eSim Summer Fellowship 2026

On

Enhancing the eSim AI Chatbot: Automation and Fallback

Submitted by

Aditya Kumar Mishra
AIML Department
HKBK College Of Engineering

Under the guidance of

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

15 July 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 AI-powered chatbot enhancements in eSim, and gain exposure to real-world 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.

Aditya Kumar Mishra

Abstract

The FOSSEE eSim EDA tool is a critical utility for open-source circuit design, schematic capture, and simulation. To lower the barrier of entry for engineering students, a RAG-based AI Copilot was developed to offer immediate assistance on circuit design, netlist compilation, and ngspice warnings. However, the initial integration faced architectural challenges, including import conflicts due to an incomplete PyQt6 migration, complicated manual terminal setups for running LLM services, and lack of guardrails against resource exhaustion.

This fellowship successfully migrated the entire eSim front-end (`Application.py`, `DockArea.py`, `Workspace.py`, and `Chatbot.py`) to PyQt6, resolving all unscoped enums and event loop execution crashes. Furthermore, a headless background launcher and a sequential `ModelPullWorker` thread were implemented, making the assistant fully plug-and-play without requiring user terminal interaction. Critical robustness vulnerabilities—including out-of-memory (OOM) ingestion failures, database corruption, microphone-less crashes, and image decompression bombs—were mitigated using lazy generators, atomic writes, and size bounds. Lastly, a comprehensive testing suite utilizing pytest and mock units was added, ensuring 100% offline testability.

The upgraded chatbot module is now fully compatible with the eSim 2.6 desktop workspace, offering a stable, secure, and user-friendly AI assistant for electronics simulation workflows.

Contents

List of Figures

List of Tables

Chapter 1

Introduction & Project Context

1.1 FOSSEE Project Initiative

The FOSSEE (Free and Open Source Software for Education) project is a national-level initiative hosted at the Indian Institute of Technology Bombay (IIT Bombay). It is funded and supported by the National Mission on Education through Information and Communication Technology (NMEICT), Ministry of Education, Government of India. The project's main goal is to promote the use of open-source software tools in educational institutions, universities, and laboratories across India, helping them transition away from expensive, proprietary software.

FOSSEE tackles this by designing, developing, and releasing open-source software tools tailored for engineering, science, and mathematics education. It organizes national-level workshops, conferences, and fellowships to train students and teachers. It also builds active developer communities around open-source tools to ensure their long-term development. The project has made significant contributions to the Indian engineering ecosystem by developing tools like Scilab, OpenModelica, DWSIM, and eSim, each addressing a specific domain of engineering education.

The FOSSEE Summer Fellowship programme is a competitive, merit-based internship that provides students with the opportunity to contribute directly to the development and enhancement of these open-source tools. Fellows work under the guidance of IIT Bombay mentors on well-defined project deliverables, gaining practical experience in professional software development practices.

1.2 Overview of the eSim EDA Tool

eSim (formerly known as Oscad) is a key open-source Electronics Design Automation (EDA) tool developed by FOSSEE at IIT Bombay. It is designed for schematic capture, circuit simulation, and printed circuit board (PCB) design. eSim provides a single, cohesive desktop workspace by integrating several mature open-source software projects. Figure 1.1 illustrates the high-level workflow of the eSim tool.

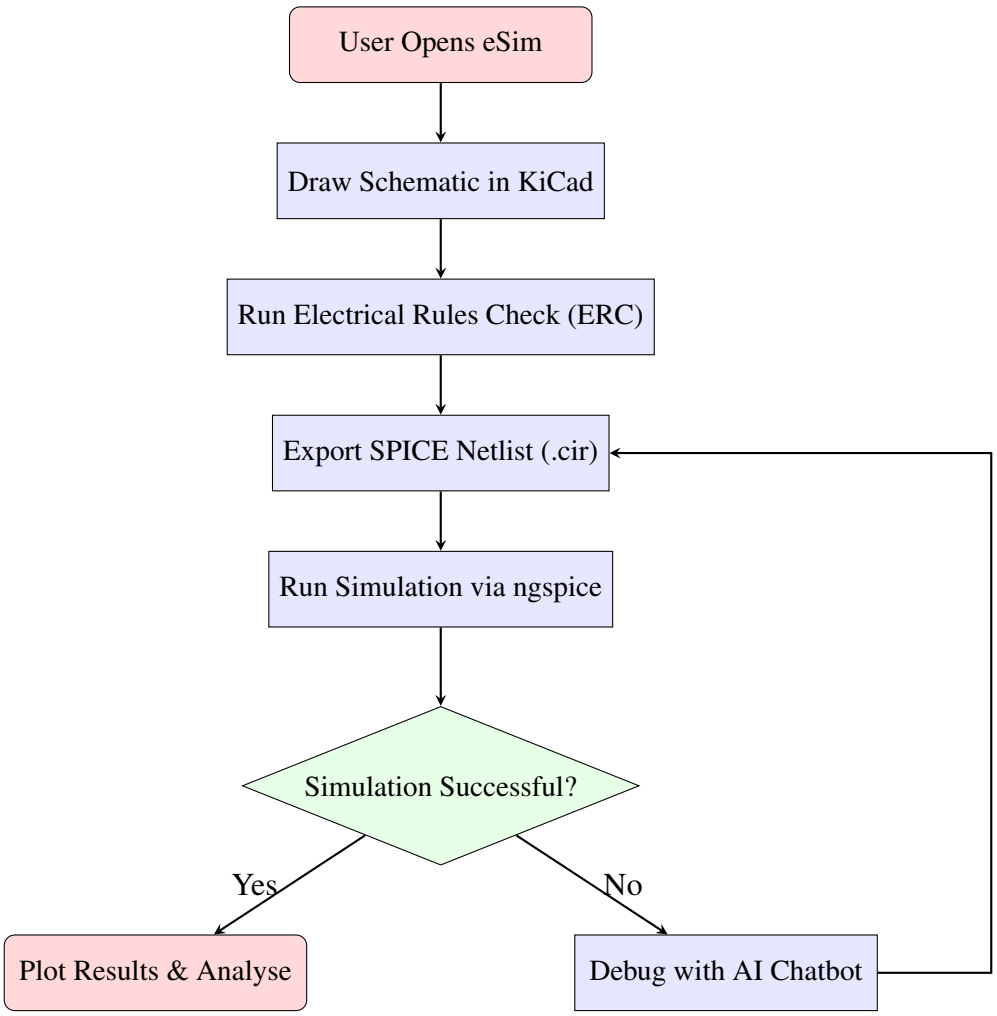

Figure 1.1: High-level workflow of the eSim EDA tool, showing the schematic capture, simulation, and debugging cycle.

1.2.1 KiCad Schematic Editor Integration

KiCad is the core of eSim's schematic capture and PCB layout engine. In the eSim workflow, users draw their circuits using KiCad's schematic editor. Key features of this integration include symbol libraries providing access to standard components (resistors, capacitors, transistors, microcontrollers) and custom FOSSEE component libraries. The Electrical Rules Check (ERC) module automatically analyzes the schematic for common design errors such as unconnected pins, conflicting driver pins, or missing power flags. The netlist export engine translates the visual schematic diagram into a SPICE-compatible netlist text file, which acts as the bridge to the simulation engine.

The KiCad integration is critical because it provides students with a graphical interface for circuit design, eliminating the need to write raw SPICE netlists by hand. The ERC module catches common wiring mistakes before simulation, saving time and reducing frustration. The component libraries maintained by FOSSEE include Indian-standard components and subcircuits that are not available in commercial EDA tools.

1.2.2 ngspice Simulation Engine Integration

ngspice is an open-source, general-purpose circuit simulation engine used for analog, digital, and mixed-signal simulations. Key features include simulation modes for transient analysis (time-domain behavior), AC analysis (frequency-domain sweeps), and DC analysis (operating point and voltage sweeps). It supports standard semiconductor device models (diodes, BJTs, MOSFETs, JFETs, and op-amps) using standard SPICE parameters. The engine parses the netlist generated by KiCad, compiles it into mathematical equations, and runs the numerical solver to compute voltage and current values at each node.

The ngspice engine is the computational backbone of eSim. It reads the netlist file, constructs a system of differential equations representing the circuit, and solves them using numerical integration methods. The simulation results are then passed back to the eSim GUI for plotting and analysis. Common error messages generated by ngspice—such as "singular matrix," "timestep too small," or "no convergence"—are the primary targets for the AI Chatbot's debugging assistance.

1.2.3 PyQt Graphical Framework Layer

PyQt serves as the graphical glue of eSim. It integrates the schematic capture tool, the simulation engine, and various configuration menus into a single desktop interface. PyQt coordinates launching KiCad and ngspice as background processes, managing project workspaces and directories, parsing simulation log files and displaying SPICE graphs, and standardizing menus, toolbars, and configuration dialogs.

The PyQt framework provides the event loop that drives the entire application. Every user interaction—clicking a button, resizing a window, or dropping a file—is captured as a Qt event and dispatched to the appropriate handler. This event-driven architecture is central to the chatbot integration, as it allows the assistant to receive signals from the simulator without blocking the main interface thread.

