

Ariyaan: A Personal Ai Assistant for Intelligent Conversation, Voice Interaction, and Productivity Support

Meganathan R¹, Ramkumar R², Indhuja N³

Department Of Artificial Intelligence And Data Science, Sengunthar Engineering College, Tiruchengode, Tamil Nadu, India

Abstract—Personal AI assistants are increasingly used to support information access, learning, programming, and everyday productivity. However, a useful assistant should combine natural-language interaction with accessible controls, conversation context, voice input and output, and a responsive interface. This paper presents ARIYAAN, a full-stack personal AI assistant designed using React, Vite, Node.js, Express, and the Google Gemini API through the official @google/genai SDK. The application supports conversational question answering, programming assistance, mathematical explanations, Markdown-formatted responses, code highlighting, multi-turn conversation context, and locally stored conversation sessions. Browser-native Web Speech APIs provide speech recognition and speech synthesis where supported by the user's browser. A voice-intent layer handles selected commands, including greetings, time and date requests, assistant identity questions, and desktop application launch requests. A settings interface supports model selection, voice preferences, speech rate, theme selection, and API connection testing. Gemini requests are routed through a backend service so that the API credential does not need to be embedded in frontend source code. The paper describes the system architecture, functional modules, implementation approach, and an evaluation plan covering response functionality, voice interaction, usability, security, and error handling. As measured performance data have not been supplied, this paper does not claim unverified accuracy, latency, or usability results. The proposed application is intended as an educational and productivity-support tool, with future work focused on persistence, accessibility, multilingual interaction, and stronger system-control safeguards.

Index Terms—ARIYAAN, Personal AI Assistant, Generative Artificial Intelligence, Google Gemini API, React, Node.js, Voice Assistant, Speech Recognition, Text-to-Speech, HumanComputer Interaction.

I. INTRODUCTION

Artificial intelligence has changed the way users search for information, learn technical concepts, write programs, and complete everyday digital tasks. Conversational AI systems allow users to express requests in natural language rather than navigating multiple menus or searching through many separate resources. These systems can provide explanations, generate text, assist with programming, and support problemsolving.

A personal AI assistant can bring several of these capabilities together in a single interface. For students, such an assistant may be useful for understanding academic topics, learning programming concepts, solving mathematical problems, and organizing questions. Voice interaction can provide an additional way to interact with the application, particularly when typing is inconvenient. A responsive design

is also important because users may access the application from desktop computers, laptops, tablets, and mobile devices. This paper presents ARIYAAN, a personal AI assistant built with React and Vite on the frontend and Node.js with Express on the backend. The system integrates Google's Gemini generative AI service through the official @google/genai SDK. The interface is designed to support

text-based conversations, multi-turn context, voice input and output, message-level audio playback, code formatting, conversationhistory, settings, and theme switching.

The name ARIYAAN is inspired by the Tamil concept Arivu + Udayaan = Arivudayaan (+ =), meaning one who possesses knowledge. This idea reflects the project's goal of providing useful knowledge and assistance through an interactive digital system. The brand motto is "From Knowledge to Understanding" ().

The major contributions of this project are:

1. A full-stack architecture connecting a responsive React interface to a server-side Gemini API integration.
2. A unified conversational interface for learning, programming, mathematics, and general information.
3. Browser-based speech recognition and speech synthesis integrated with text conversations.
4. A voice-intent layer for selected local commands and common conversational requests.
5. Conversation history, user preferences, theme switching, and error feedback designed for practical use.
6. An evaluation framework for testing functionality, usability, privacy, and reliability without assuming unmeasured results.

The remainder of this paper reviews related work, defines the problem, presents objectives and scope, describes the proposed system and methodology, outlines evaluation, discusses limitations and ethical considerations, and concludes with future enhancements.

II. RELATED WORK

Conversational agents have evolved from rule-based dialogue systems to neural language models capable of generating responses across many subject areas. Early conversational systems commonly relied on predefined patterns and narrow domains. Modern generative models can produce contextsensitive responses, although their output may still contain factual errors and requires appropriate user judgment.

Large language models provide a general interface for tasks such as question answering, summarization, code generation, and explanation. Google's Gemini API makes generative-model capabilities available to applications through programmatic requests [1]. Integrating such a service into a personal assistant requires attention to request formatting, conversation context, error handling, credential management, and service limits.

Web application frameworks also play an important role in the development of interactive assistants. React supports reusable user-interface components and state-driven rendering [2]. Node.js provides a JavaScript runtime for serverside application logic, while Express can be used to define HTTP routes and middleware [3], [4]. Separating frontend and backend responsibilities makes it possible to keep service credentials on the server and centralize request validation.

