PROJECT REPORT FOR PRECISELY

(AN INTELLIGENT REQUIREMENT GATHERING BOT)

1. Abstract

Requirement Engineering (RE) is a critical phase in the software development lifecycle, yet it remains prone to human error, ambiguity, and misclassification. Traditional manual elicitation methods often result in vague specifications that lead to project failure or scope creep. This project introduces **"Precisely,"** an intelligent chatbot designed to automate and enhance the requirement gathering process.

The system leverages a multi-model Natural Language Processing (NLP) pipeline to interact with users in natural language. It utilizes four fine-tuned Transformer models: **DistilBERT** for intent recognition, **DeBERTa-v3-Base** for classifying requirements as Functional (FR) or Non-Functional (NFR) and for predicting Quality Attributes, and **T5-Base** for detecting and correcting ambiguity.

The solution is delivered via a responsive web-based dashboard that guides users through scope definition, real-time requirement analysis, and automatic specification generation. The final output is a structured, standard-compliant PDF report, significantly reducing the time and effort required to produce high-quality Software Requirement Specifications (SRS).

---

2. Introduction

2.1 Overview

The quality of a software product is heavily dependent on the quality of its requirements. "Precisely" is developed to bridge the gap between stakeholder intent and technical specification. Unlike static forms, Precisely acts as a virtual requirement analyst, engaging users in a conversational workflow to elicit, validate, and document requirements dynamically.

2.2 System Workflow (Web Version)

The web application serves as the primary interface, built using **FastAPI** for the backend and **HTML/CSS/JavaScript** for the frontend. The workflow proceeds as follows:

1. **Project Initialization:** The user creates a project via a modal interface, defining the project name and description.
2. **Interactive Scope Wizard:** Before gathering requirements, the bot initiates a state-machine-driven interview to define the **Project Scope**. It asks specifically about:

    - Primary Users (Stakeholders)
    - Main Features (Functional boundaries)
    - Exclusions (What is out of scope)
    - Technical & Business Constraints
3. **Real-Time Classification & Validation:** As the user types requirements into the chat interface, the backend pipeline processes the text through four distinct AI models simultaneously.
    - It determines if the input is a requirement or chitchat.
    - It classifies the requirement (FR vs. NFR) and identifies specific quality attributes (e.g., Security, Usability).
    - It checks for semantic alignment with the defined scope using S-BERT.
    - It detects ambiguity and suggests corrected, specific phrasing.
4. **Dynamic Dashboard:** The interface updates in real-time. Functional requirements, Non-Functional requirements, and Constraints are automatically sorted into categorized sidebars.
5. **Report Generation:** Upon completion (triggered by the `exit` command), the system compiles all data into a comprehensive HTML report and offers a one-click PDF export function.

---

3. Problem Statement

In modern software engineering, the process of gathering requirements faces several critical challenges:

1. **Ambiguity in Natural Language:** Stakeholders often use vague terms like "fast," "user-friendly," or "robust," which are not testable or measurable. Manual review to catch these ambiguities is time-consuming.
2. **Misclassification:** Distinguishing between Functional Requirements (FR) and Non-Functional Requirements (NFR) is often confusing for non-technical stakeholders, leading to poorly structured documentation.
3. **Scope Creep:** Without constant validation against defined boundaries, new requirements often drift outside the agreed-upon project scope, leading to budget overruns and delays.
4. **Lack of Standardization:** Requirements gathered via disparate emails or meetings lack a unified format (e.g., IEEE 830), making it difficult for developers to interpret them.

"Precisely" addresses these problems by using fine-tuned Models to enforce clarity, automatically categorize inputs, and strictly validate scope in real-time.

---

4. Hardware and Software Requirements

4.1 Hardware Requirements

To ensure the efficient execution of the four Transformer-based models, the following hardware specifications are recommended:

- **Processor (CPU):** Intel Core i5 (10th Gen) or AMD Ryzen 5 equivalent (Minimum).
- **Memory (RAM):** 16 GB recommended (Minimum 8 GB required to load models into memory).
- **Graphics Processing Unit (GPU):** NVIDIA GPU with at least 4GB VRAM (Optional but recommended for faster inference via CUDA; the system can run on CPU).
- **Storage:** 10 GB free space (to store model weights, dependencies, and project data).
- **Network:** Stable internet connection for initial library downloads (once set up, the system runs locally).