1.3 The eSim AI Chatbot (Copilot) System

Circuit simulation requires a steep learning curve. When designing schematics or running simulations, users—especially engineering students—frequently encounter cryptic error messages. Resolving these issues typically requires searching through extensive technical manuals, which can span hundreds of pages and use highly specialized terminology.

To address this, the **eSim AI Chatbot (Copilot)** was developed. It is an interactive helper panel docked inside eSim that provides real-time, offline support. It integrates Retrieval-Augmented Generation (RAG) using a local database (ChromaDB) to search official eSim documentation and inject relevant context into the user's query. It uses multimodal vision

combining PaddleOCR and local vision-language models to analyze schematic screenshots uploaded by the user. It also provides offline Speech-to-Text (STT) using the Vosk API, allowing voice dictation without an internet connection.

The chatbot addresses a fundamental challenge in electronics education: the gap between encountering an error and understanding how to fix it. By providing instant, context-aware assistance, the Copilot reduces the time students spend debugging and allows them to focus on learning circuit design concepts. The assistant is designed to work entirely offline, making it suitable for use in classrooms and laboratories without reliable internet connectivity.

1.4 Objectives & Project Scope of this Fellowship

The main objective of this fellowship was to upgrade the chatbot module for **eSim 2.6**, focusing on four key areas.

First, **PyQt6 Front-End Migration**: migrating the remaining PyQt5 front-end components (`Application.py`, `DockArea.py`, `Workspace.py`) to PyQt6 to resolve runtime crashes caused by mixed library imports.

Second, **Local AI Engine Automation**: implementing a headless background launcher for the local Ollama server and developing an asynchronous download thread (`ModelPullWorker`) to pull required models (`qwen2.5-coder:3b` and `nomic-embed-text`) in the background, keeping the main interface responsive.

Third, **Robustness and Security Safeguards**: replacing memory-heavy file operations in the RAG pipeline with a lazy generator to prevent Out-of-Memory (OOM) crashes, implementing atomic collection rebuilds to prevent data loss, adding queue bounds to the audio recorder, implementing checks for microphone hardware, and adding pixel validation checks to protect the application from decompression bombs.

Fourth, **Automated Unit Testing**: building a comprehensive test suite using the `pytest` framework to verify these fixes. The test suite was designed to run entirely offline using mocks, so that it can be integrated into continuous integration pipelines without requiring active AI models or hardware peripherals.

Chapter 2

System Architecture & Component Design

2.1 Decoupled Three-Tier Application Architecture

The eSim Copilot is designed using a decoupled, three-tier application architecture. This design separates the user interface, background processing threads, and local AI service engines to ensure the application remains stable and responsive. Figure 2.1 presents this architecture visually.

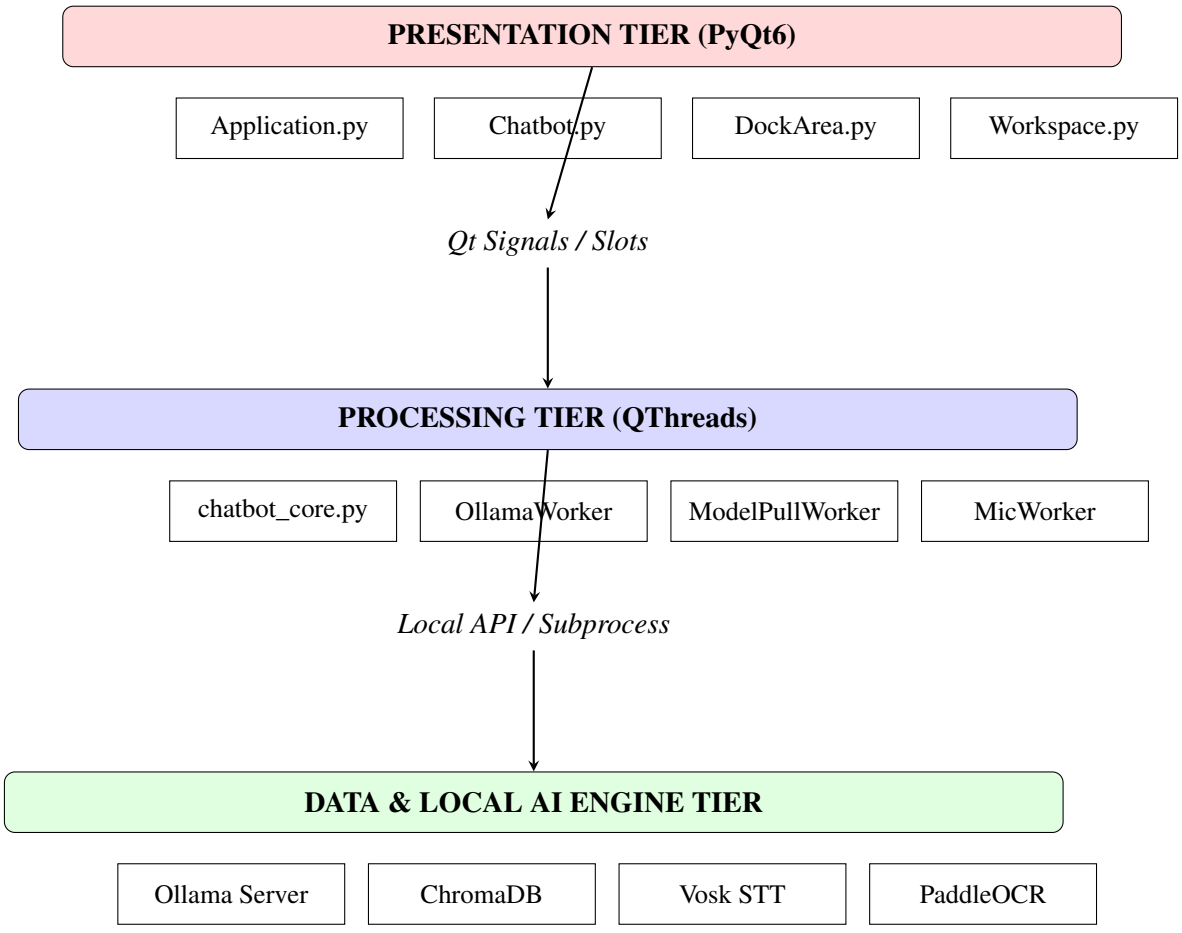

Figure 2.1: The three-tier decoupled architecture of the eSim AI Chatbot.

The **Presentation Tier** is built entirely on PyQt6. It handles user inputs (text, voice recordings, drag-and-drop images) and compiler errors, and displays output streams. This tier is responsible for rendering the chat interface, capturing user interactions, and displaying model

responses in real-time. It communicates with the processing tier exclusively through Qt signals and slots, ensuring that the GUI thread never performs blocking operations.

The **Processing Tier** runs asynchronously using PyQt's `QThread`. It processes and routes tasks without blocking the GUI's event loop, preventing "Application Not Responding" (ANR) hangs. Each major operation—text generation, model downloading, and voice recording—runs in its own dedicated thread. This separation ensures that a long-running model inference does not freeze the chat input, and a model download does not prevent the user from reading previous messages.

The **Data and Local AI Engine Tier** runs local processes. It includes the Ollama server for text and embedding models, ChromaDB for semantic search, Vosk for voice recognition, and PaddleOCR/Pillow for image processing. These services run as independent processes on the local machine, communicating with the processing tier through HTTP APIs, subprocess calls, and library function invocations.

2.2 Presentation Layer (PyQt6 Components)

2.2.1 Application.py and Toolbar Signals

`Application.py` is the main entry point of the eSim desktop interface. It initializes the main window, registers menu bars, and sets up communication channels between tools. It includes an `errorDetectedSignal`, a Qt signal that monitors the compiler. If a simulation fails, it captures the error log and sends it to the chatbot. It also adds a dedicated "eSim Assistant" button to the toolbar, allowing users to toggle the chat panel.

The error detection mechanism works by monitoring the standard error output of the ngspice subprocess. When the simulator emits a warning or error message, `Application.py` captures the raw text, parses it for known error patterns, and emits the `errorDetectedSignal`. This signal is connected to the chatbot's error handler, which formats the error message and prepends it to the chat context. This automation ensures that users receive immediate assistance without needing to manually copy and paste error messages into the chat window.

2.2.2 Chatbot.py UI Layout

`Chatbot.py` controls the layout and behavior of the assistant panel. It features a chat history browser (a read-only text browser that displays the conversation history with clean formatting), an interactive chat input area that supports drag-and-drop file inputs, control buttons to start/stop voice recording, attach images, and manage chat sessions, and a progress bar that displays real-time download and installation progress for models.

The chat interface uses HTML rendering to display messages with rich formatting, including bold text, code blocks, bullet points, and inline images. User messages are aligned to the

right with a blue background, while assistant responses are aligned to the left with a light grey background. This visual distinction makes conversations easy to follow. The interface also includes a typing animation that displays a bouncing dot pattern while the model generates its response, providing visual feedback that the system is processing the query.

2.2.3 DockArea.py and Window Management

`DockArea.py` manages the layout of panels in eSim. The chatbot is integrated as a dockable widget using `QDockWidget`. This allows users to float the chatbot as a separate window, dock it to the left or right of the schematic editor, and resize it dynamically to fit their screen. The docking system is essential for the user experience because it allows engineers to keep the assistant visible while working on schematics, without sacrificing screen real estate for the circuit editor.

