UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI
UNDERGRADUATED STUDENT

BACHELOR THESIS

by

Nguyen Tuan Anh - BA11-007
Information and Communication Technology

Thesis Title:

A middleware framework to support scientific generation using an AI approach

Supervisor:

MSc. Le Nhu Chu Hiep

Hanoi, 2025

Acknowledgement

First and foremost, I would like to express my deepest and most sincere gratitude to Ms. Le Nhu Chu Hiep, my supervisor, who has guide and supported me throughout my internship. Not only knowledge but also the lifestyle, thanks to him, I grow up everyday and overcome all challenges. Under his guidance, I gradually improved myself and achieved remarkable academic achievements. I am immensely grateful for everything he has done for me.

I would also thanks to the University of Science and Technology of Hanoi (USTH) - the place that has become my youth. USTH has educated and trained me, helping me grow and develop every day. The memories and lessons learned here will always be valuable assets in my life.

Especially, I want to express my heartfelt thanks to my love, Khanh Linh. You have always been by my side, supporting and encouraging me throughout the project. I remember each delicious meals you cooked for me and your words of encouragement have given me the huge confidence and motivation to complete all my work. You are my most emotional support, I am deeply thankful for that.

I also want to send my deepest gratitude to my mother and grandmother. Thanks to their guidance and teachings over the past 20 years, I have been able to grow up to be who I am today. Their love and sacrifices have been a great source of motivation for me to overcome all challenges.

Finally, I want to thank my late father, who passed away not long ago. Although he is no longer with us, his teachings and love continue to be a source of encouragement and strength for me to keep striving.

Declaration

My name is Nguyen Tuan Anh, ID: BA11-007, a third-year student majoring in Information and Communication Technology at the University of Science and Technology of Hanoi (USTH). My supervisor is Ms. Le Nhu Chu Hiep.

I hereby declare that the entire content presented in this thesis, titled "A middleware framework to support scientific generation using an AI approach" is the result of my own research and study. All data presented in this thesis is truthful and reflects the actual results from practical implementation.

All quoted information follows intellectual property rules, and the references are clearly listed. I take full responsibility for all the contents written in this thesis.

Hanoi, July 2025

Declarant

NGUYEN TUAN ANH

Contents

Abbreviations

AI	Artificial Intelligence
AI PELaX	Artificial Intelligence for Processing and Exploring LaTeX
API	Application Programming Interface
HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notation
LaTeX	Lamport's TeX (Typesetting System)
LLM	Large Language Model
ML	Machine Learning
NLP	Natural Language Processing
NFR	Non-functional Requirement
PNG	Portable Network Graphics
REST	Representational State Transfer
TikZ	A LaTeX package for creating graphics programmatically
UI	User Interface

List of Figures

List of Tables

Abstract

This thesis presents the development of a mobile application named AI PELaX, which integrates an AI model with a LaTeX rendering middleware to assist users—especially students and researchers—in solving scientific problems and visualizing mathematical content directly on their phones. The app is built using Flutter for cross-platform mobile development and communicates with a Flask-based backend that handles AI prompt processing and LaTeX code rendering.

The core idea of the project is to allow users to send a natural-language question related to scientific or mathematical content. This question is then processed by an AI model (such as OpenR1) to generate a response that may contain LaTeX expressions or even TikZ diagrams. These LaTeX components are extracted by the middleware, compiled into images using TeXLive, and sent back to the mobile app for immediate display within the chat interface.

Compared to traditional AI systems like ChatGPT or Claude, which only return TikZ Code as plain text, this system adds a unique layer by automatically converting TikZ code into images, improving readability and learning efficiency. The system was successfully implemented with key features such as prompt input, LaTeX and TikZ code extraction, server-side rendering, and real-time display in the app.

Several challenges were faced during the development process, including rendering delays and data loss on app restart, which are discussed in the report. Future work includes adding offline history, supporting more AI models, and improving performance.

This project demonstrates a practical and scalable way to integrate AI, LaTeX rendering, and mobile interfaces, contributing to the growing need for smarter educational tools in the digital era.

Keywords: AI-generated content; LaTeX rendering; TikZ visualization; mobile application; Flutter; Flask backend; prompt engineering; educational technology; scientific problem solving

Chapter 1: Introduction

Today, AI is becoming an important part of many areas in life, including education and science. AI tools and apps are now used more and more, giving people new ways to find and work with information. But to use these tools well, we also need the right solutions for real problems. This chapter will start by giving an overview of the background, the problems, and what this project is about.

1.1 Problem Statement

In nowadays context, the demands of finding and synthesizing scientific information from various online sources are increasing faster and faster, especially in education, research and technology development fields. The traditional searching tools such as Google Search or academic document aggregation platforms, although helping users access giant knowledge, still exist many disadvantages about aggregation information content aspect, intuitive presentation and interactive ability with the received information[1].

The strong development of Artificial Intelligence (AI) has opened up a new approach, in which Large Language Models (LLM) such as ChatGPT, Claude, or Gemini can generate structured scientific content in text base, mathematical formulas, graphs, or charts, etc. However, a big arisen problem is the large amount of this intuitive content - especially TikZ code in LaTeX - regularly returned in the text-based (plain text) form[2]. Users who want to observe this content in intuitive visualization from TikZ must compile it manually using a professional LaTeX system like Overleaf or desktop LaTeX tools. This problem causes many difficulties for people who are not familiar with this tool or do not have a suitable configuration environment.

Also, creating good questions (called "prompts") for AI is not easy. Users need to know how to write clear and correct prompts to get useful and well-formatted answers. If the prompt is not good, the AI might give results that are wrong or not what the user wants. This makes using AI for studying and doing research harder for most normal users.

Because of this problem, this project aims to build a middleware system. This system will take TikZ code from AI answers and turn it into images automatically. Users do not need to do anything manually. They can see scientific diagrams right away. This helps make a tool that supports learning, research, and writing scientific documents better and easier—especially on mobile devices.

1.2 Objectives and Scope

The main goal of this project is to create a mobile application that helps users work with scientific content generated by AI in an easy and visual way. This app will allow users to send questions to AI, receive answers with diagrams or formulas, and view them right away without needing to do anything complex.

To do this, the project builds a middleware system. This system takes LaTeX TikZ code from the AI's answer and automatically turns it into an image. The user does not need to copy or compile anything.

Besides that, the app will also:

- Help users create better prompts using a clear and friendly interface (such as drag-and-drop or form-based input).
- Support different AI models like ChatGPT, Claude, or Gemini through API.
- Make it easier for students, researchers, or anyone to use AI in learning, writing, and exploring scientific ideas.

The scope of the project focuses on:

- Mobile devices (Android) using Flutter framework.
- TikZ diagrams and math content generated by AI.
- User interface design for ease of use, especially for people without technical skills in LaTeX.

This project does not cover other types of AI-generated content like videos, full academic papers, or advanced LaTeX systems outside TikZ. It also does not train any AI model itself but uses existing models through API.

1.3 Report Structure

To help readers easily follow the content and development process of the project, this thesis is divided into six main chapters. Each chapter plays an important role in presenting different parts of the work:

- **Chapter 1: Introduction**
  Provides an overview of the context and motivation behind the project, stating the main objectives and scope.

- **Chapter 2: Requirement Analysis**
  Analyses the detailed system requirements, including non-functional requirements, user needs, and technical constraints.

- **Chapter 3: System Design**
  Analyses the general pipeline overview of the system and provides related system architecture and system implementation including the overall approach and the tools employed for implementation.

- **Chapter 4: Results and Discussions**
  Presents the results obtained from the project, analyses their significance, and compares them with existing solutions.

- **Chapter 5: Conclusion and Future Work**
  Summarizes the key findings and achievements of the project, discusses limitations, and suggests directions for future research and improvements.

Chapter 2: Requirement Analysis

To clearly define the requirements and goals of the system, and ensure that they are fully analyzed and well understood —guiding the project in the right direction and addressing real-world needs —this chapter provides a detailed description of the requirements for implementing the system.

2.1 System Requirements

In the complete system, the Artificial Intelligence for Processing and Exploring LaTeX (AI PELaX) is designed to serve as an intelligent platform that supports users—especially students and researchers—in generating, viewing, and interacting with AI-produced scientific materials. These materials may include mathematical expressions, structured text, and TikZ-based technical diagrams embedded in LaTeX.

However, within the scope of this project, the focus is narrowed to a core function: *automatically identifying and extracting TikZ code from AI responses, rendering it into images through a middleware backend, and displaying the results directly inside a mobile application*. By treating all AI output as text-based content and selectively processing LaTeX code, the system removes the need for users to interact with complex LaTeX environments or external editors like Overleaf.

The application is designed to run smoothly on Android devices using the Flutter framework, while the middleware component is built using Flask and operates locally or via a private server connection. This architecture helps minimize the need for constant internet connectivity and reduces dependency on third-party platforms. It also ensures that users can access and view scientific diagrams in real-time, directly in the app, without extra steps or technical setup.

The system's intuitive interface is especially designed for users without advanced LaTeX knowledge, offering features like form-based prompt creation and AI integration via APIs. This makes it a lightweight, portable, and practical solution for learning and research tasks involving scientific visualization.

2.2 Non-Functional Requirements

In addition to the functions that can be directly interacted with on the system, non-functional requirements (NFR) also play a crucial role in ensuring the system operates efficiently and meets quality standards. NFR typically includes factors such as **Performance**, **Scalability**, **User-friendliness**, **Offline support**, **Privacy and security**, which help define the criteria for evaluating the system's performance rather than specific behavior. The system should support the following key functions:

- **Performance:** The system should respond quickly and render images without long waiting times.
- **Scalability:** The system should work with different AI models (ChatGPT, Claude, Gemini) through APIs.
- **User-friendliness:** The interface should be simple and easy to use, even for people with no LaTeX or technical background.
- **Offline support:** The app should allow users to access saved content even when there is no internet connection.
- **Privacy and security:** User data should be safe and not shared without permission.

Each non-functional requirement above plays a crucial role in ensuring the system runs smoothly, securely, and efficiently on mobile platforms. Therefore, the project is committed to fulfilling these requirements to deliver a reliable, user-friendly, and responsive application that meets real-world expectations in supporting AI-driven scientific work.

