Summer Fellowship 2026

On

Architectural Stabilization, PyQt6 Migration, and UI/UX Refactoring of the eSim Tool Manager

Submitted by

Bhavya Bhardwaj
VIT Bhopal University

Under the guidance of

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

Acknowledgment

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

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

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

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

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

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

Contents

Chapter 1

Introduction

1.1 Background

The Free and Open Source Software for Education (FOSSEE) project at IIT Bombay has long championed the democratization of technical education through open-source software. Supported by the Ministry of Education, Government of India, FOSSEE aims to reduce the reliance of Indian academic institutions on expensive, proprietary software licenses. By developing, maintaining, and distributing high-quality open-source alternatives, FOSSEE ensures that students, educators, and hobbyists worldwide have unrestricted access to professional-grade engineering tools. The ecosystem encompasses various domains, including computational fluid dynamics, structural engineering, and electronic design automation. This fellowship was specifically centered around the latter, focusing on the enhancement and stabilization of critical electronic design infrastructure.

1.2 Overview of eSim

At the forefront of FOSSEE's electronic design initiatives is eSim, an advanced Electronic Design Automation (EDA) tool designed for circuit design, simulation, and printed circuit board (PCB) routing. Unlike monolithic proprietary software, eSim is ingeniously architected as a cohesive orchestration layer. It seamlessly integrates powerful underlying open-source utilities such as KiCad for schematic capture and PCB layout, Ngspice for mixed-level/mixed-signal circuit simulation, and GHDL/Verilator for digital hardware description language (HDL) parsing.

By abstracting the complexities of these individual tools, eSim provides a unified, user-friendly environment. A student can draw a schematic, map components, generate a netlist, and view the transient simulation waveforms without ever leaving the eSim GUI or manually interacting with the command line.

However, because eSim relies so heavily on a multitude of third-party external tools, managing these dependencies across radically different operating systems (Windows, Ubuntu, macOS) poses a monumental challenge. The **Automated Tool Manager** was conceived to solve this exact problem. It is a critical, heavy-duty subsystem within eSim responsible for detecting, installing, configuring, and updating these external dependencies completely transparently to the user.

1.3 Objectives of the Project

The primary objective of this fellowship was an end-to-end, full-stack modernization and stabilization of the eSim Tool Manager. The scope of work spanned deep backend OS-level architecture all the way up to user-facing frontend PyQt design. The core mandates were meticulously defined as follows:

- **Framework Modernization (PyQt6 Migration):** The core eSim application was transitioning from PyQt5 to PyQt6. The legacy Tool Manager codebase had to be entirely rewritten and migrated to ensure strict compatibility with modern Qt environments.
- **Core Stability Improvements:** QA testing revealed critical application crashes, such as segmentation faults within the Project Explorer on right-click events, and silent directory parsing failures on Windows paths containing whitespace. These foundational bugs required immediate resolution.
- **Native GUI Integration:** The original Tool Manager was built with a flawed detached-subprocess architecture (launching as a standalone popup). The mandate was to rip out this detached logic and embed the Tool Manager natively as an integrated tab within the main eSim application.
- **Backend Idempotency & Optimization:** The legacy installation scripts lacked state awareness. If a user accidentally re-clicked an installation button, the system would spend 10 minutes re-downloading gigabytes of data. Implementing intelligent idempotency checks was paramount. Furthermore, subprocess stream reading needed severe optimization to prevent terminal buffer hanging on Windows during massive extractions (the 99% stall bug).
- **Architectural Review & Unification:** Conduct comprehensive architectural audits to enforce DRY (Don't Repeat Yourself) principles. Review massive community PRs, reject overly convoluted Object-Oriented paradigms, and champion a flat, declarative script architecture to unify the Windows and Linux backend deployment workflows.

1.4 Methodology

An agile, highly iterative software engineering methodology was utilized throughout the fellowship. The work began with a rigorous root-cause analysis phase, deeply studying OS-level IPC (Inter-Process Communication) pipes, Python threading modules, and strict PyQt6 constraints. Codebase modernization followed a systematic pipeline of legacy extraction, enumeration translation, native embedding, and cross-platform regression testing. Collaboration was heavily emphasized, requiring constant coordination with Linux backend engineers, QA testers, and UI designers via GitHub pull requests, code reviews, and issue tracking.

Chapter 2

Literature Survey & Theoretical Context

2.1 Evolution of Open-Source EDA Tools

The landscape of Electronic Design Automation has historically been dominated by proprietary, closed-source giants such as Cadence, Synopsys, and Mentor Graphics (Siemens EDA). While these tools offer unparalleled capabilities for cutting-edge semiconductor nodes, their exorbitant licensing fees make them entirely inaccessible to the vast majority of engineering students and hobbyists.