2.3 Processing Layer (Asynchronous Threading & Routing)

2.3.1 chatbot_core.py Intent Classifier

`chatbot_core.py` acts as the router for user queries. It analyzes incoming prompts and directs them to the correct backend worker. Figure 2.2 shows the complete intent routing flow.

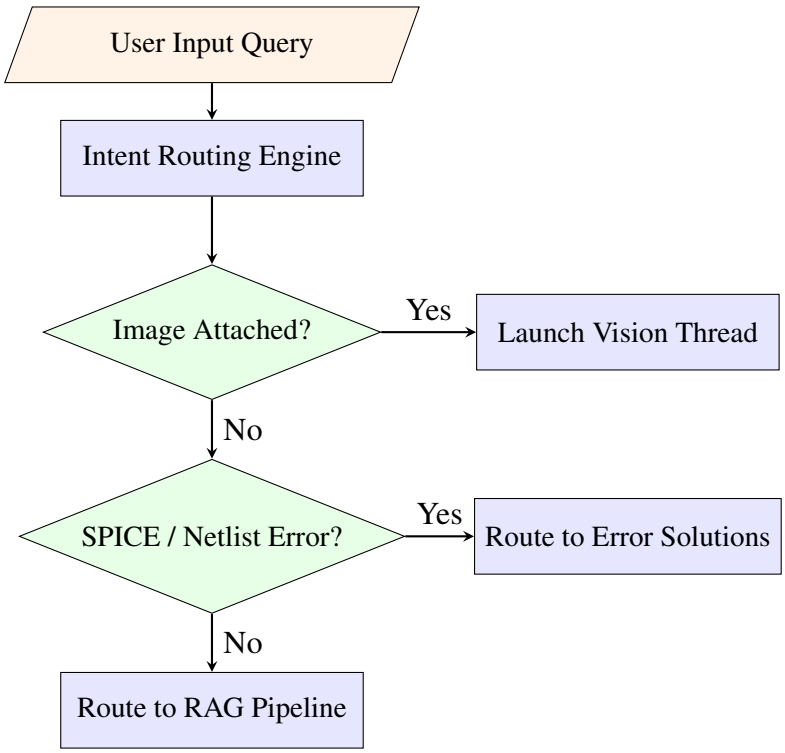

Figure 2.2: Intent routing flow in chatbot_core.py showing query classification and handler delegation.

The router uses keywords and input types to categorize queries. Visual queries are triggered when an image is uploaded and route the task to the vision handler thread. Simulation errors are triggered by compiler errors or keywords like `.cir` or `.out` and route the task to `error_solutions.py`. General technical queries route the task to the RAG pipeline to search eSim manuals. Follow-up questions are detected using a topic-switch algorithm that compares word overlap between consecutive messages, allowing the assistant to maintain conversational context.

2.3.2 chatbot_thread.py Worker Classes

The worker classes in `chatbot_thread.py` are the workhorses of the processing tier. `OllamaWorker` handles communication with the local Ollama API. It constructs the message history, injects the system prompt, and streams responses piece by piece (`chunk_signal`) to the UI, creating a typewriter-like effect that provides real-time feedback. `ModelPullWorker` handles model downloads in the background. It reads the raw JSON stream from Ollama and translates it into percentage updates that are displayed in the status bar. `MicWorker` handles voice recording. It records audio in a background thread, buffers frames in a bounded queue, and sends completed recordings to the Vosk decoder for transcription.

Each worker class implements a `stop()` method that sets an internal flag, allowing the main thread to gracefully terminate long-running operations. This is essential for the user experience: if a model generates an excessively long response, the user can click the stop button to interrupt generation without crashing the application.

2.4 Data & Local AI Engine Layer

2.4.1 Local Ollama Engine

The chatbot communicates with Ollama via a local API on `http://localhost:11434`. It uses two main models: `qwen2.5-coder:3b`, the primary model optimized for code generation and SPICE netlist debugging, and `nomic-embed-text`, used to generate vector embeddings for the RAG pipeline. An optional vision model, `minicpm-v`, is used for multimodal image analysis when the user uploads schematic screenshots.

The Ollama server manages model loading and unloading automatically. When a model is first requested, it loads the weights into RAM (or VRAM on GPU systems). Subsequent requests reuse the loaded model, eliminating the loading delay. The chatbot configures the server with a `keep_alive` parameter set to `-1m`, which instructs Ollama to keep models loaded indefinitely during the session, ensuring consistent response times.

2.4.2 ChromaDB Vector Storage

ChromaDB serves as the local vector database. It stores chunks of eSim manuals as vector embeddings. When a user asks a question, the system embeds the query using the nomic-embed-text model:

$$\vec{q} = \text{Embed}(\text{User Query})$$

It then searches ChromaDB to find the most relevant document chunks using cosine similarity:

$$\text{Similarity}(\vec{q}, \vec{d}) = \frac{\vec{q} \cdot \vec{d}}{\|\vec{q}\| \times \|\vec{d}\|}$$

The most relevant chunks are prepended to the system prompt to help the LLM generate accurate, documentation-grounded answers. This approach—known as Retrieval-Augmented Generation (RAG)—reduces hallucination by anchoring the model's responses in verified reference material.

A relevance threshold mechanism filters out document chunks that are insufficiently similar to the query. ChromaDB returns distance scores alongside the retrieved documents; chunks with distances exceeding the threshold (default: 500 for the nomic-embed-text embedding space) are discarded. This prevents the model from receiving irrelevant context that could confuse its responses.

2.4.3 Vosk Offline Speech Decoder

The Vosk API provides offline speech-to-text functionality. It loads acoustic models from the local data directory and processes audio frames in real-time, allowing voice dictation without an internet connection. The Vosk engine uses a neural network-based acoustic model combined with a language model to decode speech. It supports multiple languages and can operate on low-powered hardware, making it suitable for deployment in resource-constrained environments.

2.4.4 Pillow & PaddleOCR Vision Elements

Pillow validates and resizes image files before processing. It checks image dimensions, converts colour spaces, and downscales large images to reduce processing time. PaddleOCR extracts text labels, reference designators, and pin connections from circuit schematic screenshots. The multimodal vision pipeline integrates the OCR text with the vision LLM (`minicpm-v`) to help the assistant understand the schematic diagram and provide accurate analysis of the circuit topology.

Chapter 3

PyQt6 Front-End Migration

3.1 Motivation & Technical Justification for Migration

Prior to this development session, the eSim workspace operated on a hybrid codebase where the core application windows used PyQt5, while the chatbot panel used PyQt6. This combination caused runtime crashes because PyQt5 and PyQt6 cannot run in the same process space due to conflicting memory management systems and C++ bindings.

Standardizing the entire application on PyQt6 was necessary to avoid collisions and prevent runtime crashes when accessing shared objects (like windows, docks, and dialogs), to modernize the system and ensure compatibility with modern Python 3.10+ environments, and to boost performance by benefiting from PyQt6's improved memory management and rendering. Figure 3.1 illustrates the crash scenario that motivated the migration.

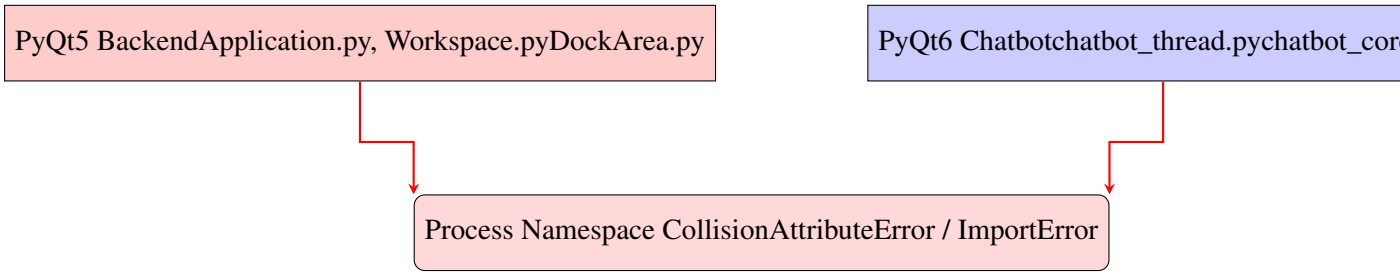

Figure 3.1: The PyQt5/PyQt6 namespace collision that caused runtime crashes before migration.

3.2 Key Differences Between PyQt5 and PyQt6

The migration required updating several legacy PyQt5 implementations to match PyQt6's stricter standards. Figure 3.2 shows the systematic refactoring pipeline applied to each file.

