Semester Long Internship Spring 2026

On

eSim Chatbot Development

Submitted by

Harshit Singh Negi
VIT Bhopal

Under the guidance of

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

June 3, 2026

Acknowledgment

Sincere gratitude is expressed to **Prof. Prabhu Ramachandran** for providing the opportunity to participate in the FOSSEE internship programme and for his sustained commitment to the advancement of open-source engineering education in India.

Acknowledgment is also extended to **Prof. Kannan M. Moudgalya** for his foundational contributions to the FOSSEE initiative and for establishing the academic framework through which this internship was undertaken.

Sincere appreciation is owed to **Sumanto Kar** (Mentor) for his continuous technical guidance and feedback throughout the duration of this project, and to **Mr. Varad Patil** and **Ms. Shanthi Priya K** (Internal Mentors) for their coordination and technical inputs at each project phase.

Gratitude is also extended to the entire FOSSEE team for their coordination, resource accessibility, and timely support.

Contents

Chapter 1

Introduction

1.1 Background

Large language models (LLMs) AI systems trained on large corpora of text to understand and generate natural language have made it practical to embed conversational assistants directly into domain-specific software tools. This capability is particularly relevant in Electronic Design Automation (EDA), where users must simultaneously reason about component specifications, SPICE syntax, simulation parameters, and circuit behaviour.

The **FOSSEE** (Free and Open Source Software for Education) project at IIT Bombay develops and distributes free engineering tools for undergraduate education across India. Its primary EDA tool, **eSim**, is a complete open-source environment for analogue, digital, and mixed-signal circuit design, built on KiCad and NgSpice. Prior to this project, eSim provided no built-in intelligent assistance; users requiring help were required to exit the application and consult external documentation.

This project integrates an AI assistant directly into the eSim application, enabling in-tool, context-aware circuit design support without any external data transmission.

1.2 Overview of eSim

eSim integrates the following open-source components under a unified Python/PyQt5 shell:

- **KiCad** schematic capture, component assignment, and netlist generation.
- **NgSpice** SPICE-standard circuit simulation, supporting:
  - Transient analysis (`.tran`)
  - AC frequency response (`.ac`)
  - DC sweep (`.dc`)
  - Operating point (`.op`)
  - Noise analysis (`.noise`)
- Python-based post-processing tools for waveform plotting and result extraction.

eSim is cross-platform (Linux, Windows, macOS) and distributed under open-source licences.

1.3 Project Objectives

label=**1.** Integrate a native chatbot panel within the eSim PyQt5 application.

lbbel=**2.** Enable fully offline, local LLM inference via Ollama with no external data transmission.

lcbel=**3.** Implement schematic image analysis using vision-capable models (VLMs).

ldbel=**4.** Develop a persistent session management system with a searchable sidebar.

lebel=**5.** Add voice input support via the `SpeechRecognition` library.

lfbel=**6.** Implement NgSpice netlist parsing and plain-language circuit analysis.

lgbel=**7.** Identify, document, and resolve all bugs in the existing prototype codebase.

1.4 Methodology

The project was executed in five sequential phases:

Phase	**Description**
1. Architecture	Codebase review; architectural decisions (Ollama backend, two-file module structure, `QThread` concurrency model).
2. Core Implementation	Chat panel, bubble rendering, model selector, session save/load, sidebar.
3. Feature Development	Vision analysis, voice input, netlist analysis, image staging strip, drag-and-drop.
4. Bug Investigation	Systematic code review; each bug documented with root cause prior to applying any fix.
5. Integration Testing	End-to-end testing on Windows inside the eSim virtual environment.

Chapter 2

Literature Survey

2.1 Large Language Models

The transformer architecture introduced the self-attention mechanism, enabling parallel processing of entire sequences and effective capture of long-range dependencies. GPT-3 demonstrated emergent few-shot capability at 175B parameters. Subsequent open-weight models LLaMA, Qwen, and Mistral achieved comparable task performance at 3–13B parameters, making local deployment on consumer hardware viable.

2.2 Local LLM Inference

A 7B parameter model at full float32 precision requires approximately 28 GB of memory. GPTQ quantisation reduces weights to 4-bit integers with minimal quality loss. The `llama.cpp` project combines quantisation with a C++ inference engine for CPU-based execution, introducing the GGUF model format. Ollama builds on this foundation, exposing a streaming REST API (`/api/chat`) that returns generated tokens incrementally essential for a responsive chat interface.

2.3 Vision-Language Models

Vision-language models (VLMs) extend transformer-based LLMs with a visual encoder (typically a Vision Transformer or CLIP model) that converts input images to visual tokens, which the language model attends to alongside text tokens. LLaVA demonstrated strong image-understanding at 7B parameters. Qwen-VL showed particularly high performance on technical documents and engineering diagrams, making it suitable for schematic analysis tasks.

2.4 PyQt5 for Scientific GUIs

PyQt5 provides Python bindings for the Qt framework. The `QTextBrowser` widget, used as the primary chat display surface, renders a substantial subset of HTML/CSS,

supports inline base64-encoded images, and exposes anchor-click events via the `anchorClicked` signal enabling styled chat bubbles with interactive inline links without requiring a full embedded browser component.

2.5 Speech Recognition

The Python `SpeechRecognition` library provides a unified interface to multiple speech-to-text backends. It manages microphone access via `PyAudio`, performs ambient noise calibration, and supports both CMU Sphinx (fully offline) and the Google Web Speech API (online, higher accuracy).

Chapter 3

User Guide

This chapter describes how to install, launch, and use the eSim AI chatbot. All steps are documented in the order a first-time user would encounter them.

3.1 Prerequisites

Before launching the chatbot, ensure the following are installed on your system:

- **eSim** (version 2.x or later) installed and working.
- **Python 3.8+** with the eSim virtual environment activated.
- **Ollama** installed and running locally. Download from `https://ollama.com` and follow the platform installer.
- At least one Ollama model pulled. For text chat:

  ```
  ollama pull qwen2.5-coder:3b
  ```

  For schematic image analysis, a vision-capable model is required:

  ```
  ollama pull qwen2.5-vl:3b
  ```

- (Optional) **PyAudio** and **SpeechRecognition** for voice input:

  ```
  pip install pyaudio SpeechRecognition
  ```

- (Optional) **Pillow** for image downscaling before sending to the model:

  ```
  pip install Pillow
  ```

3.2 Launching the Chatbot

1. Open a terminal (or PowerShell on Windows) and activate the eSim virtual environment:

```
# Windows
.\venv\Scripts\Activate.ps1

# Linux / macOS
source venv/bin/activate
```

2. Start Ollama in the background (if it is not already running):

```
ollama serve
```

3. Launch eSim:

```
cd src
python -m frontEnd.Application
```

4. In the eSim main window, click the **AI Chatbot** button in the toolbar to open the chatbot panel.

3.3 Chatbot Interface Overview

Once the chatbot window opens, the interface is divided into four areas:

Area	**Description**
Top bar	Model selector dropdown, connection status indicator (Live / Offline), and toolbar buttons (new chat, history, settings).
Chat display	Scrollable message area showing user bubbles (right, blue) and bot bubbles (left, grey).
Image strip	Thumbnail preview area that appears above the input field when images are attached.
Input row	Text input field, attach image button (), microphone button (), Send button, and Clear button.

