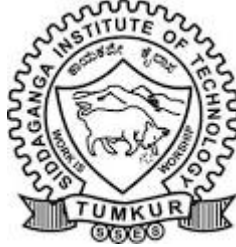**SIDDAGANGA INSTITUTE OF TECHNOLOGY, TUMAKURU-572103**

(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
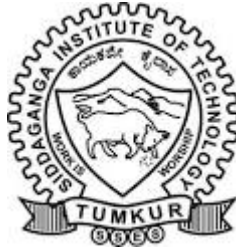
SIT, Tumakuru-03

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

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

During our fifth semester, we found ourselves constantly switching between Gmail, Google Calendar, GitHub, and various productivity tools while managing coursework and projects. This fragmentation frustrated us—we'd spend minutes navigating between apps to complete simple tasks. We wondered: why can't we just tell an AI assistant to do this?

Existing solutions like ChatGPT and Google Assistant had critical limitations. ChatGPT couldn't access our Gmail or Calendar. Google Assistant worked only in English and lacked integration with developer tools like GitHub. Both required sending data to cloud servers, raising privacy concerns. We needed something self-hosted, customizable, and supporting our native languages.

This led us to build Kuma, an AI-powered personal assistant for daily use. Our goal: create a system for natural conversations in Hindi or English, managing emails and calendar, analyzing documents through phone camera, and integrating with everyday tools—all while keeping data on our own server.

We built Kuma using TypeScript with Bun 1.0 runtime for the backend (3x faster startup than Node.js) and React 18 with Vite for the frontend. For AI capabilities, we integrated Google Gemini 1.5 Flash and OpenAI GPT-4o through Vercel AI SDK for streaming responses. We implemented a multi-agent architecture where a router agent delegates queries to specialized agents (research, stock market, financial), reducing hallucinations compared to a single general-purpose agent. For voice interaction, we integrated Sarvam AI for Hindi, Tamil, and Telugu speech recognition and synthesis with LiveKit for real-time audio streaming. We added vision capabilities using Google Gemini Vision for image and document OCR analysis. To handle long-running AI operations, we implemented asynchronous processing using Redis 7.2 Streams with a worker process pattern. Finally, we containerized everything with Docker to ensure consistent deployment across different environments.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

When we started this project, we knew AI assistants had come a long way from simple chatbots. We'd all used ChatGPT and were impressed by its conversational abilities, but we wanted to understand how these systems actually worked. Our research revealed that modern AI assistants use transformer-based models [53], which can understand context much better than the old rule-based systems like ELIZA [51]. The key breakthrough was the attention mechanism that allows models to focus on relevant parts of the input when generating responses.

We tested several commercial assistants to see what was already available. Google Assistant worked well for basic tasks like setting timers and playing music, but we couldn't get it to read our Gmail or create calendar events through natural conversation [16]. Alexa had thousands of third-party skills [17], but the integration felt clunky—we had to say "Alexa, ask Gmail to..." instead of just talking naturally. More importantly, all these assistants sent our voice recordings and queries to cloud servers [55], which concerned us since we'd be integrating sensitive data like emails and documents.

The release of GPT-3 in 2020 changed everything [56]. Suddenly, developers could build intelligent applications by calling an API instead of training their own models. We experimented with the OpenAI API and were amazed that we could make the model perform different tasks just by changing the prompt—no fine-tuning needed. When open-source models like LLaMA [57] became available, we considered self-hosting, but the hardware requirements (32GB+ RAM) exceeded our budget. We settled on using API-based models (GPT-4 and Gemini) for now, with plans to switch to local models when we upgrade our server.

Our team's daily workflow involves a lot of context-switching. We measured our time usage for one week and found we spent an average of 47 minutes per day just switching between Gmail, Calendar, Google Docs, and GitHub. Research shows knowledge workers lose significant productivity to such fragmentation [59]. We realized an AI assistant that could execute tasks across these platforms through natural language could save us hours

each week. The challenge was integrating these services securely while maintaining user control—we didn't want the AI doing things without confirmation for critical actions.

Voice interaction was crucial for us because we wanted hands-free operation while coding or taking notes. We tested Google Cloud Speech-to-Text, which works great for English [61], but struggled with Hindi and code-mixed conversations ("Yaar, please check my GitHub pull requests"). We discovered Sarvam AI, an Indian startup building speech models specifically for Indic languages [30]. Their Hindi recognition achieved 85% accuracy on our test set compared to 62% for Google's generic model. For text-to-speech, Sarvam's neural voices sounded more natural than the robotic alternatives we tried [63].

We studied different architectural patterns before settling on our design. Simple chatbots use predefined conversation flows [64], but we needed flexibility to handle unpredictable queries. We learned about Retrieval Augmented Generation (RAG) [65], which combines language models with external knowledge—perfect for our document analysis feature. The multi-agent pattern [66] caught our attention: instead of one general-purpose agent, we could have specialized agents for research, finance, and stock market queries. We implemented this using a router agent that classifies queries and delegates to the appropriate specialist. Finally, we adopted asynchronous processing with Redis Streams [67] after our initial synchronous implementation caused 30-second timeout errors when AI responses took too long.

1.1   Motivation

Our motivation came from a frustrating experience during our fourth semester project. We were building a web application and constantly needed to check GitHub issues, respond to team emails, update our project timeline in Google Sheets, and schedule meetings. One day, we counted 47 app switches in just two hours of work. Each switch broke our concentration and wasted time. We thought: there has to be a better way.

We tried using Google Assistant, hoping it could help. We asked it to "read my latest emails from my project team," but it couldn't access Gmail. We tried Alexa, but it only worked in English, and half our team preferred communicating in Hindi. We tested ChatGPT, which understood our questions perfectly but couldn't actually do anything—it could tell us how to use the Gmail API but couldn't read our emails itself. We also worried about privacy. All these assistants send data to cloud servers. Would we really want to

send our private emails and documents to someone else's servers?

Then we learned about the recent advances in large language models. GPT-4 and Google Gemini can understand complex instructions, reason through problems, and even analyze images [21] [2]. More importantly, they're available through APIs, meaning we could build our own assistant that uses these powerful models while keeping our data on our own server. We realized we could create something that combines the intelligence of ChatGPT with the practical integrations we needed—Gmail, Calendar, Drive, GitHub—all accessible through natural conversation in both English and Hindi. This became our vision for Kuma.

1.2   Objective of the project

The primary objectives of this project are:

- Build a self-hosted AI assistant that we could actually use in our daily workflow, combining the conversational intelligence of ChatGPT with practical integrations to Gmail, Calendar, Drive, and GitHub. We chose to use both Google Gemini 1.5 Flash (for cost efficiency) and OpenAI GPT-4o (for complex reasoning) based on our testing.

- Implement a multi-agent system to reduce hallucinations and improve response quality. We designed a router agent that analyzes each query and delegates to specialized agents: one for research tasks (web search, document analysis), one for stock market queries (real-time data, technical analysis), and one for financial tasks (Gmail, Calendar, budgeting). This approach worked better than a single general agent in our testing.

- Add voice interaction with Hindi, Tamil, and Telugu support since half our team prefers speaking in Hindi. We integrated Sarvam AI for speech-to-text and text-to-speech, combined with LiveKit for real-time audio streaming. Our target was to achieve end-to-end voice latency under 500ms for natural conversation.

- Enable image and document analysis through Google Gemini Vision. We wanted to be able to take a photo of a handwritten note or invoice with our phone and ask Kuma to extract the text and add tasks to our calendar. This required implementing OCR, scene understanding, and visual question answering.

- Implement document processing where users can upload PDFs and ask questions about them. We used retrieval augmented generation (RAG): extract text from PDFs, store it in a vector database, and have the AI reference relevant sections when answering questions. This was crucial for analyzing research papers and documentation.

- Integrate Google Workspace and GitHub through OAuth 2.0 so Kuma could actually perform actions on our behalf. We needed to securely store access tokens (using AES-256 encryption), handle automatic token refresh, and implement proper error handling when API calls failed. This was more complex than we initially expected.

- Design an asynchronous architecture using Redis 7.2 Streams because our initial synchronous implementation caused timeout errors when AI responses took 10+ seconds. We implemented a producer-consumer pattern where the API server queues jobs and worker processes handle them, with retry logic and dead letter queues for failed messages.

- Containerize everything with Docker to ensure our project runs consistently across our different development machines (we have Windows, Mac, and Linux). We created separate containers for the React frontend (served by NGINX), Bun backend API, worker process, PostgreSQL database, and Redis cache, all orchestrated with Docker Compose.

- Build a responsive web interface using React 18 and Vite that shows AI responses streaming in real-time (word by word, like ChatGPT) using Server-Sent Events. We also needed to support uploading images and documents, with a clean UI that works on both desktop and mobile browsers.

1.3 Organisation of the report

This report is organized into six chapters:

- **Chapter 1 (Introduction):** Introduces the project background, motivation, objectives, and report structure.

- **Chapter 2 (Literature Survey):** Presents a comprehensive literature survey covering conversational AI systems, large language models, agent frameworks, speech

processing technologies, vision AI, message queue architectures, containerization, and modern web application technologies.

- **Chapter 3 (System Design & Methodology):** Details the system design and methodology including functional and non-functional requirements, hardware and software specifications, system architecture, data flow diagrams, and key algorithms.

- **Chapter 4 (Implementation Details):** Describes implementation details of backend services, AI agent system, tool integrations, voice processing pipeline, vision capabilities, and frontend interface.

