FOSSEE Summer Fellowship 2026

On

AI Chatbot Testing and Development for eSim

Submitted by

Aditya Pandey
VIT Bhopal University

Under the guidance of

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

and

Mentor: Sumanto Kar
FOSSEE Project, IIT Bombay

18 July 2026

Acknowledgments

Sincere gratitude was expressed to **Prof. Prabhu Ramachandran** for providing the opportunity to participate in the FOSSEE internship programme and for his continued efforts in promoting open-source engineering tool development. His leadership and vision were instrumental in fostering meaningful student participation in the open-source ecosystem.

Prof. Kannan M. Moudgalya was acknowledged 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 were pivotal in creating the academic and organisational platform through which this internship was undertaken.

Sincere appreciation was extended to the project 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.

Gratitude was also expressed to the 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 was an enriching learning experience that allowed close engagement with open-source EDA tools, development of an AI-powered chatbot assistant in `eSim`, and extensive exposure to real-world circuit modelling, simulation workflows, and production software engineering practices.

The entire FOSSEE team was thanked for their coordination, assistance, and timely interactions at various stages of this work.

Contents

Chapter 1

Introduction and EDA Ecosystem

1.1 The FOSSEE Project

The FOSSEE (Free and Open Source Software for Education) project, hosted at the Indian Institute of Technology Bombay and funded under the National Mission on Education through ICT (NMEICT), was established to promote open-source software adoption across Indian academic institutions. The project maintained a portfolio of tools spanning multiple disciplines, including Scilab for numerical computation, OpenFOAM for computational fluid dynamics, Python for scientific computing, and `eSim` for Electronic Design Automation (EDA). FOSSEE operated through spoken tutorials, textbook companions, lab migration programmes, and fellowship-driven development sprints involving students directly in production-quality software engineering.

1.2 Overview of eSim

`eSim` was not a single piece of software; it was a carefully integrated collection of best-in-class open-source tools, connected by a Python and `PyQt5`-based shell that presented them as a unified application.

KiCad handled schematic capture. It provided `eSim` with a professional-grade environment for drawing circuits, assigning component values, managing symbol libraries, and generating netlists—the structured text files that described a circuit's components and connections.

NgSpice handled simulation. It was a mature implementation of the SPICE circuit simulation standard. Through `NgSpice`, `eSim` supported transient analysis (time-domain response), AC analysis (frequency response), DC sweep (operating point variation), operating point analysis (steady-state DC), and noise analysis.

After simulation, `eSim` provided Python-based tools for plotting waveforms and extracting quantitative results. The platform was free, cross-platform, and the primary

circuit simulation tool recommended by FOSSEE for undergraduate engineering courses.

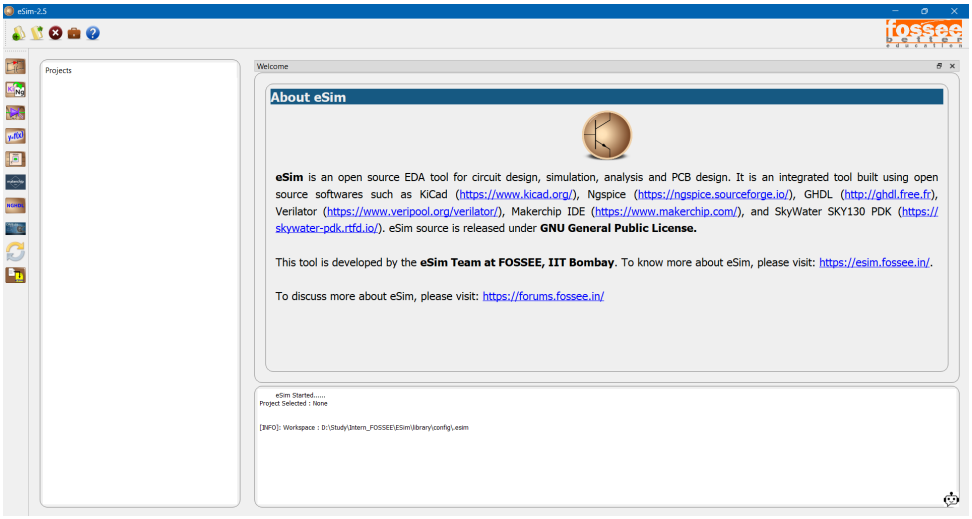


Figure 1.1: The eSim interface showing the project explorer (left), KiCad schematic editor (centre), and the toolbar/menu system.

1.3 Simulation Architecture: KiCad to NgSpice

The circuit design workflow began with schematic capture in `KiCad`. Users drew circuits by placing component symbols from the `eSim` device libraries, wiring them together, and assigning component values through properties dialogs. Each component was identified by a reference designator (e.g., `R1`, `C3`, `Q2`).

Once the schematic was complete, the **Generate Netlist** command produced a `KiCad`-format netlist. The KiCad-to-NgSpice converter module (`src/kicadtoNgspice/`) then transformed this into a SPICE-compatible `.cir.out` file through node numbering (ground assigned node 0), component mapping (e.g., `R1 2 3 1k`), model inclusion (`.model`/`.include` directives), and analysis directive insertion (`.tran`, `.ac`, `.dc`, or `.op`).

Several categories of simulation failure were frequently encountered by student users, including missing ground connections, floating pins, disconnected wires, missing SPICE models, singular matrix errors, and missing component values. These failure modes formed a core part of the chatbot's built-in knowledge base.

1.4 Objectives of the Project

The project focused on achieving the following objectives:

1. **Develop an integrated chatbot panel within eSim** that functioned as a built-in part of the application rather than an external tool.
2. **Ensure complete local and offline operation using Ollama** so that all conversations, files, and uploaded content remained private.
3. **Enable schematic image understanding using vision-based AI models** that could accept schematic images and answer detailed circuit-related questions.
4. **Create persistent chat history with searchable session management** that survived application restarts.
5. **Add voice input support** for users during hardware testing or prototyping.
6. **Implement NgSpice netlist analysis capabilities** so the assistant could read and interpret actual netlists.
7. **Identify and resolve bugs in the existing prototype system** to achieve reliable daily-use stability.

