Semester Long Internship Spring 2026

On

eSim Chatbot Development

Submitted by

Gaurav Gupta
VIT Bhopal University

Under the guidance of

Prof. Prabhu Ramachandran
Principal Investigator
Department of Aerospace Engineering
Indian Institute of Technology Bombay

June 3, 2026

Acknowledgment

I express my sincere gratitude to Prof. Prabhu Ramachandran for providing me with the opportunity to be part of the FOSSEE internship program and for his continued efforts in promoting open-source engineering tool development. His leadership and vision have been instrumental in fostering meaningful student participation in the open-source ecosystem.

I also acknowledge Prof. Kannan M. Moudgalya for his foundational role in establishing and strengthening the FOSSEE initiative. His contributions to open-source education and the development of the FOSSEE fellowship framework have been pivotal in creating the academic and organisational platform through which this internship was undertaken.

My sincere appreciation extends to my mentor, Sumanto Kar, for his continual support, technical guidance, and encouragement throughout the duration of this project. His insights and feedback played a key role in refining ideas, overcoming challenges, and ensuring timely completion of the tasks assigned to me.

I would also like to thank my internal mentors, Mr. Varad Patil and Ms. Shanthi Priya K for their valuable guidance, coordination, and technical inputs during the internship. Their mentorship contributed significantly to the clarity, progress, and successful execution of the work.

This internship has been an enriching learning experience, allowing me to work closely with open-source EDA tools, develop IC subcircuits in eSim, and gain exposure to realworld circuit modeling and simulation workflows. The knowledge acquired during this period will undoubtedly support my future academic and professional pursuits.

I would also like to thank the entire FOSSEE team for their coordination, assistance, and timely interactions at various stages of this work. Their collective efforts ensured smooth workflow, resource accessibility, and effective project execution.

Contents

1 Introduction **3**
1.1 Background 3
1.2 Overview of eSim 4
1.3 Objectives of the Project 4
1.4 Methodology 5

2 Literature Survey **6**

3 Problem Statement **9**
3.1 Problem Statement 9
3.2 Approach 10

4 Implementation **12**
4.1 Case 1: Chatbot UI Architecture and Vision Image Analysis: 12
4.2 Case 2: Bug Fixes and Reliability Improvements 18

5 Conclusion and Future Scope **22**

6 Bibliography **24**

Chapter 1

Introduction

1.1 Background

This project focuses on solving a very common problem faced by software users. Whenever users get stuck while using a tool, they usually have to stop their work, search online for solutions, read documentation or forums, and then return to the software to apply what they learned. This process interrupts workflow and wastes time. With the advancement of Large Language Models (LLMs), it has now become possible to integrate intelligent conversational assistants directly into software applications. These AI systems can understand human language, answer questions, and provide guidance instantly within the same environment where the user is working.

This concept is especially useful in the field of Electronic Design Automation (EDA). Circuit design and simulation involve many technical concepts such as component behaviour, SPICE commands, simulation settings, layout rules, and troubleshooting errors. Experienced engineers may already understand these concepts, but students and beginners often face difficulties while using simulation tools like NgSpice. Simple doubts such as understanding convergence errors, configuring transient analysis, or fixing waveform issues may take a long time to solve through manuals and online resources. Although the information is available, it is not easily accessible at the moment the student needs help.

The FOSSEE (Free and Open Source Software for Education) project at IIT Bombay develops open-source educational tools for engineering and science students across India. One of its major EDA tools is eSim, an open-source software platform for analogue, digital, and mixed-signal circuit design. eSim is built using KiCad and NgSpice and is widely used by students and educators. However, like most traditional EDA tools, eSim does not provide an integrated intelligent support system for users facing difficulties during circuit design or simulation.

To address this challenge, this project introduces an AI-powered assistant integrated directly into eSim. The assistant is designed to help users in real time by answering questions related to circuit design and simulation. It can also analyze schematic images, understand the user's problem context, and maintain continuous conversations to provide better assistance. The goal of the project is to make circuit design learning more interactive, efficient, and user-friendly, especially for students and beginners.

1.2 Overview of eSim

eSim is not a single piece of software; it is a carefully integrated collection of best-inclass open-source tools, connected by a Python and PyQt5-based shell that presents them as a unified application. KiCad handles schematic capture. It is one of the most capable open-source schematic editors available and provides eSim with a professional-grade environment for drawing circuits, assigning component values, and generating netlists the structured text files that describe a circuit's components and connections. NgSpice handles simulation. It is a mature, actively maintained implementation of the SPICE circuit simulation standard that has been the industry benchmark for over fifty years. Through NgSpice, eSim supports:

- Transient analysis: how a circuit responds over time to a time-varying input
- AC analysis: frequency response across a defined range of frequencies
- DC sweep: how the circuit's operating point changes with a swept voltage or current source
- Operating point analysis: steady-state DC voltages and currents across the circuit
- Noise analysis: noise contribution of individual components

After simulation, eSim provides Python-based tools for plotting waveforms and extracting quantitative results. The whole platform is free, cross-platform (Linux, Windows, macOS), and distributed under open-source licences. It is the primary circuit simulation tool recommended by the FOSSEE programme for undergraduate.