2.3 System Functionalities

This section provides an overview of the key functionalities that make up the proposed system. The main goal is to allow users to input scientific prompts, receive AI-generated responses in LaTeX or TikZ format, and visualize them seamlessly in the application. These functionalities are achieved through a tight integration of the Flutter frontend, the Flask-based LaTeX rendering middleware, and the AI model.

Table 2.1 below presents a summarized view of these core functionalities and how each component contributes to the system's overall operation.

Table 2.1 Overview of System Functionalities

No	Feature	Description	Input	Output
1	**Send Question**	System allows the user to input a question and send it to the AI system	Raw user input (text prompt)	–
2	**Return AI Response**	AI Model receives the question and generates an appropriate LaTeX or TikZ code-based answer	User question	AI-generated LaTeX or TikZ text

No	Feature	Description	Input	Output
3	**Render LaTeX Code as Image**	Middleware converts LaTeX and TikZ content into an image and returns it in Base64 format	Content returned from AI contains LaTeX and TikZ code	Base64 rendered image

Table 2.1 summarizes the core functionalities that define how the system operates from input to output. Each feature is carefully designed to ensure smooth interaction between the user, AI model, and LaTeX rendering engine. These functionalities provide the foundation for building and analyzing the system's architecture and designing. The next chapter, **Chapter 3: System Design**, will present detailed use case diagrams and component structures that reflect how these functionalities are implemented in practice.

Chapter 3: System Design

To convert AI-generated LaTeX content—especially TikZ diagrams—into displayable images, a Flask-based middleware was developed to act as a bridge between the language model and the frontend application. This service is responsible for extracting LaTeX code from plain text, compiling it, and converting the output into image format. The development approach focuses on designing a clear data processing pipeline, integrating LaTeX compilation tools, and ensuring smooth delivery of visual content to the user interface. The following sections outline each stage of this pipeline in detail.

3.1 Overview Pipeline

Creating a dedicated section for the system overview pipeline is important to clearly show how the AI PELaX system works from input to output. For the AI PELaX project, this high-level pipeline helps us quickly understand the main data flow, supports development planning, and identifies potential challenges in system integration or performance early. This contributes to building a more reliable and well-structured system. Below is the overview pipeline of the whole system, showing the main components and how they interact.

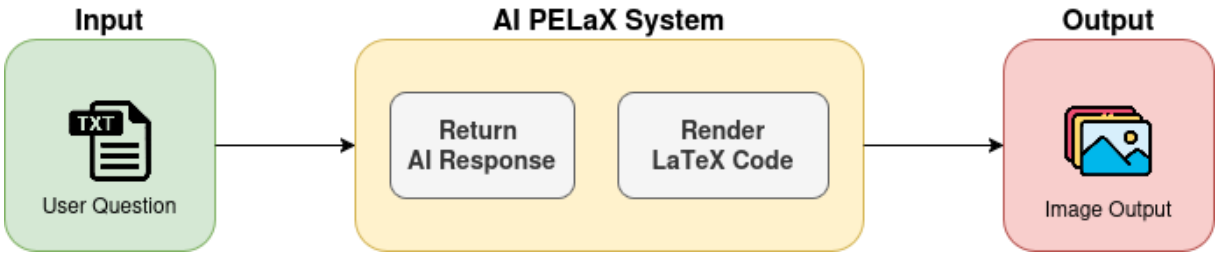

Figure 3.1 System Overview Pipeline

Figure 3.1 shows a simple overview of the AI PELaX system. It starts with the user's input in text format and ends with an image output displayed in the Flutter application. The process includes three main steps: **Receiving the user input**, **Sending the user input to the AI model**, and **Rendering the LaTeX content as an image**. This pipeline helps everyone easily understand the main flow of the system, making it easier to plan, build, and test each part.

Table 3.1 Detailed Description of the AI PELaX System Pipeline

Stage	**Component**	**Role**	**Actor**	**Input Format**	**Output Format**
Input	User Question	The user submits a text-based question, possibly requesting a diagram, formula, or image.	User	Text	Text
AI Processing	Return AI Response	The system sends the question to an AI model and receives a response that may include LaTeX or TikZ code.	AI Model	Text	Text (with LaTeX/TikZ)
Middleware	Render LaTeX Code	The middleware extracts LaTeX/TikZ code from the AI response and compiles it into an image.	Middleware	LaTeX/TikZ Code	PNG / Base64 Image
Output	Image Output	The frontend decodes the image and displays it to the user as part of the final answer.	Flutter App (UI)	Base64 Image (in JSON)	Displayed Image

The table above shows the full pipeline of the AI PELaX system. It begins with a user submitting a question and ends with the final image displayed in the app. Each stage has a clear role and works with specific input and output formats. The system connects the AI model with a middleware that processes LaTeX code into images. This ensures that users receive accurate and visual responses to their questions.

3.2 System Architecture

To better understand how the middleware processes AI-generated LaTeX content, this section outlines the complete data flow from user input to final image rendering. The middleware is designed to act as a bridge between AI-generated content and the mobile

application by converting LaTeX or TikZ code into image format. To achieve this, the entire workflow is structured into a clear pipeline consisting of four main processing steps. Figure 3.2 below illustrates this pipeline, showing the transition from user input to the final display response.

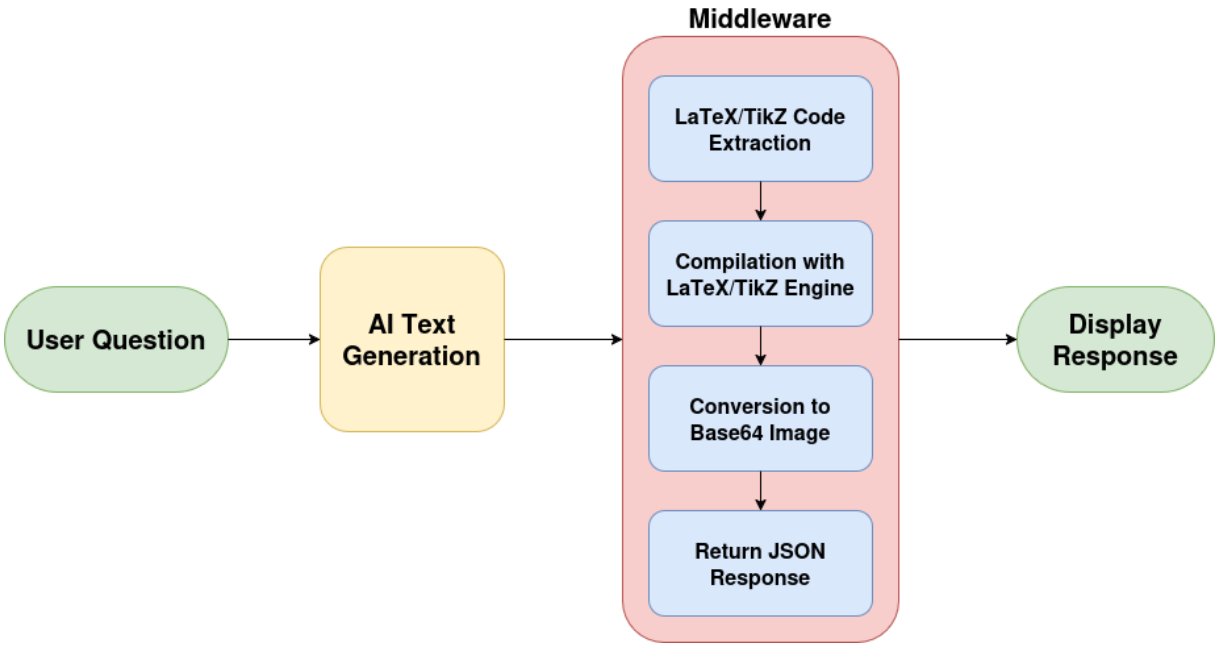

Figure 3.2 AI PELaX Architecture Overview

After the AI generates a text response containing LaTeX or TikZ code, the middleware takes over to extract, compile, and convert the relevant segments into images. Each step in the pipeline is designed to handle a specific task efficiently and independently. Table 3.2 below provides a more detailed explanation of the role and purpose of each step in the pipeline.

Table 3.2 AI PELaX Pipeline: Step-by-Step Description

Steps	**Descriptions**
User Question	The user asks a question. The question may include a request to create a diagram, formula, or image using LaTeX/TikZ code.
AI Text Generation	The AI reads the question and generates an answer. If needed, the answer includes LaTeX or TikZ code to create the requested image.
LaTeX/TikZ Code Extraction	The middleware finds and extracts the LaTeX/TikZ code from the AI's response.

Steps	Descriptions
Compilation with LaTeX/TikZ Engine	The middleware compiles the LaTeX/TikZ code using a LaTeX engine to generate an image (such as PNG or PDF).
Conversion to Base64 Image	The generated image is converted to a Base64 string. This makes it easier to send the image as part of the response.
Return JSON Response	The middleware creates a JSON response that includes the Base64 image. This response is sent back to the user interface.
Display Response	The user interface reads the JSON response, decodes the image from Base64, and shows it to the user as part of the answer.

As shown in Table 3.2, the AI PELaX system outperforms other popular AI-based systems such as ChatGPT, Claude, and Gemini in several key aspects, including full support for TikZ rendering and automatic LaTeX image generation, especially for scientific and geometric diagrams. Furthermore, AI PELaX is specifically designed to be highly suitable for educational purposes, enabling students to engage with mathematical content through clear and dynamic visualizations. These advantages contribute to an optimized pipeline—from input handling and content processing, to LaTeX code extraction, image rendering, and display on the mobile application.

The following section will present the detailed system architecture of AI PELaX, explaining how each component collaborates to realize this processing workflow.

3.3 System Implementation

After describing the processing pipeline in **Section 4.1**, this part presents the overall system architecture. The figure below shows how the main components of the system are connected and how data flows between them. It includes the frontend (where users input prompts), the middleware (which processes prompts and handles LaTeX rendering), and the backend AI model (which generates responses). This architecture helps explain how the system works as a whole, from user input to final output.

