

1. INTRODUCTION

1.1 OVERVIEW

In today's fast-paced digital environment, composing clear and effective emails can be a time-consuming task. The **Smart Email Assistant** is an intelligent, AI-powered application built using **Spring Boot** and **Spring AI**, designed to streamline email writing by automatically generating professional and context-aware email content.

This project integrates **Google's Gemini AI model** through **Spring AI**, enabling the assistant to understand user prompts and generate high-quality, natural language responses. Users can simply input a brief instruction or purpose—such as "reschedule the meeting to next Monday"—and the assistant will return a polished, ready-to-send email.

The backend is developed with **Java 23 and Spring Boot**, offering a clean, modular architecture and robust REST API. The use of **Spring AI with Gemini** ensures seamless communication with the AI model while keeping the codebase extensible and production-ready.

This Smart Email Assistant showcases the practical integration of generative AI with modern Java frameworks, offering a powerful productivity tool for users in corporate, academic, or personal settings. It highlights how AI can be used to automate routine communication tasks, save time, and ensure professional tone and clarity in emails.

1.2 OBJECTIVE

The primary objective of the **Smart Email Assistant** is to develop a secure, efficient, and intelligent platform that automates the process of composing professional emails, enabling users to generate clear and contextually relevant email content with ease. The system leverages advanced AI technology to understand user prompts and deliver accurate, polished email drafts, enhancing communication productivity. The detailed objectives are:

- **Automate Email Composition:** Enable users to generate professional and context-aware emails automatically, reducing the time and effort required for writing.
- **Integrate Advanced AI:** Utilize Google's Gemini AI model through Spring AI to provide accurate natural language understanding and generation.

- **Provide Easy Accessibility:** Offer a RESTful API interface that allows seamless integration with other applications or frontends.
- **Ensure Accuracy and Clarity:** Generate grammatically correct and well-structured emails to improve communication quality.
- **Enhance Productivity:** Streamline routine email writing tasks to save users' time and improve efficiency.
- **Build a Scalable Backend:** Develop a robust and maintainable backend application using Java and Spring Boot frameworks to support future enhancements and increased user load.

1.3 NEED OF PROJECT

The need for a **Smart Email Assistant** arises from the growing demand for faster, more efficient, and intelligent tools to manage email communication in both professional and personal settings. Traditional email writing can be time-consuming, repetitive, and prone to errors, especially when composing multiple emails daily. By leveraging AI technology, this system addresses these challenges by automating email creation, enhancing productivity, and improving communication quality.

- **Efficient Email Generation:** Automates the process of composing professional and contextually relevant emails, saving users significant time and effort.
- **Improved Accuracy:** Uses advanced AI models to generate grammatically correct and well-structured emails, reducing errors and misunderstandings.
- **User-Friendly Interface:** Provides a simple API-based interface, allowing easy integration with various applications and platforms.
- **Enhanced Productivity:** Enables users to focus on critical tasks by minimizing the repetitive work involved in drafting emails.
- **Cost-Effective Solution:** Reduces the need for manual proofreading and editing, saving both time and operational costs for businesses.
- **Supports Modern Communication Needs:** Adapts to different email tones and contexts, ensuring communication remains relevant and professional.

1.4 SCOPE

The scope of the **Smart Email Assistant** project defines its core functionalities, target users, and possibilities for future improvements. This project focuses on delivering a reliable backend application with AI-powered email generation capabilities, featuring the following functionalities and benefits:

Functional Scope

- Integration of Google Gemini AI for natural language understanding and email content generation.
- A REST API that accepts user prompts and returns contextually relevant, professionally formatted emails.
- Support for multiple users to generate emails simultaneously via API calls.
- Secure handling of user inputs to ensure privacy and data protection.
- Basic error handling and validation to manage invalid or incomplete user requests.
- Scalability to accommodate increasing user demands and advanced AI model updates.
- Extensibility for future enhancements such as multi-language support, email scheduling, and integration with email clients.

Target Audience

- **End Users:**
 - Professionals and individuals who need quick, accurate, and context-aware email generation.
 - Users seeking to save time by automating routine email writing tasks.
 - People looking for improved email quality with minimal effort.
- **Business Users:**
 - Companies and organizations aiming to enhance employee productivity by integrating AI-powered email assistants.

- Customer support teams and sales departments that require consistent and professional email communication.
- Teams needing scalable email generation solutions for high-volume communication.
- **Developers and System Administrators:**
 - Developers who want to integrate AI email generation into existing applications through REST APIs.
 - System administrators focusing on ensuring the backend's scalability, security, and maintainability.
 - Teams responsible for updating AI models and optimizing system performance.

2. FEASIBILITY STUDY

A feasibility study was conducted to evaluate the practicality of developing the Smart Email Assistant. This study assesses whether the proposed AI-powered email generation system meets user requirements, technical constraints, and operational goals, while also considering potential for future enhancements. The feasibility study includes the following aspects:

2.1 TECHNICAL FEASIBILITY

Technical feasibility evaluates the technologies, tools, and expertise required to develop the Smart Email Assistant.

- **Technology Requirements:** The project uses Java 23, Spring Boot framework, Spring AI for AI integration, and Google Gemini API for natural language processing. The backend exposes REST APIs for interaction, and Maven is used for project management.
- **System Architecture:**
 - **Client-Server Architecture:** The system operates with a backend REST API server that processes user prompts and generates email content using AI models.
 - **AI Integration:** Utilizes Google Gemini AI API for natural language understanding and generation.
 - **Database Design:** Optional storage for user prompts, generated emails, and usage logs (if implemented).
- **Development Resources:** The project leverages widely used frameworks and APIs supported by active developer communities. The development team possesses the necessary skills in Java, Spring Boot, and AI API integration.
- **Scalability:** The system is designed to handle multiple simultaneous user requests and can scale by increasing server resources or leveraging cloud services as demand grows.

2.2 OPERATIONAL FEASIBILITY

Operational feasibility ensures the system can function effectively and meet user needs.

- **User Experience:** The system provides a simple API interface that can be integrated into various client applications, enabling easy use for developers and end-users.
- **Support and Maintenance:** The modular design and thorough documentation support easy maintenance and updates of AI models and system components.
- **Staffing Requirements:** Minimal technical staff is needed to maintain backend services and handle API support, with automated AI handling most email generation tasks.

2.3 ECONOMIC FEASIBILITY

Economic feasibility evaluates the cost-effectiveness and financial viability of developing and deploying the Smart Email Assistant system.

- **Development Costs:** The project primarily requires investment in software development, including developer time for backend development, AI integration, and testing. Using open-source frameworks like Spring Boot and leveraging cloud-based AI services like Google Gemini API helps minimize infrastructure costs.
- **Operational Costs:** Ongoing expenses include cloud hosting fees, API usage charges for Google Gemini, maintenance, and periodic updates. These costs are expected to be manageable due to scalable cloud services and pay-as-you-go AI APIs.
- **Cost Savings:** Automating email generation reduces manual effort and increases productivity, resulting in significant time savings for users and organizations. This can translate to lower labor costs and faster communication cycles.
- **Return on Investment (ROI):** The system's ability to improve efficiency and reduce human error in email communication can provide a strong ROI by enabling users to focus on higher-value tasks, improving overall business performance.
- **Scalability and Future Expansion:** The project's design supports scalability, allowing it to grow with user demand without requiring major re-investment, thus optimizing long-term cost management.

3. SYSTEM REQUIREMENTS SPECIFICATION

3.1 HARDWARE REQUIREMENTS

The hardware requirements for the Smart Email Assistant project, which primarily runs as a backend service with AI integration and REST APIs, are as follows:

- **Processor (CPU):** Quad-core processor or higher (e.g., Intel i5 or AMD Ryzen 5) to efficiently handle AI requests and backend processing.
- **Memory (RAM):** At least 16 GB of RAM to support the Java Spring Boot application and AI model interactions smoothly.
- **Storage:** Minimum 256 GB SSD storage to store application files, logs, and any cached AI data for fast access.
- **Network:** Reliable high-speed internet connection is required for seamless communication with external AI APIs (e.g., Google Gemini API).
- **Operating System:** Windows 10/11, Linux, or macOS — any modern OS that supports Java 17 and Spring Boot development.

3.2 SOFTWARE REQUIREMENTS

The software requirements for the Smart Email Assistant project focus on the tools and technologies necessary for development, integration, and operation of the AI-powered email assistant.

3.2.1 Backend Development Tools

- **IntelliJ IDEA IDE:** Used as the primary integrated development environment for Java and Spring Boot development. IntelliJ IDEA Community Edition 2024.1.1 (or latest) is recommended.
- **JDK (Java Development Kit):** Java 17 or higher is required for running Spring Boot and other Java-based components efficiently.
- **Spring Boot Framework:** Provides the framework for building the RESTful backend API and managing application components.

- **Spring AI:** Used for integrating AI capabilities, including natural language processing and generation through external APIs.
- **Google Gemini API:** Utilized to provide AI-powered language models that generate email replies and smart suggestions.
- **Maven or Gradle:** Build automation tools to manage dependencies and project build lifecycle.

3.2.2 Frontend Development Tools

- **React.js:** Used to build the frontend user interface of the email assistant application.
- **Visual Studio Code (VS Code):** The primary software used for writing React.js code. VS Code is a lightweight, powerful editor widely used in the React developer community, offering extensions and tools that streamline React development.
- **Postman:** For API testing and debugging backend endpoints during development.

3.2.3 API Connectivity and Integration

1. Overview:

The backend communicates with the Google Gemini AI API to process natural language input and generate relevant email replies. This involves sending user emails or commands to the AI service and receiving AI-generated responses.

2. Key Technologies:

- **RESTful APIs:** Used for communication between frontend and backend services as well as between backend and external AI services.
- **HTTP Client (e.g., WebClient or RestTemplate in Spring Boot):** Facilitates making HTTP requests to Google Gemini API.
- **JSON:** Data exchange format for API requests and responses.

