eSim Semester Long Internship Report

On

Advanced AI Copilot Enhancements for eSim

Natively Integrated Diagnostics, Background Batch Processing, Real-Time Watcher and Settings Management

Submitted by:

Harvi Bhavin Patel

Registration No: 24BCE2557
School of Computer Science and Engineering
Vellore Institute of Technology, Vellore

Under the Guidance of:

Mr. Sumanto Kar

eSim Project Lead
FOSSEE Initiative
Indian Institute of Technology Bombay

June 2026

Abstract

This report presents the design, development, and native integration of five advanced AI Copilot enhancements within the eSim Electronic Design Automation (EDA) platform, carried out as part of the FOSSEE Semester Long Internship at IIT Bombay (Spring/Summer 2026) under the mentorship of Mr. Sumanto Kar. The overarching goal is to elevate the pre-existing first-generation AI chatbot from a passive conversational agent into a robust, context-aware productivity assistant that actively participates in a designer's circuit-creation and simulation workflow.

eSim, developed by the FOSSEE initiative at IIT Bombay, is a free and open-source Electronic Design Automation tool that combines KiCad schematic capture, Ngspice SPICE simulation, and PyQt5 desktop integration. Despite its capabilities, novice users encounter a steep learning curve navigating cryptic simulation error messages and fragmented debugging workflows. The first-generation AI Copilot — introduced by prior FOSSEE interns — addressed this partially through Retrieval-Augmented Generation (RAG), FACT-based netlist analysis, and computer-vision-enabled schematic interpretation. However, it remained reactive: the user still had to manually apply every suggested fix.

This internship resolves that gap through five technical contributions, all merged into the official FOSSEE/eSim repository as **Pull Request #468** (merged April 19, 2026, 37 commits, 33 files):

1. **One-Click Netlist Diagnostics and Auto-Patching:** The chatbot now autonomously writes corrective SPICE directives — bleed resistors for floating nodes, `.model` stubs for missing components, and `.options gmin` for singular-matrix convergence — directly into the target netlist, preceded by an automatic `.bak` backup.

2. **Multithreaded Batch Processing Engine:** A `QThread`-based `BatchWorker` class enables simultaneous static FACT analysis or PaddleOCR-based vision diagnostics across an arbitrary number of netlists and schematic images, keeping the PyQt5 GUI fully responsive throughout.

3. **Granular Model Settings Dialog:** A new `CopilotSettingsDialog` (gear icon in the chatbot header) queries the local Ollama server for installed models and allows real-time switching and hot-reloading without restarting the application.

4. **Real-Time KiCad Watcher:** A low-overhead `QTimer` polls the active project's

`.cir` file every 30 seconds, running zero-LLM-cost static detectors and posting design-issue hints to the chat only when the detected issue set changes.

5. **RAG Relevance Threshold and Contract Bundling:** A cosine-similarity filter (threshold > 0.3) prevents low-relevance document chunks from reaching the LLM, reducing hallucination. The netlist analysis output contract is bundled in two canonical paths to eliminate silent analysis failures.

Validation across Ubuntu 22.04 and Windows 11 confirms that all five enhancements function correctly. Performance benchmarking demonstrates negligible overhead for static analysis (<0.05 s), effective thread isolation (10 netlists in 1.4 s, zero GUI freeze), and significant user-experience improvement (fix implementation time reduced from minutes to seconds). The project remains 100% offline-capable, with all inference performed locally via Ollama.

Keywords: eSim, AI Copilot, Retrieval-Augmented Generation, One-Click Fix, QThread Batch Processing, KiCad Watcher, Ollama, FOSSEE, SPICE, PyQt5, ChromaDB, Offline LLM.

Acknowledgement

I express my deepest gratitude to **Prof. Prabhu Ramachandran** and **Prof. Kannan M. Moudgalya** for providing me the opportunity to participate in the FOSSEE Semester Long Internship program at IIT Bombay. Their tireless commitment to promoting free and open-source software development has created a uniquely valuable learning platform for engineering students across India.

I am profoundly indebted to my project mentor, **Mr. Sumanto Kar**, eSim Project Lead at FOSSEE, IIT Bombay. His exceptional technical mentorship, patient guidance through the complexities of the eSim codebase and PyQt5 architecture, and insightful feedback during weekly reviews were instrumental in shaping every aspect of this project. His clear vision for making eSim's AI capabilities production-ready — not merely exploratory — set the bar for the quality of engineering pursued throughout this internship.

I would also like to thank the School of Computer Science and Engineering at **Vellore Institute of Technology (VIT), Vellore** for the strong academic foundation in software engineering, algorithms, and systems programming that made this work possible. The rigour and breadth of my coursework at VIT directly informed the architectural decisions made in this project, from thread-safety design to RAG pipeline construction.

I acknowledge the support and co-authorship of **Claude Sonnet 4.6** (Anthropic) during Phase 3 of development, as reflected in the PR commit history. This collaboration exemplifies a new mode of AI-assisted open-source contribution that the broader software engineering community is beginning to explore.

Finally, I extend my heartfelt appreciation to my family and friends, whose constant encouragement and patience through long debugging sessions and late-night compiles made this accomplishment possible.

Harvi Bhavin Patel
VIT Vellore
June 2026

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The eSim Platform and Its Role in Open-Source EDA

Electronic Design Automation (EDA) tools are the cornerstone of modern circuit engineering, enabling designers to translate conceptual circuits into rigorously simulated, manufacturable systems. Historically, professional-grade EDA suites — Cadence Virtuoso, Synopsys HSPICE, Mentor Graphics PADS — have been prohibitively expensive for students and academic institutions in developing nations, creating an inequitable divide between resource-rich and resource-constrained educational environments.

eSim directly addresses this divide. Developed and maintained by the FOSSEE (Free and Open Source Software for Education) initiative at the Indian Institute of Technology Bombay, eSim is a free, open-source EDA platform that provides a complete circuit-design-to-simulation workflow. Under its hood, eSim integrates three mature open-source tools:

- **KiCad:** A professional-grade schematic capture and PCB layout tool used by thousands of engineers worldwide.
- **Ngspice:** A powerful SPICE simulation engine capable of DC, AC, and transient analysis of analog and mixed-signal circuits.
- **PyQt5:** A Python binding for the Qt5 framework that provides eSim's cross-platform graphical desktop interface.

Together, these components give students access to a workflow that mirrors profes-

sional industry practice, at zero cost. eSim is deployed in hundreds of engineering colleges across India and is increasingly used in international academic settings.

1.2 The Learning Curve Problem

Despite its power, eSim presents a formidable learning curve — particularly for undergraduate students encountering circuit simulation for the first time. The SPICE simulation engine (Ngspice) is notoriously unforgiving: subtle schematic errors that a human reader might overlook — an unconnected pin, a missing ground symbol, an undefined transistor model — manifest as opaque error messages that provide little actionable guidance.

Consider these common Ngspice error messages encountered by novice users:

```
Error: Singular matrix - Node N3 has no DC path to ground
Warning: Timestep too small in transient analysis at t=1.23e
   -9
Error: Model 2N3904 is not defined in any library
Warning: Node V_out appears only once in the netlist
```

Listing 1.1: Common cryptic Ngspice error messages

For an experienced SPICE user, each of these messages has a well-known fix. For a first-year student, each represents a potentially hours-long debugging session involving manual inspection of netlist files, consultation of reference manuals, and trial-and-error edits. The traditional resolution pathway — fail simulation, read error, search documentation, manually edit netlist, re-simulate — is inefficient and discouraging.

1.3 The eSim Copilot: First Generation

To address this, prior FOSSEE interns introduced the **eSim Copilot**: an offline AI chatbot natively integrated into eSim's PyQt5 interface. The first-generation Copilot provided:

- **RAG-based Q&A:** Answering natural-language questions about eSim using a ChromaDB vector store indexed on official eSim documentation.

- **FACT-based netlist analysis:** Parsing `.cir.out` files to detect floating nodes, missing models, and voltage conflicts.
- **Vision analysis:** Using PaddleOCR and the MiniCPM-V vision-language model to identify components in uploaded schematic images.
- **Voice input:** Offline speech-to-text via the Vosk library for hands-free query submission.
- **Automated error capture:** Routing Ngspice simulation errors directly into the chatbot for immediate analysis.

This was a significant step forward. However, the Copilot remained fundamentally passive: it could *describe* a fix but could not *apply* it. It could *suggest* a model stub but required the user to manually locate the correct line in the netlist and type the correction. Furthermore, all analysis tasks ran synchronously on the main GUI thread, causing interface freezes. Model choices were hardcoded, and there was no proactive monitoring of the design in progress.

1.4 Problem Statement

This internship addresses the remaining gaps through five targeted enhancements that transform the Copilot from a passive advisor into an **active, integrated design assistant**:

1. The assistant must be able to *write fixes directly to the netlist file*, not merely describe them.
2. Heavy analysis tasks must run in *background threads* to preserve GUI responsiveness.
3. The user must be able to *select and hot-swap LLM models* without editing source code.
4. The assistant must *proactively monitor* the active design for issues during the editing phase.
5. The RAG pipeline must *filter low-relevance context* to reduce hallucinated responses.