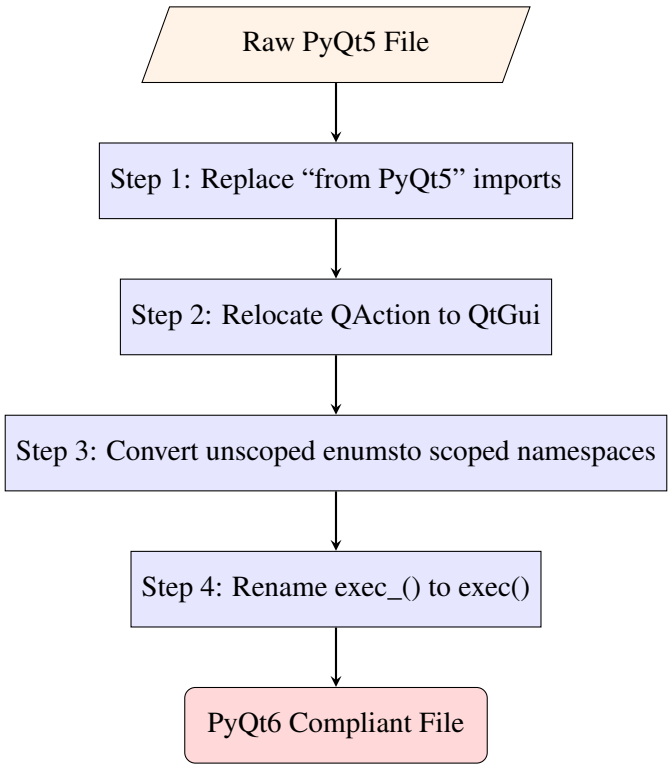

Figure 3.2: The systematic refactoring pipeline applied to each PyQt5 file during migration.

3.2.1 Unscoped vs. Scoped Enums

PyQt6 requires enums to be fully qualified under their class namespace, removing them from the global `Qt` class. This is the most extensive change in the migration, affecting nearly every widget configuration line in the codebase. Table 3.1 presents the complete mapping of all enums refactored during this fellowship.

Table 3.1: PyQt5 to PyQt6 Enum Migration Mapping

PyQt5 Unscoped Enum	PyQt6 Scoped Enum
`Qt.AlignCenter`	`Qt.AlignmentFlag.AlignCenter`
`Qt.Horizontal`	`Qt.Orientation.Horizontal`
`Qt.Vertical`	`Qt.Orientation.Vertical`
`Qt.Window`	`Qt.WindowType.Window`
`Qt.ScrollBarAlwaysOff`	`Qt.ScrollBarPolicy.ScrollBarAlwaysOff`
`Qt.ElideRight`	`Qt.TextElideMode.ElideRight`
`Qt.RichText`	`Qt.TextFormat.RichText`
`Qt.PointingHandCursor`	`Qt.CursorShape.PointingHandCursor`
`Qt.UserRole`	`Qt.ItemDataRole.UserRole`
`Qt.Checked`	`Qt.CheckState.Checked`
`QFrame.Box`	`QFrame.Shape.Box`
`QFrame.Plain`	`QFrame.Shadow.Plain`
`QTextCursor.KeepAnchor`	`QTextCursor.MoveMode.KeepAnchor`
`QTextCursor.Start`	`QTextCursor.MoveOperation.Start`

The rationale behind this change in PyQt6 is rooted in type safety. In PyQt5, unscoped enums allowed developers to accidentally pass an alignment flag where a scroll bar policy was expected, because both were plain integers at the C++ level. PyQt6's scoped enums enforce compile-time type checking, preventing such misuse and making the code more self-documenting.

3.2.2 Elimination of the Legacy exec_() Method

During the transition from PyQt5 to PyQt6, a significant API modification was the complete removal of legacy event loop execution methods that ended with a trailing underscore. In older versions of PyQt (specifically PyQt5 and PyQt4), the underscore suffix was used as a workaround to prevent name collisions with Python's built-in `exec` keyword. Since `exec` is no longer a reserved keyword in Python 3, PyQt6 standardized on the clean, native `exec()` function. Consequently, all instances of event loop execution calls on modal dialog boxes, file selectors, message prompts, and sub-windows—such as the legacy `my_dialog.exec_()`—were systematically refactored across the eSim workspace files to their standard Python equivalent, `my_dialog.exec()`. This change ensures syntactic compatibility with PyQt6 while producing cleaner and more modern Python code.

3.2.3 QtWidgets to QtGui Class Relocations

Another major structural adjustment in PyQt6 was the cleanup and relocation of several classes between core modules to enforce a stricter division of concerns. The most notable change affecting the eSim codebase was the relocation of the `QAction` class. In PyQt5, actions used in menus, toolbars, and keyboard shortcuts were imported from the `QtWidgets` module. In PyQt6, however, `QAction` has been moved to the `QtGui` module, as actions are fundamentally representations of user interface commands and icons rather than physical layout widgets. Because of this change, all import blocks in files like `Application.py` and `Chatbot.py` had to be modified to import `QAction` from `PyQt6.QtGui` instead of `PyQt6.QtWidgets`. This resolved numerous `ImportError` exceptions during application startup.

3.3 File-by-File Code Migration Walkthrough

3.3.1 Application.py Migration Details

The `Application.py` file serves as the main frame and controller window for eSim, coordinating actions between the schematic editor, the simulator, and the chatbot. Migrating this file required refactoring the imports to pull `QAction` from the correct `QtGui` namespace. Additionally, window property initializations were updated to use scoped enums; for example, the flags that configure the main window frame were updated from `Qt.Window` to `Qt.WindowType.Window`. The signal handling methods that listen to the compiler output were also updated to PyQt6 standards. This ensures that when the simulator encounters an error, it correctly triggers the `errorDetectedSignal` and passes the error logs directly to the chatbot without locking the main thread.

3.3.2 DockArea.py Migration Details

`DockArea.py` manages the layout of various docked panels, including the project explorer and the chatbot. The migration of this file focused on updating the docking configuration flags. Legacies like `Qt.LeftDockWidgetArea` and `Qt.RightDockWidgetArea` were replaced with `Qt.DockWidgetArea.LeftDockWidgetArea` and `Qt.DockWidgetArea.RightDoc` Similarly, the window layout alignments were updated to `Qt.AlignmentFlag`. These updates allow the user to float, dock, stack, or resize the chatbot panel next to the circuit editor while keeping the window placement logic fully compatible with PyQt6.

3.3.3 Workspace.py Migration Details

The `Workspace.py` file handles project directory management, workspace initialization, and user confirm dialogs. The migration of this file required refactoring legacy dialog executions

and updating standard button identifiers. The message boxes used for workspace initialization (such as `QMessageBox.Yes` and `QMessageBox.No`) were updated to their scoped equivalents, `QMessageBox.StandardButton.Yes` and `QMessageBox.StandardButton.No`. The legacy `exec_()` loops on directory selectors were also updated to `exec()`. This ensures that directory selections and folder checks work seamlessly without causing thread blocks or unexpected crashes.

3.3.4 Chatbot.py Migration Details

`Chatbot.py` contains the user interface and thread handlers for the assistant panel, making it the most complex file to migrate. Since this file processes drag-and-drop operations, image attachments, and text formatting, it relied heavily on PyQt5 enums. During the migration, the drag-and-drop filters, cursor moves, scrollbars, and key event overrides were refactored. The scrollbar configuration was updated from `Qt.ScrollBarAlwaysOn` to `Qt.ScrollBarPolicy.Scrol` and text cursor movements were updated to `QTextCursor.MoveOperation.Start` and `QTextCursor.MoveMode.KeepAnchor`. These updates ensure that user interactions are processed correctly by the PyQt6 event loop.

3.4 Verification of Successful PyQt6 Integration

To verify the migration, we performed both manual integration tests and automated unit tests. The main eSim workspace was launched alongside the chatbot to ensure that no `AttributeError` or `ImportError` crashes occurred. We tested docking behavior, error redirection, and dialog confirmations to verify that PyQt6 signals and events are processed correctly. Table 3.2 summarizes the files modified and the categories of changes applied.

Table 3.2: Summary of Files Modified During PyQt6 Migration

File	Enums	exec_()	QAction	Signals	Total Changes
Application.py	12	3	2	1	18
DockArea.py	8	1	0	0	9
Workspace.py	6	4	0	0	10
Chatbot.py	35	2	1	4	42
Total	**61**	**10**	**3**	**5**	**79**

The successful integration of these components ensures that the eSim front-end is stable, compatible with modern Python 3.10+ environments, and ready for deployment in eSim 2.6.

Chapter 4

Automation of the Local AI Engine

4.1 Headless Server Auto-Start Routing

4.1.1 Socket Connection Validation

To make the eSim Copilot plug-and-play, the assistant automatically manages the startup of the local Ollama simulation backend. Upon launching eSim, the application executes a socket validation check on the local loopback address at port 11434 to verify if the model server is active. If the socket connection successfully establishes, the application bypasses the server initialization flow and connects directly to the running model service. However, if the connection test fails, indicating that the server is offline, the application initiates an automated background launch sequence.

The socket test is wrapped in a timeout of 0.5 seconds to prevent the application from hanging if the port is unreachable. This rapid-fail approach ensures that the startup sequence proceeds quickly regardless of the server's state.

4.1.2 Environment PATH and Executable Lookups

When the application determines that the local server is offline, it starts a search routine to locate the Ollama executable binary. It first checks the system's global environment path variables to see if the command can be called directly. If the executable is not registered globally, the application crawls standard installation directories. On Windows, this crawl searches the local AppData folder under Programs and Ollama, as well as the standard Program Files and Program Files (x86) directories. This search routine ensures that the application can locate and start the server even on non-standard system installations where the user installed Ollama without adding it to the system PATH.