3.3 FUNCTIONAL REQUIREMENTS

The functional requirements define the core capabilities that the **Smart Email Assistant** must deliver to users across UI, AI integration, API handling, and security.

3.3.1 User Interface

- Users can input the content of an email or message for which they want a reply.
- The application offers a clean, user-friendly interface built using **React.js** and **Material UI**.
- Users can optionally choose a **reply tone** (e.g., professional, casual, friendly).
- The generated email reply is displayed in real time with options to **copy the text** or reuse it.
- The interface is fully responsive and integrates seamlessly with **Gmail via Chrome Extension**.

3.3.2 AI-Powered Email Reply Generation

- The frontend sends the user input (email + tone) to the backend API.
- The backend invokes the **Gemini API** (via Spring AI) to process the email context.
- A relevant, context-aware reply is generated using AI and sent back to the UI.
- Replies are generated quickly and adapt to the selected tone for better personalization.

3.3.3 API Integration and Handling

- Backend securely integrates with **Google's Gemini API** using **Spring AI**.
- **Spring Web & WebFlux** ensure support for both standard and reactive REST communication.
- Implements **error handling**, fallback responses, and logs for failed API requests.
- Ensures **scalability** with support for multiple users and parallel request handling.

- Uses **Lombok** for clean and efficient backend code.

3.3.4 Security and Data Handling

- Input validation is implemented to prevent **XSS, injection attacks**, or malformed data.
- API keys and sensitive configs are managed via environment variables (never exposed to frontend).
- Follows secure **CORS policies** and uses **HTTPS** in production deployment.
- Chrome extension only accesses Gmail UI and API endpoint on localhost:8080 (or production host).

3.4 NON-FUNCTIONAL REQUIREMENTS

The non-functional requirements describe the performance standards, quality attributes, usability expectations, and operational constraints of the **Smart Email Assistant** system.

3.4.1 Usability Requirements

- The system provides a **clean and intuitive UI** built with **React.js** and **Material UI**, ensuring a smooth user experience.
- Clear visual feedback is provided during all stages (e.g., “Generating reply...”, “Error: Please try again”).
- Chrome Extension offers **seamless integration** within Gmail, injecting the “AI Reply” button into the compose window for easy access.

3.4.2 Performance Requirements

- **Response Time:** The system should return AI-generated replies within **2–3 seconds** in most cases.
- **Throughput:** Must support **10+ concurrent users** with stable performance during peak loads.
- **Scalability:** Backend (Spring Boot with WebFlux) is designed to **scale horizontally** using load balancers and containerized deployments (e.g., Docker/Kubernetes).
- **Memory Optimization:** Efficient memory usage is ensured on both client and server sides to prevent lags or crashes during heavy use.

3.4.3 Availability and Reliability

- **Availability:** Target uptime is **99.9%**, ensuring the assistant is available when users need it.
- **Resilience:** In case of external API (Gemini) failures, fallback mechanisms (e.g., retry logic or default responses) are triggered.

- Uses **circuit breaker** patterns and graceful error handling (Spring Retry / WebClient resilience tools).
- **Monitoring tools** (e.g., Spring Actuator, Prometheus) may be used for real-time system health checks.

3.4.4 Maintainability

- The system architecture follows a **modular and layered approach** (Controller → Service → AI Layer → API Layer).
- Code is written with **clean code principles** and uses **Lombok** to reduce boilerplate.
- Follows **RESTful design** for easier integration with external tools or mobile versions in the future.
- Well-documented codebase and comments for developers; includes README, API docs (e.g., Swagger), and setup instructions.

3.4.5 Security and Privacy

- **API keys and secrets** are managed using secure environment variables (.env, config servers) and never exposed in the frontend or browser.
- All user inputs are **sanitized and validated** on both frontend and backend to prevent:
 - Cross-site scripting (XSS)
 - SQL injection (in future data-enabled versions)
 - Malformed or abusive requests
- Uses **HTTPS protocol** in production for end-to-end encryption.
- No sensitive user data (email content or replies) is stored—ensuring **zero data retention policy**.
- Follows **Content Security Policy (CSP)** and **permissions minimization** in Chrome Extension manifest.

4. DESIGN

4.1 ER-DIAGRAM

An ER Diagram is used to visually represent the key components and interactions within the Smart Email Assistant system. Although this project does not utilize a traditional relational database, the ER diagram is adapted to model the logical relationships between core conceptual entities such as **User**, **Smart Email Assistant**, **Gemini API**, and **Generated Reply**. This diagram captures how:

- A **User** provides input such as their email and message tone.
- The **Smart Email Assistant** component processes this input, sends it to the **Gemini API**, and receives a response.
- The **Gemini API** (LLM) uses parameters like prompt templates, temperature, and model version to generate a context-aware response.
- A **Generated Reply** is created and linked back to both the User and the Assistant component.

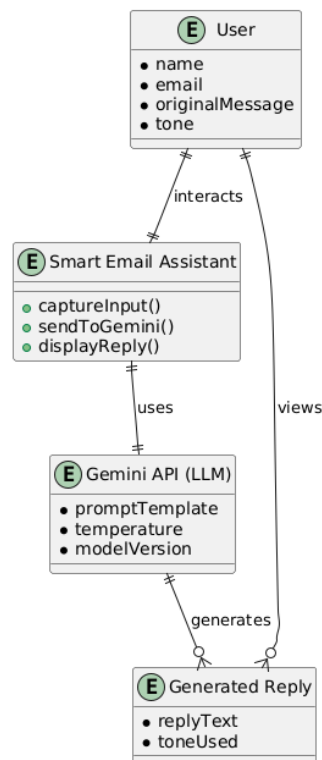


Figure - 1

4.2 DATA FLOW DIAGRAM

1. **Level 0 DFD:** Shows the overall system as a single process interacting with the user who provides email and tone, and receives the generated reply.

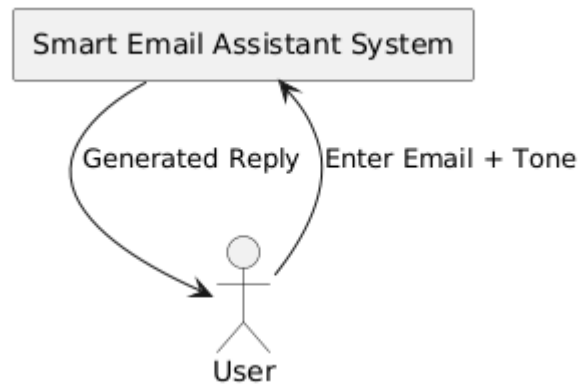


Figure – 2.1

2. **Level 1 DFD:** Breaks down the system into main processes: capturing user input, sending it to Gemini API, and displaying the reply to the user.

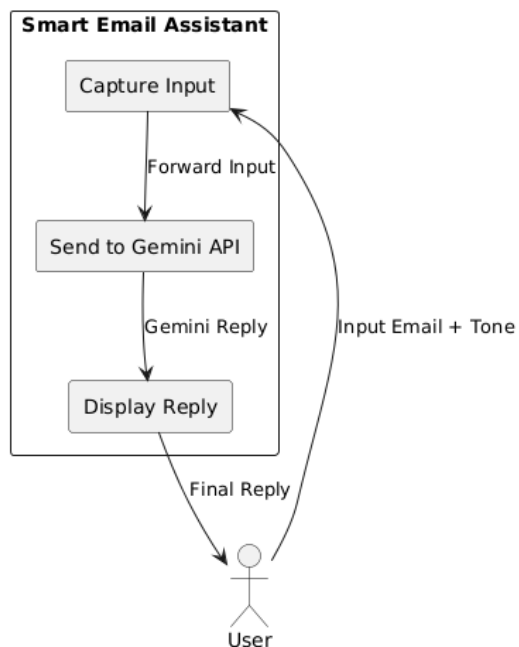


Figure – 2.2

3. **Level 2 DFD:** Details the "Send to Gemini API" process by showing steps like preparing the prompt, setting parameters, calling the API, parsing the response, and handling errors.