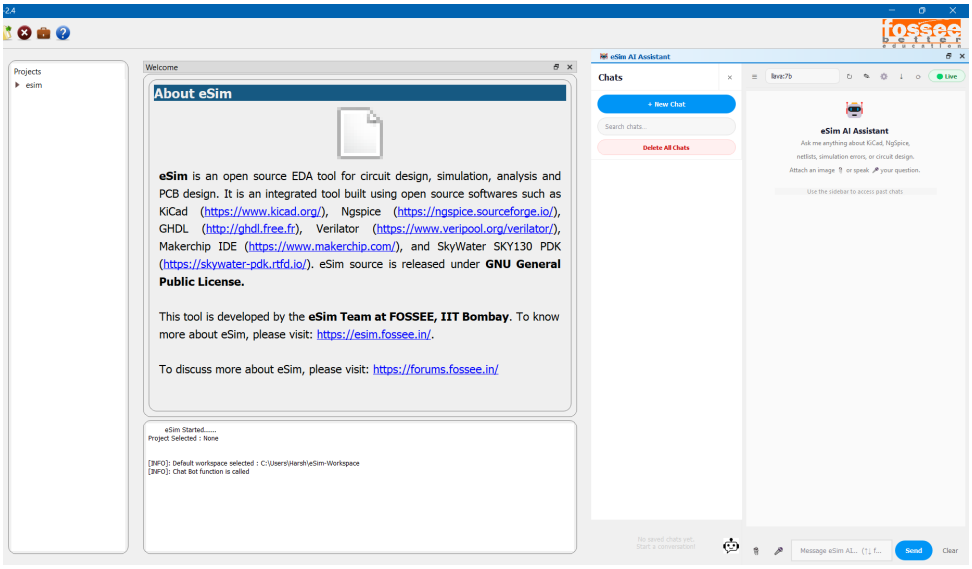


Figure 3.1: chatbot interface showing the four main areas.

3.4 Sending a Text Message

1. Click inside the input field at the bottom of the window.
2. Type your question. For example: *"How do I set up a transient analysis in NgSpice?"*
3. Press **Enter** or click **Send**.
4. The model begins generating a response immediately. Tokens appear in the chat one by one as they are produced.

Use the **up** and **down** arrow keys in the input field to navigate through previously sent messages.

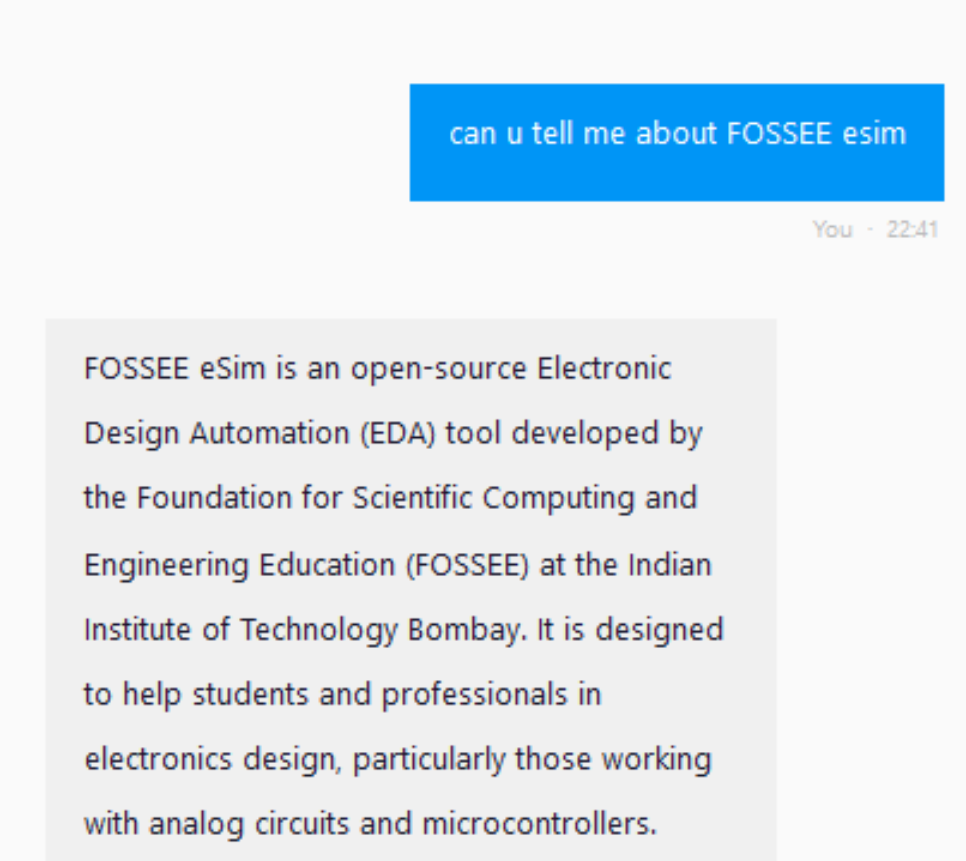


Figure 3.2: A text query and streamed response in the chatbot panel.

3.5 Analysing a Schematic Image

1. Take a screenshot of your schematic in KiCad or eSim and save it as a `.png` or `.jpg` file.
2. Click the (attach) button in the input row, or drag and drop the image file directly onto the chatbot window.
3. The image appears as a thumbnail in the staging strip above the input field. Multiple images can be attached at once.
4. Type your question about the schematic in the input field. For example: *"What is the function of this circuit and are there any design issues?"*
5. Click **Send**. The image is displayed in the chat and the vision model analyses it alongside your question.

To remove an image from the staging strip before sending, click the × button on its thumbnail.

eSim AI Assistant

llava:latest

Live

eSim AI Assistant

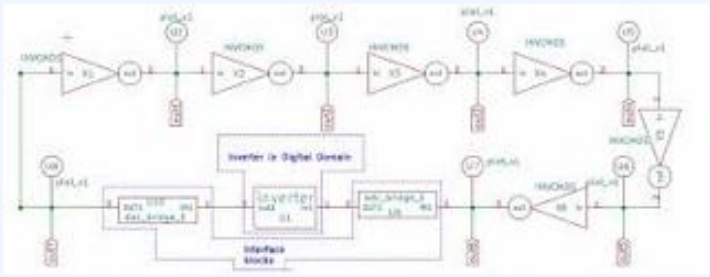
Ask me anything about KiCad, NgSpice,

netlists, simulation errors, or circuit design.

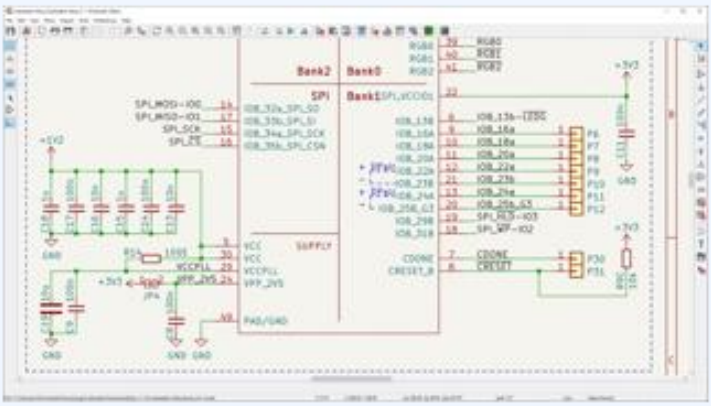
Attach an image or speak your question.