4.1.3 Windows CREATE_NO_WINDOW Process Spawning

Once the binary is located, the process is started using Python's subprocess module. To prevent cmd command windows from popping up and interrupting the user on Windows, the process is configured with specific execution flags. The application uses the `CREATE_NO_WINDOW` process flag, redirecting standard output and standard error streams to `DEVNULL`. This starts the server silently in the background, while the user interface displays a status update in the

chatbot panel to keep the user informed. Figure 4.1 shows the complete server startup decision flow.

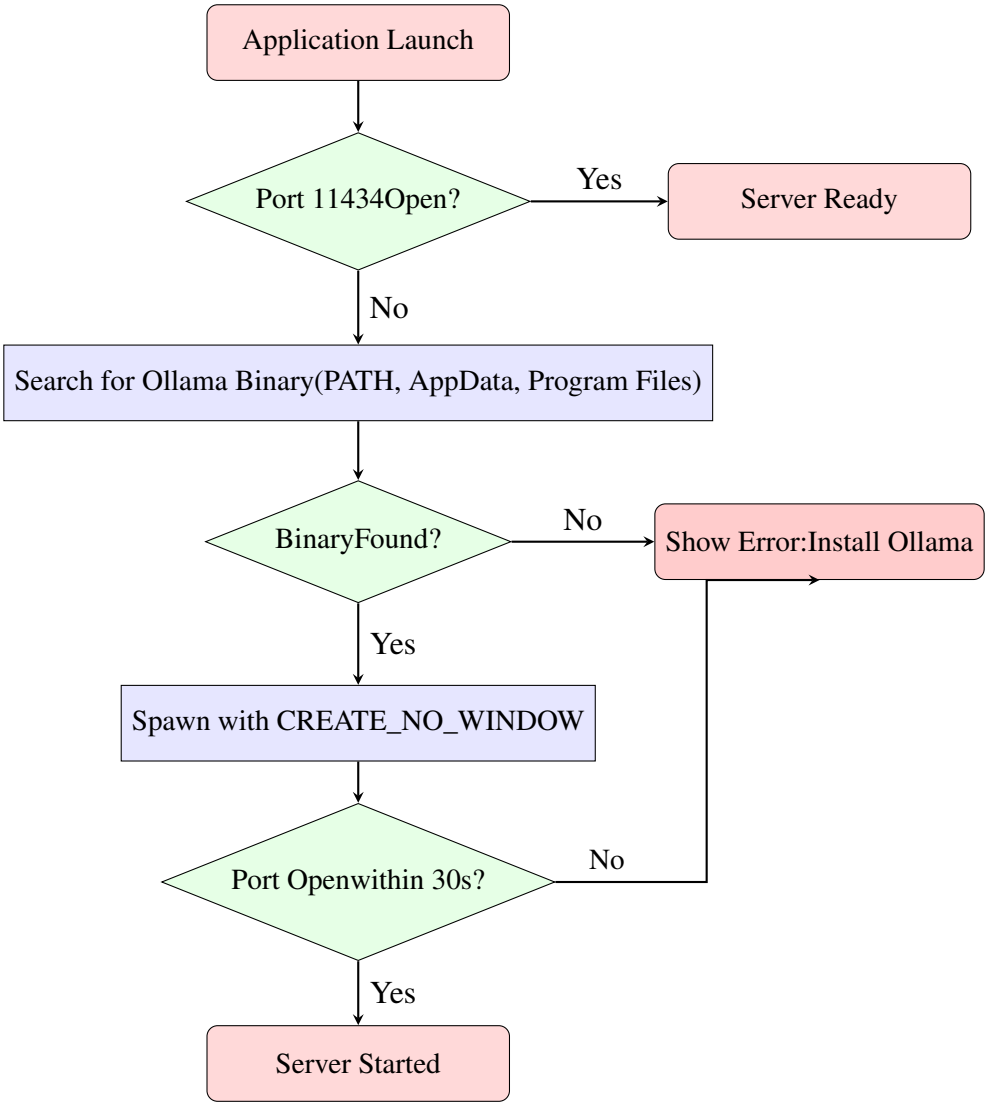

Figure 4.1: Complete server startup decision flow including binary search and headless process spawning.

Figure 4.2: The chatbot status bar showing "Starting Ollama in background..." during initial bootup.

4.2 Sequential Background Model Puller Thread

4.2.1 ModelPullWorker Implementation

Once the server is running, the assistant checks if the required model weights—specifically the Qwen coder model and the Nomic text embedding model—are cached locally. If any models are missing, the background `ModelPullWorker` thread starts downloading them. The thread monitors the download progress using a generator that yields updates from the Ollama library. Running this resource-heavy process in a background thread keeps the main user interface responsive during the download.

The `ModelPullWorker` is designed to handle one model at a time. When the chatbot detects multiple missing models, it queues them and processes them sequentially. This sequential approach prevents bandwidth contention and ensures that each model is fully downloaded and verified before the next one begins.

4.2.2 Asynchronous Download Progress Signals

The `ModelPullWorker` thread communicates with the main user interface using Qt's signal-and-slot mechanism. As the download progresses, the worker thread calculates the download percentage by comparing the completed bytes with the total expected file size. It then emits this percentage as a progress signal, which is captured by the main GUI thread to update the status bar in real-time. Figure 4.3 shows the model pulling sequence.

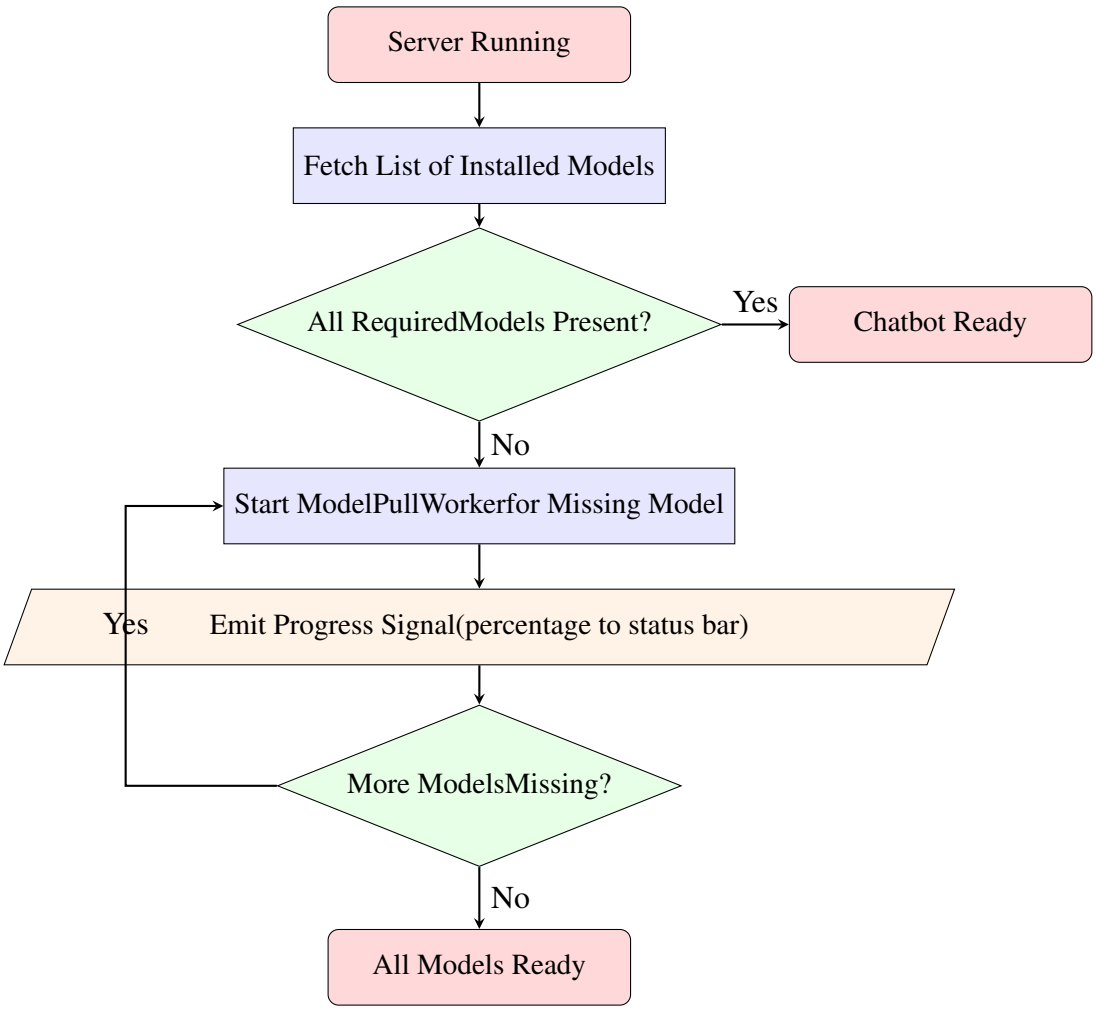

Figure 4.3: Sequential model pulling flow with progress signal emission.

Figure 4.4: The chatbot status bar showing real-time progress during model setup.