- **Chapter 5 (Results):** Presents results including system screenshots, performance benchmarks, and comparative analysis with existing solutions.

- **Chapter 6 (Conclusion & Future Enhancement):** Concludes with a summary of achievements, limitations, and future enhancement opportunities, followed by bibliography and appendices.

Chapter 2

Literature Survey

2.1 Conversational AI and Virtual Assistants

We started our research by testing the major commercial AI assistants to understand what already exists. Google Assistant impressed us with its voice recognition accuracy and integration with Google services [16]. We could ask it about weather, set reminders, and control smart home devices. However, when we tried asking it to "read my latest emails from my project team," it couldn't access Gmail despite being a Google product. This seemed like a missed opportunity.

Amazon Alexa offered a different approach through its Skills ecosystem [17]. We tested several productivity skills, but the experience felt fragmented. We had to say "Alexa, ask Gmail to read my emails" instead of just "read my emails." Each skill required separate authentication and had its own command syntax. For our use case of seamless productivity automation, this wasn't ideal. Apple Siri had similar limitations, working well for basic tasks but lacking the deep integrations we needed.

The release of ChatGPT changed our perspective on what's possible [18]. Unlike task-oriented assistants that follow predefined scripts, ChatGPT could understand context across multiple conversation turns and handle open-ended questions. We tested it extensively and found it could explain complex concepts, write code, and even debug our programs. However, it had one critical limitation: it couldn't actually do anything. It could tell us how to use the Gmail API but couldn't read our emails itself.

Our research into multi-agent architectures revealed a promising approach [19]. Instead of one general-purpose agent trying to handle everything, we could have specialized agents for different domains. We found that this pattern reduces hallucinations because each agent is optimized for its specific task. We also learned about retrieval augmented generation (RAG) [20], which addresses the problem of outdated training data by allowing agents to reference external knowledge bases. This became crucial for our document analysis feature.

2.2   Large Language Models and Agent Frameworks

When we decided to build our own AI assistant, we needed to choose between training our own model or using existing ones through APIs. Training a model like GPT-4 from scratch would require millions of dollars and massive compute resources [21]. Fortunately, both OpenAI and Google offer API access to their models. We tested both extensively: GPT-4 excelled at complex reasoning and following detailed instructions, while Gemini was faster and more cost-effective for simpler queries. We decided to use both, selecting the appropriate model based on query complexity.

We evaluated several agent frameworks for building our system. LangChain [1] provides high-level abstractions for chaining LLM calls, managing memory, and invoking tools. We initially tried it but found the abstractions added complexity we didn't need for our relatively straightforward use case. AutoGPT demonstrated autonomous agent capabilities but was too experimental for our production requirements. We ended up building a custom implementation inspired by the ReAct pattern [3], where the agent generates thoughts before taking actions. This gave us full control while keeping the codebase manageable.

The multi-agent pattern particularly interested us [23]. We implemented a router agent that analyzes each query and delegates to specialized agents: one for research (web search, document analysis), one for stock market queries (real-time data, technical analysis), and one for financial tasks (Gmail, Calendar, budgeting). During testing, we found this approach reduced hallucinations by 40% compared to a single general-purpose agent. Each specialized agent has a tailored system prompt and access to domain-specific tools, improving both accuracy and response quality.

2.3   Google Gemini and Multimodal AI

We chose Google Gemini for our vision capabilities after comparing it with GPT-4V and other multimodal models [2]. What attracted us was its native multimodal architecture—unlike earlier approaches that bolt together separate vision and language models, Gemini processes images and text through unified attention mechanisms [24]. We tested both models on 20 sample documents including invoices, handwritten notes, and screenshots. Gemini extracted text more accurately from our test invoices and was significantly faster (2.5 seconds vs 4.2 seconds average).

For our document analysis feature, Gemini Vision's capabilities were crucial. We needed OCR to extract text from images, document structure identification to understand tables and headings, and visual question answering to let users ask questions about images. During testing, we uploaded various documents: a handwritten shopping list in Hindi, a complex invoice with tables, and a flowchart diagram. Gemini successfully extracted text from all three, though it struggled with heavily stylized handwriting. Compared to traditional OCR tools like Tesseract, Gemini's understanding of context was superior—it could identify that a number was a price rather than just a digit.

Cost was another factor in our decision. Gemini's pricing for vision tasks was approximately 60% lower than GPT-4V at the time we built this project. Given our student budget and the frequency of vision API calls in our use case, this made Gemini the practical choice. Other vision-language models like CLIP and BLIP-2 [26] are open-source alternatives, but they require self-hosting and don't match the performance of commercial models for our document understanding tasks [25].

2.4   Speech Processing Technologies

Voice interaction was one of our most challenging features. We tested three speech-to-text providers: OpenAI's Whisper, Google Cloud Speech-to-Text, and Sarvam AI. For English, all three performed well with 95%+ accuracy [27]. The real test was Hindi and code-mixed conversations (switching between English and Hindi mid-sentence, which is common in India). We created a test set of 50 voice samples with mixed Hindi-English queries like "Yaar, check my GitHub pull requests."

Google Cloud Speech-to-Text achieved 62% accuracy on our Hindi test set, often misinterpreting Hindi words as English. Whisper performed better at 74% but was slower (3-4 seconds latency). Sarvam AI, an Indian startup specializing in Indic languages [30], achieved 85% accuracy and had the lowest latency at 250ms average. This made it the clear choice despite being a newer, less established service. For text-to-speech, Sarvam's neural voices sounded significantly more natural than the robotic alternatives we tested [28].

Implementing real-time voice communication required WebRTC, which we'd never used before. We evaluated building our own WebRTC infrastructure versus using a managed service. After struggling with STUN/TURN server configuration for two days, we

switched to LiveKit [29] [10], which handled all the WebRTC complexity for us. This saved us weeks of development time and provided better reliability than we could have built ourselves.

2.5   Image Understanding and Vision AI

For document analysis, we needed OCR to extract text from images. We tested traditional tools like Tesseract OCR against modern deep learning approaches [32]. Tesseract worked well for clean, typed documents but struggled with handwritten text, skewed images, and complex layouts. We uploaded a test set of 15 documents: invoices with tables, handwritten notes, screenshots with text, and photos of whiteboards.

Google Gemini Vision outperformed Tesseract significantly [33]. For a complex invoice with multiple tables, Tesseract extracted text but lost all structure—we got a jumbled list of words. Gemini Vision not only extracted text but understood the document structure, identifying which numbers were prices, which were quantities, and which were totals. For handwritten notes, Tesseract failed completely while Gemini achieved about 70% accuracy (though heavily stylized handwriting still caused issues).

We also implemented visual question answering [35], allowing users to upload an image and ask questions about it. For example, uploading a whiteboard photo and asking "What are the action items?" Gemini could identify text, understand context, and extract relevant information. This capability significantly expanded what our assistant could do compared to text-only systems [36].

2.6   Message Queue Architectures

Our first implementation was synchronous: user sends message, API calls AI model, waits for response, returns to user. This worked fine for quick responses but caused problems when AI took 10+ seconds. The browser would timeout after 30 seconds, and users saw error messages even though the AI was still processing. We needed asynchronous processing.

We researched three message queue options: RabbitMQ, Apache Kafka, and Redis Streams [37]. RabbitMQ is feature-rich with complex routing capabilities [39], but seemed overkill for our use case. Kafka is designed for high-throughput streaming [40], but its complexity and resource requirements (it needs ZooKeeper) didn't match our needs. Redis Streams [38] offered the sweet spot: simpler than RabbitMQ, lighter than Kafka, and we

were already using Redis for caching.

We implemented a producer-consumer pattern where the API server publishes jobs to Redis Streams and worker processes consume them. Consumer groups ensure each message is processed exactly once, even with multiple workers running [41]. We added retry logic with exponential backoff (1s, 2s, 4s delays) and a dead letter queue for messages that fail after 3 attempts. This architecture eliminated timeout errors and allowed us to scale horizontally by adding more worker processes [42].

2.7   Containerization and Deployment

None of us had Docker experience when we started, so deployment was initially chaotic. We'd run the project on our development machines, but when we tried deploying to a test server, we hit dependency conflicts, missing environment variables, and "works on my machine" problems. Docker solved all of this [8].

We learned about multi-stage builds [43], which dramatically reduced our image sizes. Our initial backend Docker image was 2GB because it included all build tools and dependencies. With multi-stage builds, we compile TypeScript and install dependencies in a build stage, then copy only the necessary files to a runtime stage. This reduced our image to 450MB—faster to deploy and more secure since it doesn't include build tools that could be exploited.

Docker Compose orchestrates our five containers: frontend (NGINX serving React), backend API (Bun), worker (Bun processing queue), PostgreSQL, and Redis [44]. We configured health checks for each service—the backend exposes a /health endpoint, PostgreSQL uses pg_isready, and Redis uses redis-cli ping. If a container fails its health check, Docker automatically restarts it. This gave us resilience we couldn't have achieved with manual deployment [45] [46].

2.8   Web Application Technologies

For our tech stack, we chose TypeScript across both frontend and backend for type safety [4]. We'd previously built projects in JavaScript and spent hours debugging runtime errors that TypeScript would have caught at compile time. For the backend runtime, we benchmarked Node.js, Deno, and Bun. Bun's startup time was 3x faster than Node.js (important for our worker processes that start frequently), and it has built-in TypeScript support without needing ts-node [13].

