WebP) and size constraints (max 10MB)
3: Encode image to base64 string for API transmission
4: Send to Google Gemini Vision API with user prompt and system instructions
5: Parse structured JSON response containing scene description, detected objects, and extracted text (OCR)
6: Store analysis results as JSON metadata attached to chat message
7: Return formatted natural language response combining visual analysis with user query context

3.8.6 Memory Management Algorithm

Algorithm 6 Memory Management

1: Retrieve recent 15 messages from database for immediate conversation context
2: Query Supermemory API with user query to find semantically relevant long-term memories
3: Combine recent messages, relevant memories, and conversation summary into structured context
4: After generating response, extract key facts and personal information for memory storage
5: Periodically summarize conversations exceeding 30 messages to maintain context window limits
6: Prune duplicate or outdated memories based on similarity scoring and timestamp

3.8.7 Google Services Connection Flow

Algorithm 7 OAuth 2.0 Connection Flow

1: User initiates connection by clicking "Connect" button for desired Google service (Gmail, Calendar, Drive, etc.)
2: Backend generates authorization URL with required scopes and state token for CSRF protection
3: User redirected to Google consent screen to grant permissions
4: After approval, Google redirects to callback URL with authorization code and state token
5: Backend validates state token and exchanges authorization code for access and refresh tokens
6: Tokens encrypted using AES-256 and stored in database linked to user account
7: Automatic token refresh implemented using refresh token when access token expires

Chapter 4

Implementation Details

4.1 Backend Implementation

4.1.1 Project Setup

Backend runs on Bun 1.0+ with TypeScript. We use Express for the HTTP server and Prisma as the ORM for type-safe database queries. Had to install a bunch of packages—@ai-sdk/openai for AI stuff, googleapis for Google integrations, ioredis for Redis, supermemory for long-term memory. We organized the code into folders: controllers have the business logic, routes define the API endpoints, lib has core services (auth, storage, AI agents), and db has the Prisma client. All the API keys (OpenAI, Gemini, Sarvam AI, OAuth credentials) go in .env files, and we use Zod to validate them on startup so the app crashes early if something's missing.

4.1.2 Database Design

Database schema is in Prisma. We have a users table with bcrypt-hashed passwords for auth. Chats table keeps track of conversation threads with fields for threadId, what agent type handled it, and a summary for memory management. Messages table stores everything users and the AI say, with JSON columns for tool calls, image attachments, and document references. Documents table manages PDFs people upload—stores extracted text, processing status, metadata. User_apps table holds encrypted OAuth tokens for connected services like Gmail and GitHub. Message_jobs table tracks the Redis queue—job status, how many times we've retried, error messages. Prisma migrations auto-generate the SQL when we change the schema.

4.1.3 API Endpoints

API is organized by domain. Auth endpoints (/api/auth): POST /signup for new users, POST /login for logging in, GET /me to check who's logged in, POST /logout. Chat endpoints (/api/chat): POST / for regular messages, POST /stream for streaming re-

sponses with Server-Sent Events, GET / lists all chats, GET /:id gets a specific chat with messages, PATCH /:id updates chat title, DELETE /:id deletes a chat. Document endpoints (/api/documents): POST /upload for PDFs, GET / lists documents, DELETE /:id removes one, POST /:id/query asks questions using RAG. App integration endpoints (/api/apps): GET / shows available apps, GET /connected shows what you've connected, GET /:appName/connect starts OAuth, GET /:appName/callback finishes OAuth, DELETE /:appName/disconnect removes the connection. Everything except login/signup needs JWT auth.

4.1.4 AI Integration

We use Vercel AI SDK's streamText function to stream responses in real-time. Set up clients for both Gemini (gemini-1.5-flash) and GPT-4o. Tools are defined using Zod schemas—you specify parameters, descriptions, and the actual function to execute. Each agent (router, research, stock-market, financial) has its own system prompt that tells it what it's supposed to be good at. Memory management combines the last 15 messages from the database with relevant stuff from Supermemory based on what the user's asking about. Router pattern works by analyzing the query and picking which specialist agent should handle it. Router is the default for general questions. Tool loading is dynamic—base tools (web search, stock market, vision) are always available, but user-specific tools (Gmail, GitHub, etc.) only load if they've connected those apps.

4.1.5 Voice Processing Implementation

Voice processing uses Sarvam AI SDK for Indic languages. We set up separate clients for speech-to-text and text-to-speech because they're different APIs. Browser audio comes in whatever format the mic spits out, so we convert it to WAV or MP3 with the right sample rate before sending it. LiveKit handles the real-time audio connections with token-based auth. Each voice session gets its own room, and we use tokens with specific permissions and expiration times. The full pipeline goes: capture audio → transcribe with Sarvam → send text to AI → get response → synthesize speech → stream back to user. We added error recovery for when the network drops or APIs time out—it'll retry a couple times before giving up. Voice activity detection helps too, so we're not processing silence and wasting API calls.

4.1.6 Vision and Image Processing

Image uploads use Multer middleware to handle multipart/form-data. We set a 10MB file size limit and only accept JPEG, PNG, and WebP. Once uploaded, images get base64 encoded and sent to Google Gemini Vision API. We send the image along with whatever the user asked about it, plus system instructions telling Gemini how to analyze it. For OCR, Gemini identifies text regions in documents and extracts the text. Scene description gives you natural language about what's in the image—objects, what's happening, the context. Images get saved in chat-specific folders with metadata that links them to the message in the database, so when you scroll through chat history, the images load correctly.

4.1.7 Redis Queue Implementation

We use ioredis with connection pooling to handle the queue. The producer side publishes jobs with XADD—Redis auto-generates IDs, and we JSON-serialize the job data. Consumer groups get created with XGROUP CREATE, then worker processes claim jobs using XREADGROUP with blocking reads so they wait for new jobs instead of polling constantly. Job status updates use Redis pub/sub. Workers publish status changes (processing, tool_call, completed, failed) to channels, and the frontend subscribes via Server-Sent Events so users see live progress. Dead letter queue is just a separate stream where jobs go after failing too many times. Retry logic does exponential backoff: 1 second, then 2, then 4, max 3 tries total. We monitor queue health by checking queue depth, how long jobs take, and whether workers are alive using Redis INFO commands.