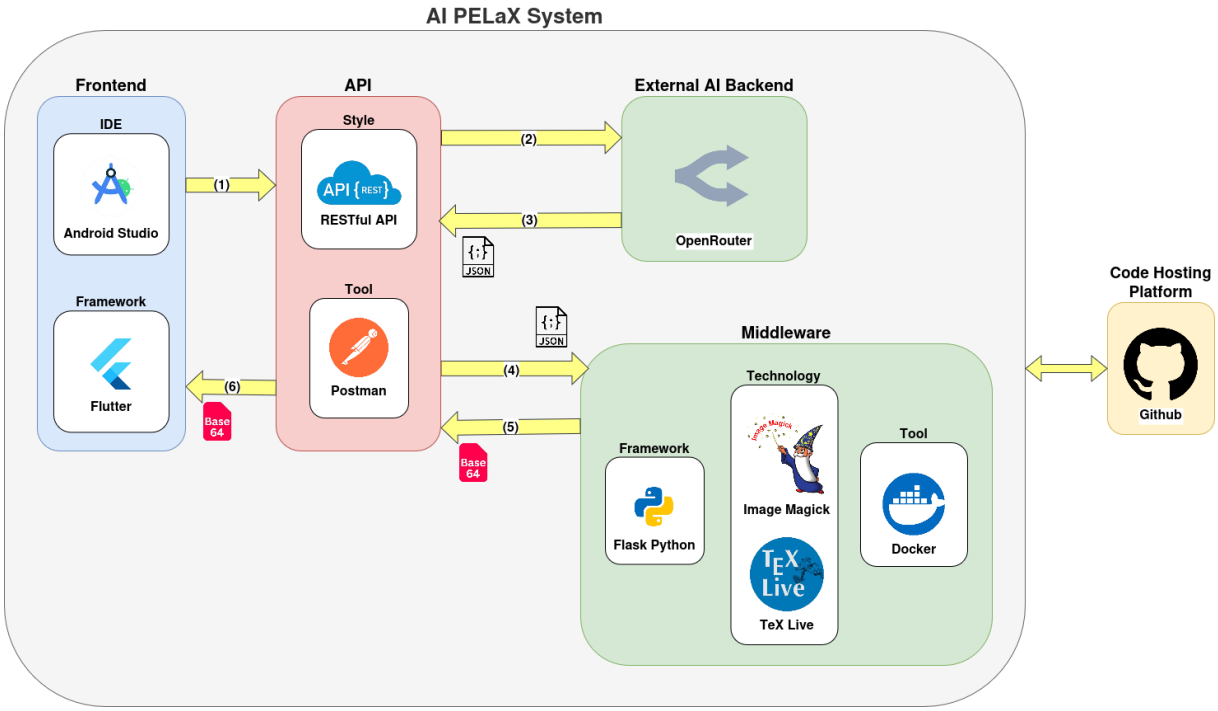

Figure 3.3 AI PELaX System Implementation

The AI PELaX system is designed based on a multi-layered architecture, promoting modularity, separation of concerns, and ease of maintenance. This architecture ensures high scalability, better system organization, and flexibility in integrating external services. Figure 3.3 illustrates the overall system architecture, which consists of five primary components: **Frontend Layer**, **API Layer**, **Middleware Layer**, **External AI Backend**, and **Code Hosting Platform**.

3.3.1 Frontend Layer

The frontend layer is developed using the **Flutter** framework, an open-source UI toolkit by Google, enabling cross-platform application development (Android, iOS, web, and desktop) from a single codebase[3]. The development process is conducted using **Android Studio**, which provides robust support for UI design, debugging, and testing[4]. The frontend layer is responsible for:

- Receiving user input (prompts).
- Sending data to the API layer via HTTP POST requests.
- Displaying the AI response, including both plain text and rendered LaTeX images (encoded in Base64 format).

The use of **Flutter** enhances development efficiency while ensuring a consistent user experience across multiple platforms[3].

3.3.2 API Layer

The API layer serves as the central communication hub between the frontend, middleware, and AI backend. It is implemented in the **RESTful style**, using standard HTTP methods (`GET` , `POST` , `PUT` , `DELETE`) and JSON as the data format. The tool **Postman** is used during development and testing for simulating API requests and verifying responses[5]. Its key responsibilities include:

- Receiving prompts from the frontend.
- Forwarding these prompts to the external AI backend (OpenRouter).
- Receiving AI-generated responses and forwarding them to the middleware for LaTeX processing.
- Returning the final processed results (including LaTeX-rendered images in Base64) back to the frontend.

This separate structure allows for clear data flow and easy integration or substitution of components.

3.3.3 Middleware Layer

The middleware is implemented using **Flask (Python)**, a lightweight web framework ideal for building microservices[6][7]. This layer focuses on processing the AI response, particularly extracting and rendering any embedded LaTeX mathematical expressions into images. Key technologies used in the middleware include:

- **TeX Live**: A comprehensive LaTeX distribution used to compile `.tex` files into `.pdf` documents[8].
- **ImageMagick**: A powerful open-source image manipulation tool used to convert `.pdf` files into `.png` images and apply further image processing[9].

- **Docker**: A containerization platform that encapsulates the entire LaTeX processing environment (**Python**, **TeX Live**, **ImageMagick**) into a portable, reproducible container. **Docker** ensures consistency across different deployment environments and simplifies deployment to production[10].

The LaTeX processing pipeline works as follows:

1. The middleware receives a JSON object containing the AI response.
2. It parses and extracts LaTeX segments using regular expressions or parsing logic.
3. It generates a `.tex` file containing the extracted expressions.
4. The `.tex` file is compiled into a `.pdf` using `pdflatex` (TeX Live).
5. The `.pdf` is converted into a `.png` using convert (ImageMagick).
6. The image is encoded in Base64 format and returned to the API layer for delivery to the frontend.

3.3.4 External AI Backend

The system integrates with **OpenRouter**, which acts as a gateway to multiple advanced AI language models such as OpenAI's GPT, Claude, LLaMA, and Mistral. Communication with **OpenRouter** is handled via RESTful API, enabling the system to offload natural language processing tasks without maintaining in-house AI infrastructure.

This integration provides flexibility and allows the AI PELaX system to scale with newer and more powerful language models without redesigning the core architecture.

Within the scope of the PELaX AI system project, the model we chose to use is **Open-R1**. Other models will have slight differences in how we survey model responses, but basically, all models can handle questions from the user quite well. Therefore, our use of the Open-R1 model is just a random choice among the models that OpenRouter can provide.

3.3.5 Code Hosting Platform

GitHub is used as the version control and code hosting platform. It enables effective source code management, team collaboration, and tracking of development history. The use of GitHub facilitates branch management, issue tracking, and supports continuous integration/deployment (CI/CD) when needed.

3.4 Flask Backend for LaTeX and TikZ Rendering

In order to handle mathematical expressions and TikZ diagrams returned by the AI model, a custom Flask backend was developed. This backend receives text responses, finds LaTeX code inside them, and converts that code into images. These images can then be shown in the app just like normal text. This makes it easier for users to understand formulas and diagrams, especially when they are complex. The backend also supports both short LaTeX expressions and full TikZ documents. The following sections describe how we analyzed AI responses and built the backend step by step.

3.4.1 Surveying AI Response Patterns

Before building the Flask backend, we conducted a survey of the common response formats that AI language models can generate in text passages containing mathematical expressions, chemical formulas, or geometric diagrams using TikZ. This survey result is an important premise for determining the exact rules to apply in the text string processing stage and the LaTeX content parsing before displaying.

Survey data source: we have used the **DeepSeek Open-R1** language model many times in educational contexts, especially high school and college-level mathematics and physics problems. The results show that the majority of mathematical expressions are returned by the model in one of the following formats:

- **Inline LaTeX Base:** `\(...\)` or `$...$`
- **Block LaTeX Base:** `\[...\]` or `$$...$$` or Markdown-style code blocks with triple backticks and the latex language tag (e.g., latex“‘...“‘)
- **Full LaTeX Document:** `\documentclass{...}` or `\usepackage{...}` or `\begin{document}...end{document}`

The Table 3.3 below summarizes the three primary types of LaTeX expressions found in AI-generated responses and describes how each is processed by the Flask backend system:

Table 3.3 Summary of LaTeX Expression Types and Processing Strategies

Expression Type	**Delimiters Used**	**Typical Use Case**	**Processing Strategy**
Inline expressions	`\(...\)`, `$...$`	Short formulas embedded within text	Extract content, convert to display mode, and render as image
Block expressions	`\[...\]`, `$$...$$`	Standalone, multiline equations	Extract full expression, render directly in display mode as image
Full LaTeX documents	\begin{...} environments	Diagrams (e.g., TikZ), tables, full docs	Wrap code in a minimal LaTeX document, compile with `pdflatex`, and convert output to PNG

This classification ensures the backend system can robustly handle various types of LaTeX content generated by the AI model. By applying different rendering strategies based on the structure of the LaTeX code, the system can maximize compatibility, readability, and user experience.

3.4.2 Backend Implementation

After surveying the types of AI responses returned by the language model (***Section 4.3.1***), we developed a backend system using Flask to automatically process and render LaTeX or TikZ code contained in those responses into image format. This backend acts as middleware between the AI model and the Flutter frontend, taking raw text with embedded LaTeX expressions and returning the same content with rendered images for each detected formula or block.

1. **Backend Architecture Overview:**
   The backend is built with the Python web framework Flask, structured into two main files:

   - `app.py` : the main Flask app that receives input, extracts LaTeX content, and returns processed results.

- latex_renderer.py : a module responsible for compiling LaTeX code and converting it into base64-encoded PNG images.

The backend exposes a single API endpoint: POST /render_latex - This endpoint expects a JSON payload containing a field ”latex_code” with the full AI response as input text.

