SmartChat

Software Engineering B.Sc. Final Project Write-up

Authors

Asaf Zafrir 205929029

Eliya Samary Atias 206484628

Nave Maymon 318887965

Supervisor

Mr. Yitzhak Nudler

31/08/2025

Approved by: ______________ Date: 31-08-2025

A thesis submitted in partial fulfillment of the requirements for the degree of BSc. in Software Engineering

Abstract

SmartChat is an AI-powered conversational platform with a chat-based interface, designed to deliver accurate, relevant, and context-aware responses that leverage the full conversation history and adapt to user-specific preferences within the domain of software engineering. The system evaluates the outputs of multiple AI platforms, selects the most suitable response, and further refines it.

Upon receiving a user query, SmartChat performs advanced prompt processing, taking into account the user's preferences, previous interactions, and feedback history. The processed prompt is then concurrently submitted to three distinct AI platforms. The system then executes a comparative evaluation to determine the most suitable response, which is then refined and presented to the user.

To continuously enhance performance and user satisfaction, SmartChat invites users to provide feedback following each interaction. This feedback is analyzed and used to inform future response selection and personalization strategies.

Acknowledgments

We would like to express our sincere and deep appreciation to Mr. Itzhak Nudler, senior lecturer in the Software Engineering Department at Shenkar College, for his unwavering support, insightful guidance, and continuous encouragement throughout all stages of this project.

His extensive knowledge, attention to detail, and thoughtful feedback greatly contributed to the success of our work. In many critical moments, his advice helped us overcome challenges.

We are especially grateful for his willingness to dedicate his time and attention far beyond expectations, always with patience, professionalism, and genuine care.

The successful collaboration and personal mentorship we received from Mr. Nudler left a meaningful impact on our academic journey. Thank you from the bottom of our hearts.

Table of Contents

1. INTRODUCTION **6**

1.1. Problem Description 6

1.2. Motivation 7

1.3. Goals 8

1.4. System Overview 8

1.5. Usage scenarios 9

1.6. Audience 14

1.7. Glossary 15

2. LITERATURE SURVEY **18**

2.1. Problem Survey 18

2.2. Solution Survey 19

2.3. Discussion and Conclusion 20

3. TECHNOLOGICAL SURVEY **21**

3.1. Problem Survey 21

3.2. Solution Survey 22

3.3. Discussion and Conclusion 25

4. REQUIREMENTS AND SPECIFICATION **27**

4.1. Functional Requirements 27

4.2. Non - Functional Requirements 30

4.3. The Scope 31

4.4. The Approach 31

5. THE ARCHITECTURE **34**

5.1. ShartChat Data 34

5.2. ShartChat Processes 37

5.3. Roles 39

5.4. Architectural Description 41

5.5. The life-cycle view of a system 43

6. SYSTEM DESIGN **45**

6.1. Data Components 45

6.2. Process Components 53

6.3. Communication Components 55

6.4. Interactions 56

7. IMPLEMENTATION **63**

7.1. Algorithm Design 63

7.2. Interfaces 66

7.3. System Configuration 69

7.4. Development Environment 71

7.5. Programming Languages 72

7.6. Limitations of the System 73
7.7. Installation and Configuration Issues 74
7.8. Information Security 75
7.9. Risk Management 77
7.10. Exceptions Management 78
7.11. Version Control 79
7.12. Project Management 81
7.13. Code 82
8. System Validation **85**
8.1. Backend Validation 85
8.2. Frontend Validation 88
8.3. Conversation Flow Integration 93
8.4. Decision-Making Logic 101
8.5. Feedback Adaptation 109
8.6. Feedback Free-Text Analysis 113
8.7. Validation Prompt per User 116
8.8. Project Scope Validation 124
8.9. Test Results & Conclusion 126
9. References **127**
10.Appendices **128**

Table of Figures

Figure 1 - System Overview Diagram 8
Figure 2 - Diagram describing the approach of smartChat 32
Figure 3 - System Architecture 41
Figure 4 - The system life cycle 43
Figure 5 - SmartChat MongoDB Entity-Relationship Diagram 50
Figure 6 – External representation diagram 52
Figure 7 – Diagram representing the flow among process components 54
Figure 8 – Activity Diagram 57
Figure 9 – Sequence Diagram 60
Figure 10 – System Flow Diagram 61
Figure 11 - Chat page view 68
Figure 12 - User setting and preferences view 68
Figure 13 - Feedback form view 69
Figure 14 - Backend Test Results: Output from running automated tests with npm test 87
Figure 15 - Frontend Test Results: Output from running automated tests 91
Figure 16 - Conversation Flow Diagram 95
Figure 17 - UI screenshot of starting a new conversation 96
Figure 18 - Frontend logs showing the new conversation flow 97
Figure 19 - Backend logs showing the new conversation flow 97
Figure 20 - UI screenshot of continuing an existing conversation 98
Figure 21 - Frontend logs showing the flow of continuing an existing conversation 99
Figure 22 - Backend logs showing the flow of continuing an existing conversation 100
Figure 23 - Feedback dimensions and possible classified values used in the NLP analysis 113
Figure 24 - Confidence threshold filtering process for NLP classification 115
Figure 25 - Results from validation prompt per user 123
Figure 26 - Software engineering-related prompts used for validation testing 124
Figure 27 - Out-of-scope prompts used for validation testing 125
Figure 28 - Summary of test results and conclusions 126

1. INTRODUCTION

In the era of advanced AI language models, users often turn to platforms like ChatGPT, Claude, and Gemini to obtain support in learning and problem-solving. While these platforms are powerful individually, each operates in isolation, requiring users to evaluate and compare responses on their own. This fragmented experience may lead to inefficiencies, inconsistencies, and a lack of personalization.

SmartChat is an intelligent chat interface designed to address these challenges by integrating multiple AI platforms into a single point of contact. The project aims to deliver accurate, relevant, and personalized answers tailored to each user's unique profile, level of expertise, and feedback history.

SmartChat allows users to submit queries or requests, which are then processed through an advanced prompt-engineering mechanism. The refined prompt is sent to three separate AI agents – ChatGPT, Claude, and Gemini – and the resulting responses are evaluated and compared using structured criteria such as relevance, accuracy, and clarity. The system then selects and delivers the most suitable answer, optimizing both the user experience and the quality of the information provided.

1.1. Problem Description

As artificial intelligence platforms continue to advance, users increasingly turn to AI-powered chat systems to support learning, solve complex problems, and assist with technical tasks. However, widely used platforms such as ChatGPT, Claude, and Gemini function in isolation, requiring users to engage with each one individually. This fragmented approach creates inefficiencies, disrupts workflow continuity, and leads to inconsistent response quality.

A key limitation of the current landscape is the absence of a single point of contact. Users seeking to leverage the strengths of multiple AI platforms must manually query each system, compare diverse responses, and subjectively determine which answer is most suitable. This process is time-consuming and error-prone, especially given that each AI model exhibits unique strengths and weaknesses–no single model consistently produces the best outcome across all query types.

A major limitation of existing chat platforms is their inability to maintain continuity across conversations. These systems do not retain user profiles, preferences, or the full history of prior interactions.

As a result, users are forced to repeatedly include relevant background information in each prompt, often leading to generic responses that fail to align with their specific expertise level or current task.

Moreover, there is currently no automated mechanism to evaluate, validate, and consolidate multiple AI-generated answers into a single, high-quality response. Users must rely on manual judgment without the support of structured assessment criteria, which undermines reliability and consistency—particularly in domains such as software engineering, where clarity, precision, and contextual understanding are essential.

SmartChat addresses these challenges by introducing an integrated platform that:

- Serves as a single point of contact for accessing multiple AI chat systems
- Automates comparative evaluation to select the most suitable answer
- Personalizes interactions using user-specific data such as preferences, expertise level, and past feedback

Through this unified and intelligent approach, SmartChat significantly improves the efficiency, quality, and relevance of AI-assisted conversations in software engineering and beyond.

1.2. Motivation

In an era where artificial intelligence has become a significant tool, there is a growing demand for systems capable of delivering the most suitable response for each user—responses that are both accurate and personalized to match their specific needs.

Although platforms like ChatGPT, Claude, and Gemini offer impressive capabilities, users are currently required to interact with each one separately, compare the responses, and choose the most suitable answer on their own.

The need for a single tool for integrating multiple AI platforms and performing the comparison and adaptation automatically is what led to the development of SmartChat. The idea is to create an intelligent system that understands context, adapts to the user, and provides the best possible answer.

1.3. Goals

Our main goal is to deliver a single point of contact, user-friendly platform that provides the most suitable response by seamlessly integrating multiple AI platforms and applying personalized adaptations. This approach ensures that users receive accurate, context-aware answers tailored to their specific needs–without the burden of manually comparing responses or repeatedly specifying their preferences and background in each prompt. Instead, users can simply ask their query, while the system automatically handles personalization and response selection on their behalf.

1.4. System Overview

SmartChat is a software platform that aims to revolutionize how users interact with multiple AI chat platforms by providing a single, streamlined interface. Designed with a focus on streamlining complex learning processes, this platform integrate multiple AI platforms - such as ChatGPT, Gemini, and Claude - to deliver precise, contextually relevant, and high-quality answers.

In doing so, the system serves a diverse audience, including students, educators, and professionals in software engineering, enabling them to engage with complex topics seamlessly. By integrating multiple AI platforms into one cohesive solution, the system overcomes the limitations of individual chatbots, offering an enhanced learning experience tailored to each user's needs.

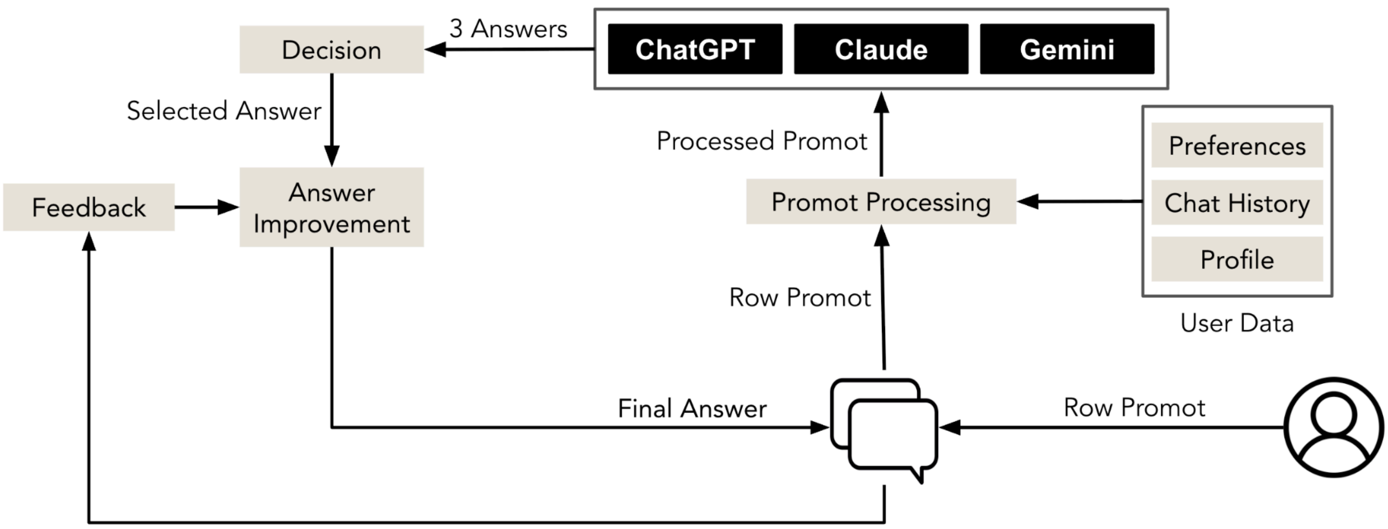

Figure 1 - System Overview Diagram

1.5. Usage scenarios

To explain the ideas behind SmartChat, several scenarios are introduced in the following sections.

1.5.1. Scenario 1 - First-Year Computer Science Student

Scenario:

Hila is a first-year computer science student struggling to grasp basic programming concepts. She opens SmartChat and submits the question: "What is recursion in Python?"

User Profile:

- **Role**: Student
- **Expertise Level**: Beginner
- **Answer Style**: Concise
- **Example Count**: One
- **Tone**: Neutral

The system:

- Retrieves Hila's profile, identifying her as a beginner.
- Enriches her prompt with contextual information and preferences.
- Sends personalized prompts to all connected platforms (**ChatGPT**, **Gemini**, **Claude**).
- Collects the responses, performs cross-evaluation, and merges the answers.
- Returns a single concise and easy-to-understand explanation, including one relevant example.

Follow-up:

After reviewing the answer, Hila feels she would benefit from seeing more than one example to better understand the concept. She updates her user settings, changing the **Example Count** from **1** to **2**, and asks the same question again.

This time, the system delivers a concise explanation accompanied by **two** relevant examples, tailored to her updated preferences. Hila finds the new answer clearer and is satisfied with the result.

This experience helps Hila understand a difficult concept with minimal effort, without needing to compare different AI responses herself.

1.5.2. Scenario 2 - Professional Software Engineer

Scenario:
David is a senior backend engineer working on optimizing query performance in a large-scale database. He asks SmartChat: "How can I improve MongoDB aggregation pipeline performance for large datasets?"

User Profile:

- **Role:** Software Engineer
- **Expertise Level:** Advanced
- **Answer Style:** Detailed
- **Example Count:** Multiple
- **Tone:** Neutral

The system:

- Retrieves David's profile, recognizing his advanced expertise and preference for detailed technical explanations.

- Enriches the prompt with his role, expertise level, and preference for multiple examples.
- Sends the enriched prompt to **ChatGPT**, **Gemini**, and **Claude** via their APIs.
- Collects the responses, evaluates them for accuracy, relevance, and clarity.
- Merges the best insights from each platform into a single, comprehensive answer with several optimization strategies and practical code examples.

Outcome:
SmartChat saves David significant time and effort. Instead of manually researching and comparing answers, he receives a single, high-quality response containing diverse optimization strategies and well-structured examples. This allows him to immediately apply the recommendations to his project without the need for additional research or validation.

1.5.3. Scenario 3 - Software Engineering Lecturer

Scenario:
Oren is a software engineering lecturer teaching an **Algorithms** course. He is preparing a practice presentation for his next class and wants to include clear, well-structured examples to help students understand the **Divide and Conquer** algorithmic approach. He opens SmartChat and asks:

> "Provide multiple educational examples of Divide and Conquer algorithms with step-by-step explanations for teaching purposes."

User Profile:

- **Role:** Lecturer
- **Expertise Level:** Advanced
- **Answer Style:** Detailed
- **Example Count:** Multiple
- **Tone:** Formal

The system:

- Retrieves Oren's profile, identifying him as an expert who prefers educational, multi-example explanations.
- Enriches the query with his teaching context, role, and requirement for step-by-step explanations.
- Sends the tailored prompt to **ChatGPT**, **Gemini**, and **Claude**.
- Collects their responses, evaluates them for clarity, educational suitability, and accuracy.
- Merges the strongest explanations and examples into a **single cohesive teaching resource**, including multiple illustrated algorithm examples such as **Merge Sort**, **Quick Sort**, and **Binary Search**, each with step-by-step breakdowns and diagrams.

Outcome:
SmartChat provides Oren with a ready-to-use set of educational examples that are both technically accurate and pedagogically effective. This saves him valuable preparation time and ensures that his presentation delivers clear, high-quality learning material to his students.

1.5.4. Scenario 4 - Tech Blogger Creating Educational Content

Scenario:
Sara runs a technology blog aimed at helping non-technical readers understand programming concepts. She asks SmartChat:

> "Explain APIs in simple terms with examples."

User Profile:

- **Role:** Software Engineer
- **Expertise Level:** Intermediate
- **Answer Style:** Detailed
- **Example Count:** Multiple
- **Tone:** Casual

The system:

- Retrieves Sara's profile, noting her preference for a conversational tone and illustrative examples.
- Enriches the query with her role and audience context to ensure accessibility.
- Sends tailored prompts to **ChatGPT**, **Gemini**, and **Claude**.
- Collects responses, evaluates them for clarity and suitability for a non-technical audience.
- Produces a final, unified explanation that uses plain language, relatable analogies, and multiple real-world examples, making it perfect for her blog readers.

Feedback usage:
After receiving the answer, Sara uses SmartChat's feedback feature to help improve future responses. She provides:

- A **rating** of **9/10** for the overall quality.

- An **open-text comment**: *"This was great, but I'd love even more real-world examples for future posts."*

The system analyzes her feedback, processes the open-text comment using its **NLP feedback model**, and classifies it under *"Request for more examples"*. This insight is saved in Sara's user profile so that future answers automatically include a greater variety of examples without her needing to request it again.

Outcome:
Sara receives high-quality, ready-to-use educational content while SmartChat learns from her feedback to make future answers even more aligned with her needs.

1.6. Audience

The SmartChat system is designed to serve individuals involved in the field of software engineering, with varying levels of technical expertise and different learning needs. The main target audiences include:

1. **Educators and Lecturers** : who seek high-quality, clear, and well-structured explanations that can be used to complement teaching materials, serve as enrichment resources, or provide in-class support.

2. **Software Engineering Students** : who are looking for assistance with the topics they are currently studying.

3. **Software Engineers and Developers** : who rely on the system to receive accurate, concise, and context-aware answers to technical questions, compare different approaches, and validate solutions.

1.7. Glossary

The following terms are divided into two categories: generic and SmartChat terms.

1.7.1. SmartChat Terminology

1. **User Profile**

   SmartChat maintains a **User Profile**, storing key attributes about the user. The system uses this profile to personalize responses according to the user's needs and preferences.

   The User Profile contains, among other things, the following elements:

   - **Role** - Identifies the user's position, such as *Student*, *Lecturer*, or *Software Engineer*.
   - **Expertise Level** - Specifies the user's knowledge level, which can be *Beginner*, *Intermediate*, or *Advanced*.
   - **Preferences** - Define how responses should be presented:
     - **Answer Style** - Determines whether the response should be brief (*Concise*) or more detailed (*Detailed*).
     - **Example Count** - Specifies how many examples should be included in the answer (*None*, *One*, or *Two*).
     - **Tone** - Controls the style in which the answer is written: *Formal*, *Casual*, or *Neutral*. This ensures that explanations match the expected communication style of the user.

2. **Prompt Processing**

   The process in which the system processes the user's input query (raw prompt) by incorporating information from the user profile and the current context, in order to generate a more accurate and effective prompt for the AI platforms.

3. **Cross-Platform Evaluation**

   A mechanism in which each platform rates the responses received from the other platforms based on predefined criteria such as relevance, accuracy, and clarity. Each platform assigns a score from 1 to 10 to the others' responses, and the system calculates an average score for final ranking.

4. **Factual accuracy**

   The correctness, reliability, and precision of the information.

5. **Answer Merging**

   After selecting the response with the highest score, the system performs an intelligent merging process, combining unique insights and valuable content from the remaining responses. This results in one unified, high-quality, and comprehensive answer that includes the best elements from all platforms.

6. **Feedback**

   The storage of insights from the user ratings, comments, and optional feedback after receiving an answer, allowing the system to learn over time, improve response quality, and enhance future personalization.

1.7.2. Generic Terminology

1. **Artificial Intelligence (AI)**

   A field within computer science focused on developing intelligent systems that simulate human cognitive abilities, such as learning, problem-solving, data analysis, and decision-making.

2. **Natural Language Processing (NLP)**

   A subfield of AI that concentrates on the understanding, processing, and generation of human language by computers, enabling natural communication between humans and machines.

3. **Chatbot**

   A software application based on NLP that enables conversation with users in natural language, via text or voice, to answer questions, provide assistance, and perform tasks.

4. **API (Application Programming Interface)**

   A set of rules, protocols, and tools that allow different software systems to communicate with one another, exchange data, and execute functions through predefined access points.

5. **Prompt**

   A query or textual input provided to an AI system, which generates a response based on it. A prompt typically includes a question or request, and may also contain context or additional information.

6. **Personalization**

   A process in which a system adapts the content or presentation of its responses according to the user's attributes and preferences, in order to enhance accuracy and user experience.

2. LITERATURE SURVEY

The development of the system addresses several challenges related to user interaction, response quality, and system efficiency. This literature survey reviews existing problems, proposed solutions, and evaluates their relevance and applicability to the project.

2.1. Problem Survey

Current AI-driven chat systems, such as ChatGPT, Claude, and Gemini, each demonstrate unique strengths but struggle to consistently deliver optimal results across all query types. Since these platforms operate independently, users must manually compare responses without structured evaluation, leading to cognitive overload and inconsistent outcomes–particularly in fields like software engineering where accuracy and clarity are essential.

The integration of multiple AI chat platforms into a single cohesive system introduces the following challenges:

1. **Multi-Platform Complexity [7]**

   Users interact with several AI platforms simultaneously, each with its own interface, input style, and response format. This fragmented experience makes it difficult to manage interactions efficiently and leads to inconsistent results across platforms

2. **Response Variability:**

   Each AI platform generates different responses [8]. As a result, users are faced with inconsistent answers across platforms and must manually compare the information to determine which response is the best for them.

3. **Limited Personalization:**

   Existing systems lack the capability to adapt responses based on user preferences, expertise levels, or interaction history.

4. **Feedback Utilization:**

   Most AI platforms do not leverage user feedback to improve future responses.

5. **Limited Adaptability:**

   Current solutions struggle to adjust to user intent or learning progress.

6. **Incomplete Fallback Mechanisms:**

   Handling ambiguous queries remains a challenge, leading to user dissatisfaction.

7. **Engagement Challenges:**

   Maintaining long-term user interest is a common shortfall of existing systems.

2.2. Solution Survey

To address the identified problems, various solutions and frameworks have been explored:

1. **Single Point of Contact**: Tools like the Jarvis Framework [1] demonstrate methods for integrating multiple chatbot platforms by standardizing prompts and managing responses through a centralized interface.

2. **Response Evaluation and Merging** [9]: Studies highlight the importance of evaluating responses based on criteria like relevance, accuracy, and clarity before consolidating them into a unified answer.

3. **Personalization Mechanisms**: Leveraging user profiles, interaction history, and preferences has been shown to enhance response quality and user satisfaction in conversational AI systems.

4. **Feedback-Driven Improvement** [10]: Incorporating feedback mechanisms enables adjustments to prompts and response strategies, improving system accuracy and relevance over time.

5. **Advanced Prompt Processing**: Develop structured prompts with templates and contextual elements for better AI responses.

2.3. Discussion and Conclusions

The surveyed literature confirms the feasibility of creating a single point of contact for integrating AI chat platforms, while also highlighting areas for improvement. Key takeaways include:

1. **Feasibility:**

   Existing frameworks like the Jarvis Framework [1] provide a strong foundation for cross-platform integration, but modifications are necessary for general-purpose use cases.

2. **Personalization:**

   Tailoring responses using user profiles and feedback is critical to delivering high-quality, relevant answers.

3. **Evaluation and Merging:**

   A structured approach to evaluating and synthesizing responses ensures consistency and reduces user effort.

This survey concludes that while significant challenges remain, the integration of multiple chat platforms into a single point of contact is both achievable and valuable. The project will incorporate best practices from existing solutions while introducing novel features like dynamic feedback-driven improvements and enhanced personalization.

3. TECHNOLOGICAL SURVEY

This section reviews the key technological challenges encountered during the development of SmartChat, alongside an analysis of relevant solutions.

3.1. Problem Survey

SmartChat addresses several key challenges related to integrating multiple AI chat platforms and providing the user with the most accurate and contextually suitable answer.

- **Fragmented User Experience**
  Each AI platform–such as ChatGPT, Claude, and Gemini–has its own interface, style, and tone. Traditionally, users must query each platform individually and manually compare answers, resulting in inefficiency, cognitive overload, and inconsistent results.

2. **Inconsistent Response Quality**
   The quality of responses varies significantly across platforms in terms of:
   - Relevance to the user's intent
   - Factual accuracy - the correctness, reliability, and precision of the information.
   - Clarity and coherence

   This variability makes it difficult to rely on any single answer without additional judgment or subject-matter expertise.

3. **Lack of Customization**
   Most platforms produce generic responses without adapting to:
   - The user's role
   - Preferred style, tone, or use of examples
   - Feedback history from prior interactions

   As a result, answers may be technically correct but fail to align with the user's preferred way of consuming information.

4. **Limited Use of Feedback**
   While some platforms allow users to rate responses or provide comments, they typically do not leverage this feedback to improve future outputs. This limits long-term learning and personalization potential.

5. **Integration Complexity**
   Integrating and managing multiple external APIs (e.g., OpenAI's ChatGPT, Anthropic's Claude, Google's Gemini) presents several challenges:
   - API rate limits
   - Latency and performance issues
   - Inconsistent error handling
   - Platform-specific behaviors

   These issues make response merging and evaluation difficult to implement reliably.