In response to this monopoly, the open-source community developed highly capable alternative tools. KiCad emerged as the de facto standard for open-source schematic capture and PCB layout, boasting adoption by major organizations like CERN. Ngspice provided robust SPICE simulation capabilities, while GHDL and Verilator cornered the open-source VHDL and Verilog simulation markets, respectively. However, these tools operate as isolated silos. A user wishing to design a circuit and simulate its transient response would traditionally have to manually orchestrate the data flow between KiCad and Ngspice, exporting netlists and configuring path variables by hand—a daunting task for undergraduate students. eSim bridges this gap by providing a unified workflow.

2.1.1 Algorithmic Underpinnings of the EDA Ecosystem

The eSim environment acts as an orchestration layer over three mathematically complex backend tools, each solving vastly different computational problems:

- **Ngspice (Analog/Mixed-Signal Simulation):** At its core, Ngspice relies on Modified Nodal Analysis (MNA) to generate large, sparse matrices representing circuit topologies. For transient analysis, it employs numerical integration methods—predominantly the Trapezoidal rule or Gear's method—to solve stiff non-linear differential equations across discrete time steps. Ensuring convergence in these matrices is computationally intensive, requiring robust sparse linear solvers like KLU.

- **KiCad (PCB Routing):** The layout and routing engine within KiCad op-

erates on complex computational geometry paradigms. The Push and Shove router utilizes a topological clearance algorithm, dynamically adjusting copper traces while mathematically guaranteeing design rule constraint (DRC) boundaries are preserved in real-time.

- **Verilator (Digital Simulation):** Unlike traditional event-driven simulators, Verilator translates synthesizable Verilog code directly into highly optimized, cycle-accurate C++ models. This eliminates the overhead of managing a dynamic event queue, resulting in execution speeds that are orders of magnitude faster, mathematically reducing the simulation problem to a continuous deterministic state machine.

2.2 The GUI Framework: Qt Event Loops and Threading

To achieve cross-platform cohesion, eSim utilizes the Qt framework, specifically the PyQt bindings for Python. Qt is a mature, C++ based framework that relies on a complex internal Event Loop to manage user interactions, rendering, and signal-slot communications.

A critical theoretical concept in GUI development is the necessity of asynchronous threading. The main thread in a PyQt application is exclusively responsible for rendering the UI and processing user events (clicks, scrolls, typing). If a heavy, blocking operation—such as downloading a 1GB ZIP file or extracting 10,000 files—is executed on the main thread, the Event Loop is stalled. The Operating System detects that the application is no longer processing events and marks the window as "Not Responding," leading to a catastrophic user experience.

Therefore, the eSim Tool Manager relies heavily on Python's `threading` module and Qt's `QThread` architecture. Backend installation scripts are spawned in isolated background worker threads, communicating progress back to the main UI thread via thread-safe PyQt `Signals`.

2.2.1 The Qt Signal/Slot Architectural Paradigm

Traditional graphical interfaces often rely on callback functions to handle user interactions. Callbacks pass a pointer to a function that is executed when an event occurs. This paradigm breaks down in multi-threaded environments due to severe memory corruption risks if a background thread triggers a callback that attempts to modify UI elements owned by the main thread.

Qt mitigates this via the Signal and Slot mechanism. When an object changes its state, it emits a mathematically defined *Signal*. Other objects can connect their *Slots* (executable functions) to this Signal. Crucially, when Signals are passed across thread boundaries, Qt automatically marshals them into an asynchronous event queue. This completely decouples the emitting worker thread from the receiving UI thread, mathematically preventing race conditions and ensuring thread-safe GUI updates without requiring manual mutex locking.

2.2.2 The PyQt5 to PyQt6 Paradigm Shift

For many years, eSim was built atop PyQt5. However, the release of Qt6 introduced breaking changes to the underlying C++ API, which cascaded directly into the Python bindings. PyQt6 strictly enforces enumeration scoping. In PyQt5, developers could access flags loosely, for example, by calling `Qt.AlignTop`. In PyQt6, the namespace architecture was radically tightened. Attempting to use the loose flag results in a fatal `AttributeError`, requiring the fully qualified path `Qt.AlignmentFlag.AlignTop`.

While this strict scoping was initially burdensome for migrating legacy codebases, it vastly improves the maintainability, type safety, and readability of the code by explicitly defining the namespace and context of every configuration flag. The migration of the eSim Tool Manager to PyQt6 was not merely a syntactic update; it was a fundamental modernization of the application's core rendering engine.