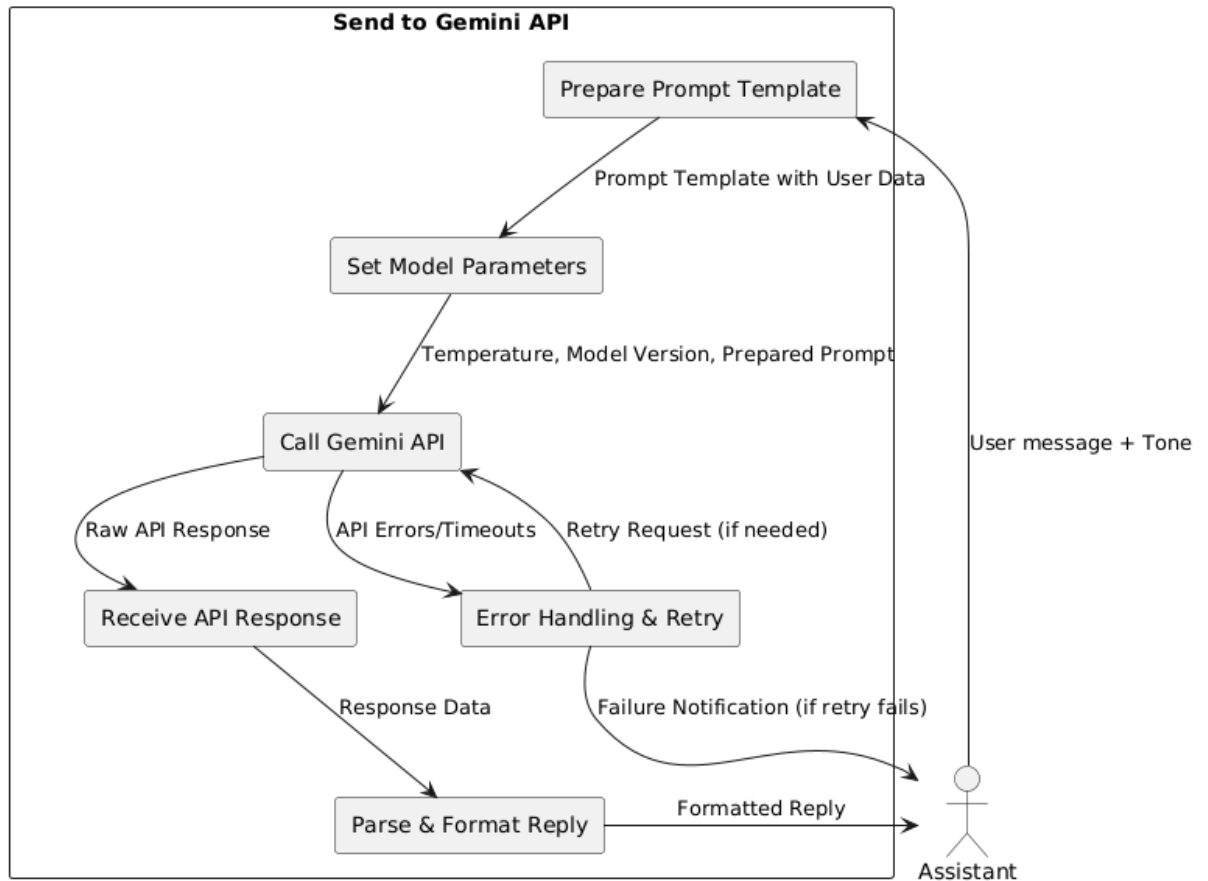


Figure – 2.3

4.3 USE CASE DIAGRAM

A Use Case Diagram represents the functional requirements of a system by showing actors and their interactions. It highlights what users can do and how they interact with system features. In your Smart Email Assistant, it maps users, system processes, and their use cases like sending messages and receiving replies. This helps understand system scope and user goals clearly.

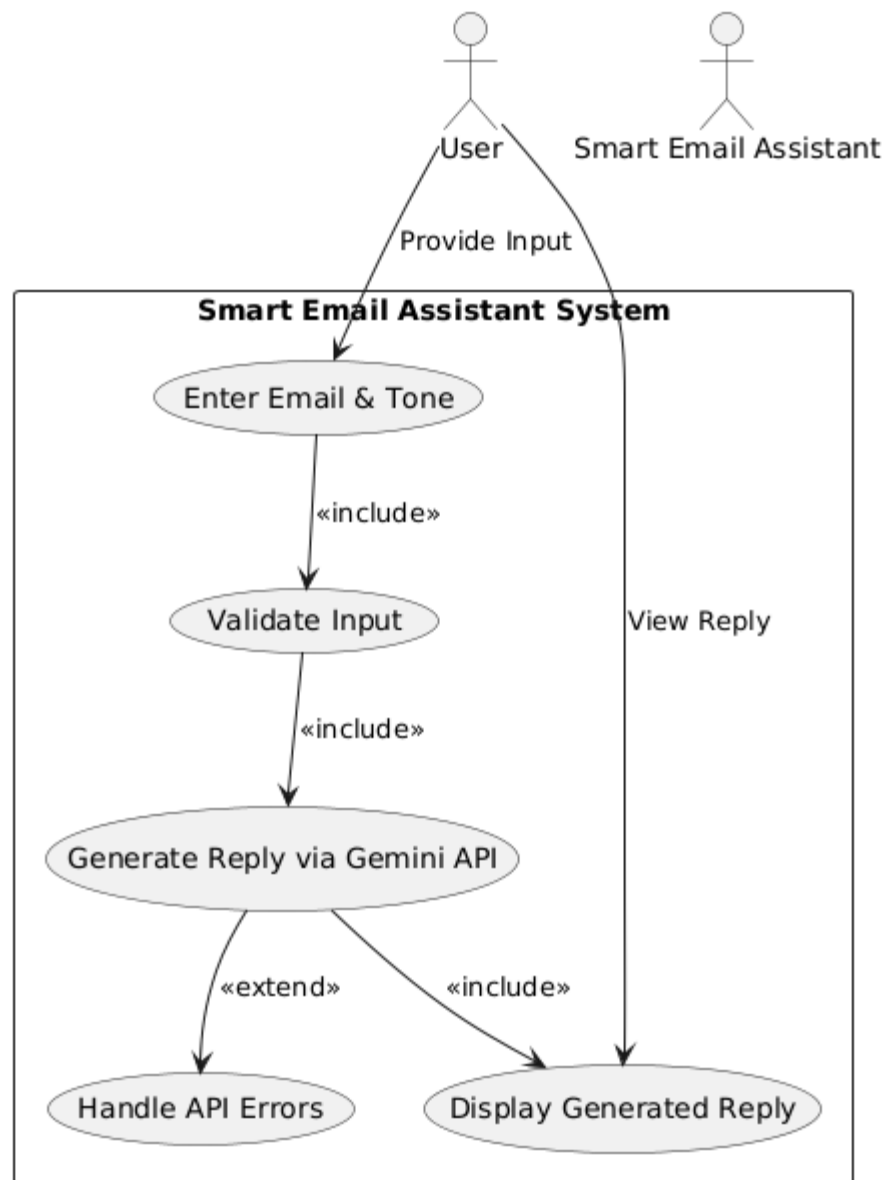


Figure - 3

4.4 SEQUENCE DIAGRAM

A Sequence Diagram shows how different components interact over time through messages. It illustrates the order of method calls between actors and system parts. In your Smart Email Assistant, it details the flow from user input to API response and error handling.

This helps visualize the dynamic behavior and communication sequence within the system.

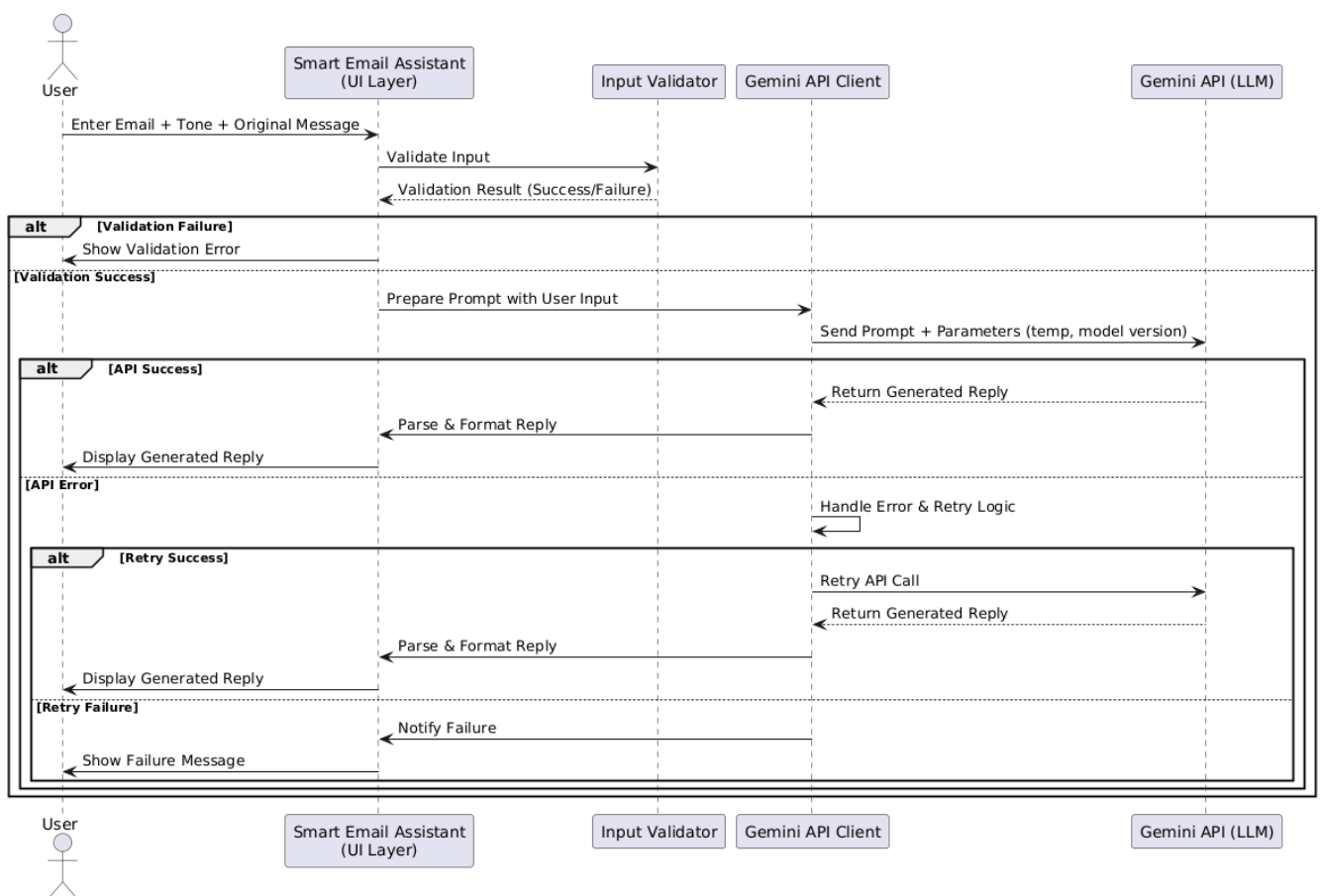


Figure - 4

4.5 ACTIVITY DIAGRAM

An Activity Diagram visually represents the workflow or process steps of a system. It shows actions, decisions, and the flow between them from start to end. In your Smart Email Assistant, it maps user input, validation, API calls, and error handling. This helps understand the system's sequence of activities and decision points clearly.