1.3 Objectives of the Project

The main purpose of this project was not simply to add a basic chatbot feature into eSim, but to create an intelligent assistant that could genuinely support the real users of the software, including students learning analogue electronics, researchers developing circuit prototypes, and teachers preparing simulation-based demonstrations. Keeping these users in mind, the project focused on achieving the following objectives:

Develop an integrated chatbot panel within eSim The assistant needed to function as a built-in part of the eSim application rather than an external tool. Users should be able to access the chatbot directly inside eSim without complicated setup procedures, apart from installing Ollama. The chatbot should also understand the user's current working environment and provide context-aware assistance. Ensure complete local and offline operation using Ollama Privacy and security were important considerations, especially for users working on confidential research projects or proprietary circuit designs. Therefore, the assistant was designed to run entirely on the user's local machine without sending any data to external servers. All conversations, files, and uploaded content remain private and offline. Enable schematic image understanding using vision-based AI models Many circuit-related problems are easier to explain visually than through text. The assistant was required to accept schematic images directly from the user, analyze them using vision-capable language models, and answer detailed questions related to the displayed circuit instead of only providing generic image descriptions. Create persistent chat history

with searchable session management Valuable discussions and troubleshooting sessions should not disappear after closing the application. The system was designed to automatically save every chat session, including both text and uploaded images. Users should be able to restore and search previous conversations easily through a sidebar interface. Add voice input support for improved usability During hardware testing or prototyping, users may not always be comfortable typing long questions. To improve accessibility and convenience, the project aimed to include voice input functionality so that users could communicate with the assistant using spoken language. Implement NgSpice netlist analysis capabilities The assistant should be capable of reading and understanding the actual NgSpice netlists generated by eSim. By analyzing these netlists directly, the chatbot can answer circuit-related questions more accurately without requiring users to manually rewrite or explain the circuit in text form. Identify and resolve bugs in the existing prototype system An earlier version of the chatbot integration already existed but suffered from several technical issues and reliability problems. One of the important objectives of this project was to perform detailed debugging, identify the root causes of errors, and implement stable fixes so the system could be used reliably in regular workflows.

1.4 Methodology

The work was carried out iteratively. The broad phases below overlapped in practice testing revealed bugs that required design changes, and design changes sometimes uncovered additional bugs. Phase 1 - Study and architecture: Reading the existing eSim codebase and prototype chatbot code thoroughly, followed by mentor discussions to agree on scope, priorities, and technical constraints. Key architectural decisions made here included the choice of Ollama as the inference backend, the two-file module structure (Chatbot.py for UI, chatbot thread.py for workers), and the QThreadbased concurrency model. Phase 2 - Core implementation: Building the chat panel, bubble rendering system, model selector, session save/load, and sidebar. Establishing the worker communication pattern through Qt signals before adding any complex features. Phase 3 - Feature development: Adding vision analysis, voice input, netlist analysis, the image staging strip, and drag-and-drop support. Each feature was built as a self-contained addition and tested in isolation before integration. Phase 4 - Bug investigation and fixing: A systematic code review of both source files, covering every method, every state variable, and every signal connection. Each bug was documented with its root cause before any fix was written. Phase 5 - Integration testing: End-to-end testing on Windows within the eSim virtual environment, covering text chat, image analysis, session deletion, window switching during generation, retry behaviour, voice transcription, and application startup.

Chapter 2

Literature Survey

2.1 Large Language Models:

The development of modern Large Language Models (LLMs) started with the introduction of the transformer architecture by Ashish Vaswani and his team in 2017. The transformer model introduced the concept of a self-attention mechanism, which allowed the system to analyze relationships between all words in a sentence at the same time. Unlike older recurrent neural networks that processed information sequentially from left to right, transformers could process entire sequences in parallel. This significantly improved training efficiency and enabled the model to better understand long-range relationships within text. A major breakthrough in the field came with the release of GPT-3 by OpenAI in 2020. Developed by Tom B. Brown and collaborators, GPT-3 contained 175 billion parameters and was trained on a massive collection of internet text. The model demonstrated the ability to perform tasks such as answering questions, generating human-like text, writing code, and solving reasoning problems without being specifically trained for each individual task. This unexpected capability, often referred to as emergent behaviour, changed the direction of AI research and showed the true potential of large-scale language models. Later, several open-weight language models such as LLaMA, Qwen, and Mistral demonstrated that smaller models could still achieve high performance when trained on better-quality datasets. Researchers including Hugo Touvron, Jinze Bai, and Albert Qiang Jiang contributed significantly to these advancements. These models generally contain between 3 and 13 billion parameters, making them efficient enough to run on consumer-grade hardware while still delivering strong performance across many tasks. Their smaller size and open availability make them highly suitable for local AI applications, including offline intelligent assistants integrated into engineering tools such as eSim.

2.2 Local LLM Inference:

Running Large Language Models locally on personal computers presents a major technical challenge because these models require a large amount of memory. For example, a model with 7 billion parameters stored in full 32-bit floating-point precision may require nearly 28 GB of memory, which is difficult to handle on standard consumer hardware. To solve this problem, researchers developed quantisation techniques that reduce the memory requirements of models by lowering the numerical precision of their weights.

One important contribution in this area was GPTQ, introduced by Elias Frantar and collaborators in 2022. GPTQ demonstrated that model weights could be compressed into 4-bit integer representations while maintaining most of the model's original performance. This reduction in precision significantly decreases memory usage and allows large models to run efficiently on ordinary hardware.

Another major advancement came from the llama.cpp project developed by Georgi Gerganov in 2023. This project combined efficient quantisation methods with an optimized C++ inference engine, making it possible to run quantised language models directly on CPUs without requiring powerful GPUs. The project also introduced the GGUF file format, which later became a widely accepted standard for distributing compressed language models.