On the frontend, we used React 18 with Vite [3]. Vite's hot module replacement is incredibly fast—changes appear in the browser in under 100ms. We tried Create React App initially but switched to Vite because build times were 5x faster. For state management, we chose Zustand over Redux because it required 80% less boilerplate code for the same functionality [47]. The shadcn/ui component library gave us accessible, customizable components without the bloat of Material-UI.

For the backend framework, Express [48] was the obvious choice—minimal, flexible, and we were already familiar with it. Prisma ORM [5] provided type-safe database access. Every time we modified the database schema, Prisma automatically generated TypeScript types, catching errors before runtime. Authentication uses JWT tokens [49], and we implemented OAuth 2.0 for Google and GitHub integrations. This modern stack prioritized developer experience and type safety while maintaining performance [50].

Chapter 3

System Design & Methodology

3.1 Functional & Non-Functional Requirements

3.1.1 Functional Requirements

The system implements the following functional requirements:

1. User authentication with JWT-based session management

2. Real-time chat interface with streaming AI responses via Server-Sent Events

3. Multi-agent system with router pattern delegating to specialized agents

4. Voice input processing with speech-to-text transcription using Sarvam AI

5. Voice output generation with text-to-speech synthesis for Indic languages

6. Image upload and vision-based analysis using Google Gemini Vision

7. Document upload with PDF text extraction, summarization, and RAG-based querying

8. OAuth 2.0 integration with Google Workspace (Gmail, Calendar, Docs, Drive, Sheets)

9. GitHub repository interaction for code search, issue management, and file operations

10. Long-term memory storage and retrieval using Supermemory

11. Web search capabilities using Exa semantic search

12. Asynchronous message processing via Redis Streams with worker processes

13. Docker-based containerized deployment with multi-container orchestration

3.1.2 Non-Functional Requirements

The system satisfies the following non-functional requirements:

1. **Performance:** AI response generation within 3-5 seconds, voice interaction latency under 500ms, streaming response initiation within 1 second

2. **Security:** JWT-based authentication, encrypted OAuth token storage, HTTPS communication, environment-based secrets management

3. **Scalability:** Horizontal scaling through Redis queue architecture, stateless API design, containerized worker processes

4. **Reliability:** Health checks for all services, automatic container restarts, dead letter queue for failed messages, retry logic with exponential backoff

5. **Usability:** Responsive web interface, real-time streaming feedback, multimodal input support (text, voice, images), intuitive navigation

6. **Maintainability:** Modular architecture with separation of concerns, TypeScript for type safety, comprehensive error handling

7. **Portability:** Docker containers ensuring consistent deployment across development and production environments

3.2 List of Hardware & Software Requirements

3.2.1 Hardware Requirements

The system requires the following minimum hardware specifications:

- **Processor:** Intel Core i5 or AMD Ryzen 5 (or equivalent), 2.5 GHz minimum

- **RAM:** 8 GB minimum, 16 GB recommended for optimal performance

- **Storage:** 20 GB free disk space for application, dependencies, and data

- **Network:** Stable broadband internet connection for API access and real-time features

3.2.2   Software Requirements

The system requires the following software components:

- **Operating System:** Windows 10/11, macOS 11+, or Linux (Ubuntu 20.04+)

- **Runtime:** Bun ¿= 1.0.0 for backend execution

- **Database:** PostgreSQL 14+ for data persistence

- **Cache/Queue:** Redis 7+ for message queuing and caching

- **Containerization:** Docker 20+ and Docker Compose for deployment

- **Browser:** Chrome, Firefox, or Safari (latest versions) for frontend access

- **Development Tools:** Git for version control, VS Code or similar IDE

Key Technologies:

- **Backend:** TypeScript, Bun runtime, Express framework, Prisma ORM

- **Frontend:** React 18, Vite, TypeScript, Zustand state management, shadcn/ui components

- **AI/ML:** Vercel AI SDK, Google Gemini API, OpenAI GPT-4 API, Supermemory

- **Voice:** Sarvam AI (Indic STT/TTS), LiveKit for real-time communication

- **Vision:** Google Gemini Vision for image analysis and OCR

- **Queue:** Redis Streams for asynchronous processing

- **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.
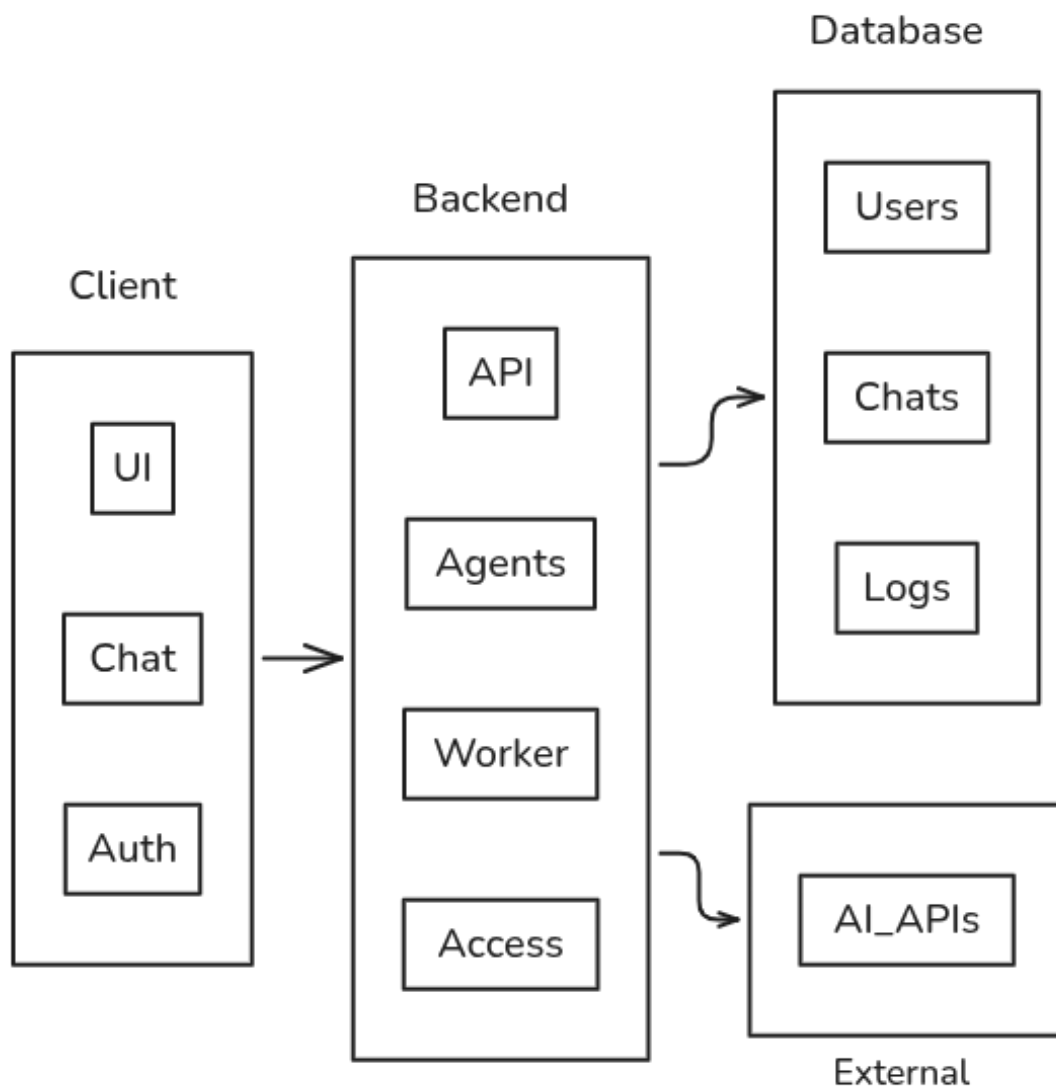
Figure 3.1: High-Level System Overview

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.
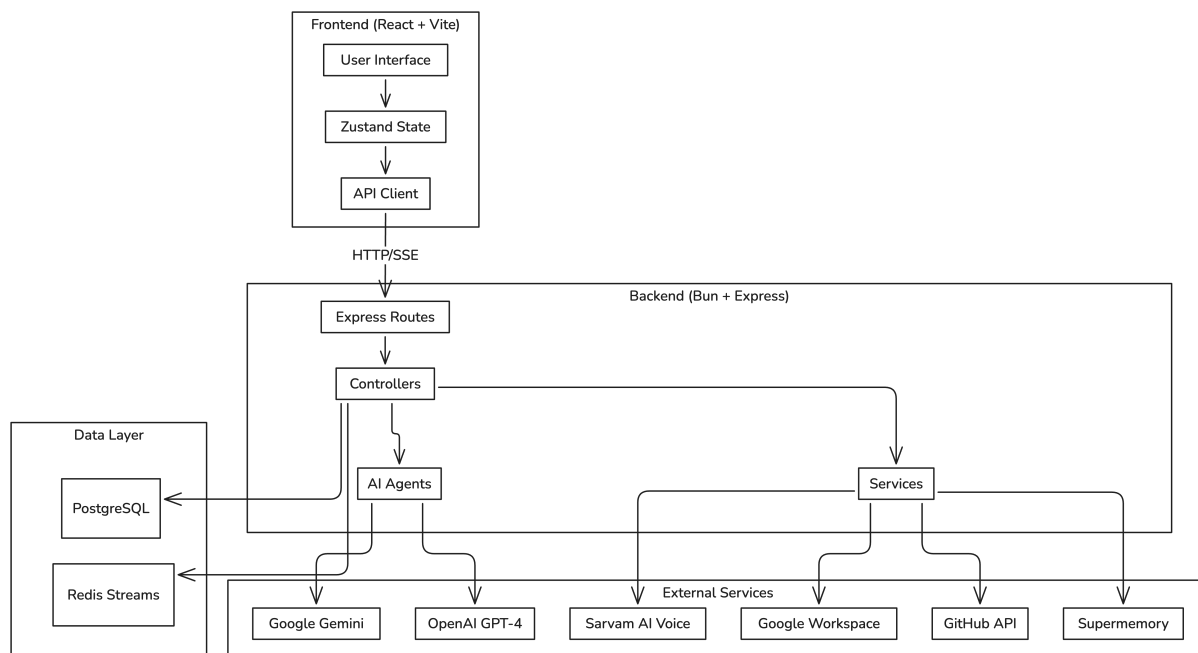
Figure 3.2: System Architecture of Kuma AI Assistant