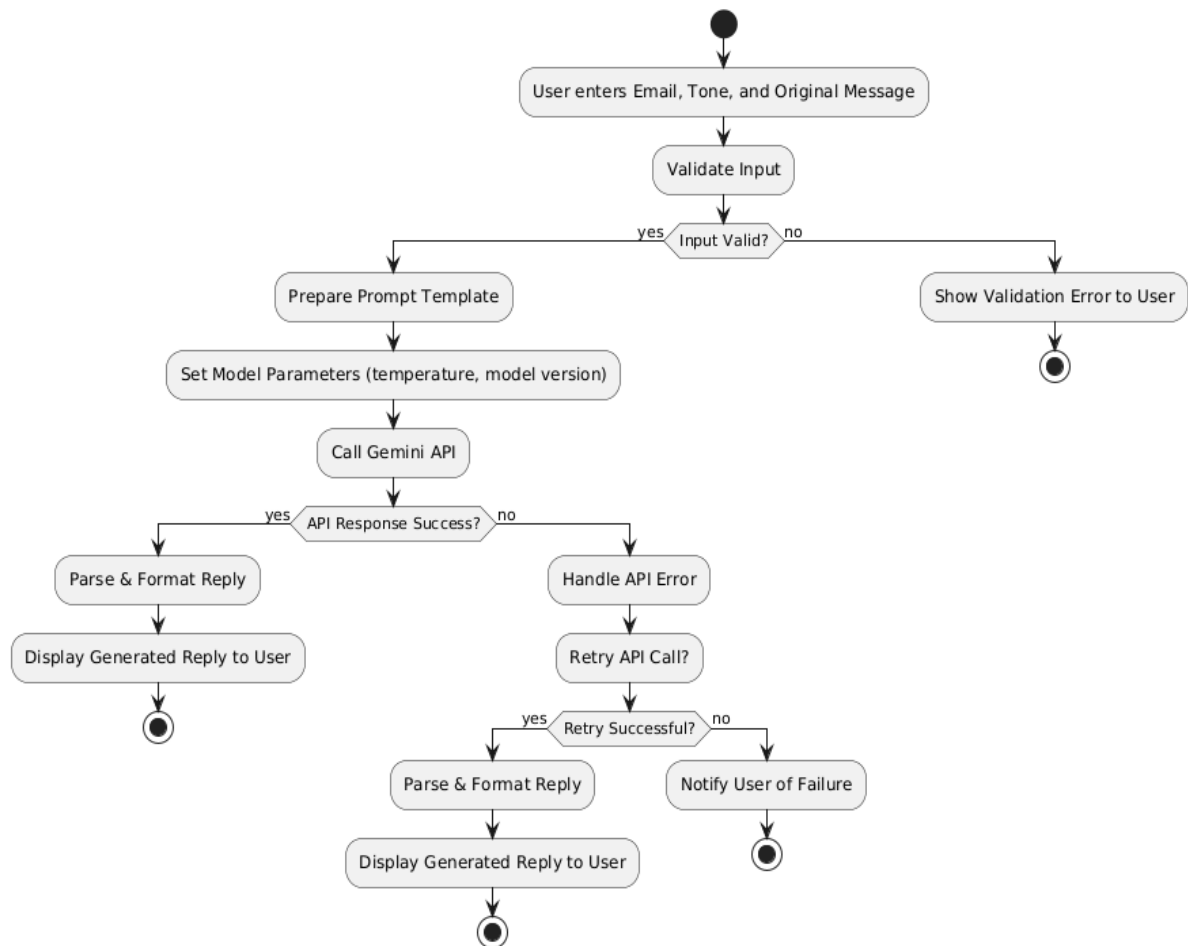


Figure - 5

5. GUI

5.1 USER INTERFACE DESIGN

Overview

The user interface of the Smart Email Assistant is designed to offer a clean, intuitive, and responsive experience for users seeking to generate AI-powered email replies efficiently. Built using modern web technologies such as React.js with Material UI, the UI facilitates seamless interaction with the backend Spring Boot application integrated with the Gemini API. Users can input their email content and desired tone, trigger AI-based reply generation, and view or copy the generated email response within a sleek and minimalistic design.

MAIN COMPONENTS OF THE UI

User Input Form

- Provides input fields for the user's email message and a dropdown to select the desired tone (e.g., Formal, Casual, Friendly).
- Includes a "Generate Reply" button to submit the input for AI processing.
- Validates inputs to ensure the message content is not empty before submission.
- Utilizes Material UI components for consistent styling and responsive behavior across devices.

Reply Display Section

- Shows the AI-generated email reply returned from the Gemini API after processing.
- Includes options to copy the reply to clipboard or regenerate with different parameters.
- Displays loading indicators during API calls to enhance user feedback and experience.

Navigation and Layout

- Uses a clean, card-based layout with distinct sections for input and output.
- Responsive design ensures usability on desktops, tablets, and mobile devices.
- Header includes branding/logo and project title for easy identification.

Error Handling and Notifications

- Provides user-friendly alerts and error messages in case of failed API calls or network issues.
- Ensures smooth user experience even in edge cases by displaying appropriate retry options.

This UI design approach ensures users can interact effortlessly with the AI-powered assistant, making email generation both fast and effective. The combination of React and Material UI supports a modern frontend experience aligned with industry standards, while integration with the Spring Boot backend ensures robust data processing and API management.