2. **Parsing and Extraction Logic:**
The AI model often returns LaTeX in multiple formats, including:

- Inline or Block formulas: \(...\) or \$...\$ or \[...\] or \$$...\$$
- Full LaTeX documents: \documentclass{...} or \usepackage{...} or \begin{document}...end{document}
- Markdown-style blocks: latex“‘...“‘

To handle these, the backend first cleans the input by removing Markdown-style latex blocks if present. Then it searches for full LaTeX documents using regular expressions and processes them separately. Any remaining content is scanned for inline LaTeX patterns.

This logic is implemented in two main functions:

- `render()` : This is the main function in `app.py` . It receives the POST request at the `/render_latex` endpoint. The function first reads the raw AI response, then:
  1. Removes any Markdown-style LaTeX blocks, such as “‘latex ... “‘ .
  2. Uses regular expressions to detect full LaTeX documents (e.g., those that contain `\documentclass{}` and `\begin{document}`).
  3. Processes these full LaTeX blocks separately by calling `render_latex_to_image()` .
  4. Passes any remaining content to `parse_inline_latex()` to extract and render inline math expressions.
- `parse_inline_latex(text)` : This function scans the input string for commonly used LaTeX math patterns, such as:
  - \(...\) (inline formulas)
  - \[...\] (display math)
  - \$$...\$$ (alternative display math)

  For each detected formula, it attempts to render it into a PNG image by calling the `render_latex_to_image()` function.

3. **Parsing and Extraction Logic:**
   The rendering logic is encapsulated in `latex_renderer.py`. This file is responsible for converting LaTeX code into PNG images and returning the result as a base64-encoded string. The steps are as follows:

   1. A unique identifier (UUID) is generated for each LaTeX rendering request.
   2. The system creates a `.tex` file and writes the full LaTeX content to it. If the input is only a math expression and not a full document, a minimal LaTeX wrapper is automatically added.
   3. The file is compiled to `.pdf` using `pdflatex`. Log and auxiliary files are saved in separate folders for debugging purposes.
   4. The compiled `.pdf` is converted to a `.png` image using ImageMagick's `convert` command.
   5. Finally, the PNG image is read as binary data and encoded in base64. This string is returned so that the frontend can display the image directly.

   If the rendering fails (e.g., due to syntax errors in the LaTeX code), the backend returns the original text as plain content instead of crashing.

4. **Parsing and Extraction Logic:**
   To keep the system organized, the backend saves all intermediate and output files in the following directories:

   - `outputs/tex/`: contains `.tex` files
   - `outputs/pdf/`: contains compiled `.pdf` files
   - `outputs/aux/`: contains `.aux` files with LaTeX metadata
   - `outputs/logs/`: contains `.log` files for debugging
   - `outputs/`: contains final `.png` images

   This directory structure supports debugging and improves maintainability.

5. **Docker Packaging for Deployment:**
   To make the backend easy to run and deploy on any machine, we packaged the entire LaTeX rendering system into a Docker container. Docker allows us to bundle all the dependencies, such as `pdflatex`, `ImageMagick`, and Python packages, into a single image.

   This ensures that the backend runs consistently across different environments without manual setup or installation errors.

   The main steps to containerize the backend are:

1. Create a `Dockerfile` that:
   - Starts from a base image with LaTeX and Python (e.g., `debian` or `python:3.10-slim`).
   - Installs `texlive-full` for full LaTeX support.
   - Installs `ImageMagick` for image conversion.
   - Copies the backend source code into the container.
   - Installs required Python libraries using `pip` and a `requirements.txt` file.
   - Sets the default command to run the Flask app.
2. Build the Docker image using:

   ```
   docker build -t aipelax-backend .
   ```

3. Run the container locally using:

   ```
   docker run -p 5000:5000 aipelax-backend
   ```

With Docker, anyone can launch the backend with a single command, without worrying about dependencies. This makes it easy to integrate with the AI model and Flutter frontend during development, testing, or deployment.

3.5 Sequence Diagram

In this section, we describe how the AI PELaX system was implemented based on the proposed architecture. The implementation focuses on key components such as the user interface for sending questions, the AI text generation module for creating responses, and the middleware that processes and renders LaTeX/TikZ code. To show how these components work together, we use sequence diagrams for each key function. These diagrams help explain the data flow and the process in a clear way. In the following parts, we present common use cases along with their corresponding sequence diagrams.

3.5.1 Sequence Diagram: Send Question

When the user enters a question through the interface, the system receives the input and forwards it to the appropriate processing component. This is the first step in the overall workflow of the AI PELaX system. The following sequence diagram describes the process of sending a question to the system.

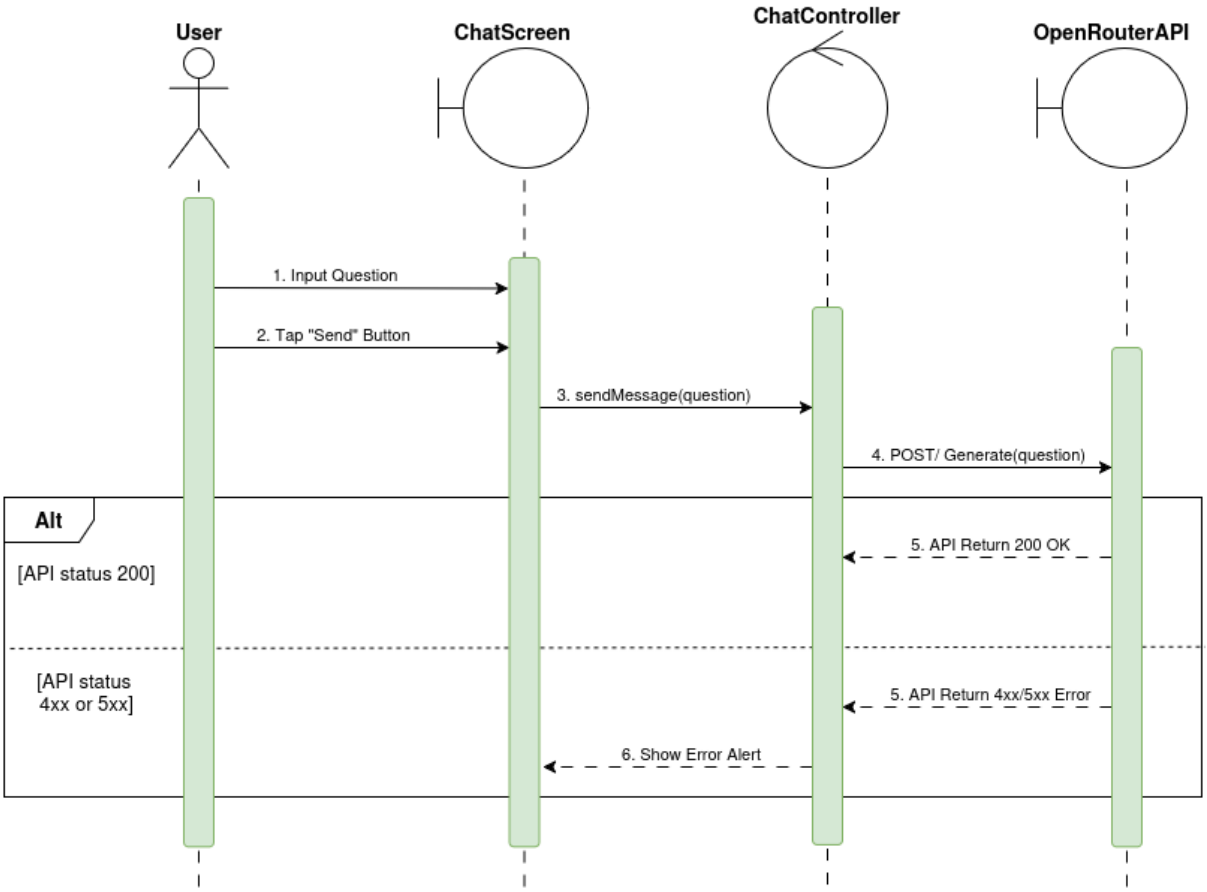

Figure 3.4 Send Question Sequence Diagram

The sequence diagram that shown in Figure 3.4 above describes the process of a user sending a question to an AI model via the Flutter application. The user enters the

question into the ChatScreen interface and presses the ”Send” button, then the question will be transferred to the ChatController for processing. The Controller performs a POST command to call the OpenRouter API to send the question to the AI model. The system checks the response from the API: if it returns successfully (status 200), the process continues; if an error occurs (status 4xx or 5xx), the system will display an error warning to the user.

Below is a detailed description of the steps in the Send Question Sequence Diagram Description:

Table 3.4 Send Question Sequence Diagram Description

Sequence Diagram: Send Question	
Criteria	**Descriptions**
Actor and Components	• **User** *(Actor)*: The person using the application. • **ChatScreen** *(Boundary)*: UI where the user enters the question and taps send. • **ChatController** *(Control)*: Handles the logic for sending the question. • **OpenRouterAPI** *(Boundary/External System)*: External API connected to the AI model.
Interaction Flow	1. The **User** inputs a question on the interface. 2. Taps the ”Send” button. 3. **ChatScreen** calls `sendMessage(question)` in **ChatController**. 4. **ChatController** sends a `POST` request to **OpenRouterAPI**. 5. **OpenRouterAPI** returns the response. 6. If there’s an error, an alert is shown on **ChatScreen**.
Conditions	5a. **Success**: `API status == 200 OK`, Sending response to middleware. 5b. **Failure**: `API status == 4xx/5xx`, Show an error alert to the user.

The Figure 3.4 and its Description Table 3.4 clearly show how users first interact with the AI PELaX system. They explain how a user's question is collected, handled by the interface and controller, and then sent to the AI model using OpenRouterAPI. The diagram also shows what happens when the API works correctly or fails, making sure the user always gets feedback. These steps are the starting point for the system to give smart answers and continue processing.

3.5.2 Sequence Diagram: Return AI Response

The Return AI Response Sequence Diagram (see Figure 3.5) describes how the AI model processes a user's question and returns a response to the backend system. The AI generates a reply that may contain LaTeX or TikZ code, which is sent back through the OpenRouterAPI. This response is then forwarded to the MiddlewareAPI for validation before continuing. The flow also handles both successful and invalid responses using `alt` conditions to manage errors. These steps ensure that only clean and valid responses are passed on to the next stage for rendering.