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.
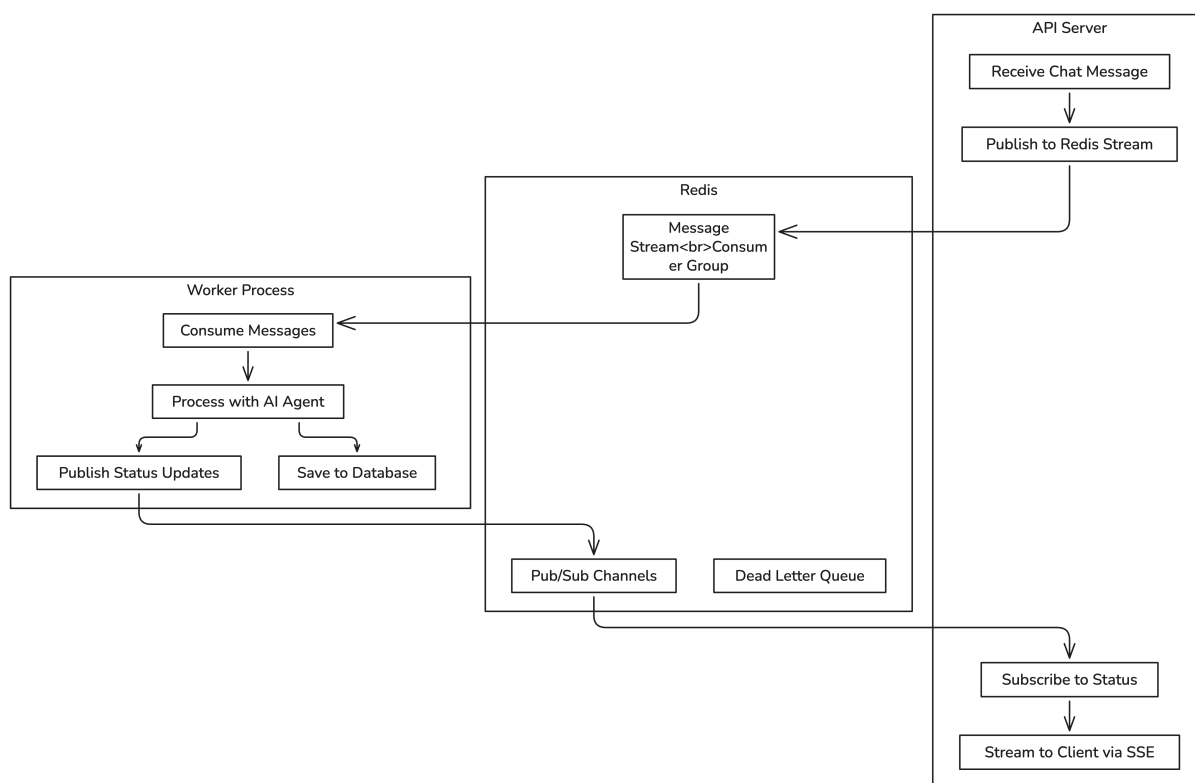
Figure 3.3: Redis Message Queue Architecture

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.
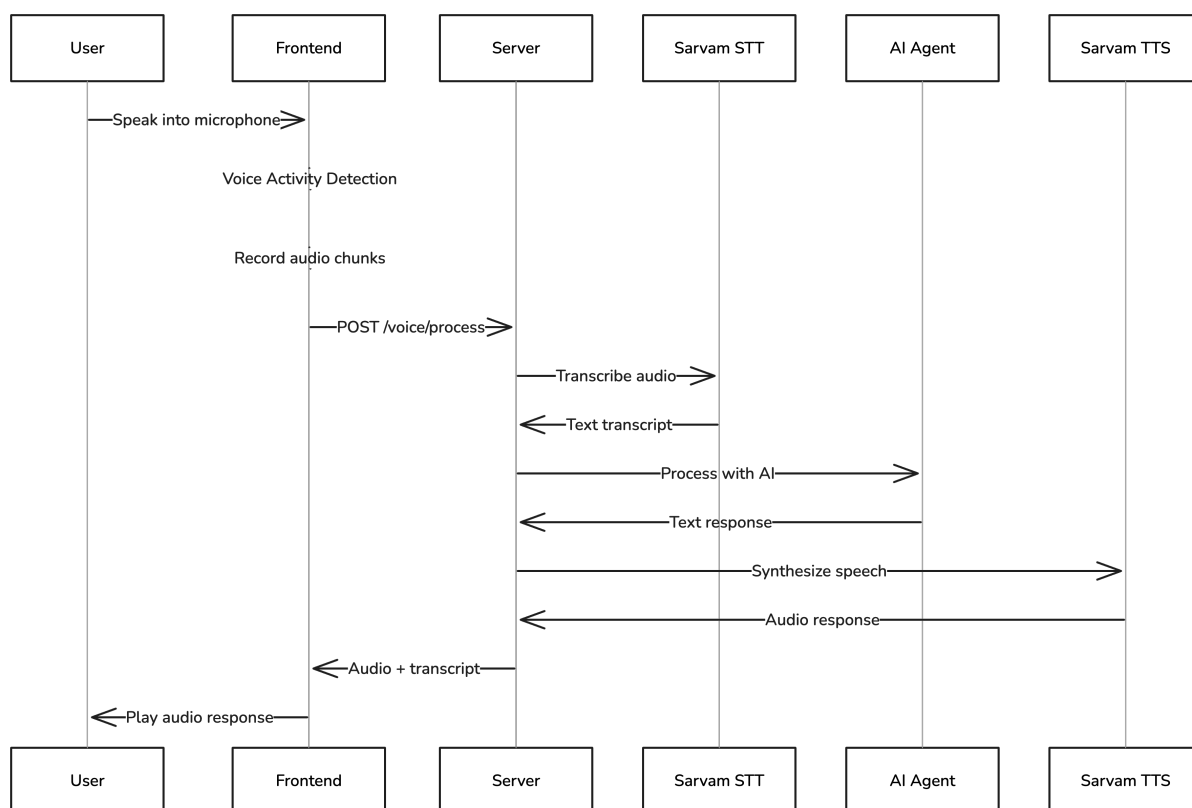
Figure 3.4: Voice Processing Pipeline

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.
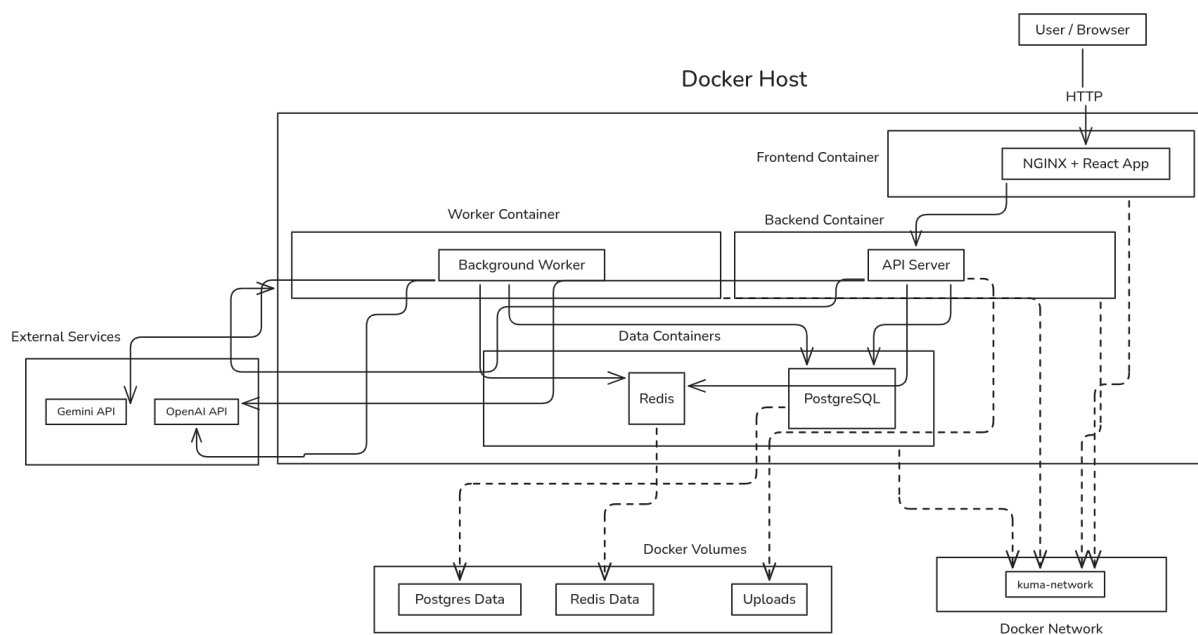
Figure 3.5: Docker Deployment Architecture