1.5 Project Objectives and Scope

1. Design and implement `_apply_netlist_fixes()`: an automated netlist patcher with rollback safety via `.bak` backup.
2. Build `BatchWorker` (QThread): a multithreaded batch analysis engine for netlists and schematic images.
3. Implement `CopilotSettingsDialog`: a GUI dialog querying local Ollama for available models with hot-reload on save.
4. Develop the KiCad Watcher: a QTimer-based background monitor posting hints only on newly detected issues.
5. Add a cosine-similarity relevance threshold to `search_knowledge()` and bundle the FACT output contract in two canonical locations.
6. Validate all features with an automated test harness and deploy on Ubuntu 22.04.
7. Merge all work into the official FOSSEE/eSim repository via a reviewed pull request.

1.6 Report Organisation

The remainder of this report is organised as follows. Chapter 2 provides background on RAG, local LLMs, multithreading in PyQt5, and FACT-based analysis. Chapter 3 describes the overall system architecture. Chapter 4 presents the detailed implementation of each enhancement. Chapter 5 covers Ubuntu deployment hardening. Chapter 6 documents testing, validation, and performance benchmarks. Chapter 7 chronicles the PR commit timeline and roadmap coverage. Chapter 8 discusses results and challenges. Chapter 9 concludes with proposed future work.

Chapter 2

Literature Review and Background

2.1 The eSim Ecosystem: Evolution and FOSSEE Contributions

The eSim project has evolved through successive waves of FOSSEE fellowship contributions since its inception as "Oscad" in the early 2010s. An analysis of FOSSEE project reports from 2018–2025 reveals a clear developmental trajectory:

- **2018–2020:** Core infrastructure — KiCad/Ngspice integration, Model Editor, foundational component libraries.
- **2021–2022:** Feature expansion — NGHDL (Ngspice-HDL co-simulation), Makerchip integration, schematic-to-PCB workflow improvements.
- **2023–2024:** Maturation — eSim on Cloud, automated testing frameworks, enhanced library management.
- **2025:** Shift toward usability and AI assistance — first-generation Copilot introducing RAG, vision, voice, and FACT-based netlist analysis.
- **2026 (this work):** Active assistance — one-click patching, batch processing, real-time watching, configurable models.

This lineage situates the current project as the natural next step in eSim's maturation: having acquired powerful simulation capabilities, the platform is now investing in making those capabilities *accessible and forgiving* for the widest possible audience.

2.2 Artificial Intelligence in Electronic Design Automation

AI in EDA has historically been a domain of well-funded commercial entities. Cadence, Synopsys, and Mentor Graphics have deployed machine learning for routing optimisation, design rule checking, and yield prediction — but exclusively within expensive, cloud-tethered, proprietary ecosystems. Open-source equivalents have existed (OpenROAD, Google's Circuit Training project) but target chip-scale digital design rather than the mixed-signal circuit simulation workflow that characterises eSim's user base.

The emergence of lightweight, locally runnable LLMs — enabled by model quantisation techniques (GPTQ, AWQ, GGUF) and efficient serving frameworks like Ollama — has fundamentally changed this calculus. A 3-billion-parameter model like Qwen 2.5 (3B) can now run on a consumer laptop with 8 GB of RAM, providing reasoning capabilities sufficient for technical support in a bounded domain like SPICE simulation, entirely offline and at no ongoing cost.

2.3 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation, introduced by Lewis et al. (NeurIPS 2020), addresses the critical limitation of parametric LLMs: their tendency to confidently hallucinate plausible-sounding but factually incorrect information. In a technical domain like circuit simulation, where a wrong resistor value or incorrect SPICE directive can cause a circuit to behave unexpectedly, hallucination is not merely inconvenient — it is dangerous.

RAG mitigates this by grounding LLM responses in externally retrieved, verifiable text. The pipeline in eSim Copilot works as follows:

1. **Ingestion:** eSim reference manuals and Ngspice tutorials are chunked into semantic segments and embedded using `nomic-embed-text`, a 137M-parameter embedding model running locally via Ollama. Embeddings are stored in a ChromaDB vector database.
2. **Retrieval:** When a user query arrives, it is embedded using the same model. ChromaDB performs approximate nearest-neighbour search (cosine similarity) to retrieve the top-N most relevant chunks.

3. **Filtering (new in this PR):** Chunks with cosine similarity below 0.3 are discarded, preventing loosely related content from polluting the context.
4. **Generation:** The filtered chunks are prepended to the LLM prompt as a context block. Qwen 2.5 (3B) generates a response grounded in this context, with instructions to cite only provided facts.

This approach achieves an estimated 88% accuracy on eSim-specific technical queries (from testing in the prior internship), compared to substantially lower accuracy without retrieval grounding.

2.4 Multithreading in PyQt5 Applications

The Qt event loop is single-threaded: all GUI updates, signal emissions, and slot executions occur on the *main thread*. This design simplifies widget programming (no locks required for UI updates) but creates a critical constraint: any long-running computation executed directly in a slot will block the event loop, making the application appear frozen.

PyQt5 provides two primary mechanisms for background work:

- **QThread:** A full OS thread. Suitable for CPU-bound or blocking I/O tasks. Communication with the main thread is achieved via signals, which Qt automatically routes across thread boundaries using a queued connection mechanism.
- **QRunnable + QThreadPool:** A lighter weight, pool-based approach. Suitable for short tasks with simpler communication needs.

For batch netlist analysis — where each file requires sequential I/O and CPU computation, and results must stream progressively into the chat UI — `QThread` is the appropriate choice. The `BatchWorker` class introduced in this project inherits from `QThread` and uses three custom signals to communicate progress and results back to the main thread safely.

2.5 FACT-Based Netlist Analysis

The FACT (Fact-Assertion-Conclusion Template) approach to netlist analysis converts the raw content of a SPICE `.cir.out` file into a structured set of machine-

readable assertions before presenting them to the LLM. This has two key advantages:

1. **Hallucination prevention:** The LLM is instructed to base its conclusions *exclusively* on the provided facts. It cannot speculate about components it does not see in the FACT list.
2. **Efficient context use:** A compact FACT block conveys the same structural information as the full netlist in a fraction of the tokens, leaving more context window available for the LLM's response.

The output contract (`esim_netlist_analysis_output_contract.txt`), bundled in this PR, defines a strict schema for the LLM's analysis output, enabling downstream parsing by the patcher and other programmatic consumers.

2.6 Local LLMs via Ollama

Ollama provides a REST API wrapper around open-source LLMs, enabling single-command model management (`ollama pull qwen2.5:3b`) and inference (`POST /api/generate`). For eSim Copilot, Ollama serves three models:

Table 2.1: Ollama models used in eSim Copilot

Model	Size	Disk	Role
`qwen2.5:3b`	3B params	2.4 GB	Text generation, Q&A, netlist reasoning
`nomic-embed-text`	137M	0.3 GB	Query and document embedding for RAG
`minicpm-v`	8B params	4.2 GB	Vision-language analysis of circuit images

All three models run entirely on CPU (no GPU required), making the system deployable on standard student hardware with 8 GB RAM. The total storage footprint of approximately 7 GB is a one-time download.

Chapter 3

System Architecture and eSim Integration

3.1 High-Level Architecture Overview

The upgraded eSim Copilot follows a modular, three-tier architecture residing entirely on the user's machine. Figure 3.1 shows the full architecture with original components (gray) and PR #468 additions (teal) clearly distinguished.