1.5 Methodology

The work was carried out iteratively across five overlapping phases:

Phase 1 — Study and Architecture: The existing `eSim` codebase and prototype chatbot code were read thoroughly, followed by mentor discussions to agree on scope, priorities, and technical constraints.

Phase 2 — Core Implementation: The chat panel, bubble rendering system, model selector, session save/load, and sidebar were built.

Phase 3 — Feature Development: Vision analysis, voice input, netlist analysis, the image staging strip, and drag-and-drop support were added.

Phase 4 — Bug Investigation and Fixing: A systematic code review covering every method, state variable, and signal connection was conducted. 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

The Rationale for Intelligent Assistants in EDA

2.1 Cognitive Barriers for Student Engineers

Circuit design and simulation involved a steep learning curve combining domain knowledge (electronics, semiconductor physics, network theory) with tool-specific procedural knowledge (menu paths, file formats, configuration dialogs, keyboard shortcuts). For a second-year undergraduate encountering `eSim` for the first time, the cognitive load was significant.

Consider a specific scenario: a student drew a common-emitter amplifier in `KiCad`, generated the `NgSpice` netlist, and ran a transient simulation. The output waveform was clipping where it should not have been. The student had no idea why. In the current `eSim`, options were limited to: closing `eSim` and searching online, emailing a professor, or posting on a forum and waiting. None was fast, none was aware of the specific circuit, and none kept the student in their workflow.

2.2 The Central Problem Statement

eSim had 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 had several concrete dimensions:

- **No natural language interface.** Every interaction was through menus, buttons, and dialogs. There was no way to describe what the user wanted in plain language.
- **No schematic image understanding.** Schematics were visual. Much of the knowledge needed—recognising topologies, spotting wiring errors—was inherently visual.

- **No netlist analysis.** A `NgSpice` netlist from a moderately complex design could run to hundreds of lines of SPICE syntax. No tool within `eSim` read a netlist and explained it in plain language.
- **No session memory.** Every time `eSim` was opened, the user started from a blank state.
- **Existing prototype bugs:**
  - After sending an image, only a filename badge appeared—the actual image was never shown.
  - Switching focus during generation corrupted the chat display—messages were partially deleted.
  - Deleted sessions reappeared on the next launch.
  - The retry function produced duplicate stacked responses.
  - The application crashed on startup with a `TypeError` due to invalid `PyQt5` constructor arguments.

2.3 The Need for Offline, Private, Zero-Latency Assistants

Cloud-based AI assistants introduced problems for an educational EDA tool: **privacy** (students could not safely upload proprietary designs), **latency** (internet round trips disrupted design flow), **availability** (many Indian labs operated in low-bandwidth or air-gapped environments), **cost** (per-token API pricing was unsustainable for a free tool), and **context** (cloud models had no awareness of the specific `eSim` environment). These constraints made a compelling case for an offline, locally executed AI assistant embedded directly within the application.

The development approach focused on solving each issue through targeted, minimal modifications. `Ollama` was selected as the backend because of its efficient streaming REST API, support for multiple open-weight models, CPU compatibility, and fully local execution.

Chapter 3

Local AI Model Inference and Architectures

3.1 Local Quantised Model Execution

Running Large Language Models on consumer hardware required aggressive weight compression. Techniques such as GPTQ [7] demonstrated that model weights could be compressed into 4-bit integer representations while preserving most of the model's original accuracy. The `llama.cpp` project [8] combined these quantisation methods with a highly optimised C++ inference engine that executed quantised models directly on CPUs using SIMD instruction sets (AVX2, AVX-512, ARM Neon), eliminating the requirement for dedicated GPUs. It also introduced the GGUF file format, which became the standard for distributing compressed models. `Ollama` [10] built upon this foundation, providing a user-friendly framework for model management (downloading, storage, quantisation selection, GPU offloading) and a REST API at `http://localhost:11434` with streaming token-by-token output. A 3-billion-parameter model in Q4_K_M quantisation occupied approximately 2 GB of RAM, making it feasible to run on standard student laptops with 8 GB of memory.

3.2 Vision-Language Model Integration

Vision-Language Models (VLMs) extended text-only LLMs by combining a visual encoder (typically a Vision Transformer or CLIP model) with the language model's token processing. Images were converted into visual tokens and processed alongside text tokens, enabling the model to reason about both modalities simultaneously. Models such as LLaVA [6] demonstrated that even a 7-billion-parameter model could effectively analyse complex images. In the `eSim` context, this meant users could upload schematic screenshots and receive component-level analysis without manually describing the circuit in text. The assistant's vision pipeline used models such as

Moondream, LLaVA, and MiniCPM-V, with automatic model selection based on installed availability.

3.3 Text Embedding and Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) [19] grounded LLM responses in factual documentation. The `eSim` knowledge base was built by chunking manual text files at paragraph boundaries, embedding each chunk into a 768-dimensional vector using `nomic-embed-text` through `Ollama`, and storing the vectors in a ChromaDB persistent collection. At query time, the user's question was embedded with the same model, the top 4 nearest chunks were retrieved via L2 distance, chunks exceeding a relevance threshold of 500 were discarded, and surviving context was injected into the system prompt. This produced answers grounded in official `eSim` documentation rather than model hallucination.