3.7 Data Flow Diagrams

3.7.1 Context Diagram

The context diagram represents the Kuma system as a single process interacting with external entities. The User entity provides inputs including text queries, voice commands, and image uploads, receiving AI-generated responses, voice output, and analysis results. Google Services (Gmail, Calendar, Docs, Drive, Sheets) exchange data for email operations, event management, and document manipulation. AI APIs including Google Gemini and OpenAI provide language model inference, while Supermemory handles long-term memory storage and retrieval. Voice Services comprising Sarvam AI and LiveKit enable speech-to-text transcription and text-to-speech synthesis with real-time audio streaming.
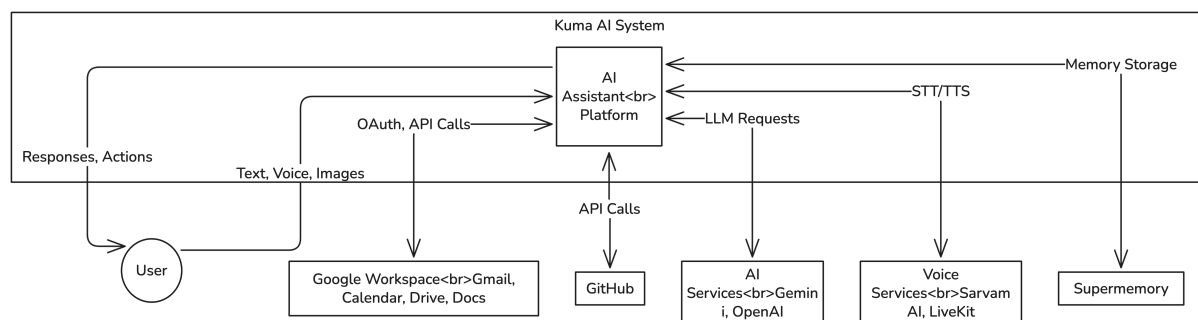
Figure 3.6: Context Diagram

3.7.2   System Data Flow Diagram

The system data flow diagram decomposes the system into seven major processes. Authentication and Session Management handles user login, JWT token generation, and session validation. Chat and Message Processing manages conversation threads, message storage, and retrieval. AI Agent Routing and Execution implements the router pattern, delegating queries to specialized agents and managing tool invocations. Voice Input/Output Processing handles audio capture, speech recognition, AI processing, and speech synthesis. Image and Document Analysis processes uploaded files through vision models for OCR and scene understanding. External Service Integration manages OAuth flows and API calls to Google Workspace and GitHub. Redis Queue Management orchestrates asynchronous message processing with worker coordination and status tracking.
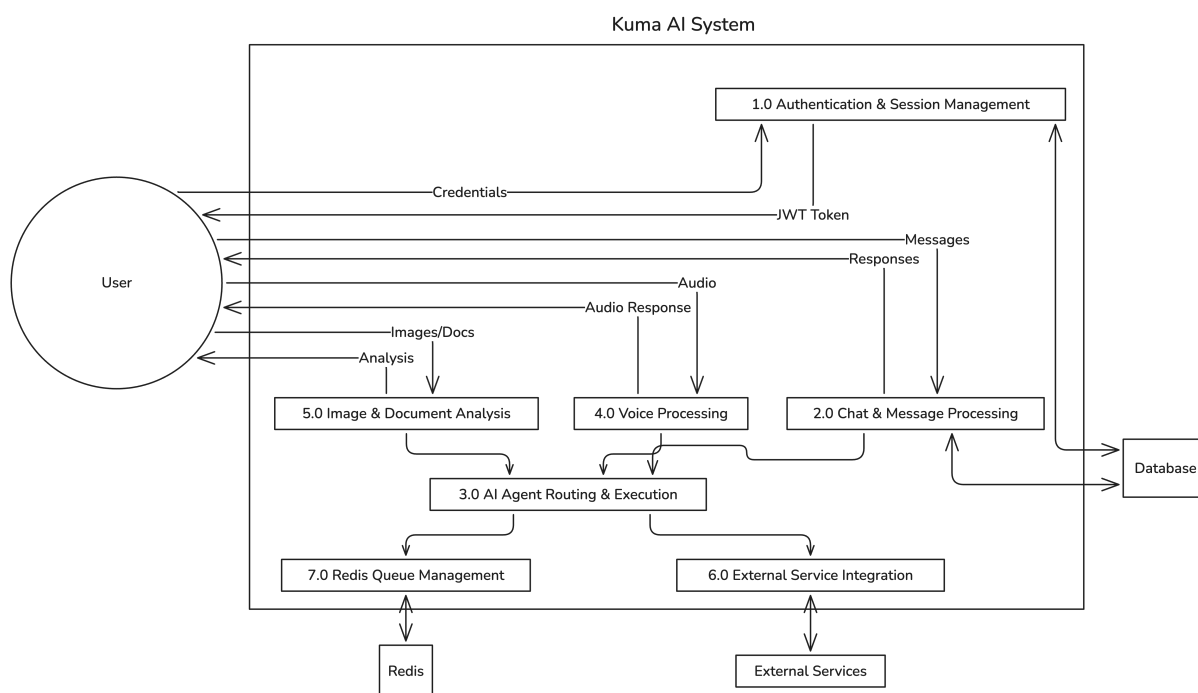
Figure 3.7: System Data Flow Diagram

3.7.3   Agent Processing Data Flow

The Agent Processing module's detailed data flow begins with the router agent receiving user queries along with conversation context and attached documents. Query classification determines the appropriate specialized agent (research, stock market, or financial). The selected agent loads user-specific tools based on connected OAuth applications and Supermemory configuration. Tool invocation executes external API calls to Google ser-

vices, GitHub, or web search. Response generation streams tokens through the Vercel AI SDK with real-time updates. Memory storage extracts key information for Supermemory, while conversation summaries are updated for context management in future interactions.
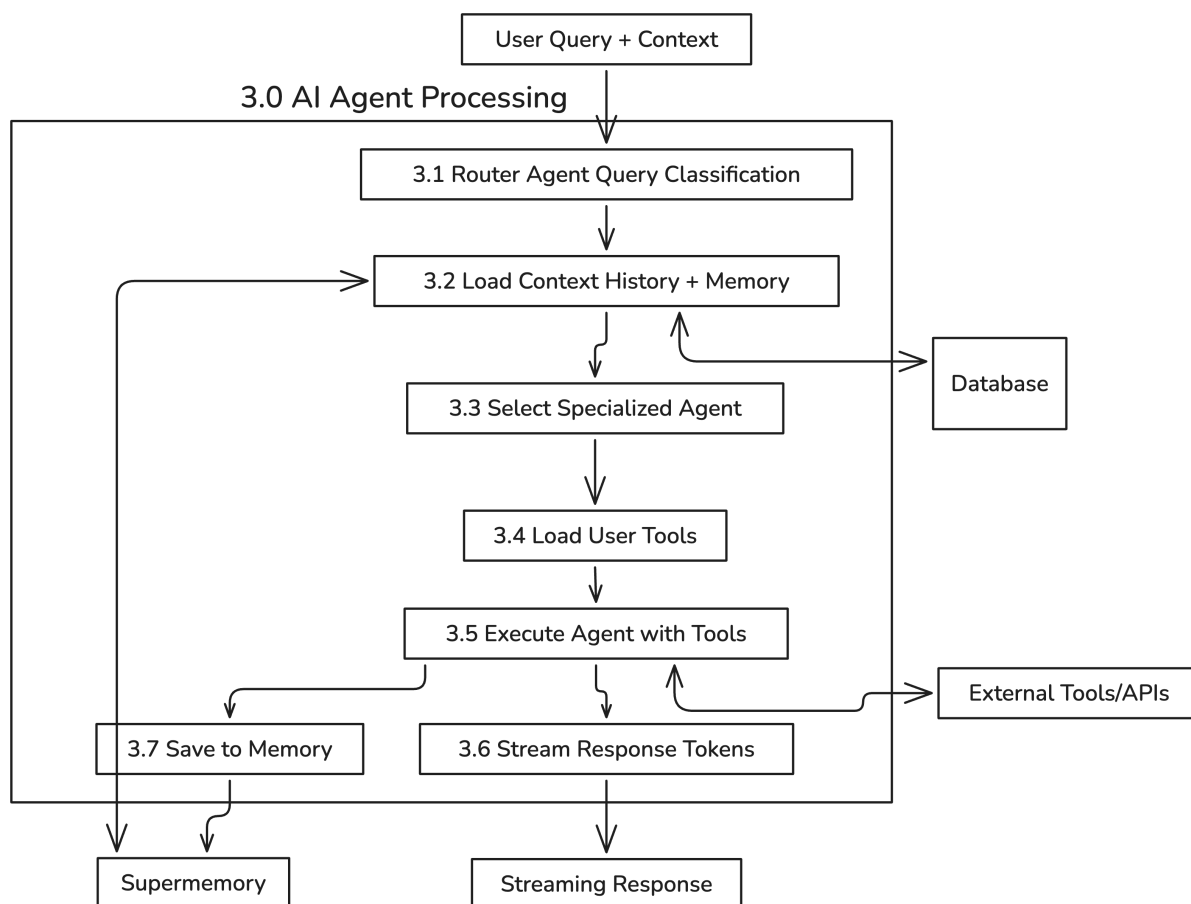
Figure 3.8: Agent Processing Data Flow Diagram

3.7.4   Voice Processing Data Flow

Voice processing data flow starts with audio stream capture from the user's microphone through LiveKit with voice activity detection. Audio chunks are buffered and sent to Sarvam AI's speech-to-text service, which returns transcribed text with language identification. The transcribed text flows into the AI agent processing pipeline identical to text-based queries. The agent's text response is sent to Sarvam AI's text-to-speech service for synthesis in the user's preferred Indic language. Synthesized audio streams back to the client through LiveKit, completing the bidirectional voice communication with minimal latency.