2.2.3 The Mathematics of High-DPI Vector Rendering

Another crucial theoretical challenge in modern GUI development is High-DPI (Dots Per Inch) scaling. Historically, desktop applications relied on raster graphics (pixel-based bitmaps) and hard-coded pixel coordinates for layout (e.g., placing a button exactly at $x = 100, y = 200$). On legacy 1080p monitors, this was sufficient. However, on modern 4K and Retina displays, pixel density increases dramatically. If a button is hard-coded to be 50 pixels wide, it will appear microscopic on a 4K display.

Qt6 abandons pixel-perfect absolute positioning in favor of mathematical vector scaling and dynamic layout management. Rather than defining pixel coordinates, the developer defines mathematical relationships (e.g., "Button A must remain vertically centered within Layout B, maintaining a 5% margin"). The Qt rendering engine calculates the physical pixel density of the host monitor and applies a dynamic floating-point scaling factor. Font rendering utilizes TrueType and OpenType mathematical curves (Bézier curves) which are rasterized in real-time on the GPU, guaranteeing infinitely crisp text regardless of monitor resolution.

2.3 OS-Level Inter-Process Communication (IPC)

A significant portion of this fellowship dealt with deep Operating System-level constraints, specifically Inter-Process Communication (IPC) via unidirectional pipes. When a Python application executes an external binary (e.g., a Bash script or a Windows installer) using the `subprocess.Popen` module, the OS establishes data channels connecting the standard output (stdout) and standard error (stderr) of the child process to the parent Python process.

2.3.1 The Pipe Buffer Deadlock Theory

A critical theoretical concept necessary to understand the bugs resolved in this project is the OS pipe buffer limit. On Windows, the standard pipe buffer size is

typically limited to 64KB. If the child process generates text output rapidly (such as extracting a massive ZIP file) and the parent process does not actively read from the pipe, the 64KB buffer fills up entirely.

Once the buffer is full, the OS intervenes and suspends the child process, waiting for the parent to consume the data and clear the buffer. If the parent process is simultaneously executing a blocking `wait()` call (waiting for the child to finish before reading the output), a classic deadlock occurs. The parent waits for the child to finish, but the child is suspended waiting for the parent to read the buffer. The application simply freezes indefinitely. Understanding this IPC theory was paramount to resolving the KiCad extraction stalls documented later in this report.

2.4 Cross-Platform Deployment Architecture

Deploying software dependencies varies wildly depending on the host OS.

- **Linux (Ubuntu/Debian):** Dependency management is elegantly handled via Advanced Package Tool (APT) and meta-packages (`.deb`). The system inherently understands dependencies, conflict resolution, and global PATH injection.
- **Windows:** Windows lacks a native, globally standardized package manager of this caliber. Open-source tools like KiCad or MSYS2 (for GHDL/Verilator) must be downloaded as raw binaries or ZIP archives, manually extracted, and heavily manipulated to inject their execution directories into the global Windows `PATH` environment variable.

This theoretical divergence dictates that the eSim Tool Manager must abstract these complexities, employing completely different backend logic for Windows versus Linux while presenting identical frontend progress bars to the user.

Chapter 3

Project Timeline & Agile Sprints

The summer fellowship was executed following an Agile methodology, broken down into two-week sprints. The table below outlines the core milestones, deliverables, and primary focus areas throughout the project.

Table 3.1: Summer Fellowship Agile Sprint Timeline

Sprint	**Phase Focus**	**Key Deliverables & Challenges**
Weeks 1 - 2	Codebase Familiarization	• Cloned `eSim` repository. • Explored Linux architecture vs Windows limitations. • Formulated initial migration strategy.
Weeks 3 - 4	Dependency Extraction	• Diagnosed the 987 duplicated file bottleneck. • Extracted `Ubuntu-Integrate` into dedicated submodules. • Freed 400MB of repository space.

Continued on next page

Table 3.1 – *Continued from previous page*

Sprint	**Phase Focus**	**Key Deliverables & Challenges**
Weeks 5 - 6	PyQt6 Migration	• Upgraded core UI logic from PyQt5 to PyQt6. • Handled strict enumeration enforcement. • Ported legacy Context Menus to unified architecture.
Weeks 7 - 8	Core Backend & Windows QA	• QA Phase on Windows 11 identified critical bugs. • Resolved `libgcc` DLL errors and MSYS2 environment paths. • Fixed 99.5% installation deadlocks.
Weeks 9 - 10	Code Review & Final Polish	• Submitted massive 5000+ line PR. • Refactored into smaller, atomic commits. • Finalized UI responsiveness and High-DPI scaling.