Use the sidebar to access past chats

Switched to vision model: llava:latest



images2.jpeg



img.jpeg

explain each

You · 12:08

3.6 Using Voice Input

1. Ensure a microphone is connected and PyAudio is installed (see Section 3.1).
2. Click the (microphone) button. The button changes colour to indicate that recording has started.
3. Speak your question clearly. Recording stops automatically after a 1–2 second pause in speech.
4. The transcribed text is inserted into the input field. Review and edit it if needed, then press **Send**.

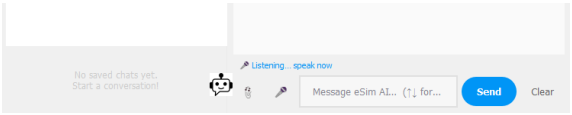


Figure 3.4: Microphone button active during voice recording.

3.7 Session History and Search

All conversations are saved automatically. To access past sessions:

1. Click the (history) button in the top bar. The session sidebar opens on the left.
2. Each saved session is listed with its title and date. Click any session to restore the full conversation, including all images.
3. To search through past sessions, type in the search bar at the top of the sidebar. The list filters in real time.
4. To delete a session, hover over it in the sidebar and click the × delete button. The session is removed immediately and permanently.
5. To start a new blank session, click **New Chat** in the top bar.

3.8 Retrying a Response

If the model's response is incorrect or unsatisfactory:

1. Find the bot response bubble you want to regenerate.
2. Click the **Retry** link in the footer of that bubble.
3. The chat display is trimmed back to just before that response and the model generates a fresh answer for the same question.

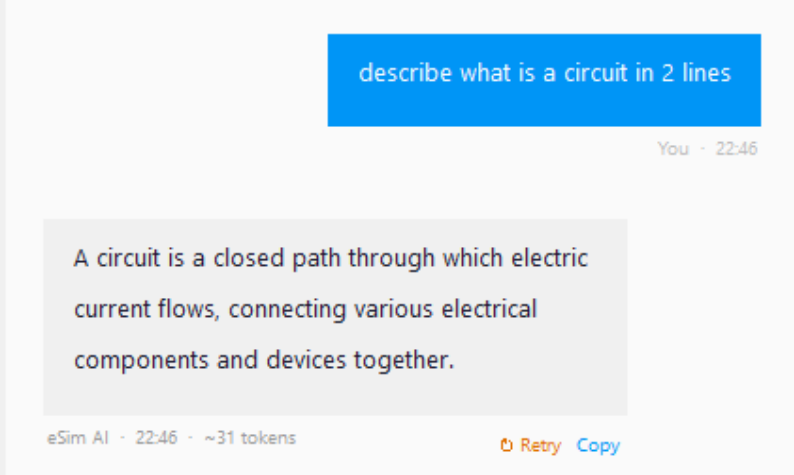


Figure 3.5: Message and Copy links in the footer of a bot response bubble.

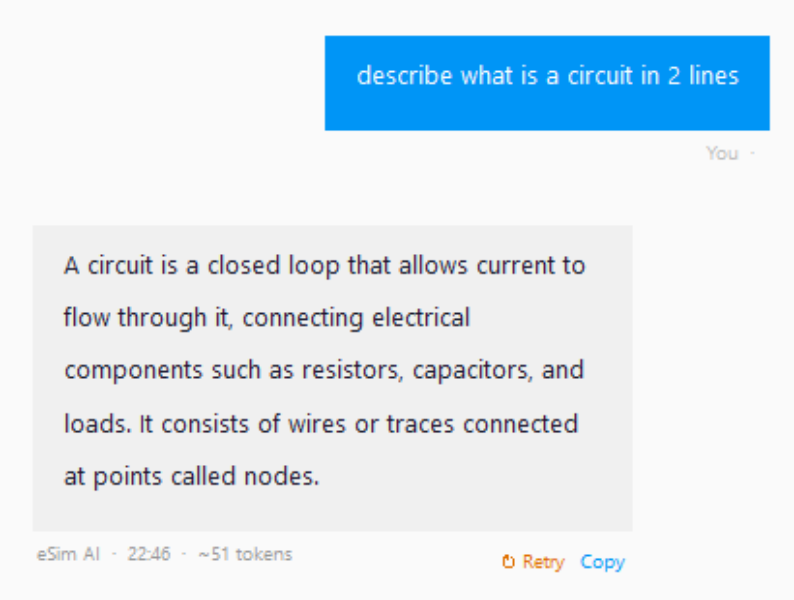


Figure 3.6: After Retry.

3.9 Changing the Model

1. Click the model name dropdown in the top bar of the chatbot window.
2. All locally available Ollama models are listed.
3. Select the desired model. The change takes effect from the next message.

For text-only queries, a smaller model such as `qwen2.5-coder:3b` is fast and accurate. For schematic image analysis, select a vision-capable model such as `qwen2.5-vl:3b` or `llava`.

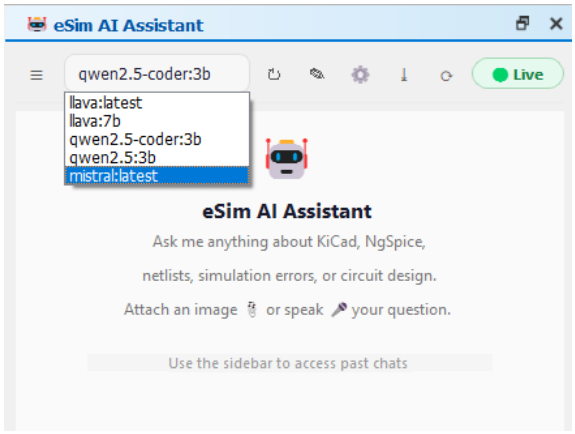


Figure 3.7: Model selector dropdown showing available locally installed Ollama models.

Chapter 4

Problem Statement

4.1 Problem Definition

Prior to this project, eSim provided no mechanism for a user to ask a natural language question about their circuit and receive a contextual answer within the application. The specific deficiencies identified are as follows:

- **No natural language interface.** All interaction is through menus, buttons, and dialogs, presenting a high barrier for new users unfamiliar with tool-specific vocabulary.
- **No schematic image understanding.** Circuit diagnosis is inherently visual. A text-only interface cannot address questions about what is visible in a schematic.
- **No netlist analysis.** NgSpice netlists are not readily human-readable; no tool existed within eSim to parse and explain them in plain language.
- **No session memory.** Each application launch begins from a blank state with no record of prior interactions.

4.2 Prototype Bugs

An early chatbot prototype existed but contained five bugs that prevented reliable use:

#	Bug	Observed Behaviour
1	Image not displayed	Attaching and sending an image rendered only a text filename badge; no image was shown.
2	Chat corruption	Switching focus during model generation caused real message content to be deleted from the display.
3	Session ghost	Deleted sessions reappeared in the sidebar after restarting the application.
4	Duplicate retry	Clicking Retry appended a new response below the old one instead of replacing it.
5	Startup crash	Application crashed before the window appeared with `TypeError: 'fixedSize' is an unknown keyword argument`.