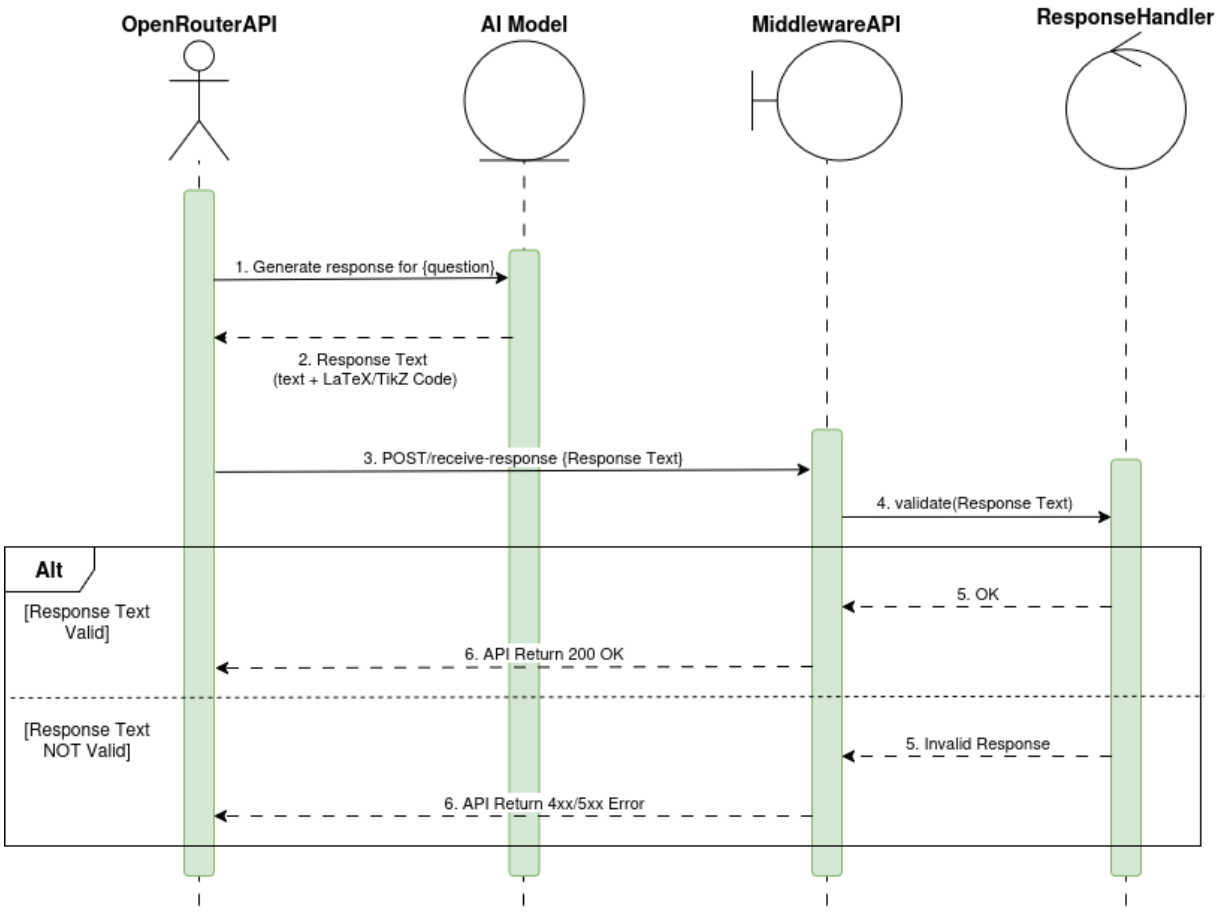

Figure 3.5 Return AI Response Sequence Diagram

Below is **Table 3.5: Return AI Response Sequence Diagram Description** with further breakdown of each interaction.

Table 3.5 Return AI Response Sequence Diagram Description

Sequence Diagram: Return AI Response	
Criteria	**Descriptions**
Actor and Components	• **OpenRouterAPI** *(Actor)*: Sends user question to the AI model. • **AI Model** *(Entity)*: Generates response with text and LaTeX. • **MiddlewareAPI** *(Boundary)*: Handles the AI response and sends it for validation. • **ResponseHandler** *(Control)*: Validates the response text from the AI.
Interaction Flow	1. **OpenRouterAPI** asks the AI Model to generate a response for the user's question. 2. **AI Model** returns the response text (with LaTeX/TikZ). 3. **OpenRouterAPI** sends the response text to MiddlewareAPI via `POST` method. 4. **MiddlewareAPI** validates the response by calling **ResponseHandler**. 5. **ResponseHandler** returns validation result. 6. **MiddlewareAPI** sends success `(200 OK)` or failure `(4xx/5xx Error)` back to **OpenRouterAPI**.
Conditions	5a. If the response text is valid: **MiddlewareAPI** returns `(200 OK)` to **OpenRouterAPI**. 5b. If the response text is not valid: **MiddlewareAPI** returns `(4xx/5xx Error)`.

The description provided in Table 3.5, along with the interactions shown in Figure 3.5, clearly defines how the system processes and validates the AI-generated response. It ensures a reliable communication flow between the AI model and the backend, handling both valid and invalid outcomes gracefully. This diagram acts as a foundation for error management and prepares the system for the next rendering step.

3.5.3 Sequence Diagram: Render LaTeX Code

The Render LaTeX Code Sequence Diagram (see Figure 3.6) shows how the system processes and renders LaTeX or TikZ code from an AI-generated response. After the Flutter app receives the AI response, it sends the data to the MiddlewareAPI, which then breaks down and processes LaTeX segments through several backend components. If rendering is successful, the output is encoded and returned as Base64 image data. If there is an error, such as LaTeX compile failure, an appropriate error message is sent back to the app. This flow ensures smooth LaTeX handling or clear feedback when rendering fails.

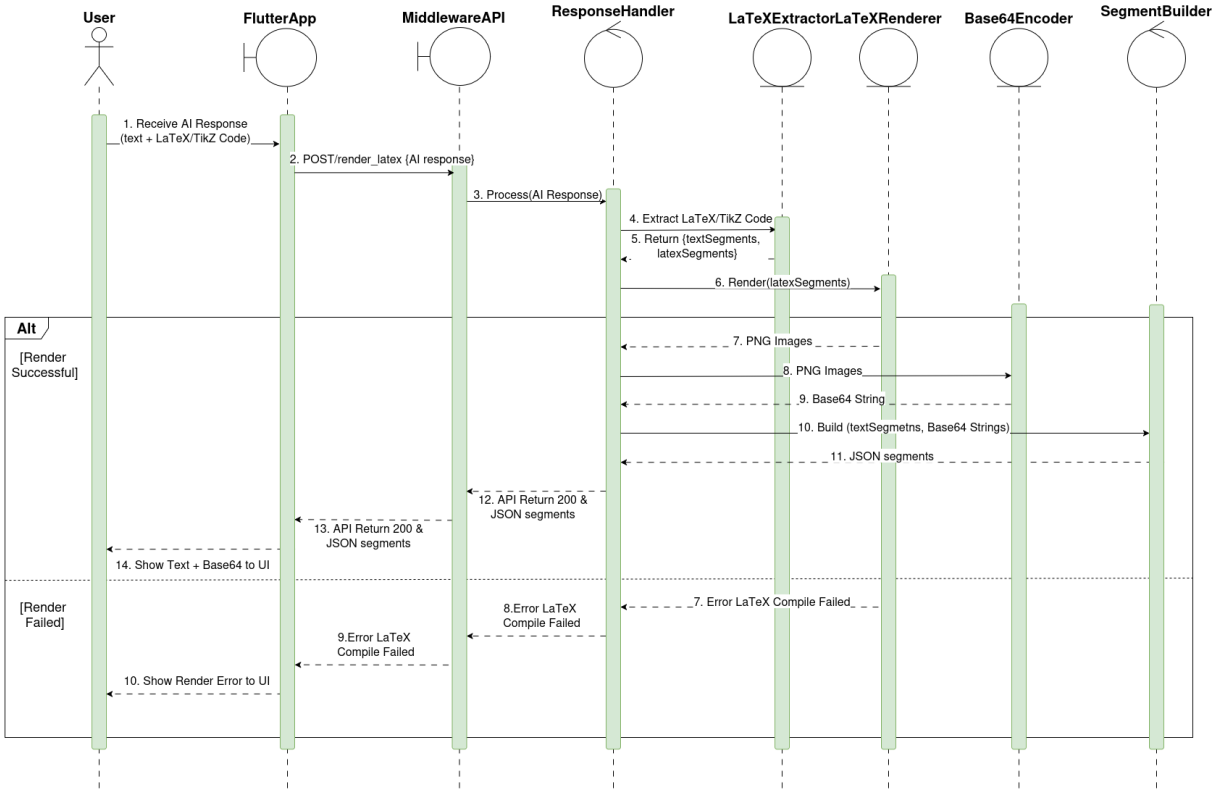

Figure 3.6 Render LaTeX Code Sequence Diagram

Below is **Table 3.6: Render LaTeX Code Sequence Diagram Description**, which explains each part of the sequence in more detail.

Table 3.6 Render LaTeX Code Sequence Diagram Description

Sequence Diagram: Render LaTeX Code	
Criteria	**Descriptions**
Actor and Components	• **User** *(Actor)*: A person using the Flutter app to send questions and view answers. • **FlutterApp** *(Boundary)*: The mobile app that interacts with the user and backend. • **MiddlewareAPI** *(Boundary)*: The backend API that receives and processes AI responses. • **ResponseHandler** *(Control)*: Component that separates plain text and LaTeX/TikZ code. • **LaTeXExtractor** *(Entity)*: Identifies LaTeX code blocks. • **LaTeXRenderer** *(Entity)*: Compiles LaTeX/TikZ code into PNG images. • **Base64Encoder** *(Entity)*: Encodes PNG images into Base64 strings. • **SegmentBuilder** *(Control)*: Combines text and images into a single structured JSON.
Interaction Flow	1. User receives AI response with LaTeX. 2. FlutterApp sends this response to MiddlewareAPI (render_latex). 3-5. MiddlewareAPI processes the response, extracts and separates LaTeX. 6-8. LaTeXRenderer tries to compile LaTeX segments to PNGs. 9. PNGs are encoded as Base64. 10-11. SegmentBuilder creates structured JSON with text and image data. 12-13. MiddlewareAPI returns the result (success or failure). 14. FlutterApp shows output or error to the user.