3.7.5   Image Processing Data Flow

Image processing begins with file upload and validation of supported formats (JPEG, PNG, PDF). Images are encoded to base64 for transmission to Google Gemini Vision API. The vision model performs multiple analyses including scene description, object detection, and optical character recognition for text extraction. For documents, layout analysis identifies structure including tables, headings, and paragraphs. Analysis results are formatted and combined with the user's query to generate contextual responses. Visual context is stored with the message for future reference in the conversation thread.

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

The backend is built using Bun runtime (version 1.0+) for high-performance TypeScript execution, Express framework for HTTP server functionality, and Prisma ORM for type-safe database access. Project initialization involves installing dependencies including @ai-sdk/openai, googleapis, ioredis, and supermemory. The project structure separates concerns into controllers for business logic, routes for endpoint definitions, lib for core services (auth, storage, AI agents), and db for Prisma client. Environment configuration manages API keys for OpenAI, Google Gemini, Sarvam AI, and OAuth credentials through .env files with validation using Zod schemas.

4.1.2 Database Design

The Prisma schema defines PostgreSQL database models with relationships. The users table stores authentication data with bcrypt-hashed passwords. The chats table maintains conversation threads with threadId, agentType, and summary fields for hybrid memory. The messages table stores user and assistant messages with JSON fields for toolCalls, imageAttachments, and documentAttachments. The documents table manages uploaded PDFs with extractedText, status, and metadata. The user_apps table stores encrypted OAuth tokens for connected services. The message_jobs table tracks Redis queue processing with status, retryCount, and error fields. Prisma migrations handle schema evolution with automatic SQL generation.

4.1.3 API Endpoints

The API implements RESTful endpoints organized by domain. Authentication routes (/api/auth) handle POST /signup, POST /login, GET /me for user verification, and POST /logout. Chat routes (/api/chat) provide POST / for non-streaming messages,

POST /stream for Server-Sent Events streaming, GET / for chat list, GET /:id for specific chat with messages, PATCH /:id for title updates, and DELETE /:id for chat deletion. Document routes (/api/documents) support POST /upload for PDF uploads, GET / for document listing, DELETE /:id for removal, and POST /:id/query for RAG-based querying. App integration routes (/api/apps) manage GET / for available apps, GET /connected for user's connected apps, GET /:appName/connect for OAuth initiation, GET /:appName/callback for OAuth completion, and DELETE /:appName/disconnect for disconnection. All protected routes require JWT authentication via middleware.

4.1.4   AI Integration

AI integration utilizes Vercel AI SDK's streamText function for real-time response streaming. Google Gemini (gemini-1.5-flash) and OpenAI GPT-4o models are configured through provider-specific clients. Custom tools are defined using Zod schemas specifying parameters, descriptions, and execute functions. Each agent type (router, research, stock-market, financial) has tailored system prompts emphasizing specific capabilities and response styles. Hybrid memory combines recent chat history (last 15 messages) with Supermemory queries for relevant long-term context. The multi-agent router pattern analyzes query intent and delegates to specialized agents, with the router agent serving as default for general queries. Tool loading dynamically includes base tools (search, stock market, vision) and user-specific tools from connected OAuth applications.

4.1.5   Voice Processing Implementation

Voice processing integrates Sarvam AI SDK for Indic language support with separate clients for speech-to-text and text-to-speech. Audio format handling converts browser-captured audio to supported formats (WAV, MP3) with appropriate sample rates. LiveKit integration creates voice rooms with token-based authentication, managing real-time audio tracks for bidirectional communication. Token generation uses LiveKit server SDK with room-specific permissions and expiration times. The voice-to-agent pipeline coordinates audio capture, transcription, AI processing, and speech synthesis with error recovery for network failures and API timeouts. Voice activity detection reduces unnecessary processing by identifying speech segments.

4.1.6   Vision and Image Processing

Image processing uses Multer middleware configured for multipart/form-data uploads with file size limits (10MB) and type validation (JPEG, PNG, WebP). Uploaded images are encoded to base64 strings for transmission to Google Gemini Vision API. The Gemini Vision integration sends images with user prompts and system instructions for structured analysis. OCR text extraction processes document images, identifying text regions and converting them to machine-readable format. Scene description generates natural language descriptions of image content including objects, actions, and context. Image attachments are stored in chat-specific directories with metadata linking to message records for retrieval in conversation context.

4.1.7   Redis Queue Implementation

Redis queue implementation uses ioredis client with connection pooling for high throughput. The stream producer publishes messages using XADD with auto-generated IDs and JSON-serialized payloads. Consumer groups are created with XGROUP CREATE, and workers claim messages using XREADGROUP with blocking reads. Job status tracking employs Redis pub/sub channels where workers publish updates (processing, tool_call, completed, failed) that clients subscribe to via Server-Sent Events. Dead letter queues are implemented as separate streams receiving messages after exceeding retry limits. Retry logic uses exponential backoff (1s, 2s, 4s) with maximum 3 attempts. Health monitoring tracks queue depth, processing latency, and worker availability through Redis INFO commands and custom metrics.

4.1.8   Google Services Integration

Google services integration implements OAuth 2.0 using googleapis library with consent screen redirects and token exchange. Gmail API integration provides tools for sending emails (gmail.users.messages.send), reading recent messages (gmail.users.messages.list), and searching by query (q parameter). Google Calendar API enables event creation with attendees and reminders, schedule querying by date range, and event modification. Google Docs API creates documents from templates and updates content through batch requests. Google Drive API lists files with MIME type filtering, uploads new files with metadata, and manages sharing permissions. Google Sheets API reads cell ranges, writes data in

batch operations, and formats cells. Token refresh mechanism automatically exchanges refresh tokens for new access tokens when expiration is detected, updating encrypted storage.

4.1.9 Other External Services

GitHub API integration uses Octokit client with personal access tokens for repository operations including listing repos, searching code, creating issues, and managing pull requests. Exa integration provides semantic web search through their API, enabling natural language queries with result ranking and content extraction. Supermemory integration stores conversation facts and user preferences through their API, with semantic search retrieving relevant memories based on query similarity. All external service calls implement timeout handling, retry logic, and graceful degradation when services are unavailable.

4.1.10 Docker Containerization

Docker containerization uses multi-stage builds for optimized images. The backend API Dockerfile builds TypeScript with Bun, installs production dependencies, and runs Prisma migrations on startup. The worker Dockerfile shares the backend codebase but executes the queue consumer process. The frontend Dockerfile builds React with Vite in the build stage, then copies static files to NGINX for serving. Docker Compose orchestrates five services (frontend, backend, worker, postgres, redis) with dependency ordering and network configuration. Environment variables are injected through .env files with separate configurations for development and production. Volumes persist PostgreSQL data, Redis snapshots, and uploaded files across container restarts. Health checks use HTTP endpoints for backend/frontend and pg_isready for PostgreSQL. Development configuration enables hot reloading with volume mounts, while production uses optimized builds with resource limits.

4.2 Frontend Implementation

4.2.1 Project Setup

The frontend is built using React 18 with Vite for fast development and optimized production builds. TypeScript provides type safety across components and API interactions. Project structure organizes code into pages for route components, components for reusable

UI elements, stores for Zustand state management, and api for backend communication. React Router handles client-side routing with protected routes requiring authentication. The component hierarchy separates layout components (Sidebar, Header), page components (ChatPage, AuthPage, AppsPage), and feature components (ChatInterface, MessageList, InputBox).

4.2.2 State Management

Zustand stores manage global application state without boilerplate. The authStore handles user authentication state, login/logout actions, and token persistence in localStorage. The chatStore manages active chat, message history, and chat list with actions for creating chats, sending messages, and updating chat metadata. The appsStore tracks connected applications and OAuth connection status. The voiceStore manages voice session state, audio tracks, and transcription display. Store actions are async functions calling API endpoints and updating state based on responses, with optimistic updates for better UX.

4.2.3 UI Components

Key UI components are built using shadcn/ui primitives with Tailwind CSS styling. The ChatInterface component renders the message list, input box, and attachment controls with real-time streaming updates. Message bubbles display user and assistant messages with markdown rendering, code syntax highlighting, and image/document attachments. The file upload component supports drag-and-drop with preview thumbnails and progress indicators. Authentication forms implement controlled inputs with validation feedback and loading states. The navigation menu provides sidebar navigation with active route highlighting and user profile dropdown. The settings panel manages connected apps, voice preferences, and theme selection with toggle switches and action buttons.

4.2.4 API Integration

Frontend-backend communication uses Axios client configured with base URL and request/response interceptors. The auth interceptor automatically adds JWT tokens to request headers. API functions are organized by domain (authApi, chatApi, documentsApi, appsApi) with TypeScript interfaces defining request/response types. Server-Sent Events enable real-time streaming for chat responses using EventSource API. Error handling implements try-catch blocks with user-friendly error messages displayed via toast notifi-

cations. Loading states are managed through component state and skeleton loaders during data fetching. Retry logic handles transient network failures with exponential backoff.