6. **Data Privacy and Session Continuity**
   Delivering personalized responses across sessions requires securely storing user preferences and interaction history, while adhering to privacy and data protection standards. This adds complexity in:
   - Secure data storage
   - Authentication and token management
   - Session continuity and retrieval mechanisms

3.2. Solution Survey

To address the challenges outlined in the previous section, **SmartChat** was developed with carefully designed, customized solutions. These solutions leverage modern AI technologies, a modular architecture, and feedback-driven learning strategies to deliver accurate, relevant, and personalized responses.

1. Integration with Multiple External AI Platforms

SmartChat integrates three leading AI chat platforms:

- **ChatGPT**
- **Claude**
- **Gemini**

SmartChat processes the user's raw prompt together with user profile data. The resulting enriched prompt is then sent to all three platforms, and the system receives a response from each one.

2. Cross-Platform Answer Evaluation and Answer Merging

The system implements a Cross-Platform Evaluation process, where each AI platform (ChatGPT, Claude, Gemini) is asked to independently evaluate the responses generated by the other platforms.

As part of the cross-platform evaluation process, SmartChat sends an evaluation request to each AI platform, asking it to critically assess the quality of another platform's response. The evaluation focuses on three main criteria: **relevance, accuracy, and clarity**. Each platform is instructed to provide a **single overall score on a scale from 1 to 10**, with the expectation that scores remain objective and critical (i.e., only truly exceptional answers should receive the highest score). These evaluations are then collected and used as part of the decision-making process to identify the most suitable response.

In this way, each platform returns a single overall score (1–10) based on three predefined evaluation criteria:

- **Relevance**: Measures how well the response aligns with the user's question and intent, covering all required aspects without including irrelevant information.
- **Factual Accuracy**: Assesses the correctness, reliability, and precision of the information, avoiding logical or factual errors and providing valid examples where appropriate.
- **Clarity**: Evaluates how clear, readable, and understandable the response is, ensuring it is logically structured and written in a style appropriate to the user's expertise level.

It is important to emphasize that each platform conducts its evaluation independently, without being influenced by the scores or judgments of the other platforms. For example, When Claude evaluates ChatGPT's response, it does so solely on the basis of the given answer and the defined criteria, without knowing how Gemini has rated it. This approach ensures objectivity and reduces potential biases between platforms.

After the evaluation stage, the Answer Merging module aggregates all scores, identifies the response with the highest average score, and selects it as the base answer. In the case of a tie, a tie-resolution mechanism is applied: for the first user question the winner is chosen randomly, while for subsequent questions the decision is based on the historical win record of each platform for that specific user.

The merging logic then enhances the selected response by incorporating **unique insights** extracted from the other responses. The result is a final answer that is comprehensive, accurate, relevant, and clear – enriched with complementary perspectives to provide a high-quality and well-balanced output.

3. Personalized Answer Refinement Based on User Profile

Once the most suitable answer is selected, the system applies final refinements to the merged response, ensuring full alignment with the user's preferences and most recent feedback.

4. Feedback Collection and NLP-Based Analysis

After each interaction, the user is invited to provide feedback consisting of three components:

1. A numerical rating from 1 to 10
2. An open-text comment
3. A selection from predefined feedback questions

The collected feedback is analyzed to generate **insights** that the system uses to improve future responses. Specifically, the open-text comments are processed using a **trained NLP model** that classifies the feedback into predefined categories. These insights are then integrated into the system's personalization and adaptation mechanisms, allowing SmartChat to continuously refine its responses over time.

5. Modular and Flexible Cloud Architecture

The backend is implemented in **Node.js** with **Express** and **MongoDB**, combining internal logic with external API integration. The architecture follows a **modular design**, where each module is responsible for a specific stage in the processing pipeline.
This modularity facilitates easier testing, incremental updates, and scalable automation, ensuring the system remains maintainable and adaptable to future requirements.

3.3. Discussion and Conclusion

The technological challenges addressed in this project–such as inconsistent responses across platforms, insufficient personalization, and limited use of feedback–reflect the real-world difficulties of developing a single point of contact system.

- Insufficient Personalization: Existing platforms do provide some customization, but this typically requires users to restate their preferences or context in every prompt. What is missing is an automatic, persistent adaptation mechanism that continuously aligns answers with the user's role, expertise level, and communication style–without placing the burden on the user each time.

- Limited Use of Feedback: Current systems provide feedback options only in very limited ways, such as allowing users to refine the next prompt or give a simple thumbs up/down rating. However, they do not support the submission of structured feedback, such as rating scores, free-text comments, or predefined multiple-choice answers. As a result, feedback is not systematically analyzed or incorporated into future responses, limiting its effectiveness in improving personalization and long-term system adaptation.

SmartChat effectively overcomes these challenges through a **modular, scalable architecture** that integrates leading AI platforms–**ChatGPT**, **Claude**, and **Gemini**–into a seamless user experience. Importantly, the system **automatically makes the decision on the user's behalf** by selecting the most suitable response, eliminating the need for users to manually determine which platform provided the best answer. Likewise, SmartChat removes the burden of **manually crafting detailed and in-depth prompts for each query**, as it automatically enriches prompts with the user's preferences, role, and context.

Each system component plays a critical role in delivering this capability:

1. **User Management** - Maintains and provides user-specific data, including insights derived from the most recent feedback, for inclusion in prompt generation.

2. **Prompt Processing** - Generates a rich, in-depth, customized prompt based on the user's raw query, stored user data and insights, and chat history for contextual continuity.

3. **Cross-Platform Integration** - Sends the processed prompt to multiple AI platforms via their APIs and retrieves their responses.

4. **Answer Merging** – Executes a series of evaluation and synthesis steps that result in a unified, high-quality final answer.

5. **Feedback** – Enables users to submit feedback, which is analyzed and transformed into insights for improving future interactions.

6. **Communication** – Uses Socket.IO to provide smooth, interactive, and responsive conversational experiences.

By combining advanced prompt processing, intelligent multi-source answer synthesis, and adaptive feedback-driven learning, SmartChat delivers responses that are not only accurate and relevant but also personalized, efficient, and consistent.

This work demonstrates both the **feasibility** and **value** of integrating multiple AI systems into a single, intelligent platform. It also establishes a foundation for future systems capable of **dynamically adapting** to user needs in educational, professional, and other domain-specific contexts.

4. REQUIREMENTS AND SPECIFICATION

This section outlines the system's functional and non-functional requirements, as well as the overall scope and development approach of the SmartChat project.

4.1. Functional Requirements

The following section defines the core functional requirements of the SmartChat system. These requirements describe the behavior of the system from the user and system perspectives, ensuring that all critical functionalities are implemented to meet the project goals.

1. **User Registration and Authentication**

   1.1. The system shall allow users to register with an email and password.

   1.2. The system shall support secure user login and logout operations.

   1.3. The system shall verify user credentials and manage session tokens securely.

2. **Chat Interface**

   2.1. The system shall allow users to submit free-text questions via a chat interface.

   2.2. The system shall allow users to submit feedback via a button in the chat interface.

   2.3. The system shall allow users to edit their profile via a button in the chat interface.

3. **Intent Recognition**

   3.1. The system shall use **Dialogflow** to recognize the intent of user queries and determine whether they fall within the scope of the software engineering domain.

   3.2. If a query does not match the intended domain, the system shall either reject the query or route it through a fallback mechanism.

4. **User Profile Management**

   4.1. The system shall allow users to define and update their personal profile.

   4.2. The system shall store user preferences for use in future prompt generation.

4.3. The system shall associate feedback insights with the user profile.

5. Prompt Processing

5.1. The system shall enrich the raw prompt with:

- User profile data
- Historical chat context
- Feedback-based insights

5.2. The system shall construct a personalized prompt to be sent to external AI platforms.

6. Cross- Platform Integration

6.1. The system shall connect to multiple AI platforms via their APIs (e.g., ChatGPT, Claude, Gemini).

6.2. The system shall send the processed prompt to all connected platforms concurrently.

6.3. The system shall receive and log responses from each platform.

7. Answer Evaluation and Merging

7.1. The system shall evaluate the responses using criteria such as relevance, clarity, and factual accuracy.

7.2. The system shall perform a meta-evaluation, where each platform scores the others.

7.3. The system shall select the best response and merge it with valuable insights from other responses.

7.4. The system shall refine the final answer based on the user's preferences before presenting it.

- ▪ Select the highest-rated response as a base
- ▪ Enrich it with unique insights from other answers
- ▪ Finalize the merged answer according to the user's preferences

8. Feedback Collection and Analysis

8.1. The system shall allow users to provide structured feedback after each interaction, including:

- A numerical rating (1-10)
- Free-text comment
- Selection from predefined feedback options

8.2. The system shall process free-text feedback using an NLP model to classify it into actionable categories.

8.3. The system shall analyze the feedback and store derived insights for future use.

9. Communication

9.1. The system shall enable message exchange between the frontend and backend using Socket.IO.

9.2. The system shall display generated responses.

Note: While the system supports response display, it does not guarantee minimal latency. Due to the multiple API calls to external platforms and the functional complexity of the backend processing pipeline, response times may vary.

9.3. The system shall ensure session persistence and stable communication throughout the interaction.

4.2. Non - Functional Requirements

This section delineates the non-functional requirements that underpin the foundation of SmartChat.

Implemented

1. **Chat Interface Development**

   Initially, we considered using Slack as the chat interface due to its robust infrastructure and built-in features. However, we found that Slack could not provide the full level of functionality and visibility we required for a final project. To address these limitations and gain greater control over the system behavior, we chose to develop a custom chat interface. This approach allowed us to tailor the system precisely to our needs and integrate advanced features such as customizable user interactions, and complete data flow visibility.

Not Implemented

1. **Security**

   While the platform incorporates standard security measures provided by the tools and frameworks we used, we chose not to explore advanced security considerations within the scope of this project. Security is handled through the built-in capabilities of these tools, and our documentation does not extend beyond what is inherently provided by them.

2. **Performance Guarantees**

   The system does not guarantee minimal latency. Due to the multiple API calls to external platforms and the functional complexity of the backend processing pipeline, response times may vary.

4.3. Specification: The Scope of the SmartChat Project

SmartChat is an innovative platform designed to integrate multiple AI platforms into a single point of contact interface. The system aims to deliver accurate, high-quality, and tailored responses, eliminating the need for users to manually compare outputs from different AI platforms.

SmartChat supports the creation of detailed user profiles, including attributes such as role, expertise level, tone, and preferred answer style. This information is used to generate enriched, context-aware prompts that are sent to ChatGPT, Claude, and Gemini.

Through a combination of cross-platform response evaluation, intelligent answer merging, and NLP-based feedback analysis, the system produces a single, unified, and contextually relevant answer for each user query.

The current scope of the project is focused exclusively on the software engineering domain, ensuring that all queries, prompt processing, and evaluation are aligned with this field. Additionally, the system prioritizes technical integration, personalized response processing, and a smooth user experience via both Slack and custom web-based interfaces.

4.4. The Approach

SmartChat was developed as an innovative system that enables users to receive accurate, unified, and personalized answers by simultaneously integrating multiple AI platforms. The platform is designed as an intelligent and integrative solution that simplifies the response process and enhances the user experience when interacting with AI systems.

Instead of requiring users to analyze responses from each AI platform separately, SmartChat sends a customized prompt to ChatGPT, Claude, and Gemini based on the user's profile. It then performs a cross-platform evaluation, intelligently merges the responses, and delivers a single, high-quality, consolidated answer.

Each user has a personalized profile that includes their role, expertise level, preferred tone, answer style, and desired number of examples. These attributes are used during prompt construction to ensure responses are context-aware, relevant, and tailored to the user.

To ensure that queries fall within the intended software engineering domain, SmartChat uses Dialogflow for intent recognition. If the intent is not recognized as software-engineering-related, the system routes the query through a fallback mechanism, ensuring domain-specific accuracy.

Throughout the interaction, SmartChat collects user feedback, including both numerical ratings and open-text comments. This feedback is analyzed using a dedicated NLP model, which extracts actionable insights that are incorporated into the user's profile and used to continuously improve the quality, relevance, and personalization of future responses.

The platform operates through a chat platform interface React-based web interface, with message communication powered by Socket.IO. It maintains chat session state and presents all responses seamlessly as if they were coming from a single intelligent assistant.

This approach enables a smooth, intelligent, and dynamic user experience, merging the capabilities of multiple AI platforms into one cohesive interface. It brings to life the true potential of cross-platform AI integration with personalized interaction and continuous learning at its core.

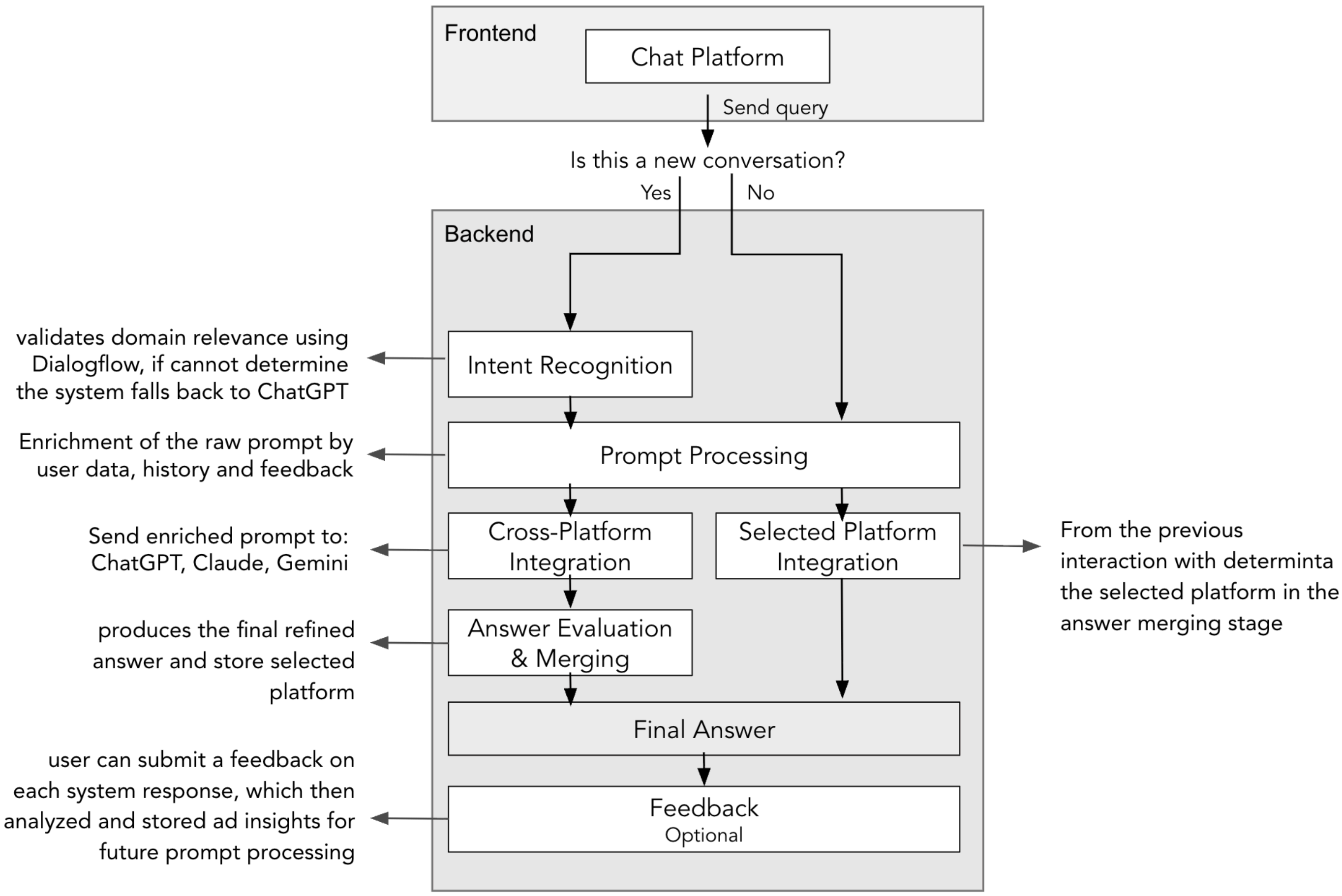

Figure 2 - Diagram describing the approach of smartChat

The diagram shows how SmartChat processes user queries for both new and existing conversations. New queries go through **Intent Recognition** (Dialogflow with ChatGPT fallback), **Prompt Processing**, and **Cross-Platform Integration** to gather responses from multiple AI platforms, followed by **Answer Evaluation & Merging** to produce the final answer. Existing conversations bypass the multi-platform step and continue with the **previously selected platform**. In both cases, enriched prompts use user data, history, and feedback insights. Final answers are displayed to the user, and optional feedback is analyzed to improve future responses.

5. THE ARCHITECTURE

SmartChat is built on a **client-server architecture** to support seamless interaction between the user interface and backend services. The client is a **React-based web application** hosted on **Render**, communicating with a **Node.js/Express backend** (also on Render) via **HTTP** for standard requests and **Socket.IO** for event-driven interactions.

When a user interacts with the chat platform, **s**ocket custom events trigger specific handlers, which call the necessary backend components to process the request. A message can either **start a new conversation**—processed through the f**ull SmartChat pipeline** from intent recognition to answer merging—or **continue an existing conversation**, in which case it is **routed only to the AI platform selected** in the previous merging step.

Persistent data, including user profiles, chat history, AI responses, feedback, and insights, is stored in a **MongoDB database**.

The **technology stack** includes:

- **Frontend:** React with Socket.IO client (Render hosting)
- **Backend:** Node.js/Express with Socket.IO server (Render hosting)
- **Database:** MongoDB for all stateful data
- **External Services:** Dialogflow for intent recognition; ChatGPT, Claude, and Gemini for AI responses

This architecture enables SmartChat to integrate multiple AI platforms, deliver personalized responses, and refine output through intelligent evaluation and feedback analysis.

5.1. SmartChat Data

SmartChat stores and processes several categories of structured data in **MongoDB**, supporting personalization, conversation continuity, system evaluation, and feedback-driven improvements. The main collections and their purposes are outlined below.

5.1.1. The Internal and External Representations

- **Internal Representation**
  Each conversation is stored as a structured sequence of messages, where every message record includes the user identifier, chat identifier, timestamp, and the full text of the message. Alongside these core fields, each message also contains a flag indicating whether feedback was provided ("GotFeedback"). In parallel, the system maintains a dedicated User schema. This schema stores user registration details, authentication data, and personalized preferences–such as answer style, tone, or level of detail–along with generated insights derived from feedback analysis. Importantly, users can edit their preferences at any time through the frontend, and these updates are immediately reflected in the database. Finally, feedback data itself is persisted as a separate collection, linked to both the user and the corresponding conversation. Each feedback entry includes numerical ratings, free-text comments, and the NLP-derived insights that summarize qualitative aspects like clarity, detail, or tone.

- **External Representation**
  The external representation of SmartChat data is realized through the user interface, which functions as the primary chat platform. Within this interface, the user can initiate a new conversation by sending a first message, or alternatively, continue an existing conversation by adding further messages to it. Users are also able to view a list of all their conversations and open any of them to review the history and continue interacting.

  For every system-generated response displayed in the chat, the user is given the opportunity to submit feedback, which may include:

  - A numerical rating on a scale from 1 to 10
  - A free-text comment
  - Responses to predefined feedback questions (via selectable options)

  In addition, the interface allows users to edit their personal details and preferences, ensuring the system can tailor responses to their evolving needs. Core account management features are also supported, including signing in, signing out, and registering as a new user.

5.1.2. Initial Start-up Data

At user registration, the system collects initial preferences and stores them in the User schema. These preferences guide prompt processing and can later be updated through the frontend.

When starting a new conversation, the system initializes:

- The chat session record,
- An empty message list,
- Connection to the user's stored profile preferences,
- Feedback structures for collecting interaction data.

5.1.3. Data Collection

During each chat session, SmartChat collects and persists:

- **User input** (raw query and enriched prompt).
- **AI platform responses** (single or multiple, depending on pipeline path).
- **Evaluation results**, including scoring and selected platform decision.
- **Final merged answer**, which may integrate multiple platform responses.
- **Feedback data**, stored and associated with the user.

If the session is a **continuation of an existing conversation**, the system skips cross-platform integration and only queries the **previously selected platform**.

5.1.4. View Persistence

SmartChat ensures persistence of all system data:

- **Conversations** - Users can review their chat history or continue past discussions seamlessly.
- **Profiles** - Preferences and NLP-derived insights persist across sessions, ensuring continuous personalization.
- **Feedback** - Collected ratings and qualitative insights remain stored and are used to refine system behavior.

5.2. SmartChat Processes

The SmartChat system follows a structured, event-driven client-server process flow that ensures personalized prompt creation, multi-platform AI integration, and continuous improvement through feedback analysis. The main processes are described below.

User Identification & Session Initialization

- When a user accesses the SmartChat web interface, authentication is performed via the backend using stored credentials in the MongoDB database.
- Upon successful login, the system retrieves the user profile, including role, expertise level, preferences, and relevant historical insights from past feedback.
- A persistent session is established over Socket.IO to enable bidirectional communication.

User Query Submission

- The client emits a message submit event with the raw query to the backend via Socket.IO.

Intent Recognition

- The backend sends the query to **Dialogflow** to determine whether it belongs to the software engineering domain.
- If **Dialogflow cannot determine the intent**, the system falls back to **ChatGPT** for intent recognition.
- If the query is classified as outside the domain, a fallback response is returned to the client without further processing.

Prompt Processing

- The backend enriches the raw query using:
  - User role and expertise level.
  - Tone, style, and example count preferences.
  - Insights from recent feedback to adapt future answers.
  - Relevant conversation history for maintaining context.

Cross-Platform AI Integration

- The processed prompt is sent to multiple external AI platforms (ChatGPT, Claude, Gemini) via their respective APIs.
- Each platform returns an independent response to the backend.

Supervisor (Evaluation and Merging)

- Each AI platform scores the other platforms' responses based on criteria such as clarity, accuracy, and relevance.
- The system selects the best-performing response as the base and integrates unique insights from others to produce a refined, merged output.
- The system refines the final answer according to the user's preferences.

Response Delivery

- The final unified answer is sent back to the client via Socket.IO for immediate display.

Feedback Collection & Processing

1. **Collection**

   After receiving the system's answer, the user can submit feedback via a dedicated UI component, providing:

   - A numerical rating (1-10).
   - A free-text comment (Qualitative).
   - Selections from predefined feedback questions (e.g., clarity, tone, length, examples).
2. **Storage**
   The backend creates a new Feedback document in MongoDB, linked to the corresponding Chat and User.

3. **Feedback Analyze and Insight Generation**

   Before saving (creating) the new feedback, the backend analyzes the submitted feedback and generates suggestions.

   The **free-text** feedback is analyzed using a **custom-trained NLP.js model** to identify improvement suggestions across key dimensions such as clarity, tone, length, level of detail, and example count.

   Ultimately, the suggestions are translated into **insight** associated with the feedback before it is stored in the database.

4. **Profile Update**
   - The generated insights are stored in the Feedback document and associated with the user profile.

- These insights are applied in future Prompt Processing to adapt responses to the user's preferences and past feedback.

5.3. Roles

The **SmartChat** system involves both **human** and **system-based** roles, each contributing to the end-to-end process of delivering accurate, personalized, and unified responses within the software engineering domain.

1. End User

- **Description:** The primary consumer of the system's capabilities. End users ask questions, receive merged and context-aware answers, and provide feedback to guide future improvements.

- **Profile Types:**

  - **Student** – Learns concepts, reviews examples, and requests explanations.
  - **Software Engineer** – Seeks targeted, implementation-ready answers for professional tasks.
  - **Lecturer** – Generates teaching materials, validates content, and creates structured examples for instructional use.

- **Personalization:** User role, expertise level, tone, preferred answer style, and example count are stored in their profile and directly influence prompt generation and response formatting.

2. System (SmartChat Core)

- **Description:** The backend engine that manages the full processing pipeline, including:

  - **Prompt Generation:** Builds enriched prompts based on user profile data.
  - **Integration:** Dispatches prompts to multiple AI platforms.
  - **Evaluation:** Compares and ranks responses based on relevance and quality.
  - **Merging:** Produces a single consolidated answer.
  - **Personalization:** Adapts future answers using insights from feedback.

3. External Systems

- **Description:** Third-party AI and NLU services integrated into SmartChat to provide response generation and intent recognition.
- **Components:**
    - **Dialogflow:** Performs intent recognition to determine whether a user query belongs to the **software engineering** domain. If it cannot be confidently determined, the system falls back to **ChatGPT-based intent recognition** to maintain domain enforcement.
    - **AI Platforms (ChatGPT, Claude, Gemini):** Generate candidate responses for each enriched prompt and participate in cross-platform evaluation.

5.4. Architectural Description

The **SmartChat** system is designed to seamlessly integrate multiple AI platforms into a single point of contact conversational experience.

User queries, submitted via the web interface, are sent through **Socket.IO** to the backend, where **Dialogflow** (with a ChatGPT fallback) verifies that the query belongs to the software engineering domain.