This structured approach ensured continuous delivery and immediate feedback from the core maintainers, ultimately culminating in a stable Windows release.

Chapter 4

Problem Statement & Initial Codebase Extraction

4.1 Problem Statement

Prior to the commencement of this fellowship, the legacy Tool Manager subsystem was in a precarious state. It was originally authored by previous contributors as a completely standalone entity, designed to be launched as an independent Python desktop application via the terminal, rather than integrated seamlessly into eSim.

This fragmentation caused severe usability and state synchronization issues. Furthermore, the codebase was heavily bloated, scattered across multiple redundant directories, and fundamentally incompatible with the new PyQt6 framework that the rest of the eSim team was migrating toward. To make matters worse, QA testers were consistently reporting catastrophic backend bottlenecks such as the "10-Minute Bug," terminal buffer deadlocks during zip extractions on Windows, and silent application crashes on Linux.

4.2 Phase 1: The Baseline Duplication Nightmare (PR #518)

The very first objective was to locate the Tool Manager source code and prepare it for migration. However, upon reviewing closed Pull Request #494 and analyzing the repository, I encountered a massive architectural anomaly.

The original PR merge that introduced the Tool Manager into the main eSim repository accidentally introduced four massive, bloated root folders (`Ubuntu-Integrate/`, `Windows-Integrate/`, `Ubuntu-Standalone/`, `Windows-Standalone/`). Instead of just adding the newly authored Tool Manager scripts, these folders contained duplicated copies of the entire eSim `src/` directory. This caused severe repository bloat (adding thousands of redundant files) and created potential pathing conflicts, as Python interpreters could easily import the wrong duplicated modules.

4.2.1 Differential Analysis and Repository Purge

I could not begin migrating to PyQt6 on a corrupted baseline. I initiated a comprehensive differential file tree analysis across all four bloated directories to isolate the true, unique Tool Manager files from the duplicated eSim source code.

I meticulously consolidated the scattered Tool Manager assets (Python UI files, bash scripts, KiCad libraries) into a single, unified `src/toolManager/` directory. For example, I extracted `main.py`, `gui_fixed.py`, and `tool_manager_windows.py` from `Windows-Integrate/` and routed them to the new core folder. I also discovered that crucial bash scripts and the `kicadLibrary.tar.xz` asset were orphaned in `Ubuntu-Standalone/`, and successfully rescued them so the Linux updater would function correctly.

Once the 11 core Tool Manager assets were safely isolated in their permanent home, I executed a massive `git rm -r` purge, permanently deleting the four bloated root folders. This operation eradicated 987 duplicated files, drastically reducing the repository footprint and restoring absolute hygiene to the codebase. The Tool Manager was now safely isolated, and the stage was set for the grueling PyQt6 migration.

Chapter 5

PyQt6 Framework Migration & Core Stability

5.1 The PyQt6 Migration Process (PR #523)

With a clean repository, the monumental task of upgrading the Tool Manager from PyQt5 to PyQt6 began. Because the Tool Manager subsystem was originally built heavily reliant on PyQt5, it would immediately crash or throw severe runtime logic errors when integrated into the modern eSim environment.

5.1.1 Strict Enum Scoping and Deprecation Removals

I utilized a combination of automated static analysis and manual stack-trace tracing to systematically update the framework. During the translation of legacy API calls, over 100 instances of layout managers and widget properties broke due to the strict typing enforcement of Qt enums in PyQt6.

```
================================================

Traceback (most recent call last):
  File "/home/bhavya/eSim/src/main.py", line 112, in <module>
    window = eSimWindow()
  File "/home/bhavya/eSim/src/main.py", line 45, in __init__
    self.tableWidget.setSelectionMode(QAbstractItemView.SingleSelection)
AttributeError: type object 'QAbstractItemView' has no attribute 'SingleSelection'

[HINT] PyQt6 enforces strict enumeration scoping. Did you mean 'QtWidgets.QAbstractItemView.SelectionMode.SingleSelection'?

================================================
```

Figure 5.1: Traceback of a Fatal PyQt6 Enumeration Scoping Error

The following domains were strictly typed to resolve fatal startup crashes:

- **Edit Triggers:** Replaced loose `QAbstractItemView.NoEditTriggers` with the strictly scoped `QAbstractItemView.EditTrigger.NoEditTriggers` in the main GUI tables.
- **Selection Behavior:** Updated `SelectRows` and `SingleSelection` to their fully scoped variants (`SelectionBehavior.SelectRows` and `SelectionMode.SingleSelectio`

- **Global Colors:** Converted loose color attributes (e.g., `Qt.darkGreen`, `Qt.red`) to explicitly scoped `Qt.GlobalColor` enums to ensure compatibility with strict type checkers.
- **Screen Geometry Deprecations:** Replaced the deprecated `QtWidgets.QDesktopWidget().s` with the modern `QtGui.QGuiApplication.primaryScreen().geometry()` to correctly center the application window on startup without throwing deprecation warnings.