Building on these technologies, Ollama provides an easy-to-use framework for managing and running local language models. Ollama handles tasks such as model downloading, storage management, quantisation selection, and GPU acceleration when available. It also offers a REST API for generating chat responses. One of its important features is streaming output, where generated tokens are sent progressively instead of waiting for the entire response to complete. This streaming capability improves responsiveness and creates a smoother real-time conversational experience for chatbot applications integrated into tools like eSim.

2.3 Vision-Language Models:

Traditional Large Language Models (LLMs) are designed to process only text-based input and cannot directly understand images. Vision-Language Models (VLMs) extend the capabilities of LLMs by combining image understanding with natural language processing. These models use a visual encoder, commonly based on architectures such as Vision Transformers (ViT) or CLIP models, to convert images into visual tokens. The language model then processes these visual tokens together with text tokens, allowing it to understand both visual and textual information simultaneously.

One of the early successful open-weight Vision-Language Models was LLaVA, introduced by Haotian Liu and collaborators in 2023. LLaVA demonstrated that even a relatively small 7-billion-parameter model could effectively analyze complex images and provide accurate answers to detailed visual questions. This showed that multimodal AI systems could perform advanced reasoning tasks involving both images and text.

Another important model in this area is Qwen-VL, developed by researchers including Jinze Bai. Qwen-VL achieved strong performance in understanding technical documents, engineering diagrams, charts, and structured tables. These capabilities make it especially useful for applications involving circuit schematics and electronic design analysis.

In the context of eSim, Vision-Language Models provide a significant improvement over traditional text-only assistants. Instead of requiring users to manually describe their circuit schematic in words, users can simply upload a screenshot of their design and ask questions directly about the visible components or circuit behaviour. The model analyzes both the schematic image and the user's query together, enabling more accurate, context-aware, and efficient assistance during circuit design and simulation tasks.

2.4 PyQt5 for Scientific Tool GUIs:

PyQt5 is the Python binding for the Qt framework, which is one of the most widely used and reliable cross-platform graphical user interface (GUI) frameworks available today. Qt provides a rich collection of tools and components for developing modern desktop applications. Features such as the widget-based interface system, signal-and-slot communication mechanism, multithreading support, and HTML rendering capabilities make it highly suitable for building complex scientific and engineering software applications. Since eSim itself is developed using PyQt5, it became the most practical and compatible choice for implementing the chatbot module within the existing application environment. Using the same framework ensured smooth integration between the chatbot interface and the main eSim system while maintaining consistency in design and functionality. A particularly important component used in the project was the QTextBrowser widget, which served as the primary display area for the chatbot conversation interface. QTextBrowser supports a large subset of HTML and CSS, allowing the development of visually styled chat messages directly inside the application. It also supports inline Base64-encoded images, enabling image previews to appear naturally within chat conversations. Additionally, QTextBrowser provides support for clickable hyperlinks through its signal callback mechanism. These features made it possible to create an interactive and user-friendly chat interface containing styled message bubbles, embedded image thumbnails, and clickable footer actions without requiring the integration of a separate web browser component. This lightweight yet powerful approach helped maintain performance while improving the overall user experience inside eSim.

2.5 Speech Recognition in Python:

SpeechRecognition is a widely used Python library that provides a simple and unified interface for implementing speech-to-text functionality in applications. The library supports multiple speech recognition backends, making it flexible for both online and offline voice processing systems. The library works together with PyAudio to access the system microphone and capture live audio input from the user. It also includes features such as automatic ambient noise adjustment, which helps improve recognition accuracy by calibrating background noise levels before recording begins. In addition, the library can automatically segment audio based on silence detection, allowing speech to be processed more efficiently and naturally. SpeechRecognition supports several recognition engines, including CMU Sphinx for completely offline speech recognition and the Google Web Speech API for higher-accuracy online speech processing. This combination of offline and online backends provides flexibility depending on the user's internet availability and system requirements. In this project, the SpeechRecognition library was used to add voice input functionality to the eSim chatbot assistant. This feature allows users to ask questions using spoken language instead of typing, making interaction with the assistant more convenient during hardware testing, circuit prototyping, or other situations where typing may be difficult. The dual-backend support also ensures that voice input remains functional in both internet-connected and fully offline environments.

Chapter 3

Problem Statement

3.1 Problem Statement

To understand why this project was needed, it helps to think about a specific situation. Imagine a second-year electrical engineering student who has just drawn a common-emitter amplifier in KiCad, generated the NgSpice netlist, and run a transient simulation. The output waveform is clipping where it should not be. The student has no idea why. In the current eSim, their options are: close eSim and search online, email a professor, or post on a forum and wait. None of these is fast, none is aware of the specific circuit causing the problem, and none keeps the student in their workflow. This is the central problem: eSim has no way for a user to ask a question about their circuit and receive an intelligent, contextual answer without leaving the application. This general problem has several concrete dimensions: No natural language interface. Every interaction with eSim is through menus, buttons, and dialogs. There is no way to describe what you want in plain language. This creates a high barrier for new users who do not yet know the tool's vocabulary or where to look for specific features. No schematic image understanding. Schematics are visual. Much of the knowledge needed to work with them recognising circuit topologies, understanding component arrangement, spotting a wiring error is inherently visual. A text-only interface cannot address questions about something the user is looking at on screen. No netlist analysis. A NgSpice netlist from a moderately complex design can run to hundreds of lines of SPICE syntax that is not particularly human-readable. There is no tool within eSim that reads a netlist and explains in plain language what it represents, what simulation it is set up to run, or what might be causing it to fail. No session memory. Every time eSim is opened, the user starts from a blank state. There is no record of previous questions and answers, no way to continue a line of inquiry started in a previous session. Existing prototype bugs. An early chatbot prototype existed but had several bugs serious enough to prevent reliable daily use:

- After sending an image, only a filename text badge appeared in the chat the actual image was never shown.
- If the user switched focus to another application window while the model was generating a response, the chat display became corrupted when they returned real messages were partially deleted.