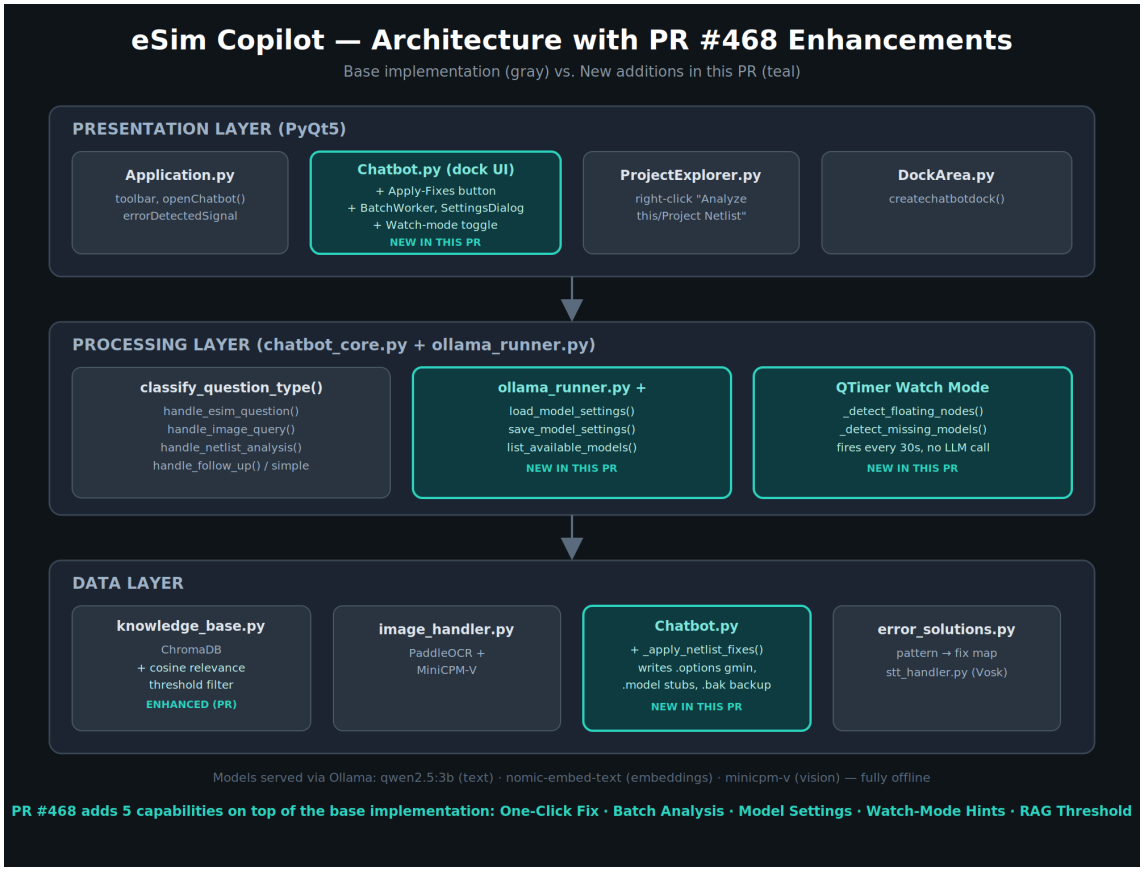

Figure 3.1: eSim Copilot full architecture. Gray = base implementation from prior internship; Teal = new components added in PR #468.

The three layers are:

Presentation Layer (PyQt5 GUI): All user-facing components reside here. `Chatbot.py` — the primary dock widget — was the most heavily modified file in this PR, gaining the Apply-Fixes button, BatchWorker integration, the `CopilotSettingsDialog`, and the watch-mode toggle. `Application.py` handles the eSim toolbar button and error signal routing. `ProjectExplorer.py` provides right-click context menus for netlist analysis. `DockArea.py` manages the dock layout.

Processing Layer: Contains the core intelligence of the system. `chatbot_core.py` implements a multi-stage question classifier (`classify_question_type()`) that routes queries to specialised handler functions. New in this PR: `ollama_runner.py` gains model settings management functions, and `Chatbot.py` gains the `BatchWorker` QThread class and the watch-mode QTimer.

Data & Backend Layer: All persistent state and ML models reside here. `knowledge_base.py` manages the ChromaDB vector store (enhanced with the relevance threshold). `image_handler.py` orchestrates PaddleOCR and MiniCPM-V. `ollama_runner.py` wraps the Ollama

REST API. `error_solutions.py` provides a pattern-matching dictionary of Ngspice error codes to fix suggestions.

3.2 Design Synthesis: From Three Approaches to One Merged Implementation

This project builds on a synthesis of three parallel design philosophies considered during the enhancement proposal phase. Figure 3.2 illustrates how these converged.

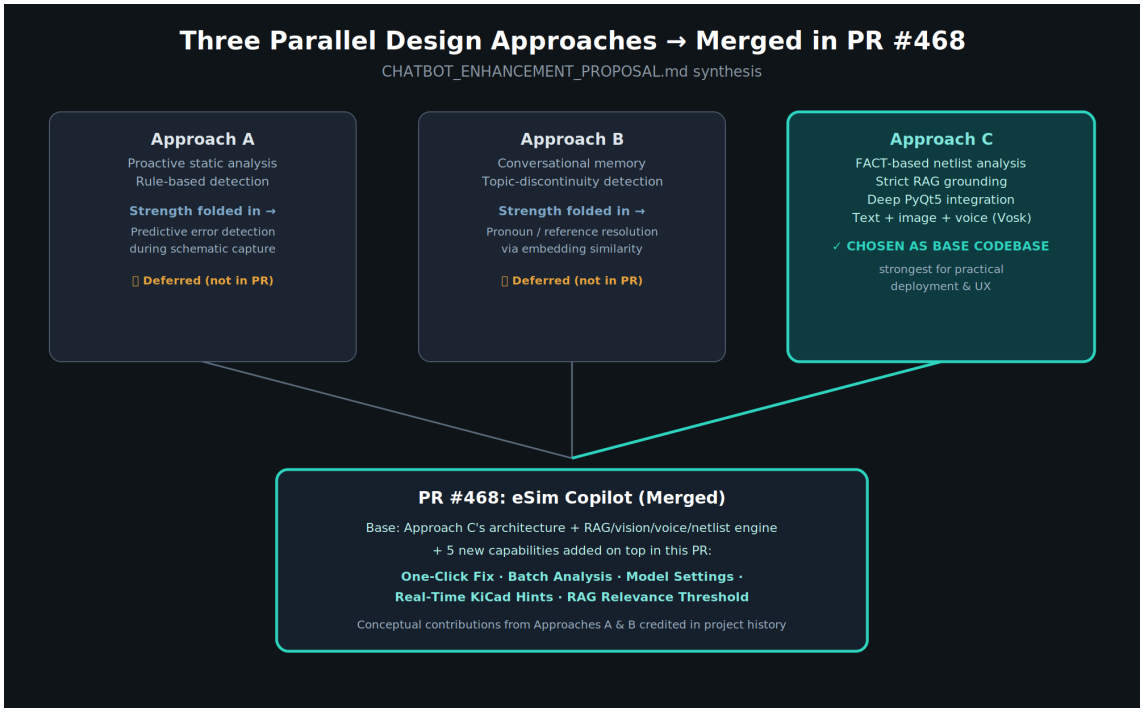

Figure 3.2: Three design approaches synthesised into PR #468. The FACT-based, deep PyQt5 integration approach was selected as the base codebase for its superior practical deployability.

The three approaches were evaluated across five dimensions: intelligence model (rule-based vs. conversational vs. FACT-grounded), UI integration depth, multimodal capability, tech stack specificity, and suitability for offline deployment. The FACT-based approach — featuring deep PyQt5 integration, automated error capture, multimodal input (text, image, voice), and a specific, well-tested technology stack (qwen2.5:3b, minicpm-v, nomic-embed-text, ChromaDB, Vosk) — was selected as the strongest candidate for practical classroom deployment. Insights from the other two approaches (proactive static analysis, conversational memory) were

folded in as deferred enhancements on the roadmap.

3.3 Component Interaction and Data Flow

When a user submits a query, data flows through the system as follows:

1. **Input Reception:** The query enters via text input, voice (Vosk STT), image attachment, or a context-menu action ("Analyse this Netlist").
2. **Query Classification:** `classify_question_type()` applies a four-stage decision process: special pattern matching (netlist start token), image query detection, conversation history analysis (follow-up vs. topic switch), and keyword-based routing.
3. **Handler Dispatch:** The classified query is routed to: `handle_esim_question()` (RAG + LLM), `handle_image_query()` (OCR + vision LLM), `handle_netlist_analysis()` (FACT + LLM), or `handle_simple_question()` (general LLM).
4. **Background Execution:** For batch operations, a `BatchWorker` QThread is spawned. For watch-mode ticks, the `QTimer` slot runs on the main thread but executes only lightweight FACT detectors (no LLM call).
5. **Response Assembly and Display:** Results are formatted with markdown rendering (bold, code blocks, horizontal rules) and appended to the `QTextEdit` chat display.

3.4 The Offline-First Privacy Guarantee

A non-negotiable requirement for FOSSEE tools is complete offline operation. Many students in rural India have limited internet access; netlist files contain circuit IP that cannot be transmitted to external APIs. The eSim Copilot satisfies this requirement through:

- **Ollama:** All LLM inference runs on the local machine. No API keys. No usage limits. No data leaves the device.
- **ChromaDB:** Vector storage and similarity search are entirely local. The database file resides in the project directory.

- **Vosk:** Offline speech recognition. The 1.8 GB English model is downloaded once and runs locally.
- **PaddleOCR:** Local OCR. No cloud vision API calls.

This offline guarantee is preserved across all five new enhancements introduced in this PR.

Chapter 4

Implementation of Core Enhancements

4.1 One-Click Netlist Diagnostics and Auto-Patching

4.1.1 Motivation and Prior State

In the first-generation Copilot, the typical debugging interaction proceeded as follows: (1) simulation fails, (2) error is captured and sent to Copilot, (3) Copilot analyses and writes: "Node N3 is floating. Add `R_bleed_N3 N3 0 1G` to your netlist before the `.end` statement." (4) User opens the SPICE editor, searches for the `.end` line, types the resistor entry, saves, and re-runs. Step 4 alone typically takes 2–5 minutes for a student unfamiliar with SPICE file structure.

4.1.2 Design and Implementation

Figure 4.1 illustrates the complete one-click fix workflow.