Voice interaction is another important area. The Web Speech API provides browser capabilities for speech recognition and speech synthesis, although browser support and behavior vary by platform [5]. Speech recognition converts spoken input into text, while speech synthesis reads generated text aloud. These capabilities can make an assistant easier to use, but they should not be treated as universally available or perfectly accurate.

ARIYAAN combines these established technologies in one application. Its focus is not on developing a new language model; instead, it is on integrating generative AI, browser voice facilities, selected intent handling, conversation management, and user-interface controls into a single personal assistant.

1 Problem Statement

Users often rely on separate applications for asking questions, learning programming, solving mathematical problems, listening to text, and accessing common desktop utilities. Switching between tools can interrupt a task and make the user experience less consistent. A text-only interface may also be inconvenient for users who prefer speaking or listening to responses.

A personal assistant that connects natural-language input to a generative AI service must handle several engineering challenges. It should maintain relevant conversation context, present generated answers clearly, provide predictable voice controls, respond appropriately when an API request fails, and avoid exposing secret API credentials in client-side code. It should also distinguish between conversational requests and local commands that require operating-system permissions.

The problem addressed by this project is the design and implementation of a responsive, full-stack AI assistant that combines text conversation, generative AI responses, optional voice interaction, selected command intents, and basic conversation management in a unified application. The system is designed to address the following requirements:

- Accept natural-language queries through a chat interface.
- Send supported requests to the Gemini service through a backend API.
- preserve conversation context for follow-up questions.
- Present answers with readable Markdown, formatted code, and copy controls.
- Support speech input and speech output where browser capabilities are available.
- Provide settings for model selection, voice preferences, theme, and API connection testing.
- Store conversation sessions and client preferences locally.
- Display understandable errors for missing credentials, invalid requests, service limits, and network failures.

2 Objectives And Scope

The primary objective is to develop ARIYAAN as an accessible personal AI assistant that supports learning, programming, mathematical problem-solving, and general productivity through text and optional voice interaction. The specific objectives are:

- To create a responsive user interface using React and Vite.

- To integrate the Google Gemini API using the official @google/genai SDK.
- To implement a Node.js and Express backend that mediates requests to the AI service.
- To support multi-turn conversations and locally stored chat sessions.
- To use browser-native speech recognition and speech synthesis for voice interaction where supported.
- To implement selected intents for greetings, time and date requests, identity questions, and desktop application launch requests.
- To provide voice, model, temperature, theme, and connection settings.
- To define practical tests for functional correctness, error handling, security, and usability.

The scope includes a web application and browser-supported voice features. It does not claim to train a new foundational language model. The voice dataset recorder collects user-recorded samples for later organization and possible future research; its existence alone does not mean that those samples have been used to fine-tune a model. Desktop application launching depends on the actual command-runner implementation, operating-system permissions, and security restrictions. The system should not execute arbitrary spoken commands without validation and explicit safeguards.

3. Proposed System

ARIYAAN uses a client-server design. The frontend manages the chat interface, user interactions, local preferences, conversation display, and browser voice APIs. The backend exposes application endpoints and sends supported AI requests to the Gemini API. The backend loads the API credential from environment configuration rather than embedding it in frontend code.

System Architecture

The overall workflow is represented in Fig. 1.