4.2 Software Requirements

The project is built upon a robust open-source stack:

- **Operating System:** Windows 10/11, Linux (Ubuntu 20.04+), or macOS.
- **Programming Language:** Python 3.10 or higher.
- **Machine Learning Frameworks:**
  - **PyTorch:** Core framework for model loading and inference.
  - **Hugging Face Transformers:** For utilizing the pre-trained and fine-tuned models.
- **Backend Framework:**
  - **FastAPI:** For creating the high-performance REST API.
  - **Uvicorn:** ASGI server for serving the application.
- **Frontend Technologies:**
  - **HTML5 / CSS3:** For the dashboard structure and styling.
  - **Vanilla JavaScript (ES6+):** For asynchronous API calls and DOM manipulation.
- **AI Models & Libraries:**
  - **Sentence-Transformers (S-BERT):** `all-MiniLM-L6-v2` for semantic scope similarity.
  - **Spacy (`en_core_web_sm`):** For linguistic parsing and verb extraction.
  - **xhtml2pdf:** For generating downloadable PDF reports.

4.3 Model Specifications (The "Brain")

The intelligence of the system relies on four specifically fine-tuned models:

1. **Ambiguity Detector:**
   - **Base Model:** T5-Base (Text-to-Text Transfer Transformer).
   - **Function:** Rewrites vague inputs into specific, measurable statements.
2. **FR/NFR Classifier:**
   - **Base Model:** DeBERTa-v3-Base.
   - **Function:** Binary classification to distinguish "Functional" vs. "Non-Functional" inputs.
3. **Quality Attribute Classifier:**

- **Base Model:** DeBERTa-v3-Base.
- **Function:** Multi-class classification to identify specific attributes based on the ISO/IEC 25010 standard.
- **Labels:**
  - `__label__US`: Usability
  - `__label__SE`: Security
  - `__label__PE`: Performance
  - `__label__RA`: Reliability/Availability
  - `__label__SC`: Scalability
  - `__label__PO`: Portability
  - `__label__LE`: Legal and Compliance
  - `__label__MA`: Maintainability
  - `__label__CO`: Compatibility
  - `__label__AV`: Availability

4. **Intent Classifier:**
   - **Base Model:** DistilBERT (Distilled version of BERT).
   - **Function:** Determines the user's interaction goal.
   - **Intents:** `requirement` (processing needed), `finish_gathering`, `greeting`, `chitchat`, `system_setup`.