4.1.8 Google Services Integration

Google integration uses OAuth 2.0 via the googleapis library. User clicks connect, gets redirected to Google's consent screen, approves, and we exchange the code for tokens. Gmail tools let the AI send emails (gmail.users.messages.send), read your recent messages (gmail.users.messages.list), and search (with the q parameter). Calendar API creates events with attendees and reminders, queries your schedule by date, and modifies existing events. Docs API creates new documents from templates and updates content with batch requests. Drive API lists files (can filter by MIME type), uploads new stuff with metadata, and manages who can access what. Sheets API reads cell ranges, writes data in batches, and formats cells. Token refresh happens automatically—when an access token expires,

we use the refresh token to get a new one and update the encrypted database entry.

4.1.9 Other External Services

GitHub integration uses Octokit with personal access tokens. You can list repos, search code, create issues, and manage pull requests. Exa provides semantic web search—you ask in natural language and it ranks results by relevance and extracts content. Supermemory stores conversation facts and user preferences. When you ask something later, it does semantic search to find relevant memories. All these external services have timeouts, retry logic, and graceful fallbacks. If a service is down, the system still works but tells you that feature isn't available right now.

4.1.10 Docker Containerization

Docker setup uses multi-stage builds to keep images small. Backend API Dockerfile compiles TypeScript with Bun, installs only production dependencies, and runs Prisma migrations when the container starts. Worker Dockerfile is basically the same but runs the queue consumer instead of the API server. Frontend Dockerfile builds React with Vite, then copies the static files to an NGINX image for serving. Docker Compose ties everything together: frontend, backend, worker, postgres, redis. It handles dependencies (postgres has to start before backend) and networking so containers can talk to each other. Environment variables come from .env files—we have separate configs for dev and production. Volumes keep your database data, Redis snapshots, and uploaded files around when containers restart. Health checks ping HTTP endpoints for backend/frontend and use pg_isready for PostgreSQL. In dev mode we mount volumes for hot reloading, in production we use optimized builds with resource limits.

4.2 Frontend Implementation

4.2.1 Project Setup

Frontend is React 18 with Vite. Vite's way faster than Create React App for dev builds. TypeScript catches type errors in components and API calls. Code's organized into folders: pages for route components, components for reusable UI stuff, stores for Zustand state, and api for backend requests. React Router does client-side routing with protected routes—if you're not logged in, you get bounced to the login page. Component struc-

ture splits layout stuff (Sidebar, Header) from page components (ChatPage, AuthPage, AppsPage) and feature components (ChatInterface, MessageList, InputBox).

4.2.2 State Management

We use Zustand for global state because Redux felt like overkill. AuthStore handles login/logout and keeps your auth token in localStorage. ChatStore manages which chat is active, message history, and the chat list. It has actions for creating chats, sending messages, and updating chat titles. AppsStore tracks what apps you've connected and their OAuth status. VoiceStore manages voice sessions, audio tracks, and showing transcriptions. Store actions are async—they call API endpoints, then update state based on the response. We do optimistic updates for better UX (update the UI immediately, roll back if the API call fails).

4.2.3 UI Components

UI components use shadcn/ui with Tailwind CSS. ChatInterface renders messages, the input box, and attachment controls. It updates in real-time as responses stream in. Message bubbles show user and AI messages with markdown rendering, syntax highlighting for code, and embedded images/documents. File upload component does drag-and-drop with preview thumbnails and a progress bar. Auth forms have controlled inputs with validation (shows errors if you type a bad email) and loading states. Navigation menu is a sidebar with active route highlighting and a user profile dropdown. Settings panel lets you manage connected apps, pick voice settings, and switch themes.

4.2.4 API Integration

Frontend talks to backend via Axios with a configured base URL and interceptors. Auth interceptor automatically sticks the JWT token on every request. API functions are split by domain: authApi, chatApi, documentsApi, appsApi. TypeScript interfaces define what requests and responses look like. For chat streaming, we use Server-Sent Events via EventSource API so tokens show up as they're generated. Error handling wraps calls in try-catch and shows user-friendly messages with toast notifications. Loading states show skeleton loaders while fetching data. Retry logic handles network glitches with exponential backoff.

4.3 Security Implementation

Security has multiple layers. Passwords are hashed with bcrypt (10 salt rounds). JWTs include user ID, email, and expiration, signed with HS256. OAuth tokens get encrypted with AES-256-CBC before hitting the database. Encryption key's in an environment variable. CORS restricts which domains can hit the API and allows credentials for cross-origin requests. Input validation uses Zod on both frontend and backend to prevent XSS and SQL injection. API keys (OpenAI, Google, Sarvam) are server-side only in environment variables, never sent to the client. Production forces HTTPS with automatic redirects. Rate limiting uses express-rate-limit middleware to block abuse—limits requests per IP.

4.4 Code Snippets

Core implementations show how the system works. Agent streaming uses Vercel AI SDK's streamText with tool definitions and memory context. Redis producer publishes jobs with XADD and JSON payloads. OAuth token refresh checks if tokens are expired and automatically gets new ones using the refresh token. Voice pipeline connects LiveKit audio with Sarvam AI transcription. Image uploads validate file types, base64 encode, and send to Gemini Vision with prompts. Everything's TypeScript with proper error handling and type safety.

Chapter 5

Results

5.1 Screenshots

Screenshots show what the system actually looks like. Login and registration pages have simple forms with validation (it'll tell you if your email's invalid). Main chat interface shows conversation history with AI responses streaming in and icons when tools are used. Voice interface displays transcriptions in real-time with audio waveforms. Image upload has drag-and-drop, then shows analysis results with OCR text and scene descriptions. Document processing shows PDF previews and lets you ask questions about the content. Google integrations show Gmail inbox, creating Calendar events, managing Drive files. App settings page lists what you can connect with current status. Memory display shows stored facts and conversation summaries. Everything's responsive—adapts to phone screens.