Sequence Diagram: Rendert LaTeX Code	
Criteria	**Descriptions**
Conditions	• **Render Successful**: All LaTeX compiles succeed; Base64 strings and JSON segments are returned with HTTP 200. • **Render Failed**: One or more LaTeX segments fail; error message is returned instead.

As shown in both Figure 3.6 and Table 3.6, the rendering sequence is a carefully structured pipeline to process, compile, and return LaTeX content efficiently. The use of multiple dedicated components ensures that each stage—from extraction to rendering and encoding—is clearly separated and easy to debug. When everything works correctly, users see both text and rendered formulas in the UI. If a failure occurs, the system gracefully catches it and informs the user. This diagram provides a strong foundation for understanding LaTeX rendering in the application.

Chapter 4: Results and Discussions

After the design and implementation phases, this chapter presents the experimental results and evaluation of the system developed as part of the project. In addition, the discussion section provides an in-depth analysis of strengths, limitations, and areas for future improvement based on both user feedback and testing observations. These insights are essential for assessing the system's level of completeness and its potential for real-world deployment.

4.1 Results

This section briefly outlines the main results of the system. Each result corresponds to a completed feature or milestone that was implemented, tested, and verified during development.

1. **Full Integration Between Flutter App, AI Model, and Middleware**
   The entire pipeline—from user input on the Flutter app to AI processing and LaTeX rendering—was integrated successfully.

   - The user is able to enter a scientific question in natural language (e.g., math problems, graph requests, or TikZ drawing instructions).
   - The question is sent to an AI model hosted on a backend, and the model processes it effectively to produce LaTeX-based content.
   - The middleware receives this LaTeX content, detects all math or TikZ code, and renders it into image format (Base64), which the app can display.

   This seamless connection ensures a responsive and consistent user experience.

2. **LaTeX Code is Accurately Rendered into Images**
   A Flask-based middleware was developed to extract LaTeX code and render it into image format using TeXLive and ImageMagick.

   - The system supports multiple LaTeX segments in a single response.
   - Images are encoded in Base64 and returned to the Flutter frontend.
   - Rendering time is optimized to ensure low latency, typically under 3 seconds per prompt.

   The rendered images are clean, readable, and correctly represent the original LaTeX expressions.

3. **Flutter App Receives and Displays Responses Smoothly**
   The frontend is built using Flutter and supports dynamic rendering of both plain text and Base64 images returned from the middleware.

   - The chat interface is designed to handle mixed content (text and images).
   - Image messages load instantly once received and fit well into the chat layout.
   - Error cases such as malformed LaTeX or empty responses are also handled gracefully.

   The app provides a natural, intuitive experience for users exploring AI-generated scientific content.

4. **Comprehensive Testing and Debugging**
   All components were tested individually and together:

   - Use case testing ensured that each major user action worked correctly.
   - LaTeX rendering was tested with diverse input types (formulas, drawings, equations).
   - Network requests between app, model, and middleware were verified for correctness and timing.

   No critical bugs were found during final testing, and the system performed stably under normal usage.

4.2 Discussion

After successfully building and testing the system, the next step is to look back at what has been achieved, compare it with other similar systems, and reflect on the challenges faced during development. This part helps summarize key lessons learned and highlights areas that still need improvement in the future.

4.2.1 Comparison with other systems

To better understand the strengths and limitations of the proposed system, it is useful to compare it with existing AI-powered platforms that also deal with scientific or LaTeX-based content. These include general-purpose AI models such as ChatGPT and Claude, and specialized tools like Mathpix which offer LaTeX rendering capabilities. The table below summarizes the comparison based on several key criteria:

Table 4.1 Comparison between AI PELaX and other AI-based systems

Criteria	AI PELaX	ChatGPT	Claude	Gemini
Platform Type	Mobile App	Web-based, Mobile App	Web-based, Mobile App	Web-based, Mobile App
Prompt Support	General, scientific prompts	General, scientific prompts	General, scientific prompts	General, scientific prompts
LaTeX Output	**Yes**	**Yes**	**Yes**	**Yes**
TikZ Support	**Full rendering**	Text only	Text only	Text only
Automatic LaTeX Image Rendering	**Yes**	**Yes**	No	**Yes**
Suitable for Education	Highly suitable	Partial	Partial	Partial

As shown in Table 4.1, AI PELaX fills a unique gap by providing an end-to-end experience for generating, processing, and visually rendering scientific content—especially TikZ—directly on a mobile device. Unlike other systems, which only return TikZ as plain text, AI PELaX renders the output as images, making it easier for students and researchers to interpret the content quickly without additional tools like Overleaf or manual compilation.

While general AI systems like ChatGPT, Claude, and Gemini are powerful in natural language understanding, they are not optimized for scientific visualization. In contrast, AI PELaX is designed specifically for this use case, offering a more focused and user-friendly solution for those working with technical subjects.

4.2.2 Challenges and Limitations

Despite the successful integration of major components, several challenges were encountered during the development process:

- **LaTeX Rendering Errors:** Some AI-generated LaTeX expressions were incomplete or syntactically incorrect, leading to rendering failures. This required additional validation and error-handling logic in the middleware.
- **Flutter UI Limitations:** Rendering dynamic images in a scrollable, chat-style interface sometimes caused layout issues or performance slowdowns on lower-end devices.
- **Slow Response Time:** Another limitation is the slow response time when rendering LaTeX, mainly due to the heavy processing involved in compiling LaTeX code and converting it to image format. This delay can affect the overall user experience.
- **Model Inaccuracy:** The AI model occasionally misunderstood prompts, especially when they contained ambiguous or highly advanced scientific topics. This affected the quality of the generated LaTeX code.
- **Data Loss on Restart:** Currently, the app does not persist any user data between sessions. Once the application is closed or restarted, all prompt history and rendered results are lost. This significantly reduces the user experience, especially for those who want to revisit previous content without re-entering the same questions.

Despite these limitations, the system achieved its intended goals. These challenges also point toward promising areas for future improvement, such as better question understanding and richer error feedback for LaTeX issues.

Chapter 5: Conclusion and Future Work

After the research, design, and implementation phases, this thesis has successfully completed the "AI PELaX" project—a system designed to process and visualize scientific content generated by large language models (LLMs), especially LaTeX TikZ code. This chapter summarizes the key achievements, evaluates how well the project objectives have been met, and reflects on the system's limitations. Based on this evaluation, several future development directions will be proposed to improve user experience and expand the system's application in educational and research contexts.

5.1 Conclusion

In the era of rapid advancements in artificial intelligence, large language models (LLMs) such as ChatGPT, Claude, and Gemini have demonstrated great potential in generating structured scientific content, including text, mathematical formulas, and visual diagrams using LaTeX. However, one of the major limitations is that most of this output—especially TikZ code—is returned in plain text format, making it difficult for ordinary users to compile and view without technical expertise. From this practical challenge, the "AI PELaX" project was developed as a modern and efficient solution to automatically process and visualize LaTeX TikZ content.

This project successfully proposed and implemented a middleware system acting as an intermediary between the user-facing Flutter application and the AI model. The middleware automatically extracts LaTeX segments from the AI's response, compiles them into displayable images (in base64 format) using a professional LaTeX engine, and returns them for direct rendering in the app. This enables users to view scientific diagrams and graphs without needing technical tools or configuration.

Throughout the development process, the project focused not only on technical completeness but also on user experience and system feasibility. The application interface is user-friendly, the data processing workflow is fast and secure, and the system is capable of operating in a local network environment to ensure data safety.

In conclusion, AI PELaX has proven its practical value and potential application in education, academic research, and modern scientific environments. It marks an important first step toward deeper integration between AI and scientific visualization systems in the near future.

5.2 Future Work

Although the AI PELaX system has successfully fulfilled its original functional objectives and can be deployed in real-world use cases, there remains significant potential for future improvements and expansion. To evolve into a robust and domain-specific scientific tool, the following development directions are proposed:

- **Cross-platform and multilingual expansion**
  To broaden its accessibility, AI PELaX can be developed into a web-based or desktop platform in addition to its current Flutter implementation. Moreover, by incorporating multilingual support (e.g., Vietnamese and English), the system can serve a wider audience including students, educators, and researchers around the world.

- **Saving Prompt and Response History**
  A simple yet valuable future development direction is the implementation of a prompt and response history feature. This allows users to conveniently revisit previous prompts and AI responses, reducing the need to repeatedly send similar questions. The history can be stored entirely on the local device using solutions such as SharedPreferences or sqflite, ensuring offline access and maintaining data privacy.

- **Interactive editing of TikZ diagrams**
  Currently, TikZ diagrams are rendered as static images with no possibility for user interaction. Future improvements should focus on enabling direct interaction with diagram elements, similar to how it works in GeoGebra. This includes features such as dragging, resizing, and modifying shapes or styles. Such functionality requires integrating an interactive graphics engine and ensuring two-way mapping between visual edits and the original TikZ code.

- **Performance optimization through caching**
  Since many prompts (e.g., standard mathematical queries) may be reused or repeated, implementing a smart caching mechanism can significantly improve system responsiveness. Cached AI outputs and image renderings can be stored and retrieved instantly, thereby reducing redundant computation and lowering the system load, especially at scale.

In conclusion, the proposed future work directions not only aim to improve the current system but also expand its practical applicability. Focusing on user-friendliness, scalability, and performance optimization will help the system better meet real-world demands for visualizing scientific content in a more intuitive and accessible way. Moving forward, these improvements can serve as a foundation for smarter and more specialized applications in fields such as education, research, and knowledge dissemination.

References

[1] F. R. Jensenius, M. Htun, D. J. Samuels, D. A. Singer, A. Lawrence, and M. Chwe, "The benefits and pitfalls of google scholar," *PS: Political Science & Politics*, vol. 51, no. 4, pp. 820–824, 2018.