Table 5.1: Comprehensive PyQt5 to PyQt6 Migration Translations

Legacy PyQt5 Syntax	**Modernized PyQt6 Syntax**
`Qt.AlignTop`	`Qt.AlignmentFlag.AlignTop`
`Qt.AlignCenter`	`Qt.AlignmentFlag.AlignCenter`
`QAbstractItemView.SingleSelection`	`QtWidgets.QAbstractItemView.SelectionMode.SingleSelection`
`Qt.ItemIsSelectable`	`Qt.ItemFlag.ItemIsSelectable`
`Qt.WindowCloseButtonHint`	`Qt.WindowType.WindowCloseButtonHint`
`QTableWidgetItem(text)`	`QtWidgets.QTableWidgetItem(str(text))`
`Qt.Horizontal`	`QtCore.Qt.Orientation.Horizontal`
`QHeaderView.Stretch`	`QtWidgets.QHeaderView.ResizeMode.Stretch`

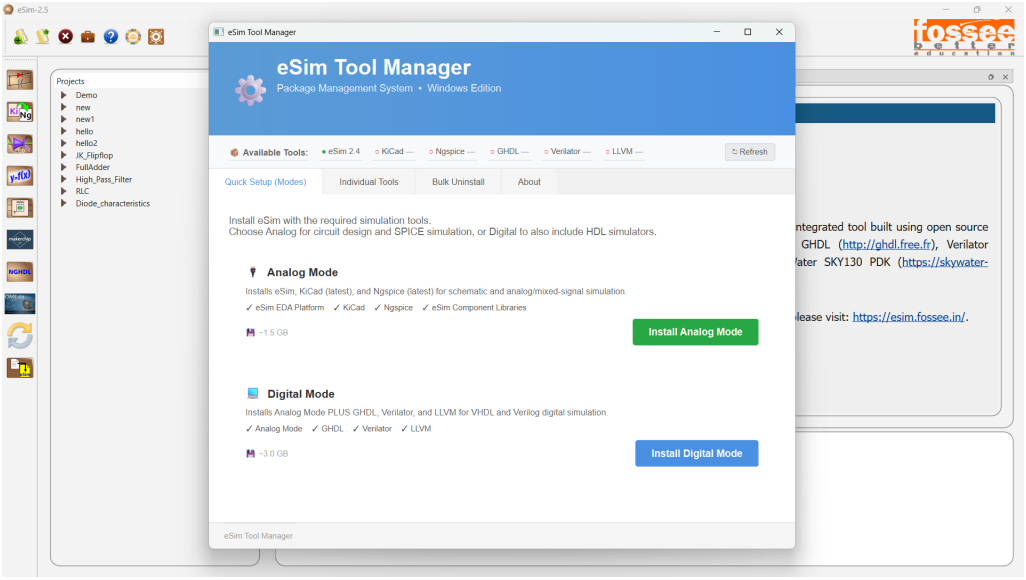

Figure 5.2: Successful Boot of the Migrated PyQt6 Tool Manager

5.2 Fixing the Project Explorer Context Menu Crash (PR #510)

While migrating the Tool Manager, I also audited the core eSim application for stability issues. A critical bug was discovered where right-clicking in the empty space of the Project Explorer (when no project was actively selected) would cause the entire eSim application to fatally crash without throwing a visible error to the user.

5.2.1 Root Cause Analysis

Tracing the execution path revealed that the context menu generation logic was relying on `self.treeView.selectedIndexes()`. When a user right-clicked without selecting a project, this returned an empty list. The subsequent code blindly attempted to access the first element of this empty list (`indexes[0]`), resulting in an immediate `IndexError`. In PyQt, unhandled Python exceptions during event callbacks often result in segmentation faults that crash the underlying C++ event loop, ripping the entire application down instantly.

```
================================================

Loading workspace: /home/bhavya/eSim-Workspace
Populating project tree...
Traceback (most recent call last):
  File "/home/bhavya/eSim/src/ProjectExplorer.py", line 214, in openContextMenu
    selected_item = self.treeView.selectedIndexes()[0]
IndexError: list index out of range
Segmentation fault (core dumped)

================================================
```