User



React + Vite User Interface

Chat, Settings, History, Theme, Voice Controls

4. Client Request And Intent Handling

Text Input / Speech-to-Text / Validated Local Intents

Node.js + Express Backend

a. Request Validation, Error Handling, Api Key Protection

Google Gemini API
Generated Response
Response Rendering

b. Markdown, Code Highlighting, Copy, Optional Text-To-Speech

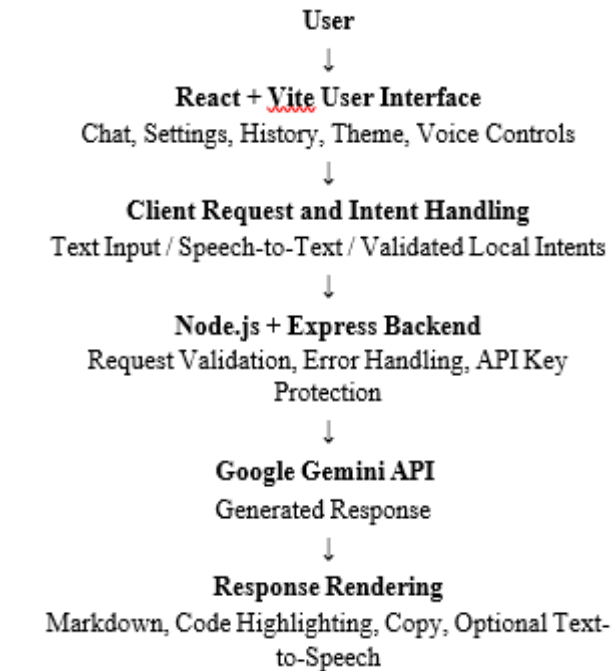


Fig. 1. High-level architecture of the ARIYAAN personal AI assistant.

A user submits a query through the chat interface or uses the microphone when speech recognition is supported. The application determines whether the request matches a supported local intent. Simple intents may be handled locally. General questions and complex requests are sent to the backend, which validates the request and calls the Gemini API. The generated response is returned to the frontend, rendered in the conversation, and optionally read aloud using speech synthesis.

Conversation context is included in requests according to the application's conversation-management implementation. Client-side conversation sessions and settings are stored in LocalStorage as described in the project design. LocalStorage is appropriate for basic browser persistence, but it is not a secure, synchronized, or server-backed database and can be cleared by the user or browser.

Functional Modules

- 1. User Interface and Chat Module:** Provides the welcome screen, suggestion cards, chat input, message display, loading indicators, and responsive layout. Suggested topics include programming, mathematics, problem-solving, and project architecture.
- 2. Gemini AI Integration Module:** Connects the backend to the Gemini API through @google/genai. It sends user prompts and relevant context to the configured model and returns generated responses. The model identifier should be configured and verified against the available API models.
- 3. Conversation Context and History Module:** Maintains relevant context for follow-up questions and supports conversation sessions. The interface can generate session titles, display saved sessions, and allow users to delete local history.
- 4. Voice Interaction Module:** Uses browser speech recognition for speech-to-text and speech synthesis for text-to-speech where supported. The microphone interface may indicate states such as idle, listening, processing, speaking, and error. Individual assistant messages can include controls to replay or stop audio.
- 5. Voice Intent and Command Module:** Recognizes selected phrases for greetings, assistant identity, time and date requests, and predefined desktop application launch requests. Local commands must use an allowlist of supported actions and must not accept arbitrary shell commands derived directly from speech.
- 6. Settings and Personalization Module:** Provides controls for available Gemini models, creativity temperature, voice input and output, available synthesized voices, speech rate, theme, and API connection testing. The application should validate settings before using them.
- 7. Response Formatting Module:** Displays generated text using Markdown, renders code blocks with syntax highlighting, and provides a copy-code action. Mathematical notation and code should remain readable across supported screen sizes.
- 8. Dataset Recorder Module:** Provides a standalone recorder page for collecting categorized personal voice samples. The supplied project description specifies more than 30 command prompts, progress persistence in LocalStorage, and downloaded WAV samples organized by category. The actual sample rate, channel format, encoding, and file integrity should be verified in implementation tests before these properties are reported as measured facts
- 9. Error Handling and Security Module:** Handles missing API credentials, invalid credentials, rate limits, network timeouts, empty input, and unsupported browser voice features. It keeps API credentials on the server and validates requests before calling external services.

Technology Stack

The proposed implementation uses the technologies summarized in Table I.

Table I Ariyaan Technology Stack

Layer	Technology / Role
Frontend	React 18 and Vite for the web inter-face
Styling	Vanilla CSS, design tokens, responsive layouts, glassmorphism
Icons	Lucide icon library
Response display	Marked and High-light.js
Backend	Node.js and Express REST API
AI service	Google Gemini API using @google/genai
Voice	Browser Web Speech APIs

Conversation Context

Let a conversation contain a sequence of messages:

React and Vite provide the frontend development environment. Vanilla CSS defines the visual design, including dark and light themes. Express provides server routes for frontend requests and AI integration. The Gemini service generates responses, while browser speech APIs provide speech recognition and synthesis independently of the language model. LocalStorage persists selected browser-side data between sessions.

III. METHODOLOGY

The development methodology consists of requirement analysis, interface design, backend implementation, AI integration, voice-intent handling, persistence, and testing. Each stage has a distinct responsibility and should be validated before the application is demonstrated.

1 Requirement Analysis

The main user requirements are grouped into conversational assistance, programming and mathematics support, voice interaction, settings, session history, and error handling.