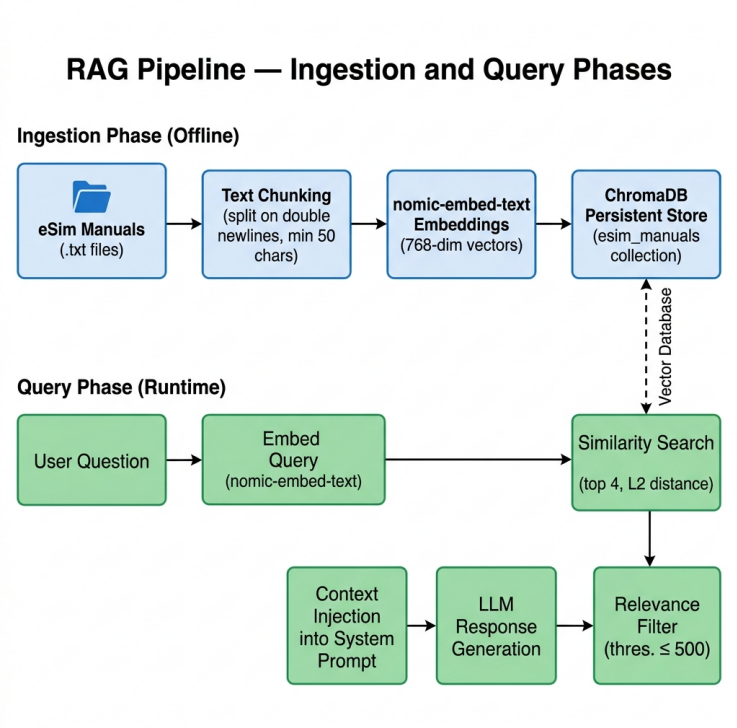


Figure 3.1: RAG pipeline showing the offline ingestion phase (top) and the runtime query phase (bottom). Manual text files were chunked, embedded, and stored in ChromaDB. At query time, the user's question was embedded, matched, filtered, and injected into the LLM system prompt.

3.4 Speech Recognition

Two speech-to-text pathways were implemented. The primary path used the Vosk API with the `vosk-model-small-en-us-0.15` acoustic model (∼40 MB), capturing audio from the default microphone at 16 kHz and processing chunks in real time with silence detection. The secondary path used the `SpeechRecognition` library with the Google Web Speech API (higher accuracy, requires internet) and CMU Sphinx as a fully offline fallback.

Chapter 4

Core System Implementation Details

4.1 System Architecture Overview

The chatbot system was designed using a strict separation between the graphical user interface (main thread) and background processing (worker threads). No blocking operations were executed on Qt's main UI thread.

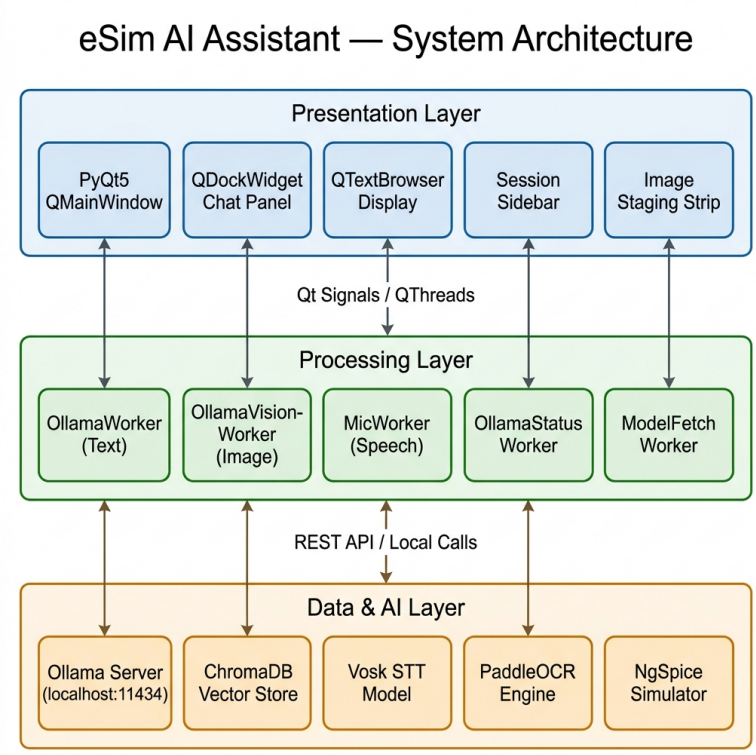


Figure 4.1: Three-layer system architecture. The presentation layer (blue) handled GUI rendering on the main thread. The processing layer (green) executed background workers as QThreads. The data/AI layer (orange) contained external services and databases.

4.1.1 Module Structure

The first file, `Chatbot.py` (2855 lines), contained the complete `PyQt5` user interface layer: the main `ChatbotGUI` window class, the `SessionSidebar` widget, the `ChatHistoryViewer` dialog, and a custom `_HistoryLineEdit` input component with command-style up/down history navigation. This file operated entirely on the main thread and performed no blocking I/O.

The second file, `chatbot_thread.py` (651 lines), contained all background worker classes:

- **OllamaWorker** — Text chat via `Ollama /api/chat` with streamed token-by-token responses.
- **OllamaVisionWorker** — Multimodal image+text analysis using vision-capable models.
- **MicWorker** — Microphone capture and speech-to-text transcription.
- **OllamaStatusWorker** — Server availability polling (port 11434).
- **ModelFetchWorker** — Installed model list retrieval.

Communication between workers and UI was performed using Qt's thread-safe `pyqtSignal` mechanism.