5.2 MODULES SCREENSHOT

1. SPRING BOOT PROJECT (Spring intilizer)

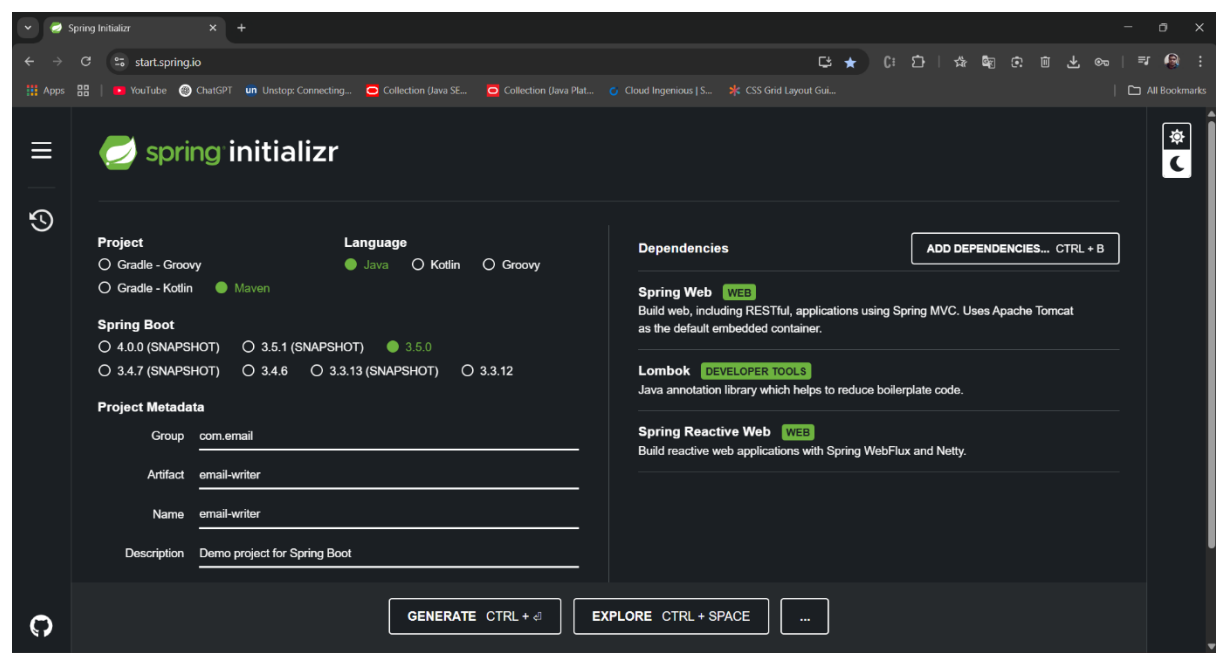


Figure – 6

2. GEMINI API GENERATE

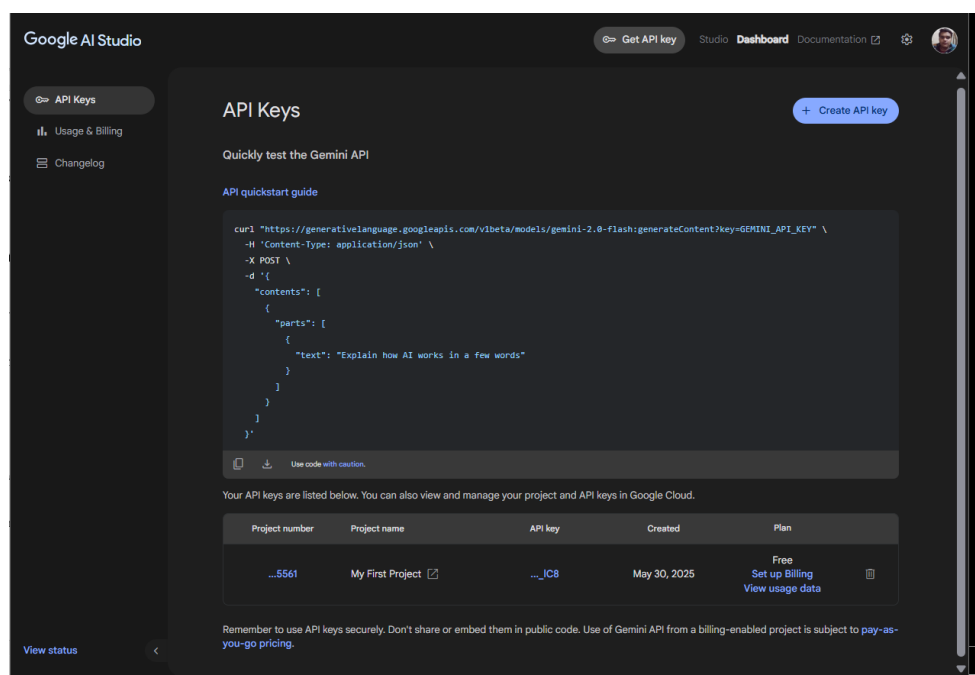


Figure – 7

3. GEMINI API CHECK

3.1 POSTMAN CHECK GEMINI API

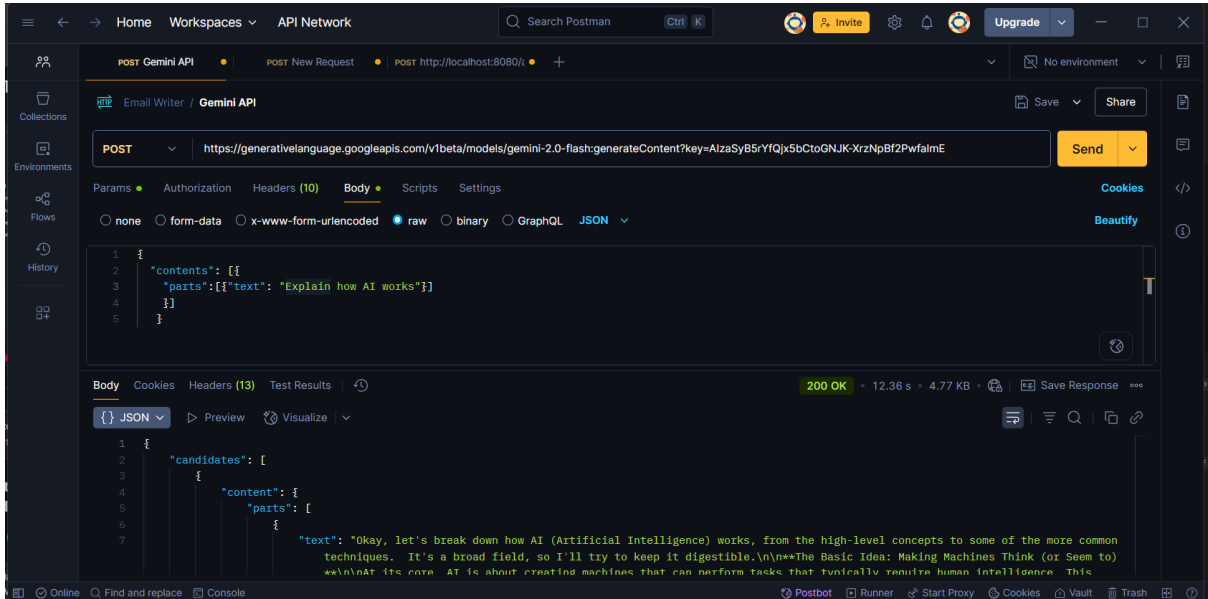


Figure – 8

4. POSTMAN CHECK POST REQUEST FOR API RESPONSE

4.1 POST REQUEST

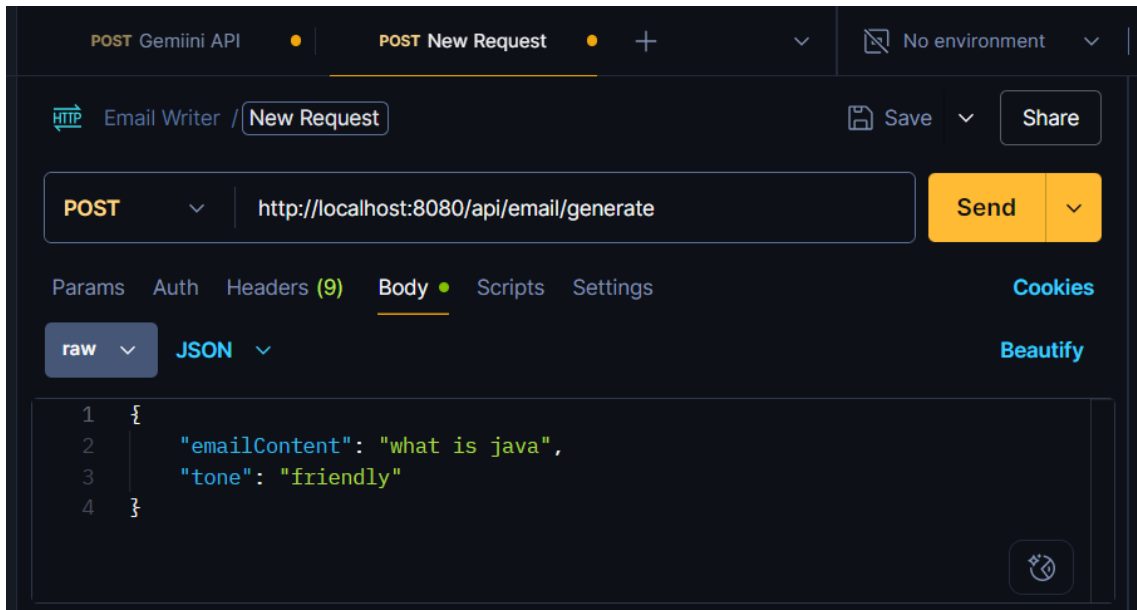
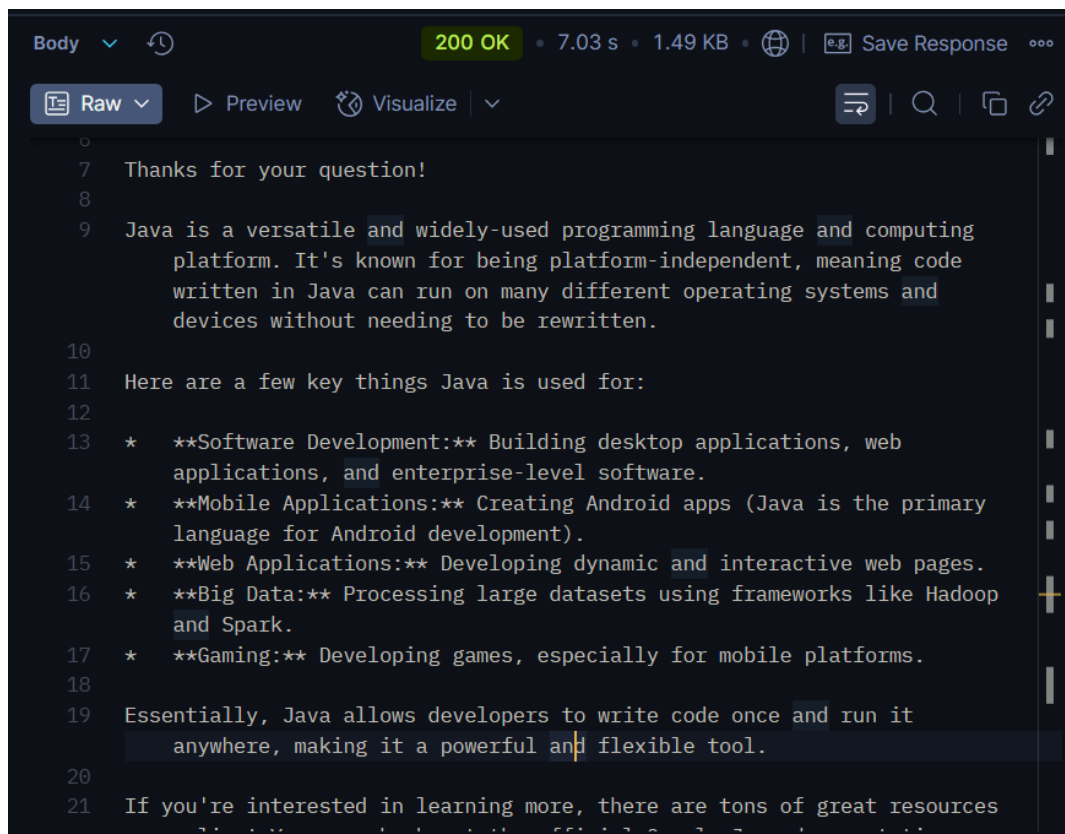


Figure – 9

4.2 POST RESPONSE

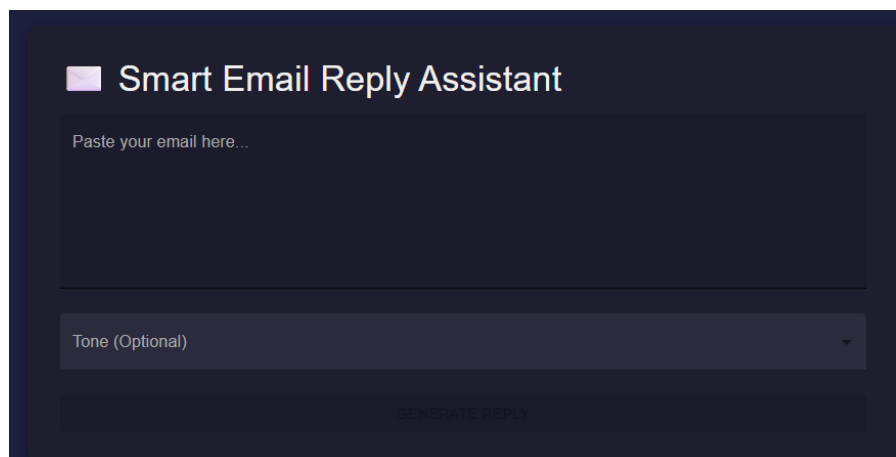


```
Body 200 OK • 7.03 s • 1.49 KB | Save Response
Raw Preview Visualize
{
  "message": "Thanks for your question!",
  "content": "Java is a versatile and widely-used programming language and computing platform. It's known for being platform-independent, meaning code written in Java can run on many different operating systems and devices without needing to be rewritten.",
  "key_things": "Here are a few key things Java is used for:",
  "examples": [
    "**Software Development:** Building desktop applications, web applications, and enterprise-level software.",
    "**Mobile Applications:** Creating Android apps (Java is the primary language for Android development).",
    "**Web Applications:** Developing dynamic and interactive web pages.",
    "**Big Data:** Processing large datasets using frameworks like Hadoop and Spark.",
    "**Gaming:** Developing games, especially for mobile platforms."
  ],
  "summary": "Essentially, Java allows developers to write code once and run it anywhere, making it a powerful and flexible tool.",
  "resources": "If you're interested in learning more, there are tons of great resources"
}
```

Figure - 10

5. EMAIL-WRITER-REACT

5.1 App.jsx



Smart Email Reply Assistant

Paste your email here...