Chapter 5

Implementation

5.1 System Architecture

5.1.1 Module Structure

The chatbot is implemented across two Python source files with a strict boundary: no blocking operation may execute on Qt's main thread.

- `Chatbot.py` - PyQt5 UI layer: `ChatbotGUI` main window, `SessionSidebar` widget, `ChatHistoryViewer` dialog, custom `_HistoryLineEdit` with history navigation, and all HTML bubble generation. Runs exclusively on the Qt main thread.
- `chatbot_thread.py` Background worker classes, each inheriting from `QThread`:
  - `OllamaWorker` streams text chat completions.
  - `OllamaVisionWorker` sends base64-encoded images with text to a VLM.
  - `MicWorker` - captures and transcribes audio.
  - `OllamaStatusWorker` polls Ollama server availability.

  All inter-thread communication uses Qt signals, which are thread-safe by design.

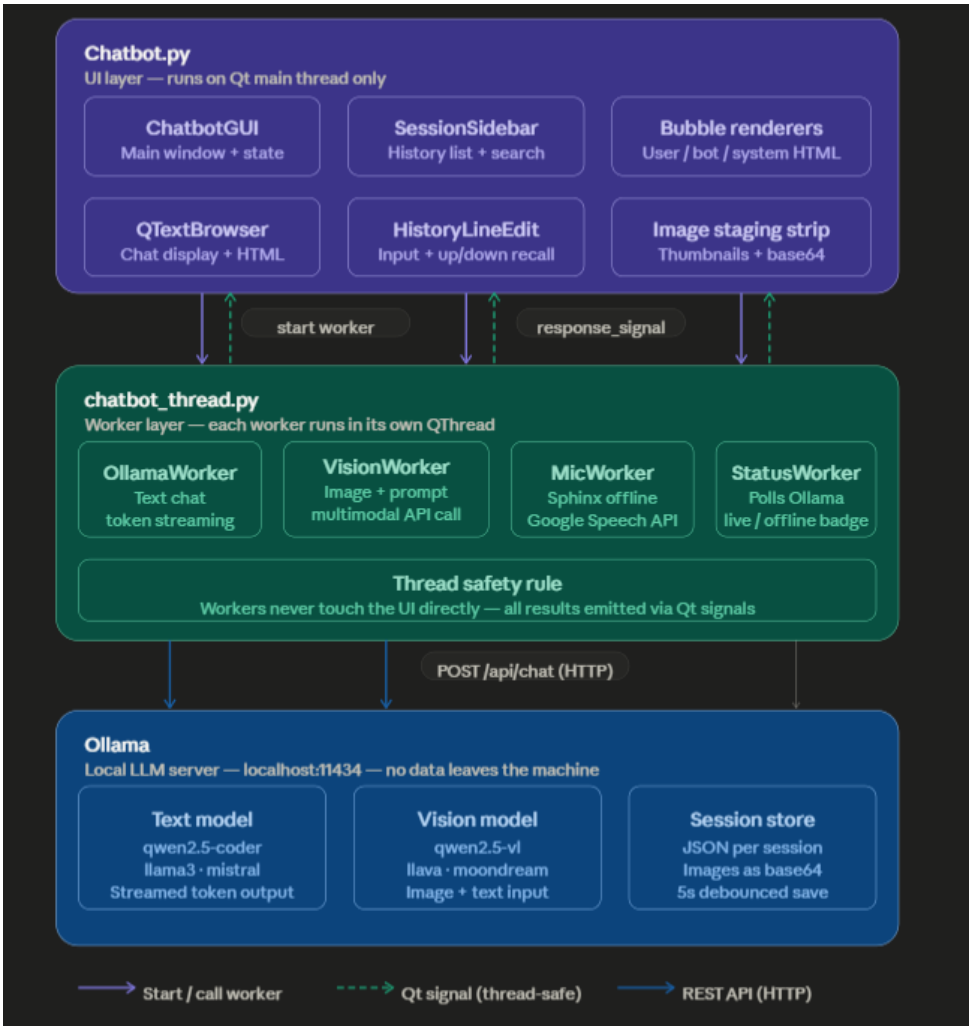


Figure 5.1: Chatbot system architecture overview.

5.1.2 Chat Bubble Rendering

The chat display is rendered as HTML inside a `QTextBrowser`. Three bubble types are defined:

Type	Alignment	Content
User	Right, blue	Raw user text, timestamp.
Bot	Left, grey	Markdown-rendered response (bold, italic, code, H1–H4, links), timestamp, token count, inline Retry and Copy links.
System	Centre, amber	Status messages (Ollama start, model switch, connection state).

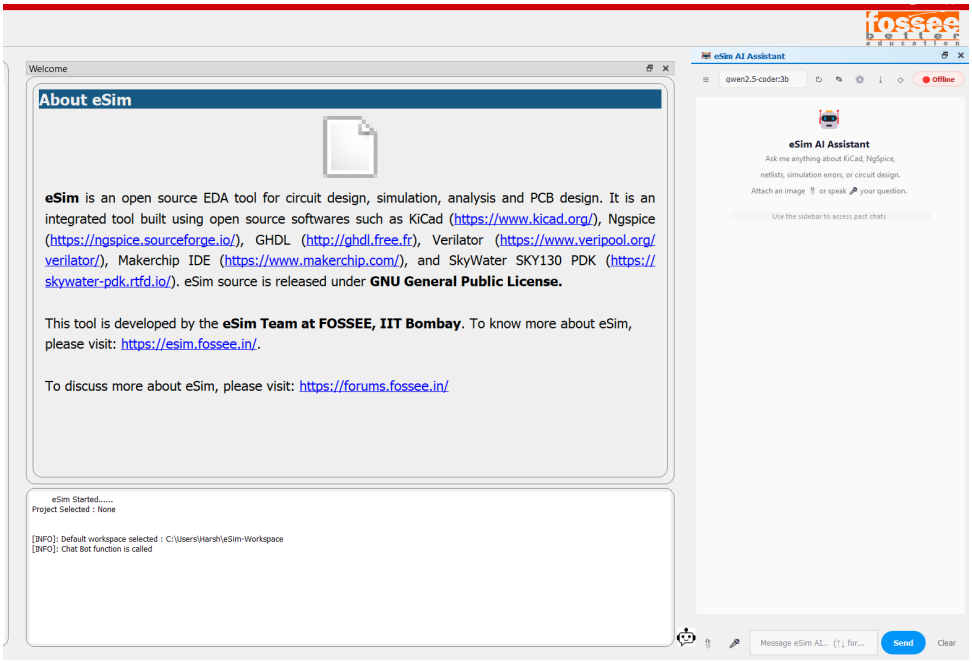


Figure 5.2: Chat bubble rendering in the eSim chatbot panel.