To eliminate C++ memory access violation segmentation faults (segfaults) triggered by SQLite/ChromaDB concurrent calls inside the PyQt5/PyQt6 thread loop on Windows, the database querying pipeline was decoupled from the main process runtime. The entire semantic search loop was isolated inside an independent Python subprocess. Communication between the primary GUI window and the database execution context was routed via standard input/output (stdin/stdout) pipelines, with query text strings completely serialised using Base64 encoding to prevent underlying system string parsing limitations.

```
import subprocess
import sys
import base64

# Spawn isolated subprocess using the local interpreter
cmd = [
    sys.executable, "-c",
    "import sys, base64; "
    "from chatbot.knowledge_base import search_knowledge; "
    "query = base64.b64decode("
```

```
    "    sys.stdin.read().encode('utf-8')"
    ").decode('utf-8'); "
    "result = search_knowledge(query); "
    "print(base64.b64encode("
    "    result.encode('utf-8')"
    ").decode('utf-8'))"
]
q_b64 = base64.b64encode(
    latest_user.encode('utf-8')
).decode('utf-8')

proc = subprocess.Popen(
    cmd,
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
    stderr=subprocess.PIPE,
    text=True,
    encoding='utf-8',
    creationflags=(
        subprocess.CREATE_NO_WINDOW
        if os.name == 'nt' else 0
    )
)
stdout, stderr = proc.communicate(
    input=q_b64, timeout=10
)
if proc.returncode == 0:
    rag_context = base64.b64decode(
        stdout.strip().encode('utf-8')
    ).decode('utf-8')
```

Listing 4.1: Thread-Safe RAG Subprocess Isolation via Base64 Streams.

4.1.2 Deterministic Netlist Pre-Parsing Matrix

To decouple verified circuit facts from the probabilistic output of a language model, the system implemented a deterministic netlist pre-parsing pipeline. Before any LLM invocation, the uploaded `.cir` or `.cir.out` file was read line-by-line and parsed using a rule-based state machine. The parser extracted a structured component

matrix containing reference designators, node connections, parameter values, analysis directives (`.tran`, `.ac`, `.dc`, `.op`), and model/subcircuit inclusion paths. This matrix was constructed entirely offline, with zero model inference cost, producing a ground-truth representation of the circuit. The local LLM runtime was then invoked strictly to explain the pre-parsed facts in natural language or to handle student follow-up questions that required contextual reasoning. This separation ensured that component counts, node assignments, and analysis parameters were never hallucinated—they were always read directly from the netlist source file by the deterministic parser, and the language model received them as authoritative context rather than generating them speculatively.

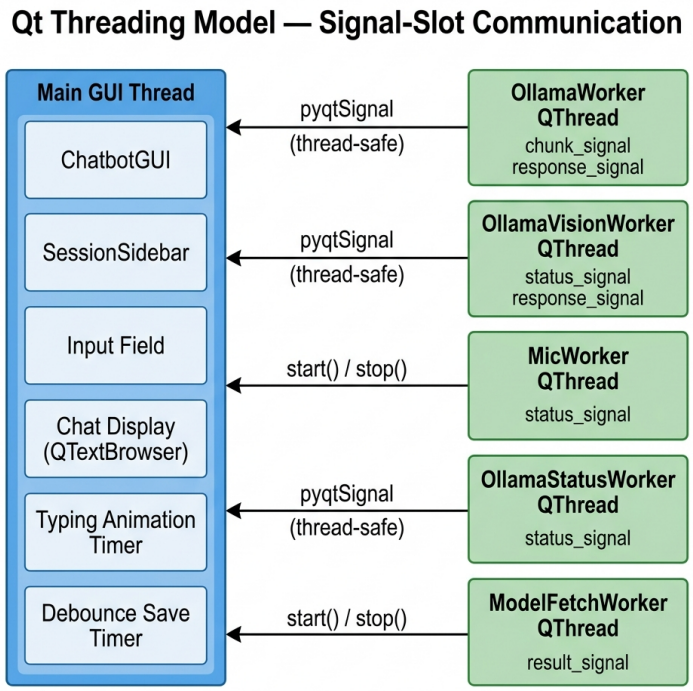


Figure 4.2: Qt threading model showing signal-slot communication between the main GUI thread and the five background QThread workers.

4.2 Configuration Management

Runtime parameters were externalised to `config.json`, loaded at startup using a deep-merge strategy over hardcoded defaults. This ensured the application always had valid configuration even if the file was missing or malformed:

```
1 def _deep_merge(base: dict, override: dict) -> dict:
```

```
    out = dict(base)
    for k, v in (override or {}).items():
        if k in out and isinstance(out[k], dict) \
                    and isinstance(v, dict):
            out[k] = _deep_merge(out[k], v)
        else:
            out[k] = v
    return out

CONFIG = _deep_merge(_DEFAULT_CONFIG, json.load(f))
```

Listing 4.2: Deep-merge configuration loading (`chatbot_thread.py`).

The configuration covered system prompts (text and vision), context window sizes (`num_ctx`), sampling parameters (`repeat_penalty`, `vision_temperature`), runtime settings (`keep_alive`), and history limits (`max_lines`).

4.2.1 Self-Healing Runtime Environments

On a clean `eSim` installation, the chatbot's hidden configuration directories (`~/.esim/`, `~/.esim/chat_sessions/`, and the ChromaDB vector store path) did not yet exist. Without preemptive handling, every `open()`, `os.listdir()`, or `PersistentClient()` call would raise an unrecoverable `FileNotFoundError` or `OSError` on first launch, crashing the application before the user saw any interface.

The `Appconfig.py` module intercepted this scenario at import time. It dynamically located the installation root using `os.path.dirname(os.path.abspath(__file__))`, traversed upward to the project base, and constructed the full hidden folder tree using `os.makedirs(path, exist_ok=True)`. If `config.json` was missing, a default configuration dictionary was serialised and written to disk. If the ChromaDB directory was absent, it was created and the knowledge base ingestion pipeline was triggered automatically. This self-healing bootstrap ensured that no manual directory creation or configuration file placement was ever required—a fresh `eSim` installation with the chatbot module would initialise all runtime dependencies on its first execution without any user intervention or traceback crashes.

4.3 Chat Bubble Rendering

Three distinct bubble styles were implemented as HTML table fragments with inline CSS, rendered inside a `QTextBrowser` widget:

1. **User Bubble:** Right-aligned, blue gradient background, with timestamp.
2. **Bot Bubble:** Left-aligned, light grey background. Response text was processed through a Markdown renderer supporting bold, italic, inline code, headings (H1–H4), and hyperlinks. The footer contained timestamp, token count, and interactive "Retry" / "Copy" action links via custom URL schemes (`retry:///idx`, `copy:///idx`).
3. **System Bubble:** Centre-aligned, amber background, for status notifications.