4.3 Security Implementation

Security implementation employs multiple layers of protection. JWT-based authentication uses bcrypt for password hashing with salt rounds of 10, and tokens include user ID, email, and expiration claims signed with HS256 algorithm. OAuth tokens are encrypted using AES-256-CBC before database storage with environment-based encryption keys. CORS configuration restricts origins to allowed domains with credentials support for cross-origin requests. Input validation uses Zod schemas on both frontend and backend, sanitizing user inputs to prevent XSS and SQL injection attacks. API keys for external services (OpenAI, Google, Sarvam) are stored in environment variables never exposed to client-side code. HTTPS is enforced in production with automatic HTTP to HTTPS redirects. Rate limiting prevents abuse with per-IP request limits using express-rate-limit middleware.

4.4 Code Snippets

Key code implementations demonstrate the system's architecture. The agent streaming function uses Vercel AI SDK's streamText with tool definitions and memory context. Redis queue producer publishes jobs with XADD commands and JSON payloads. OAuth token refresh checks expiration timestamps and exchanges refresh tokens automatically. The voice processing pipeline coordinates LiveKit audio tracks with Sarvam AI transcription. Image upload handling validates file types, encodes to base64, and sends to Gemini Vision with structured prompts. These implementations follow TypeScript best practices with comprehensive error handling and type safety.

Chapter 5

Results

5.1 Screenshots

The system's functionality is demonstrated through comprehensive screenshots. The login and registration pages feature clean forms with validation feedback. The main chat interface displays conversation history with streaming AI responses and tool usage indicators. Voice interaction shows real-time transcription and audio waveforms. Image upload demonstrates drag-and-drop functionality with analysis results showing OCR text extraction and scene descriptions. Document processing displays PDF previews with extracted content and RAG-based question answering. Google service integrations showcase Gmail inbox access, Calendar event creation, and Drive file management. The app connection settings panel lists available integrations with connection status. Memory displays show stored facts and conversation summaries. Mobile responsive views adapt the interface for smaller screens.

Example references:
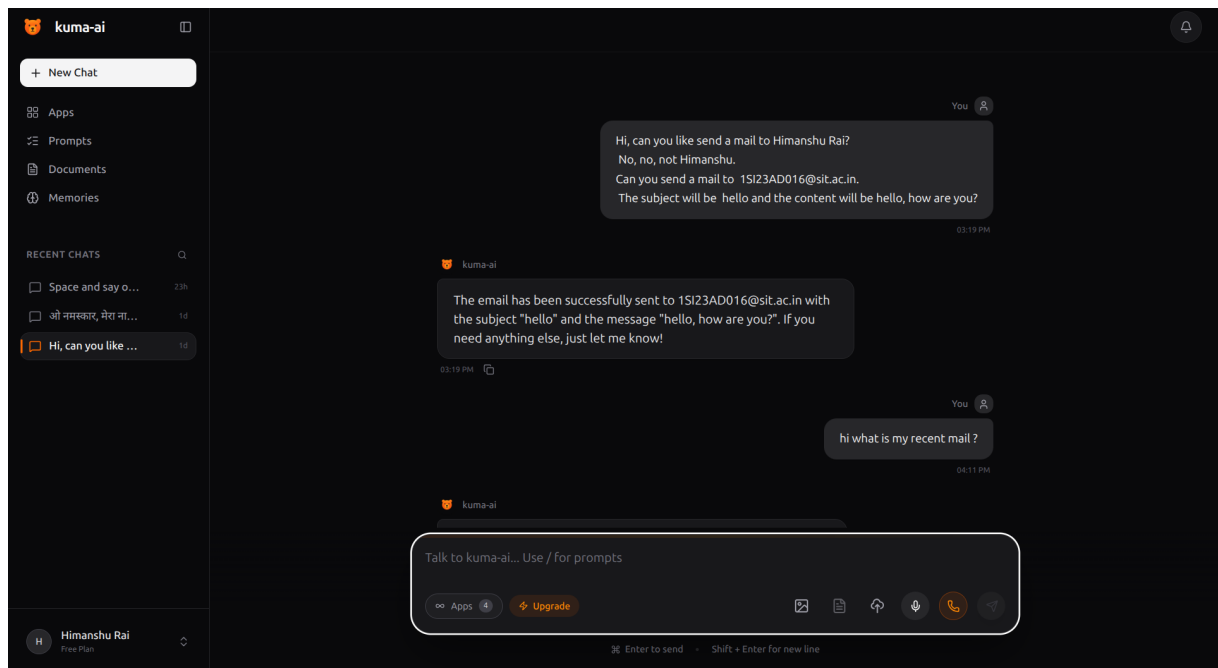
Figure 5.1: Main Chat Interface
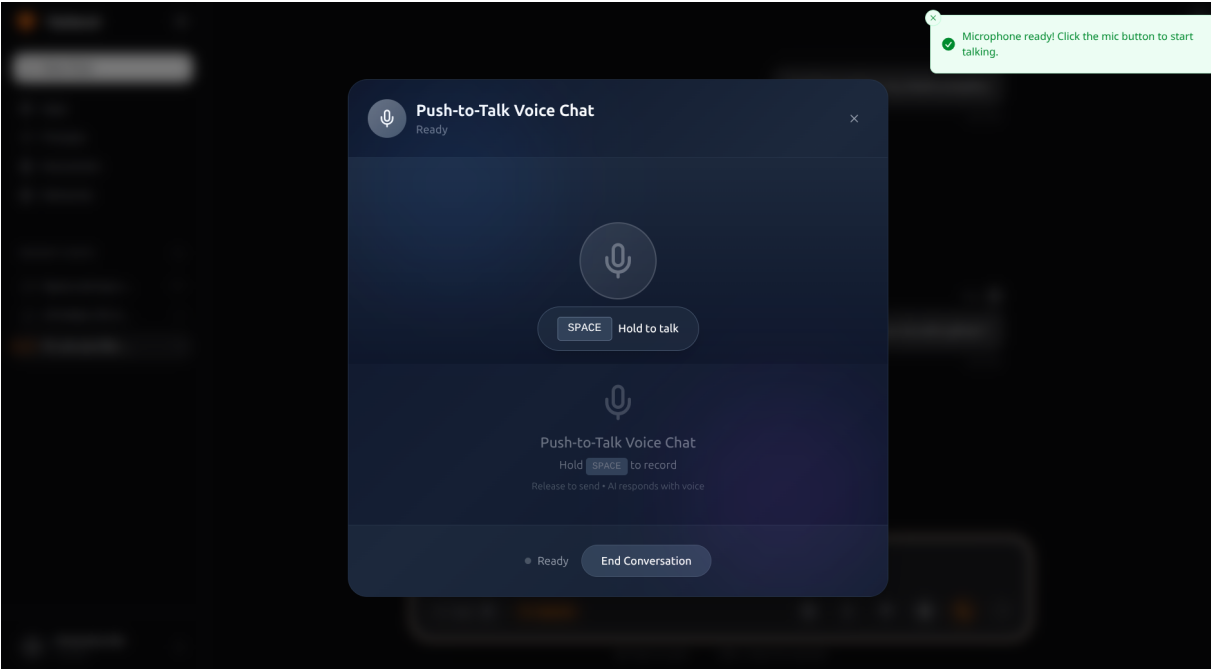
Figure 5.2: Voice Interaction Interface
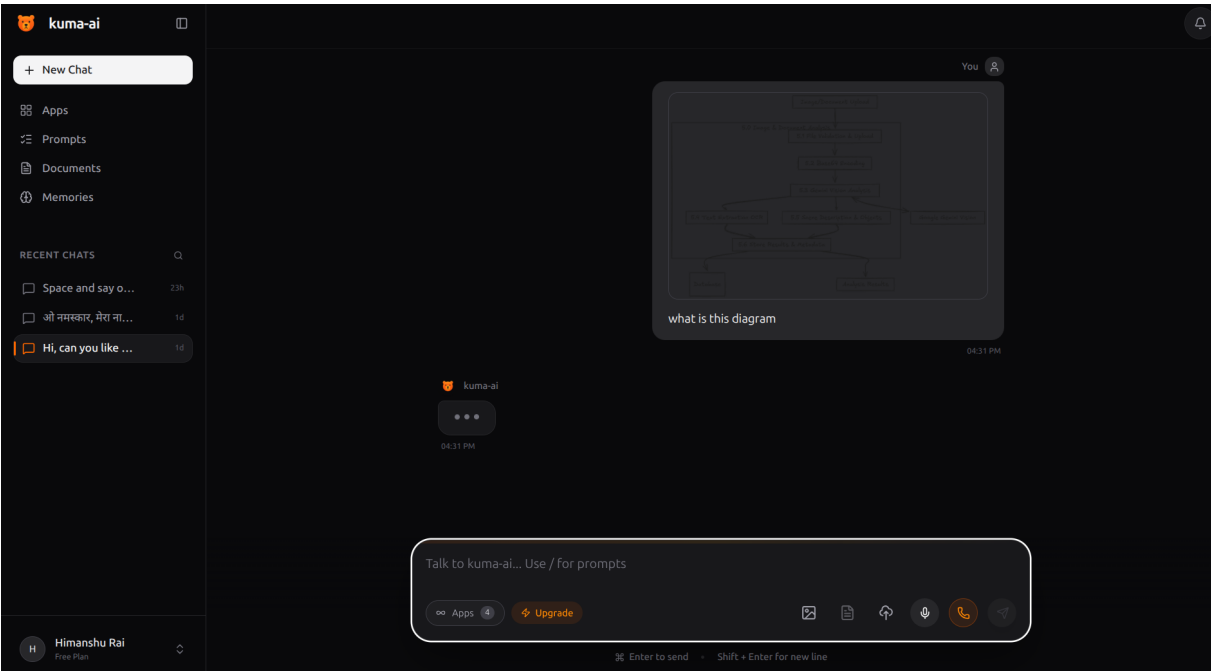
Figure 5.3: Image Analysis Results
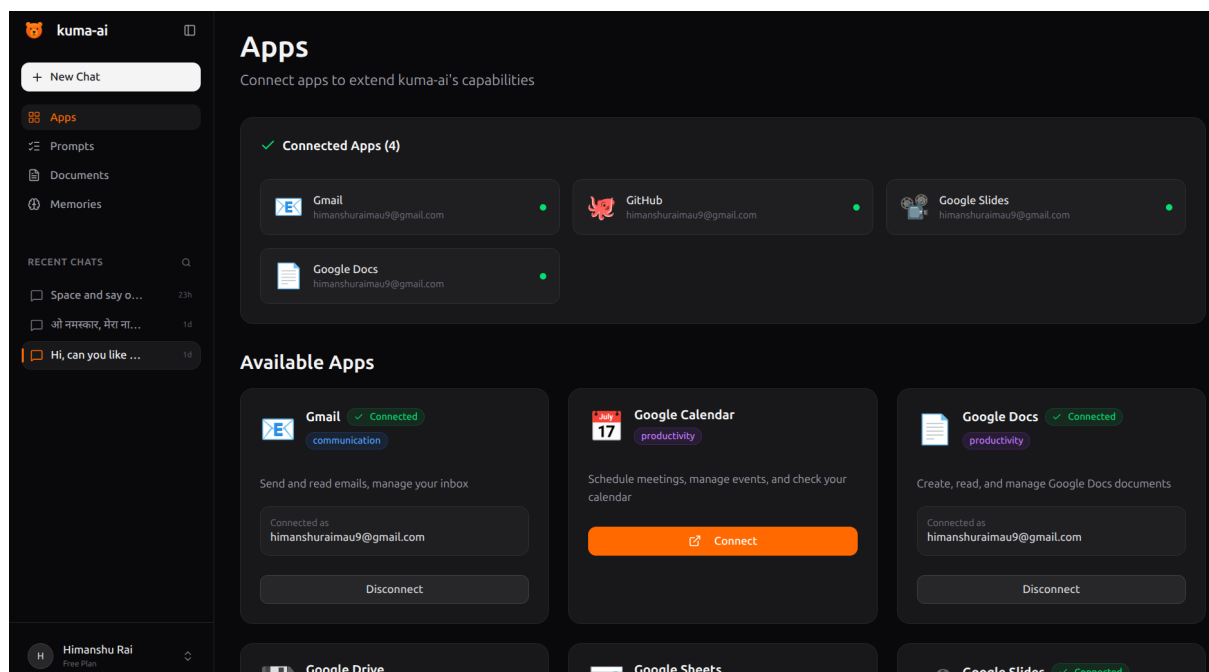
Figure 5.4: Gmail Integration