• Deleting a session from the sidebar appeared to work, but the session reappeared the next time eSim was launched.

• The retry function produced two stacked responses in the chat instead of replacing the old one with a new one.

• The application crashed on startup with a TypeError before the window even appeared, because of invalid PyQt5 constructor arguments.

3.2 Approach

The development approach for this project focused on solving each issue individually through targeted and minimal modifications. Instead of redesigning the entire system, the goal was to identify the root cause of every problem and implement precise fixes that improved reliability without creating additional issues elsewhere in the application.

To implement the chatbot interface, the existing PyQt5 window architecture of eSim was extended. The chatbot was designed as a separate floating panel integrated directly into the main eSim interface, allowing users to interact with the assistant without leaving the application environment.

For local Large Language Model inference, Ollama was selected as the backend platform because of its efficient streaming REST API, support for multiple open-weight models, compatibility with CPU-based systems, and fully local execution. Since Ollama operates entirely on the user's machine, it also ensured complete privacy by preventing external data transmission.

One important issue involved incorrect image handling inside the chat interface. To resolve this problem, the image encoding process was moved before the chat display update stage. Additionally, the previous filename-based image badge system was replaced with inline Base64 image rendering using the existing image thumbnail HTML rendering functionality already used for restoring previous sessions.

Another critical issue occurred when users switched windows during the chatbot typing animation, causing display corruption. Previously, the typing animation relied on fragile integer-based character positions that became invalid after document reflow. This was corrected by replacing the position tracking system with named HTML anchor tags, which are preserved safely within Qt document fragment metadata even when the document structure changes dynamically.

The session deletion system also contained a bug where pending autosave operations could interfere with file removal. To solve this issue, both the delete-session handler and the clear-session function were updated to stop the debounce timer and reset the pending-save flag before deleting session files from storage.

The retry functionality for chatbot responses was redesigned as well. The original status-bar retry button was removed and replaced with an inline "Retry" link embedded directly inside each chatbot response bubble. A dedicated retry method was implemented to trim the chat history up to the selected response, rebuild the visible conversation without outdated messages, and automatically restart either the text-based or vision-based worker depending on whether the previous user message included an image.

Finally, the application startup crash was resolved by removing unsupported constructor keyword arguments that were incompatible with PyQt5 components. These invalid parameters were replaced with proper PyQt5 property setter methods, ensuring stable initialization and preventing runtime crashes during application launch.

Chapter 4

Implementation

4.1 Case 1: Chatbot UI Architecture and Vision Image Analysis:

4.1.1 Module Structure

The chatbot system was designed using a modular architecture divided across two separate Python source files. A strict separation was maintained between the graphical user interface and background processing components to ensure that no blocking operations were executed on Qt's main UI thread. This design helped maintain application responsiveness and prevented the interface from freezing during long-running tasks such as model inference or speech recognition.

The first file, Chatbot.py, contains the complete PyQt5 user interface layer. It includes the main ChatbotGUI window class, the SessionSidebar widget for displaying saved chat sessions, the ChatHistoryViewer dialog for reviewing previous conversations, and a custom HistoryLineEdit input component that supports command-style up and down history navigation. This file also contains all functions responsible for generating styled HTML chat bubbles displayed within the interface. Since this module operates entirely on the Qt main thread, it does not perform any blocking input/output operations directly.

The worker classes implemented in this module include:

OllamaWorker This worker sends text-based chat history to the Ollama /api/chat endpoint and receives streamed responses token by token from the language model. OllamaVisionWorker This worker processes one or more Base64-encoded images together with a text prompt and sends them to a vision-capable Ollama model for multimodal analysis and response generation. MicWorker This worker handles microphone input and converts spoken audio into text using either CMU Sphinx for offline recognition or the Google Web Speech API for higher-accuracy online transcription. OllamaStatusWorker This worker continuously checks whether the local Ollama server is active and emits status signals so that the interface can display a live online/offline connection indicator.

Communication between the worker threads and the graphical user interface is

performed using Qt's signal-and-slot mechanism. Worker threads emit Qt signals whenever results, status updates, or generated responses are available. Since Qt signals are thread-safe, this approach provides a reliable and safe method for transferring data from background threads back to the main UI thread without causing synchronization issues or application instability.

4.1.2 Chat Bubble Rendering:

The chatbot conversation interface is implemented using the QTextBrowser widget provided by PyQt5. Instead of using a separate browser engine, Qt's built-in HTML rendering capabilities were used to display styled chat messages directly inside the application. Each message is dynamically generated as an HTML table fragment combined with inline CSS styling and then appended to the QTextBrowser document.

The chat interface was designed using three different bubble types to clearly distinguish between user messages, AI responses, and system notifications.

User Bubble

User messages are displayed as right-aligned chat bubbles with a blue gradient background. These bubbles contain the original text entered by the user along with a small timestamp indicating when the message was sent. The right alignment helps visually separate user input from chatbot responses.

Bot Bubble

Chatbot responses are displayed as left-aligned bubbles with a light grey background. Before rendering, the generated response text is processed through a Markdown renderer that supports formatting features such as bold text, italic text, inline code formatting, headings from H1 to H4, and clickable hyperlinks. This formatting improves readability and allows technical explanations to be displayed more clearly.