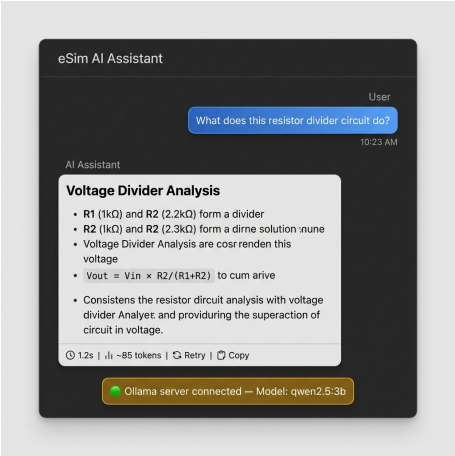


Figure 4.3: Inline HTML-rendered chat conversation frame with left/right oriented dialogue boxes and Markdown compilation structure.

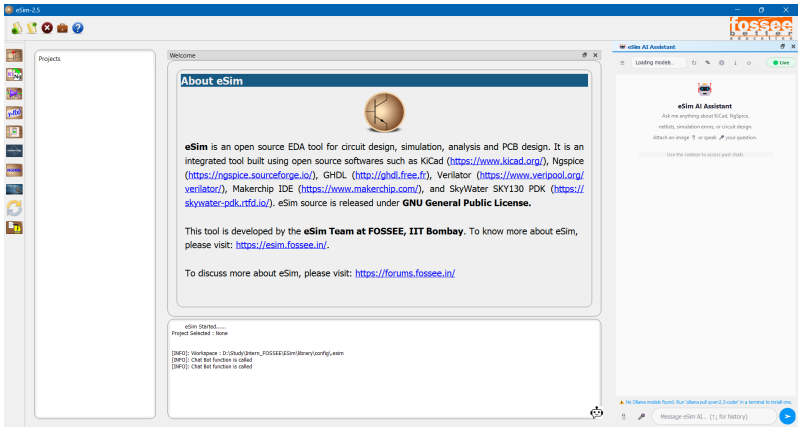


Figure 4.4: The eSim interface with the AI Assistant panel open, showing styled message bubbles, model selector, and input toolbar.

4.4 Vision Analysis Pipeline

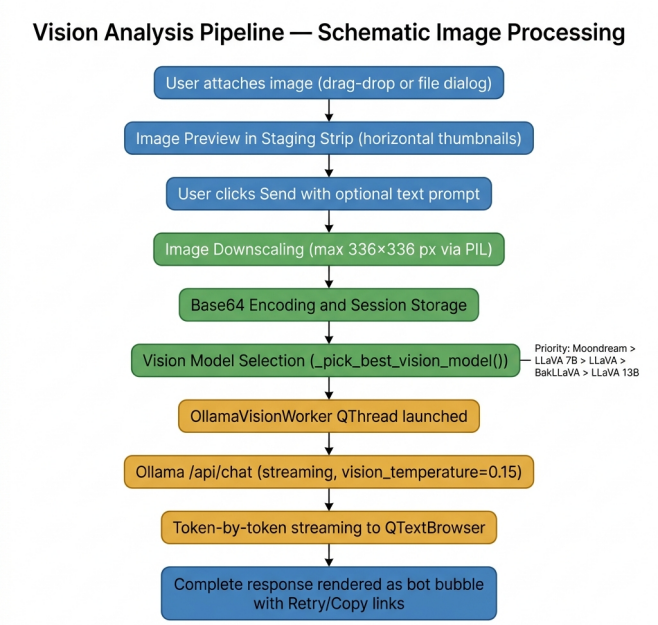


Figure 4.5: Vision analysis pipeline from image attachment through Base64 encoding, model selection, and streamed response generation.

The pipeline operated through the following sequence: the user attached images via the attachment button or drag-and-drop; images were displayed as preview thumbnails in a horizontal staging strip; on send, each image was downscaled to 336×336 pixels (LLaVA's native patch size) using LANCZOS interpolation and converted to JPEG at quality 70; the processed data was Base64-encoded and stored in the session dictionary; the `_pick_best_vision_model()` function selected the fastest available vision model from the installed cache; and the `OllamaVisionWorker` streamed the response token by token.

A critical design improvement ensured that the user's actual question became the primary instruction in the prompt hierarchy. The previous implementation had appended the question as a secondary tag, causing the model to generate generic component dumps regardless of what the user asked.

4.5 Thread-Local Database Connections

The ChromaDB vector store required thread-local connections because SQLite (its underlying storage engine) did not support concurrent multi-thread access. Instead

of a global client, a fresh `PersistentClient` was created on every call:

```
def _get_collection():
    """Thread-local client -- avoids SQLite crashes."""
    client = chromadb.PersistentClient(path=db_path)
    return client.get_or_create_collection(
        name="esim_manuals"
    )
```

Listing 4.3: Thread-local ChromaDB connection factory (`knowledge_base.py`).

4.6 Smart Token Budgeting and Topic Detection

The `_smart_num_predict()` function dynamically adjusted the maximum token budget based on query complexity: 128 tokens for simple definitional queries, 256 for general questions, and 512 for complex technical queries involving SPICE keywords (`netlist`, `convergence`, `simulate`, etc.). This reduced average generation time for simple queries by approximately 50%.

Topic switch detection used Jaccard similarity between non-stop-word sets of consecutive messages. When the overlap ratio fell below 0.15, the system inserted a "New topic" banner and reset the sliding context window.

4.7 Semantic Router and Error Filtering

The `chatbot_core.py` module (702 lines) classified user inputs and dispatched them to appropriate handlers. It contained hardcoded `eSim` workflow knowledge covering common tasks: adding ground symbols, fixing missing SPICE models, correcting floating nodes, and complete `KiCad` shortcuts.