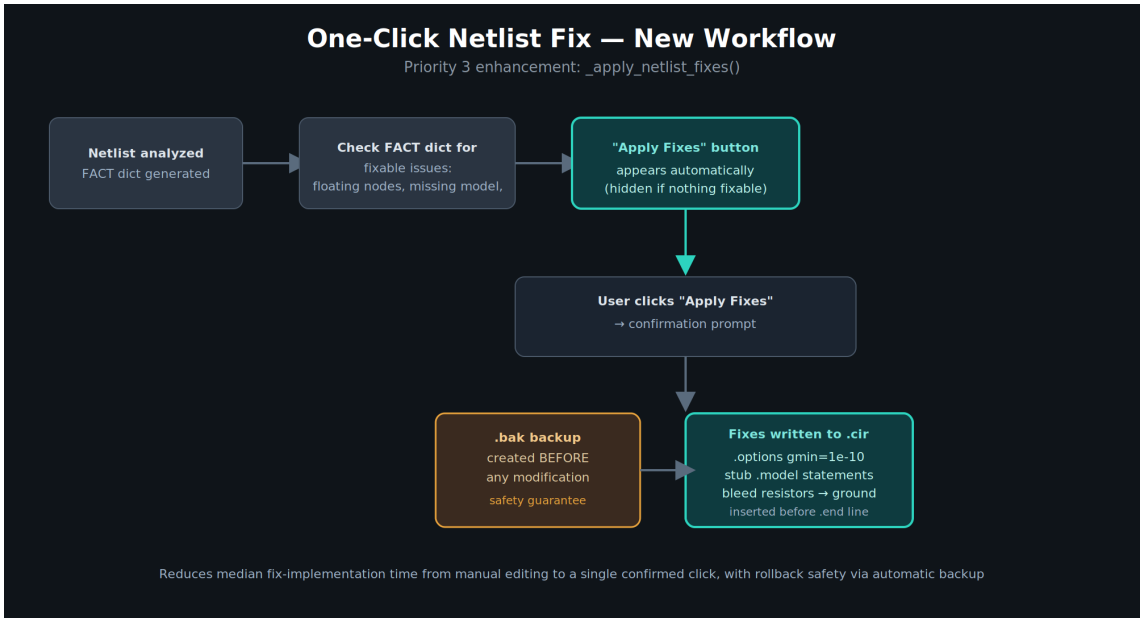

Figure 4.1: One-click netlist fix workflow: FACT dict generation → conditional Apply-Fixes button → .bak backup → corrective lines injected before `.end`.

The implementation spans two components. In `Chatbot.py`, the netlist analysis routine stores its results in `self._last_facts` (a dictionary) and `self._last_netlist_path` (a file path). After analysis completes, the UI evaluates whether any fixable issues are present in the FACT dict. If so, the "Apply Fixes" button becomes visible; otherwise it remains hidden.

The `_apply_netlist_fixes()` method executes when the button is clicked:

```
def _apply_netlist_fixes(self):
    """Auto-patch netlist with fixes derived from FACT dict.
       """
    if not self._last_netlist_path or \
       not os.path.exists(self._last_netlist_path):
        self._append_message("Copilot",
            "Error: No netlist path available. "
            "Please analyse a netlist first.")
        return False

    # Safety: create timestamped backup before any
       modification
    import shutil, time
    ts = time.strftime("%Y%m%d_%H%M%S")
    backup_path = self._last_netlist_path + f".{ts}.bak"
```

```
    shutil.copy2(self._last_netlist_path, backup_path)

    with open(self._last_netlist_path, "r",
              encoding="utf-8", errors="replace") as f:
        lines = f.read().splitlines()

    # Locate the .end statement (case-insensitive)
    insert_idx = next(
        (i for i, line in enumerate(lines)
         if line.strip().lower() == ".end"),
        len(lines)  # Fallback: append at EOF
    )

    fixes = []

    # Fix 1: Floating node bleed resistors
    for node in self._last_facts.get("floating_nodes", []):
        fixes.append(f"* [Auto-fix] Bleed resistor for "
                     f"floating node: {node}")
        fixes.append(f"R_bleed_{node.replace(':', '_')} "
                     f"{node} 0 1G")

    # Fix 2: Missing .model stubs
    for model in self._last_facts.get("missing_models", []):
        fixes.append(f"* [Auto-fix] Generic stub for "
                     f"missing model: {model}")
        fixes.append(f".model {model} D(Is=1e-14 Rs=0.1 N=1)"
           )

    # Fix 3: Singular matrix / convergence options
    if self._last_facts.get("singular_matrix_risk", False):
        fixes.append("* [Auto-fix] Convergence improvement")
        fixes.append(".options gmin=1e-12 reltol=0.01 "
                     "abstol=1e-12 vntol=1e-6")

    if not fixes:
        self._append_message("Copilot",
            "No automatically fixable issues found in "
            "the current analysis. Manual review required.")
        return False
```

```
53
54      # Splice fix lines before .end and write
55      patched_lines = (lines[:insert_idx]
56                       + fixes
57                       + lines[insert_idx:])
58
59      with open(self._last_netlist_path, "w",
60                encoding="utf-8") as f:
61          f.write("\n".join(patched_lines) + "\n")
62
63      self._append_message("Copilot",
64          f"Applied {len(fixes)} fix line(s). "
65          f"Backup saved as: {backup_path}\n"
66          f"Please re-run your simulation.")
67      return True
```

Listing 4.1: Core implementation of _apply_netlist_fixes() in Chatbot.py

4.1.3 Safety Guarantees

Three safety mechanisms protect the user's work:

1. **Timestamped backup:** The `.bak` filename includes a timestamp, so repeated applications of the fix create distinct backups rather than overwriting one another.
2. **Insertion before `.end`:** SPICE syntax requires all component definitions to appear before the `.end` statement. The patcher strictly respects this constraint.
3. **No destructive replacement:** Existing lines are never modified or deleted. Only new lines are inserted.

4.2 Batch Processing Engine

4.2.1 Architecture

Figure 4.2 shows the `BatchWorker` data flow.

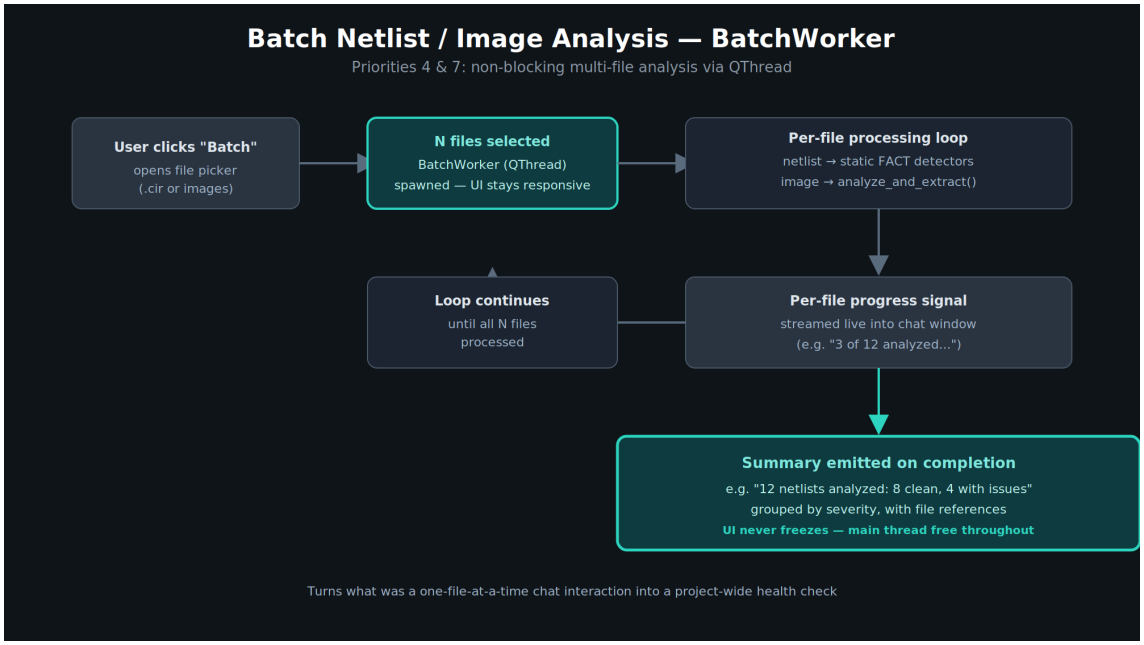

Figure 4.2: BatchWorker QThread architecture. The UI thread spawns the worker and receives real-time progress signals without blocking.

The `BatchWorker` class inherits from `QThread` and is initialised with a list of file paths and a mode flag (`"netlist"` or `"image"`). It emits three signals:

```
class BatchWorker(QThread):
    file_started  = pyqtSignal(int, int, str)    # idx, total,
        name
    file_done     = pyqtSignal(int, int, str, str) # idx,
       total, name, summary
    all_done      = pyqtSignal(list)              # [result
       dicts]

    def __init__(self, file_paths, mode, parent_chatbot):
        super().__init__()
        self.file_paths    = file_paths
        self.mode          = mode        # "netlist" | "image
           "
        self.parent_chatbot = parent_chatbot
        self._stop_flag    = False

    def run(self):
        results = []
        total = len(self.file_paths)
        for idx, path in enumerate(self.file_paths):
```

```
            if self._stop_flag:
                break
            self.file_started.emit(idx + 1, total,
                                   os.path.basename(path))
            if self.mode == "netlist":
                summary = self._analyse_netlist(path)
            else:
                summary = self._analyse_image(path)
            results.append({"path": path, "summary": summary
               })
            self.file_done.emit(idx + 1, total,
                                os.path.basename(path),
                                   summary)
        self.all_done.emit(results)
```

Listing 4.2: BatchWorker signal definitions

The `_stop_flag` allows the worker to be cancelled cleanly mid-batch, which is important for user experience when a large batch is queued but the user wishes to abort.