4.3 Resolution of Signal Naming Conflicts and Thread Safety

To ensure thread safety and prevent crashes during startup, we resolved several concurrency issues. Custom thread signals were renamed from "finished" to "result_ready" to avoid naming conflicts with PyQt's built-in `QThread.finished` event. This conflict caused the application to crash when a background thread completed its task, because PyQt interpreted the custom signal as the thread's termination signal and attempted to clean up resources prematurely.

Additionally, the chatbot input controls are disabled until the startup check returns a success flag, preventing users from sending prompts before the models are fully loaded. This mutex-like control flow ensures that the application transitions through its startup states in a deterministic order: server check, model verification, model download (if needed), and finally, chatbot activation.

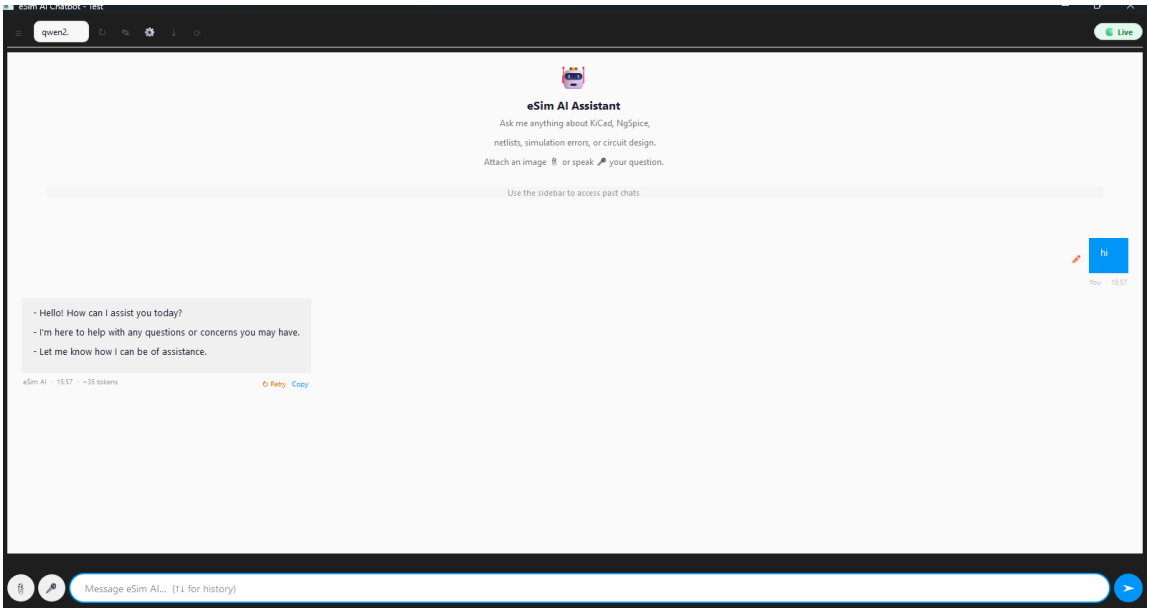

Figure 4.5: The fully initialized eSim AI Chatbot interface responding to a user's prompt.

Chapter 5

Robustness, Security, and Edge-Case Guardrails

5.1 Retrieval-Augmented Generation (RAG) Security

5.1.1 Lazy Paragraph Streaming Reader

To prevent resource exhaustion during the ingestion of large PDF manuals, the database pipeline uses a lazy paragraph generator. The legacy ingestion script read the entire document into system memory using a standard `read()` call, which led to high memory usage. We replaced this with a generator function that opens the file and reads it line-by-line. The generator buffers lines until it encounters a double newline, signaling a paragraph boundary, and then yields the completed block. Figure 5.1 illustrates this streaming mechanism.

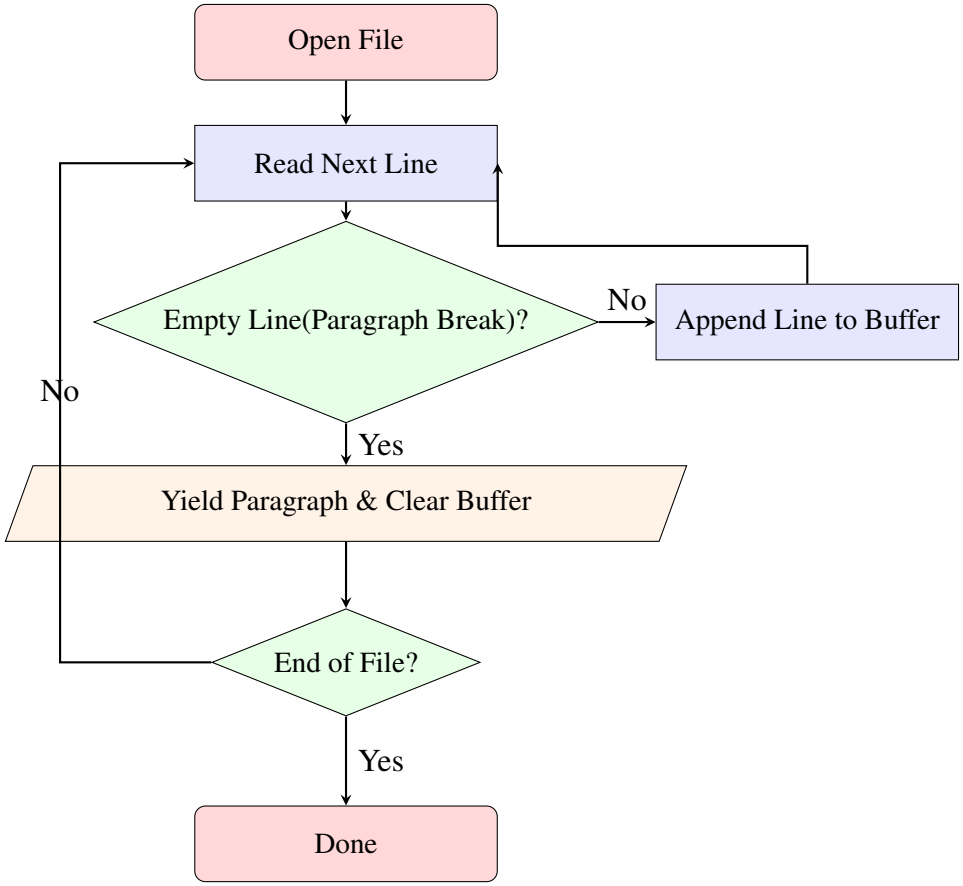

Figure 5.1: Lazy paragraph streaming reader flow, processing files line-by-line instead of loading entirely into memory.

This reduces memory usage from a factor dependent on the total file size to a constant factor based on the size of the largest single paragraph:

$$\text{Legacy Memory Complexity} = \mathcal{O}(N) \quad (\text{where } N \text{ is the file size in characters})$$

$$\text{New Memory Complexity} = \mathcal{O}(M) \quad (\text{where } M \text{ is the size of the largest paragraph})$$

This improvement is critical for production deployment, where eSim manuals can exceed several megabytes of raw text. Without the lazy reader, ingesting such files on systems with limited RAM (common in educational laboratories) would cause the application to crash or become unresponsive.

5.1.2 Atomic In-Memory Collection Rebuilds

The ingestion pipeline was also updated to prevent database corruption. Previously, the database collection was dropped *before* calculating the new embeddings. If the embedding process failed midway (due to a network timeout, for example), the database was left empty, resulting in complete data loss. Figure 5.2 contrasts the old and new approaches.

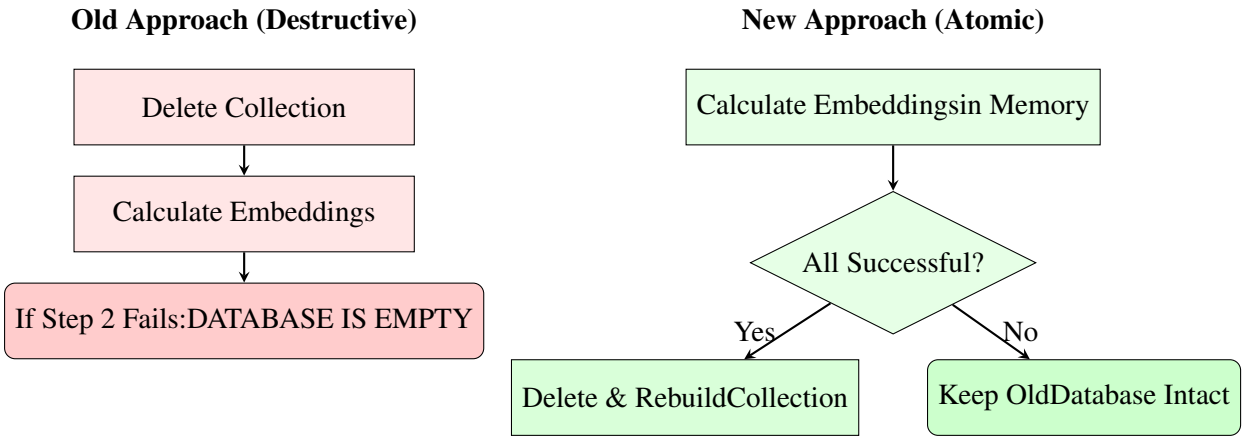

Figure 5.2: Comparison of old destructive ingestion vs. new atomic rebuild approach.