Tone (Optional)

GENERATE REPLY

Figure - 11

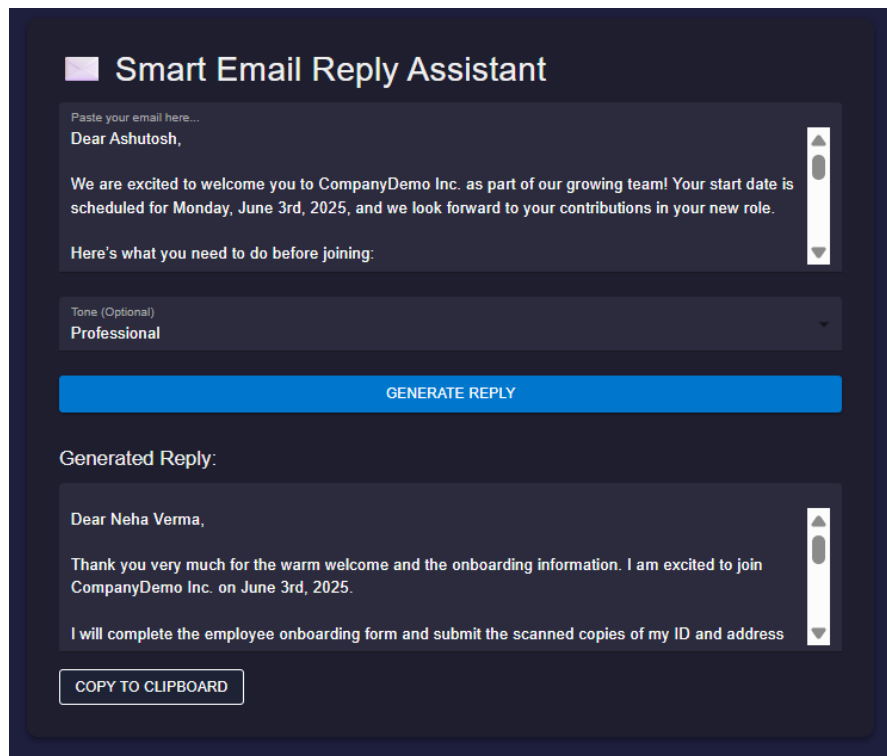


Figure – 12

6. email-writer-ext (EXTENSION)

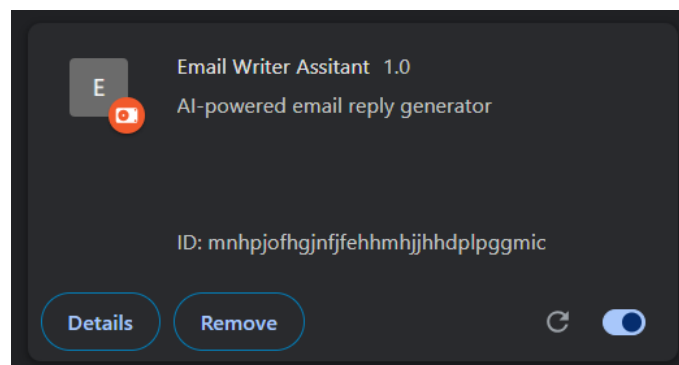


Figure – 13

7. EXTENSION INTEGRATE WITH GMAIL

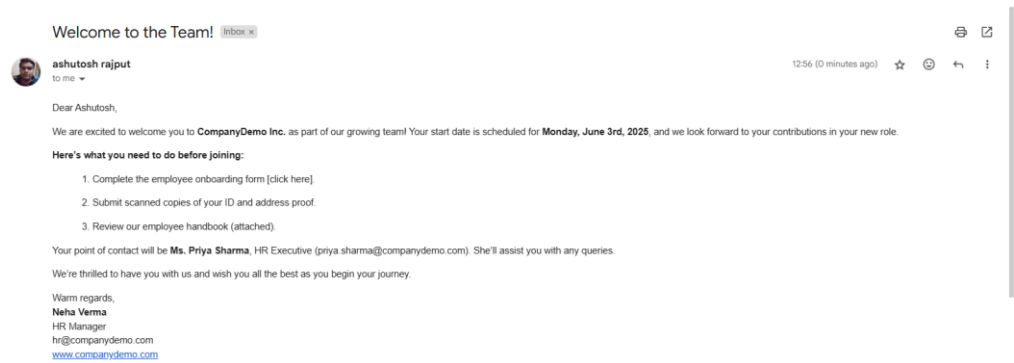


Figure – 14

8. EMAIL OUTPUT GENERATE WITH AI

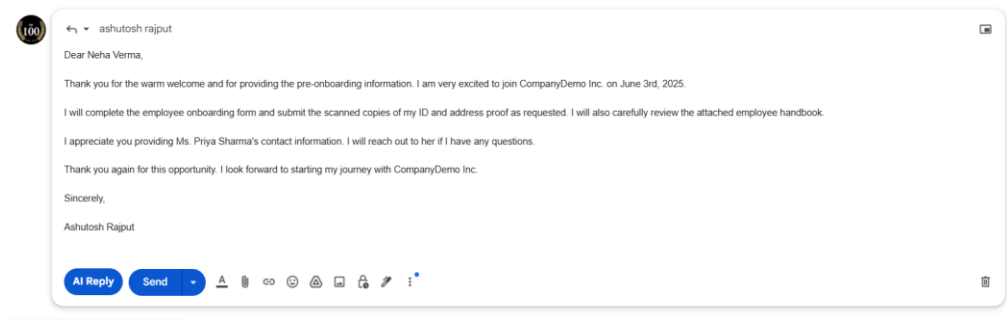


Figure – 15

6. CODING

6.1 PROGRAMMING LANGUAGES AND TOOLS USED

6.1.1 Programming Languages

- **Java** – Used for backend development with Spring Boot
- **JavaScript** – Used for Chrome Extension and frontend logic

6.1.2 Backend Technologies

- **Spring Boot** – Core backend framework
- **Spring Web & Spring WebFlux** – For building RESTful and reactive APIs
- **Spring AI** – Integrates AI capabilities using Gemini/OpenAI
- **Lombok** – Simplifies boilerplate code using annotations
- **Gemini API** – Google’s generative AI for crafting intelligent email replies

6.1.3 Frontend / Extension Technologies

- **Chrome Extension (JavaScript)** – Injected into Gmail UI for seamless user experience
- **React.js** – Optional UI layer for advanced components
- **Material UI** – UI component library for styling and responsiveness

6.1.4 Development Tools

- **IntelliJ IDEA** – For backend development and debugging
- **Visual Studio Code** – For frontend and Chrome Extension development
- **Postman** – For testing backend API endpoints
- **Chrome Developer Tools** – For inspecting and debugging Gmail DOM and extension behavior

6.1.5 Libraries & APIs

- **Axios / Fetch API** – For frontend-backend communication
- **HTML DOM Manipulation** – Used by the Chrome Extension to modify Gmail interface

6.2 CODE ARCHITECTURE AND ORGANIZATION

1. BACKEND API (SPRING BOOT + SPRING AI INTEGRATION)

- **Description:**

The backend is a RESTful API built with Spring Boot. It accepts requests from both the web frontend and Chrome extension, prepares prompts for AI, calls the Gemini/OpenAI API via Spring AI, and returns generated responses.

- **Features:**

- POST endpoint `/api/email/generate` to receive email content and tone
- Processes input and constructs AI prompt templates
- Calls AI model using Spring AI integration
- Handles errors, retries, and response formatting
- Configuration for API keys and model versions

- **Technology Stack:**

- Java with Spring Boot framework
- Spring AI starter for LLM (large language model) integration
- Maven for dependency management

- **Workflow:**

Frontend or extension sends request → Controller receives and validates → Service generates AI prompt and calls model → AI response returned to caller.

Backend Project Directory

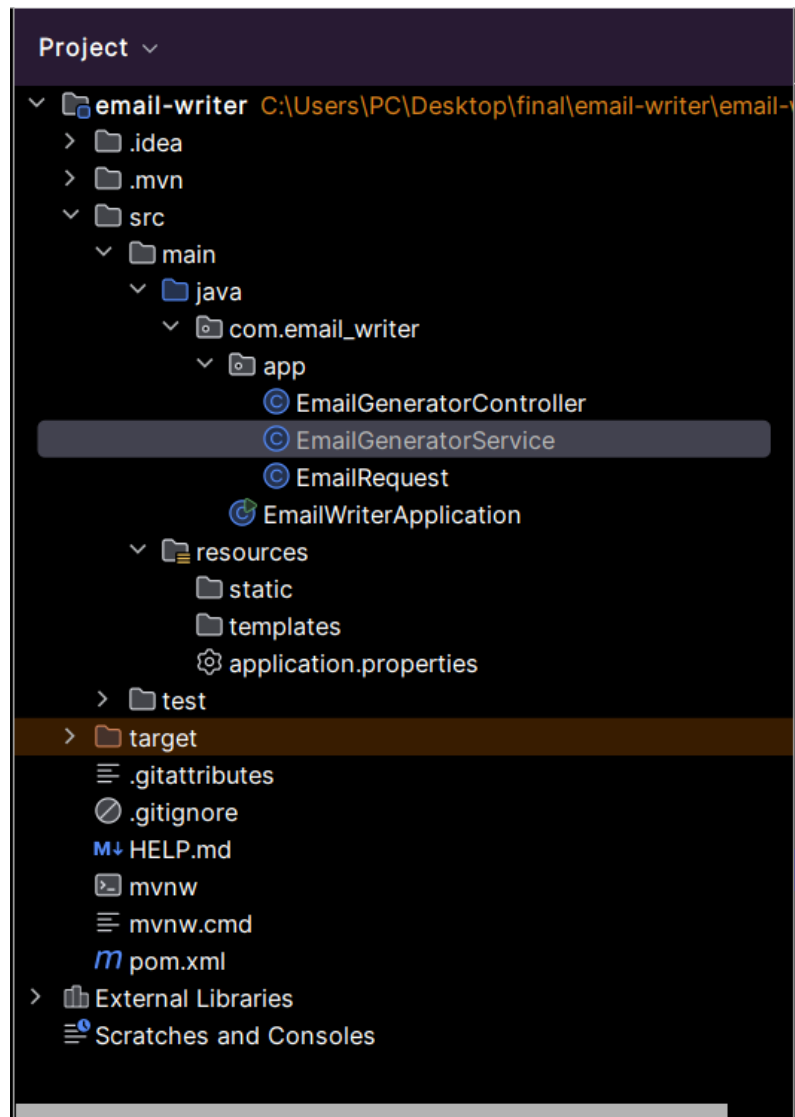


Figure - 16

2. FRONTEND WEB APPLICATION (REACT + MATERIAL UI)

- **Description:**

The web frontend offers a responsive, user-friendly interface where users can type email content and generate AI-powered replies. It is built using React and styled with Material UI for a modern and professional look.