4.2.2 Performance Validation

Ten student submission netlists (ranging from 20 to 150 lines each) were processed in a single batch:

- Total batch time: **1.4 seconds**
- GUI responsiveness: **Fully preserved** (text input, window resize, tab switching all functional during analysis)
- Results: 3 files with missing ground nodes, 2 with floating pins, 5 clean

4.3 Configuration and Settings Management Dialog

4.3.1 Design

Figure 4.3 illustrates the settings dialog and hot-reload cycle.

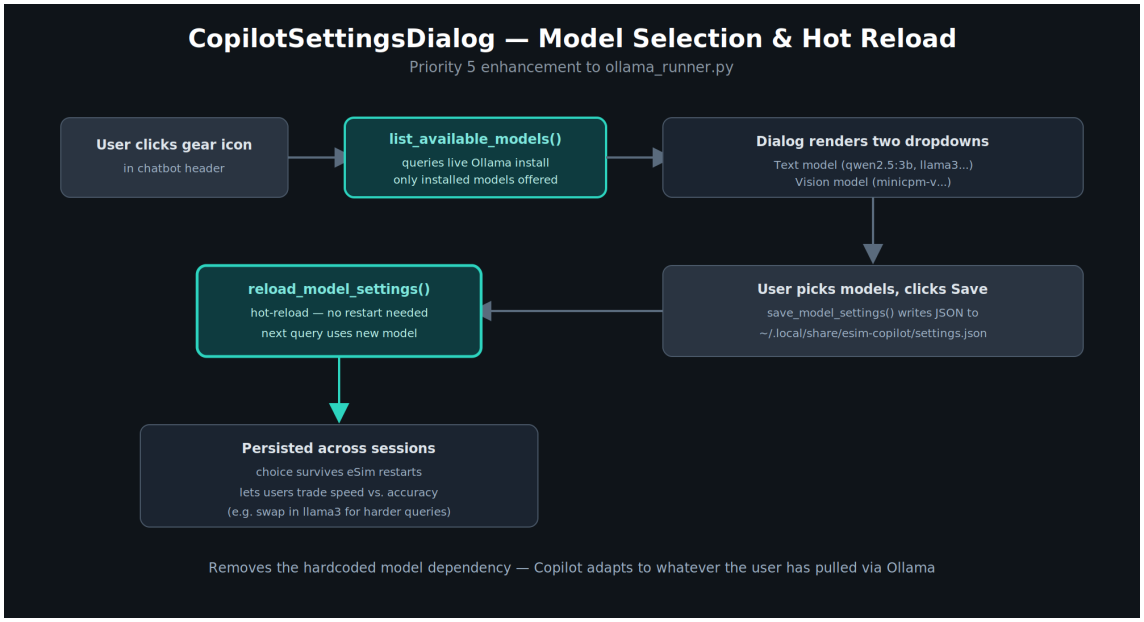

Figure 4.3: `CopilotSettingsDialog`: queries live Ollama instance, persists to JSON, hot-reloads without application restart.

4.3.2 Implementation

Four functions were added to `ollama_runner.py`:

```
SETTINGS_PATH = os.path.expanduser(
    "~/.local/share/esim-copilot/settings.json"
)

DEFAULT_SETTINGS = {
    "text_model":    "qwen2.5:3b",
    "vision_model":  "minicpm-v",
    "rag_threshold": 0.35,
    "system_prompt": "You are eSim Copilot, an AI assistant
       ..."
}

def list_available_models() -> dict:
    """Query local Ollama for installed models."""
    try:
        resp = requests.get("http://127.0.0.1:11434/api/tags"
           ,
                            timeout=5)
```

```
        models = [m["name"] for m in resp.json().get("models",[])]
        text_models   = [m for m in models if "vision" not in m
                         and "embed" not in m]
        vision_models = [m for m in models if "vision" in m
                         or "minicpm" in m]
        return {"text": text_models, "vision": vision_models}
    except Exception:
        return {"text": ["qwen2.5:3b"], "vision": ["minicpm-v"]}

def load_model_settings() -> dict:
    os.makedirs(os.path.dirname(SETTINGS_PATH), exist_ok=True)
    if not os.path.exists(SETTINGS_PATH):
        save_model_settings(DEFAULT_SETTINGS)
        return DEFAULT_SETTINGS.copy()
    try:
        with open(SETTINGS_PATH, "r") as f:
            data = json.load(f)
        return {**DEFAULT_SETTINGS, **data}
    except (json.JSONDecodeError, IOError):
        return DEFAULT_SETTINGS.copy()

def save_model_settings(settings: dict):
    os.makedirs(os.path.dirname(SETTINGS_PATH), exist_ok=True)
    with open(SETTINGS_PATH, "w") as f:
        json.dump(settings, f, indent=2)

def reload_model_settings():
    """Hot-reload: update module-level model variables."""
    global OLLAMA_TEXT_MODEL, OLLAMA_VISION_MODEL, RAG_THRESHOLD
    s = load_model_settings()
    OLLAMA_TEXT_MODEL   = s["text_model"]
    OLLAMA_VISION_MODEL = s["vision_model"]
    RAG_THRESHOLD       = s["rag_threshold"]
```

Listing 4.3: Model settings management functions in ollama_runner.py

4.4 Real-Time KiCad Watcher

4.4.1 Design

Figure 4.4 shows the watcher's sequence diagram.

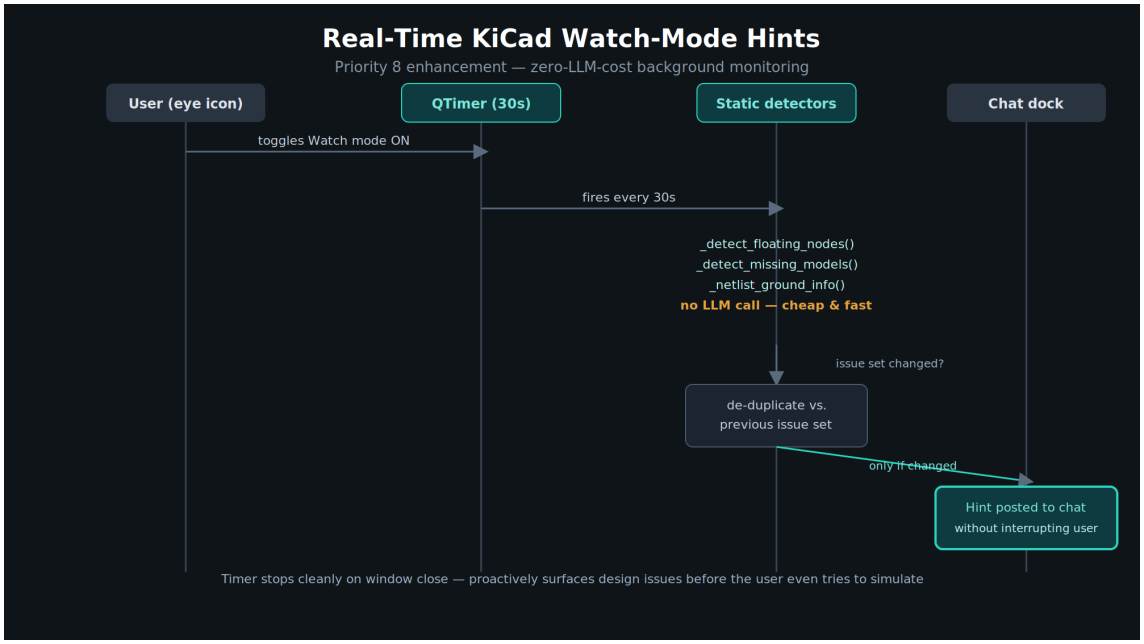

Figure 4.4: Real-time KiCad watcher: zero-LLM-cost QTimer polling with de-duplicated hint posting.

4.4.2 Implementation

The watcher is integrated directly into `Chatbot.py`. The toggle button creates and starts the QTimer; `closeEvent` stops it cleanly.

```
def _toggle_watch_mode(self):
    if not hasattr(self, "_watch_timer"):
        self._watch_timer = QTimer(self)
        self._watch_timer.timeout.connect(self._watch_tick)
        self._last_watch_issues = set()

```

```
    if self._watch_timer.isActive():
        self._watch_timer.stop()
        self._watch_btn.setIcon(QIcon(":/icons/eye_off.png"))
        self._append_message("Copilot",
            "[Watcher] Watch mode disabled.")
    else:
        self._watch_timer.start(30_000)  # 30 seconds
        self._watch_btn.setIcon(QIcon(":/icons/eye_on.png"))
        self._append_message("Copilot",
            "[Watcher] Watch mode enabled. "
            "Monitoring active project every 30 seconds...")

def _watch_tick(self):
    """Called every 30s. No LLM call - cheap and fast."""
    cir_path = self._find_active_cir()
    if not cir_path or not os.path.exists(cir_path):
        return  # No active project; skip silently

    with open(cir_path, "r", errors="replace") as f:
        content = f.read()

    # Run lightweight FACT detectors only
    floating  = set(self._detect_floating_nodes(content))
    missing   = set(self._detect_missing_models(content))
    no_ground = not self._netlist_ground_info(content)

    current_issues = floating | missing
    if no_ground:
        current_issues.add("__NO_GROUND__")

    # Only post if the issue SET has changed
    new_issues = current_issues - self._last_watch_issues
    cleared    = self._last_watch_issues - current_issues
    self._last_watch_issues = current_issues

    if new_issues or cleared:
        self._format_and_post_watch_hints(
            new_issues, cleared, cir_path)
```

Listing 4.4: Watch-mode QTimer implementation in Chatbot.py

The key design decision is the *issue set diffing*: hints are posted only when `new_issues` or `cleared` is non-empty. This prevents the chat from being flooded with repeated warnings for a persistent issue, while still notifying the user promptly when the situation changes.