The **Prompt Processing** component then generates a personalized, context-rich prompt based on the user's profile, chat history and feedback insights. This enriched prompt is sent to **ChatGPT**, **Claude**, and **Gemini**, whose responses are collected, cross-evaluated, and merged into a refined final answer.

On each response, the user can submit **feedback**, which is analyzed into insights, these insights are stored and applied to improve future interactions.

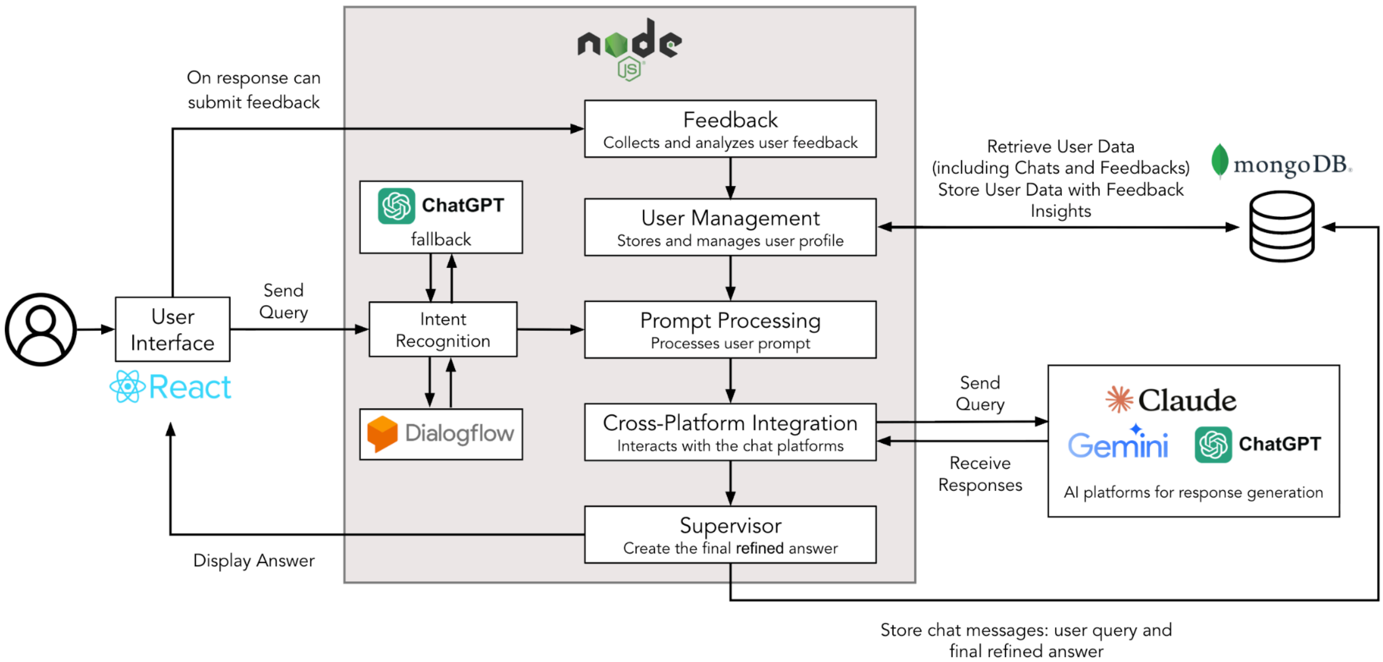

Figure 3 - System Architecture

Architecture Explanation

The SmartChat system follows a client-server design in which a React web client communicates with a Node.js/Express backend over HTTP and Socket.IO for events. Users submit queries through the **User Interface**, which emits a message submit event to the backend.

1. **Intent Recognition**
   The backend first validates domain relevance using **Dialogflow**. If Dialogflow cannot determine the intent with sufficient confidence, the system falls back to **ChatGPT-based intent recognition**. Out-of-scope queries receive a fallback response without further processing.

2. **User Management**
   The **User Management** component retrieves the user's profile (role, expertise level, tone, answer style, example count) , chat history and any stored insights derived from prior feedback. Users may also **edit their profile** directly from the UI; updates are persisted and immediately affect subsequent prompts.

3. **Prompt Processing**
   The **Prompt Processing** component constructs a personalized, context-aware prompt by combining the raw query, user profile, recent chat history, and feedback-based insights. This ensures consistent, role-appropriate guidance for downstream models.

4. **Cross-Platform Integration**
   The enriched prompt is dispatched to multiple external AI platforms—**ChatGPT**, **Claude**, and **Gemini**—via their APIs. Responses are collected along with timing and status metadata.

5. **Supervisor (Evaluation & Merging)**
   Returned answers are compared, including **cross-evaluation** signals (where available). The Supervisor selects the best candidate, merges complementary insights from the others, and produces the **final refined answer**, which is streamed back to the client via Socket.IO.

6. **Feedback**
   After viewing the answer, the user may submit **feedback** (rating `1-10`, free-text comment, and predefined selections). The system analyzes the free text using a custom **NLP model** to extract dimension–value insights (e.g., clarity: simplify; tone: neutral; examples: multiple). These insights are stored and fed into future prompt processing for continuous personalization.

7. **Data Persistence (MongoDB)**
   MongoDB persists user profiles, chats, messages, feedback, insights, and response artifacts. The backend reads profiles and recent history during prompt construction and writes messages, merged answers, and feedback results for auditability and continuity.

Operational note: Minimal latency is not guaranteed due to multiple external API calls and backend processing complexity.

5.5. The life-cycle of a system

The SmartChat system operates through a continuous, iterative cycle that ensures accurate, personalized responses and ongoing improvement over time. Each interaction follows a structured sequence, from initial query submission to feedback-driven refinement, allowing the system to adapt to individual user preferences and maintain alignment with the software engineering domain.

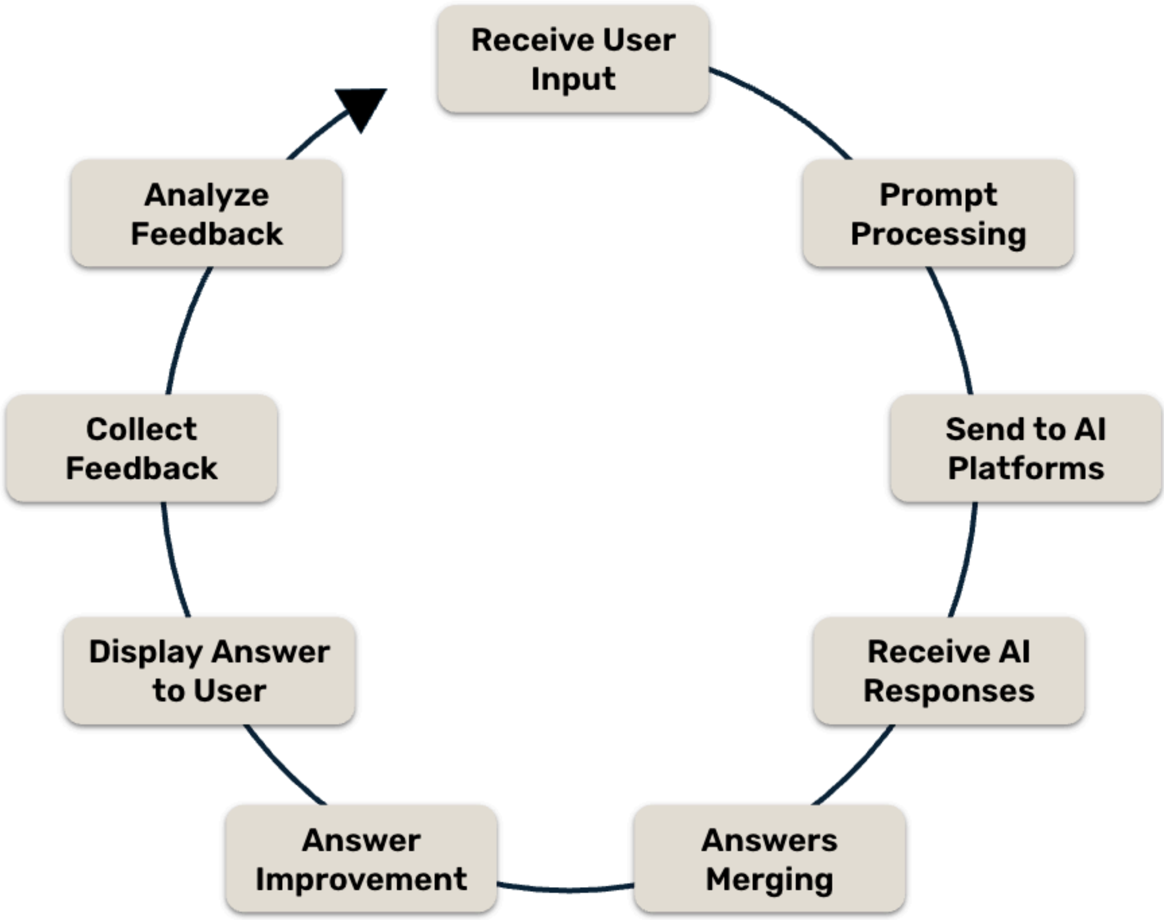

Figure 4 - The system life cycle

Stages Explanation

1. **User Input**
   The user submits a query via the React-based interface.

2. **Prompt Processing**
   The backend transforms the raw query into a personalized prompt, incorporating profile details, preferences, and historical insights.

3. **Send to AI Platforms**
   If the message [1 -user input] starts a new conversation, it will follow the full SmartChat pipeline: Intent Recognition → Prompt Processing → Cross-Platform Integration → Answer Merging. If it continues an existing conversation, the system bypasses multi-platform processing and sends the query only to the previously selected AI platform (stored on user data).

4. **Receive AI Responses**
   Candidate responses are returned from the AI platforms for further processing.

5. **Answers Merging**
   The Supervisor evaluates the responses, selects the best one, and may integrate additional insights from other platform outputs.

6. **Display Answer to User**
   The refined answer is returned to the user interface for immediate display.

7. **Collect Feedback**
   The user may provide ratings, qualitative feedback, and responses to predefined questions. Feedback is stored in MongoDB and analyzed using a custom NLP model.

8. **System Improvement**
   Feedback is converted into structured insights and stored in the user profile, influencing future prompt construction and improving answer quality.

6. SYSTEM DESIGN

The following chapter discusses the system design of SmartChat, emphasizing how it efficiently manages diverse data types and ensures seamless interaction in a chat environment. The chapter reviews the internal and external representations of data, presenting the user interfaces and the processes involved in handling messages between the user and the system. In addition, it details the core process components and communication technologies, highlighting their contribution to the continuous operation of integrating multiple AI platforms, merging personalized answers, and leveraging user feedback to achieve ongoing improvement in response quality.

6.1. Data Components

The SmartChat system employs a modern data management architecture tailored for multi-platform AI interactions. At its core, MongoDB serves as the primary NoSQL database, organizing information into collections that represent users, chats, messages, feedback, platform responses, and evaluation results. This flexible schema ensures persistence for historical records while supporting the dynamic and high-frequency nature of live chat environments.

Key collections include User (profiles, authentication, preferences), Chat (conversation metadata, timestamps, participants), Message (user prompts, raw AI platform responses, merged answers), and Feedback (ratings, free-text comments, NLP-based classifications). In addition, PlatformResponse and Evaluation entities capture cross-platform evaluation scores, insights, and tie-resolution records produced by the Answers Merging Module.

The backend repositories implement CRUD operations across these collections, ensuring efficient and consistent access patterns. On top of this data layer, service modules integrate with external AI platforms (ChatGPT, Claude, Gemini), retrieve raw answers, and feed them into the merging pipeline modules – including crossPlatformEvaluation, combiningEvaluations, answerMerging, insightsExtractor, and refineAnswerWithPreferences. These modules process evaluations, consolidate insights, and persist refined answers enriched with metadata that supports both operational decision-making and analytical queries.

On the frontend side, structured JSON payloads are exchanged with the backend via REST APIs (for chat and user management) and WebSockets (for message streaming). This architecture guarantees synchronization of user preferences, evaluation results, and merged responses across all clients, while maintaining the low latency required for an interactive chat experience.

By combining MongoDB's document-oriented flexibility with SmartChat's modular merging pipeline, the system achieves a balanced data strategy that supports scalability and persistence responsiveness – all of which are critical to delivering accurate, adaptive, and personalized answers in a live multi-platform chat environment.

6.1.1. Internal Representation

The internal representation of data within SmartChat is designed to support efficient processing, merging, and personalization of answers in a multi-platform AI chat environment.

For the SmartChat system, data structures are organized into objects representing Users, Chats, Messages and Feedback as illustrated in the class diagram. These entities are implemented as modular classes, enabling easy manipulation and access to relevant information.

- The User class manages profile details, authentication, and preferences that guide personalization.
- The Chat class encapsulates the overall conversation context, including participants and creation time.
- The Message class manages the flow of communication, storing user prompts, raw AI responses, and merged answers.
- The Feedback class records user ratings, comments, and NLP-based classifications, enabling iterative system improvement.

This modular representation ensures SmartChat can manage dynamic interactions, evaluate responses, and refine answers.

User Data

The **User** collection stores identity, role, and personalization data for each registered user. Key fields include:

- **Personal Details** – first_name, last_name, email (login and identification).
- **Authentication** – Encrypted password, JWT methods for secure token generation and verification.
- **Role & Expertise** – role (Student, Lecturer, Software Engineer) and expertiseLevel (Beginner, Intermediate, Advanced) to influence prompt construction.
- **Preferences** – preferences.answerStyle, preferences.exampleCount, preferences.tone define desired answer characteristics.
- **Performance Tracking** – platformWins counters for ChatGPT, Claude, and Gemini to record historical selection success rates.
- **History References** – feedbackHistory (array of Feedback IDs) and conversationHistory (array of Chat IDs) to maintain user interaction history.

Chat Data

The **Chat** collection represents an individual conversation session.

Fields include:

- **User Association** - userId links the chat to its owner.
- **Metadata** - title, createdAt, iterations (number of exchanges), selectedPlatform (if applicable), isCompleted.

  The selectedPlatform field stores the AI platform chosen in the answer merging step. For continued conversations, this value ensures that subsequent queries are routed only to this platform instead of repeating full multi-platform processing.

- **Content References** - userPrompt (initial query) and messages (array of Message IDs).
- **Feedback Link** - feedback holds a **reference (ObjectId)** to a **single Feedback document**, representing the **specific feedback entry** associated with this chat session.
- **State Flags** - hasNewMessages for unread messages tracking.

Chats serve as the **container** for user-system exchanges, allowing context continuity across turns.

Message Data

The **Message** collection stores individual message instances exchanged during a chat.

Fields include:

- **Associations** - userId and chatId for relational mapping.
- **Content** - sender (e.g., user, SmartChat), text (message content), timestamp.
- **Feedback Tracking** - gotFeedback flag to indicate whether this message has received specific user evaluation.

This data supports **context retrieval** for prompt enrichment (in prompt processing) and conversation history display in the UI.

Feedback Data

The **Feedback** collection captures structured evaluations from users.

Fields include:

- **Basic Metrics** – rating (1-10) for overall quality assessment.
- **Qualitative Comments** – qualitative (free-text) for open-ended opinions.
- **Structured Details** – interactionDetails object containing targeted feedback categories (examplesEnough, detailLevel, clarity, tone, length).
- **Insights** – insights object containing generated suggestions (suggestionByRating, suggestionByQualitative, suggestionBySpecific). These insights are **automatically generated** by the system when a new feedback entry is created. The backend calls the analyzeFeedback(feedback) function, processes the filled feedback, and assigns the resulting analysis to feedback.insights before persisting the document (feedbackRepository.create(feedback)).
- **Timestamps** – createdAt for chronological tracking.

The **insights** are later retrieved by the **Prompt Processing** stage to refine future responses according to the user's preferences and past interactions.

System Data & Socket Events

Although not persisted in MongoDB, interaction state is handled via **Socket.IO events**.

Fields include:

- **User Identification** – identify_user links a socket to a User for session management.
- **Chat Lifecycle** – create_chat, request_chat_list, join_room, and send_message manage conversation flow.
- **Response Processing** – Chat and message events trigger the **ResponseProcessorHandler**, which calls AI platforms, collects responses, and stores them in the relevant Chat and Message documents.

These events connect the **runtime state** of the conversation with **persistent storage**, ensuring responsiveness while maintaining long-term records in MongoDB.

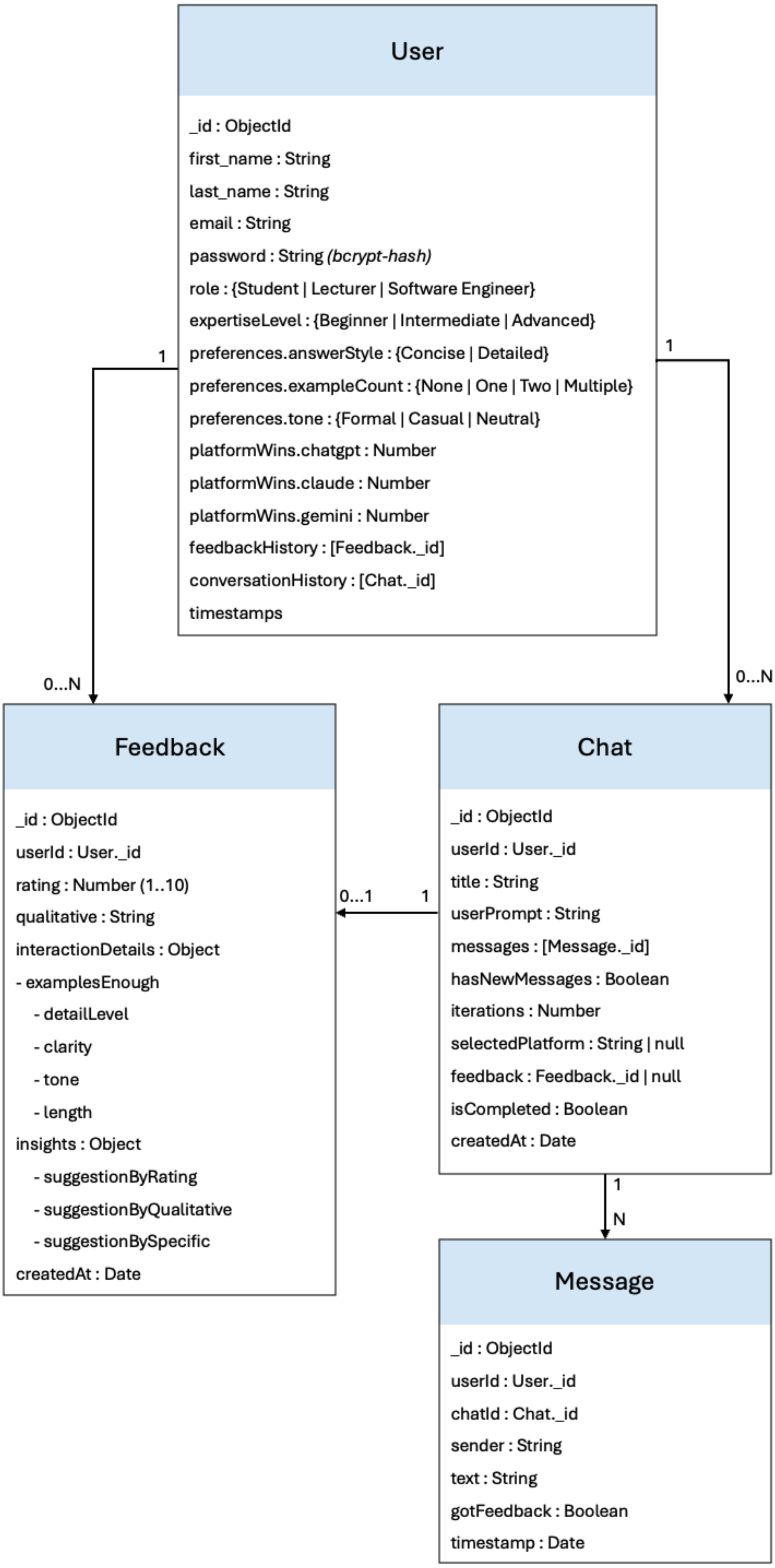

Figure 5 - SmartChat MongoDB Entity–Relationship Diagram

6.1.2. External Representation

The external representation of SmartChat is centered around an intuitive, chat-based interface designed for end-users seeking accurate, adaptive, and personalized answers. The SmartChat platform provides a structured environment where users can initiate conversations, manage ongoing chats, and review merged responses from multiple AI models. The interface supports core operations, including creating new chats, sending prompts, viewing raw platform answers, and receiving the final unified response enriched with insights.

Within the platform, each chat can contain multiple messages, and each message is enriched with associated evaluations, feedback, and tie-resolution data. The interface visually represents this lifecycle – from user query to merged answer – allowing users to clearly track how responses are generated and refined. Users can seamlessly transition between asking questions, providing feedback, and reviewing system improvements, ensuring a smooth interactive workflow.

The visual design of the SmartChat interface emphasizes simplicity and responsiveness, making it easy to manage multiple chat sessions simultaneously. Each conversation is represented as a structured thread of exchanges, with the ability to drill down into individual responses, platform scores, and user feedback. The platform also supports clear visualization of merged results, enabling users to see how final answers were shaped through cross-platform evaluation and preference refinement.

This structured yet flexible external representation ensures that users maintain full control over their interactions while receiving transparent, high-quality, and personalized insights through an intuitive and organized chat interface.

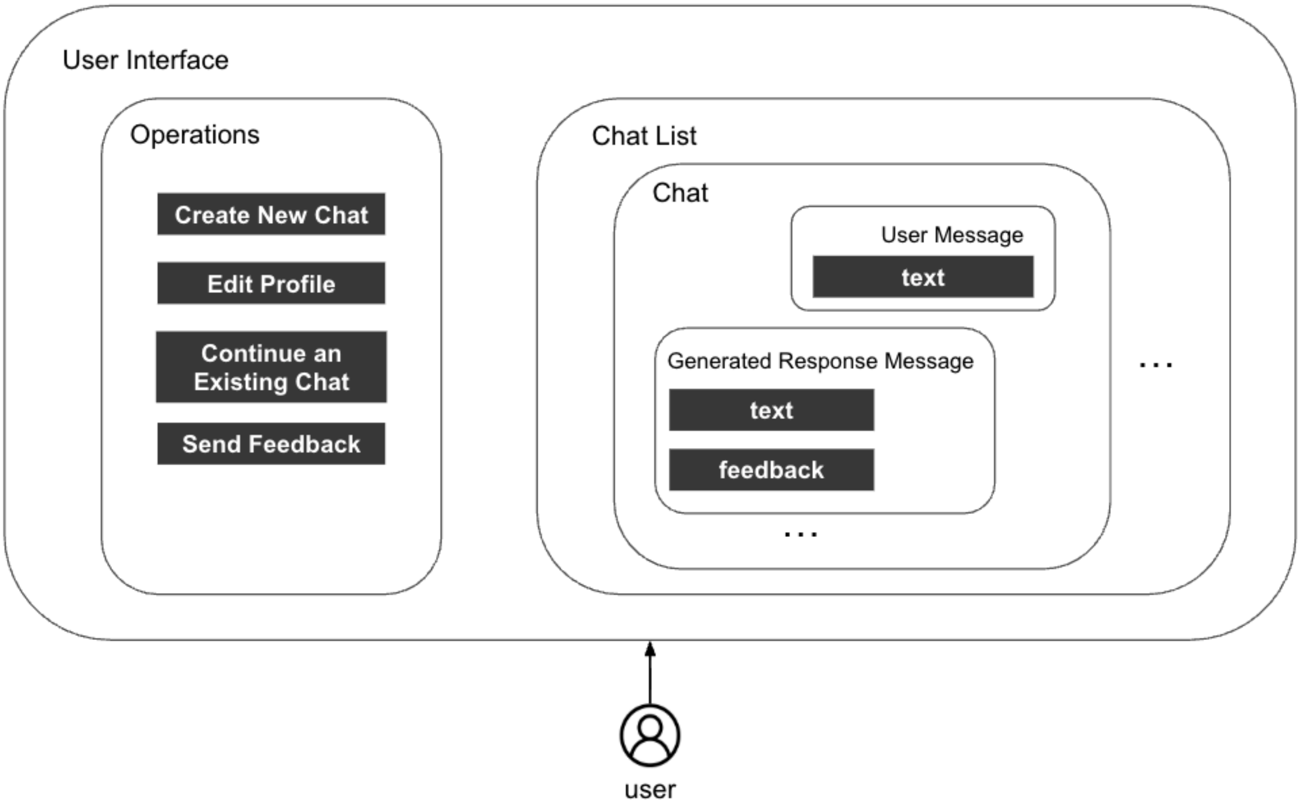

Figure 6 – External representation diagram

6.2. Process Components

The SmartChat system is composed of several process-oriented modules, each responsible for a distinct stage in the handling of user interactions:

1. Intent Recognition
Incoming queries are analyzed through Dialogflow to determine whether they belong to the software engineering domain. If the intent cannot be confidently classified, a fallback mechanism is triggered, forwarding the query to ChatGPT for intent recognition.