[2] J. Belouadi, S. P. Ponzetto, and S. Eger, "Detikzify: Synthesizing graphics programs for scientific figures and sketches with tikz, 2024b," *URL https://arxiv.org/abs/2405.15306*, 2024.

[3] A. Tashildar, N. Shah, R. Gala, T. Giri, and P. Chavhan, "Application development using flutter," *International Research Journal of Modernization in Engineering Technology and Science*, vol. 2, no. 8, pp. 1262–1266, 2020.

[4] J. DiMarzio, *Beginning android programming with android studio*. John Wiley & Sons, 2016.

[5] P. P. Kore, M. J. Lohar, M. T. Surve, and S. Jadhav, "Api testing using postman tool," *International Journal for Research in Applied Science and Engineering Technology*, vol. 10, no. 12, pp. 841–43, 2022.

[6] J. Chan, R. Chung, and J. Huang, *Python API Development Fundamentals: Develop a full-stack web application with Python and Flask*. Packt Publishing Ltd, 2019.

[7] N. Loubser, "Creating a restful api: Flask," in *Software Engineering for Absolute Beginners: Your Guide to Creating Software Products*. Springer, 2021, pp. 193–233.

[8] TeX Users Group, "Tex live – a comprehensive tex system," https://www.tug.org/texlive/, 2025, accessed: 2025-06-28.

[9] M. Still, "Imagemagick," 2006.

[10] I. Miell and A. Sayers, *Docker in practice*. Simon and Schuster, 2019.

Appendix

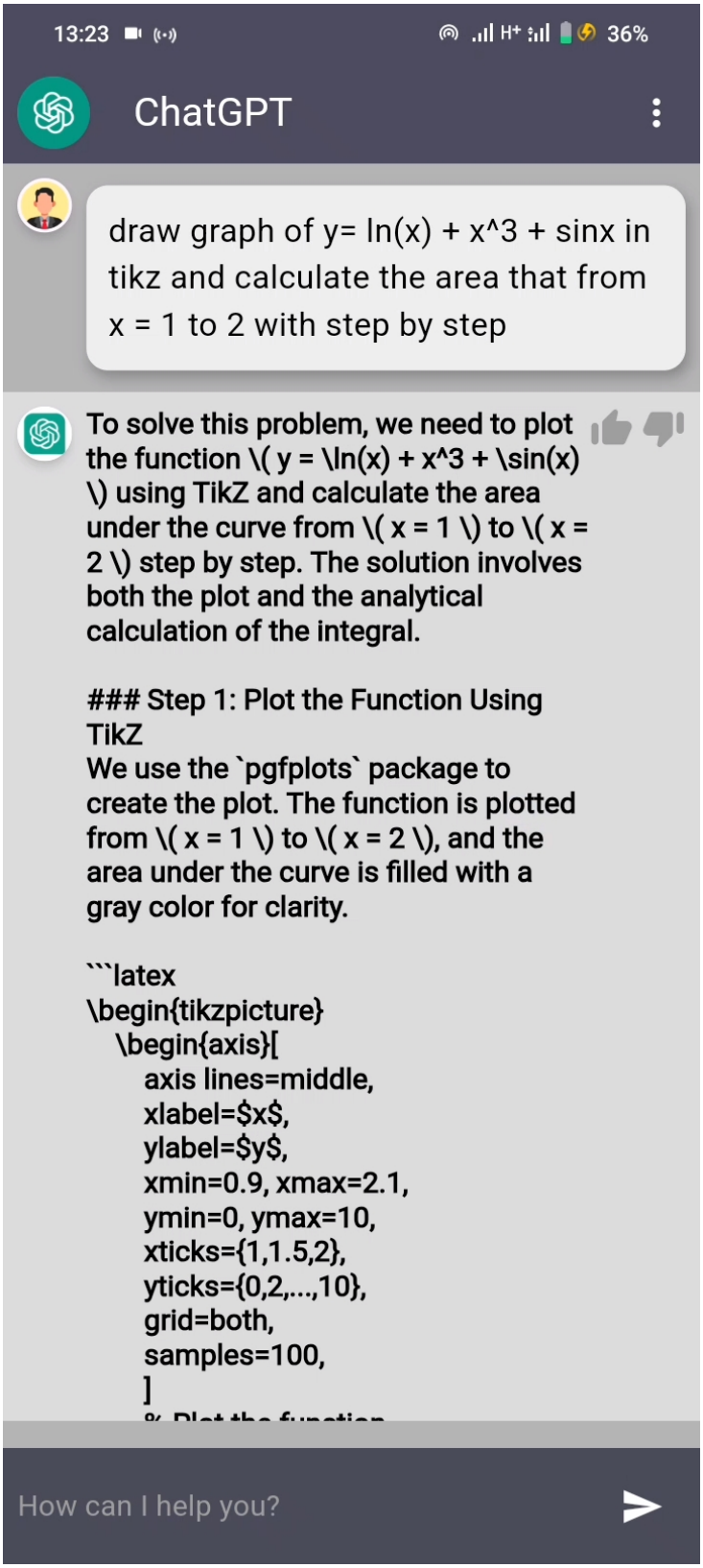

Figure 5.1 AI Raw Text-Based Response (Before apply Middleware)

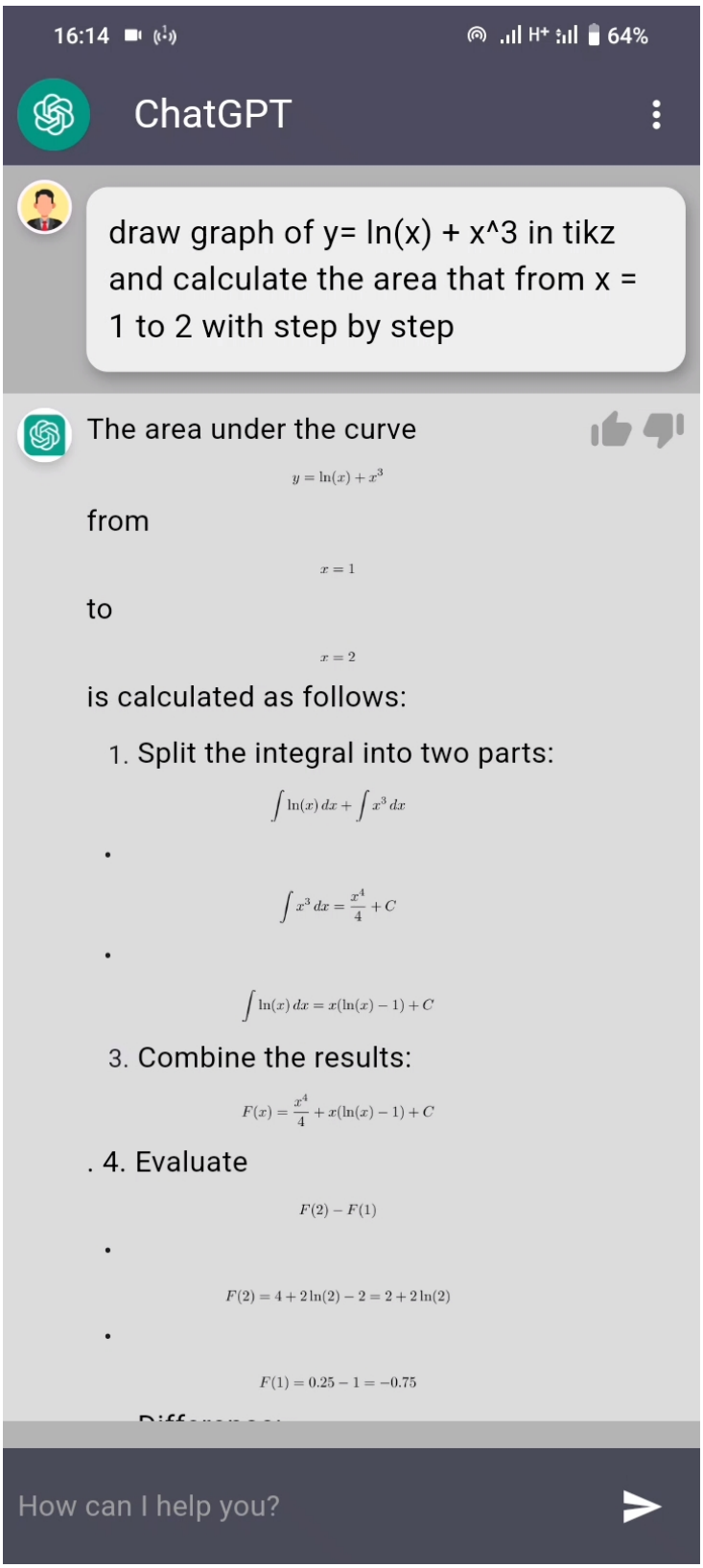

Figure 5.2 AI Response with LaTeX Image (After apply Middleware)