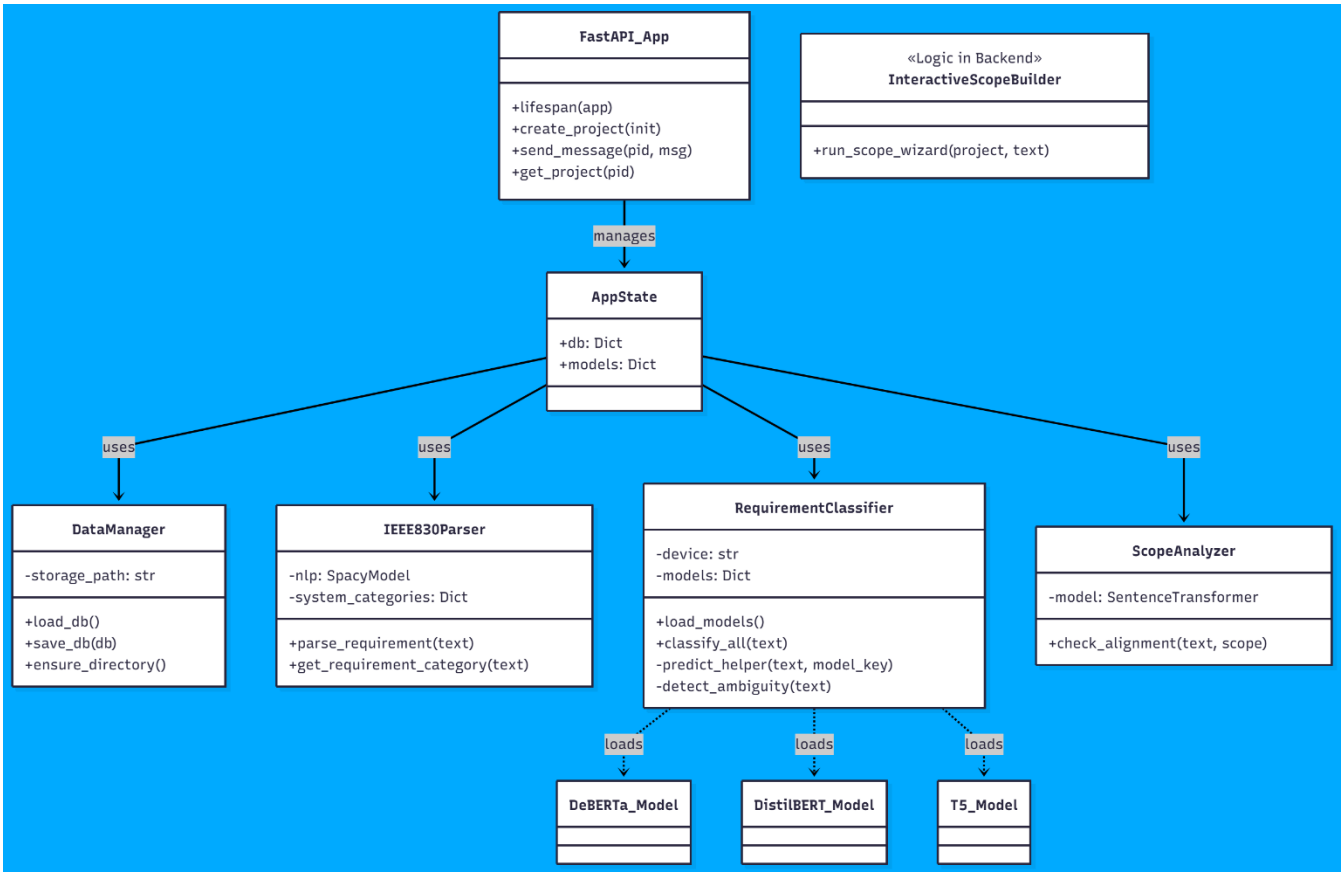
---

5. Diagrams (Structural and Behavioural)

5.1 Structural Diagrams:

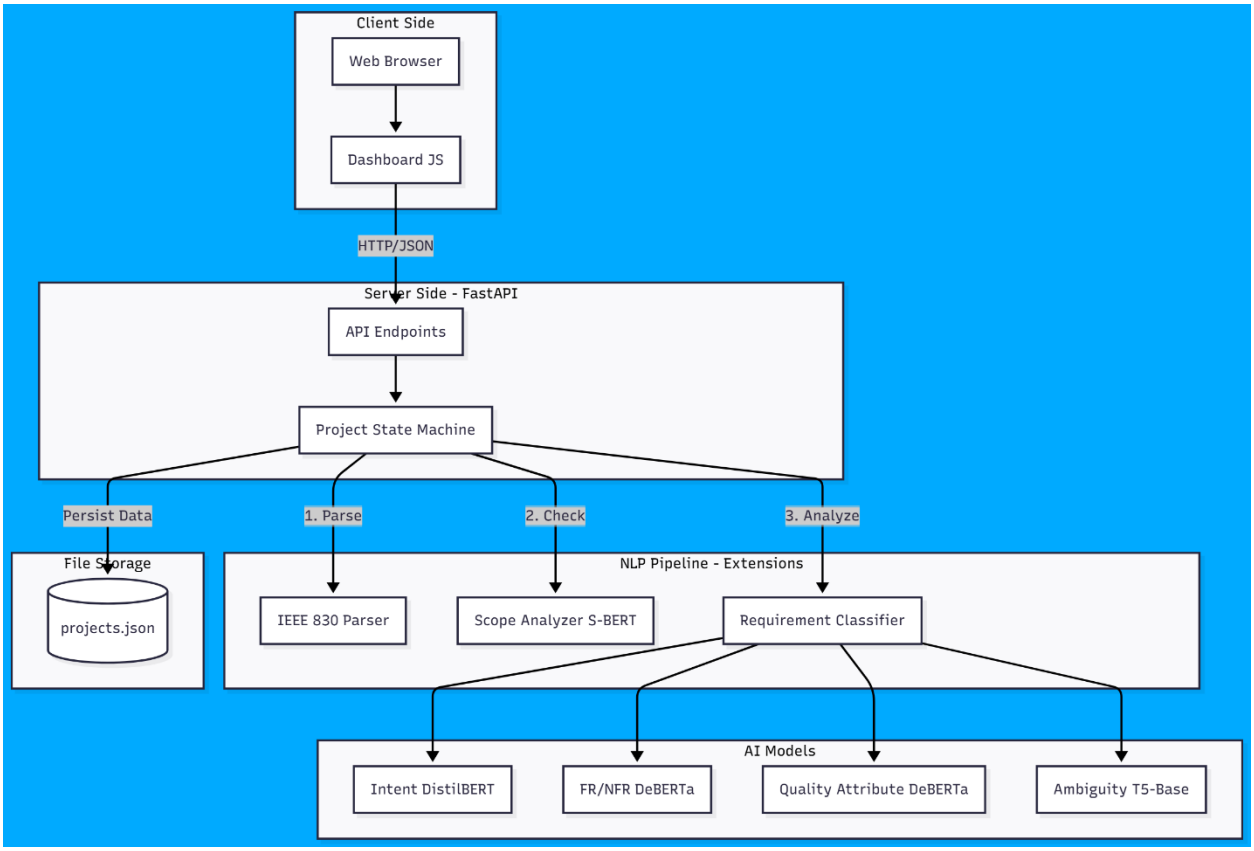
5.1.1 Class Diagram:

This diagram represents the object-oriented structure of the backend logic, detailing the relationships between the main application state, data management, and the specific analysis classes defined in `extensions.py` and `backend.py`.



5.1.2 Component Diagram:

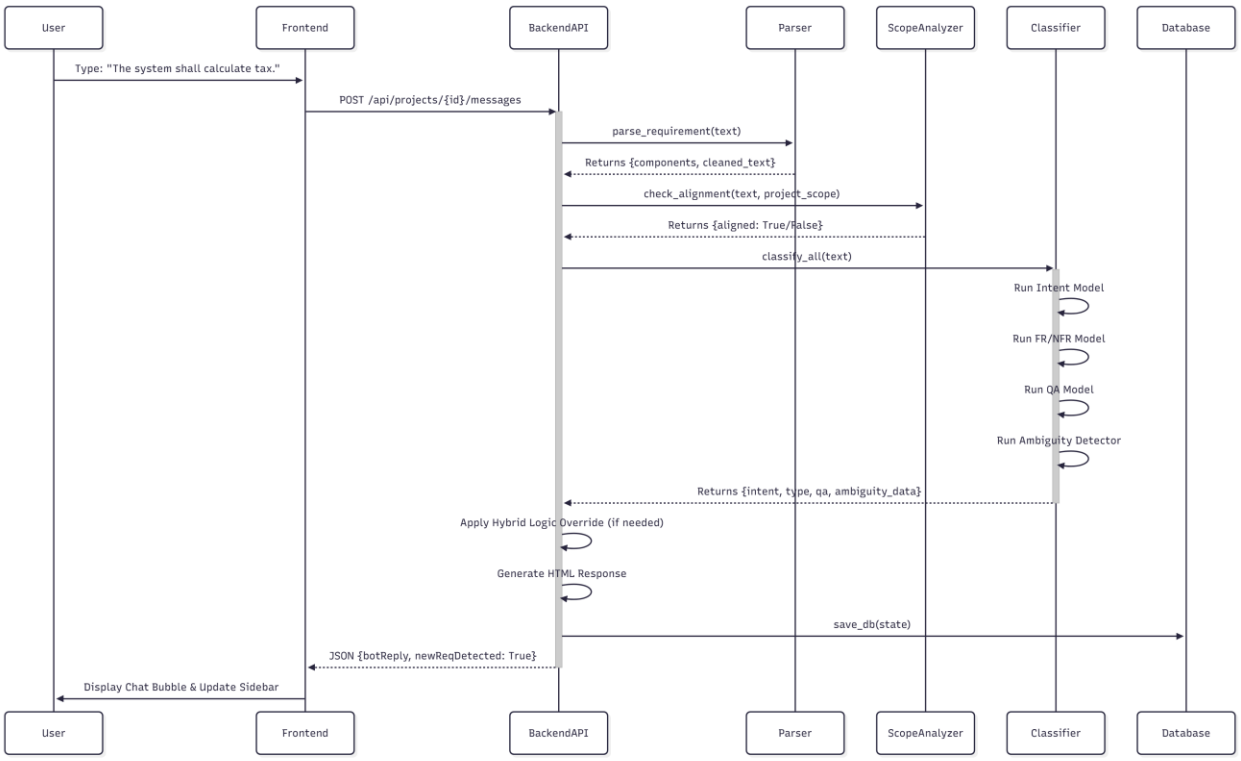
This diagram illustrates the high-level architecture of "Precisely," showing how the frontend interacts with the backend API and how the backend integrates with various AI models and storage.