2. Prompt Processing
Once intent is established, the system generates an enriched and personalized prompt. This step leverages the user's stored profile, including preferences and historical insights extracted from prior feedback, to adapt the prompt for maximum relevance.

3. Cross-Platform Integration
The processed prompt is dispatched to external AI platforms–ChatGPT, Claude, and Gemini–via their respective APIs. Their candidate responses are collected asynchronously and passed to the evaluation pipeline.

4. Answer Evaluation and Merging
Each candidate response is scored based on relevance, accuracy, and clarity. The Supervisor component selects the most suitable response as the primary output, while complementary insights from other platforms are merged into the final answer presented to the user.

5. Feedback Analysis
After receiving the system's response, the user may submit structured feedback (rating, text, predefined questions). This feedback is processed through an NLP-based analysis pipeline that extracts insights and updates the user's profile for use in subsequent interactions.

6. User and Chat Management
Core processes also include handling new user registrations, authentication, conversation creation, continuation of existing conversations, and management of user preferences. These operations ensure continuity of personalized interactions across sessions.

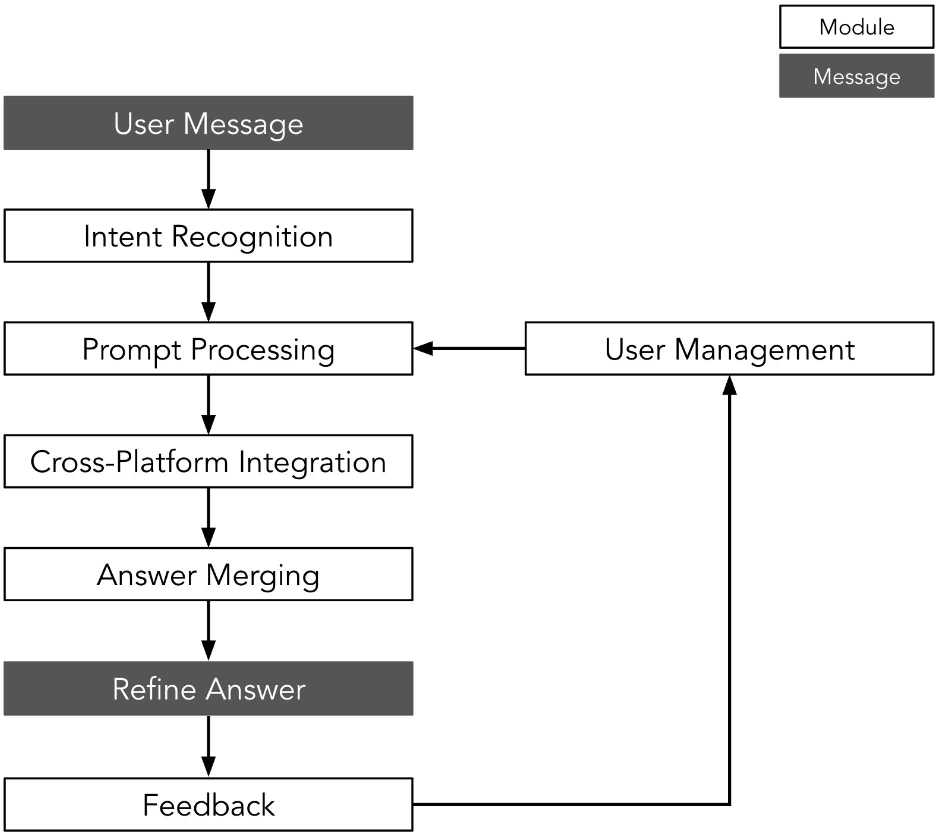

Figure 7 – Diagram representing the flow among process components

6.3. Communication Components

SmartChat relies on a layered communication model to ensure interaction between clients, backend processes, and external services:

1. Client-Server Communication
The frontend (React) communicates with the backend (Node.js/Express) via:

- **Socket.IO**: Event-driven messaging, enabling chat submission, live updates, and immediate feedback responses.

- **HTTP/REST**: Used for operations such as authentication, profile management, and retrieving historical chat data.

2. Backend-Database Communication
All stateful data–user profiles, conversations, AI responses, and feedback–is persisted in MongoDB. The backend accesses and updates these collections through repository patterns, ensuring consistency and separation of concerns.

3. Backend-External Services Communication
The backend integrates multiple external systems:

- **Dialogflow API** for intent recognition and domain classification.

- **OpenAI, Anthropic, and Google AI APIs** (ChatGPT, Claude, Gemini) for multi-platform response generation.
  Requests to these services are made over secure HTTPS, with appropriate authentication tokens, and results are received asynchronously.

4. Internal Component Communication
Backend modules communicate through direct function calls and shared data structures.

5. Response Delivery
Processed and merged answers are returned to the frontend through Socket.IO events. The user interface then renders the final response in the conversation window, along with feedback options.

6.4. Interactions

This section will present different diagrams in order to describe the SmartChat system design.

6.4.1. Activity diagram

The following activity diagram provides a visual representation of how SmartChat manages the flow of user interactions. It shows the sequence of actions beginning with user login, followed by sending a message, and branching into two possible paths: creating a new conversation or continuing an existing one. Each path highlights the distinct processing components involved, including intent recognition, prompt processing, cross-platform integration, or direct communication with a previously selected platform. The diagram also illustrates how responses are delivered back to the user and how user feedback is captured, ensuring that the system continuously improves personalization and accuracy over time.

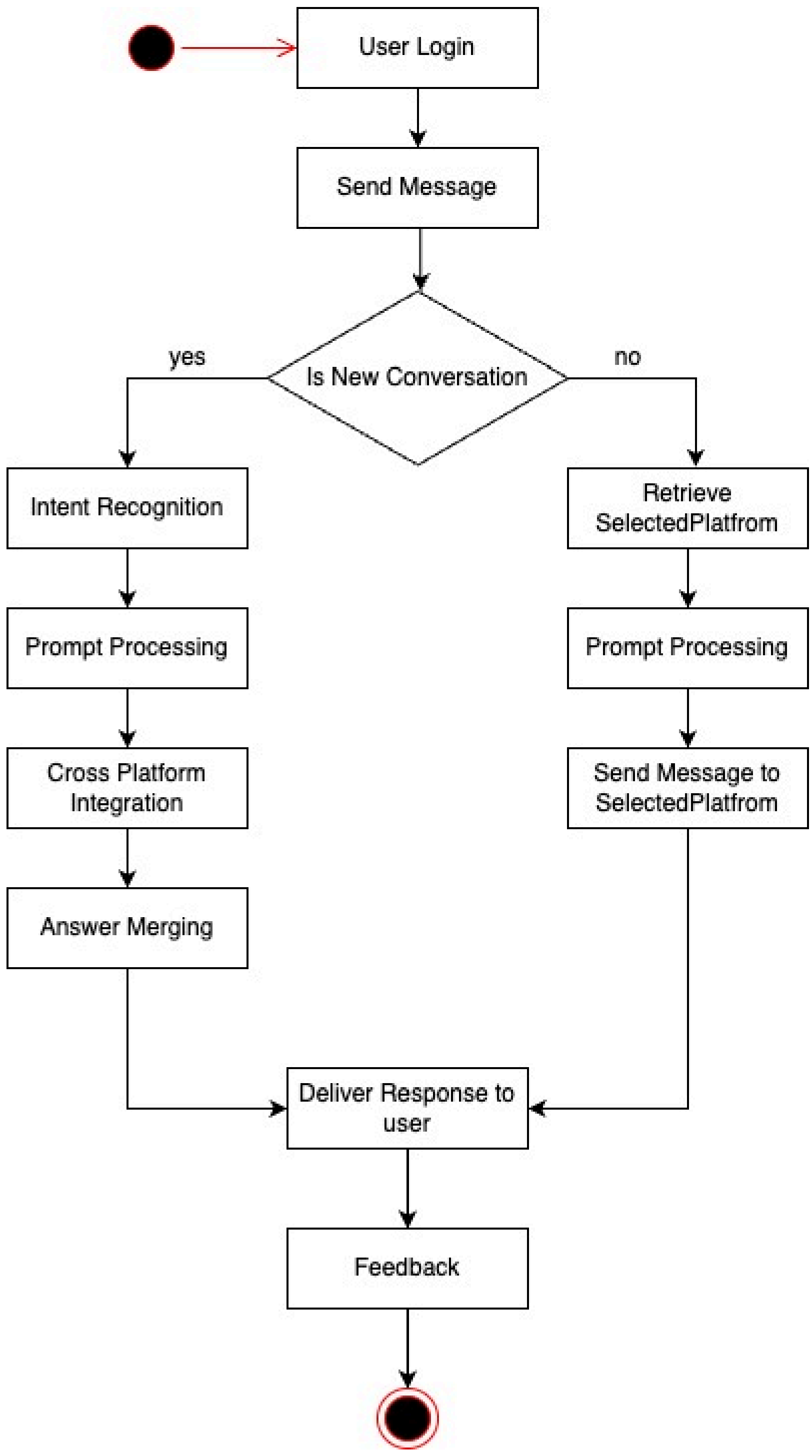

Figure 8 – Activity Diagram

This diagram describes the sequential flow of actions in the SmartChat system, beginning from user login and ending with the submission of feedback. Each activity represents a key stage in the interaction pipeline, ensuring smooth processing of user queries and intelligent integration with multiple AI platforms.

1. User Login
The process begins when the user logs into the system. Authentication ensures that user-specific data, such as preferences and conversation history, can be retrieved for personalization.

2. Send Message
Once logged in, the user initiates an interaction by sending a message through the chat interface. This triggers the system's processing flow.

3. Is New Conversation
At this stage, the system checks whether the incoming message belongs to a new conversation or is a continuation of an existing one:

- **New Conversation (Yes Path):** The system proceeds to intent recognition to determine the domain and purpose of the user's query.

- **Existing Conversation (No Path):** The system retrieves the previously selected platform from past interactions for consistent and efficient responses.

4. Intent Recognition (New Conversation Path)
For new conversations, the system applies natural language processing (NLP) techniques, supported by tools such as Dialogflow, to recognize the user's intent. If the intent cannot be clearly determined, the system defaults to ChatGPT for general-purpose interpretation.

5. Prompt Processing
The raw user input is enriched by integrating context, user preferences, and historical feedback. This ensures that the final prompt sent to AI platforms is personalized and contextually accurate.

6. Cross-Platform Integration / Selected Platform Integration

- **Cross-Platform Integration (New Conversation):** The enriched prompt is dispatched to multiple AI platforms (ChatGPT, Claude, Gemini).

- **Selected Platform Integration (Existing Conversation):** For ongoing conversations, the system sends the prompt exclusively to the previously chosen platform for continuity.

7. Answer Merging
In cases of cross-platform interaction, SmartChat evaluates and merges responses from multiple AI platforms. The evaluation process selects the relevant and accurate parts of each response, producing a refined final answer.

8. Deliver Response to User

The system returns the generated answer to the user via the chat interface. This answer is both contextual and aligned with user preferences and prior interactions.

9. Feedback Submission

Finally, the user is invited to provide feedback on the response. Feedback can include a rating (`1-10`), free-text comments, and structured answers to predefined questions. This data is analyzed and stored for future use, ensuring continuous improvement of system accuracy and personalization.

6.4.2. Sequence diagram

The sequence diagram presented below illustrates the dynamic interactions between the key components of the SmartChat system. The sequence diagram emphasizes the chronological order of operations and the exchange of messages between users, the frontend, backend, APIs, and the database. It highlights how the system handles both new and existing conversations, integrates user feedback, and ensures that responses are continually refined and personalized.

By visualizing these interactions step by step, the diagram provides a clear understanding of how SmartChat orchestrates its internal processes and communicates with external AI platforms to deliver reliable, adaptive, and user-centered responses.

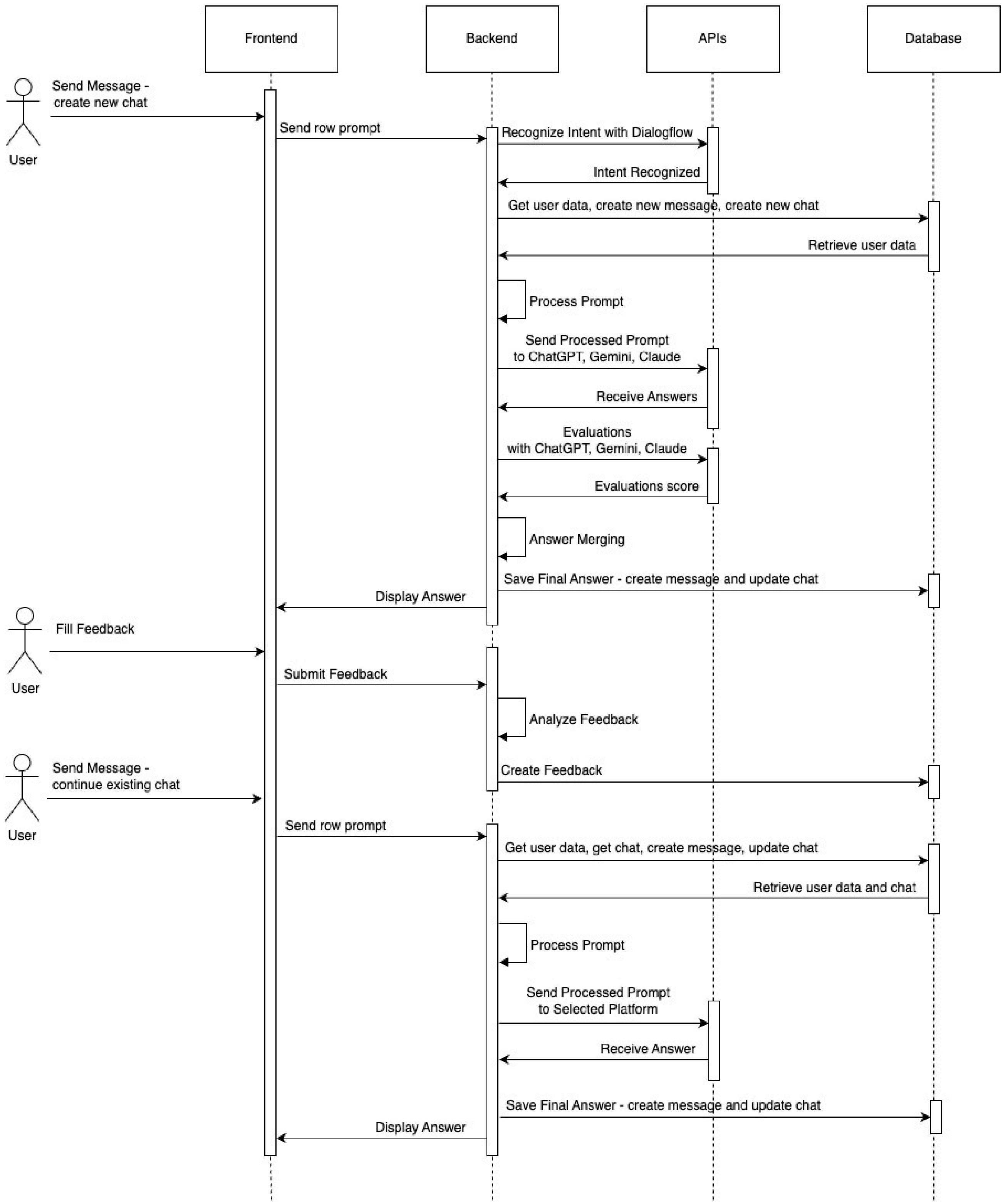

Figure 9 – Sequence Diagram

6.4.3. System Flow

The following diagram illustrates the overall **system flow of SmartChat**, capturing how different components of the platform interact with each other to deliver personalized, adaptive responses to users. It highlights the journey of a user request from the frontend through backend processes, integration with external AI platforms, and storage in the database, as well as the essential role of user feedback and profile updates in refining the system over time.

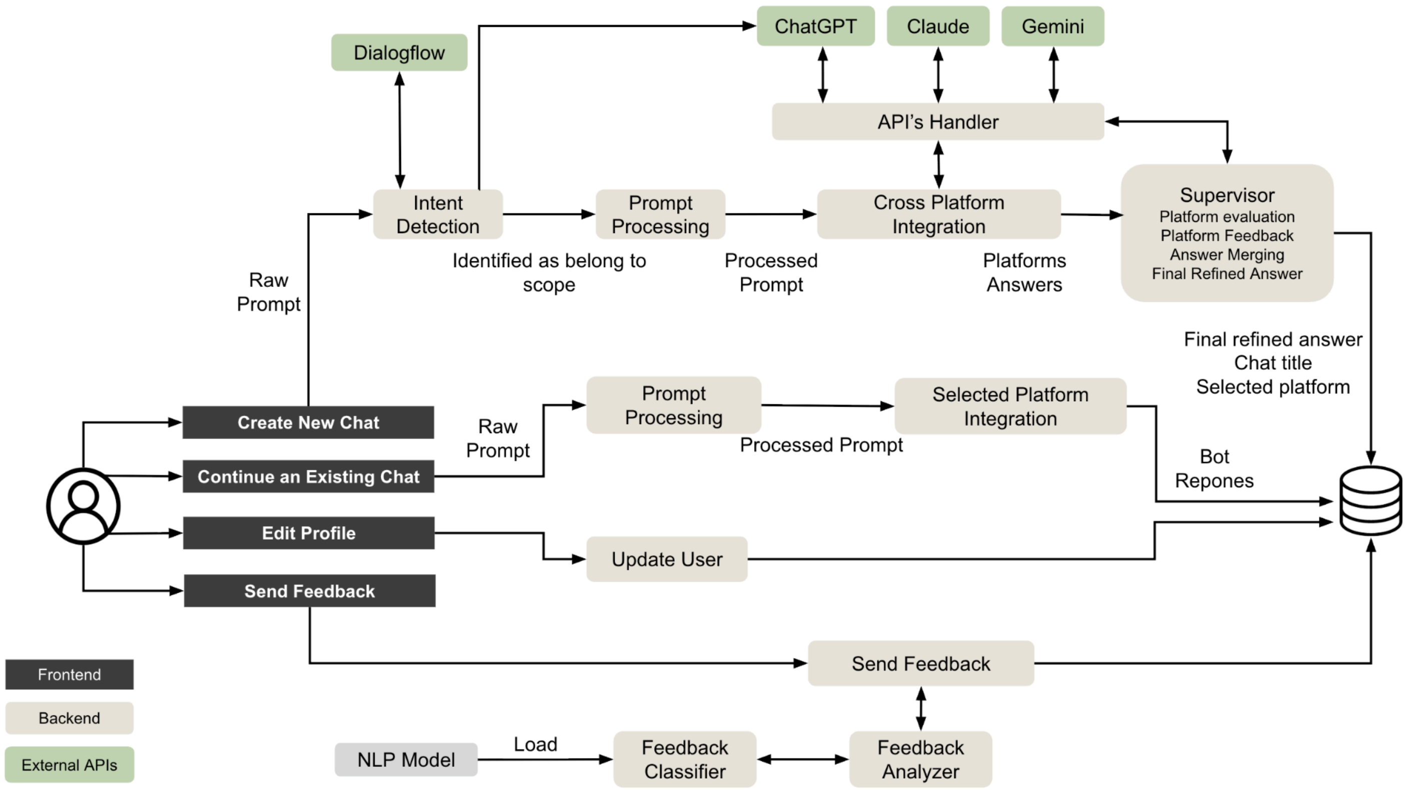

Figure 10 – System Flow Diagram

The diagram shows how SmartChat handles the complete cycle of interaction between the **user, backend system, external AI platforms, and database**.

1. **User Interaction (Frontend)**
    - The flow begins when the user creates a new chat, continues an existing conversation, edits their profile, or submits feedback.
    - These actions generate raw prompts, updates, or feedback data, which are passed to the backend.

2. **Intent Detection and Prompt Processing (Backend)**

    ○ Raw prompts are analyzed by **Dialogflow** for **Intent Detection**, ensuring they fall within the system's scope.

    ○ If valid, the input is refined in **Prompt Processing**, preparing it for platform evaluation.

3. **Cross-Platform and Selected Platform Integration**

    ○ For multi-platform use, the processed prompt is sent through **Cross Platform Integration**, which connects to external APIs (**ChatGPT, Claude, Gemini**) via the API Handler.

    ○ Their responses are collected and passed to the **Supervisor**.

    ○ For ongoing chats, the system can instead use **Selected Platform Integration** to communicate directly with a chosen platform.

4. **Supervisor and Evaluation**

    ○ The **Supervisor** ensures high-quality results by evaluating platform responses, merging answers, and applying feedback.

    ○ It produces the **final refined answer**, which includes chat titles and selected platform details.

5. **Database Persistence**

    ○ All data – including chat history, answers, and user updates – is stored in the **Database**, enabling chat continuity and personalized experiences.

6. **Feedback and Adaptation**

    ○ Users provide feedback on responses, which is analyzed by the **Feedback Analyzer** and classified using the **NLP Model**.

    ○ The results influence future evaluations and platform selection, creating a loop of **continuous system improvement**.

7. IMPLEMENTATION

The implementation of *SmartChat* was carried out as a full-stack development project, combining a Node.js (Express) backend, a React-based frontend, and a MongoDB database. The system was designed to operate as a modular pipeline that supports conversational interaction, integrates multiple external AI platforms, and adapts responses to user-specific contexts. Each component of the implementation was carefully engineered to ensure reliability, scalability, and usability, while adhering to the defined domain of software engineering. The following subsections describe in detail the algorithmic design, system interfaces, graphical user interface, configuration, and supporting development practices that enabled the construction of the final system.

7.1. Algorithm Design

The SmartChat system was designed as a modular pipeline, where each stage is responsible for a specific aspect of processing the user's query and generating a refined, domain-specific response. The workflow is divided into six main modules:

1. **Chat Interface** - The process begins with the user interacting through a dedicated chat interface implemented in React. The frontend provides a conversational environment similar to widely used AI chat applications, allowing users to send messages and receive responses. Communication with the backend is managed via an open WebSocket channel, enabling bidirectional and low-latency data transfer between client and server.
   In addition to basic message exchange, the interface allows users to configure **response parameters** that directly influence the system's output. These parameters include the **tone of the response** (e.g., formal or informal), the **number of examples** to be included in the answer, and the **level of detail** desired. By exposing these controls at the interface level, the system empowers users to customize the conversational style and depth of explanations according to their preferences and needs.

2. **Intent Recognition** - Upon receiving a new prompt from the user, the system performs an intent recognition step to determine whether the query falls within the domain of software engineering. This module was implemented using **Dialogflow**, a Google Cloud service that enables natural language classification based on predefined intents and training data. Dialogflow analyzes the input and returns an intent label, which the system uses to decide whether to proceed with further processing.

Since Dialogflow requires sufficient training examples to achieve high accuracy, a **fallback mechanism** was also implemented. In cases where Dialogflow cannot reliably classify the intent, the system applies a prompt-engineering approach that reformulates the query and sends it to **ChatGPT** for classification. ChatGPT is asked to explicitly return whether the query belongs to the software engineering domain. This dual-layer approach ensures robust intent recognition while balancing the need for accuracy with system flexibility.

3. **User Profile Management & Prompt Processing** - Once a basic prompt is classified as relevant, the system proceeds to prompt processing, which constitutes the core of *SmartChat*'s personalization capabilities. In this stage, the raw basic prompt is inserted into a **structured template** that guides the generation of contextually accurate responses.

   The template incorporates data from the **user profile**, including:

   - Past interactions and conversation history.
   - Explicit preferences such as **desired tone, level of detail, and number of examples**.
   - Historical feedback provided on previous responses.

   These parameters, controlled by the user through the chat interface, directly influence the behavior of the prompt processing module. By combining static templates with dynamic user-specific data, the system ensures that **basic prompts submitted to the AI platforms are both contextually relevant and personalized**.

   This approach allows SmartChat to produce responses that reflect the user's preferences while maintaining domain-specific accuracy. The processed prompt is then forwarded to the **cross-platform integration module** for further handling.

4. **Cross-Platform Integration** - Once the basic prompt has been processed with user-specific context, it is dispatched to three AI platforms–**OpenAI ChatGPT, Anthropic Claude, and Google Gemini**–which operate in **parallel**. Each platform runs an independent session, enabling the system to collect responses simultaneously and reduce overall latency.

   To enhance the quality of each output, SmartChat performs **iterative prompt processing** within each session. After receiving an initial response from an AI platform, the system constructs a follow-up prompt instructing the AI to **"improve or refine the previous answer"**. This cycle is repeated **three to five times per session**, generating a sequence of increasingly polished responses from each platform.

The integration module handles all external communication, including:

- Sending the basic prompt and iterative improvement instructions.
- Receiving and normalizing responses across platforms.
- Managing errors, timeouts, and retries for robust operation.