Requirements are mapped to functional modules so that each feature can be tested independently.

2 Frontend Development

The frontend is implemented using React components and Vite. The chat view displays user messages and AI responses. The settings interface manages the selected model, temperature, theme, voice toggles, voice choice, and speech rate. Responsive CSS should support desktop, tablet, and mobile layouts. Markdown rendering and code highlighting are applied to AI responses, and copy controls provide a practical way to reuse code snippets.

3 Backend And Api Integration

The Express backend receives frontend requests and forwards supported prompts to the Gemini API. The API key is read from an environment variable, for example:

GEMINI_API_KEY=your_actual_api_key PORT=5000

The values above are placeholders and must be configured locally. The backend should validate that input is present, restrict payload size, handle upstream failures, and return consistent error messages. Secrets must not be committed to a public repository. CORS should be configured for the intended frontend origin rather than opened broadly without a reason.

$C_t = \{m_1, m_2, \dots, m_t\}, (1)$

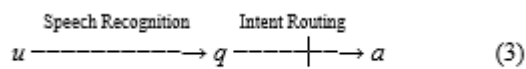
where m_i represents a user or assistant message and t is the current turn. For a follow-up query q_{t+1} , the application can provide a relevant portion of C_t together with the new query to the model:

$r_{t+1} = G(C_t, q_{t+1}, \theta), (2)$

where G denotes the configured generative model, θ represents model settings, and r_{t+1} is the generated response. The practical context window and exact message format depend on the model API and implementation. Older messages may need to be truncated or summarized to control request size.

4, Voice Processing

Voice interaction has two main stages. Speech-to-text converts an utterance into text, which is then routed to the intent handler or chat pipeline. Text-to-speech converts the response into audio when voice output is enabled.



5. Speech Recognition Intent Routing

Here, u is the spoken utterance, q is the recognized text, and a is the resulting action or AI response. For generative requests, the response is received from the backend before optional speech synthesis. Browser support, microphone

permission, network conditions, and language configuration can affect the experience.

6. Intent Handling

The intent layer identifies only predefined request categories. Examples include greetings, time and date queries, identity questions, and allowlisted application-launch commands. Intent matching should normalize text carefully and handle ambiguous phrases without triggering unsafe actions. Application-launch operations require an explicit mapping from recognized intent to a known executable or safe operating-system action. Raw user speech must never be treated as an unrestricted terminal command.

7. Local Persistence

The project uses LocalStorage for browser-side session history and client settings. Data can be serialized into JSON for storage and restored when the application opens. Because LocalStorage is accessible to scripts running in the same origin and is not encrypted by default, sensitive information such as API keys, passwords, or confidential conversation content should not be stored there without a suitable security design. Users should be able to clear stored sessions and understand that browser-local data may not synchronize across devices.

8. Error Handling

The application should handle empty input before submission and distinguish failures such as missing configuration, rejected credentials, rate limiting, server errors, and network timeouts. Error messages should guide the user without exposing secret keys, internal stack traces, or unnecessary infrastructure details. The frontend should also detect unavailable voice features and provide a text-based alternative.

9. Implementation And Configuration

The project description specifies separate frontend and backend folders and npm scripts for setup, development, and production. The expected development workflow is summarized below; actual script names must match the supplied package.json files.

1. Install dependencies from the project root using the configured setup script.
2. Configure the backend environment file with a valid Gemini API key and the desired port.
3. Start the Express backend, described in the project configuration as using port 5000.
4. Start the Vite frontend, described in the project configuration as using port 5173.
5. Verify that frontend requests reach the backend and that backend responses are rendered correctly.

6. Build the frontend and start the production server using the configured npm scripts.

The supplied project description lists the following commands:

```
npm run setup npm run dev:backend npm run dev:frontend  
npm run build:frontend npm start
```

Functional Test Cases

Table II presents representative test cases.

A. Evaluation Metrics

For response time, a test can record the elapsed time between submission and receipt of a completed response:

$T_{response} = t_{received} - t_{submitted}$. (4)

Response time depends on network quality, server processing, model selection, request size, and service load. It should be measured over repeated trials and summarized using appropriate statistics such as median and range. Functional pass rate can be calculated as:

$\frac{10 \text{ Npassed}}{11 \text{ Nexecuted}}$

$P_{pass} = \frac{10 \text{ Npassed}}{11 \text{ Nexecuted}} \times 100\%$. (5)

11 Nexecuted

This metric summarizes the proportion of executed test cases that meet their expected outcomes. It does not measure the factual correctness of all AI-generated content. Content quality should be assessed separately using a defined rubric, representative prompts, and human review.