The footer section of each bot bubble contains additional metadata, including the response timestamp, an estimated token count, and interactive action links such as "Retry" and "Copy". These actions are implemented using custom URL schemes like retry:///idx and copy:///idx. The QTextBrowser widget intercepts these custom links through its anchorClicked signal, allowing the application to trigger internal functions without requiring external browser navigation.

System Bubble

System-generated notifications are displayed as centered bubbles with an amber-colored background. These bubbles are used for status-related messages such as notifying the user when the Ollama server starts, when the chatbot switches to a vision model, or when the connection to the backend is lost. The centered alignment and distinct color scheme help users easily identify system-level information separately from normal chat interactions.

4.1.3 Image Attachment and Vision Analysis

The vision analysis feature allows users to upload schematic images directly into the chatbot and ask questions related to the displayed circuit. This workflow begins when the user either clicks the attachment button or drags and drops an image file onto the chatbot window. Once selected, the attached images are displayed as small

preview thumbnails in a horizontal staging area positioned above the text input field. Users also have the option to remove any image from the staging strip before sending the request. When the user clicks the Send button, the following sequence of operations is performed:

Image Processing and Resizing Each selected image is first read from disk. If the Pillow library is available, the image is automatically resized to a maximum resolution of 320×240 pixels. This resolution is sufficient for vision-language models to identify component labels, wire names, and circuit symbols while keeping the request size manageable for efficient local inference.

Base64 Encoding and Session Storage The processed image data is converted into Base64-encoded format and stored inside a session-level dictionary structure. Each image is indexed using a timestamp value so that it can later be correctly linked to the associated conversation message when restoring previous chat sessions.

Inline Image Rendering in Chat Display To provide immediate visual feedback, the application inserts the uploaded image directly into the chat interface using an inline HTML ¡img¿ tag with a Base64 data URI. This allows users to confirm exactly which image has been attached and sent to the AI model without relying on external file references.

Launching the Vision Worker Thread After the image preparation step is complete, an OllamaVisionWorker thread is started. This worker receives the image paths, the user's text prompt, and the currently selected vision-capable model name as input parameters.

The OllamaVisionWorker communicates with the Ollama /api/chat endpoint using a multimodal request format. In this request, each uploaded image is transmitted as a Base64-encoded attachment together with the user's textual question. The system prompt and prompt-generation logic are specifically designed so that the user's actual question remains the primary instruction to the model. This approach enables the assistant to focus on answering specific circuit-related queries based on the visible schematic content rather than generating only generic image descriptions.

The vision system prompt defines the chatbot's behaviour while analyzing schematic images. The assistant is instructed to behave as an expert electronics engineer integrated inside eSim. The prompt directs the model to carefully read all visible information from the uploaded schematic image, including component labels, reference designators, values, pin numbers, and net names. The system prompt also defines several important behavioural rules:

As a result, the vision model often ignored the actual question and repeatedly generated the same generic multi-section component analysis regardless of what the user asked. The updated design solved this issue by making the user's question the primary instruction within the prompt hierarchy. This significantly improved response relevance, allowing the assistant to provide focused, context-aware answers directly related to the uploaded schematic and the user's intended query.

This design matters because the previous implementation appended the user's question as a secondary "Additional question from the user" tag at the end of a long schematic analysis instruction. The model would prioritise the detailed primary instruction and produce a generic six-section component dump regardless of what the

user had actually asked.

4.1.4 Typing Animation with Window-Switch Safety

During response generation, the chatbot interface displays an animated typing indicator consisting of three dots that continuously cycle through different fill patterns. This animation provides visual feedback to users that the AI model is actively generating a response. The animation is controlled using a QTimer that triggers updates every 400 milliseconds. On each timer event, the typing indicator bubble is updated by modifying the underlying QTextDocument associated with the QTextBrowser. These updates are performed using QTextCursor operations, which allow sections of the HTML-rendered document to be modified dynamically.

However, this approach introduced a serious reliability issue. When the user switched focus away from the application window, Qt's HTML rendering engine automatically reflowed the QTextBrowser document during repaint operations. Document reflow changes the internal character positions of rendered content. As a result, the previously stored integer position no longer pointed to the correct location within the document. When the next timer update occurred, the cursor sometimes pointed into the middle of existing chat messages instead of the typing indicator region. The update operation would then accidentally overwrite or delete parts of the conversation history. Users returning to the application after switching windows could therefore find sections of the chat missing or corrupted. To solve this issue, the typing indicator system was redesigned using a stable HTML anchor-based positioning mechanism. Instead of tracking the typing indicator through fragile integer offsets, a named HTML anchor tag was inserted into the document as a permanent marker. Qt internally preserves anchor metadata even when document reflow occurs, making the anchor location stable across repaint and layout updates. Using this approach, the typing animation can reliably locate and update the correct section of the document without risking modification of surrounding chat content. This fix completely eliminated the conversation corruption issue caused by window switching and significantly improved the stability of the chat rendering system.