By running these **parallel, iterative sessions**, SmartChat ensures that the final set of candidate responses is both **high-quality and diverse**, ready for comparative evaluation in the subsequent module.

5. **Answer Evaluation and Merging** - Once all iterative responses from the three AI platforms have been collected, SmartChat evaluates the answers to determine the most suitable response for the user.

   The **evaluation process** works as follows: each candidate answer is sent to the **two other AI platforms** with a specially engineered prompt that instructs them to evaluate the answer based on predefined parameters such as **relevance, technical accuracy, clarity, and contextual alignment**. Each evaluating AI provides a **score from 1 to 10**, reflecting the quality of the answer.

   After scoring, the system selects the answer with the **highest evaluation score** as the primary response. The two remaining answers are then **merged as supplementary information**, appended to the chosen answer to provide additional insights or alternative perspectives without overriding the primary response.

   Finally, a **post-processing stage** is applied to ensure coherence, refine phrasing, format code snippets if present, and remove redundancies. The resulting response is comprehensive, contextually relevant, and aligned with the user's preferences.

   This method allows SmartChat to **leverage cross-platform evaluation** to objectively rank answers while still incorporating diverse perspectives from all three AI platforms.

6. **Feedback Collection and Analysis** - After delivering the finalized response to the user, SmartChat provides an option for **explicit feedback**. Users can provide feedback by rating the answer, leaving free-text comments, and responding to built-in multiple-choice questions.
   These inputs are captured through the chat interface and transmitted to the backend.

The collected feedback is stored in the **user profile database**, where it is linked to the corresponding basic prompt, chosen answer, and any supplementary information provided. This historical feedback enables the system to **adapt and improve over time**, informing prompt processing and enhancing personalization.

During subsequent interactions, the feedback is used to:

- Adjust templates and parameters in prompt processing to better match user preferences,
- Enhance personalization by considering prior ratings and comments in generating new responses.

By systematically integrating user feedback into the system's processing pipeline, SmartChat achieves **continuous performance improvement and higher user satisfaction**, while maintaining domain-specific accuracy and contextual relevance.

7.2. Interfaces

SmartChat's interface design emphasizes simplicity, usability, and efficiency, ensuring that users can interact naturally with the system while also benefiting from advanced personalization features.

The Chat Interface serves as the core point of interaction, designed to mimic the familiar look and feel of modern conversational platforms such as ChatGPT. Users can initiate new conversations, send prompts, and receive AI-generated responses. The interface supports enhanced response formatting, including code blocks and markdown rendering.

The User Preferences Panel provides users with granular control over how responses are generated. Within this interface, users can adjust parameters such as the tone of the response (formal or casual), the desired number of examples, and the level of detail included in answers. These configurations are stored as part of the user profile and directly inform the prompt processing process, enabling a more tailored conversational experience.

From a system perspective, SmartChat exposes well-defined interfaces for communication between its components. The frontend communicates with the backend through REST APIs for general operations (e.g., authentication, session management) and WebSockets for chat interactions. The backend, in turn, integrates with external AI platforms (ChatGPT, Claude, Gemini) and Dialogflow for intent recognition. MongoDB Atlas serves as the database interface, managing user profiles, preferences, and conversation history.

Together, these interfaces establish a seamless flow of information from user input to AI-powered response generation, while maintaining a user-friendly and configurable interaction experience.

7.2.1 Human-Computer Interfaces (GUI)

The graphical user interface (GUI) of SmartChat is designed to provide a responsive, user-friendly, and interactive chat experience, similar to contemporary AI chat applications. The frontend, implemented in React with JSX, serves as the primary point of interaction between the user and the system.

Key features of the GUI include:

- **Messaging** through an open WebSocket connection, allowing messages to be sent and received instantaneously (figure 11).
- **Customizable response parameters,** where users can set the tone of the answer, the number of examples, and the level of detail for the responses. These parameters directly influence the AI's output and enhance personalization (figure 12).
- **Enhanced message rendering,** using the react-markdown and remark-gfm packages to display Markdown and GitHub-flavored code snippets. This allows responses, especially those containing code, to be formatted in a ChatGPT-like style, improving readability and usability for software engineering content (figure 11).
- **Feedback controls,** enabling users to provide ratings, comments, or a binary good/bad signal for each response (figure 13).
- **Conversation history,** which displays previous messages in a structured format to maintain context and support follow-up questions (figure 11).
- **Error and system notifications,** informing users when an AI platform is unavailable or when a prompt falls outside the software engineering domain.

The design prioritizes clarity and simplicity, minimizing cognitive load while maintaining the flexibility to adjust output according to user preferences. Visual elements such as message bubbles, color coding for AI vs. user messages, and highlighted code snippets contribute to a familiar and intuitive experience.

Overall, the GUI bridges the user and the complex backend logic, ensuring that SmartChat's advanced multi-AI processing remains accessible and transparent to the user.

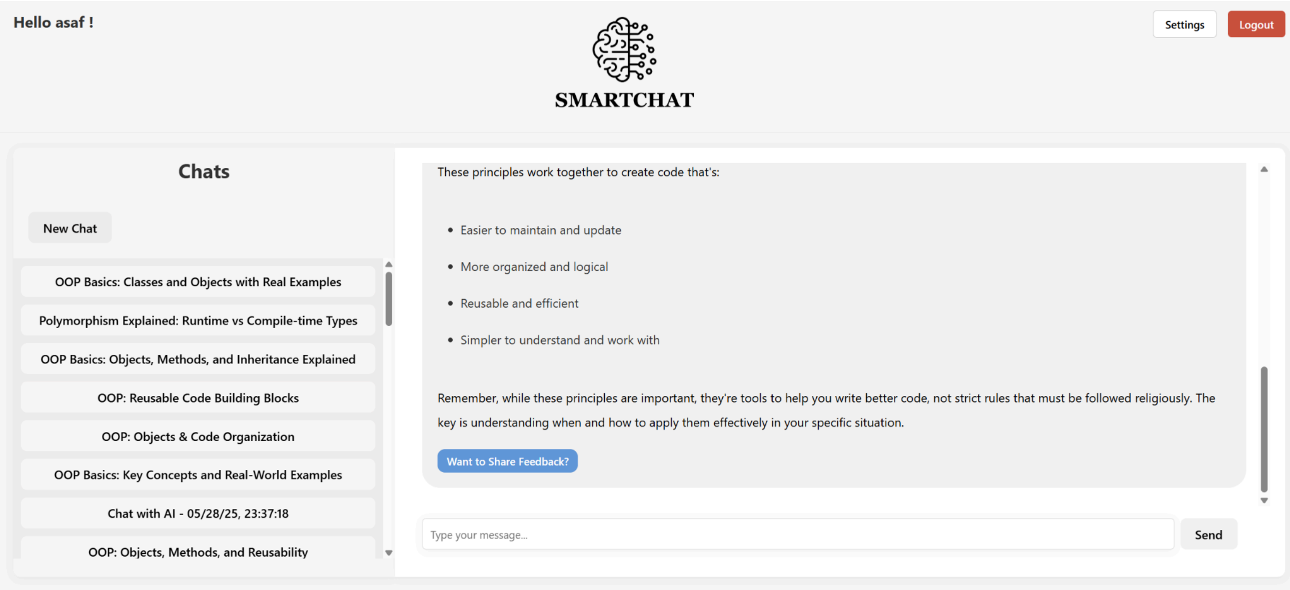

Figure 11 - Chat page view

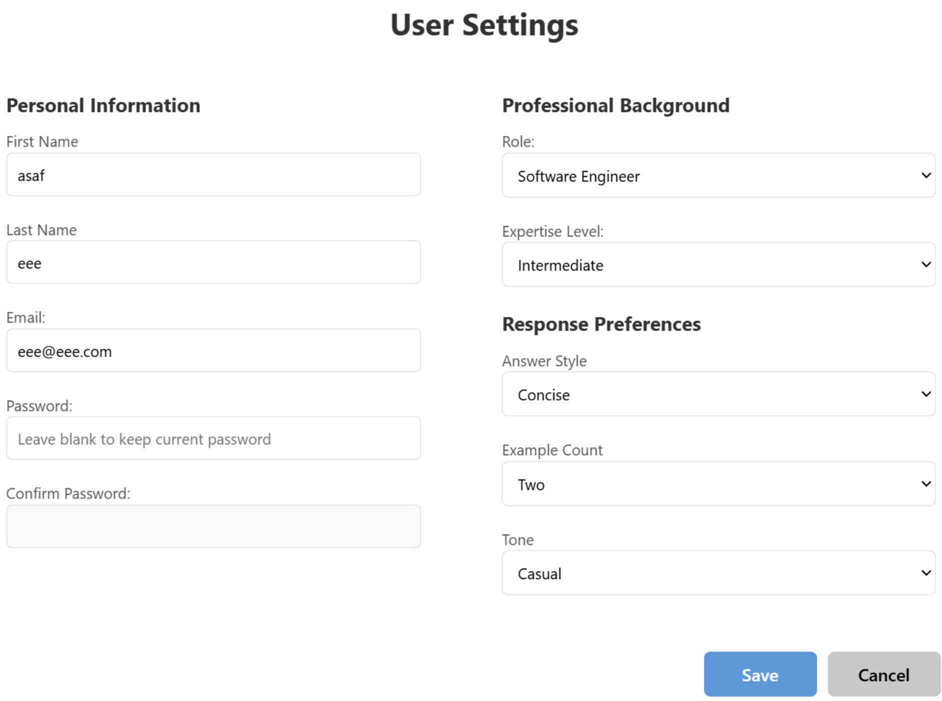

Figure 12 - User setting and preferences view

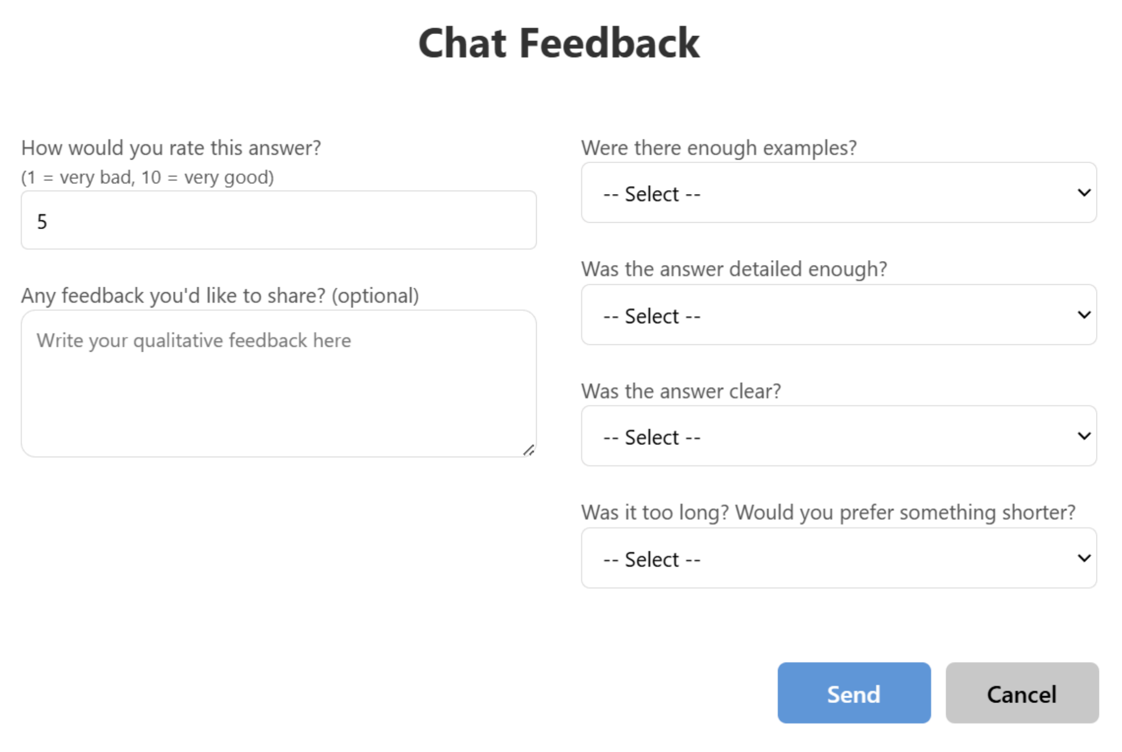

Figure 13 - Feedback form view

7.3. System Configuration

SmartChat is deployed as a full-stack application with clearly defined backend, frontend, and database components, optimized for conversational interaction and cross-platform AI integration. The system configuration is structured as follows:

Backend (Node.js / Express):

- Hosts the core APIs for chat handling, user management, and AI integration.
- Manages communication with the frontend via WebSocket connections.
- Implements modules for intent recognition, prompt processing, cross-platform integration, answer evaluation, and merging.
- Handles external API calls to AI platforms (ChatGPT, Claude, Gemini) and manages iterative prompt processing.

- Deployed and hosted using Render, ensuring continuous availability and simplified cloud deployment.

Frontend (React / JSX):

- Provides the chat interface, parameter controls, feedback submission, and conversation history.

- Utilizes packages such as react-markdown and remark-gfm to render Markdown and GitHub-flavored code snippets for enhanced readability.

- Connects to the backend via WebSockets for messaging and via REST API calls for general page data and user profile updates.

- Also hosted on Render, providing a reliable and scalable platform for the client interface.

Database (MongoDB Atlas):

- Stores user profiles, including preferences, interaction history, and feedback records.

- Maintains logs of conversations, AI responses, and evaluation scores to support personalization and system improvement.

- Hosted externally on MongoDB Atlas, providing cloud-based scalability, security, and high availability.

Communication Configuration:

- The backend and frontend communicate via WebSocket channels for chat messages and RESTful APIs for other interactions.

- External AI integrations are performed through secure HTTP requests with proper error handling and retries.

This system configuration ensures scalability, maintainability, low-latency response delivery, and reliable cloud hosting, while allowing SmartChat to integrate multiple AI platforms and maintain a responsive, user-centered interface.

7.4. Development Environment

The development environment for SmartChat was carefully structured to support collaborative work and maintain high code quality. The team primarily used **Visual Studio Code** as the integrated development environment, leveraging its extensive plugin ecosystem to enhance productivity across JavaScript, JSX, and related technologies.

Version Control and Collaboration:

- Git was employed for version control, with repositories hosted on GitHub to manage code storage and collaboration.

- A structured branching strategy was implemented, including feature branches, a development branch, and a main branch, ensuring organized code integration and deployment.

Project Management:

- Jira was used for task tracking and project management, allowing the team to link user stories, tasks, and code commits. This integration facilitated automated updates of task statuses based on code changes and streamlined the development workflow.

- Regular code reviews were conducted via GitHub pull requests, promoting knowledge sharing and maintaining code quality standards across the team.

Cloud Development and Hosting:

- Both the frontend and backend were built and hosted using Render, providing a scalable and reliable cloud environment.

- MongoDB Atlas served as the external database, offering high availability, security, and cloud-based scalability.

This development environment provided a flexible, collaborative, and scalable setup, allowing the team to efficiently implement, test, and deploy the SmartChat system while maintaining high standards of quality and reproducibility.

7.5. Programming Languages

The SmartChat project leveraged a focused set of programming languages to address its frontend, backend, and AI integration requirements.

Frontend Development:

- JavaScript was extensively used in conjunction with the React framework.
- JSX enabled a component-based architecture, allowing for modular, reusable, and maintainable code.
- Additional packages, such as react-markdown and remark-gfm, were employed to render Markdown and GitHub-flavored code snippets, providing a ChatGPT-like presentation of technical content.

Backend Development:

- Node.js served as the primary language for backend services due to its asynchronous, event-driven architecture, which is well-suited for chat applications.
- Express was used to streamline API development, routing, and middleware management, supporting integration with multiple AI platforms and WebSocket communication for instant messaging.

Database and Data Management:

- The system utilizes MongoDB (hosted on Atlas), a NoSQL document-oriented database, for storing user profiles, chat history, and feedback data.
- Its flexible schema allows efficient handling of unstructured data such as AI-generated responses, iterative outputs, and user interactions.

This combination of programming languages and database technologies allows SmartChat to balance communication, personalization, and multi-AI integration, resulting in a scalable, responsive, and robust platform for software engineering-focused conversational AI.

7.6. Limitations of the System

Despite its robust design, SmartChat faces several limitations that are important to acknowledge.

Reliance on External AI APIs:
The system depends heavily on external AI platforms–ChatGPT, Claude, and Gemini–for generating responses. This dependency introduces potential latency issues, API rate limits, and service interruptions, which can impact the delivery of chat responses, especially under high user loads.

Latency from Iterative Processing:
The iterative prompt processing process, where outputs are repeatedly refined and evaluated across multiple AI platforms, can introduce additional response delays. While this enhances answer quality, it may slightly affect the immediacy expected in conversations.

Intent Recognition Limitations:
Although SmartChat uses DialogFlow and fallback classification via ChatGPT, the system may occasionally misclassify basic prompts that are ambiguous or outside the software engineering domain. This can result in inaccurate determinations of whether a prompt should be processed or rejected.

Scope of Personalization:
User-specific parameters such as tone, level of detail, and number of examples provide some customization, but the system's personalization is limited by the capabilities of the underlying AI platforms. The AI may not fully adapt to all individual user preferences or context nuances.

Dependency on External Services:
The use of Render for hosting and MongoDB Atlas for database storage introduces external points of failure. Any disruptions, maintenance, or policy changes in these services could impact system availability or performance.

Overall, while SmartChat delivers advanced multi-AI conversational capabilities, these limitations highlight areas where performance, personalization, and reliability may be constrained by external factors and system design choices.

7.7. Installation and Configuration Issues

The installation and configuration of SmartChat involved several challenges that required careful attention to ensure smooth deployment and reliable operation.

Dependency and Version Management:

- Managing dependencies between frontend and backend services was critical. npm was used to handle JavaScript and React packages, including react-markdown and remark-gfm. Strict version control was applied via package.json to maintain consistency across development and deployment environments.

Secure Access and API Keys:

- Configuring secure access to external AI platforms (ChatGPT, Claude, Gemini) required proper handling of API keys. Environment variables were extensively used to keep sensitive information separate from the codebase, ensuring security and flexibility across multiple deployment environments.

- Connection to MongoDB Atlas also required secure authentication and network configuration, including IP whitelisting and properly formatted connection URIs.

Network and Communication:

- Establishing WebSocket connections for messaging necessitated careful planning of ports, firewall rules, and backend/frontend alignment. Misconfigurations could prevent instant message delivery or disrupt parameter controls and feedback submission.

Cloud Hosting Configuration:

- Hosting both frontend and backend on Render required proper build scripts, environment setup, and domain configuration. Continuous deployment pipelines had to be carefully managed to ensure that updates to either component did not break functionality.

Potential Pitfalls:

- Misalignment between frontend and backend versions could lead to runtime errors.
- Improper handling of parallel API calls and iterative prompt processing could introduce latency or failed responses.

By addressing these installation and configuration challenges systematically, the SmartChat system achieves secure, reliable, and maintainable deployment, ensuring consistent performance for end users.

7.8. Information Security

SmartChat implements multiple measures to ensure the security of user data, system components, and interactions with external AI platforms.

User Data Protection:

- All sensitive user information, including profiles, preferences, and feedback, is securely stored in MongoDB Atlas, which provides encryption at rest and in transit.
- Access to user data is controlled through role-based permissions in the backend, ensuring that only authorized processes and personnel can access sensitive information.

API Key and Credential Management:

- API keys for external AI platforms (ChatGPT, Claude, Gemini) are stored in environment variables, preventing hardcoding of credentials in the codebase.
- Access to these environment variables is restricted to authorized backend services, reducing the risk of exposure.

Secure Communication:

- All communication between frontend and backend is conducted over HTTPS to prevent interception or tampering of data.

- WebSocket connections for chat messages are secured using WSS (WebSocket Secure), ensuring encrypted transmission of messages.

Database Security:

- MongoDB Atlas provides network isolation, IP whitelisting, and authentication mechanisms to protect against unauthorized access.
- Regular backups and monitoring are enabled to safeguard against data loss and to maintain system integrity.

External Service Security:

- Interactions with external AI platforms are performed over secure channels, with proper error handling to prevent leakage of sensitive information.
- Hosting on Render ensures that backend and frontend services benefit from built-in cloud security measures, including automatic updates, secure deployment pipelines, and DDoS protection.

User Privacy and Compliance:

- SmartChat minimizes data collection to only what is necessary for system functionality and personalization.
- Feedback and interaction logs are anonymized when used for system improvement to protect user privacy.

By implementing these security measures, SmartChat ensures confidentiality, integrity, and availability of both user data and system resources, maintaining trust and compliance with best practices for cloud-based applications.

7.9. Risk Management

SmartChat incorporates a structured approach to identify, assess, and mitigate potential risks that could affect system performance, reliability, and security.

External API Failures:

- SmartChat depends on external AI platforms (ChatGPT, Claude, Gemini) for generating responses. To mitigate the risk of service outages or rate limits, the system implements fallback mechanisms, including alternative API calls and retry strategies.

- Responses from AI platforms are processed in parallel, reducing the impact of a single platform failure on overall system performance.

Data Loss or Corruption:

- User data, chat history, and feedback are stored in MongoDB Atlas, which provides automated backups and high-availability clusters.

- The system regularly validates database operations to prevent data corruption, and transaction mechanisms are used where applicable to ensure consistency.

Security Risks:

- API keys and sensitive user data are stored in environment variables and encrypted where appropriate.

- Access controls, HTTPS/WSS communication, and secure hosting on Render reduce the risk of unauthorized access, man-in-the-middle attacks, or data breaches.

System Scalability and Performance:

- The iterative prompt processing and multi-platform integration could lead to latency under high load. SmartChat mitigates this through parallel execution, connection pooling, and optimized query handling.

- Load testing and monitoring are performed to anticipate bottlenecks and scale services as needed.

Operational Risks:

- Misconfigurations during deployment or updates could disrupt system availability. To minimize this, continuous integration and deployment pipelines are used, and staging environments replicate production settings for testing changes.

- Team coordination and task management through Jira ensures clear assignment of responsibilities and timely response to issues.

By proactively identifying and addressing these risks, SmartChat maintains reliable performance, data integrity, and secure operations, even in the presence of external dependencies and high user activity.

7.10. Exceptions Management

SmartChat implements a comprehensive exceptions management strategy to handle errors, unexpected inputs, and failures gracefully, ensuring a smooth user experience and maintaining system stability.

Prompt Validation and Classification Errors:

- When a user submits a basic prompt, it is first validated for format and relevance.

- If the prompt is outside the software engineering domain or ambiguous, the intent recognition module triggers a fallback response informing the user that the topic cannot be processed.

API and External Service Failures:

- Failures from AI platforms (ChatGPT, Claude, Gemini) are detected and logged. The system implements retry mechanisms and alternative API routes to minimize disruption.

- Partial failures, where one or two platforms fail, do not prevent the user from receiving a response, as SmartChat continues processing available outputs.

Backend and Database Errors:

- Errors in database operations, such as failed insertions or retrievals from MongoDB Atlas, are captured with proper try-catch blocks.

- Users are informed of temporary system issues without exposing technical details, and all exceptions are logged for further analysis.

Frontend Error Handling:

- The React interface handles network or WebSocket failures by displaying clear error messages and providing options to retry actions.
- Rendering issues, such as invalid Markdown or code snippets, are managed to prevent UI crashes, ensuring that messages display correctly even in error scenarios.

Logging and Monitoring:

- All exceptions, including prompt validation errors, API failures, and system faults, are logged with timestamps and context for troubleshooting.
- Monitoring tools alert developers to recurring issues or critical failures, enabling timely corrective actions.

By systematically managing exceptions, SmartChat ensures robustness, reliability, and user-friendly error handling, even in the presence of unexpected events or external service disruptions.

7.11. Version Control

SmartChat employs a structured version control strategy to maintain code quality, facilitate collaboration, and ensure smooth development and deployment processes.

Git and GitHub:

- Git is used as the primary version control system, with repositories hosted on GitHub to provide a centralized platform for code storage, collaboration, and history tracking.
- All team members work on feature branches, which are merged into a development branch for integration testing in the staging environment before deployment.

Branching Strategy and Environments:

- Feature branches allow parallel development without affecting the main codebase.

- The development branch is deployed to the staging environment, where new features are thoroughly tested and validated.

- The main branch always contains the fully tested and production-ready version of the system. This ensures that the production environment remains a stable, fully functional version, reflecting the perfect work state of SmartChat at all times.

Collaboration and Code Reviews:

- GitHub pull requests are used to merge feature branches into the development branch, providing opportunities for code review, knowledge sharing, and quality assurance.

- Automated CI/CD pipelines can be integrated with pull requests to run tests and validate code before merging, reducing the likelihood of introducing errors.

Version History and Rollbacks:

- Git maintains a complete history of commits, enabling the team to track changes, identify regressions, and roll back to previous stable versions if necessary.