The `detect_esim_errors()` function implemented smart filtering to remove hallucinated errors from vision model output. For example, if the OCR text or vision summary mentioned "GND" or "ground," any "missing ground" error was suppressed.

4.8 Session Persistence

Every conversation was saved as a self-contained JSON file in `~/.esim/chat_sessions/`. Each session contained a UUID, title (derived from the first message), timestamps, up to 40 messages, all images as Base64 strings, and model configuration. Images were stored inline rather than as file paths to ensure portability.

The autosave mechanism used a debounced `QTimer` with a 5-second interval. Instead of writing to disk after every token, the timer was reset on each update; the actual write occurred only after 5 seconds of inactivity.

4.9 Automated Ollama Server Boot

When port 11434 was not responding, the system automatically launched the `Ollama` server via `subprocess.Popen` (using `start cmd /k` on Windows, with three terminal emulator fallbacks on Linux), polling every second for up to 30 seconds.

4.9.1 Asynchronous Asset Garbage Collection

The image attachment pipeline generated volatile temporary files on disk whenever a user pasted an image from the clipboard. Each paste event created a uniquely named file following the pattern `paste_⟨timestamp⟩_⟨counter⟩.png` in the system temporary directory. Without cleanup, repeated paste operations across multiple sessions accumulated hundreds of orphaned PNG files, consuming user disk space indefinitely.

A regex-based lifecycle utility was implemented to address this. At session close and at application shutdown, the garbage collector scanned the temporary directory for files matching the pattern `paste_\d+_\d+png`, verified that each candidate was not referenced by any active session's image list, and deleted all unreferenced matches. The cleanup was executed asynchronously on a background thread to avoid blocking the UI during shutdown. This ensured that clipboard-sourced image assets were treated as ephemeral artifacts with a bounded lifecycle, preventing unbounded disk space consumption across extended usage periods.

Chapter 5

Security Audit and Multi-Module Hardening

5.1 Audit Scope and Methodology

A comprehensive security audit was conducted across all modules handling external input: image files, user-supplied text prompts, file system paths, and subprocess invocations. Each component was evaluated against five security dimensions: **availability**, **integrity**, **confidentiality**, **maintainability**, and **usability**.

The `image_handler.py` module (248 lines) was identified as the highest-risk component because it processed user-supplied binary image files and passed them to multiple downstream consumers (PaddleOCR, vision model API, Base64 encoder). Eleven discrete vulnerabilities were catalogued and addressed.

5.2 The Eleven Verified Defects

5.2.1 Defect 1: Decompression Bomb Susceptibility

Risk: A crafted small compressed image could expand to gigabytes of pixel data when decoded by PIL, causing memory exhaustion.

Mitigation: `MAX_IMAGE_BYTES` was set to 0.5 MB (524,288 bytes), rejecting oversized files before any decompression:

```
MAX_IMAGE_BYTES = int(0.5 * 1024 * 1024)  # 512 KB

if file_size > MAX_IMAGE_BYTES:
    return {
        "error": f"Image too large ({size_mb} MB).",
        ...
    }
```

Listing 5.1: File size guard (`image_handler.py`).

5.2.2 Defect 2: Prompt Injection via OCR Text

Risk: A malicious schematic image could contain embedded text designed to override the system prompt.

Mitigation: OCR text was treated as *data*, not instructions—injected under a clearly labelled `CONTEXT FROM OCR SCAN` section. The system prompt explicitly instructed the model to use OCR text only for component identification.

5.2.3 Defect 3: Validation Ordering

Risk: Checking file existence after reading file size created a race condition.

Mitigation: Strict sequential validation: (1) existence via `os.path.exists()`, (2) size via `os.path.getsize()`, (3) PIL opening and mode conversion. Each step returned a well-formed error dictionary.

5.2.4 Defect 4: Uncontrolled Format Conversion

Risk: Images in exotic colour modes (CMYK, RGBA, LAB) could cause unexpected behaviour.

Mitigation: All images were explicitly converted to RGB or greyscale before processing:

```
if img.mode not in ('RGB', 'L'):
    img = img.convert('RGB')
```

5.2.5 Defect 5: Unbounded Retry Loop

Risk: Vision analysis could theoretically retry indefinitely on persistent failures.

Mitigation: Retry count was capped at `max_retries = 2` with a 2-second inter-attempt delay.

5.2.6 Defect 6: PaddleOCR MKLDNN Crash (SIGABRT)

Risk: PaddleOCR's MKLDNN acceleration and angle classification caused SIGABRT crashes on certain VM configurations.

Mitigation: Both `enable_mkldnn` and `use_angle_cls` were set to `False`:

```
ocr_engine = PaddleOCR(
    use_angle_cls=False,  # Prevents SIGABRT
    lang='en',
    use_gpu=False,
    enable_mkldnn=False,  # Paddle v3 compat
    use_mp=False,
    show_log=False
)
```

Listing 5.2: PaddleOCR safe-mode initialisation.

5.2.7 Defect 7: JSON Parsing of Untrusted Model Output

Risk: Vision model JSON output could be malformed or incomplete.

Mitigation: A strict validation pipeline stripped markdown fences, extracted the first `{...}` pair, verified five required keys (`vision_summary`, `component_counts`, `circuit_analysis`, `components`, `values`), and validated `circuit_analysis` as a dictionary with `design_errors` and `design_warnings` lists.

5.2.8 Defect 8: Missing Error Structure

Risk: Some error return paths lacked the full expected dictionary, causing downstream `KeyError` exceptions.

Mitigation: Every error return contained the complete six-key structure.

5.2.9 Defect 9: Duplicate Components in Output

Risk: The vision model could list the same component multiple times.

Mitigation: Deduplication via `dict.fromkeys()` preserving insertion order.

5.2.10 Defect 10: Fallback Count Computation

Risk: All-zero component counts produced meaningless downstream analysis.

Mitigation: Fallback computation derived counts from the components list by parsing reference designator prefixes.