5.2 Behavioral Diagrams:

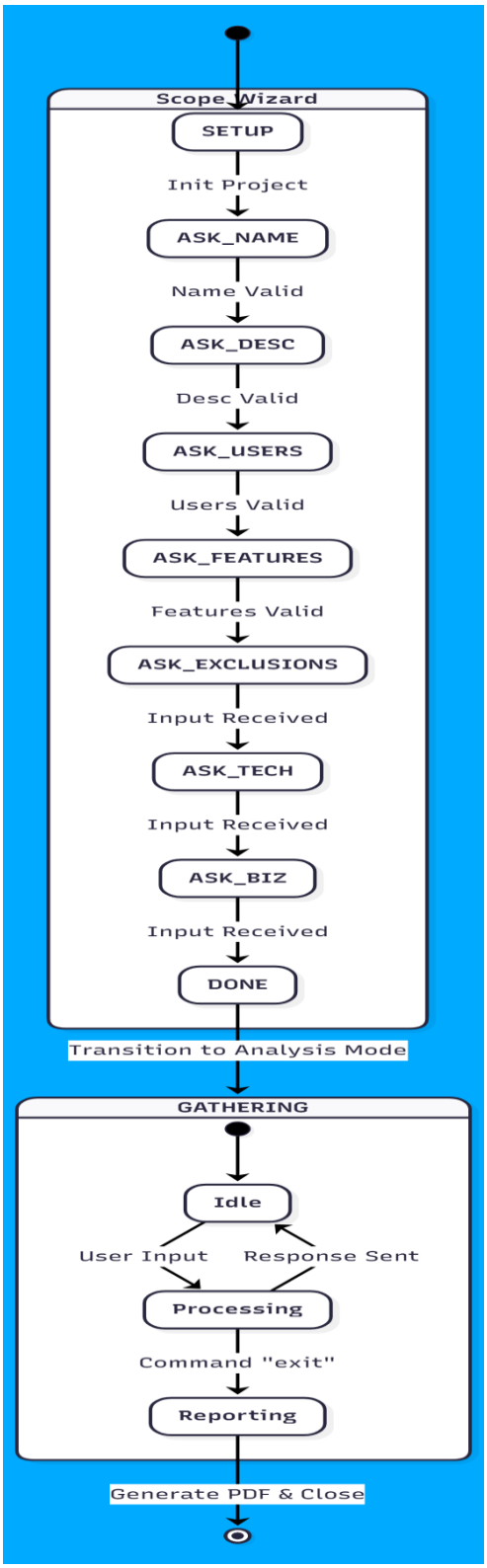
5.2.1 Sequence Diagram (Requirement Analysis Flow)

This sequence diagram details the exact flow of data when a user types a requirement into the chat. It visualizes the pipeline processing order: Parsing -> Scope Check -> ML Classification -> Response Generation.



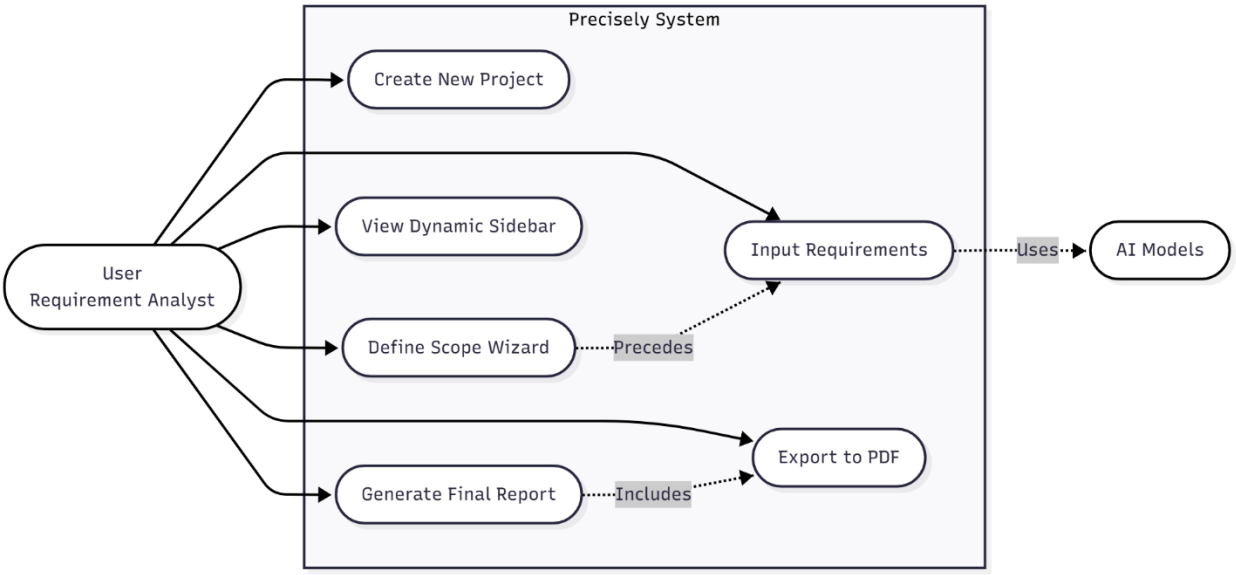
5.2.2 State Machine Diagram (Scope Wizard)