- Tags and release notes document major releases and updates, supporting clear version tracking and deployment management.

By implementing this structured version control approach, with staging and production environments, SmartChat ensures organized development, safe deployment, and a stable production system that always reflects the perfectly tested work version.

7.12. Project Management

SmartChat's development process was organized using structured project management practices to ensure timely delivery, clear task allocation, and coordinated collaboration among team members.

Task Tracking and Planning:

- The team utilized Jira to manage user stories, tasks, and bugs. Each feature or module was broken down into clearly defined tasks, with priorities, deadlines, and assigned team members.

- Jira boards provided a visual representation of workflow, showing tasks in stages such as "To Do," "In Progress," "Code Review," "Testing," and "Done," which allowed the team to monitor progress in real time.

Integration with Version Control:

- Jira was integrated with GitHub, linking commits and pull requests to specific tasks and user stories. This provided traceability from code changes back to project requirements and ensured that all development work aligned with planned tasks.

- Automated updates of task statuses based on GitHub activity streamlined progress tracking and minimized manual overhead.

Collaboration and Coordination:

- Regular stand-up meetings were held for quick updates, and weekly meetings were conducted to review completed tasks, discuss upcoming priorities, and adjust the project plan as needed. This allowed the team to maintain focus and adapt to changing requirements effectively.

- Code reviews through GitHub pull requests promoted knowledge sharing, adherence to coding standards, and early detection of potential issues.

Workflow for Deployment:

- Development followed a clear workflow: features were implemented in feature branches, merged into the staging environment for testing, and finally deployed to production only after passing all tests and validations.

- This workflow ensured that the production environment always reflected the fully tested and approved version of the system, reducing the risk of introducing errors during deployment.

Documentation and Knowledge Management:

- Jira was used alongside documentation in the repository to maintain records of design decisions, task histories, and technical notes, supporting team collaboration and onboarding of new members.

By combining Jira for task tracking, GitHub for version control, structured workflows, and weekly review meetings, SmartChat maintained organized development, efficient collaboration, and reliable delivery of a complex multi-component AI chat system.

7.13. Code

The code for the **SmartChat** project is hosted on **GitHub**, organized into two main repositories corresponding to the **frontend** and **backend** components. This separation ensures modularity, maintainability, and independent deployment of the system's services.

SmartChat Frontend

- **Description:** This repository contains the user interface developed in **React (JSX)**. It provides the chat-like interface modeled after ChatGPT, with support for sending prompts, receiving responses, and controlling user parameters such as tone, number of examples, and level of detail.

- **Repository:** SmartChat-Frontend

SmartChat Backend

- **Description:** This repository contains the server-side code built with **Node.js** and **Express**. It serves as the system's core, providing APIs to the frontend, integrating with external AI platforms (ChatGPT, Claude, Gemini), managing socket-based chat communication, and handling user authentication and database interactions with MongoDB Atlas.

- **Repository:** SmartChat-Backend

Installation and Setup

1. Clone the repositories to your local machine:

```
git clone https://github.com/asafzaf/SmartChatFront.git
git clone https://github.com/asafzaf/SmartChat.git
```

2. For each repository, navigate to the directory and install the necessary dependencies:

```
cd SmartChat-Frontend
npm install

cd SmartChat-Backend
npm install
```

3. Create a `.env` file in the root directory of both repositories with the required environment variables.

    - Include API keys for external AI platforms (ChatGPT, Claude, Gemini).

    - Add MongoDB Atlas connection strings.

    - Configure server ports and any additional secrets.

    - Refer to this file for relevant env variables.

4. Start the backend server:

```
npm run dev
```

5. Start the frontend application:

```
npm run dev
```

6. Open your browser and navigate to:

```
http://localhost:5173
```

to access the **SmartChat** interface. From here, you can interact with the system, send prompts, receive responses, and provide feedback.

Live App

The project is deployed on **Render**, a modern cloud platform that simplifies hosting web applications. Render was chosen because it provides seamless deployment, automatic scaling, and continuous integration with our GitHub repository. This allowed us to quickly push updates and ensure that the system remained accessible and reliable throughout development and testing. By hosting both the backend and frontend on Render, we achieved a smooth deployment pipeline that made the project available to users without manual server management.

URL: https://smartchatfront.onrender.com/

8. System Validation

This section presents a comprehensive testing report for the SmartChat system–a multi-agent conversational platform that selects and adapts responses from various AI models based on user profile, message content, and feedback.

The goal of the testing process was to ensure correctness, robustness, and personalized behavior throughout the system's lifecycle: from backend logic and socket communication to adaptive decision-making and feedback analysis. Each test was designed to verify a critical component of the system, using unit testing, integration testing, NLP-based feedback evaluation, and scenario-driven validation.

8.1. Backend Validation

Overview

This document outlines the backend testing strategy implemented using the Jest framework for the server-side components of the system. The backend is built with Node.js and Express, and includes controllers, routes, and repository layers. The goal of the testing process is to ensure the correctness, robustness, and reliability of the backend logic by covering both unit and integration test cases.

Testing Framework and Tools

- **Jest**: A JavaScript testing framework used for writing and running tests.\
- **Supertest**: Used for simulating HTTP requests in integration tests.
- **Mocking Tools**: Jest's built-in mocking capabilities are used to mock external dependencies and repository functions.

Test Organization

The tests are organized into two main categories:

1. **Unit Tests**: These tests are written to validate the behavior of individual controller functions in isolation. Repository functions and other external dependencies are mocked.
2. **Integration Tests**: These tests validate the system's behavior through Express routes, testing the full request/response cycle.

Test Coverage

The following backend components are tested:

1. Authentication Controller

- signIn: Validates authentication logic including email and password matching, and token generation.
- signUp: Tests user creation and token assignment.
- protectedRoute: Middleware that verifies the presence of a valid access token and user ID before granting access to protected routes.

2. User Controller

- updateUser:Tests user update.
- getUsers: Retrieves all users from the repository.
- getUser: Retrieves a specific user by ID.
- createUser: Creates a new user in the system.

3. Chat Controller

- getChatList: Fetches the user's chat list using a protected route.
- deleteChat: Deletes a chat and all its associated messages.

4. Message Controller

- getMessages: Retrieves all messages associated with a specific user and chat.
- createMessage: Creates and stores a new message.
- markMessageAsFeedbackGiven: Updates a message to reflect that feedback has been given.

5. Feedback Controller

- saveFeedback: Validates and stores user feedback, ensuring the related chat and user exist and that feedback is not duplicated.

Mocking and Isolation

To ensure that unit tests focus only on controller logic:

- Repository modules (e.g., userRepository, chatRepository) are mocked using jest.mock(...).
- Methods such as find, create, retrieve, and update are replaced with mock implementations using mockResolvedValue() or mockRejectedValue() as appropriate.

Test Results

The following results were obtained by executing the test suite using the command `npm test`:

```
> smart-chat@1.0.0 test
> NODE_ENV=test jest

 PASS  tests/auth.test.js
 PASS  tests/message.test.js
 PASS  tests/user.test.js
 PASS  tests/chat.test.js
 PASS  tests/feedback.test.js

Test Suites: 5 passed, 5 total
Tests:       32 passed, 32 total
Snapshots:   0 total
Time:        1.843 s, estimated 2 s
Ran all test suites.
::ffff:127.0.0.1 - - [18/Jun/2025:10:55:02 +0000] "PUT /api/user/60f7a6f8d5a4c12d4c8b4567 HTTP/1.1" 200 555 "-" "-"
::ffff:127.0.0.1 - - [18/Jun/2025:10:55:02 +0000] "PUT /api/user/60f7a6f8d5a4c12d4c8b4567 HTTP/1.1" 404 77 "-" "-"
::ffff:127.0.0.1 - - [18/Jun/2025:10:55:02 +0000] "GET /api/chat/user123/list HTTP/1.1" 200 102 "-" "-"
::ffff:127.0.0.1 - - [18/Jun/2025:10:55:02 +0000] "GET /api/chat/user123/list HTTP/1.1" 404 44 "-" "-"
::ffff:127.0.0.1 - - [18/Jun/2025:10:55:02 +0000] "DELETE /api/chat/chat123 HTTP/1.1" 204 - "-" "-"
::ffff:127.0.0.1 - - [18/Jun/2025:10:55:02 +0000] "DELETE /api/chat/nonexistent HTTP/1.1" 404 44 "-" "-"
::ffff:127.0.0.1 - - [18/Jun/2025:10:55:02 +0000] "POST /api/feedback/ HTTP/1.1" 201 226 "-" "-"
::ffff:127.0.0.1 - - [18/Jun/2025:10:55:02 +0000] "POST /api/feedback/ HTTP/1.1" 404 29 "-" "-"
::ffff:127.0.0.1 - - [18/Jun/2025:10:55:02 +0000] "POST /api/feedback/ HTTP/1.1" 404 29 "-" "-"
::ffff:127.0.0.1 - - [18/Jun/2025:10:55:02 +0000] "POST /api/feedback/ HTTP/1.1" 400 52 "-" "-"
::ffff:127.0.0.1 - - [18/Jun/2025:10:55:02 +0000] "POST /api/feedback/ HTTP/1.1" 400 38 "-" "-"
```

Figure 14 - Backend Test Results: Output from running automated tests with `npm test`.

Conclusion

The Jest test suite confirms that the backend system operates as expected under both typical and edge case scenarios. All test suites and test cases passed successfully. By covering controllers, middleware, and routes, the backend testing strategy provides confidence in the system's stability and correctness.

8.2. Frontend Validation

Overview

This document outlines the frontend testing strategy implemented using the Jest framework for the frontend components of the system. These tests validate the integrity, behavior, and user experience of the chat application's messaging logic, ensuring that critical frontend components interact correctly with the backend via Socket.io.

Test Frameworks and Tools

All frontend tests are written using:

- **Jest**: JavaScript testing framework for assertions and mocking.
- **React Testing Library**: Provides utilities to test React components in a way that resembles real user interaction.

Test Coverage

1. **useSocket.test.js**

   Purpose

   To test the custom React hook useSocket (useSocket.js), which encapsulates the logic for initializing and managing Socket.io connections and chat state.

   Key Aspects Covered

   - **Socket Initialization and Cleanup**
     Verifies that a Socket.io connection is established when the component mounts and is correctly cleaned up when unmounted.
   - **Message Sending Logic**
     Validates that:
     - When a user sends a message in a new conversation, createNewChat is triggered.
     - When continuing an existing conversation, sendMessageToExistingChat is used.
   - **Chat Selection Handling**
     Ensures that selecting a chat correctly updates state variables such as selectedChatId, isLoading, and chatMessages.
   - **Resource Cleanup**
     Confirms that disconnections and event listeners are properly removed on component unmount, avoiding memory leaks or duplicate event emissions.

2. **chatHandlers.test.js**

Purpose

To test the event handling logic in chatHandlers.js, which is responsible for managing incoming chat-related socket events.

Key Aspects Covered

- **Chat List Updates**
  Validates that the chat list is updated when receiving new chats via socket events.

- **Chat Selection Logic**
  Confirms that selecting a chat updates the chat context and triggers message fetch.

- **New Chat Event Handling**
  Ensures the system correctly appends and manages newly created chats in the UI state.

3. **messageHandlers.test.js**

Purpose

To test the message event handlers in messageHandlers.js, which manage incoming messages and typing indicators from the server.

Key Aspects Covered

- **Bot Response Handling**
  Ensures that incoming AI-generated responses are appended to the correct chat context.

- **Message State Updates**
  Verifies that local state updates reflect messages in the chat window.

- **Typing and Feedback Logic**
  Tests system behaviors such as:

  - Showing the typing indicator when AI is generating a response.

  - Updating feedback status on message objects when users submit feedback.

Summary Table

Test File	Main Source(s) Tested	Main Features Covered
useSocket.test.js	useSocket.js, socket.js	Socket connection, message sending, chat selection, cleanup
chatHandlers.test.js	chatHandlers.js	Chat list updates, chat selection, new chat events
messageHandlers.test.js	messageHandlers.js	Bot responses, message updates, typing indicator, feedback

Test Results

The following results were obtained by executing the test suite.

```
> smart-chat-front@0.0.0 test
> jest --coverage

 PASS  tests/socket.test.js
 PASS  tests/useSocket.test.js
 PASS  tests/AuthContext.test.jsx
 PASS  tests/SignInForm.test.jsx
 PASS  tests/SignUpForm.test.jsx
 PASS  tests/MessageInput.test.jsx
 PASS  tests/ChatList.test.jsx
 PASS  tests/MessageComponent.test.jsx
 PASS  tests/auth.test.js
 PASS  tests/FormButton.test.jsx
 PASS  tests/UserPreferencesModal.test.jsx
 PASS  tests/messageHandlers.test.js
 PASS  tests/main.test.jsx
 PASS  tests/AuthContainer.test.jsx
 PASS  tests/App.test.jsx
 PASS  tests/chat.test.js
 PASS  tests/MessageList.test.jsx
 PASS  tests/LoadingSpinner.test.jsx
 PASS  tests/ChatWindow.test.jsx
 PASS  tests/user.test.js
 PASS  tests/connectionHandlers.test.js
 PASS  tests/feedback.test.js
 PASS  tests/api.conf.test.js
 PASS  tests/chatHandlers.test.js
 PASS  tests/AppContainer.test.jsx
---------------------------|---------|----------|---------|---------|-------------------
File                       | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
---------------------------|---------|----------|---------|---------|-------------------
All files                  |     100 |      100 |     100 |     100 |
 src                       |     100 |      100 |     100 |     100 |
  App.jsx                  |     100 |      100 |     100 |     100 |
 src/api                   |     100 |      100 |     100 |     100 |
  api.conf.js              |     100 |      100 |     100 |     100 |
 src/components/auth       |     100 |      100 |     100 |     100 |
  FormButton.jsx           |     100 |      100 |     100 |     100 |
  SignInForm.jsx           |     100 |      100 |     100 |     100 |
  SignUpForm.jsx           |     100 |      100 |     100 |     100 |
 src/components/chat       |     100 |      100 |     100 |     100 |
  ChatList.jsx             |     100 |      100 |     100 |     100 |
  ChatWindow.jsx           |     100 |      100 |     100 |     100 |
  MessageComponent.jsx     |     100 |      100 |     100 |     100 |
  MessageInput.jsx         |     100 |      100 |     100 |     100 |
  MessageList.jsx          |     100 |      100 |     100 |     100 |
 src/components/general    |     100 |      100 |     100 |     100 |
  LoadingSpinner.jsx       |     100 |      100 |     100 |     100 |
  UserPreferencesModal.jsx |     100 |      100 |     100 |     100 |
 src/containers            |     100 |      100 |     100 |     100 |
  AppContainer.jsx         |     100 |      100 |     100 |     100 |
  AuthContainer.jsx        |     100 |      100 |     100 |     100 |
 src/context               |     100 |      100 |     100 |     100 |
  AuthContext.jsx          |     100 |      100 |     100 |     100 |
 src/handlers/hooks        |     100 |      100 |     100 |     100 |
  useSocket.js             |     100 |      100 |     100 |     100 |
 src/handlers/socket       |     100 |      100 |     100 |     100 |
  chatHandlers.js          |     100 |      100 |     100 |     100 |
  connectionHandlers.js    |     100 |      100 |     100 |     100 |
  messageHandlers.js       |     100 |      100 |     100 |     100 |
---------------------------|---------|----------|---------|---------|-------------------

Test Suites: 25 passed, 25 total
Tests:       184 passed, 184 total
Snapshots:   0 total
Time:        2.223 s
Ran all test suites.
```

Figure 15 - Frontend Test Results: Output from running automated tests

Conclusion

These tests provide a robust foundation to ensure that the **SmartChat** frontend behaves as expected during communication workflows. They simulate user behavior, socket events, and UI changes–ensuring reliable and maintainable interactions between components and server responses.

As new components or features are introduced, this document should be updated to maintain full test coverage transparency.

8.3. Conversation Flow Integration

This test verifies the full conversation flow, covering the communication pipeline from the frontend to the backend and back, ensuring correct logic and data transmission for both initiating a new chat and continuing an existing one.

Overview

This logic is designed not only to maintain conversation continuity but also to **optimize performance and platform utilization**.

When a user starts a **new chat**, the system executes a **resource-intensive initialization process** that analyzes the user's prompt and other contextual parameters, one of them is to determine the **most suitable AI platform** (e.g ChatGPT, Gemini, Claude) to handle the conversation. This selection process involves complex logic and is critical for ensuring high-quality responses tailored to the user's needs.

When the user **continues an existing conversation**, the system reuses the **previously selected platform** to maintain consistency. However, it's important to note that this is **not a simple message relay** – the application still performs **non-trivial logic** such as **analyzing the prompt**, **tracking the message history**, **applying user preferences and insights from prior feedback (if given)**, and **managing the interaction flow** to ensure that responses remain coherent, relevant, and personalized.

This layered approach allows the system to strike a balance between **smart decision-making** and **runtime efficiency**, ensuring that both new and ongoing conversations are handled intelligently and responsively.

What is Being Tested

Frontend

- **userSocket.js -** handleSend function calls 'createNewChat' when a user starts a new conversation, calls 'sendMessageToExistingChat' when user continues an existing conversation.
- **Socket.js -** 'createNewChat' emits the 'create_chat' socket event, 'sendMessageToExistingChat' emits the 'send_message' socket event.

Backend

- **socket.js -** socket on:'create_chat' calls handelCreateChat function, socket on:'send_message' calls handelSendMessage function
- <u>When Create New Chat</u>
    - **chatEventHandler.js -** handelCreateChat invokes processAIResponse
    - **response-processor.handler.js -** processAIResponse generate and return responseData which includes the *final refined answer*.

- When Send Message
    - **messageEventHandler.js -** handelSendMessage generate bot message and call processAIMessage
    - **response-processor.handler.js -** processAIMessage call responseToNewMessage function
    - **responseToNewMessage.js -** responseToNewMessage generates *bot messages.*

Conversation Flow Diagram

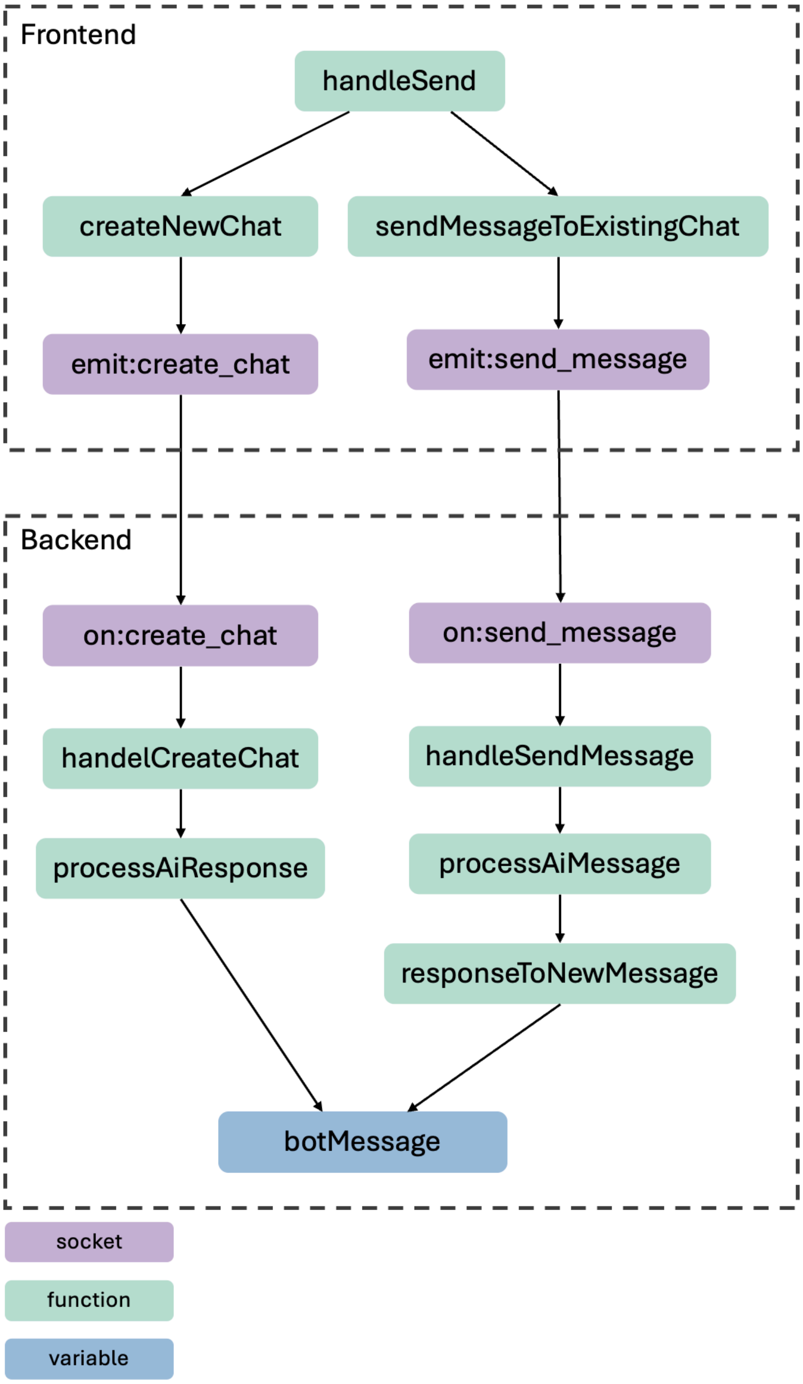

Figure 16 - Conversation Flow Diagram

Test Results

The system determines the appropriate action–whether to initiate a new chat or continue an existing one–and triggers the corresponding logic flow to handle the user's input effectively. The system is working as expected, confirming that the test was successful.

Starting a New Conversation

- **System UI (chat)**

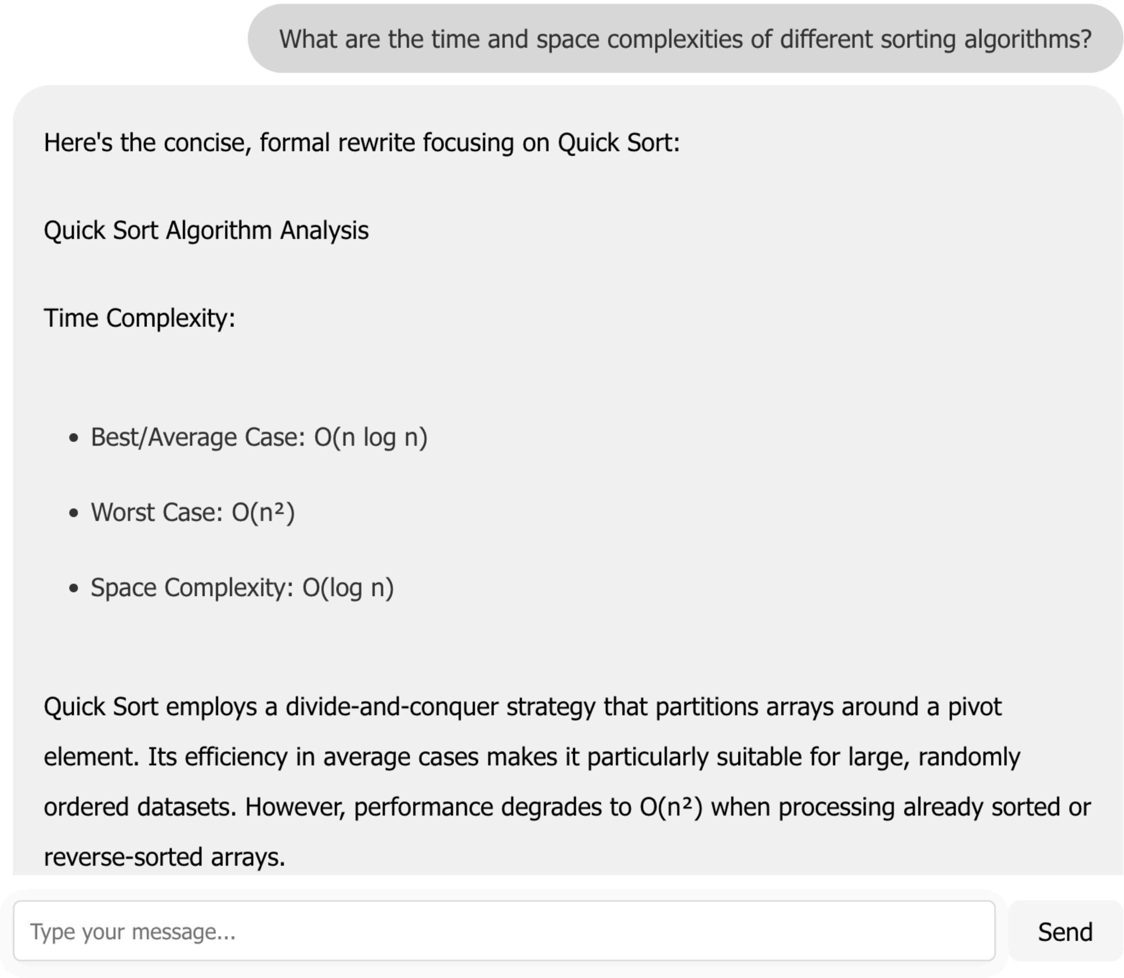

Figure 17 - UI screenshot of starting a new conversation

- **Front Logs**

```
Starting Test                  AppContainer.jsx?t=1750329654999:102
====================           AppContainer.jsx?t=1750329654999:103
Frontend
====================
User messages: What are        AppContainer.jsx?t=1750329654999:104
the time and space complexities of different sorting
algorithms?
User want to create a new      AppContainer.jsx?t=1750329654999:107
chat:
Creating new chat                       socket.js?t=1750325726518:162
```

Figure 18 - Frontend logs showing the new conversation flow.