5.2 Feature Implementation

5.2.1 Vision Image Analysis

When the user attaches an image (via the attachment button or drag-and-drop), the following pipeline executes on Send:

1. Each image is read from disk and, if Pillow is available, downscaled to 320×240 pixels maximum.
2. The image bytes are base64-encoded and stored in a session-level dictionary, keyed by timestamp.
3. An inline HTML `<img>` tag using the base64 data URI is inserted into the chat display immediately.
4. An `OllamaVisionWorker` thread is started with the image paths, user text, and selected model name.

```
_VISION_SYSTEM_PROMPT = """You are an expert electronics
engineer and the AI assistant inside eSim, an open-source
EDA tool by FOSSEE at IIT Bombay.

Rules:
- Read every visible label, net name, component reference,
  value, and pin number from the image.
- If the user asks a specific question, answer THAT
   question
```

```
  directly and completely.
- If no question is given, describe the circuit: identify
  its function, list components, and flag design issues.
- Never refuse to analyse. If parts of the image are
   unclear,
  do your best and note any uncertainty.
- Match the length of your response to the complexity of
  the question, not to a fixed template.
"""
```

Listing 5.1: Vision system prompt (`chatbot_thread.py`)

```
def _build_schematic_vision_prompt(extra_prompt,
   image_count):
    n = ("this schematic" if image_count == 1
         else f"these {image_count} schematics")
    if extra_prompt and extra_prompt.strip():
        return (
            f"Looking at {n}: {extra_prompt.strip()}\n\n"
            "Base your answer on what is visible in the
               image."
        )
    else:
        return (
            f"Please analyse {n}. Identify the circuit's "
            "function, list all visible components with
               their "
            "reference designators and values, name the
               nets "
            "and signal rails, and flag any potential
               issues."
        )
```

Listing 5.2: Prompt builder that prioritises the user's question (`chatbot_thread.py`)

The user's question is promoted to the primary instruction. The prior implementation appended it as a secondary tag after a fixed analysis template, causing the model to produce a generic component dump regardless of what had been asked.

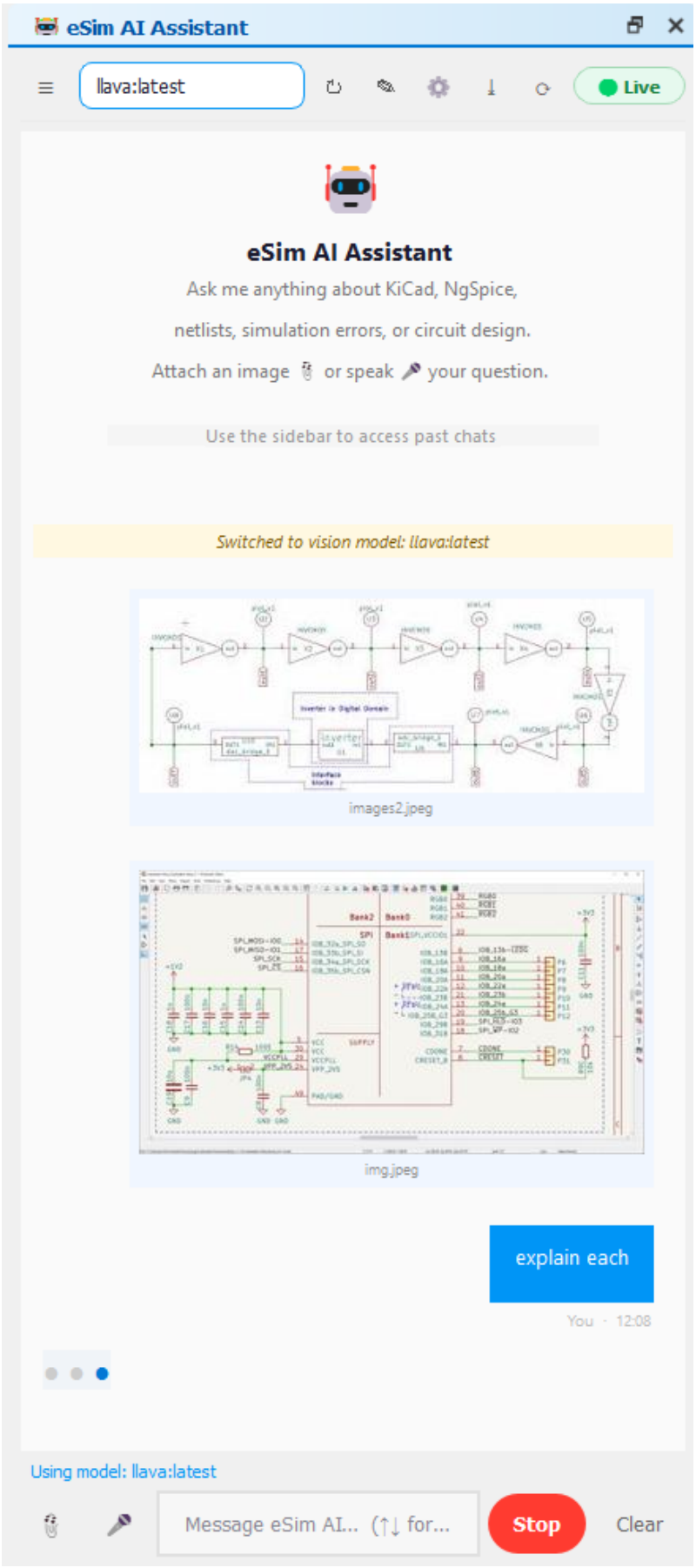


Figure 5.3: Image staging strip and vision model response for a schematic query.

5.2.2 Typing Animation with Window-Switch Safety

A three-frame dot animation is displayed while the model generates a response, driven by a `QTimer` at 400 ms intervals. The original implementation tracked the indicator's document position as a plain integer character offset (`_typing_start_pos`). Qt's HTML layout engine reflows document character positions on repaint events; when the user switched focus, the stored offset became invalid and the next timer tick deleted real message content.

The fix introduces a named HTML anchor as a position-independent sentinel:

```
_TYPING_ANCHOR = '<a name="_typing_anchor_"></a>'

def _show_typing_bubble(self):
    self._typing_frame = 0
    cursor = QTextCursor(self.chat_display.document())
    cursor.movePosition(QTextCursor.End)
    cursor.insertHtml(self._TYPING_ANCHOR + _typing_bubble
       (0))
    self._scroll_to_bottom()
    self._typing_anim_timer.start(400)
```