This diagram represents the internal logic of the Scope Definition Wizard. It shows how the bot transitions between different questions (Name -> Description -> Users...) based on the user's input until the scope is fully defined.



5.2.3 Use Case Diagram

This diagram provides a high-level view of the user's interactions with the "Precisely" system, distinguishing between project management actions and the core requirement engineering tasks.



6. How it Works

The "Precisely" system operates on a client-server architecture where the frontend manages user interaction and the backend orchestrates the complex logic of state management, natural language processing, and data persistence. The system's operation can be divided into four distinct phases:

6.1 Phase 1: Session Initialization & Project Creation

The workflow begins when the user accesses the web dashboard.

1. **Frontend Action:** The user clicks "New Project" and submits a project name (e.g., "E-Commerce App") and description via a modal form.
2. **API Request:** A `POST` request is sent to the `/api/projects` endpoint.
3. **Backend Processing:**
    - The backend initializes a new session entry in the in-memory database (persisted to `projects.json`).
    - A **Project State Object** is created with the status set to `SETUP` and the wizard step set to `ASK_USERS`.
    - The initial system prompt is generated and returned to the chat interface.

6.2 Phase 2: Interactive Scope Definition (The Wizard)

Before gathering requirements, the system enforces a boundary definition phase using a **Finite State Machine (FSM)**.

- **Logic:** The backend tracks the current `wizard_step` (e.g., `ASK_FEATURES`, `ASK_EXCLUSIONS`).
- **Interaction:**
    1. The bot asks a specific question (e.g., "What are the primary users?").
    2. The user responds in natural language.
    3. **Smart Parsing:** The backend uses a custom `parse_smart_list` function to intelligently split comma-separated values while respecting quotation marks (e.g., treating `"Linux, Windows"` as a single item).
    4. **State Transition:** The data is saved to the project's `scope` dictionary, and the state advances to the next step.
- **Completion:** Once all constraints are defined, the project status shifts from `SETUP` to `GATHERING`, enabling the AI analysis pipeline.

6.3 Phase 3: The Intelligent NLP Pipeline (Core Loop)

This is the heart of the system. Every time the user enters a requirement, the backend processes the text through a sequential pipeline:

Step A: Linguistic Parsing (Spacy) The `IEEE830Parser` uses the **Spacy** library to perform Part-of-Speech (POS) tagging. It extracts the subject, action verb, and object to validate if the sentence is grammatically complete (e.g., ensuring "The system" [Subject] "shall calculate" [Verb] "tax" [Object] exists).

Step B: Scope Alignment (Sentence-Transformers) The `ScopeAnalyzer` converts the user's input into a high-dimensional vector using **S-BERT**. It calculates the **Cosine Similarity** between this vector and the vectors of the previously defined "Exclusions." If similarity is high (> 0.45), the system flags the requirement as "Out of Scope."

Step C: Multi-Model Classification The text is simultaneously fed into four fine-tuned Transformer models:

1. **Intent Classifier (DistilBERT):** Determines if the input is a Functional or Non-Functional requirement.
2. **FR/NFR Classifier (DeBERTa-v3):** Performs a specialized binary classification to tag the requirement as `FR` (Functional) or `NFR` (Non-Functional).
3. **Quality Attribute Classifier (DeBERTa-v3):** Predicts specific attributes like *Security*, *Usability*, or *Performance*.
4. **Ambiguity Detector (T5-Base):** Generates a rewritten version of the requirement if it detects vague terms (e.g., changing "process quickly" to a specific, measurable statement).