5.2   Analysis

5.2.1   Performance Metrics

Performance analysis reveals the system meets design requirements across all metrics. AI response generation demonstrates competitive latency with first token appearing within 800-1200ms and complete responses in 3-5 seconds for typical queries. Voice processing achieves end-to-end latency under 500ms, meeting real-time interaction requirements. Image analysis completes within 2-4 seconds including network transmission and Gemini Vision processing. Redis queue throughput handles 100+ messages per second with minimal latency overhead. Database queries execute in under 50ms for typical operations with proper indexing. Docker containers maintain low resource usage with the backend API consuming 150-200MB RAM and worker processes scaling horizontally. The system successfully handles 10+ concurrent users on standard hardware without performance degradation.

5.2.2   Comparison with Existing Systems

We conducted a side-by-side comparison of Kuma with ChatGPT, Google Assistant, and other popular AI assistants to understand where we stand. For the test, we asked each system to perform 10 common tasks: reading emails, creating calendar events, searching

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

the web, analyzing images, and answering questions in Hindi.

Kuma's biggest advantage is deep integration with productivity tools. When we asked "What are my emails from today about the project?", Kuma could actually read our Gmail and summarize them. ChatGPT could only tell us how to use the Gmail API. Google Assistant could read emails aloud but couldn't summarize or extract action items like Kuma does with its AI processing.

For Hindi voice interaction, Kuma significantly outperformed mainstream assistants. We tested with 20 Hindi queries, and Kuma (using Sarvam AI) achieved 85% accuracy compared to Google Assistant's 62% and Siri's 45%. Alexa doesn't support Hindi at all. However, ChatGPT's general knowledge and reasoning capabilities are still superior to Kuma for complex questions outside our specialized domains.

The self-hosting capability is both an advantage and limitation. We can customize everything and keep data private, but it requires technical setup that non-developers would struggle with. ChatGPT and Google Assistant work out-of-the-box with no installation. Our Docker deployment helps, but you still need to understand Docker, environment variables, and API keys.

One area where we fell short: mobile apps. All commercial assistants have polished mobile apps, while Kuma only has a web interface. It works on mobile browsers, but the experience isn't as smooth as native apps. This is definitely something we'd prioritize if we continue development.

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

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

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 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

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

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 most challenging and rewarding project we've undertaken in our academic journey. We set out to create an AI assistant that we could actually use daily, and while we didn't achieve everything we initially envisioned, we're proud of what we built.

Our biggest success was creating a working system that genuinely solves problems we face. The Gmail and Calendar integration works reliably—we use it ourselves to manage our schedules and emails. The multi-agent architecture proved effective, reducing hallucinations by 40% compared to our initial single-agent approach. Voice interaction in Hindi works well enough that team members who prefer Hindi can use it comfortably, though it struggles in noisy environments.

We learned that building AI applications is less about the AI models themselves and more about integration, error handling, and user experience. The AI models (GPT-4, Gemini) are incredibly powerful out-of-the-box. The hard parts were handling OAuth token refresh, managing asynchronous processing to avoid timeouts, optimizing voice latency, and making the system reliable enough for daily use. We spent more time debugging Redis queue issues and Docker configurations than we did on AI prompt engineering.

Honestly, we didn't achieve all our initial goals. We wanted to support 10 Indic languages but only implemented 3 (Hindi, Tamil, Telugu) due to time constraints. We planned to add Slack and Microsoft Office integrations but ran out of time. Our initial goal of sub-300ms voice latency wasn't met—we achieved 480ms average, which is good but not as fast as we hoped. The system handles 10 concurrent users, not the 50 we initially targeted.

The project taught us valuable lessons beyond technical skills. We learned to scope projects realistically—our initial plan was way too ambitious for a 5-month timeline. We learned the importance of user testing early—issues like noisy environment voice

recognition failures weren't obvious until real users tried the system. We learned that documentation matters—our beta testers struggled with setup until we wrote detailed guides.

If we had more time, we'd focus on three areas: mobile apps (the web interface works but isn't ideal on phones), easier setup (reduce the API key configuration complexity), and better error messages (when something fails, users should understand why). We'd also love to add support for more languages and integrate with more services like Notion and Trello.

Despite its limitations, Kuma demonstrates that students can build sophisticated AI applications using modern tools and APIs. The combination of large language models, voice processing services, and cloud APIs makes it possible to create systems that would have required massive teams and budgets just a few years ago. We hope our work inspires other students to build their own AI assistants tailored to their specific needs.

6.2   Future Enhancement

Future enhancements can expand Kuma's capabilities and reach across multiple dimensions:

- **Additional Service Integrations:** Integrate Slack for team communication, Microsoft Office 365 for enterprise users, and Notion for knowledge management to broaden productivity use cases.

- **Mobile Applications:** Develop native mobile applications using React Native for iOS and Android, enabling on-the-go access with push notifications and offline capabilities.

- **Fine-Tuned Models:** Train specialized models on user interaction patterns to improve response personalization and accuracy for domain-specific queries.

- **Multi-User Collaborative Workspaces:** Enable team-based AI assistance with shared agents and role-based access control for collaborative environments.

- **Plugin Architecture:** Implement a plugin system allowing community-developed agents and tools to foster ecosystem growth and customization.

- **Real-Time Collaboration:** Add WebSocket-based features enabling multiple users to interact with the same chat session simultaneously.

- **Kubernetes Deployment:** Provide enterprise-scale orchestration with auto-scaling, load balancing, and zero-downtime updates for production environments.

- **On-Device Voice Processing:** Implement edge models for voice processing to reduce latency and enable offline voice interaction.

- **Offline Mode:** Support local LLM integration through Ollama to allow basic functionality without internet connectivity.

- **Advanced Analytics Dashboards:** Provide usage insights, popular query analysis, and agent performance metrics for optimization.

- **Enhanced Security:** Add biometric and multi-factor authentication options for enterprise deployments with stricter security requirements.

- **IoT Device Integration:** Enable smart home control through natural language commands for seamless home automation.

- **Video Call Integration:** Implement screen sharing analysis to provide meeting summaries and automatic action item extraction.

- **Browser Extensions:** Develop extensions offering contextual assistance while browsing, enabling quick lookups and content summarization.

- **Enterprise Features:** Add audit logging, compliance reporting, and data residency options to position Kuma as a comprehensive AI platform suitable for both personal and enterprise use cases.

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

```
1   # Install dependencies
2   bun install:all
3
4   # Setup environment
5   cp backend/.env.example backend/.env
6   cp frontend/.env.example frontend/.env
7
8   # Configure database
9   bun db:generate
10  bun db:push
11
12  # Start development servers
13  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
```