Listing 5.3: Anchor sentinel insertion (`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:
                    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
```

Listing 5.4: Position-independent anchor search (`Chatbot.py`)

Qt stores anchor names in fragment-level metadata rather than as character content, making them immune to position shifts caused by reflow. The search by anchor name always locates the correct fragment regardless of reflow history.

5.2.3 Session Persistence

Each conversation is automatically saved as a JSON file under `~/.esim_chatbot/sessions/`. The session record contains:

- Session UUID, title (derived from the first user message), and timestamps.
- Full message history (up to 40 entries).
- All images as base64-encoded strings, enabling complete session replay even if the original files have been deleted.
- Model name, temperature, and token limit settings.

Saves are debounced by a five-second `QTimer` to avoid I/O jitter during streaming. The `SessionSidebar` provides real-time search, and clicking any entry replays the full conversation text and images exactly as it appeared.

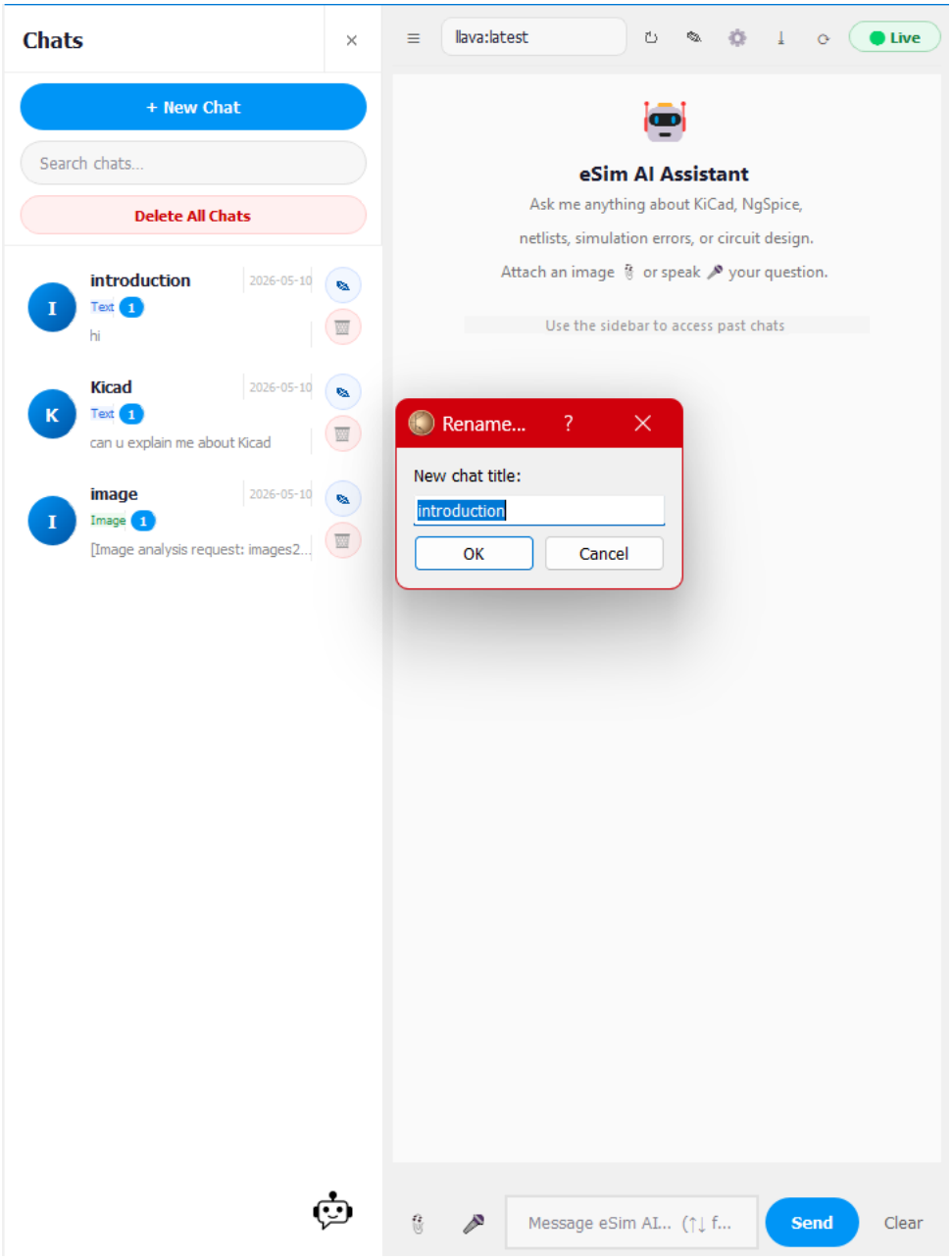


Figure 5.4: Session sidebar showing saved conversations with Remane functionality.

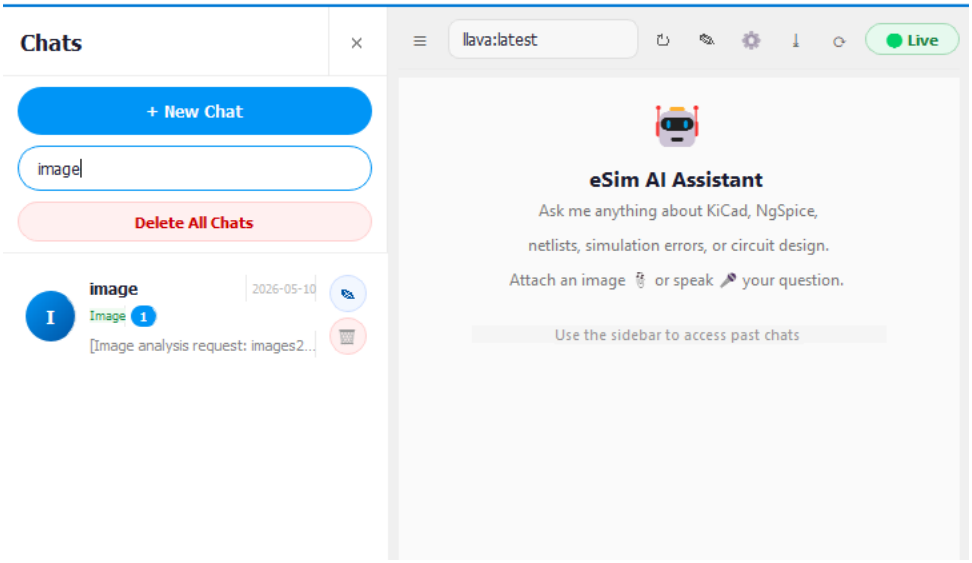


Figure 5.5: Session sidebar showing saved conversations with Search functionality.

5.2.4 Voice Input

The `MicWorker` background thread manages the complete voice pipeline:

1. Opens the default microphone via `PyAudio`; samples ambient noise for one second to calibrate the silence threshold.
2. Records audio until a 1–2 second pause is detected.
3. Attempts transcription via CMU Sphinx (offline).
4. Falls back to the Google Web Speech API if Sphinx fails or is unavailable.
5. Emits the transcribed text via a Qt signal; the main thread inserts it into the input field for user review before sending.

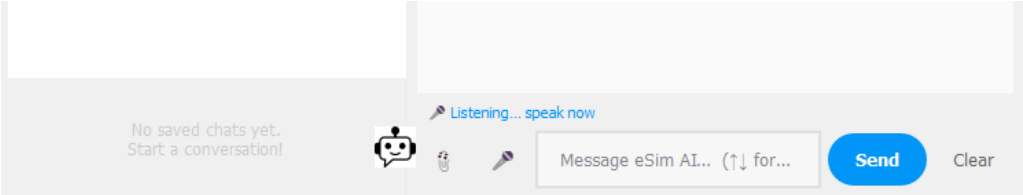


Figure 5.6: Session sidebar showing saved conversations with Search functionality.

5.2.5 NgSpice Netlist Analysis

The `analyse_netlist()` method is invoked when the user opens the chatbot with a netlist loaded. It performs the following operations:

1. Reads and tokenises the netlist file line by line.

2. Classifies components by reference prefix: R, C, L, D, Q, U/X (subcircuits), V/I (sources).

3. Collects all unique node names (excluding ground, node 0).

4. Identifies analysis directives: `.tran`, `.ac`, `.dc`, `.op`, `.noise`, `.model`, `.include`.

5. Constructs a structured plain-language summary and sends it to the LLM along with the full netlist (truncated at 80 lines), requesting: component function description, overall circuit identification, simulation directive interpretation, and potential issue flagging.

5.3 Bug Fixes

5.3.1 Bug 1 - Images Not Displayed After Sending

Root cause: The chat display was updated before image encoding completed. The existing `_image_thumbnail_html()` function used correctly during session replay was never called during a live send.

Fix: Image encoding was moved before the chat display update. `_image_thumbnail_html(b64, fname)` is now called for each image, inserting a rendered base64 `<img>` tag directly into the user bubble.

5.3.2 Bug 2 - Chat Corruption on Window Switch

Root cause: The typing indicator's position was tracked as an integer character offset (`_typing_start_pos`). Qt's HTML layout engine reflows document character positions on repaint events triggered by focus changes. The stored offset became invalid after reflow, causing the next timer tick to select and delete real message content.

Fix: Integer position tracking was replaced with a named HTML anchor sentinel (see Listings 5.3–5.4). Qt stores anchor names in fragment metadata, which is independent of character positions and survives any number of reflow events.

5.3.3 Bug 3 - Deleted Sessions Reappearing

Root cause: Session saves are debounced by a five-second timer. If a session was deleted within five seconds of its last message, the delete handler removed the JSON file from disk, but the debounce timer continued to count. When it fired, it recreated the file via `_flush_save()`.

Fix: Both the delete handler (`_on_session_deleted`) and `clear_session()` now stop the debounce timer and clear the pending-save flag as their first action:

```
def _on_session_deleted(self, deleted_id: str):
    if deleted_id == self._current_session_id:
        self._abort_worker()
        # Stop the debounce timer before removing the file
            .
```

```
        # Without this, _flush_save() fires and recreates
        # the deleted file, causing it to reappear on
           relaunch.
        self._save_debounce_timer.stop()
        self._save_pending = False

        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
```

Listing 5.5: Debounce cancellation on session delete (`Chatbot.py`)

5.3.4 Bug 4 - Retry Producing Duplicate Responses

Root cause: Three distinct issues: (1) The Retry button was only shown for error-prefixed responses. (2) The retry method launched the LLM worker without first removing the previous response bubble, so the new response appended below the old one. (3) The method always constructed `OllamaWorker` (text-only), even when the original query included images.

Fix: The status-bar `QPushButton` was removed. A Retry hyperlink using a custom URL scheme (`retry:///idx`) is now embedded in the footer of every bot bubble and intercepted via `QTextBrowser.anchorClicked`.

The new `_retry_response(idx)` method:

1. Locates the target response in `chat_history` by index.
2. Slices the history, removing the target and all subsequent entries.
3. Calls `_rebuild_chat_html_from_history()` to redraw the display from the trimmed history, cleanly removing the stale bubble.
4. Inspects the last user message for an image reference and starts `OllamaVisionWorker` or `OllamaWorker` accordingly.

```
retry_href = f'retry:///{response_idx}'
copy_href  = f'copy:///{response_idx}'