A. Results Reporting Template

Table III is provided for completion after actual tests have been run.

No numerical results should be inserted until testing has been performed. Testing should include different query lengths, repeated follow-up questions, unsupported browser capabilities, API failures, and relevant desktop configurations. A model's generated answer should be checked for factual correctness rather than judged only by whether a response was returned.

The described local development addresses are http:

http://localhost:5000 for the backend and http://localhost:5173 for the frontend. The production application is intended to be served through the backend at port 5000. These addresses and commands should be checked against the actual project configuration before final submission.

The voice dataset recorder is described as accessible at http://localhost:5173/recorder. It allows users to record

categorized samples and download them for later organization. The recorder should clearly request microphone permission, indicate recording status, and avoid recording without user action.

3.12 Evaluation Framework

ARIYAAN should be evaluated through functional testing, integration testing, voice testing, security review, and usability assessment. The project description does not include measured experimental data; therefore, the test cases and tables in this section are templates for reporting actual results rather than evidence that the tests have already passed.

IV. DISCUSSION

ARIYAAN integrates several capabilities that are commonly separated across applications. A conversational interface can help users move between general questions, learning tasks, programming support, and mathematical explanations without changing tools. Multi-turn context supports followup questions, while Markdown and code highlighting make responses easier to read and reuse.

Voice input and output provide an alternative interaction mode. Their usefulness depends on microphone permissions, browser support, speech-recognition quality, voice availability, and environmental noise. Browser-native speech features can vary between browsers and operating systems, so the application should retain a fully usable text interface when voice features are unavailable.

The server-side API proxy is an important architectural decision because it prevents the Gemini credential from being placed directly in frontend source code. This protection is not sufficient on its own: the backend still requires input validation, appropriate rate limiting, careful error handling, and secure deployment configuration. If the application is exposed to other users, authentication and per-user controls may be required to prevent unauthorized usage and excessive API costs.

The local command feature is useful for demonstrating an assistant that can connect conversation to predefined desktop actions. It also introduces security risks if implemented too broadly. Commands should be restricted to a fixed allowlist, validated independently of the AI response, and protected against arbitrary shell execution. Users should be clearly informed when an action affects their device.

LocalStorage supports simple session persistence without requiring a database server. However, it does not provide multi-device synchronization, robust account management, or server-side backups. The project should therefore describe this as local browser persistence rather than a full databasebacked conversation service.

TABLE II Recommended Functional Test Cases

ID	Test	Input / Action	Expected Behaviour	Status
T1	General question	Ask “What is artificial intelligence?”	A readable response is returned through the backend and displayed in chat.	To test
T2	Follow-up context	Ask “Explain it with an example” after T1	The response should use the preceding topic when conversation context is enabled.	To test
T3	Code generation	Request a Python prime-number program	The response should render code with syntax highlighting and a copy control.	To test
T4	Mathematics	Ask to solve 25×48 step by step	The response should include a calculation and explanation.	To test
T5	Missing key	Start the backend without valid API configuration	A friendly configuration error should be shown without exposing secrets.	To test
T6	Voice greeting	Say “Hello ARIYAAN” with microphone enabled	Speech is recognized and the greeting intent or chat pipeline responds.	To test
T7	Voice output	Enable speech output and request an explanation	The response is read aloud when browser speech synthesis is available.	To test
T8	Settings	Change theme, model, and speech rate	Supported preferences are applied and persist as designed.	To test
T9	Local command	Say “Open Calculator” on a supported desktop configuration	Only the predefined, allowlisted action is triggered if configured and permitted.	To test
T10	Invalid input	Submit an empty message or simulate a network failure	The request is blocked or a clear error is displayed.	To test

TABLE III Evaluation Results Template

Metric	Measured Value
Tests executed	[Enter value]
Tests passed	[Enter value]
Functional pass rate	[Calculate value]
Median response time	[Measure value]
Voice recognition success	[Measure value]
Voice synthesis success	[Measure value]
Usability feedback	[Summarize findings]

Limitations And Ethical Considerations

The application depends on an external generative AI service. Generated answers may be incomplete, outdated, misleading, or incorrect. Users should verify important

technical, academic, and mathematical information rather than treating every response as authoritative.

The availability and behavior of Gemini models can change over time. Model identifiers, API limits, pricing, and supported features should be checked against the current provider documentation. The paper therefore refers to configured Gemini models rather than asserting that a particular model identifier will remain available indefinitely.