- **Back Logs**

````
====================
Backend
====================
Start creating chat:
Handling create chat
Processing AI response
Final refined answer: Here's the concise, formal rewrite focusing on Quick Sort:

Quick Sort Algorithm Analysis

Time Complexity:
- Best/Average Case: O(n log n)
- Worst Case: O(n²)
- Space Complexity: O(log n)

Quick Sort employs a divide-and-conquer strategy that partitions arrays around a
 it particularly suitable for large, randomly ordered datasets. However, performa
 or reverse-sorted arrays.

Example implementation in Python:
```python
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)
```
````

Figure 19 - Backend logs showing the new conversation flow.

- **Summary**:
    - When a user creates a new conversation (new chat) the new message is received properly, the proper function is called and the socket event is triggered accordingly.
    - The triggered socket event initiates the creation of a new chat, generates the final response, and sends it back to the user.

Continue an existing conversation

- **System UI (same chat)**

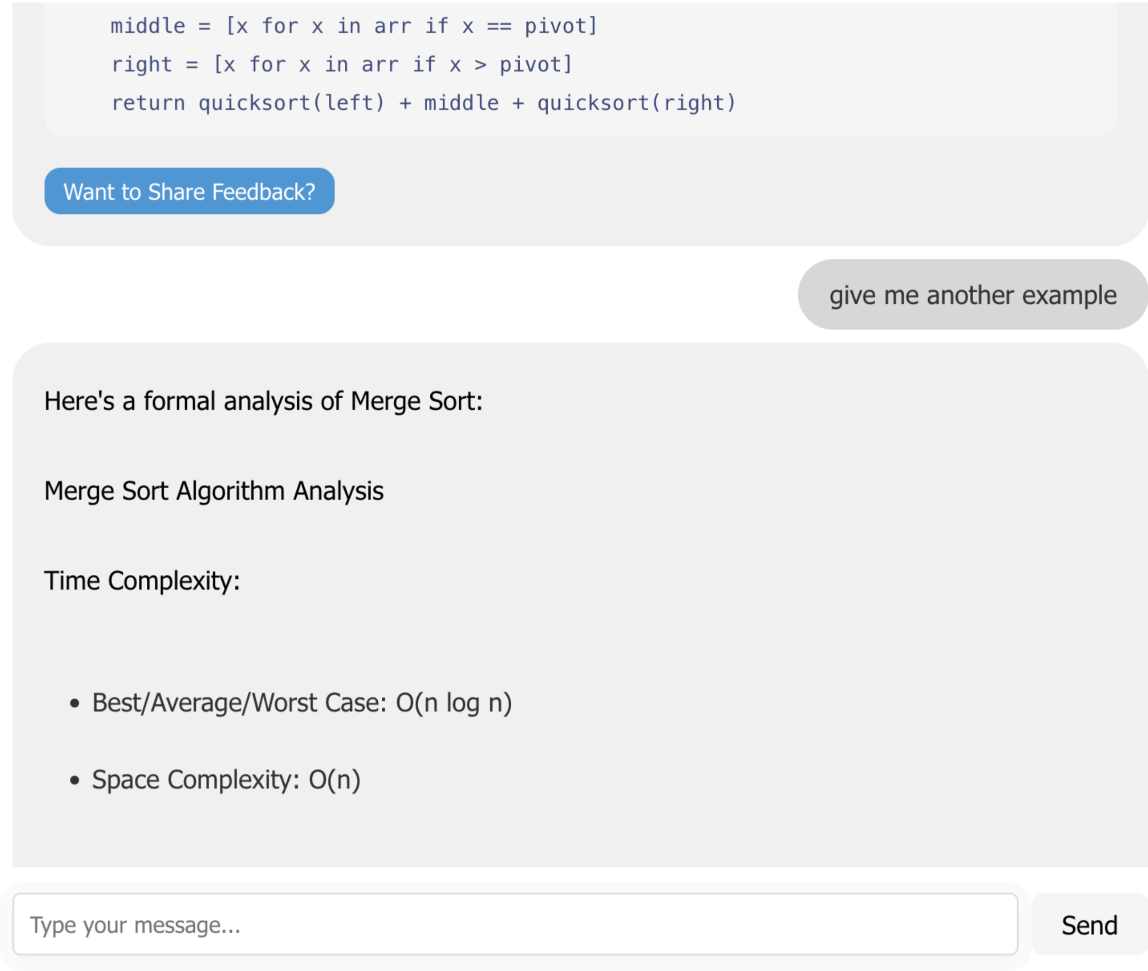

Figure 20 - UI screenshot of continuing an existing conversation.

- **Front Logs**

Starting Test	AppContainer.jsx?t=1750329654999:102
==================== Frontend ====================	AppContainer.jsx?t=1750329654999:103
User messages: give me another example	AppContainer.jsx?t=1750329654999:104
User want to continue existing chat:	AppContainer.jsx?t=1750329654999:118
Continue existing chat	socket.js?t=1750325726518:127

Figure 21 - Frontend logs showing the flow of continuing an existing conversation.

- **Back Logs**

```
====================
Backend
====================
Start sending message:
Handling send message
Processing AI message
Generating response to the new message
Selected platform: claude
Response generated: Here's a formal analysis of Merge Sort:

Merge Sort Algorithm Analysis

Time Complexity:
- Best/Average/Worst Case: O(n log n)
- Space Complexity: O(n)

Merge Sort utilizes a stable, divide-and-conquer methodology that consistently delivers
in guaranteed O(n log n) complexity, though it requires additional memory allocation.

Example implementation in Python:
```python
def mergesort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = mergesort(arr[:mid])
    right = mergesort(arr[mid:])

    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result
```
```

Figure 22 - Backend logs showing the flow of continuing an existing conversation.

- **Summary**:
    - When a user continues an existing conversation the new message is received properly, the proper function is called and the socket event is triggered accordingly.
    - The triggered socket event determines the selected platform, generates a response, and returns it to the user.

8.4. Decision-Making Logic

Overview

This test validates the **decision-making logic** performed by the system. Specifically, it ensures that SmartChat correctly selects the most appropriate platform per prompt and **generates a relevant and refined response**, without requiring the user to review or compare multiple answers manually.

This confirms that the platform selection and response refinement logic is working as expected.

Without the System:

- The user manually sends **Prompt 1** to:
  - ChatGPT
  - Gemini
  - Claude
- Each platform returns a response:
  - ChatGPT → Answer 1
  - Gemini → Answer 1
  - Claude → Answer 1
- The user **reads all answers**, compares them, and manually decides which one is best.
- The user repeats the same process for **Prompt 2**, and so on.

With the System (SmartChat):

- The user sends **Prompt 1** to **SmartChat**.
- The system evaluates multiple platforms **on behalf of the user**, selects the most suitable one based on internal logic (e.g., prompt analysis, personal information and preferences, feedback, etc), and returns a single refined **Answer 1**.
- The user sends **Prompt 2** to SmartChat, and **receives an optimized answer automatically**, and so on.

Tested User Profile Settings:

- **Role**: Student
- **Expertise Level**: Intermediate
- **Answer Style**: Concise

- **Example Count**: One
- **Tone**: Formal

Test Details:

Test 1

Prompt #1:

I'm a student in the software engineering department in the university.

This semester I'm taking the course 'Operating Systems'.

According to the syllabus, the course describes the general (conceptual), principles related to modern operating systems.

As well as how these principles are implemented in specific modern operating systems.

Namely: Linux, Windows and MacOS.

By modern I mean: an operating system that I installed and executed in a computer with a 64-bit multi-core CPU.

The professor described to us that from the kernel point of view, processes and threads are objects that it creates and manages.

The kernel also provides various system calls that are related processes and threads. Such as system calls to create a process or a thread, terminate the execution of a process or a thread, etc.

How does the fact that an operating system creates and manages processes and threads relates to the fact that the major role of an operating system is to provide virtual machine?

Answer #1:

The functionality of a virtual machine (VM) within an operating system is fundamentally dependent on its kernel's management of processes and threads. The kernel creates the illusion of dedicated hardware for each program by meticulously managing resources and isolating processes. For example, the kernel allocates a distinct virtual address space to each process, preventing interference and ensuring data integrity. This isolation, coupled with resource scheduling, allows multiple programs to concurrently execute on a single physical machine, each appearing to possess its own dedicated hardware.

Prompt #2:

Is it correct to describe each process and each thread is provided with a virtual machine?

Answer #2:

No. While a virtual machine provides the illusion of dedicated hardware, it is not allocated to each process or thread individually. The operating system kernel manages resources at a higher level, providing a virtual machine to the entire system, upon which processes and threads operate. For example, a single virtual machine might host multiple processes, each with its own virtual address space managed by the kernel within that VM.

Prompt #3:

Following our discussion so far, I want to phrase my questions in another way: is it correct to describe that the concepts of processes and threads are the kernel's implementation mechanism, to 'create' this illusion, this fake, named 'a virtual machine'?

Answer #3:

No. Processes and threads are mechanisms used by the kernel to implement the virtual machine, not the mechanism that creates the illusion itself. The virtual machine is a higher-level abstraction encompassing resource management, process isolation, and other functionalities. For example, the kernel uses processes and threads to manage the execution of multiple programs within a single virtual machine, but the virtual machine itself is the overarching environment providing the illusion of dedicated hardware.

Prompt #4:

1. Describe what does "execution context" means?

   Provide a very detailed, accurate, academic and highly technical explanation.

2. Describe what it means that the process gets "its own execution context"?

   Provide a very detailed, accurate, academic and highly technical explanation.

3. Does a process is also considered by the operating system to be a unit of execution? Exactly like a thread?

   And if so, what is really the difference between a process and a thread from the operating system point of view, as far as execution is concerned?

   Does a process also get scheduled to the CPU?

   Provide a very detailed, accurate, academic and highly technical explanation.

Answer #4:

1. An execution context is the complete set of information that defines the state of a process or thread at a given point in time. This includes the program counter (PC), register values, stack, heap, and memory mapping. It represents everything necessary for the operating system's scheduler to resume execution of the process or thread exactly where it left off.

2. A process "getting its own execution context" means that the operating system allocates a unique and independent set of resources–as described above–to that process. This isolation ensures that one process cannot interfere with the execution of another. This isolation is crucial to the virtual machine's functionality.

3. Yes, a process is a unit of execution, and it is scheduled to the CPU, just like a thread. The key difference lies in the level of isolation and resource ownership. A process has its own independent virtual address space, while threads within the same process share the same address space. This shared address space allows for efficient inter-thread communication but necessitates mechanisms to prevent race conditions and other concurrency issues. For example, consider a web server process. This process might handle multiple client requests concurrently using multiple threads, all sharing the same code and data structures within the process's address space. Each thread, however, maintains its own execution context (stack, register values, etc.) within that shared space.

Prompt #5:

Provide code examples from both Linux and Windows kernels, illustrating thread and process creation and scheduling.

Answer #5:

Providing kernel-level code examples for process and thread creation and scheduling is impractical within this concise format. The code is extensive, system-specific, and deeply intertwined with the kernel's architecture. A single example would be insufficient to illustrate the complexities involved. Further, direct code access to kernel internals requires significant privilege and understanding of the operating system's architecture. Consult relevant kernel documentation and source code for detailed examples.

Prompt #6:

Explain to me:

1. The full task_struct structure managed by Linux, together with an explanation of each of the fields.
2. The full EPROCESS structure managed by Windows, together with an explanation of each of the fields.
3. The full ETHREAD structure managed by Windows, together with an explanation of each of the fields.
4. The full corresponding structure is managed by macOS, together with an explanation of each of the fields.

Answer #6:

Providing a complete description of the task_struct (Linux), EPROCESS and ETHREAD (Windows), and the macOS equivalent (which varies across versions and is not a single, straightforward structure like the others) is beyond the scope of a concise response. These structures are extremely large and complex, containing hundreds of fields. A single example, focusing on a small subset of key fields, is impractical due to their size and intricate interdependencies. Consulting the respective operating system's kernel source code and documentation is necessary for a comprehensive understanding. For instance, the Linux kernel's documentation for task_struct is a starting point for exploration.

Prompt #7:

I want all the explanations to be very refined and detailed, Ignore the demand of the concise response

Answer #7:

A comprehensive explanation of the task_struct (Linux), EPROCESS and ETHREAD (Windows), and their macOS equivalents is infeasible within a reasonable response length. These data structures are exceptionally large and complex, evolving across kernel versions. A complete field-by-field description would be voluminous and quickly become outdated. However, a high-level overview with examples of key fields can be provided.

Linux task_struct: This structure represents a process or thread in the Linux kernel. It contains hundreds of fields managing various aspects of the process's lifecycle, including:

- state: Represents the current state of the process (e.g., TASK_RUNNING, TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE).
- pid: The process ID (PID), a unique identifier for the process.
- mm: A pointer to the mm_struct, which describes the process's memory map (virtual address space). This is crucial for memory management and isolation.
- thread.rsp: (For threads) Points to the thread's stack.
- thread.sp: (For threads) Stack pointer.
- thread.flags: Flags indicating various thread states and attributes.
- real_parent and parent: Pointers to the parent process.

Example Field (Linux task_struct): mm The mm_struct points to the process's memory management structure. This structure details the virtual memory mappings for the process, specifying which virtual addresses map to which physical pages, page permissions (read, write, execute), and other memory-related attributes. Its existence is central to the virtual memory subsystem and the creation of the illusion of dedicated hardware.

Windows EPROCESS and ETHREAD: Windows uses two key structures: EPROCESS for processes and ETHREAD for threads. EPROCESS contains high-level process information, including security attributes, handles, and the process's memory space. ETHREAD contains thread-specific information such as the thread's context, scheduling priority, and stack pointer.

- EPROCESS Key Fields: UniqueProcessId (PID), ImageFileName, pointers to memory structures, security descriptors.
- ETHREAD Key Fields: Tcb (Thread Control Block), stack pointer, registers, scheduling priority.

macOS: macOS's kernel structures are less readily documented and more internal. They are also significantly more intricate than the Linux and Windows equivalents, making a concise description impractical. The underlying concepts remain similar, though. Instead of a single structure like task_struct, macOS uses a more complex and distributed approach to represent processes and threads.

Accessing and interpreting the kernel structures directly requires advanced knowledge of operating system internals, kernel programming, and significant caution due to the potential for system instability. Consulting the relevant OS kernel documentation and source code is essential for detailed understanding.

Test 2

Prompt #1:

I'm a student in the software engineering department in the university.

This semester I'm taking the course 'Operating Systems'.

According to the syllabus, the course describes the general (conceptual)principles related to modern operating systems.

As well as how these principles are implemented in specific modern operating systems.

Namely: Linux, Windows and MacOS.

By modern I mean: an operating system that I installed and executed in a computer with a 64-bit multi-core CPU.

What is 'object code'?

Please provide a precise, rigorously detailed, and contemporary definition of this term.

Answer #1:

Object code is the machine-specific binary representation of a compiled program, preceding the linking stage. It comprises machine instructions, data, a symbol table mapping symbolic names to relative addresses, and relocation information specifying addresses needing adjustment during linking.

For example, a simple C function compiled for x86-64 Linux will produce an object file (.o) containing sections like .text (machine code), .symtab (symbol table), and .reloc (relocation information), all crucial for the linker to create a fully executable program. The object file is not self-contained; linking resolves relative addresses to create an executable.

Prompt #2:

What is the object module format supported by Linux version 6.11?

Answer #2:

Linux kernel version 6.11 primarily uses the ELF (Executable and Linkable Format) object module format.

Prompt #3:

What are the constituent parts of this object module?

I request an extended and comprehensive exposition, thoroughly explaining and contextualizing these components.

Answer #3:

An ELF object module comprises several sections, each serving a specific purpose in the compilation and linking process. Key sections include the .text section containing the machine code instructions; the .data section holding initialized global and static variables; the .bss section for uninitialized global and static variables; the .rodata section for read-only data such as string constants; the .symtab section containing a symbol table mapping symbolic names (e.g., function and variable names) to their addresses; and the .reloc section specifying addresses that require adjustment during the linking process to resolve relative addresses into absolute addresses.

For example, an ELF object file for a C program might have a .text section with the machine code for a function, a .data section containing the value of a global integer variable, and a .symtab entry linking the name of that variable to its location within the .data section. These sections are crucial for the linker to combine multiple object files and resolve external references, ultimately creating an executable file.

Prompt #4:

What is the 'text section'?

Please provide an accurate, rigorously detailed, and modern definition of this term.

Answer #4:

The .text section of an ELF object module contains the machine code instructions of a program. This section holds the compiled binary representation of the program's executable code, ready for execution by the CPU. For example, a C function's compiled instructions, after optimization by the compiler, will reside within the .text section. The linker combines the .text sections from multiple object files to create the final executable's code segment.

Prompt #5:

What is the data section'?

Please provide an accurate, rigorously detailed, and modern definition of this term.

Answer #5:

The .data section of an ELF object module contains initialized global and static variables. This section holds the initial values assigned to these variables at compile time. These variables retain their values throughout the program's execution.

For example, a global integer variable declared as int global_var = `10;` in a C program will have its value (`10`) stored in the .data section of the resulting object file. The linker combines the .data sections from multiple object files to create the data segment of the final executable.

Prompt #6:

Which component of the Linux kernel is responsible for loading an object module into memory?

Answer #6:

The Linux kernel's component responsible for loading object modules into memory is the kernel's dynamic loader, typically ld.so (or a variant thereof depending on the specific distribution and kernel version). This loader handles the loading, relocation, and linking of shared libraries (.so files) and other dynamically linked object modules at runtime. For example, when a program using a dynamically linked library is executed, ld.so maps the library's .text and .data sections into the program's address space, resolves symbol references, and performs any necessary relocation adjustments before transferring control to the program's entry point.

Prompt #7:

What is a 'memory image'?

Please provide an accurate, rigorously detailed, and modern definition of this term.

Answer #7:

A memory image is a representation of a program's state in the computer's main memory at a particular point in time. It includes the program's code (typically from the .text section of its executable file), its initialized data (from the .data section), its uninitialized data (from the .bss section), the stack, the heap, and the program's registers. For example, a running process's memory image would contain the current values of its variables, the instructions being executed, and the contents of its stack and heap. This image is crucial for debugging, process management (e.g., creating a core dump), and virtual memory management.

Prompt #8:

What is the relationship between an object module and its corresponding memory image?

Answer #8:

The object module provides the blueprint for the memory image. The memory image is the runtime instantiation of the object module's contents within the computer's RAM. The object module's .text, .data, and .bss sections are loaded into memory, forming the code segment, data segment, and uninitialized data segment of the memory image, respectively. The linker resolves addresses and performs relocation, mapping the relative addresses in the object module to absolute addresses in the memory image.

For example, a function's instructions in the .text section of an object module will occupy specific memory locations within the code segment of the memory image, allowing the CPU to execute them. The memory image also includes dynamically allocated memory (heap) and the program's stack, which are not directly represented in the object module but are essential components of the program's runtime environment.

Test Result

During this test, the user submitted a sequence of various prompts to the system. Each prompt was handled by SmartChat, which internally performed all necessary processing and returned a single refined response–without exposing the decision-making logic to the user.

- All processing steps, including platform selection and response generation, were thoroughly logged for each prompt.
- The responses provided were consistently relevant, clear, and aligned with the user's configured profile preferences.
- At no point did the user need to compare or select between multiple AI-generated answers.
- For follow-up messages within the same conversation, the system consistently reused the previously selected platform to maintain conversational continuity.

All platform selection details, evaluation decisions, and full request/response cycles are documented in supplementary logs listed in **Appendix 1**.

Conclusion

The system successfully applied its platform selection logic based on prompt characteristics and user preferences. It delivered a unified, refined response per prompt without exposing the complexity of multi-platform processing to the user. The test confirms that the decision-making component of SmartChat is functioning correctly and reliably across varying prompt scenarios.

8.5. Feedback Adaptation

Overview

This test verifies that the feedback mechanism within SmartChat operates reliably and effectively. When a user provides feedback on a response–either through a rating, qualitative comment, or by specifying aspects like clarity or detail level–the system is expected to store this information and apply it to future interactions.

Tested User Profile Settings

- **Role**: Software Engineer
- **Expertise Level**: Intermediate
- **Answer Style**: Concise
- **Example Count**: One
- **Tone**: Formal

Test Details

Prompt #1:

Can you explain the Merge Sort algorithm?

Answer #1:

Merge Sort is a divide-and-conquer algorithm. It recursively splits the array into halves, sorts each half, and merges them.