Step D: Hybrid Logic Override To maximize accuracy, a heuristic logic layer runs after the AI models. It scans for strong functional verbs (e.g., *calculate, compute, store*). If found, it overrides any potential AI misclassification, ensuring that action-oriented requirements are correctly tagged as "Functional" even if they contain NFR-sounding keywords like "password."

6.4 Phase 4: Real-Time Synchronization & Visualization

The system ensures the user interface remains consistent with the backend state without full page reloads.

- **Dynamic Sidebars:** As requirements are classified, the frontend receives a JSON response containing the requirement's `type`. JavaScript logic automatically injects the requirement card into the correct sidebar column (Functional vs. Non-Functional).
- **Constraint Visualization:** Constraints defined during the Wizard phase are split and rendered individually in the "Constraints" tab, giving users a constant view of project limitations.

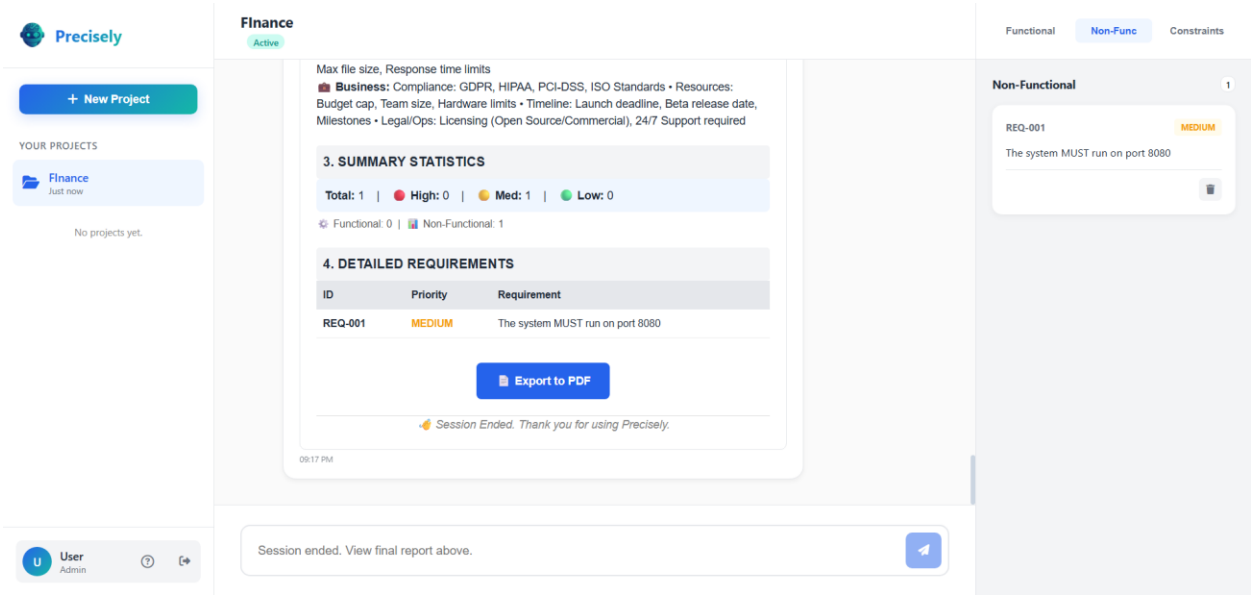
6.5 Phase 5: Automated Reporting

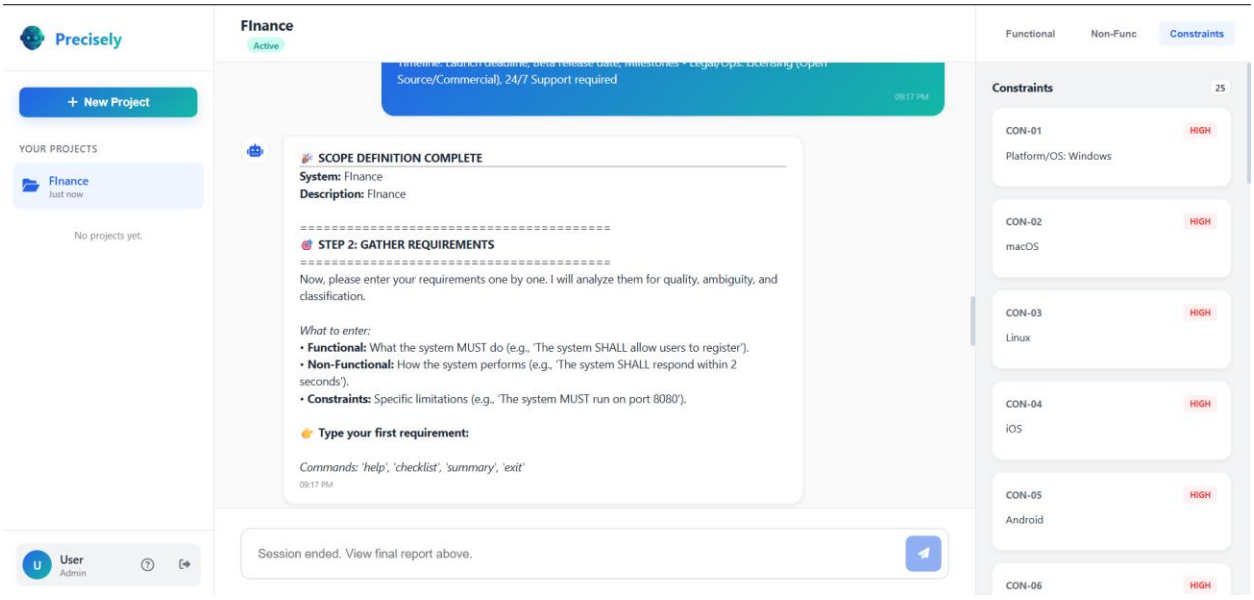
Upon receiving the `exit` command:

1. The system freezes the chat input to prevent further modification.

2. The backend compiles a statistical summary (Requirements by Priority, FR vs. NFR counts).
3. An HTML report is generated and rendered directly in the chat window.
4. The **Export Module** uses `xhtml2pdf` to convert this HTML view into a downloadable PDF file, providing a standardized SRS document ready for stakeholders.

7. Screen shots of output screens





Precisely: Comprehensive User Guide

Welcome to **Precisely**, your AI-powered Requirement Engineering assistant. This guide covers fundamental Software Engineering concepts, how to navigate the dashboard, and the step-by-step workflow to generate professional specifications.

1. Core Concepts: Understanding Requirements

Before using the bot, it is essential to understand the three main categories of software requirements. Precisely automatically classifies your input into these buckets.