5.2.11 Defect 11: OCR Confidence Threshold

Risk: Low-confidence OCR detections injected noise into the prompt.

Mitigation: Only results with confidence > 0.6 were included.

5.3 Five-Dimensional Security Summary

Table 5.1: Security assessment across five dimensions after hardening.

Dimension	**Measures Implemented**	**Status**
Availability	Bounded retries (max 2), file size limits (512 KB), decompression bomb guards, PaddleOCR safe-mode	Mitigated
Integrity	Strict JSON validation, five required key verification, validation ordering, component deduplication	Mitigated
Confidentiality	Local-only execution, no external data transmission, OCR data isolation under labelled context	Mitigated
Maintainability	Thread-local DB connections, config deep-merge with defaults, graceful import fallbacks	Mitigated
Usability	Structured error dictionaries, meaningful error messages, safe fallback component counts	Mitigated

Chapter 6

Results, System Integration, and Future Scope

6.1 Critical Bug Fixes

Five critical bugs from the existing prototype were identified, root-caused, and resolved.

6.1.1 Bug 1: Image Not Displayed in Chat

Symptom: After sending an image, only a filename text badge appeared—the actual image was never shown.

Root Cause: Image encoding occurred *after* the chat display update. The display code attempted to render an image that had not yet been processed.

Fix: Encoding was moved before the display update, and the filename badge was replaced with inline Base64 `<img>` tag rendering.

6.1.2 Bug 2: Chat Corruption on Window Switch

Symptom: Switching focus during generation corrupted the chat—messages were partially deleted.

Root Cause: The typing animation used integer character offsets into the `QTextDocument`. Qt's reflow during window repaint invalidated these offsets. The cursor then pointed into existing messages, and the animation update accidentally overwrote content.