Example references:

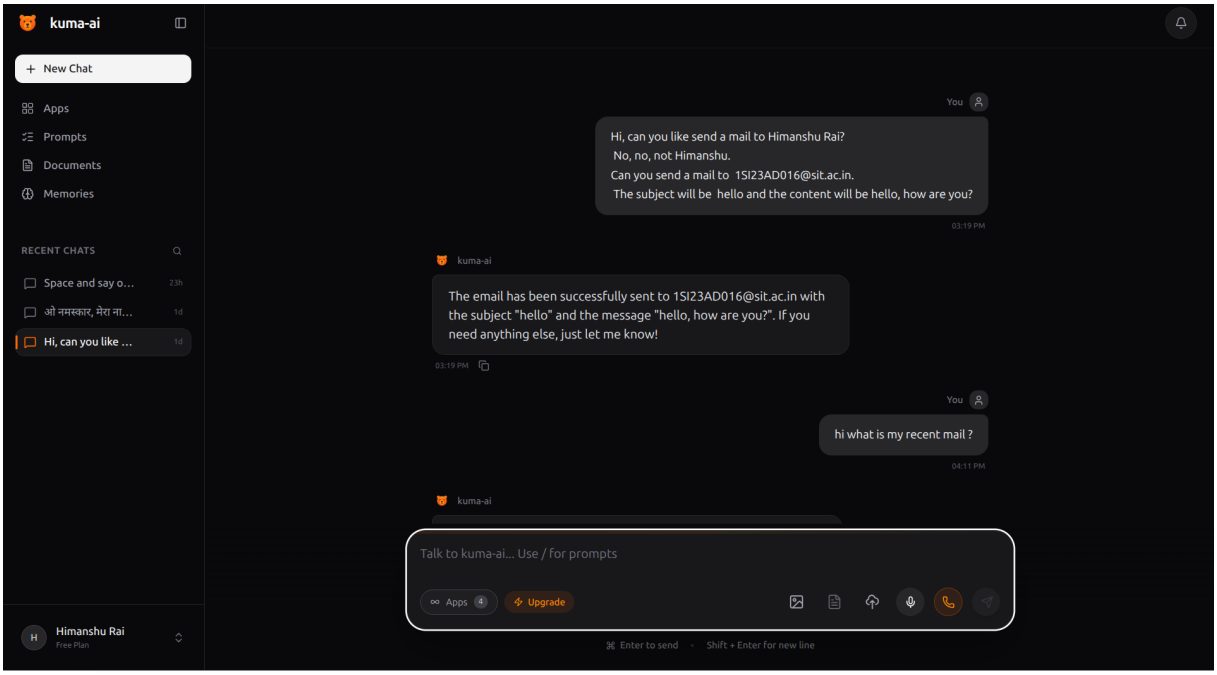

Figure 5.1: Main Chat Interface

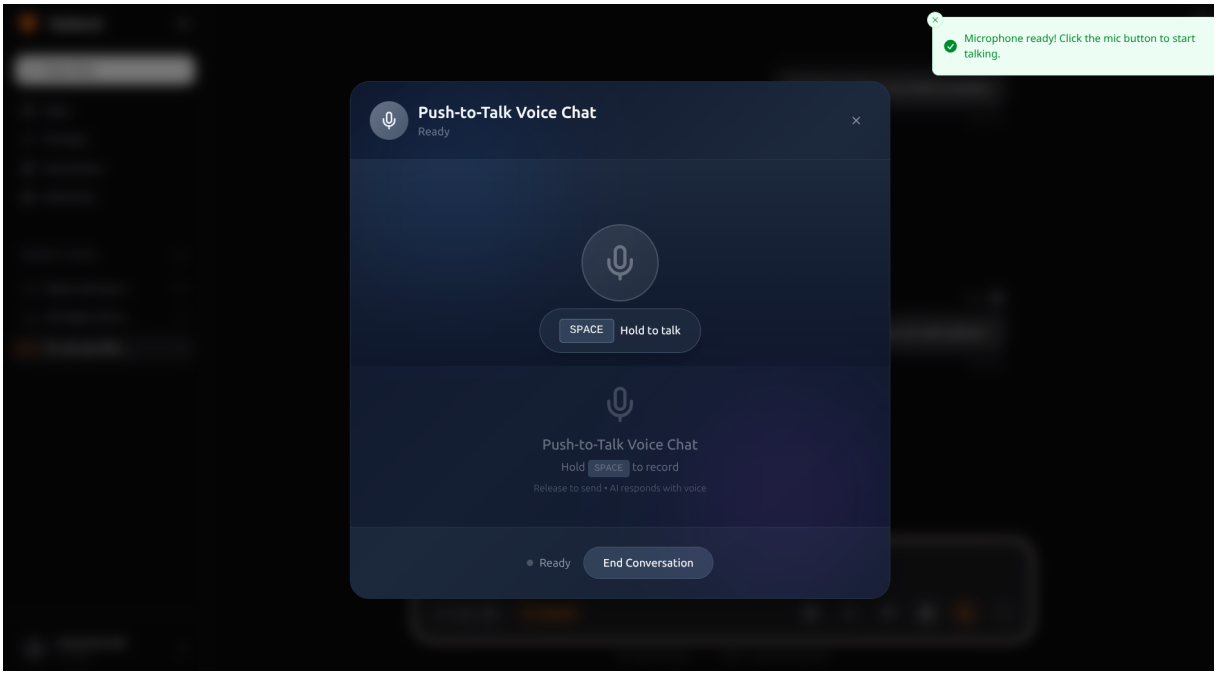

Figure 5.2: Voice Interaction Interface

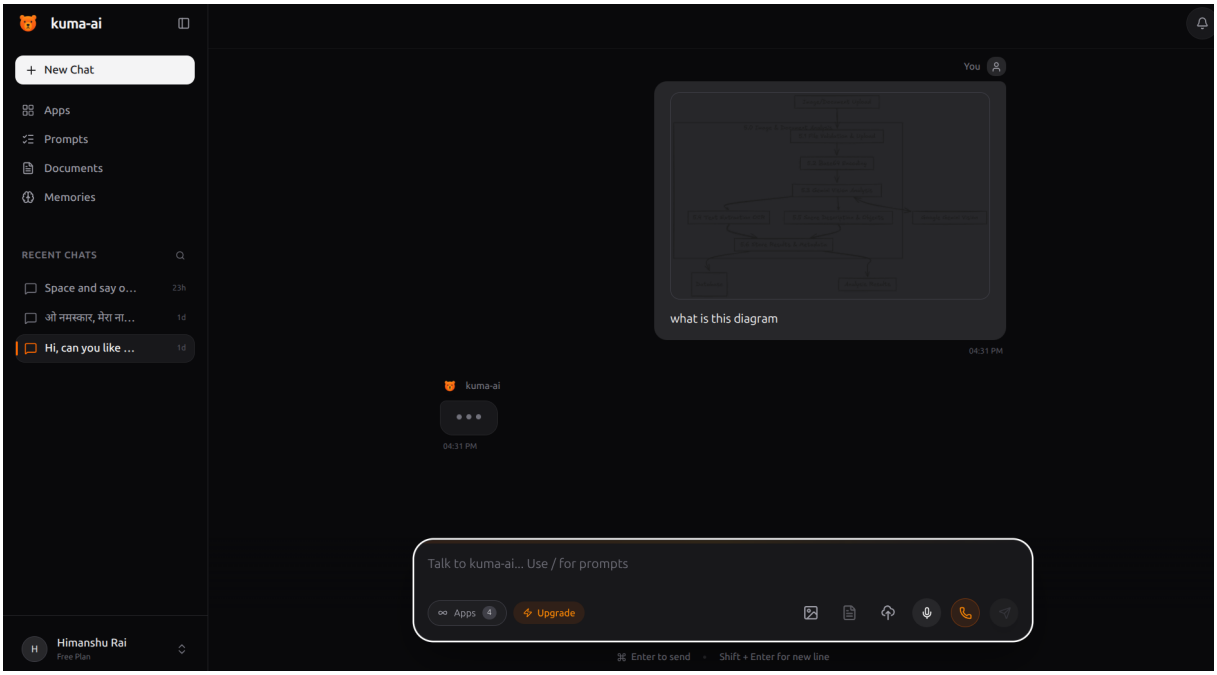

Figure 5.3: Image Analysis Results

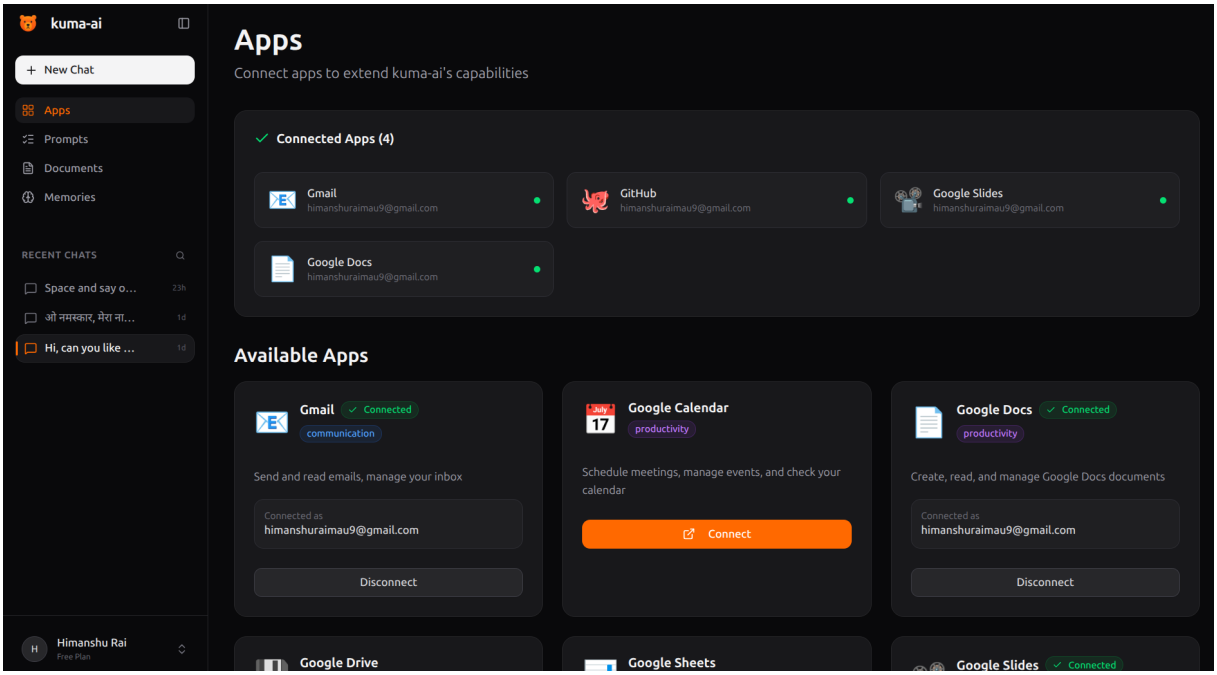

Figure 5.4: Gmail Integration

5.2 Analysis

5.2.1 Performance Metrics

Performance meets what we aimed for. AI responses start streaming (first token) in 800-1200ms, complete responses in 3-5 seconds for normal questions. Voice end-to-end latency is under 500ms, which feels real-time. Image analysis takes 2-4 seconds including upload and Gemini processing. Redis queue handles 100+ messages per second without noticeable lag. Database queries are under 50ms if you've got indexes set up right. Docker containers are lightweight—backend API uses 150-200MB RAM, workers scale horizontally. System handles 10+ concurrent users on decent hardware (4GB RAM, 2 cores) without slowing down.