Category	Definition	Real-World Examples
FUNCTIONAL (FR)	**"What the system DOES."** Describes a specific behavior, function, or calculation the system must perform. If this is missing, the system is incomplete.	• "The system shall calculate tax." • "Users shall log in via Email." • "The app shall send notifications."
NON-FUNCTIONAL (NFR)	**"How the system BEHAVES."** Describes quality attributes such as Performance, Security, Reliability, and Usability.	• "The system shall load in < 2 seconds." • "Data must be encrypted (AES-256)." • "The interface should be user-friendly."
CONSTRAINT	**"The LIMITATIONS."** Technical or business restrictions that limit design choices. These are usually non-negotiable.	• "Must run on Linux servers." • "Development budget is $50k." • "Must comply with GDPR."

8. References

1. **Sanh, V., et al.** "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter." *arXiv preprint arXiv:1910.01108*, 2019. (Used for Intent Classification).

2. **He, P., et al.** "DeBERTa: Decoding-enhanced BERT with Disentangled Attention." *arXiv preprint arXiv:2006.03654*, 2020. (Used for FR/NFR and Quality Attribute Classification).

3. **Raffel, C., et al.** "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer." *Journal of Machine Learning Research*, 2020. (T5 Model used for Ambiguity Detection).

4. **Reimers, N., & Gurevych, I.** "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks." *EMNLP*, 2019. (Used for Scope Analysis).

5. **Hugging Face.** "Transformers Documentation." Available: https://huggingface.co/docs/transformers/. (Library used for model implementation).

9. Appendix

Due to the extensive nature of the codebase, which includes fine-tuned model weights, backend logic, and frontend assets, the full source code is not included in this document.

The complete project repository, including setup instructions and documentation, is published on GitHub and can be accessed at the following link:

GitHub Repository: https://github.com/ahmadali-2k06/RequirementBot