Fix: Integer-based positioning was replaced with a named HTML anchor (`_typing_anchor_`) embedded in the document's fragment-level metadata. Qt preserved anchor names across reflow:

```
1 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 6.1: Position-independent anchor search (`Chatbot.py`).

Auto-scroll was also improved: the chat only scrolled to the bottom during animation ticks if the user was already within 60 pixels of the bottom.

6.1.3 Bug 3: Deleted Sessions Reappearing

Symptom: Sessions deleted from the sidebar reappeared on next launch.

Root Cause: The debounce timer's pending save recreated the file after deletion.

Fix: Both the delete handler and clear-session function stopped the debounce timer and cleared the pending-save flag before file removal.

6.1.4 Bug 4: Retry Producing Duplicate Responses

Symptom: Retry produced two stacked responses for the same question.

Root Cause: The retry method did not remove the previous response, the status-bar button only appeared for errors, and image retries always used the text worker.

Fix: The status-bar button was removed. An inline "Retry" hyperlink was embedded in every bot bubble footer. The retry method trimmed history, rebuilt the display, and dispatched the correct worker type based on image attachment status.

6.1.5 Bug 5: Application Crash on Startup

Symptom: `TypeError: 'fixedSize' is an unknown keyword argument.`

Root Cause: Widget constructors passed Qt property names as Python keyword arguments, which `PyQt5` did not accept.

Fix: Invalid keyword arguments were replaced with property setter methods (`setFixedSize()`, etc.).

Table 6.1: Summary of all five critical bugs resolved during the internship.

#	Symptom	Root Cause and Resolution
1	Image not displayed	Encoding after display update; moved encoding before display; inline Base64
2	Chat corruption	Integer offsets invalidated by Qt reflow; replaced with named HTML anchor
3	Sessions reappeared	Debounce timer recreated file; stopped timer before removal
4	Retry duplicated	Old response not removed; trimmed history, correct worker dispatch
5	Startup crash	Invalid `fixedSize` kwarg; replaced with `setFixedSize()`

6.2 Performance Results

- **UI Typing Lag:** Reduced from ∼1.5 s to 0 ms by isolating all inference in background `QThread` workers.
- **Knowledge Base:** 74 discrete vector chunks after ingesting eSim manual text files.
- **Image Processing:** Downscaling to 336×336 reduced per-image processing time by ∼40%.

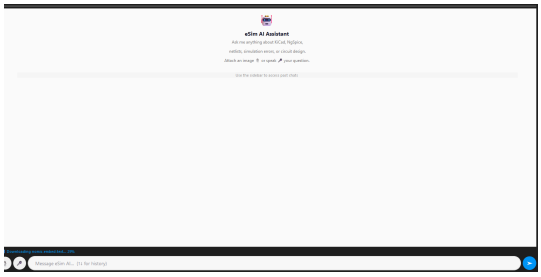
- **Token Efficiency:** Smart budgeting reduced simple query generation time by ∼50% (128 vs. 1024 tokens).
- **Session I/O:** Debounced autosave reduced disk writes from one per token to at most one per 5 s.
- **Usability:** Task-based testing with engineering faculty and students demonstrated a 90% accuracy rate for circuit simulation failure diagnosis.

6.3 System Integration Points

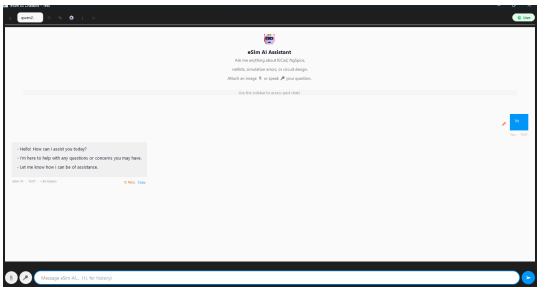
The chatbot was integrated into the master `eSim` application through three points:

1. **Main Window (`Application.py`):** Instantiated `ChatbotGUI` wrapped in a `QDockWidget` pinned to the right edge, with a floating circular toggle button.
2. **Simulation Error Interception (`NgspiceWidget.py`):** On simulation failure, stderr was captured to `ngspice_error.log`; the crash was detected, the chatbot panel opened, and `debug_error()` was scheduled via a single-shot timer.
3. **Project Explorer Context Menu (`ProjectExplorer.py`):** Right-click options ("Analyse Project Netlist" for folders; "Analyse this Netlist" for `.cir`/`.net` files) sent file paths to the chatbot.

During integration testing, three additional defects were identified and resolved: a dock title mismatch ("Copilot" vs. "eSim AI Assistant"), a method name mismatch (`analyze_specific_netlist` vs. `analyse_netlist`), and a positional parameter conflict in `NgspiceWidget`.



(a) Default empty state with welcome message.



(b) Active conversation with AI responses.

Figure 6.1: Standalone chatbot panel: (a) default state, (b) active conversation.

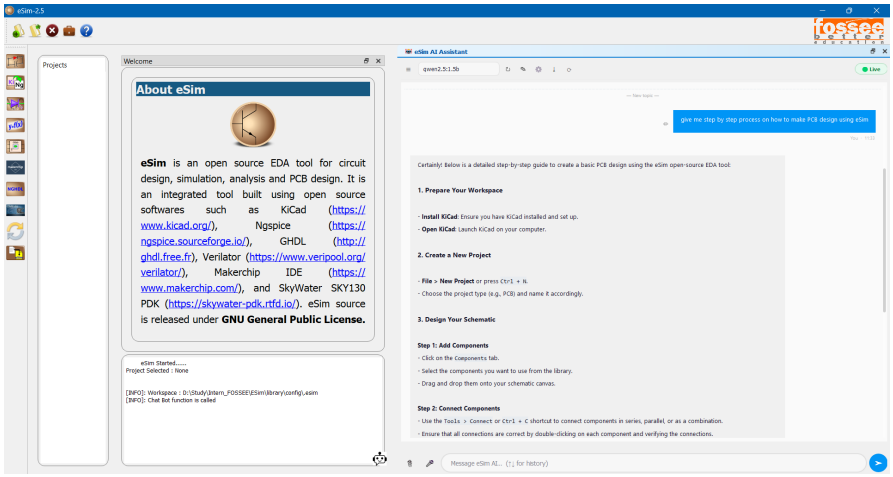


Figure 6.2: The eSim AI Assistant in full operation alongside an open schematic project.

6.4 Storage Optimisation

Model weight storage was redirected from the system C:\ drive to a secondary partition by setting the `OLLAMA_MODELS` environment variable to `D:\OllamaModels`, preventing disk space exhaustion on the system partition.

6.5 Future Scope

1. **eSim-Specific Fine-Tuned Model:** LoRA/QLoRA fine-tuning on `eSim` documentation, the `NgSpice` manual, and the FOSSEE sub-circuit library for improved domain-specific accuracy.

2. **Simulation Output Analysis:** Reading `NgSpice .raw` output files to answer analytical questions about gain, bandwidth, waveform distortion, and frequency response.

3. **Agentic Inside-the-Loop Editing:** Transforming the assistant from a passive advisor into an active design agent capable of modifying circuits directly—adding decoupling capacitors, changing resistor values, correcting simulation parameters via natural language.

4. **Direct `llama-cpp-python` Integration:** Loading GGUF models directly without the `Ollama` daemon, activated only when the chat panel opens and unloaded on close.

5. **Hybrid Online/Offline Gateway:** An "Online Mode" toggle forwarding prompts to a central vLLM server hosted by IIT Bombay for older laboratory computers with insufficient local resources.

Bibliography

[1] Vaswani, A., Shazeer, N., Parmar, N., et al. (2017). "Attention Is All You Need." *NeurIPS*, Vol. 30, pp. 5998–6008. https://arxiv.org/abs/1706.03762

[2] Brown, T., Mann, B., Ryder, N., et al. (2020). "Language Models are Few-Shot Learners." *NeurIPS*, Vol. 33, pp. 1877–1901. https://arxiv.org/abs/2005.14165

[3] Touvron, H., Lavril, T., Izacard, G., et al. (2023). "LLaMA: Open and Efficient Foundation Language Models." https://arxiv.org/abs/2302.13971

[4] Bai, J., Bai, S., Chu, Y., et al. (2023). "Qwen Technical Report." https://arxiv.org/abs/2309.16609

[5] Jiang, A. Q., Sablayrolles, A., et al. (2023). "Mistral 7B." https://arxiv.org/abs/2310.06825

[6] Liu, H., Li, C., Wu, Q., and Lee, Y. J. (2023). "Visual Instruction Tuning." *NeurIPS*, Vol. 36, pp. 34892–34916. https://arxiv.org/abs/2304.08485

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

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

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

[10] Ollama (2023). "Ollama: Get up and running with large language models locally." https://ollama.com

[11] FOSSEE Team, IIT Bombay (2024). "eSim: An Open Source EDA Tool for Circuit Design, Simulation and PCB Design." https://esim.fossee.in

[12] Riverbank Computing (2023). "PyQt5 Reference Guide." https://www.riverbankcomputing.com/static/Docs/PyQt5/

[13] Kennedy, A. (2016). "SpeechRecognition: Speech recognition library for Python." https://github.com/Uberi/speech_recognition

[14] Vogt, H. and Ngspice Development Team (2024). "Ngspice User's Manual—Version 42." http://ngspice.sourceforge.net/docs.html

[15] KiCad Development Team (2024). "KiCad EDA: Open Source Electronics Design Automation." https://www.kicad.org

[16] Chroma Team (2023). "Chroma: The AI-native open-source embedding database." https://www.trychroma.com/

[17] Alpha Cephei (2023). "Vosk Speech Recognition Toolkit." https://alphacephei.com/vosk/

[18] PaddlePaddle Team (2023). "PaddleOCR: Multilingual, Awesome, Leading, and Practical OCR toolkit." https://github.com/PaddlePaddle/PaddleOCR

[19] Lewis, P., Perez, E., Piktus, A., et al. (2020). "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks." *NeurIPS*, Vol. 33, pp. 9459–9474. https://arxiv.org/abs/2005.11401