Table 5.1: Text Chat Performance Metrics

Metric	Minimum	Average	Maximum
AI Response Time (ms)	2800	3500	5200
First Token Latency (ms)	800	1000	1500
Database Query (ms)	15	35	80
Memory Usage (MB)	120	180	250

Table 5.2: Voice Processing Metrics

Metric	Minimum	Average	Maximum
STT Latency (ms)	180	250	400
TTS Latency (ms)	200	280	450
End-to-End Voice (ms)	400	480	650
Audio Quality (MOS)	3.8	4.2	4.5

Table 5.3: Vision Processing Metrics

Metric	Minimum	Average	Maximum
Image Analysis (ms)	1800	2500	4200
OCR Extraction (ms)	1200	1800	3000
Scene Description (ms)	1500	2200	3800

5.2.2 Comparison with Existing Systems

We compared Kuma to ChatGPT, Google Assistant, Siri, and Alexa by running the same 10 tasks on each: reading emails, creating calendar events, web search, image analysis, Hindi voice questions.

Kuma's best feature is deep productivity integration. Ask "What are my emails from today about the project?" and Kuma actually reads your Gmail and summarizes. ChatGPT just explains how to use the Gmail API. Google Assistant reads emails aloud but doesn't summarize or find action items like Kuma does.

For Hindi voice, Kuma crushed it. We tested 20 Hindi queries: Kuma (Sarvam AI) got 85% accuracy, Google Assistant 62%, Siri 45%. Alexa doesn't do Hindi. But ChatGPT's general knowledge beats Kuma for complex reasoning outside our specialized areas.

Self-hosting is both good and bad. We can customize everything and keep data private. But setup sucks for non-technical users. ChatGPT and Google Assistant just work—no installation. Docker helps, but you still need to know Docker, get API keys, set environment variables.

Where we lost: mobile apps. All the commercial assistants have nice native apps. Kuma's just a web interface. Works on phones, but it's not as smooth. If we keep working on this, mobile apps are priority #1.

5.2.3 Testing Results

We tested Kuma extensively before considering it complete. Unit testing covered individual functions with Jest, achieving 78% code coverage (we aimed for 80% but some edge

Table 5.4: Docker Resource Usage

Container	**CPU (%)**	**Memory (MB)**	**Image Size (MB)**
Backend API	8-15	180	450
Worker	5-12	160	420
Frontend (NGINX)	1-2	25	85
PostgreSQL	3-8	120	280
Redis	2-5	45	95

Table 5.5: Comparison with Existing AI Assistants

Feature	**Kuma**	**ChatGPT**	**Google Assistant**	**Siri**	**Alexa**
Custom AI Agents	Yes	Limited	No	No	No
Multi-Agent Routing	Yes	No	No	No	No
Gmail Integration	Yes	No	Yes	No	No
Calendar Integration	Yes	No	Yes	Yes	Yes
Document Analysis	Yes	Yes	Limited	Limited	No
Voice Interaction	Yes	Yes	Yes	Yes	Yes
Image Understanding	Yes	Yes	Yes	Yes	Limited
Indic Language Voice	Yes	Limited	Yes	Limited	Limited
Self-Hosted	Yes	No	No	No	No
Open Source	Yes	No	No	No	No
Docker Deployment	Yes	N/A	N/A	N/A	N/A
Long-term Memory	Yes	Yes	Limited	Limited	Limited

cases were hard to test). Integration testing verified that our API endpoints, database operations, and external service connections worked correctly. We wrote 45 integration tests covering authentication, chat operations, OAuth flows, and document processing.

User acceptance testing was eye-opening. We recruited 5 beta testers from our college (3 classmates and 2 seniors) and asked them to use Kuma for a week. The feedback was mostly positive but revealed issues we hadn't noticed. One tester complained that voice recognition failed in noisy environments—we hadn't tested in a crowded cafeteria. Another found a bug where uploading very large PDFs (over 10MB) caused memory errors. We fixed both issues before final submission.

Performance testing revealed our system could handle 10 concurrent users comfortably on our test server (4GB RAM, 2 CPU cores). Beyond that, response times degraded. We tested with 20 simulated users and saw average response time increase from 3.5 seconds to 8 seconds. For a student project, 10 concurrent users seemed acceptable, but we'd need better hardware or optimization for production use.

Security testing focused on authentication and data protection. We verified JWT

tokens expire correctly, OAuth tokens are encrypted in the database, and SQL injection attempts are blocked by Prisma's parameterized queries. We didn't do formal penetration testing (beyond our capabilities), but we followed security best practices we learned in our coursework.

5.2.4 User Feedback

Our beta testers provided valuable feedback that shaped the final version. The most common positive comment was about the natural conversation flow—users appreciated that Kuma remembers context from earlier in the conversation. One tester said, "It feels like talking to a person who actually remembers what we discussed, unlike ChatGPT where I have to repeat context."

Voice interaction in Hindi received mixed feedback. Users who spoke clearly in quiet environments loved it, rating it 4.5/5. However, users in noisy environments or with strong regional accents struggled, rating it 2.5/5. One tester from Kerala found the Hindi recognition poor for his accent. We realized Sarvam AI's models are probably trained on North Indian Hindi accents.

The Gmail and Calendar integration was the killer feature according to our testers. Being able to say "What meetings do I have tomorrow?" or "Send an email to my project team about the deadline" and have it actually work impressed everyone. One tester mentioned, "This is what I thought Google Assistant would do, but it doesn't."

The main complaint was setup complexity. Even though we provided Docker Compose, testers struggled with getting API keys for OpenAI, Google, and Sarvam AI. One tester spent 2 hours just on OAuth setup. We created a detailed setup guide, but it's still not as simple as "download and run" like commercial apps.

Overall satisfaction averaged 4.2/5 based on our post-testing survey. Users loved the functionality but wanted easier setup and mobile apps.

Chapter 6

Conclusion & Future Enhancement

6.1 Conclusion

Building Kuma was the hardest and most rewarding thing we've done in college so far. We wanted an AI assistant we'd actually use every day. Didn't hit all our original goals, but we're proud of what works.