Voice capabilities depend on browser implementation and permissions. Speech recognition can misinterpret words, especially with accents, background noise, or multilingual speech. A recognized command should not automatically be trusted when it would cause a consequential action.

API keys are sensitive credentials and must remain on the server. Environment files should be excluded from version control, and keys accidentally published to a repository should be revoked and replaced. Production deployments should use HTTPS, secure environment configuration, request validation, and appropriate access controls.

Conversation history and recorded voice samples may contain personal information. Users should know where data is stored, how to delete it, and whether it is sent to external services. Voice samples should be collected with consent and handled carefully. The dataset recorder should not be described as training a model unless an actual training or fine-tuning process has been carried out and evaluated.

Finally, an AI assistant should not replace professional advice or independent judgment in high-impact situations. The system should communicate its limitations and encourage users to verify important outputs.

Future Enhancements

Several enhancements can extend ARIYAAN beyond its current project scope:

- Multilingual support: Improve Tamil and other language interaction, including recognition, response generation, and speech synthesis, with explicit language selection and testing.
- Mobile application: Develop a dedicated mobile version or installable progressive web application with a touch-friendly interface and suitable permission handling.
- Persistent database: Add a secure database and optional user accounts for managed conversation history and cross-device synchronization.
- Improved memory controls: Allow users to inspect, edit, clear, and selectively retain conversation context.
- Safer desktop integration: Add a carefully permissioned command registry, confirmation prompts for sensitive actions, and audit logs for local actions.
- Voice dataset research: Validate recorded sample format, improve metadata and consent management, and investigate whether the dataset is suitable for a clearly defined speech-recognition research task.
- Retrieval-augmented generation: Allow users to ask questions about their own authorized documents using a retrieval pipeline, with source references and clear limits.
- Accessibility improvements: Improve keyboard navigation, screen-reader labels, contrast, focus indicators, and adjustable text sizes.
- Monitoring and evaluation: Record privacy-conscious operational metrics such as response time and error rates, and periodically test answer quality and robustness.
- Deployment hardening: Add rate limiting, request size limits, security headers, monitoring, and production configuration checks.

V. CONCLUSION

This paper presented ARIYAAN, a full-stack personal AI assistant designed to support intelligent conversation, learning, programming assistance, mathematical problem solving, and productivity. The proposed system combines a React and Vite frontend with a Node.js and Express backend and integrates the Google Gemini API through the official @google/genai SDK.

The application design includes multi-turn conversation context, locally stored chat sessions, Markdown and code formatting, browser-based speech recognition and synthesis, message-level audio playback, settings, theme switching, and selected voice intents. The backend-mediated API design helps keep secret credentials out of frontend source code. The project also includes a concept for collecting categorized voice samples for possible future research.

The paper described the system architecture, functional modules, implementation workflow, and a test plan for evaluating core features. Since measured test results were not provided, no unsupported performance figures or claims of experimental success have been made. The final evaluation should report actual functional pass rates, response-time measurements, voice behavior, and user feedback collected under documented conditions.

ARIYAAN demonstrates how generative AI and web technologies can be integrated into a practical personal assistant. Its future development should prioritize reliability, security, user control, multilingual accessibility, responsible voice-command execution, and transparent evaluation. The intended goal is to help users move from access to information toward clearer understanding, consistent with the motto “From Knowledge to Understanding.”

REFERENCES

1. Google, “Gemini API documentation,” Google AI for Developers. [Online]. Available: <https://ai.google.dev/gemini-api/docs>. Accessed: Oct. 11, 2026.
2. Meta Platforms, “React documentation.” [Online]. Available: <https://react.dev/>. Accessed: Oct. 11, 2026.
3. OpenJS Foundation, “Node.js documentation.” [Online]. Available: <https://nodejs.org/docs/latest/api/>. Accessed: Oct. 11, 2026.
4. OpenJS Foundation, “Express documentation.” [Online]. Available: <https://expressjs.com/>. Accessed: Oct. 11, 2026.
5. MDN Web Docs, “Web Speech API.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Speech_API. Accessed: Oct. 11, 2026.

6. Vite, “Vite documentation.” [Online]. Available: <https://vite.dev/guide/>. Accessed: Oct. 11, 2026.
7. Marked, “Marked documentation.” [Online]. Available: <https://marked.js.org/>. Accessed: Oct. 11, 2026.
8. Highlight.js, “Highlight.js documentation.” [Online]. Available: <https://highlightjs.org/>. Accessed: Oct. 11, 2026.