- **Features:**

- Input form for email content
- Tone selection options (e.g., professional, casual)
- Displays the AI-generated reply
- Loading indicators and error handling for smooth user experience

- **Technology Stack:**

- React.js (functional components with hooks)
- Material UI for design and styling
- Fetch API for communication with backend

- **Workflow:**

User enters email content → selects tone → clicks “Generate” → frontend sends API request → backend processes and returns AI reply → frontend displays generated reply.

Frontend Project Directory

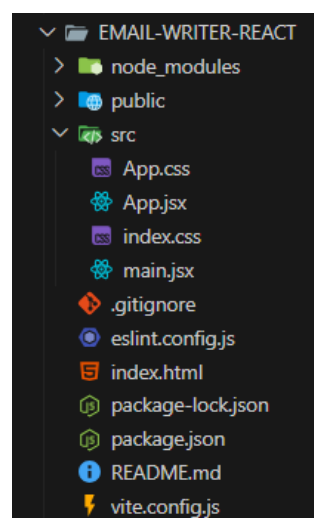


Figure - 17

3. CHROME EXTENSION (CONTENT SCRIPT IN JAVASCRIPT)

- **Description:**

The Chrome extension injects an “AI Reply” button near the Gmail compose window’s toolbar. When clicked, it captures the drafted email content, sends it to the backend, and inserts the AI-generated reply directly into the Gmail compose box.

- **Features:**

- Injects button dynamically into Gmail compose toolbar
- Listens to user interaction (button click)
- Sends POST requests to backend API
- Inserts generated reply into the compose textbox
- Detects Gmail compose window open/close events for dynamic UI injection

- **Technology Stack:**

- JavaScript (ES6+)
- Chrome Extension APIs
- DOM manipulation for UI changes

- **Workflow:**

Gmail compose window opens → MutationObserver detects and injects “AI Reply” button → user clicks button → email content sent to backend → AI-generated reply returned → reply inserted into Gmail compose box.

Extension Project Directory

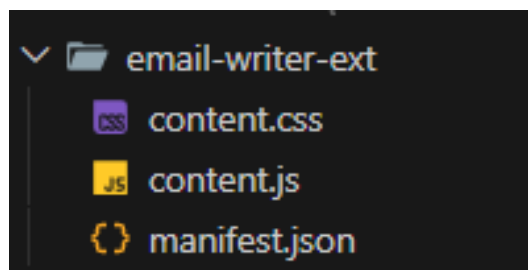


Figure - 18

6.3 KEY CODE SNIPPETS

Below are some key code snippets adapted for the **Smart Email Assistant** project, structured to demonstrate core features like AI-powered email generation using Spring Boot, Spring AI, and React.

1. BACKEND API (SPRING BOOT + SPRING AI INTEGRATION)

1.1 EmailWriterApplication

```
@SpringBootApplication

public class EmailWriterApplication {

    public static void main(String[] args) {

        SpringApplication.run(EmailWriterApplication.class, args);

    }

}
```

1.2 EmailGeneratorController

```
@RestController

@RequestMapping("/api/email")

//@AllArgsConstructor

@CrossOrigin(origins = "*")

public class EmailGeneratorController {

    private final EmailGeneratorService emailGeneratorService;

    public EmailGeneratorController(EmailGeneratorService emailGeneratorService) {

        this.emailGeneratorService = emailGeneratorService;

    }

    @PostMapping("/generate")
```

```

    public ResponseEntity<String> generateEmail(@RequestBody EmailRequest
emailRequest) {

        String response = emailGeneratorService.generateEmailReply(emailRequest);

        return ResponseEntity.ok(response);

    }
}

```

1.3 EmailGeneratorService

```

@Service

public class EmailGeneratorService {

    private final WebClient webClient;

    @Value("${gemini.api.url}")
    private String geminiApiUrl;

    @Value("${gemini.api.key}")
    private String geminiApiKey;

    public EmailGeneratorService(WebClient.Builder webClientBuilder) {

        this.webClient = webClientBuilder.build();

    }

    public String generateEmailReply(EmailRequest emailRequest){

        String prompt = buildPrompt(emailRequest);

        // Craft a request

```



```

Map<String, Object> requestBody = Map.of(
    "contents", new Object[] {
        Map.of("parts", new Object[] {
            Map.of("text", prompt)
        })
    }
);

// Do request and get response

String response = webClient.post()
    .uri(geminiApiUrl + geminiApiKey)
    .header("Content-Type", "application/json")
    .bodyValue(requestBody)
    .retrieve()
    .bodyToMono(String.class)
    .block();

```

// Extract Response and Return

```

return extractResponseContent(response);
}

private String extractResponseContent(String response) {
    try {
        ObjectMapper mapper = new ObjectMapper();
        JsonNode rootNode = mapper.readTree(response);
    }
}

```

```

        return rootNode.path("candidates")

            .get(0)

            .path("content")

            .path("parts")

            .get(0)

            .path("text")

            .asText();
    } catch (Exception e) {

        return "Error processing request: " + e.getMessage();
    }
}

```

```

private String buildPrompt(EmailRequest emailRequest){

    StringBuilder prompt = new StringBuilder();

    prompt.append("Generate a professional email reply for hte following email content.
Please don't generate a subject line ");

    if (emailRequest.getTone() != null && !emailRequest.getTone().isEmpty()) {

        prompt.append("Use a ").append(emailRequest.getTone()).append(" tone.");
    }

    prompt.append("\nOriginal email: \n").append(emailRequest.getEmailContent());

    return prompt.toString();}
}

```

1.4 EmailRequest

```
@Data

public class EmailRequest {

    private String emailContent;

    private String tone;

    public String getEmailContent() {

        return emailContent;

    }

    public String getTone() {

        return tone;

    }

}
```

1.5 application.properties

```
spring.application.name=email-writer
```

```
gemini.api.url = ${GEMINI_URL}
```

```
gemini.api.key = ${GEMINI_KEY}
```

2. FRONTEND WEB APPLICATION (REACT + MATERIAL UI)

2.1 App.jsx

```
import { useState } from 'react';

import './App.css';

import { Box, Button, CircularProgress, Container, FormControl, InputLabel, MenuItem,
Select, TextField, Typography, Paper } from '@mui/material';

import axios from 'axios';

function App() {

  const [emailContent, setEmailContent] = useState("");

  const [tone, setTone] = useState("");

  const [generatedReply, setGeneratedReply] = useState("");

  const [loading, setLoading] = useState(false);

  const [error, setError] = useState("");

  const handleSubmit = async () => {

    setLoading(true);

    setError("");

    try {

      const response = await axios.post('http://localhost:8080/api/email/generate', {

        emailContent,

        tone,

      });

    }
```

```

        setGeneratedReply(typeof response.data === 'string' ? response.data :
JSON.stringify(response.data));

    } catch (error) {

        setError('Failed to generate email reply. Please try again.');
```

console.error(error);

```

    } finally {

        setLoading(false);

    }

};

return (

    <Container maxWidth="md" sx={{ py: 4 }}>

        <Paper elevation={4} sx={{ p: 4, bgcolor: '#1e1e2f', borderRadius: 3 }}>

            <Typography variant="h4" gutterBottom color="#f0f0f0">

                📧 Smart Email Reply Assistant

            </Typography>

            <TextField

                fullWidth

                multiline

                rows={6}

                variant="filled"

                label="Paste your email here..."

                value={emailContent}

                onChange={(e) => setEmailContent(e.target.value)}

```

```

sx={{ mb: 3 }}

InputProps={{ sx: { backgroundColor: '#2b2b3d', color: '#fff' } }}

InputLabelProps={{ sx: { color: '#aaa' } }}

/>

<FormControl fullWidth variant="filled" sx={{ mb: 3 }}>

  <InputLabel sx={{ color: '#aaa' }}>Tone (Optional)</InputLabel>

  <Select

    value={tone}

    onChange={(e) => setTone(e.target.value)}

    sx={{ backgroundColor: '#2b2b3d', color: '#fff' }}

  >

    <MenuItem value="">None</MenuItem>

    <MenuItem value="professional">Professional</MenuItem>

    <MenuItem value="casual">Casual</MenuItem>

    <MenuItem value="friendly">Friendly</MenuItem>

  </Select>

</FormControl>

<Button

  variant="contained"

  fullWidth

  onClick={handleSubmit}

  disabled={!emailContent || loading}

  sx={{