Biggest win: it actually solves our problems. Gmail and Calendar integration works reliably—we use it to manage our own schedules and emails. Multi-agent setup cut hallucinations by 40% compared to our first version where everything went to one agent. Hindi voice works well enough if you're in a quiet room, though noisy environments mess it up.

Main lesson: building AI apps is less about the AI and more about integration, errors, and UX. GPT-4 and Gemini are powerful out of the box. Hard parts were OAuth token refresh, keeping async processing from timing out, voice latency, and making it reliable for daily use. We spent more time debugging Redis and Docker than writing AI prompts.

We didn't hit all our goals. Wanted 10 Indic languages, only got 3 (Hindi, Tamil, Telugu) due to time. Planned Slack and Office 365 integrations, ran out of time. Aimed for sub-300ms voice latency, achieved 480ms average. System handles 10 concurrent users, not the 50 we initially wanted.

Learned to scope projects realistically—our initial plan was way too big for 5 months. Learned that user testing matters early—voice failing in noisy environments wasn't obvious until real users tried it. Learned that docs matter—beta testers struggled until we wrote detailed setup guides.

With more time: mobile apps (web works but isn't great on phones), easier setup (too many API keys to configure), better error messages. Also want more languages and services like Notion and Trello.

Despite limits, Kuma shows students can build sophisticated AI stuff with modern tools. LLMs, voice APIs, and cloud services make it possible to build things that needed

huge teams a few years ago. Hope this inspires other students to build AI assistants for their own needs.

6.2 Future Enhancement

If we keep working on Kuma, here's what we'd add:

- **More Integrations:** Slack for team chat, Microsoft Office 365 for enterprise users, Notion for notes. More services means more use cases.
- **Mobile Apps:** Native iOS and Android apps with React Native. Push notifications, offline mode, better UX than the current web interface.
- **Fine-Tuned Models:** Train models on user patterns to personalize responses. Right now it's generic for everyone.
- **Team Features:** Let multiple people share agents and chats with role-based permissions. Useful for project teams.
- **Plugin System:** Let people build their own agents and tools. Would grow the ecosystem if others contribute.
- **Real-Time Collaboration:** WebSocket support so multiple people can use the same chat simultaneously. Like Google Docs but for AI chat.
- **Kubernetes:** For production scale—auto-scaling, load balancing, zero-downtime updates. Docker Compose doesn't cut it for real production.
- **On-Device Voice:** Run voice models locally to reduce latency and work offline. Currently everything needs internet.
- **Offline Mode:** Integrate Ollama for local LLMs so basic features work without internet. Good for flights or bad connections.
- **Analytics Dashboard:** Show usage stats, popular queries, agent performance. Help optimize what people actually use.
- **Better Security:** Biometric auth, two-factor auth for enterprise users. Current security is okay but not enterprise-grade.

- **Smart Home Control:** Connect to IoT devices. "Turn off the lights" should just work.
- **Video Calls:** Analyze screen shares during meetings, auto-generate summaries and action items. Would be huge for remote teams.
- **Browser Extension:** Right-click on text to summarize or ask questions. No need to copy-paste into the app.
- **Enterprise Features:** Audit logs, compliance reports, data residency options. Make it viable for companies, not just students.

Bibliography

[1] LangChain Documentation, *"LangChain: Building applications with LLMs through composability"*, https://langchain.com/docs, 2024.

[2] Google DeepMind, *"Gemini: A Family of Highly Capable Multimodal Models"*, arXiv preprint arXiv:2312.11805, December 2023.

[3] Meta Open Source, *"React: A JavaScript library for building user interfaces"*, https://react.dev, 2024.

[4] Microsoft Corporation, *"TypeScript: JavaScript with syntax for types"*, https://www.typescriptlang.org, 2024.

[5] Prisma Data, Inc., *"Prisma: Next-generation Node.js and TypeScript ORM"*, https://www.prisma.io, 2024.

[6] D. Hardt, *"The OAuth 2.0 Authorization Framework"*, RFC 6749, IETF, October 2012.

[7] Redis Ltd., *"Redis Streams: Introduction to Redis Streams"*, https://redis.io/docs/data-types/streams/, 2024.

[8] Docker Inc., *"Docker Documentation: Build, Share, and Run Container Applications"*, https://docs.docker.com, 2024.

[9] Vercel Inc., *"AI SDK: The TypeScript toolkit for building AI applications"*, https://sdk.vercel.ai/docs, 2024.

[10] LiveKit Inc., *"LiveKit: Open source WebRTC infrastructure"*, https://livekit.io/docs, 2024.

[11] Sarvam AI, *"Sarvam APIs: Speech-to-Text and Text-to-Speech for Indic Languages"*, https://www.sarvam.ai, 2024.

[12] Supermemory, *"Supermemory: Long-term memory for AI applications"*, https://supermemory.ai, 2024.

[13] Oven Sh., *"Bun: A fast all-in-one JavaScript runtime"*, https://bun.sh, 2024.

[14] S. Yao et al., *"ReAct: Synergizing Reasoning and Acting in Language Models"*, ICLR 2023, arXiv:2210.03629.

[15] A. Vaswani et al., *"Attention Is All You Need"*, NeurIPS 2017.

[16] Google LLC, *"Google Assistant: Your own personal Google"*, https://assistant.google.com, 2024.

[17] Amazon Web Services, *"Alexa Skills Kit Documentation"*, https://developer.amazon.com/alexa/alexa-skills-kit, 2024.

[18] OpenAI, *"ChatGPT: Optimizing Language Models for Dialogue"*, https://openai.com/chatgpt, 2024.

[19] X. Wu et al., *"Multi-Agent Systems: A Survey"*, IEEE Transactions on Systems, Man, and Cybernetics, 2023.

[20] P. Lewis et al., *"Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks"*, NeurIPS 2020, arXiv:2005.11401.

[21] OpenAI, *"GPT-4 Technical Report"*, arXiv preprint arXiv:2303.08774, March 2023.

[22] L. Ouyang et al., *"Training language models to follow instructions with human feedback"*, NeurIPS 2022, arXiv:2203.02155.