Listing 4.4: Position-independent anchor search (Chatbot.py)

```
def _find_typing_anchor_cursor(self):
doc = self.chat_display.document()
block = doc.begin()
while block.isValid():
it = block.begin()
while not it.atEnd():
frag = it.fragment()
if frag.isValid():
fmt = frag.charFormat()
try:
# PyQt5 API: anchorNames() returns a list
names = fmt.anchorNames()
```

```
matched = "_typing_anchor_" in (names or [])
except AttributeError:
matched = False

if matched:
cursor = QTextCursor(doc)
cursor.setPosition(frag.position())
return cursor
it += 1
block = block.next()
return None
```

Qt stores anchor names in the document's fragment-level metadata rather than encoding them as character content. This means they are immune to position shifts caused by reflow. The search by anchor name finds the correct fragment regardless of what character offset it now sits at, and the replacement proceeds safely. As a secondary improvement, the auto-scroll was changed so that the chat only scrolls to the bottom during animation ticks if the user is already within 60 pixels of the bottom. Previously it forced a scroll to the bottom on every tick, which would interrupt the user if they scrolled up to re-read an earlier message while waiting for a response.

4.1.5 Session Persistence and the Sidebar

The chatbot system includes a persistent session management feature that automatically saves every conversation for future access. All chat sessions are stored inside a dedicated sessions directory located within the user's home folder. Each conversation is saved as a self-contained JSON file, allowing sessions to be restored completely even after the application is closed. Every session file contains several important pieces of information, including:

A universally unique identifier (UUID) used to uniquely identify the session

A human-readable session title automatically generated from the user's first message

Creation and last-modified timestamps

The complete chat message history, limited to the most recent 40 messages

All images associated with the session stored as Base64-encoded strings

The model configuration settings used during the session, including model name, temperature, and token limit values

One important design decision was storing images directly inside the session file using Base64 encoding instead of saving only file paths. This approach ensures that sessions remain fully portable and self-contained. Even if the original image files are deleted, renamed, or moved from their original locations, the chatbot can still reconstruct the complete conversation with all inline image thumbnails exactly as they originally appeared. To improve performance and reduce unnecessary disk activity, the session-saving mechanism uses a debounced autosave strategy implemented with a QTimer. Instead of writing session data to disk after every individual message update, the system resets a five-second timer whenever new activity occurs. The actual disk write operation only takes place if no additional updates arrive dur-

ing the five-second interval. This approach minimizes input/output overhead and prevents performance issues during response streaming, where the language model may generate many tokens every second. The chatbot also includes a dedicated SessionSidebar widget for browsing saved conversations. The sidebar displays all stored sessions along with their titles, creation dates, and the total number of messages contained within each session. A real-time search bar positioned at the top of the sidebar allows users to quickly filter sessions while typing. When a user selects a session from the sidebar, the entire conversation is reconstructed inside the chat interface. All text messages, styled chat bubbles, timestamps, and inline image thumbnails are restored sequentially so that the conversation appears exactly as it did during the original interaction. This feature provides users with a seamless way to revisit previous discussions, troubleshooting sessions, and schematic analyses at any time.

4.1.6 Voice Input

The MicWorker background thread manages the complete voice input pipeline: 1. Opens the default microphone using PyAudio and samples ambient noise for one second to calibrate the silence threshold.

2. Records audio until a pause in speech is detected (typically 1–2 seconds of silence).

3. Attempts transcription via CMU Sphinx, which runs entirely offline using a local acoustic model.

4. If Sphinx fails or is not installed, falls back to the Google Web Speech API,which requires an internet connection but achieves significantly higher accuracy, particularly for technical vocabulary.

5. Emits the transcribed text string via a Qt signal back to the main thread, which inserts it into the text input field where the user can review and edit it before sending.

The microphone button in the UI changes appearance while recording to provide clear visual feedback, and the status bar shows a recording indicator.

4.1.7 NgSpice Netlist Analysis

The analyse netlist() method is called by eSim when the user invokes the chatbot with a netlist file loaded. The method does the following: 1. Reads the netlist file and splits it into lines.

2. Parses the lines to extract component references, grouping them by type: resistors (R), capacitors (C), inductors (L), diodes (D), transistors (Q), integrated circuits and subcircuits (U, X), voltage and current sources (V, I).

3. Collects all unique node names that appear in the netlist (excluding ground, node 0).

4. Identifies SPICE analysis directives: .tran, .ac, .dc, .op, .noise, and any .model or .include statements.

5. Constructs a structured plain-language summary of all of the above.

6. Sends this summary to the LLM along with the full netlist text (truncated at 80 lines for very large designs) and a prompt asking it to: describe what each component does in the context of the circuit, explain the overall circuit function, interpret the simulation directives, and flag any potential issues such as missing ground connections, floating nodes, or unusual parameter values.

4.2 Case 2: Bug Fixes and Reliability Improvements

The second major strand of this project was identifying and eliminating all bugs in the existing prototype. This section documents each bug precisely: what the user experienced, what was actually causing it, and how it was fixed.

4.2.1 Bug 1 – Images Not Displayed After Sending

What the user observed

After attaching an image and clicking Send, the chat showed only a small filename badge such as "img.jpeg". The actual image thumbnail did not appear in the conversation window.

Root cause

The problem occurred because the image-related operations were executed in the wrong sequence. The function staged images bubble was called first to update the chat display, but this function only created a text badge containing the filename. The image was encoded afterward for storage purposes. Although the function already existed and correctly generated inline Base64 thumbnails during session replay, it was never called during live message sending.

Fix

The image encoding step was moved before the chat display update. After encoding, the function image thumbnail is now called for each attached image, which inserts an actual inline thumbnail directly into the chat bubble. The filename badge is still kept as a fallback in situations where image encoding fails for all attached images.