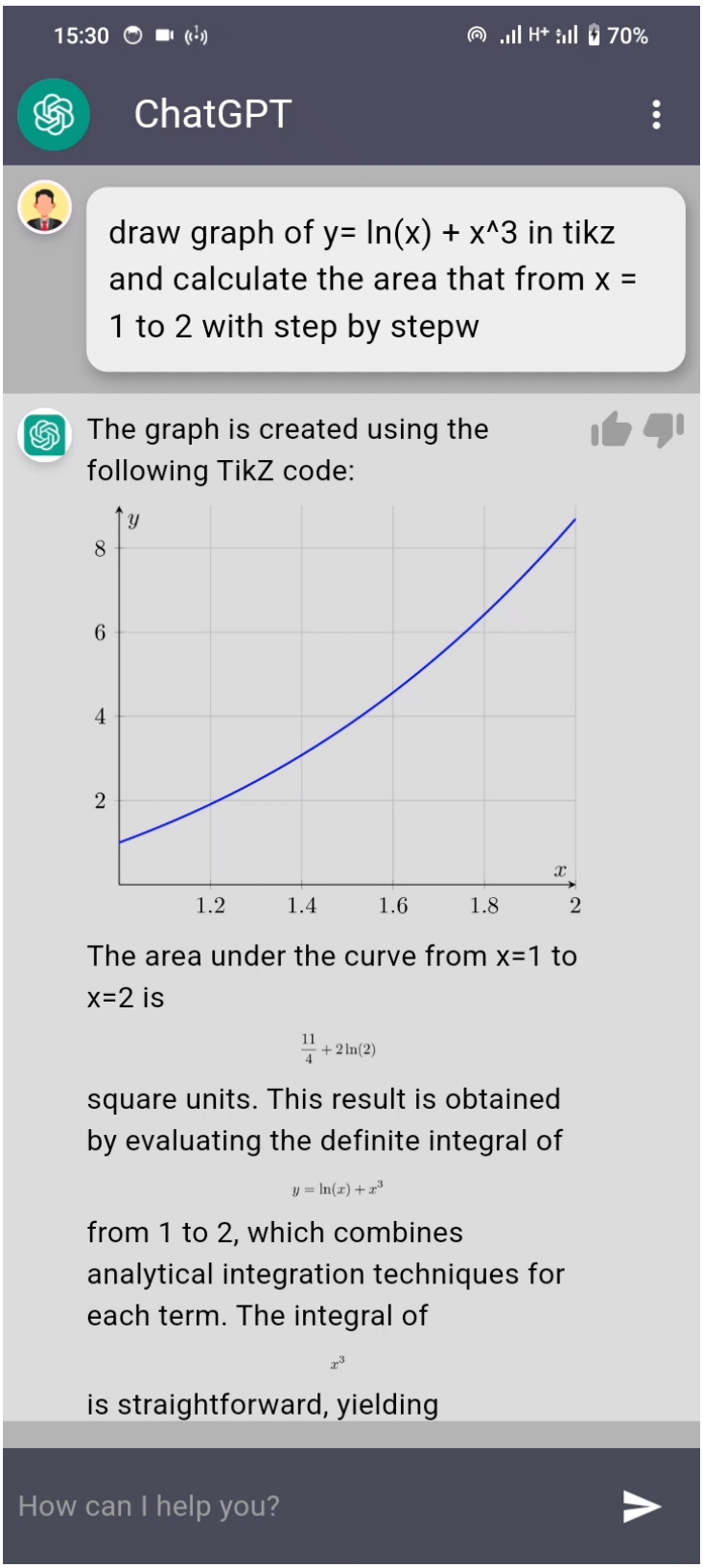

Figure 5.3 AI Response with both TikZ and LaTeX Image (After apply Middleware)

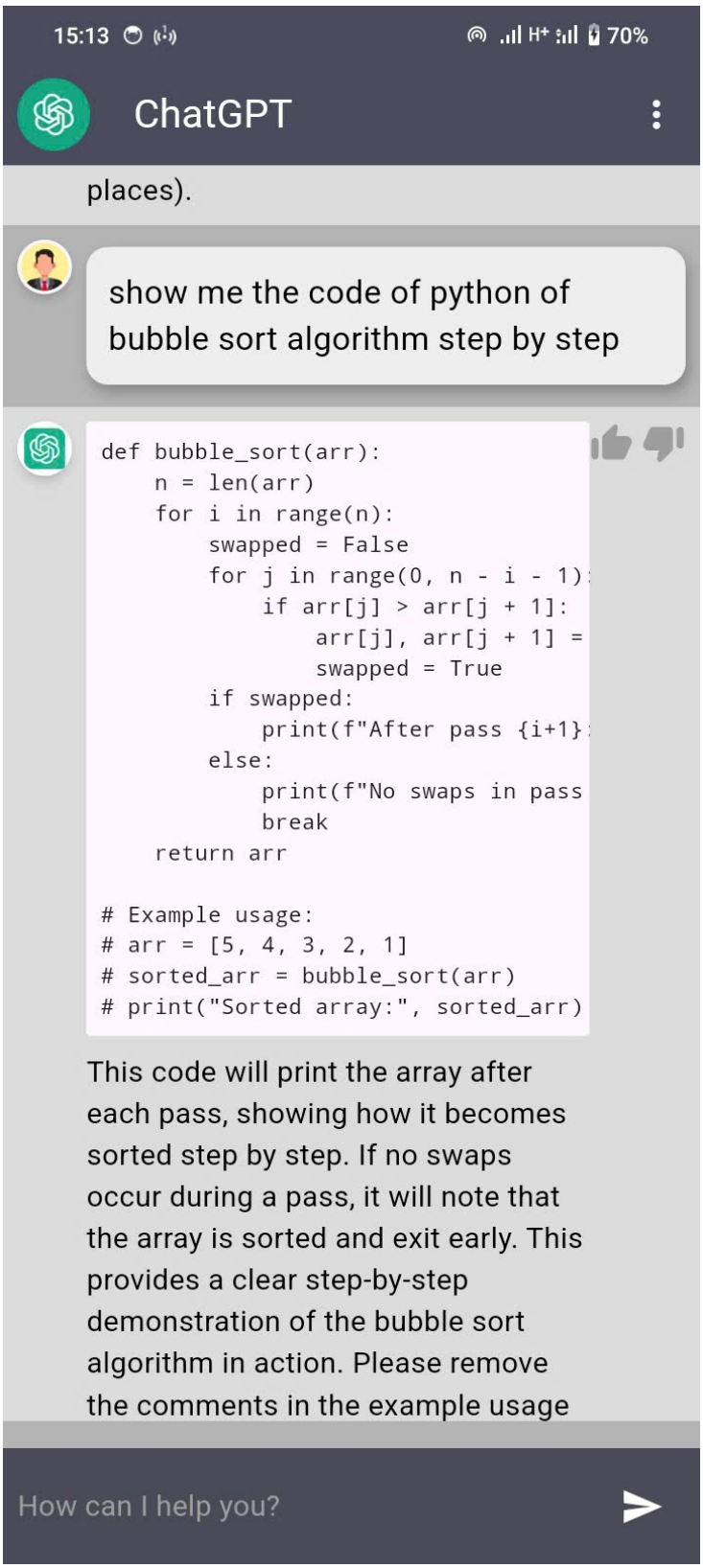

Figure 5.4 AI Response with Box of Block Code (After apply Middleware)

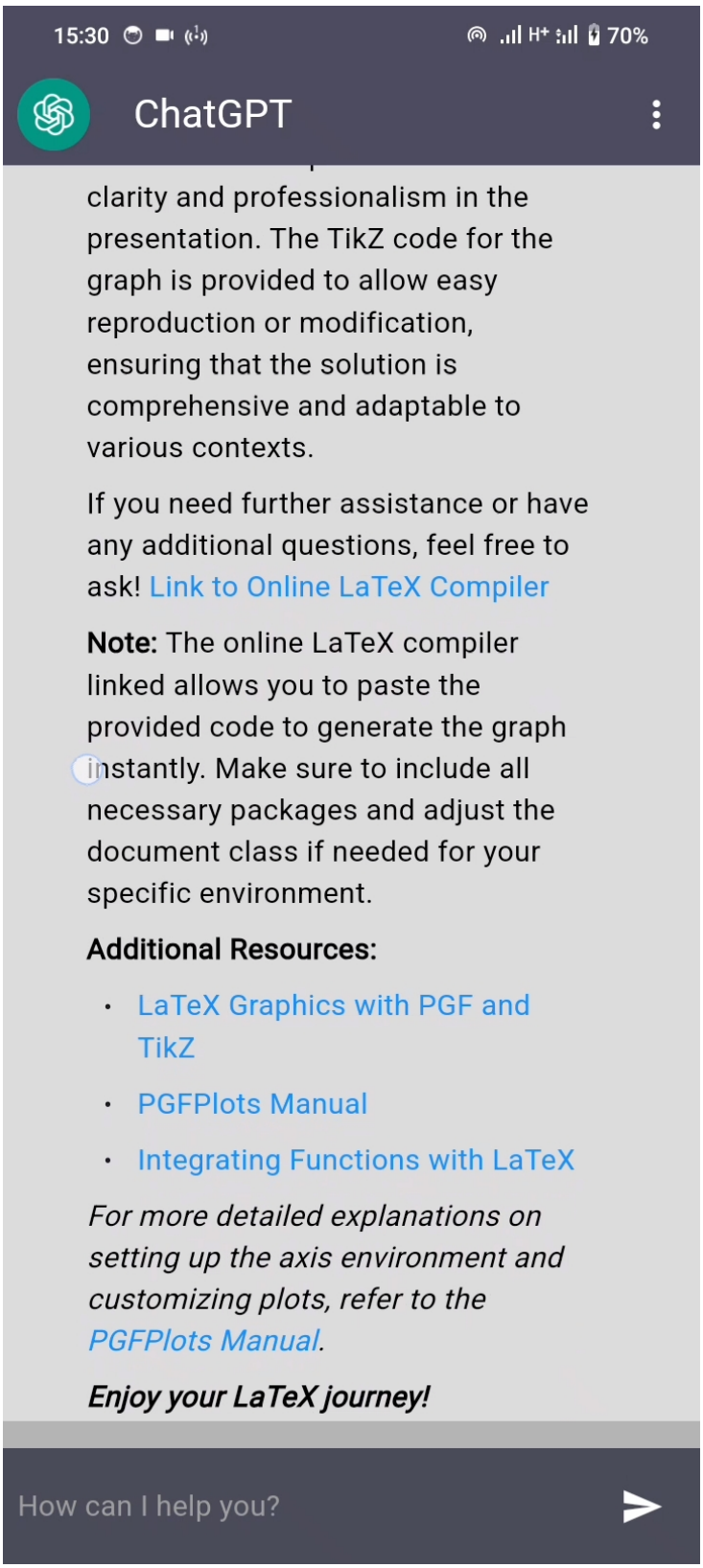

Figure 5.5 AI Response with Markdown (After apply Middleware)

To whom it may concern,

I, MSc.Le Nhu Chu Hiep, certify that the thesis/internship report of Mr.Nguyen Tuan Anh is qualified to be presented in the Internship Jury 2024–2025.

Hanoi, July 2nd, 2025

Supervisor's signature