```

```

    bgcolor: '#0077cc',

    '&:hover': { bgcolor: '#005fa3' },

    color: 'fff',

  }}

>

  {loading ? <CircularProgress size={24} color="inherit" /> : 'Generate Reply'}

</Button>

{error && (

  <Typography color="error" sx={{ mt: 2 }}>

    {error}

  </Typography>

)}

{generatedReply && (

  <Box sx={{ mt: 4 }}>

    <Typography variant="h6" color="#f0f0f0" gutterBottom>

      Generated Reply:

    </Typography>

    <TextField

      fullWidth

      multiline

      rows={6}

      variant="filled"

      value={generatedReply}

```

```

        InputProps={{ readOnly: true, sx: { backgroundColor: '#2b2b3d', color: '#fff' }
    }}

    />

    <Button

        variant="outlined"

        sx={{ mt: 2, color: '#fff', borderColor: '#888', '&:hover': { borderColor: '#fff' }
    }}

        onClick={() => navigator.clipboard.writeText(generatedReply)}

    >

        Copy to Clipboard

    </Button>

</Box>

)}

</Paper>

</Container>

);

}

export default App;

```


3. CHROME EXTENSION (CONTENT SCRIPT IN JAVASCRIPT)

3.1 content.js

```
console.log("Email Writer Extension - Content Script Loaded");

function createAIButton() {

    const button = document.createElement('div');

    button.className = 'T-I J-J5-Ji aoO v7 T-I-atl L3';

    button.style.marginRight = '8px';

    button.style.backgroundColor = '#0b57d0'; //

    button.style.borderRadius = '20px'; // Rounded sides

    button.style.color = '#fff'; // White text for better contrast

    button.style.fontWeight = 'bold';

    button.style.padding = '0 12px'; // Add padding to look better

    button.innerHTML = 'AI Reply';

    button.setAttribute('role','button');

    button.setAttribute('data-tooltip','Generate AI Reply');

    return button;

}

function getEmailContent() {

    const selectors = [

        '.h7', '.a3s.aiL', '.gmail_quote', '[role="presentation"]' ];

    for (const selector of selectors) {

        const content = document.querySelector(selector);

        if (content) {
```

```

        return content.innerText.trim();
    }

    return "";
}
}

```

```

function findComposeToolbar() {
    const selectors = [
        '.btC',
        '.aDh',
        '[role="toolbar"]',
        '.gU.Up'
    ];

    for (const selector of selectors) {
        const toolbar = document.querySelector(selector);

        if (toolbar) {
            return toolbar;
        }

        return null;
    }
}

```

```

function injectButton() {
    const existingButton = document.querySelector('.ai-reply-button');

    if (existingButton) existingButton.remove();
}

```

```

const toolbar = findComposeToolbar();

if (!toolbar) {

    console.log("Toolbar not found");

    return;

}

console.log("Toolbar found, creating AI button");

const button = createAIButton();

button.classList.add('ai-reply-button');

button.addEventListener('click', async () => {

    try {

        button.innerHTML = 'Generating...';

        button.disabled = true;

        const emailContent = getEmailContent();

        const response = await fetch('http://localhost:8080/api/email/generate', {

            method: 'POST',

            headers: {

                'Content-Type': 'application/json',

            },

            body: JSON.stringify({

                emailContent: emailContent,

                tone: "professional"
            })
        });
    }
});

```

```

        })

    });

    if (!response.ok) {
        throw new Error('API Request Failed');
    }

    const generatedReply = await response.text();

const composeBox = document.querySelector('[role="textbox"][g_editable="true"]');

    if (composeBox) {
        composeBox.focus();

        document.execCommand('insertText', false, generatedReply);
    } else {
        console.error('Compose box was not found');
    }

} catch (error) {
    console.error(error);

    alert('Failed to generate reply');
} finally {
    button.innerHTML = 'AI Reply';

    button.disabled = false;
}

});

```

```

    toolbar.insertBefore(button, toolbar.firstChild);
  }

const observer = new MutationObserver((mutations) => {
  for(const mutation of mutations) {
    const addedNodes = Array.from(mutation.addedNodes);
    const hasComposeElements = addedNodes.some(node =>
      node.nodeType === Node.ELEMENT_NODE &&
      (node.matches('.aDh, .btC, [role="dialog"]') || node.querySelector('.aDh, .btC,
[role="dialog"]'))
    );

    if (hasComposeElements) {
      console.log("Compose Window Detected");
      setTimeout(injectButton, 500);
    }
  }
});

observer.observe(document.body, {
  childList: true,
  subtree: true
});

```

3.2 manifest.json

```
{
  "name": "Email Writer Assitant",
  "description": "AI-powered email reply generator",
  "version": "1.0",
  "manifest_version": 3,
  "permissions": ["activeTab", "storage"],
  "host_permissions": [
    "http://localhost:8080/*",
    "*/://mail.google.com/*"
  ],
  "content_scripts": [
    {
      "js": ["content.js"],
      "matches": ["*/://mail.google.com/*"],
      "css": ["content.css"],
      "run_at": "document_end"
    }
  ],
  "web_accessible_resources": [
    {
      "resources": [ "icons/*" ],
      "matches": ["*/://mail.google.com/*"]
    }
  ],
  "action": {
    "default_title": "Email Writer Assitant"
  }
}
```

7. FUTURE SCOPE

The **Smart Email Assistant** demonstrates the potential of AI in enhancing digital communication. Future improvements can make the system more intelligent, accessible, and enterprise-ready.

7.1 Multi-Language Support

Extend support for multiple languages using multilingual LLMs, allowing users to generate replies in their native language.

7.2 Context-Aware and Threaded Responses

Enable conversation tracking so replies consider previous emails in the thread, improving relevance and coherence.

7.3 Voice-to-Email Integration

Incorporate speech-to-text functionality to allow users to dictate emails and receive AI-generated replies via voice input.

7.4 Custom Tone and Style Adaptation

Allow the AI to adapt to user-specific writing styles and offer tone options like formal, friendly, or technical.

7.5 Cross-Platform Plugin Deployment

Deploy the assistant as a plugin for Gmail, Outlook, and browsers like Chrome, Edge, and Firefox.

7.6 Sentiment and Emotion Analysis

Detect the emotional tone of messages and adjust AI responses for empathy, professionalism, or urgency.

7.7 Smart Summarization and Categorization

Add features to summarize long email threads and auto-categorize emails based on urgency, topic, or sentiment.

7.8 AI Feedback and Learning Loop

Implement user feedback mechanisms to rate responses and use the data to improve future outputs.

7.9 Security and Compliance Enhancements

Enhance data protection with role-based access, OAuth, encryption, and compliance with GDPR and other privacy standards.

7.10 Integration with CRM and Productivity Tools

Enable integration with platforms like Salesforce, Zoho CRM, or Slack to support sales, customer service, and team collaboration.

8. CONCLUSION

- The **Smart Email Assistant** project showcases the transformative potential of generative AI in redefining email communication by delivering context-aware, tone-appropriate, and highly personalized automated responses. By integrating Google's cutting-edge **Gemini API**, the assistant leverages state-of-the-art large language models capable of understanding nuanced user intent and generating high-quality, relevant email replies with minimal user effort.
- The architecture effectively combines a robust **Spring Boot backend** for managing business logic, API calls, and user session handling, with the Gemini API's AI-powered natural language generation. The system's modular design also allows easy integration with frontend platforms such as Gmail extensions or standalone web applications, thus providing users with a seamless and intuitive experience.
- Throughout the project, key challenges such as ensuring **data privacy**, managing **API rate limits**, and designing a **scalable and maintainable backend** were addressed. The use of Spring Boot's MVC architecture and RESTful principles ensures clean separation of concerns and easier future enhancements.
- The assistant significantly enhances productivity for professionals by automating routine communication tasks like drafting replies, managing follow-ups, and even suggesting email summaries, thereby reducing cognitive load and saving valuable time.
- Although the current version implements core features such as input capture, API integration, and response rendering, the system is designed to be highly extensible.
- Future iterations could include advanced functionalities like:
 - **Tone adaptation** to match different communication styles (formal, casual, persuasive).
 - **Multi-language support** enabling global usability.
 - **Voice input and output** for hands-free operation.
 - **Contextual memory** to remember ongoing conversation threads.

- **Integration with calendar and task management tools** for proactive email suggestions.
- Moreover, ethical considerations such as **transparency in AI-generated content**, user consent, and data security will be vital in the ongoing development to build trust and comply with regulations.
- In conclusion, the Smart Email Assistant not only demonstrates a strong grasp of backend development, API integration, and AI technologies but also represents a pivotal step toward creating intelligent, user-friendly digital communication assistants that can redefine how we interact with email and enhance professional workflows in the digital age.

9. REFERENCES

1. **Google AI – Gemini API Documentation**
<https://ai.google.dev/>
(Official guide to using Gemini LLM APIs for AI-based functionalities.)
2. **Spring Boot Documentation – Spring.io**
<https://docs.spring.io/spring-boot/docs/current/reference/html/>
(Comprehensive reference for developing REST APIs using Spring Boot.)
3. **PlantUML Official Documentation**
<https://plantuml.com/>
(Used to generate ER and DFD diagrams for the project design section.)
4. **Baeldung – Spring Boot Tutorials**
<https://www.baeldung.com/>
(In-depth articles on Java, Spring Boot, and RESTful API integration.)
5. **Stack Overflow – Programming Help**
<https://stackoverflow.com/>
(Community-based support for resolving development issues.)
7. **"Spring in Action" by Craig Walls**
Publisher: Manning Publications
(Highly recommended for learning Spring Boot concepts and building real-world Java applications.)
8. **"Java: The Complete Reference" by Herbert Schildt**
Publisher: McGraw-Hill Education
(A foundational book for mastering Java, which supports the backend of this project.)
9. **"Designing Data-Intensive Applications" by Martin Kleppmann**
Publisher: O'Reilly Media
(For understanding scalable system design and data flow – useful in conceptual design.)
10. **"Artificial Intelligence: A Modern Approach" by Stuart Russell & Peter Norvig**
Publisher: Pearson
(Provides theoretical insights into AI, useful for understanding the logic behind language models like Gemini.)