4.2.2 Bug 2 – Chat Corruption During Window Switching

What the user observed

While the model was generating a response, switching to another application window and then returning to the chatbot sometimes caused the chat display to become corrupted. Portions of earlier conversation messages disappeared, or the animated typing indicator appeared in place of actual chat content.

Root cause

The typing animation system tracked the location of the animation bubble using an integer character offset variable called typing start pos inside the QText Browser document. Every 400 milliseconds, the animation timer called cursor.set Position and selected everything from that position to the end of the document, replacing the selection with the next animation frame.

The problem occurred when the user switched to another application window. During repaint handling, Qt's HTML layout engine automatically reflowed the QTextBrowser document. This reflow changed the internal character positions of document content. Although the stored integer offset was initially correct, after reflow it pointed to a completely different location, often inside an existing chat message bubble. As a result, the next animation update selected and removed real conversation content instead of updating only the typing indicator.

Fix

To solve the issue, the integer-based tracking mechanism was replaced with a stable named HTML anchor sentinel. When the typing animation bubble is first inserted into the document, the string ¡a name="typing anchor"¿ is placed immediately before it.

Qt's HTML parser stores anchor names inside the document fragment metadata layer instead of treating them as normal character content. Because of this, the anchor remains stable even when document reflow changes character positions.

Whenever the animation timer needs to update or remove the typing indicator, the application scans the document fragments and searches for a fragment whose anchorNames() list contains the sentinel anchor name. Once found, a new cursor is constructed using the fragment's actual current position inside the document. This ensures the correct location is always identified regardless of how many times the document has been reflowed.

The main PyQt5 API used for this solution is QTextCharFormat.anchorNames(), which returns the list of anchor names associated with a document fragment. In C++ Qt, the corresponding method is named anchorName() in singular form, while PyQt5 exposes it as anchorNames() returning a list. The implementation handles both forms defensively to maintain compatibility across different PyQt5 versions.

4.2.3 Bug 3 - Deleted Sessions Reappearing After Restart

What the user saw: Clicking the delete button on a session in the sidebar appeared to work. The session disappeared from the list. But after closing eSim and reopening it, the deleted session was back in the sidebar.

Root cause: Session saves are debounced with a five-second timer. When th euser sent a message, the timer was set. If within five seconds of that last message the user deleted the session, the delete handler correctly removed the session JSON

file from disk. But the debounce timer was still counting. When it fired, it called flush save(), which called save current session(), which wrote the session back to disk recreating the file that had just been deleted. On the next application launch, the sidebar scanned the sessions directory, found the recreated file, and displayed it again.

Fix: Both the session delete handler (on session deleted) and the clear session function (clear session) now immediately stop the debounce timer and clear the pending-save flag as the very first action before any file removal:

Listing 4.5: Debounce cancellation in on_session_deleted (Chatbot.py)

```
def _on_session_deleted(self, deleted_id: str):
if deleted_id == self._current_session_id:
self._abort_worker()

# Cancel the debounce timer immediately.
# Without this, _flush_save() fires after
# 5-second window and recreates the file we are
# about to delete, making it reappear on next launch.
self._save_debounce_timer.stop()
self._save_pending = False

# Now safe to remove the session file.
session_file = os.path.join(
_SESSIONS_DIR,
f"{self._current_session_id}.json"
)

try:
if os.path.exists(session_file):
os.remove(session_file)
except Exception:
pass

# ... reset all session state ...
```

4.2.4 Bug 4 - Retry Producing Duplicate Responses

What the user saw: The Retry button in the status bar was only visible after error responses (responses starting with an error indicator). When it did appear and the user clicked it, a new response was generated but it appeared below the old one, leaving two bot responses stacked in the chat for the same user question. Additionally, retrying an image analysis session always used the text worker, producing irrelevant results. Root cause: Three separate problems: 1. The status-bar button's show/hide logic only triggered for error-prefixed responses, making it invisible for normal responses the user might want to retry. 2. The retry method retry last()

launched the LLM worker without first removing the previous bot response bubble from the chat display. The new response simply appended below the old one. 3. The method always constructed an OllamaWorker (text-only) regardless of whether the original query had included images, so image analysis retries produced text-only answers. Fix: The status-bar QPushButton was removed entirely. A "Retry" hyperlink is now embedded in the footer row of every bot bubble, using a custom URL scheme:

4.2.5 Bug 5 - Application Crash on Startup

What the user saw: The application crashed immediately when launched with: TypeError: 'fixedSize' is an unknown keyword argument The eSim window never appeared. Root cause: Several widget constructors in Chatbot.py passed Qt property names (fixedSize, fixedHeight) as Python keyword arguments to the constructor. In Qt's C++ API these are writable properties, but PyQt5 does not map them as constructor parameters. PyQt5 raises a TypeError immediately for any unrecognised keyword argument, so the application could not start at all. Six widgets were affected: the attachment button, the microphone button, both close buttons (in the sidebar and the history viewer), the image thumbnail card widget, and thethumbnail remove button.

Fix: The invalid keyword arguments were removed from all constructor calls.The size values were applied using the correct PyQt5 property setter methods immediately after construction:

Listing 4.7: Corrected PyQt5 widget sizing - before and after

```
# Before (crashes on import):
self.attach_button = QPushButton(icon, fixedSize=QSize(38,38))
self.mic_button    = QPushButton(icon, fixedSize=QSize(38,38))

# After (correct PyQt5 pattern):
self.attach_button = QPushButton(icon)
self.attach_button.setFixedSize(38, 38)

self.mic_button = QPushButton(icon)
self.mic_button.setFixedSize(38, 38)
```

Chapter 5

Conclusion and Future Scope

Conclusion

At the beginning of this internship, the chatbot system existed only as an unstable prototype that crashed before the application window could even open. By the end of the project, it had been transformed into a fully functional AI assistant integrated directly into eSim. The final system is capable of maintaining multi-turn conversations using locally running Large Language Models, analysing schematic screenshots through vision-language models, answering circuit-related questions, saving and restoring chat sessions with embedded images, accepting voice input, and interpreting NgSpice netlists. Most importantly, the entire system operates completely offline, ensuring that no user data is transmitted outside the local machine.