Example (Python):

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)


def merge(left, right):
    result = []
```

```
i = j = 0
while i < len(left) and j < len(right):
  if left[i] < right[j]:
    result.append(left[i])
    i += 1
  else:
    result.append(right[j])
    j += 1
return result + left[i:] + right[j:]
```

Feedback #1:

- **How would you rate this answer?** 6
- **Any feedback you'd like to share?** Could use more explanation and slightly longer response.
- **Were there enough examples?** Yes
- **Was the answer detailed enough?** No
- **Was the answer clear?** Yes
- **Was it too long? Would you prefer something shorter?** too short

---

Prompt #2:

What are some best practices for using dependency injection in large-scale Node.js applications?

Answer #2:

Using Dependency Injection in large-scale Node.js applications helps improve modularity, testability, and maintainability.

Best Practices:

1. **Use an IoC Container** like InversifyJS to manage bindings and lifecycles.
2. **Favor constructor injection** for clarity.
3. **Avoid global state** by scoping services properly.
4. **Design for interfaces** to decouple modules.

Example:
When building a microservice, inject a logging service into each component instead of importing it directly.

Feedback #2:

- **How would you rate this answer?** 8
- **Any feedback you'd like to share?** Could use more explanation and slightly longer response.
- **Were there enough examples?** Yes
- **Was the answer detailed enough?** Yes
- **Was the answer clear?** No
- **Was it too long? Would you prefer something shorter?** Just fine

---

Prompt #3:

How can you ensure data integrity and consistency in an event-driven system with eventual consistency?

Answer #3:

Ensuring data integrity in an eventually consistent system requires a combination of strategies:

1. **Idempotent event handling:** Events must be safe to process multiple times.
2. **Event versioning:** Handle changes in event schema over time.
3. **Distributed transaction patterns** like the Saga pattern, which breaks a transaction into steps, each with a compensating action.
4. **Message deduplication** using IDs or event logs.
5. **Read-model reconciliation** via background jobs to resync denormalized views.

Example:
A hotel booking service may emit a "RoomReserved" event. If the payment fails, a compensating "CancelReservation" event rolls back the state.

Explanation:
These practices help developers build fault-tolerant, consistent experiences despite asynchronous, distributed workflows. It also improves auditability and recovery.

Feedback #3:

- **How would you rate this answer?** 9
- **Any feedback you'd like to share?** The format and tone were spot on. Great use of structure and example explanation.

- **Were there enough examples?** Yes
- **Was the answer detailed enough?** Yes
- **Was the answer clear?** Yes
- **Was it too long? Would you prefer something shorter?** Just fine

Conclusion

This test confirms that SmartChat effectively incorporates user feedback into its response logic. Each feedback cycle influenced the structure, tone, and content of subsequent answers, resulting in increasingly aligned and contextually appropriate responses. The system's behavior meets the expectations of a responsive and adaptive AI assistant.

8.6. Feedback Free-Text Analysis

Overview

This test validates the functionality and accuracy of SmartChat's NLP-driven feedback interpretation module, which processes qualitative (free-text) user feedback. The module classifies such feedback into structured dimension-value pairs using a custom-trained NLP model built with NLP.js. These dimensions reflect user expectations around clarity, tone, length, detail, and example count.

The goal is to ensure that the system can extract actionable insights from natural language input, thereby enhancing its ability to personalize responses and improve conversational relevance based on user input.

Dimensions and values

```
{
 examples: ['none', 'one', 'multiple'],
 tone: ['formal', 'casual', 'neutral'],
 length: ['shorter', 'longer'],
 detail: ['concise', 'detailed'],
 clarity: ['simplify', 'more_clear']
}
```

Labels

Dimension	Feedback Value Suggestions
examples	none, one, multiple
tone	formal, casual, neutral
length	shorter, longer
detail	concise, detailed
clarity	simplify, rephrase or more_clear

Figure 23 - Feedback dimensions and possible classified values used in the NLP analysis.

Directory Structure

```
/feedback-nlp
├── train.js          # Trains the NLP model on labeled feedback
├── classify.js       # Loads and uses the trained model for classification
├── test-feedback.js  # Script to simulate feedback analysis using test input
├── data/
│   └── feedback.json # JSON dataset containing training feedback samples
├── model.nlp         # (Generated) Serialized trained NLP model
```

Test Procedure

1. **Dataset Format**

   The training dataset is located in the data/feedback.json file. This dataset is organized as a flat object in which each key represents a combination of a feedback dimension and its corresponding value (e.g., "examples.multiple" or "tone.formal"). Each key maps to an array of representative user feedback strings expressing that specific intent.

2. **Model Training**

   To initiate training, execute the train.js script. This script processes the grouped examples, extracts the labels from the key names (e.g., examples.multiple → dimension: examples, value: multiple), and uses them to train a natural language classification model using NLP.js.

   Upon successful training, the model is serialized and stored as model.nlp, which later be used for inference.

3. **Testing**

   To test the trained model, we execute the test-feedback.js script. This script simulates the operation of an analysis middleware by passing sample user feedback through the classification pipeline. It invokes classify.js, which loads the model.nlp file, processes the input, and returns predicted dimension-value pairs representing the system's interpretation of the feedback.

Test Results

1. **Confidence Threshold Filtering**

   Following classification, each predicted label includes a confidence score in the range `[0,1][0, 1][0,1]`, representing the model's certainty. Based on empirical testing, it was observed that predictions with a score below **`0.4`** were often unreliable or semantically incorrect. To ensure high-quality outputs, a **minimum confidence threshold of `0.4`** was established.

During inference, the system filters out low-confidence predictions using the following logic:

```
for (const category in nlpResults) {
  const { value, score } = nlpResults[category];
  if (score >= 0.4) {
    highConfidenceResults[category] = { value, score };
  }
}
```

Figure 24 - Confidence threshold filtering process for NLP classification.

Only dimension-value pairs with a score ≥ `0.4` are retained in the final classification output. This strategy improves overall reliability by reducing false positives and noisy label predictions.

2. **Result File**

   Detailed results from the test runs can be found in the attached log files listed in **Appendix 2**.

Conclusion

The feedback free-text analysis module demonstrates strong alignment with expected performance goals. The NLP model successfully interprets natural language feedback and maps it to structured configuration values, allowing the system to adapt future responses based on user preferences.

The use of a confidence threshold ensures robustness by eliminating noisy or low-certainty predictions. Based on the test results, the classification model is reliable enough for use in a production environment where response adaptation is critical.

8.7. Validation Prompt per User

Overview

This document outlines the validation process for evaluating personalized responses in the SmartChat system. The validation is based on a *Validation Prompt* generated per user, using individual profile attributes such as role, expertise level, preferred answer style, number of examples, and tone.

Each response returned from the AI platforms is assessed against these criteria to verify that it not only delivers accurate information, but also matches the user's expectations and communication preferences. This ensures that the final merged answer is both technically correct and personally relevant.

Test Details:

Tested Prompt:

What are the differences between multithreading and multiprocessing in operating systems?

User 1

Profile:

- **Role**: Student
- **Expertise Level**: Beginner
- **Answer Style**: Concise
- **Example Count**: None
- **Tone**: Neutral

Answer:

> Multithreading uses multiple threads within a single process, sharing memory and requiring synchronization to avoid race conditions. Multiprocessing uses multiple independent processes, each with its own memory space, eliminating the need for explicit synchronization but requiring inter-process communication for data sharing. The choice depends on the application's needs, balancing the efficiency of shared memory against the overhead of inter-process communication.

Analysis:

> Based on the defined user profile (a beginner-level student) and the selected preferences, the answer provided is a very good fit. The style is concise, with short and clear sentences, as requested. No examples were included, which matches the "none" preference. The tone is neutral–not too formal and not too casual–making it suitable for students. In terms of knowledge level, the explanation covers basic concepts like shared memory and

inter-process communication, using simple and clear language. Overall, the answer matches what a beginner would expect when looking for a general understanding of the differences between multithreading and multiprocessing.

User 2

Profile:

- **Role**: Lecturer
- **Expertise Level**: Advanced
- **Answer Style**: Detailed
- **Example Count**: Multiple
- **Tone**: Formal

Answer:

Dear Nave,

This response aims to provide a detailed and formal analysis of multithreading and multiprocessing, clarifying their nuanced distinctions through multiple examples. My previous responses, while comprehensive, lacked the depth and precision required for a thorough understanding. This revision addresses this deficiency.

Multithreading: A Detailed Perspective

Multithreading, a mechanism within a single process, allows for the concurrent execution of multiple threads sharing the same address space. This shared memory model, while offering significant efficiency gains, introduces complexities that demand careful management. The operating system's scheduler interleaves the execution of these threads, creating the illusion of parallelism on single-core processors and achieving true parallelism on multi-core systems. This interleaving enhances responsiveness, a crucial advantage in many applications.

Advantages of Multithreading:

Enhanced Responsiveness: Multithreading is particularly beneficial in interactive applications. Consider a graphical user interface (GUI) application performing a lengthy calculation. Without multithreading, the application would freeze during this computation, rendering it unresponsive to user input. However, by offloading the calculation to a separate thread, the main thread remains free to handle user interactions, maintaining responsiveness. For example, a word processor can continue to accept user input (like typing) while simultaneously spell-checking a document in the background.

Efficient Resource Sharing: The shared memory space allows threads to access and modify data structures without the overhead of data copying inherent in inter-process communication. This is particularly advantageous for data-intensive applications. Imagine a large image processing application; multithreading allows different threads to work on separate parts of the image simultaneously, accessing and updating the shared image data directly. This significantly reduces processing time compared to a single-threaded approach.

Simplified Concurrency (in specific cases): For tasks naturally divisible into independent sub-tasks, multithreading can offer a simpler programming model than multiprocessing. For instance, processing a list of independent calculations can be readily parallelized using threads, simplifying the code compared to the more complex inter-process communication required in multiprocessing.

Challenges of Multithreading:

Race Conditions: Race conditions occur when multiple threads access and modify shared data concurrently without proper synchronization mechanisms. The final state of the data becomes unpredictable and dependent on the unpredictable order of thread execution, often resulting in erroneous outcomes. Consider a shared counter variable incremented by multiple threads. Without synchronization (e.g., mutex locks), each thread might read the counter's value, increment it, and write it back, leading to a final count significantly lower than expected.

Deadlocks: Deadlocks arise when two or more threads become blocked indefinitely, each waiting for another to release a resource that it holds. This often stems from improper locking sequences or circular resource dependencies. A classic example involves two threads, each holding a lock on a different resource and attempting to acquire the lock held by the other. Neither thread can proceed, resulting in a deadlock.

Livelocks: Livelocks represent a subtle yet significant concurrency problem. Threads continuously change their state in response to each other, creating a situation where no progress is made. Unlike deadlocks, threads are not blocked; they are actively consuming resources while remaining perpetually unproductive. Imagine two robots attempting to pass each other in a narrow corridor; they repeatedly step aside to let the other pass, resulting in a livelock.

Starvation: A thread might be repeatedly denied access to a shared resource, even when the resource is available. This is often caused by unfair scheduling algorithms or prioritization schemes. A low-priority thread might be perpetually preempted by higher-priority threads, effectively starving it of necessary resources.

Atomicity: Ensuring atomicity - the indivisibility of an operation - is crucial. Many operations seemingly atomic at a higher level are not at the hardware level, requiring explicit synchronization primitives (e.g., atomic instructions or locks) to guarantee their integrity.

Multiprocessing: A Precise Perspective

Multiprocessing involves the execution of multiple independent

processes, each possessing its own private memory space, resources, and

execution context. This isolation is a key advantage, although it necessitates

inter-process communication (IPC) mechanisms to facilitate data exchange.

Advantages of Multiprocessing:

Fault Isolation: The isolation of processes is a crucial benefit. A failure in one process generally doesn't impact others, enhancing the overall system's robustness and reliability.

This is particularly critical in safety-critical systems where a single process failure shouldn't compromise the entire system.

True Parallelism: On multi-core systems, multiprocessing readily achieves true parallelism, with multiple processes concurrently executing on different cores. This contrasts with multithreading, where true parallelism is less guaranteed due to scheduler limitations and resource contention.

Resource Management: Each process possesses its own virtual address space, simplifying memory management and reducing the risk of memory corruption or leaks affecting other processes.

Analysis:

Based on the defined user profile (a lecturer with advanced expertise) and the selected preferences, the answer is an excellent fit. The style is highly detailed, providing in-depth explanations structured into clear subtopics like advantages and challenges. Multiple examples are included, such as GUI responsiveness, image processing, and concurrency scenarios, matching the "multiple" preference. The tone is formal and professional, as expected for an academic audience. In terms of knowledge level, the explanation introduces advanced terminology and concepts such as *livelock*, *atomicity*, and *fault isolation*, making it well-suited for an expert reader seeking a deep technical comparison between multithreading and multiprocessing.

User 3

Profile:

- **Role**: Software Engineer
- **Expertise Level**: Advanced
- **Answer Style**: Concise
- **Example Count**: One
- **Tone**: Neutral

Answer:

Here's a concise, neutral rewrite focusing on the key differences between multithreading and multiprocessing: Multithreading and multiprocessing are two approaches to concurrent

execution:

Multithreading:

· Single process with multiple threads

· Shared memory space (efficient communication)

· Lower resource overhead

· Faster context switching (`0.5-1`μs)

Example: A web browser using multiple threads - one thread handles user interface updates while another loads web content, sharing the same

memory

space.

Multiprocessing:

· Multiple independent processes

· Isolated memory spaces

· Higher resource usage

· Slower context switching (100-150μs)

· Better for CPU-intensive tasks

The choice between them depends primarily on the application's needs:

multithreading is typically better for I/O-bound tasks, while

multiprocessing

is preferred for CPU-bound operations requiring true parallelism.

Analysis:

Based on the defined user profile (a software engineer with advanced expertise) and the selected preferences, the answer is a very good fit. The style is concise, clearly highlighting the key differences between multithreading and multiprocessing without unnecessary elaboration. One relevant example is provided, as requested, illustrating how a web browser uses multithreading for UI and content loading. The tone is neutral and professional, aligning with the preference for objectivity. In terms of knowledge level, the explanation includes technical terms and performance metrics (like context switching times), making it suitable for an advanced user seeking a quick yet precise comparison.

User 4

Profile:

- **Role**: Student
- **Expertise Level**: Intermediate
- **Answer Style**: Detailed
- **Example Count**: Two
- **Tone**: Casual

Answer:

Alright, let's get into the nitty-gritty of multithreading versus multiprocessing. These are both ways to run lots of stuff seemingly at once, but they do it in totally different ways, leading to very different results. Think of it like this: multithreading is like having a bunch of chefs all working in the same kitchen, while multiprocessing is like having several kitchens, each with its own chef.

Multithreading: The Shared Kitchen
Imagine a bustling restaurant kitchen. Multiple chefs (threads) are all working on different dishes (tasks) simultaneously within the same kitchen (process). They all share the same ingredients (memory), pots, pans, and ovens (system resources). This sharing is great for speed because they can grab what they need quickly. But, it also means they need to be super careful not to grab the same ingredient at the same time - that's a recipe for disaster (a race condition!). To prevent chaos, they have to use special systems (mutexes, semaphores, etc.) to coordinate who gets what when. Messing this up leads to unpredictable and frustrating bugs.

On a single-core system, the chefs take turns working, but it still seems like they're all cooking at once. On a multi-core system, some chefs might actually cook at the same time, but the operating system decides who gets which stove. In Python, there's a thing called the Global Interpreter Lock (GIL) that limits how much true parallel cooking happens, even on multi-core systems.

Multithreading is perfect for tasks where the chefs spend a lot of time waiting, like waiting for ingredients to arrive (I/O-bound tasks, like waiting for data from a network). A web server handling multiple requests is a classic example: while one chef waits for a customer's order, another can start preparing a different dish.

Multiprocessing: Separate Kitchens
Now picture several separate kitchens (processes), each with its own chef (process) and its own complete set of ingredients and equipment. Each chef works independently, so there's no risk of them stepping on each other's toes. However, if one chef needs something from another kitchen, they have to use a delivery service (inter-process communication, or IPC). This adds a bit of overhead - it takes time to send messages back and forth. But, independence is awesome for avoiding those nasty race conditions.

Multiprocessing shines when the work is all about cooking (CPU-bound tasks), like preparing a complex dish that requires lots of chopping and stirring. Rendering a high-resolution image is a great example: you can split the image into smaller parts, each processed by a separate kitchen (process) simultaneously. This makes the whole process dramatically faster, especially on systems with multiple cores.

In short: Multithreading is fast but requires careful coordination; multiprocessing is slower to set up but avoids many coordination headaches and scales better on multi-core systems. Choose the right tool for the job!

Analysis:

Based on the defined user profile (a student with intermediate expertise) and the selected preferences, the answer is a strong match. The style is detailed, offering clear explanations and covering both multithreading and multiprocessing in depth. Two vivid and accessible examples are included–the shared kitchen and separate kitchens analogies–along with real-world computing examples like web servers and image rendering. The tone is casual and engaging, using friendly language and metaphors that make complex concepts easier to understand. The explanation introduces technical terms like mutexes, GIL, and IPC but explains them in a simple, student-friendly way.

User 5

Profile:

- **Role**: Lecturer
- **Expertise Level**: Beginner
- **Answer Style**: Concise
- **Example Count**: One
- **Tone**: Formal

Answer:

Multithreading and multiprocessing are concurrency techniques differing fundamentally in resource management. Multithreading utilizes a single process with multiple threads sharing memory, enabling faster context switching but risking race conditions. In contrast, multiprocessing employs multiple independent processes, each with its own memory space, preventing race conditions but requiring inter-process communication, introducing overhead. A text editor, for example, might use multithreading for user interface updates and autosaving, while employing multiprocessing for computationally intensive tasks like syntax highlighting.

Analysis:

The answer is a good fit. The style is concise, summarizing the main differences between multithreading and multiprocessing clearly and efficiently. One relevant example is included, as requested, demonstrating how a text editor might use both techniques. The tone is formal and academic, aligning well with the user's role as a lecturer. In terms of knowledge level, the explanation avoids overly technical details while still presenting accurate and essential concepts.

Summary

This test evaluated the effectiveness of SmartChat's personalized response system using a standardized validation prompt across a diverse set of user profiles. Each response was tailored to the user's specific attributes, including role, expertise level, answer style, example count, and tone. The goal was to ensure that every returned response was not only factually correct but also aligned with the user's individual communication and comprehension preferences.

For each user, the system adjusted the structure, detail, tone, and examples in the generated response. Analyses were conducted to assess how closely each answer matched the expected parameters. These analyses confirmed that SmartChat successfully adapts its outputs to meet the expectations of varied user profiles, from beginner students to expert lecturers.

Test Results

User	User Profile	Profile Summary	Result Summary
1	Beginner Student, Concise, No Examples, Neutral Tone	The answer was brief, easy to understand, and omitted examples as requested. It matched the user's beginner level and reading expectations.	Fully Aligned
2	Advanced Lecturer, Detailed, Multiple Examples, Formal Tone	The answer was comprehensive and deeply technical. It included multiple advanced examples, aligned well with a lecturer's needs.	Fully Aligned
3	Advanced Engineer, Concise, One Example, Neutral Tone	The answer was structured, precise, and included one relevant technical example. Tone and complexity matched the profile.	Fully Aligned
4	Intermediate Student, Detailed, Two Examples, Casual Tone	The answer was highly readable, example-rich, and used casual metaphors appropriate for an intermediate student.	Fully Aligned
5	Beginner Lecturer, Concise, One Example, Formal Tone	The answer balanced formality and simplicity well. It was concise and included one relevant use case, appropriate for the profile.	Fully Aligned

Figure 25 - Results from validation prompt per user.

Conclusion

The test demonstrates that SmartChat's personalization logic is functioning correctly. Each user's response adhered closely to their declared profile settings, as confirmed by the detailed analysis. The system accurately adjusted the tone, level of detail, and structure of each response to suit different user roles and expectations.

SmartChat can therefore be considered reliable in delivering contextually appropriate and preference-aligned responses based on user-specific profiles.

8.8. Project Scope Validation

Overview

This test validates whether the SmartChat system correctly identifies and processes only **in-scope questions** related to the software engineering domain, while appropriately rejecting or flagging **out-of-scope queries** that fall outside the intended subject area. This ensures that the system maintains topic relevance and supports users in a focused and professional manner.

Tested Scenarios

In-Scope Questions

All of the following software engineering-related prompts were **accepted and responded to appropriately**, demonstrating the system's alignment with its intended knowledge domain:

Prompt	Scope Category	Purpose
"What are the main differences between monolithic and microservices architectures?"	System Design	Validates architectural knowledge
"How does the Agile development lifecycle work in a real project?"	SDLC Methodologies	Confirms lifecycle understanding
"What are the time and space complexities of different sorting algorithms?"	Algorithms	Tests algorithmic reasoning
"How can I prevent race conditions in a multi-threaded Node.js application?"	Concurrency	Assesses thread control and safety
"What is the purpose of CI/CD, and how can I set it up with GitHub Actions?"	DevOps	Verifies toolchain knowledge and deployment understanding

Figure 26 - Software engineering–related prompts used for validation testing

Each of these questions received a relevant, well-formulated response, consistent with the user's configured preferences.

Out-of-Scope Questions

The following unrelated questions were **not processed** by the system, demonstrating appropriate scope enforcement:

Prompt	Reason for Rejection
"What are the health benefits of the Mediterranean diet?"	Non-technical, health topic
"How do I apply for a tourist visa to Canada?"	Administrative/legal, unrelated
"What is the symbolism in Shakespeare's Macbeth?"	Literature, out of domain
"What's the best fertilizer for growing tomatoes?"	Agricultural, non-SE topic
"Can you help me write a breakup letter?"	Personal advice, not technical

Figure 27 - Out-of-scope prompts used for validation testing

The system either returned a predefined scope-related message or rejected these prompts without generating an AI answer.

Conclusion

The test results confirm that **SmartChat correctly distinguishes between in-scope and out-of-scope queries**. The system accepted all relevant software engineering questions and rejected those unrelated to the domain, thereby validating the scope enforcement logic.

This behavior ensures the platform maintains **professional focus and content integrity**, aligning with its intended use in the software engineering field.

8.9. Test Results & Conclusion

The validation process was carried out through a series of controlled test scenarios, usability evaluations, and functional verifications. Each module of the system–such as prompt processing, multi-platform integration, answer merging, personalization, and feedback analysis–was assessed independently and in combination to verify end-to-end performance.

In addition, user-focused validation scenarios were conducted to evaluate the effectiveness of SmartChat in delivering accurate, personalized, and coherent responses across various roles and expertise levels. These scenarios tested the system's ability to correctly enrich prompts, select the most suitable response, and adapt based on user feedback.

The results presented demonstrate the system's reliability, alignment with the project scope.

Test Name	Result Summary	Conclusion
Backend Unit Tests	All 32 unit tests passed using Jest.	Core logic is stable, with valid responses and proper error handling.
Conversation Flow Integration	Chat flow triggers appropriate logic (create_chat or send_message) and returns valid responses.	Full user-to-backend-to-UI message cycle works as expected.
Decision-Making Logic	System auto-selected a single platform per prompt and maintained continuity across messages.	The selection algorithm is reliable and transparent to the user.
Feedback Adaptation	Responses adjusted appropriately after structured feedback.	System adapts future answers based on detailed user preferences and feedback.
Feedback Free-Text Analysis (NLP)	NLP model accurately classified qualitative feedback using a confidence threshold.	The classifier is effective for extracting useful insights from free text.
Validation Prompt per User	Personalized responses generated based on user role, expertise, tone, and preferences.	The system delivers appropriately tailored answers per user profile.
Project Scope Validation	SE-related prompts were accepted; non-relevant prompts were filtered out.	SmartChat enforces domain boundaries, ensuring it remains focused on software engineering use cases.

Figure 28 - Summary of test results and conclusions

The test results confirm that SmartChat behaves as expected across its key modules. This behavior ensures accurate response generation, meaningful user adaptation, and reliable operation under varying conditions.

9. References

1. Multi-platform Chatbot Modeling and Deployment with the Jarvis Framework, SpringerLink. https://doi.org/10.1007/978-3-030-21290-2_12

2. Google DialogFlow Documentation.

   https://cloud.google.com/dialogflow/

3. OpenAI ChatGPT API Documentation.

   https://platform.openai.com/docs/

4. Prompt Engineering and LLMs with Langchain https://www.pinecone.io/learn/series/langchain/langchain-prompt-templates/

5. Prompt Engineering Best Practices: Tips, Tricks, and Tools https://www.digitalocean.com/resources/articles/prompt-engineering-best-practices

6. Best practices for prompt engineering with the OpenAI API https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-the-openai-api

7. Gaskill, N., & Osterman, C. (2025, August 24). *From fragmentation to integration: How AI-powered site networks are powering clinical research*. Tempus. https://www.tempus.com/resources/content/articles/from-fragmentation-to-integration-how-ai-powered-site-networks-are-powering-clinical-research

8. Tancredi, L. (2024, February 21). *AI strategies series: Inconsistent and outdated responses*. FactSet. https://insight.factset.com/ai-strategies-series-inconsistent-and-outdated-responses

9. Applause. (2024, August 23). *Evaluate AI model quality with prompt and response grading*. Applause. https://www.applause.com/blog/evaluate-ai-model-quality-with-prompt-and-response-grading/

10. Jha, A. (2024, February 21). *Building better LLMs: A guide to feedback-driven optimisation*. Medium. https://medium.com/@aartijha96/feedback-driven-ai-the-key-to-building-better-llms-627518e364cc

10. Appendices

Appendix 1 -

decision-making-test1.log

decision-making-test2.log

Appendix 2 -

feedback-nlp-test-results.txt

תקציר

SmartChat היא פלטפורמת שיחה המופעלת על ידי בינה מלאכותית עם ממשק מבוסס צ'אט, שנועדה לספק תגובות מדויקות, רלוונטיות ומודעות להקשר, הממנפות את היסטוריית השיחה המלאה ומתאימות להעדפות ספציפיות למשתמש בתחום הנדסת התוכנה. המערכת מעריכה את התפוקות של פלטפורמות בינה מלאכותית מרובות, בוחרת את התגובה המתאימה ביותר ומשפרת אותה עוד יותר.

עם קבלת שאילתת משתמש, SmartChat מבצעת הנדסת הנחיות מתקדמת, תוך התחשבות בהעדפות המשתמש, באינטראקציות קודמות ובהיסטוריה של משוב. ההנחיה המעובדת נשלחת בו זמנית לשלוש פלטפורמות בינה מלאכותית שונות. לאחר מכן המערכת מבצעת הערכה השוואתית כדי לקבוע את התגובה המתאימה ביותר, אשר לאחר מכן משפרת ומוצגת למשתמש.

כדי לשפר באופן מתמיד את הביצועים ואת שביעות רצון המשתמשים, SmartChat מזמינה משתמשים לספק משוב לאחר כל אינטראקציה. משוב זה מנותח ומשמש לתכנון בחירת תגובות עתידיות ואסטרטגיות התאמה אישית.

הכרת תודה

ברצוננו להביע את הערכתנו ותודתנו הכנה והעמוקה למר יצחק נודלר, מרצה בכיר במחלקה להנדסת תוכנה במכללת שנקר, על תמיכתו הבלתי מתפשרת, ההכוונה המעמיקה והעידוד המתמשך לאורך כל שלבי הפרויקט.
ידיעותיו הנרחבות, תשומת הלב לפרטים ומשובו המחושב תרמו רבות להצלחת עבודתנו. ברגעים משמעותיים רבים, עצותיו סייעו לנו להתגבר על אתגרים.
אנו אסירי תודה במיוחד על נכונותו להקדיש מזמנו ותשומת לבו מעל ומעבר למצופה, תמיד בסבלנות, מקצועיות ואכפתיות אמיתית. שיתוף הפעולה המוצלח והליווי האישי שקיבלנו ממר נודלר הותירו חותם משמעותי במסע האקדמי שלנו. תודה מכל הלב.

SHENKAR שנקר
ENGINEERING. DESIGN. ART. הנדסה. עיצוב. אמנות.
The Pernick Faculty of Engineering. הפקולטה להנדסה ע"ש פרניק

SmartChat

הנדסת תוכנה

ספר פרויקט גמר לתואר ראשון בוגר במדעים B.Sc

מאת :

אליה סמרי אטיאס

אסף צפריר

נוה מימון

מנחה :

מר נודלר יצחק

31/08/2025

עבודה זו הוגשה כחלק מהדרישות לקבלת תואר ראשון (B.Sc.) בהנדסת תוכנה