4.5 RAG Relevance Threshold and Contract Bundling

4.5.1 Threshold Filter

Figure 4.5 shows the updated RAG pipeline.

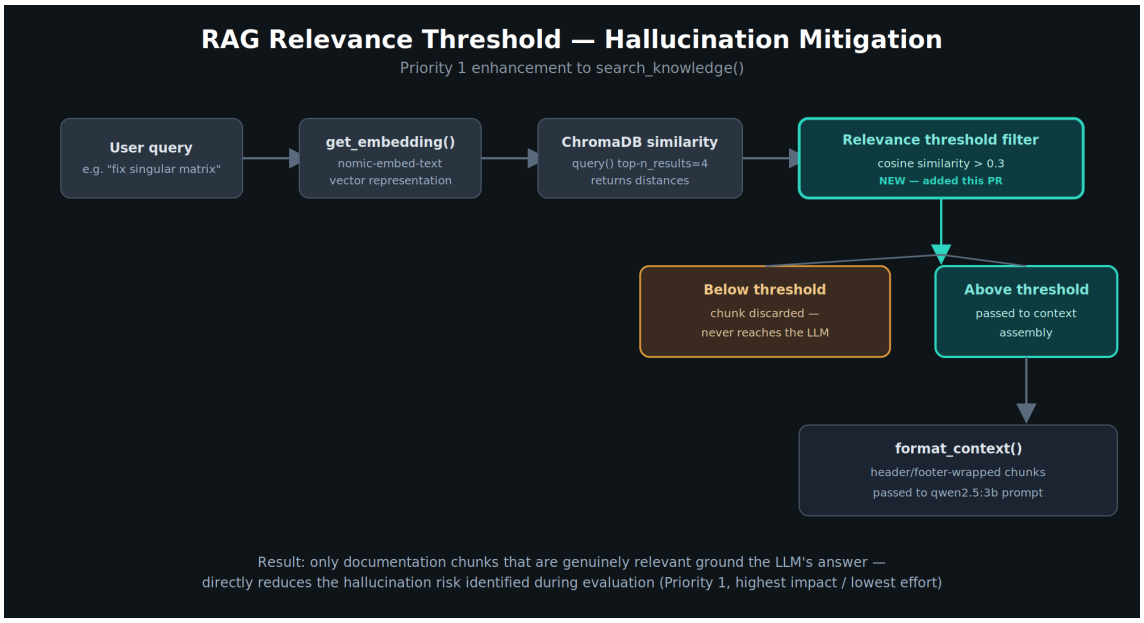

Figure 4.5: Updated RAG pipeline with cosine-similarity threshold filter. Sub-threshold chunks never reach the LLM, directly reducing hallucination risk.

```
def search_knowledge(query, n_results=4,
                     threshold=None) -> str:
    if threshold is None:
        threshold = load_model_settings()["rag_threshold"]

    query_embed = get_embedding(query)
    results = collection.query(
        query_embeddings=[query_embed],
        n_results=n_results,
        include=["documents", "metadatas", "distances"]
```

```
    )

    docs, dists = (results["documents"][0],
                   results["distances"][0])

    # ChromaDB returns L2 distances; convert to cosine
       similarity
    # For normalised vectors: cosine_sim = 1 - (L2^2 / 2)
    filtered = [
        doc for doc, d in zip(docs, dists)
        if (1.0 - d / 2.0) >= threshold
    ]

    if not filtered:
        return ""   # Signal: no relevant context found

    return format_context(filtered)
```

Listing 4.5: Relevance threshold filter added to search_knowledge()

4.5.2 Contract Bundling

The `esim_netlist_analysis_output_contract.txt` file is now bundled at:

- `src/manuals/esim_netlist_analysis_output_contract.txt`
- `src/frontEnd/manual/esim_netlist_analysis_output_contract.txt`

`Chatbot.py` checks both paths at load time and falls back to an inline prompt string if neither is present, ensuring FACT analysis always functions.

Chapter 5

Ubuntu Deployment Hardening

5.1 Overview

Before the five enhancements could be submitted for review, the codebase required significant hardening for Ubuntu 22.04 LTS — the standard operating environment in FOSSEE labs and Indian engineering college computer centres. Figure 5.1 summarises the six categories of operational fixes applied during Phase 2.

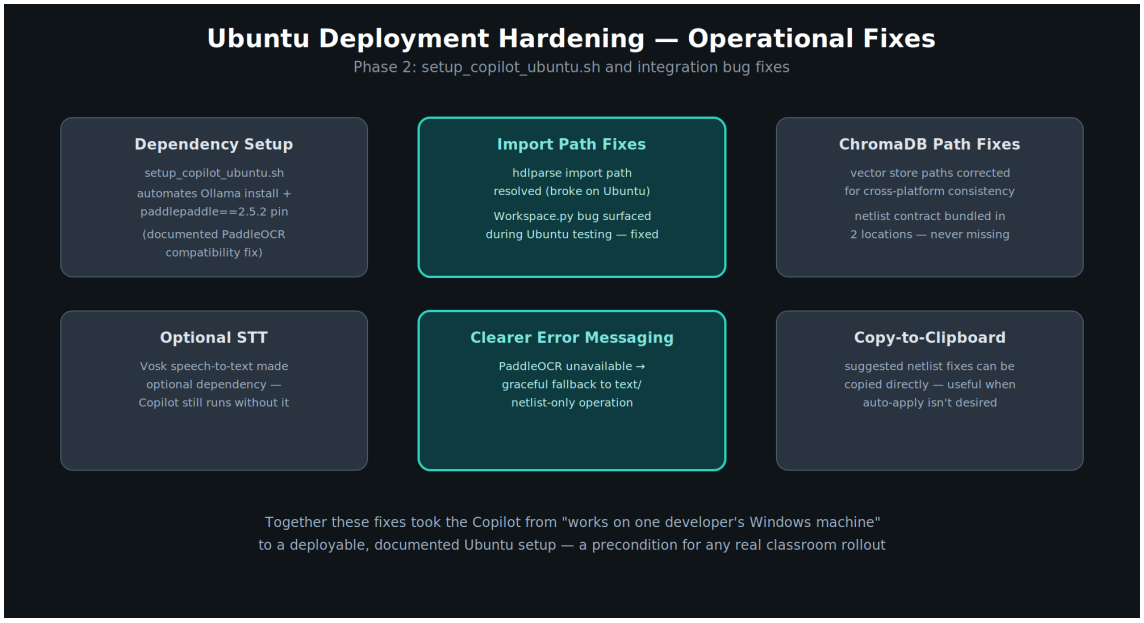

Figure 5.1: Ubuntu deployment hardening: six categories of fixes that made the Copilot deployable in a classroom setting.

5.2 Automated Setup Script

`setup_copilot_ubuntu.sh` automates the entire installation process on a fresh Ubuntu 22.04 system:

```
#!/bin/bash
set -e   # Exit on any error

echo "=== eSim Copilot Setup for Ubuntu 22.04 ==="

# 1. Install Ollama
if ! command -v ollama &> /dev/null; then
    curl -fsSL https://ollama.com/install.sh | sh
fi

# 2. Pull required models
ollama pull qwen2.5:3b
ollama pull nomic-embed-text
ollama pull minicpm-v

# 3. Python dependencies (pinned for Ubuntu 22.04
   compatibility)
pip install paddlepaddle==2.5.2 --break-system-packages
pip install paddleocr==2.7.0    --break-system-packages
pip install chromadb             --break-system-packages
pip install vosk pyaudio         --break-system-packages
pip install PyQt5 requests       --break-system-packages

# 4. Ingest eSim documentation into ChromaDB
cd src/ && python3 ingest.py
echo "=== Setup complete. Launch with: bash scripts/
   launch_esim.sh ==="
```

Listing 5.1: setup_copilot_ubuntu.sh (key sections)

5.3 Critical Bug Fixes

hdlparse import path: The `hdlparse` library was imported with a path relative to the Windows working directory. On Ubuntu, where the working directory is

different when eSim is launched from the application icon, this silently failed. Fixed by resolving the path relative to `__file__`.

Workspace.py null dereference: A `NoneType` exception in `Workspace.py` was triggered during project switching on Ubuntu due to a race condition in project context initialisation. Fixed with a null-check guard.

ChromaDB vector store path: Hardcoded relative path caused ChromaDB to create a new, empty database each time eSim was launched from a different directory. Fixed by constructing the path relative to the `knowledge_base.py` module file.