To resolve this, the ingestion script compiles all paragraph chunks and calculates their embeddings in-memory first. The existing database collection is deleted and recreated only if all embeddings are successfully calculated. If any part of the process fails, the database remains untouched, ensuring system reliability. This atomic approach guarantees that the knowledge base always contains valid, searchable data, even in the face of transient failures in the embedding service.

5.2 Speech-to-Text (STT) Robustness

5.2.1 Audio Queue Size Bounding

The offline voice recognition system uses a background thread to process audio frames. In earlier versions, the audio recorder pushed raw frames into an unbounded queue. If a user left the microphone active indefinitely, the queue would expand and consume all available system RAM, eventually causing the application to crash. To prevent this, the audio recording thread uses a bounded queue with a maximum limit of one thousand frames. If the queue is full, new audio frames are silently discarded to keep memory usage stable. Figure 5.3 shows this bounded queue mechanism.

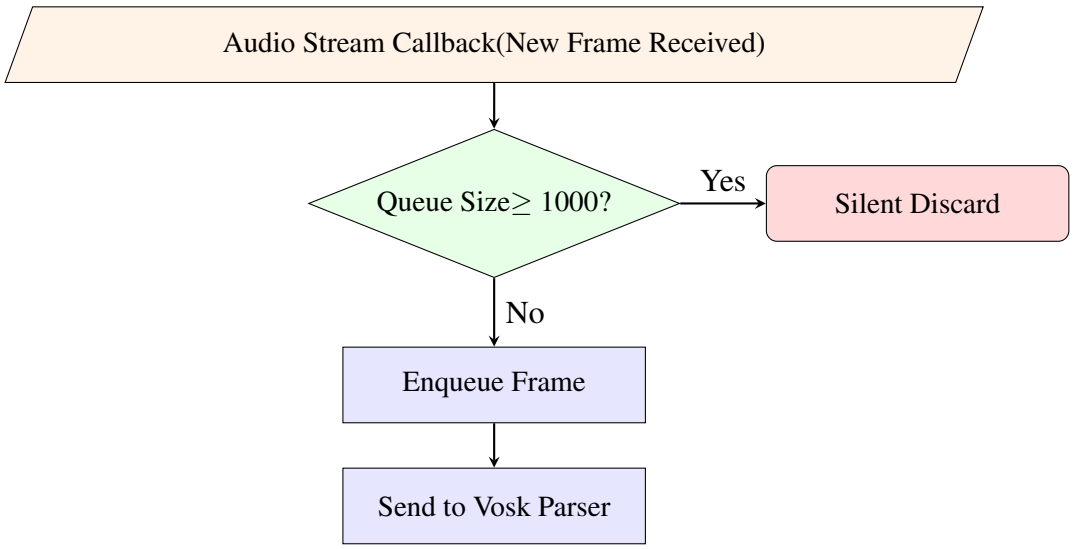

Figure 5.3: Bounded audio queue mechanism preventing memory exhaustion during voice recording.

5.2.2 Microphone Hardware Exception Handling

To prevent crashes on systems without audio hardware (such as headless virtual machines or servers), we wrapped the microphone initialization in a try-except block. The application attempts to open the recording stream; if it fails due to a missing or denied microphone, the application catches the error and disables the voice recording buttons in the GUI. This allows the user to continue using the text chat interface normally rather than crashing the entire application. This graceful degradation is essential for deployment in diverse environments, from fully-equipped workstations to minimal server installations.

5.2.3 Vosk JSON Output Decoding Guard

The offline Speech-to-Text engine parses microphone signals and returns predictions as JSON-formatted strings. If the audio signal contains line noise, the generated string can be corrupted,

causing standard JSON decoders to crash. To prevent this, the parsing routine uses a validation check. If the string is malformed, the decoder catches the exception, discards the corrupted frame, and waits for the next clean audio block. This defensive parsing strategy ensures that a single corrupted audio frame does not crash the entire recording session.

5.3 Vision Decompression Bomb Guard

5.3.1 Max Image Pixels Validation Checks

To protect the chatbot from decompression bombs (large, highly-compressed images that consume gigabytes of RAM when decompressed), the image upload interface checks the image dimensions before processing it. When a user uploads an image, the application reads its metadata to calculate the total pixel count (width multiplied by height). If the total exceeds ten million pixels (10 Megapixels), the application raises a `ValueError` and blocks further processing, protecting the system from memory exhaustion:

$$\text{Pixels} = \text{Width} \times \text{Height} \leq 10{,}000{,}000$$

This check is performed before the image is decoded into a full bitmap in memory, which is the point at which a decompression bomb would cause damage. By reading only the image header metadata, the check itself uses negligible memory.

5.3.2 Surfacing Design Errors to GUI

If an image fails the pixel check, the chatbot catches the exception and displays a clear error message in the chat panel. Surfacing the error in the GUI helps the user understand the issue while ensuring the backend stays online. Figure 5.4 shows the complete image validation flow.

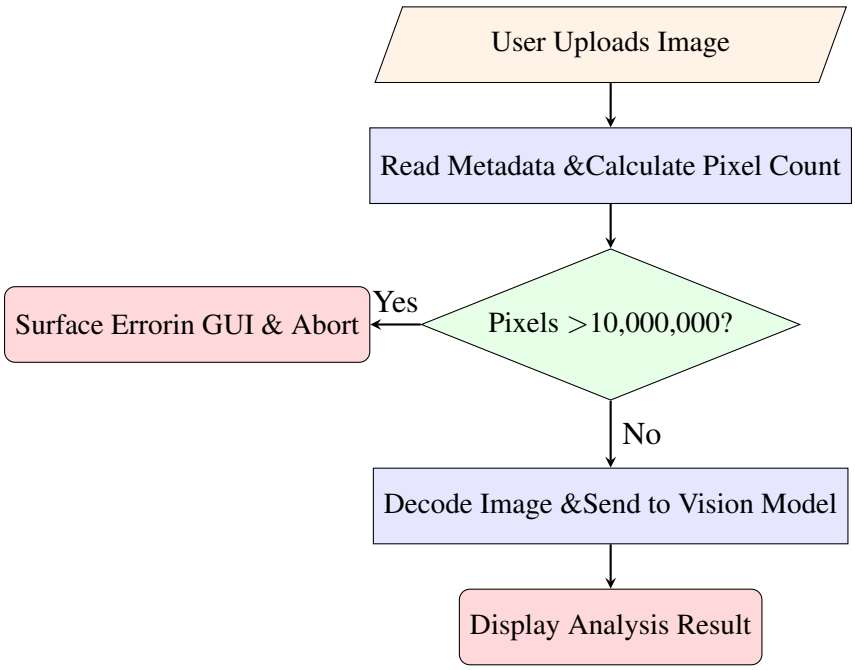

Figure 5.4: Image validation flow with decompression bomb guard.

Chapter 6

Automated Verification & Unit Testing Suite

6.1 Pytest Integration & Testing Methodology

To ensure system stability, we built an automated verification suite inside the chatbot tests directory. FOSSEE's software development guidelines require all modules to include isolated, offline-runnable unit tests. The testing suite is built on Python's built-in `unittest` library and the `pytest` framework. Because the chatbot relies on local AI models and hardware accessories, running the tests directly would require active models, databases, and microphones. To allow the tests to run offline during compilation or build pipelines, we used mocks to isolate the code from these hardware and API dependencies. Figure 6.1 shows the test execution pipeline.

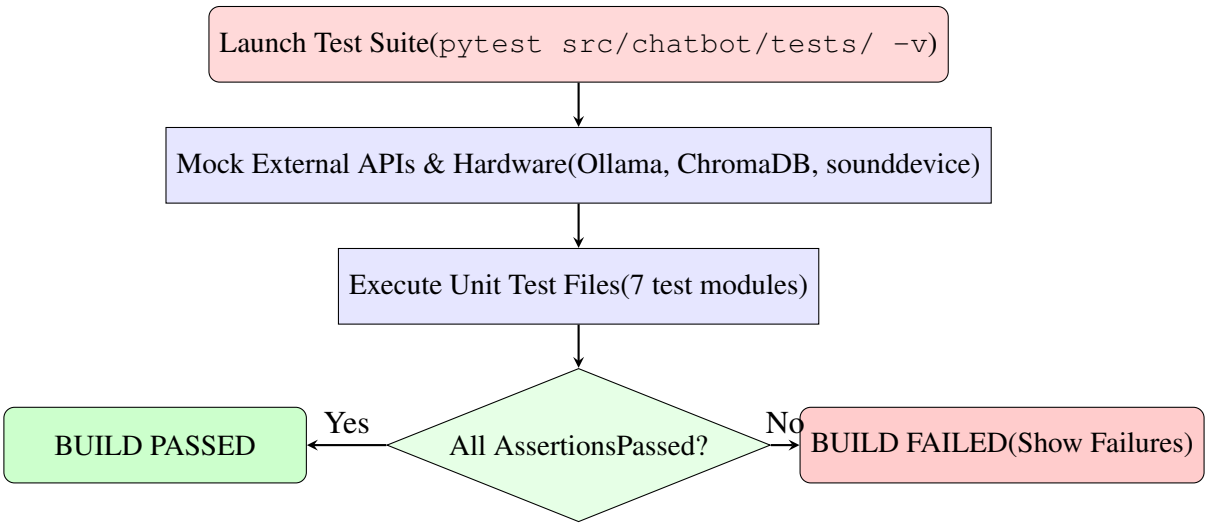

Figure 6.1: Test suite execution pipeline showing mocking, execution, and assertion verification.