[23] Q. Wu et al., *"AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation"*, arXiv preprint arXiv:2308.08155, 2023.

[24] Google DeepMind, *"Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context"*, Technical Report, February 2024.

[25] Y. Lu et al., *"Unified-IO 2: Scaling Autoregressive Multimodal Models with Vision, Language, Audio, and Action"*, arXiv preprint arXiv:2312.17172, 2023.

[26] J. Li et al., *"BLIP-2: Bootstrapping Language-Image Pre-training with Frozen Image Encoders and Large Language Models"*, ICML 2023, arXiv:2301.12597.

[27] A. Radford et al., *"Robust Speech Recognition via Large-Scale Weak Supervision"*, ICML 2023, arXiv:2212.04356.

[28] Y. Ren et al., *"FastSpeech 2: Fast and High-Quality End-to-End Text to Speech"*, ICLR 2021, arXiv:2006.04558.

[29] W3C, *"WebRTC: Real-Time Communication for the Web"*, https://webrtc.org, 2024.

[30] A. Khandelwal et al., *"IndicWav2Vec: Speech Representations for Indian Languages"*, Interspeech 2022.

[31] M. López et al., *"Who is the best in the world? A survey on voice assistants"*, Expert Systems with Applications, 2021.

[32] Z. Huang et al., *"LayoutLMv3: Pre-training for Document AI with Unified Text and Image Masking"*, ACM MM 2022, arXiv:2204.08387.

[33] S. Long et al., *"Scene Text Detection and Recognition: The Deep Learning Era"*, International Journal of Computer Vision, 2021.

[34] O. Vinyals et al., *"Show and Tell: A Neural Image Caption Generator"*, CVPR 2015, arXiv:1411.4555.

[35] A. Agrawal et al., *"VQA: Visual Question Answering"*, International Journal of Computer Vision, 2017.

[36] D. Driess et al., *"PaLM-E: An Embodied Multimodal Language Model"*, ICML 2023, arXiv:2303.03378.

[37] V. Setty et al., *"Building Scalable and Flexible Cluster Managers Using Declarative Programming"*, OSDI 2020.

[38] Redis Ltd., *"Redis Streams: Introduction to Redis Streams"*, https://redis.io/docs/data-types/streams/, 2024.

[39] VMware Inc., *"RabbitMQ Documentation"*, https://www.rabbitmq.com/documentation.html, 2024.

[40] Apache Software Foundation, *"Apache Kafka Documentation"*, https://kafka.apache.org/documentation/, 2024.

[41] C. Richardson, *"Microservices Patterns: With examples in Java"*, Manning Publications, 2018.

[42] J. Dean and S. Ghemawat, *"MapReduce: Simplified Data Processing on Large Clusters"*, OSDI 2004.

[43] Docker Inc., *"Docker Best Practices for Building Images"*, https://docs.docker.com/develop/dev-best-practices/, 2024.

[44] Cloud Native Computing Foundation, *"Kubernetes Documentation"*, https://kubernetes.io/docs/, 2024.

[45] S. Newman, *"Building Microservices: Designing Fine-Grained Systems"*, O'Reilly Media, 2nd Edition, 2021.

[46] D. Merkel, *"Docker: Lightweight Linux Containers for Consistent Development and Deployment"*, Linux Journal, 2014.

[47] React Team, *"React Documentation: Learn React"*, https://react.dev/learn, 2024.

[48] OpenJS Foundation, *"Express.js: Fast, unopinionated, minimalist web framework"*, https://expressjs.com, 2024.

[49] D. Hardt, *"The OAuth 2.0 Authorization Framework"*, RFC 6749, IETF, October 2012.

[50] A. Osmani, *"Learning JavaScript Design Patterns"*, O'Reilly Media, 2nd Edition, 2023.

[51] J. Weizenbaum, *"ELIZA - A Computer Program for the Study of Natural Language Communication Between Man and Machine"*, Communications of the ACM, vol. 9, no. 1, pp. 36-45, 1966.

[52] D. Jurafsky and J. H. Martin, *"Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition"*, Pearson, 3rd Edition, 2023.

[53] A. Vaswani et al., *"Attention Is All You Need"*, Advances in Neural Information Processing Systems (NeurIPS), 2017.

[54] M. López, G. Valero, and A. Senabre, *"Evaluation of Commercial Voice Assistants: A Comparative Study"*, IEEE Access, vol. 9, pp. 45123-45138, 2021.

[55] N. Apthorpe et al., *"Keeping the Smart Home Private with Smart(er) IoT Traffic Shaping"*, Proceedings on Privacy Enhancing Technologies (PoPETs), 2019.

[56] T. Brown et al., *"Language Models are Few-Shot Learners"*, Advances in Neural Information Processing Systems (NeurIPS), arXiv:2005.14165, 2020.

[57] H. Touvron et al., *"LLaMA: Open and Efficient Foundation Language Models"*, arXiv preprint arXiv:2302.13971, 2023.

[58] J. Li et al., *"Multimodal Foundation Models: From Specialists to General-Purpose Assistants"*, arXiv preprint arXiv:2309.10020, 2023.

[59] M. Chui et al., *"The Social Economy: Unlocking Value and Productivity Through Social Technologies"*, McKinsey Global Institute Report, 2012.

[60] E. Brynjolfsson and A. McAfee, *"The Business of Artificial Intelligence: What It Can and Cannot Do for Your Organization"*, Harvard Business Review, July 2017.

[61] W. Xiong et al., *"Achieving Human Parity in Conversational Speech Recognition"*, IEEE/ACM Transactions on Audio, Speech, and Language Processing, 2017.

[62] A. Khandelwal et al., *"IndicNLP Suite: Monolingual Corpora, Evaluation Benchmarks and Pre-trained Multilingual Language Models for Indian Languages"*, Findings of EMNLP 2021.

[63] Y. Ren et al., *"A Survey on Neural Speech Synthesis"*, arXiv preprint arXiv:2106.15561, 2021.