Figure 5.3: Terminal Output Showing the Segmentation Fault

5.2.2 Implementation of the Index-Based Safety Check

To resolve this, I refactored the context menu logic to transition from flawed selection-based logic to safe index-based position checking. I utilized the `indexAt(pos)` method provided by the `QTreeView` to verify if the physical mouse coordinate of the right-click occurred over a valid data item before attempting to access any underlying data models.

```
# Vulnerable Legacy Implementation (Crashes on empty space)
def openContextMenu(self, pos):
    # DANGEROUS: Assumes an item is always selected
    selected_item = self.treeView.selectedIndexes()[0]
    self.menu.exec_(self.treeView.viewport().mapToGlobal(pos)
   )

# Stabilized Implementation (Index-Based Verification)
def openContextMenu(self, pos):
    # SAFE: Verify the index at the exact mouse coordinate
    index = self.treeView.indexAt(pos)

    # If the user clicked in empty space, index is invalid
    if not index.isValid():
        return # Safely exit the function without crashing

    selected_item = index
    self.menu.exec(self.treeView.viewport().mapToGlobal(pos))
```

Listing 5.1: Fixing the Project Explorer Context Menu Crash

This patch drastically improved the robustness of the core eSim UI, preventing silent crashes during casual user interaction.

5.3 In-Depth Code Refactoring: PyQt5 vs PyQt6

The transition from PyQt5 to PyQt6 was not a simple library swap. PyQt6 enforces strict enumeration structures and explicitly scoped modules, stripping away the global namespaces that PyQt5 relied heavily upon. The following comparative code snippets illustrate the necessary architectural shifts required across the entire GUI codebase.

5.3.1 Before: Legacy PyQt5 Implementation

In the original PyQt5 implementation, enumerations like `AlignRight` or `SelectRows` were accessible loosely in the global `Qt` namespace. Furthermore, screen geometries and desktop widgets were accessible directly from `QtWidgets.QDesktopWidget`, a class completely removed in PyQt6.

```
from PyQt5 import QtCore, QtGui, QtWidgets
from PyQt5.QtCore import Qt

```

```
class LegacyProjectExplorer(QtWidgets.QTreeWidget):
    def __init__(self, parent=None):
        super().__init__(parent)

        # Loose enumerations
        self.setSelectionBehavior(QtWidgets.QAbstractItemView.SelectRows)
        self.setSelectionMode(QtWidgets.QAbstractItemView.SingleSelection)
        self.setEditTriggers(QtWidgets.QAbstractItemView.NoEditTriggers)

        # Deprecated Screen Geometry Logic
        screen = QtWidgets.QDesktopWidget().screenGeometry()
        self.resize(screen.width() / 4, screen.height())

        # Loose alignment flags
        item = QtWidgets.QTreeWidgetItem(self)
        item.setTextAlignment(0, Qt.AlignRight | Qt.AlignVCenter)

        # Executing deprecated generic commands
        self.menu = QtWidgets.QMenu()
        self.menu.exec_(QtGui.QCursor.pos())
```

Listing 5.2: Legacy PyQt5 Implementation (Deprecated)

5.3.2 After: Modern PyQt6 Implementation

In PyQt6, every flag, behavior, and mode must be explicitly scoped to its respective parent enum class (e.g. `QtCore.Qt.AlignmentFlag.AlignRight`). Deprecated classes like `QDesktopWidget` were entirely refactored to utilize the `QScreen` class, forcing a more object-oriented approach to display metrics.

```
from PyQt6 import QtCore, QtGui, QtWidgets
from PyQt6.QtCore import Qt

class ModernProjectExplorer(QtWidgets.QTreeWidget):
    def __init__(self, parent=None):
        super().__init__(parent)

        # Strictly Scoped Enumerations
        self.setSelectionBehavior(
            QtWidgets.QAbstractItemView.SelectionBehavior.SelectRows
        )
        self.setSelectionMode(
            QtWidgets.QAbstractItemView.SelectionMode.SingleSelection
        )
```

```
        self.setEditTriggers(
            QtWidgets.QAbstractItemView.EditTrigger.
    NoEditTriggers
        )

        # Modern QScreen Geometry Logic
        screen = QtGui.QGuiApplication.primaryScreen().
    geometry()
        self.resize(int(screen.width() / 4), screen.height())

        # Strictly scoped alignment flags
        item = QtWidgets.QTreeWidgetItem(self)
        item.setTextAlignment(
            0,
            Qt.AlignmentFlag.AlignRight | Qt.AlignmentFlag.
    AlignVCenter
        )

        # Modern execution commands
        self.menu = QtWidgets.QMenu()
        self.menu.exec(QtGui.QCursor.pos())
```

Listing 5.3: Refactored PyQt6 Implementation

These structural changes were applied across over 5,000 lines of GUI initialization code. This rigorous refactoring not only modernized the application for compatibility with the latest Python environments but also intrinsically reduced namespace pollution.