A major part of the project involved identifying and resolving critical bugs in the existing implementation. This debugging work became one of the most important contributions of the internship because software features are only valuable when they function reliably and safely. Each issue investigated during the project had a clear technical root cause, and understanding these causes in depth allowed stable and long-term solutions to be implemented instead of temporary fixes or workarounds.

Future Scope

The developed chatbot system already provides a stable and functional foundation for intelligent assistance inside eSim. However, there are several possible improvements and extensions that could further enhance its capabilities and usefulness for students, researchers, and professional engineers.

eSim-Specific Fine-Tuned Language Model

Currently, the chatbot uses general-purpose Large Language Models combined with carefully designed system prompts. Although this approach performs well, a model specifically fine-tuned on eSim documentation, the NgSpice manual, and the FOSSEE sub-circuit library could provide much more accurate and domain-specific responses. Since the required training material already exists, future work could focus on preparing datasets and performing supervised fine-tuning to create a dedicated eSim AI model.

Simulation Output Analysis

At present, the chatbot can analyze circuit netlists and schematic images, but it

does not directly interpret simulation output results. A future enhancement could allow the assistant to read NgSpice .raw output files after simulation execution and answer analytical questions related to circuit performance. For example, the assistant could help explain issues such as unexpected gain reduction, bandwidth limitations, waveform distortion, or frequency response characteristics. This feature would make the chatbot significantly more useful for practical engineering workflows.

Agentic Actions Inside eSim

Another important extension would involve transforming the chatbot from a passive assistant into an active design agent capable of modifying circuits directly inside eSim. Instead of only answering questions, the assistant could perform actions such as adding missing decoupling capacitors, changing resistor values, correcting simulation parameters, or updating circuit configurations based on natural language instructions provided by the user. Implementing this capability would require deeper integration with eSim's internal circuit representation and editing system.

Persistent Multi-Turn Image Context

The current implementation maintains image context for only a limited number of follow-up interactions after an image is uploaded. Future improvements could introduce a more advanced context-management system that preserves image embeddings and visual references across multiple conversational turns. This would enable users to have longer and more natural discussions about a particular schematic without repeatedly uploading the same image.

Chapter 6

Bibliography

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention Is All You Need. Advances in Neural Information Processing Systems (NeurIPS), Vol. 30, pp. 5998–6008. Available at: https://arxiv.org/abs/1706.03762

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., et al. (2020). Language Models are Few-Shot Learners. Advances in Neural Information Processing Systems (NeurIPS), Vol. 33, pp. 1877–1901. Available at: https://arxiv.org/abs/2005.14165

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T.,et al. (2023). LLaMA: Open and Efficient Foundation Language Models. arXiv preprint arXiv:2302.13971. Available at: https://arxiv.org/abs/2302.13971

Bai, J., Bai, S., Chu, Y., Cui, Z., Dang, K., Deng, X., et al. (2023). Qwen Technical Report. arXiv preprint arXiv:2309.16609. Available at: https://arxiv.org/abs/2309.16609

Liu, H., Li, C., Wu, Q., and Lee, Y. J. (2023). Visual Instruction Tuning. Advances in Neural Information Processing Systems (NeurIPS), Vol. 36, pp. 34892–34916. Available at: https://arxiv.org/abs/2304.08485

Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. (2022). GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. arXiv preprint arXiv:2210.17323. Available at: https://arxiv.org/abs/2210.17323

Gerganov, G. (2023). llama.cpp: Inference of Meta's LLaMA model in pure C/C++. GitHub repository. Available at: https://github.com/ggerganov/llama.cpp [Accessed May 2026]

Blocklove, J., Garg, S., Karri, R., and Pearce, H. (2023). Chip-Chat: Challenges and Opportunities in Conversational Hardware Design. arXiv preprint arXiv:2305.11973. Available at: https://arxiv.org/abs/2305.11973 FOSSEE Team, IIT Bombay. eSim: An Open Source EDA Tool for Circuit Design, Simulation and PCB Design. Avail-

able at: https://esim.fossee.in [Accessed May 2026]

Ollama (2023). Ollama: Get up and running with large language models locally. Available at: https://ollama.com [Accessed May 2026]

Riverbank Computing Limited (2023). PyQt5 Reference Guide. Available at: https://www.riverbank[Accessed May 2026]

Kennedy, A. (2016). SpeechRecognition: Speech recognition library for Python. Python Package Index (PyPI). Available at: https://github.com/Uberi/speechrecognition[AccessedM
H. and Ngspice Development Team (2024). Ngspice User's Manual - Version42. Available at: http://ngspice.sourceforge.net/docs.html [Accessed May 2026]

KiCad Development Team (2024). KiCad EDA: Open Source Electronics Design Automation. Available at: https://www.kicad.org [Accessed May 2026]

Clark, A. and the Pillow Development Team (2024). Pillow: The friendly PIL fork. Python Package Index (PyPI). Available at: https://python-pillow.org [Accessed May 2026]