# Footer HTML fragment:
f'<a href="{retry_href}" style="color:#e07000;'
f'font-size:10px;text-decoration:none;">&#8635; Retry</a>'
f'  '
f'<a href="{copy_href}" style="color:#0095f6;'
f'font-size:10px;text-decoration:none;">Copy</a>'
```

Listing 5.6: Retry link embedded in every bot bubble (`Chatbot.py`)

5.3.5 Bug 5 - Application Crash on Startup

Root cause: Six widget constructors passed Qt property names (`fixedSize`, `fixedHeight`) as Python keyword arguments. PyQt5 does not map Qt properties as constructor parameters and raises `TypeError` immediately, preventing the window from appearing.

Fix: Invalid keyword arguments were removed from all constructors. Sizes are applied using the correct PyQt5 setter methods post-construction:

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

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

Listing 5.7: Corrected PyQt5 widget sizing

5.3.6 Additional Code Quality Improvements

- **Dead state variable.** `_typing_start_pos` was still initialised and assigned after the anchor rewrite but never read. All references were removed.
- **Dead timer.** `_thinking_timer` was connected to `_animate_thinking()`, which contained only a `pass`. Both the timer and the empty method were removed.
- **Unreachable `break`.** An inner break condition inside the session image replay loop was a strict subset of the outer while condition and could never be reached; it was removed.
- **Smart token budget.** `_smart_num_predict()` previously sampled only `User:`-prefixed lines for complexity estimation, producing an empty sample when recent history contained only bot turns. Both user and bot lines are now included in the sample.
- **Developer patch notes.** All `# FIX #N:` annotations were removed or replaced with accurate inline documentation.

Summary of Contributions

The table below provides a consolidated view of all features implemented and bugs resolved during this internship. The project delivered nine fully working features and fixed five pre-existing bugs that prevented reliable use of the chatbot prototype.

Total: 9 features implemented | 5 bugs resolved | 3 source files modified (`frontEnd_Chatbot.py`, `chatbot_thread.py` ,`frontEnd_application.py`)

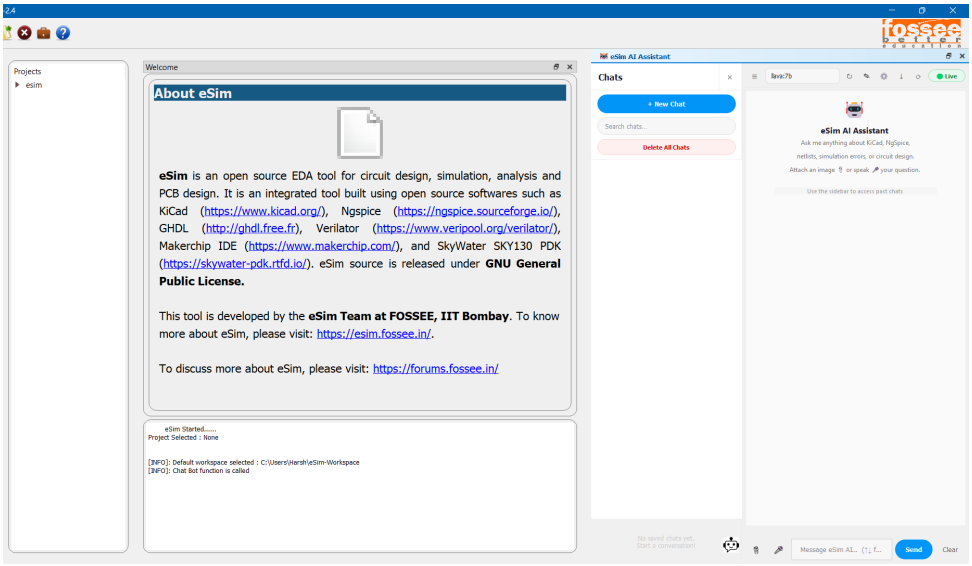


Figure 5.7: eSim chatbot complete user interface.

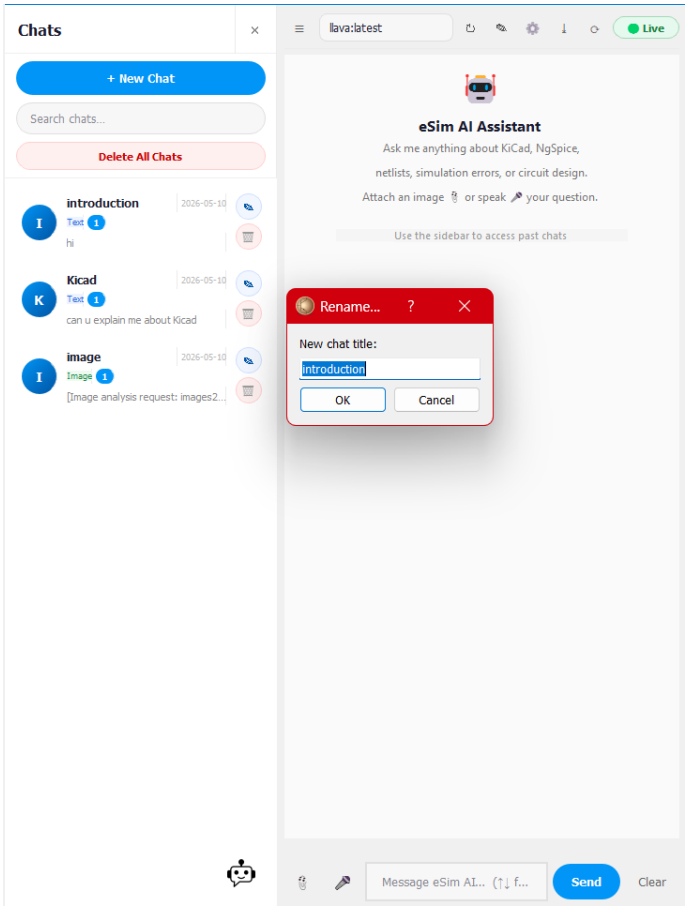


Figure 5.8: All session are saved in sidebar.

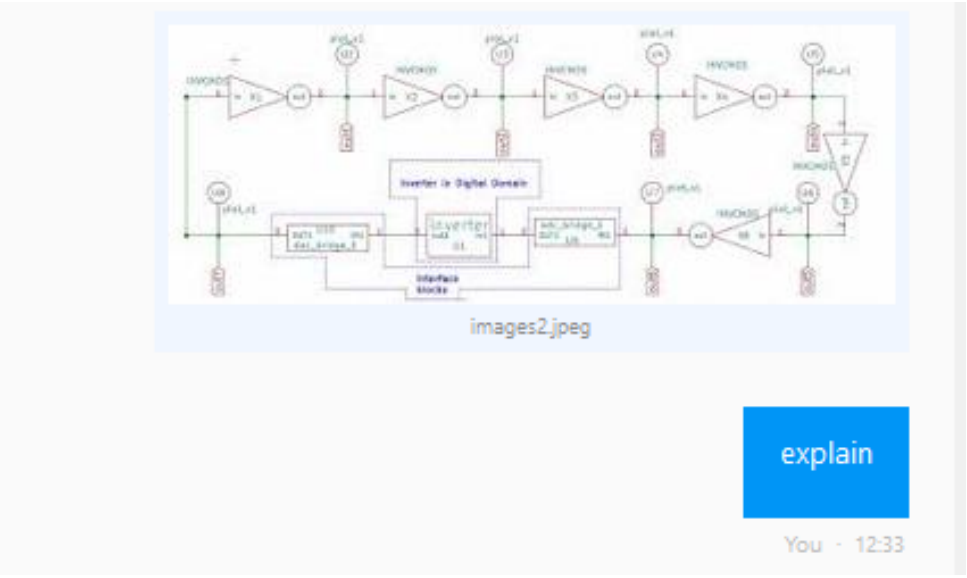


Figure 5.9: image now renders in the chat

Table 5.1: Complete work summary — eSim Chatbot Development

Feature / Fix	**Status**	**Details**
Part A — Features Implemented		
Native chatbot panel	Implemented	Floating window integrated directly inside eSim.
Local inference via Ollama	Implemented	Fully offline — no user data sent externally.
Text chat	Implemented	Streaming token-by-token display via Ollama API.
Schematic image analysis	Implemented	Vision model; supports multiple images per message.
Session persistence	Implemented	JSON files with full base64 image storage and restore.
Session search sidebar	Implemented	Real-time text search across all past sessions.
Voice input	Implemented	CMU Sphinx (offline) and Google Speech API (online).
NgSpice netlist analysis	Implemented	Parses components, nodes, directives; explains circuit.
Multi-model support	Implemented	Any Ollama model selectable at runtime from the UI.
Part B — Bugs Fixed		
Bug #1 Image not displayed	Fixed	Inline base64 thumbnail rendered on send.
Bug #2 Chat corruption	Fixed	Anchor-based sentinel replaces integer position tracking.
Bug #3 Session reappearing	Fixed	Debounce timer cancelled immediately on delete.
Bug #4 Duplicate retry	Fixed	Per-bubble retry link; history trimmed before regeneration.
Bug #5 Startup crash	Fixed	Invalid constructor kwargs replaced with PyQt5 setters.

Chapter 6

Conclusion and Future Scope

Conclusion

This project delivered a fully functional AI assistant integrated into the eSim EDA tool, capable of:

- Multi-turn conversation with a locally hosted LLM via Ollama.
- Schematic image analysis with vision-capable models.
- Persistent session storage and restoration, including embedded images.
- Voice input with offline and online transcription backends.
- NgSpice netlist parsing and plain-language circuit explanation.

All five bugs in the existing prototype were identified, their root causes determined, and targeted fixes applied. The most technically significant was Bug #2 (chat corruption), which required understanding Qt's internal HTML document reflow behaviour and identifying `QTextCharFormat.anchorNames()` as a mechanism for position-independent fragment identification.

Future Scope

1. **eSim-specific fine-tuned model.** A model fine-tuned on eSim documentation, the NgSpice manual, and the FOSSEE subcircuit library would yield substantially more accurate domain-specific responses. The training data already exists.
2. **Simulation output analysis.** Reading NgSpice `.raw` output files post-simulation to answer quantitative questions (e.g., 3 dB bandwidth, gain margin) directly from simulation results.
3. **Agentic actions.** Extending the chatbot to execute actions within eSim modifying component values, adjusting simulation parameters in response to natural language instructions, via tighter integration with eSim's internal data model.

4. **Persistent multi-turn image context.** A context management strategy retaining image tokens across multiple follow-up turns for extended schematic discussions.

5. **Community session sharing.** A mechanism to publish annotated session files through the FOSSEE platform, building a shared knowledge base for the eSim community.

Bibliography

1. Bai, J., et al. (2023). *Qwen Technical Report.* arXiv:2309.16609. https://arxiv.org/abs/2309.16609

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

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

4. Blocklove, J., Garg, S., Karri, R., and Pearce, H. (2023). *Chip-Chat: Challenges and Opportunities in Conversational Hardware Design.* arXiv:2305.11973. https://arxiv.org/abs/2305.11973

5. FOSSEE Team, IIT Bombay. *eSim: An Open Source EDA Tool for Circuit Design, Simulation and PCB Design.* https://esim.fossee.in [Accessed May 2026]

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

7. Riverbank Computing Limited (2023). *PyQt5 Reference Guide.* https://www.riverbankcomputing.com/static/Docs/PyQt5/ [Accessed May 2026]

8. Kennedy, A. (2016). *SpeechRecognition: Speech recognition module for Python.* https://github.com/Uberi/speech_recognition [Accessed May 2026]

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

10. Clark, A. and the Pillow Development Team (2024). *Pillow: The friendly PIL fork.* https://python-pillow.org [Accessed May 2026]