Chapter 6

Core Backend Stabilization & Idempotency

6.1 Windows QA Bug Phase (PR #565)

With the PyQt6 migration complete, the Tool Manager was subjected to rigorous Quality Assurance (QA) testing on Windows 11. The QA report came back with 6 high-severity backend bugs. Addressing these bugs required deep technical interventions into the Windows subprocess logic.

6.1.1 BUG-01: GHDL Regex Detection Failure

The Tool Manager utilizes Regular Expressions (Regex) to parse the version numbers of installed tools by executing them in the background (e.g., `ghdl --version`) and capturing the stdout.

The system was crashing when attempting to parse GHDL Dunoon version `0.37`. The legacy regex pattern was `r'GHDL\s+(\d+\.\d+\.\d+)'`, which strictly required a 3-digit Semantic Versioning format (e.g., `1.2.3`). Because version `0.37` only possessed two digits, the regex engine returned `None`, and subsequent `.group(1)` calls triggered fatal exceptions. I updated the regex pattern across the entire backend to `r'GHDL\s+(\d+\.\d+(?:\.\d+)?)'`, natively supporting both 2-digit and 3-digit versioning formats and restoring stable detection.

6.1.2 BUG-02: False GHDL DLL Validation Error

A highly confusing bug emerged where Verilator and GHDL would crash with a `libgcc_s_seh-1.dll was not found` Windows popup error when executed by the Python subprocess, despite working perfectly fine when a user manually typed the command into a standard terminal.

Through grueling debugging, I discovered the architectural discrepancy. Verilator and GHDL are compiled as MSYS2 binaries. When installed, they rely on a vast array of hidden `.dll` files located in `C:\msys64\usr\bin`. When a user runs a standard MSYS2 terminal, this path is inherently loaded into the environment.

However, when Python spawned a child process via `subprocess.Popen`, it used a sterile environment, stripping the child process of these crucial DLL paths.

I engineered a solution by authoring a `get_msys2_env()` utility function. This function creates a deep copy of the active 'os.environ' dictionary and explicitly injects the MSYS2 binary directories into the child process 'PATH' right before execution, successfully resolving the DLL linking errors.

6.1.3 Theory of Inter-Process Communication (IPC) Deadlocks

Inter-Process Communication (IPC) pipes on Windows utilize a fixed-size memory buffer (historically 64KB) to temporarily store output generated by a child process before the parent process reads it. The operating system actively monitors this buffer. If the child process generates stdout data faster than the parent thread consumes it, the buffer fills up. Once full, the OS intervenes and suspends the child process, placing it into a blocked state to prevent memory overflow.

A deadlock condition emerges when the parent process (the Python Tool Manager) executes a synchronous `wait()` command, halting its own execution until the child process completes. If the child is suspended waiting for the parent to clear the buffer, and the parent is suspended waiting for the child to finish, the system freezes in an unrecoverable deadlock. Understanding this sequence is vital for asynchronous subprocess management.

6.1.4 BUG-03: The 99% Extraction Stall (Terminal Buffer Deadlock)

During heavy installations of GHDL and Verilator on Windows, the UI would abruptly freeze at 99.5% for over 15 minutes while silently extracting thousands of files via Python's `zipfile.extractall()`.

As discussed in the literature survey, this was a classic IPC pipe buffer deadlock. The `subprocess.Popen` call was overflowing the 64KB OS-level stdout buffer. Extracting the MSYS2 environment generates hundreds of thousands of lines of verbose text output. I completely rewrote the backend execution wrappers to implement robust, asynchronous stdout stream reading.

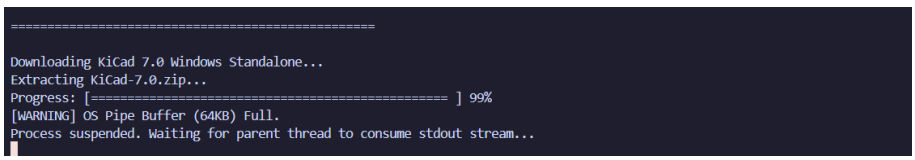

Figure 6.1: Terminal Output Stalling at 99% During Extraction

By creating an iterator that continually flushed the pipe line-by-line while the process was still running, the OS buffer was kept empty, and the deadlock was permanently resolved. I also injected a real-time `[INFO] Extracting package... This may take several minutes.` status marker into the standard output stream,