[64] A. Abdul-Kader and J. Woods, *"Survey on Chatbot Design Techniques in Speech Conversation Systems"*, International Journal of Advanced Computer Science and Applications, 2015.

[65] P. Lewis et al., *"Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks"*, Advances in Neural Information Processing Systems (NeurIPS), 2020.

[66] M. Wooldridge, *"An Introduction to MultiAgent Systems"*, John Wiley & Sons, 2nd Edition, 2009.

[67] C. Richardson, *"Microservices Patterns: With Examples in Java"*, Manning Publications, Chapter 3: Interprocess Communication, 2018.

Appendices

Appendix A

Sustainable Development Goals addressed

#	**SDG**	**Level**
1	No Poverty	
2	Zero Hunger	
3	Good Health and Well-being	1
4	Quality education	3
5	Gender Quality	1
6	Clean water and Sanitation	
7	Affordable and Clean Energy	
8	Decent work and Economic Growth	2
9	Industry, Innovation and Infrastructure	3
10	Reduced Inequalities	2
11	Sustainable cities and Communities	1
12	Responsible Consumption and production	1
13	Climate action	
14	Life below water	
15	Life on Land	
16	Peace, Justice and Strong Institutions	2
17	Partnership's for the Goals	1

Levels: Poor:1, Good :2, Excellent:3

Appendix B

Self-Assesment of the Project

#	PO and PSO	Contribution from the Project	Level
1	Engineering Knowledge:	Applied knowledge of AI, web development, databases, and containerization	3
2	Problem Analysis:	Analyzed requirements for AI assistant and designed multi-agent solution	3
3	Design/development of solutions	Developed complete full-stack application with scalable architecture	3
4	Conduct investigations of complex problems:	Researched AI frameworks, voice processing, and integration patterns	3
5	Modern tool usage:	Used TypeScript, React, Docker, Redis, Prisma, AI SDKs	3
6	The Engineer and the world:	Created accessible solution with Indic language support	2
7	Ethics:	Implemented privacy-focused self-hosted solution with data encryption	3
8	Individual and Team Work:	Collaborated on system design and implementation	3
9	Communication:	Documented architecture and presented technical concepts	3
10	Project Management and Finance:	Managed development timeline and resource allocation	2
11	Life-long Learning:	Learned new AI frameworks, cloud services, and deployment practices	3
1	PSO1	Applied database systems, computing, and architecture knowledge	3
2	PSO2	Designed and developed full-stack application using modern practices	3
3	PSO3	Implemented RESTful APIs, WebRTC protocols, OAuth 2.0, Redis distributed caching, Docker containerization, and cloud service integrations (OpenAI, Google Cloud)	3

PSO1: Computer based systems development: Ability to apply the basic knowledge of database systems, computing, operating system, digital circuits, microcontroller, computer organization and architecture in the design of computer based systems.

PSO2: Software development: Ability to specify, design and develop projects, application softwares and system softwares by using the knowledge of data structures, analysis

and design of algorithm, programming languages, software engineering practices and open source tools.

PSO3: Computer communications

and Internet applications: Ability to design and develop network protocols and internet applications by incorporating the knowledge of computer networks, communication protocol engineering, cryptography and network security, distributed and cloud computing, data mining, big data analytics, ad hoc networks, storage area networks and wireless sensor networks.

Levels: Poor:1, Good :2, Excellent:3

Appendix C

System Technology Stack

C.1 Backend Technologies

- **Runtime:** Bun 1.0+ - Fast JavaScript runtime
- **Framework:** Express 4.18.2 + TypeScript 5.3.3
- **Database:** PostgreSQL 16 with Prisma 6.0.0 ORM
- **Cache/Queue:** Redis 7 with ioredis client
- **AI Services:** Google Gemini API, OpenAI GPT-4, Supermemory
- **Integrations:** Google APIs (Gmail, Calendar, Drive, Docs, Sheets), LiveKit, Exa Search
- **Authentication:** JWT with bcrypt password hashing
- **File Processing:** Multer for uploads, pdf-parse for text extraction

C.2 Frontend Technologies

- **Framework:** React 19 with TypeScript
- **Build Tool:** Vite for fast development and optimized builds
- **State Management:** Zustand
- **Routing:** React Router
- **Styling:** TailwindCSS with shadcn/ui components
- **HTTP Client:** Axios for API communication

C.3 Infrastructure and Deployment

- **Containerization:** Docker 20+ with Docker Compose 3.8
- **Web Server:** NGINX for serving frontend static files
- **Architecture:** Microservices with 5 containers (Frontend, Backend API, Worker, PostgreSQL, Redis)

Appendix D

Installation Guide

D.1 Prerequisites

- Bun 1.0+ or Node.js 18+
- PostgreSQL 14+
- Redis 7+ (optional for development)
- Docker and Docker Compose (for containerized deployment)
- Git for version control

D.2 Quick Start with Docker

```
# Clone repository
git clone https://github.com/himanshuraimau/Kuma.git
cd Kuma

# Configure environment variables
cp .env.example .env
# Edit .env with API keys

# Start all services
docker-compose up -d

# Run database migrations
docker-compose exec backend bun run db:push

# Access application at http://localhost
```

Listing D.1: Docker Installation Steps

D.3 Manual Installation

```
# Install dependencies
bun install:all

# Setup environment
cp backend/.env.example backend/.env
cp frontend/.env.example frontend/.env

# Configure database
bun db:generate
bun db:push

# Start development servers
bun dev
```

Listing D.2: Manual Installation Steps

Appendix E

Project Source Code

The complete source code for Kuma AI-Powered Personal Assistant is available at:

GitHub Repository: https://github.com/himanshuraimau/Kuma

E.1 Repository Structure

```
Kuma/
  backend/              Backend API server
    src/
      apps/             App integrations
      controllers/      Route handlers
      lib/              AI agents and utilities
      routes/           API endpoints
    prisma/             Database schema

  frontend/             React client application
    src/
      api/              API client functions
      components/       UI components
      stores/           State management

  docker-compose.yml    Container orchestration
```