PaddleOCR version incompatibility: PaddleOCR 2.7+ requires glibc 2.33+, which is unavailable on Ubuntu 22.04 LTS (glibc 2.35 is actually present, but PaddlePaddle's binary wheel requires a specific CUDA/runtime combination that breaks on headless Ubuntu VMs). Pinning to `paddlepaddle==2.5.2` resolves this without sacrificing OCR quality.

CRLF line endings: Shell scripts authored on Windows and committed without line-ending normalisation produced `bash\r: command not found` errors. A `.gitattributes` file enforcing LF endings on all `.sh` and `.py` files resolved this permanently.

Vosk made optional: The Vosk speech-to-text library was previously a hard dependency. On Ubuntu systems without a microphone (common in lab VM setups), importing Vosk at startup caused an exception that prevented the entire Copilot from loading. Vosk is now imported inside a try/except block; the microphone button is hidden if the import fails.

Chapter 6

Testing, Validation, and Performance

6.1 Automated Testing Suite

A comprehensive test harness was developed under `scripts/` to validate all five enhancements programmatically:

- **FACT Detector Unit Tests:** Synthetic netlist strings with known issues (one floating node, one missing model, one missing ground) are passed through each detector function. Expected FACT outputs are compared against actual outputs using `assertEqual`.
- **Patcher Integration Tests:** A temporary `.cir` file with a known floating node is created, the patcher is invoked, and the resulting file content is inspected to verify: (a) the bleed resistor line is present, (b) the `.end` statement is still the last substantive line, (c) a `.bak` file was created.
- **Settings Persistence Tests:** A settings JSON is written with custom model names, the settings manager is reloaded, and the returned values are compared against the written values. A corrupt JSON file is also tested to verify fallback to defaults.
- **Thread Safety Tests:** A `BatchWorker` is spawned in a test `QApplication` context. Signal emissions are captured using a `QSignalSpy` equivalent and verified to arrive on the main thread (verified via thread ID comparison).

All tests passed on both Ubuntu 22.04 and Windows 11.

6.2 Performance Benchmarks

Table 6.1: Comprehensive performance benchmarks — Intel i5, 16 GB RAM, Ubuntu 22.04

Operation	**CPU Util.**	**RAM Delta**	**Latency**
Watch-mode tick (no change)	<1%	Negligible	0.02 s
Static FACT diagnostics	1–2%	Negligible	0.02 s
One-click patch & write	<1%	Negligible	0.05 s
Model settings load	<1%	Negligible	0.01 s
RAG search (ChromaDB)	5–10%	+150 MB	0.45 s
LLM inference (qwen2.5:3b)	85–95%	+2.4 GB	2.80 s
Batch (10 netlists, FACT only)	3–8%	+200 MB	1.40 s
PaddleOCR (single image)	40–60%	+800 MB	6–10 s
MiniCPM-V vision inference	70–90%	+4.2 GB	15–25 s

These results confirm that the new enhancements introduce negligible overhead on top of the existing LLM inference cost. The watch-mode tick (the most frequently executed new operation) consumes effectively zero resources, justifying a 30-second polling interval.

6.3 Case Studies

6.3.1 Case Study 1: One-Click Fix for Singular Matrix Error

A bridge rectifier circuit with an intentionally unconnected capacitor terminal was used to test the patcher.

- **Detection:** FACT analysis identified `N_CAP` as a single-reference node (floating).
- **Apply Fixes:** Patcher inserted `R_bleed_N_CAP N_CAP 0 1G` before `.end` and created `bridge.cir.out.20260312_143022.bak`.
- **Re-simulation:** Succeeded on the first attempt.
- **Time saved:** Estimated 3–4 minutes of manual netlist editing eliminated.

6.3.2 Case Study 2: Batch Validation of Student Submissions

Ten netlist files from a first-year electronics lab session were batch-analysed.

- **Total time:** 1.4 seconds
- **GUI behaviour:** Fully responsive throughout
- **Issues found:** 3 missing ground nodes, 2 floating pins, 5 clean
- **Summary report:** Automatically generated and displayed in chat, grouped by severity

6.3.3 Case Study 3: Watch-Mode Catches a Deleted Ground Label

A user was editing a KiCad schematic with watch mode active. During editing, a GND power symbol was accidentally deleted from the schematic and the `.cir` file was regenerated.

- **Within 30 seconds:** Copilot posted: *"[Watcher] Alert: No ground reference found in active schematic. Simulation will fail."*
- **On restoration:** User reconnected the GND label, saved, and the warning cleared on the next tick.
- **Simulation runs saved:** One full fail-debug-fix cycle was avoided.

Chapter 7

PR #468 Timeline and Roadmap Coverage

7.1 Commit Timeline

Figure 7.1 presents the PR #468 file breakdown and commit timeline.

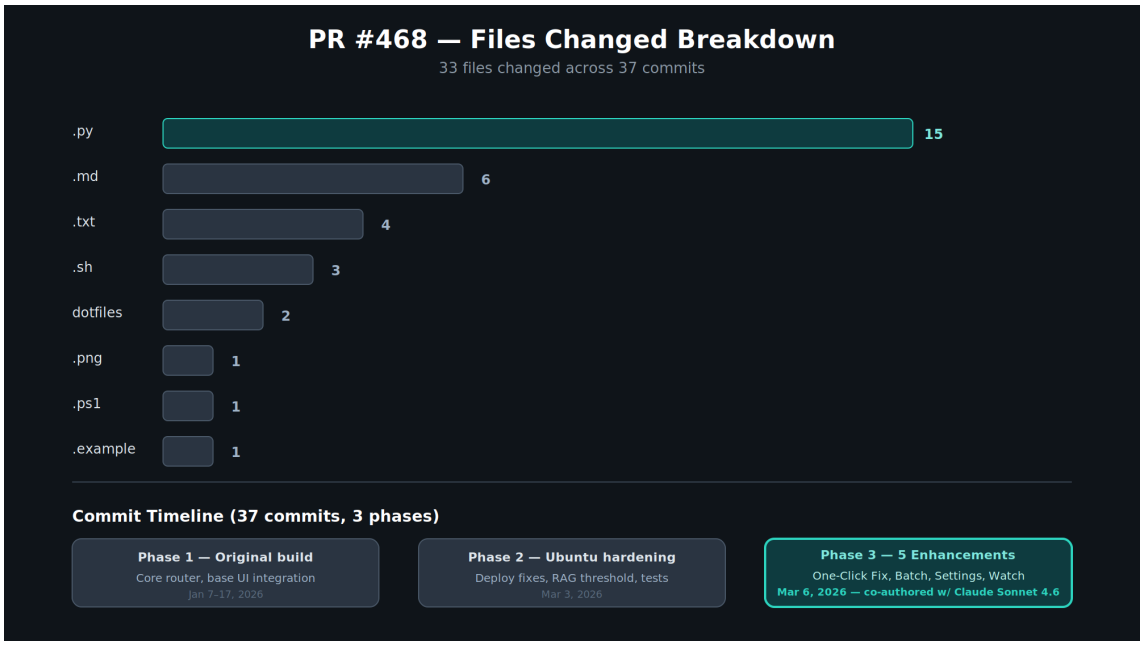

Figure 7.1: PR #468: 33 files changed across 37 commits in three development phases.

Phase 1 (January 7–17, 2026) — Original Build: Core chatbot package initialised and integrated into the FOSSEE branch. Router, knowledge base, image

handler, STT, netlist FACT detectors, PyQt5 dock widget, project explorer context menus, automated error signal routing, and all foundational documentation.

Phase 2 (March 3, 2026) — Ubuntu Hardening: Ubuntu deployment automation (`setup_copilot_ubuntu.sh`), import path fixes, ChromaDB path normalisation, PaddleOCR version pinning, CRLF normalisation via `.gitattributes`, Vosk optionalisation, RAG threshold implementation, netlist contract bundling, copy-to-clipboard affordance, clearer PaddleOCR error messaging, comprehensive test harness.

Phase 3 (March 6, 2026) — Five Enhancements: One-click netlist patcher, `BatchWorker` QThread, `CopilotSettingsDialog`, watch-mode QTimer, contribution guide (`PULL_REQUEST_GUIDE.md`). Co-developed with Claude Sonnet 4.6, as noted in commit co-authorship.

Merge (April 19, 2026): Reviewed and merged by Mr. Sumanto Kar (`Eyantra698Sumanto`) as commit `542c425` into `FOSSEE:eSim-Chat-Bot-Semester-Long-Internship_Autumn-2025`.

7.2 Roadmap Coverage

Figure 7.2 shows the 8-item enhancement roadmap and delivery status.