6.2 Unit Test Implementation Breakdown

Table 6.1 summarizes the test files, the number of test cases in each, and the categories of vulnerabilities they verify.

Table 6.1: Test Suite Coverage Summary

Test File	Test Cases	Vulnerabilities Verified
test_knowledge_base.py	5	Non-destructive ingestion, lazy reading, relevance thresholding
test_stt_handler.py	4	Queue overflow, mic exception, JSON parsing
tests_image_handler.py	3	Decompression bomb, pixel validation, GUI error surfacing
test_chatbot_core.py	8	Intent routing, topic-switch detection, query classification
test_chatbot_thread.py	5	Signal naming, worker stop flags, config loading
test_error_solutions.py	3	Error pattern matching, fallback responses
test_ollama_runner.py	4	Embedding generation, API timeout handling

6.2.1 Knowledge Base Verification (test_knowledge_base.py)

The knowledge base test suite verifies the RAG ingestion pipeline. It contains test cases that mock the database client and check if the database is protected from partial failures. One test case simulates a failure during embedding generation; it asserts that the collection-dropping method is never called, verifying that the database is not corrupted by failed updates. Another test case mocks the file reading interface; it asserts that the system reads documents line-by-line rather than using `read()` calls, verifying the memory-efficient lazy reader. Lastly, the suite mocks the vector distance output to verify that chunks with relevance values below the threshold are filtered out, ensuring that the RAG pipeline delivers only pertinent context to the model.

6.2.2 Speech-to-Text Verification (test_stt_handler.py)

The Speech-to-Text test suite verifies the offline recording and parsing processes. It contains a test case that sends a stream of audio frames to the recorder queue; it asserts that once the queue reaches one thousand frames, all subsequent frames are discarded, verifying the queue boundary limit. Another test case mocks a missing audio device to verify that the application disables STT controls gracefully instead of throwing unhandled exceptions. Finally, a test case feeds malformed strings to the Vosk decoder to verify that JSON parsing errors are caught and handled without crashing the recording thread.

6.2.3 Image Handler Verification (tests_image_handler.py)

The image handler test suite verifies the image preprocessing and validation checks. It contains test cases that mock the Pillow image object. One test case feeds a mock image exceeding ten million pixels and asserts that the parsing method raises a `ValueError`. Another test case verifies that this exception is correctly passed to the GUI and displayed as a design error, verifying that the application is protected from decompression bombs.

6.3 Mocking Strategies for Offline Verification

The unit testing suite uses `unittest`'s patch decoration and `MagicMock` classes to isolate the application from external dependencies. We mock the ChromaDB client to simulate database collection actions, which allows us to test ingestion logic without writing to the disk. Similarly, we mock the local Ollama API to return predefined text and embedding vectors. We also mock the `sounddevice` library's recording stream to simulate microphone inputs. These mocking strategies allow the entire test suite to execute quickly and reliably in any environment, making it suitable for integration into automated build pipelines.

The mocking approach follows the principle of testing behaviour, not implementation. Each mock replaces the external dependency at the boundary layer, allowing the test to verify that the application logic responds correctly to both successful and failed interactions with the dependency. This makes the tests resilient to changes in the external APIs while still verifying the correctness of the application's error handling and data processing logic.

Chapter 7

Deployment, Performance & Future Scope

7.1 Ubuntu VM Deployment & Setup Script

The eSim AI Copilot is designed to run in a virtualized Ubuntu environment, making it easy to deploy on virtual machines or Windows Subsystem for Linux (WSL). We created an automated setup shell script to configure system dependencies, Python virtual environments, and local model backends. The script installs system libraries like ngspice, KiCad, PortAudio, and X11 graphics libraries.

To run eSim over remote SSH connections, the setup script configures the Qt platform backend to use the X Window System. It sets the platform environment variable to use the xcb backend, which directs PyQt to forward GUI windows over the SSH X11 tunnel to the host client. This allows users to access the desktop interface remotely while running the simulation engine on a server.

The deployment script follows a seven-step sequential flow. First, it installs system-level packages required for the graphical interface and audio processing. Second, it creates an isolated Python virtual environment to prevent dependency conflicts with the host system. Third, it installs all Python dependencies from the requirements file, including special handling for the `hdlparse` package which requires a downgraded version of `setuptools`. Fourth, it installs PaddlePaddle for OCR processing. Fifth, it downloads and installs the Ollama binary. Sixth, it pulls the required language and embedding models. Seventh, it downloads and configures the Vosk offline speech model.

7.2 CPU vs. GPU Latency Comparison

We tested the chatbot's performance on both CPU-only and GPU-enabled systems to evaluate response times. Local LLM generation speeds are measured in tokens per second, where a token is roughly equivalent to a word.

On CPU-only environments (such as standard virtual machines with 8GB of RAM), the chatbot experiences higher latency. The initial model loading sequence takes up to eight seconds, and response generation speeds average around five tokens per second. On GPU-enabled systems (with dedicated CUDA-compatible graphics cards), the performance increases significantly. Model loading is nearly instant, and response generation speeds reach forty tokens per

second. This comparison shows that while the chatbot is fully usable on standard CPU systems, a GPU-enabled hardware setup provides a much smoother user experience.

Table 7.1 summarizes the performance metrics across different hardware configurations.

Table 7.1: Performance Comparison: CPU vs. GPU Inference

Metric	**CPU-Only (8GB)**	**GPU (CUDA)**	**Improvement**
Model Load Time	8 seconds	$<$1 second	8$\times$
Generation Speed	5 tokens/sec	40 tokens/sec	8$\times$
Idle RAM Usage	$\sim$6 GB	$\sim$2 GB (+ VRAM)	3$\times$
First Response Latency	$\sim$10 seconds	$\sim$2 seconds	5$\times$

7.3 Future Recommendations for the AI Chatbot

7.3.1 Conversational Netlist Debugging and Direct File Editing

A key recommendation for the next phase of the AI Chatbot is the implementation of conversational netlist editing. Currently, when the chatbot analyzes a netlist and suggests a fix, the user must manually open the SPICE editor to apply it. In the future, the chatbot's interface could include a direct editing channel. If the user approves a fix suggested in the chat, the chatbot would modify the SPICE file directly in the background. This would allow the user to troubleshoot and fix simulation errors entirely through natural language commands in the chat window.

7.3.2 Federated Knowledge Ingestion and Retraining

Another promising area for the chatbot is the integration of a federated knowledge ingestion system. If the chatbot fails to resolve a simulation error and the user fixes it manually, the assistant could prompt the user to explain the fix in the chat. The chatbot would save this explanation to its local vector database, helping it learn from user feedback. Additionally, these local corrections could be anonymously shared with a central FOSSEE server. The server would group common fixes and periodically update the global knowledge base, allowing all eSim users to benefit from shared solutions.

7.3.3 Fine-Tuning Local Models for eSim Workflows

Lastly, future work could focus on fine-tuning a small language model specifically for eSim and ngspice workflows. While the current model performs well on standard coding tasks, it can struggle with complex analog circuit topologies and specialized SPICE model parameters. Fine-tuning a lightweight model (such as a 3B parameter model) on a dedicated dataset of

SPICE netlists, FOSSEE schematics, and electronics textbooks would improve the chatbot's accuracy while keeping memory usage low, ensuring a faster and more reliable offline assistant.

Bibliography

[1] Riverbank Computing Limited (2024). *PyQt6 Class Reference and API Documentation.* Riverbank Computing. Reference material for PyQt6 signals, slots, and GUI layouts.

[2] Vogt, H. (2024). *Ngspice User Manual Version 42.* The ngspice development team. Technical documentation for analog circuit simulation and SPICE netlist syntax.

[3] KiCad Developers Association (2024). *KiCad Schematic Editor Manual.* The KiCad project documentation. Reference for electrical rules checks (ERC) and schematic layout definitions.

[4] Ollama Team (2024). *Ollama API Reference and Model Management Manual.* Ollama Open Source Project. API documentation for local LLM inference and weight distribution.

[5] ChromaDB Team (2024). *Chroma: The AI-native open-source embedding database.* ChromaDB project documentation. API references for vector space management and similarity search metrics.

[6] Vosk Speech Recognition (2024). *Vosk API and Acoustic Model Documentation.* Alpha Cephei. Offline audio processing and acoustic model reference manuals.

[7] Pillow Project (2024). *Pillow (PIL Fork) Image Library Documentation.* Python Software Foundation. Image validation, pixel calculation, and preprocessing reference.

[8] PaddlePaddle Developers (2023). *PaddleOCR: Practical Ultra-Lightweight OCR System.* Baidu Open Source. OCR model deployment and text layout extraction guides.

[9] Pytest Development Team (2024). *Pytest: help writing better programs.* Python Software Foundation. Unit testing practices, assertions, and mock execution guidelines.