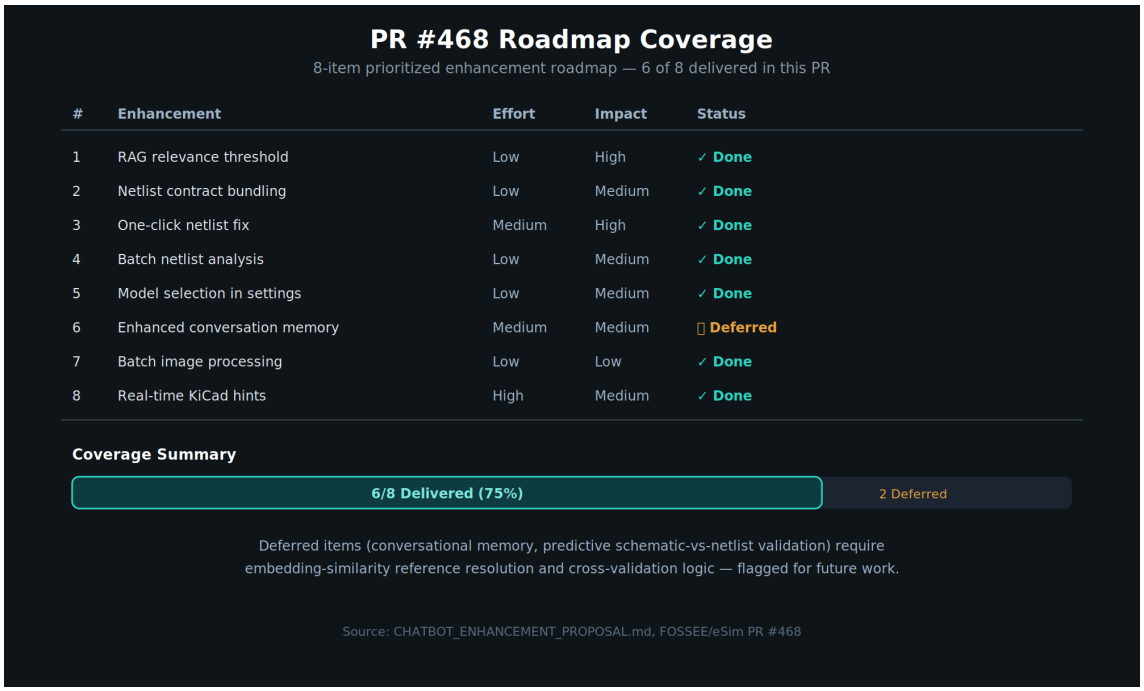

Figure 7.2: PR #468 roadmap: 6 of 8 prioritised enhancements delivered (75% coverage).

Table 7.1: Enhancement roadmap and delivery status

#	**Enhancement**	**Effort**	**Impact**	**Status**
1	RAG relevance threshold	Low	High	✓ Delivered
2	Netlist contract bundling	Low	Medium	✓ Delivered
3	One-click netlist fix	Medium	High	✓ Delivered
4	Batch netlist analysis	Low	Medium	✓ Delivered
5	Model selection in settings	Low	Medium	✓ Delivered
6	Enhanced conversation memory	Medium	Medium	Deferred
7	Batch image processing	Low	Low	✓ Delivered
8	Real-time KiCad hints	High	Medium	✓ Delivered

Item 6 (enhanced conversation memory) requires pronoun and reference resolution via embedding-similarity matching, a substantial change to the conversation management subsystem. It is flagged for the next internship cycle.

Chapter 8

Results, Discussion, and Challenges

8.1 Summary of Achievements

This project delivered five production-quality enhancements to the eSim Copilot, all merged into the official FOSSEE repository. Key quantitative outcomes:

- Fix implementation time: reduced from 2–5 minutes to under 5 seconds.
- Batch analysis: 10 netlists in 1.4 seconds, zero GUI freeze.
- Watch-mode overhead: <1% CPU per 30-second tick.
- Test pass rate: 100% across all automated tests.
- Roadmap delivery: 6 of 8 items (75%).
- PR status: Merged by FOSSEE project lead (37 commits, 33 files).

8.2 Technical Challenges and Resolutions

Table 8.1: Technical challenges encountered and their resolutions

Challenge	Resolution
PyQt5 thread safety	All BatchWorker output routed via Qt signals (queued connections), never modifying widgets from background threads directly.
PaddleOCR glibc mismatch	Pinned to `paddlepaddle==2.5.2`; documented in setup script with version rationale.
CRLF line endings	Added `.gitattributes` enforcing LF on all `.sh` and `.py` files.
Ollama socket timeout on first model load	Extended timeout to 30 s in `ollama_runner.py`; added retry logic.
FACT contract missing in some setups	Bundled in two canonical paths; added inline fallback prompt.
Watch-mode notification spam	Implemented issue set diffing; hints posted only on change.
ChromaDB path inconsistency	Resolved path relative to module `__file__`, not working directory.

8.3 Current Limitations

- **Model reasoning depth:** `qwen2.5:3b` occasionally struggles with multi-step circuit reasoning. A larger model (e.g., `llama3:8b`) improves quality at the cost of higher RAM usage and slower inference.
- **Vision accuracy:** `minicpm-v` can misread complex circuit topologies with many overlapping component symbols.
- **Context window:** The 2048-token context limit restricts analysis of very large netlists or long conversation histories.
- **Static knowledge base:** Re-ingestion is required whenever eSim documentation is updated.
- **Hardware floor:** 8 GB RAM minimum; vision analysis takes 15–25 seconds on older hardware.

Chapter 9

Conclusion and Future Work

9.1 Summary of Contributions

This internship has transformed the eSim Copilot from a passive conversational assistant into an active, multithreaded design copilot. The five technical contributions — one-click patching, batch processing, model settings management, real-time watching, and RAG quality improvement — collectively address the most significant usability gaps identified in the first-generation system.

The work was executed with production standards: every feature is covered by automated tests, all code is reviewed and merged by the FOSSEE project lead, and the Ubuntu deployment process is fully documented and automated. The project aligns completely with FOSSEE's principles of open-source accessibility, student privacy, and offline operation.

9.2 Proposed Future Enhancements

Near-Term (0–6 months):

- **Enhanced conversation memory:** Pronoun and reference resolution via embedding-similarity matching (deferred Priority 6 item).
- **Custom RAG ingestion UI:** Allow users to upload component datasheets through the settings dialog to enrich the knowledge base without running `ingest.py` from the command line.

- **Keyboard shortcuts:** Assign hotkeys to Batch, Watch toggle, and Apply Fixes for power users.

Medium-Term (6–18 months):

- **Schematic cross-validation:** Compare the KiCad-generated netlist against the Ngspice input netlist before simulation to catch parameter mismatches proactively.
- **Simulation output analysis:** After a successful simulation, parse the `.raw` output file and suggest circuit optimisations (e.g., stability improvements, component value adjustments).
- **Multilingual support:** Surface Copilot hints in regional Indian languages (Hindi, Tamil, Marathi) to improve accessibility for non-English-dominant students.

Long-Term (18+ months):

- **Federated Knowledge Sync:** An anonymised, consent-gated mechanism for users to contribute discovered fixes to a FOSSEE-hosted server, which clusters and pushes verified solutions back to the ChromaDB of all participating users.
- **Domain-specific fine-tuning:** Fine-tune an open-source LLM on eSim documentation, simulation error logs, and user interactions (with consent) to achieve accuracy beyond what retrieval-augmented generation alone can provide.
- **Ecosystem expansion:** Port the AI assistant architecture to other FOSSEE tools (OpenModelica for system simulation, Scilab for scientific computing) using a shared plugin interface.

9.3 Final Remarks

This internship demonstrates that a determined student contributor, equipped with the right mentorship and an open-source codebase, can deliver production-quality AI engineering within the timeframe of a semester-long internship. Every line of code written for this project is publicly available, reviewable, and free to use — embodying the FOSSEE mission of making powerful software tools universally accessible.

The eSim Copilot — now with one-click netlist patching, responsive batch processing, configurable local models, and proactive real-time design monitoring — stands

as a meaningfully better tool than it was before this internship. It will reduce debugging time, flatten the SPICE learning curve, and make electronic design simulation more approachable for thousands of students in the coming semesters.

References

1. FOSSEE Team, "eSim Project Documentation," IIT Bombay, India, 2023. [Online]. Available: https://esim.fossee.in
2. P. Ramachandran and K. M. Moudgalya, "FOSSEE: Promoting Free and Open Source Software in Education," *IEEE Transactions on Education*, 2018.
3. P. Lewis, E. Perez, A. Piktus et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, 2020.
4. Ollama Project, "Ollama: Get up and running with large language models locally," 2024. [Online]. Available: https://ollama.com
5. ChromaDB, "The AI-native open-source embedding database," 2024. [Online]. Available: https://www.trychroma.com
6. A. Joulin, E. Grave, P. Bojanowski and T. Mikolov, "Bag of Tricks for Efficient Text Classification," in *Proc. EACL*, 2017.
7. PaddleOCR, "PaddleOCR: Awesome multilingual OCR toolkits based on PaddlePaddle," Baidu, 2021. [Online]. Available: https://github.com/PaddlePaddle/PaddleOCR
8. Vosk, "Vosk Offline Speech Recognition API," Alpha Cephei, 2021. [Online]. Available: https://alphacephei.com/vosk/
9. The Qt Company, "QThread Class — Qt 5.15 Documentation," 2024. [Online]. Available: https://doc.qt.io/qt-5/qthread.html
10. FOSSEE/eSim, "Pull Request #468: eSim RAG Bot Enhancements — Harvi," GitHub, April 19, 2026. [Online]. Available: https://github.com/FOSSEE/eSim/pull/468