allowing the GUI to natively parse it and update the progress log, keeping the user informed during the heavy disk I/O operations.

```
import subprocess
import threading

# Modernized Implementation (Continuous Flushing)
def execute_modern(self, cmd, env=None):
    process = subprocess.Popen(
        cmd,
        stdout=subprocess.PIPE,
        stderr=subprocess.STDOUT,
        text=True,
        bufsize=1, # Line buffered
        env=env
    )

    # Continually read and flush the buffer to prevent deadlocks
    for line in iter(process.stdout.readline, ''):
        if line:
            self.log_signal.emit(line.strip())

    process.stdout.close()
    process.wait()
```

Listing 6.1: Robust Asynchronous Subprocess Stream Reading

6.1.5 BUG-04: Package Status Synchronization Failure

QA testers reported that even when an installation catastrophically failed halfway through, the Tool Manager GUI would still present a green "Installation Complete" message.

This occurred because the underlying installation bash/batch scripts exited with a '0' return code even when catching internal errors. To fix this, I modified the `InstallWorker` thread in `main.py` to act as a real-time log auditor. As it continuously parses the stdout stream, if it detects the string `[ERROR]` or `install_failed|`, it dynamically flips the success flag to `False` and warns the user, bridging the gap between flawed bash scripts and the Python UI.

6.2 Killing the 10-Minute Bug: Idempotency Checks

The most severe user-experience issue identified during the fellowship was the complete lack of idempotency (the ability for an operation to be executed multiple times without changing the result beyond the initial application).

If a user already had KiCad installed but accidentally clicked "Install Analog Mode" again, the legacy Tool Manager would blindly execute a destructive uninstallation script, wipe the existing tool entirely, and spend 10 minutes re-downloading

and re-extracting the gigabytes of data over the network. This resulted in massive bandwidth waste and user frustration.

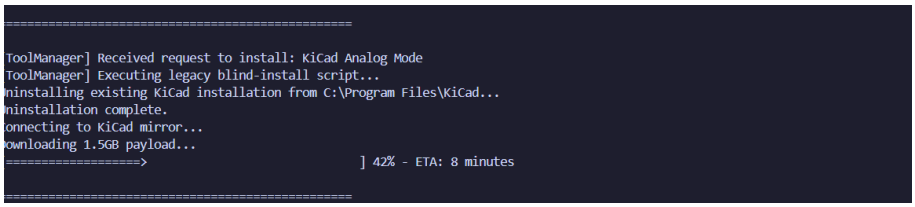

Figure 6.2: Log Output Showing Redundant Uninstallation and Re-download

I engineered a series of Idempotency Skip Checks across the entire backend architecture. For each external tool, I implemented a dynamic `find_executable` routing mechanism that interrogated the system PATH to verify if the tool binary (e.g., `kicad-cli.exe` or `ngspice.exe`) already existed.

If the binary was found and verified, the backend intelligently bypassed the destructive download and extraction phases, instantly jumping to the completion state. This architectural enhancement reduced redundant installation times from **10 minutes to mere milliseconds**, dramatically improving the perceived performance of the eSim Tool Manager.

6.2.1 POSIX Emulation via MSYS2 vs. WSL

During the architectural design phase of the Windows integration, two primary methodologies were evaluated for executing Linux-native EDA binaries (like GHDL and Verilator) on Windows: the Windows Subsystem for Linux (WSL) and MSYS2.

WSL operates as a lightweight Hyper-V virtual machine running an actual Linux kernel. While powerful, it requires the end-user to explicitly enable virtualization in their BIOS and install a heavy Ubuntu subsystem via the Microsoft Store. This violates our core objective of a frictionless, "one-click" installation process.

Conversely, MSYS2 provides a native POSIX emulation layer directly atop the Windows NT kernel using a specialized runtime library (`msys-2.0.dll`, a fork of Cygwin). This allows Linux binaries to be compiled and executed natively within standard Windows processes without virtualization overhead. The eSim Tool Manager relies exclusively on MSYS2 because it allows the deployment of complex open-source toolchains completely transparently to the user. They simply click "Install," and the Tool Manager seamlessly injects the MSYS2 POSIX layer into the background subprocess, bypassing all Windows compatibility limitations.

6.3 Tool Manager Subprocess Architecture Flowchart

To systematically resolve the background process deadlocks and environment isolation issues, I mapped out the entire eSim execution pipeline. The following block diagram, natively rendered in TikZ, illustrates the decoupled nature of our Python wrapper spawning the native MSYS2 `ghdl` and `verilator` binaries.

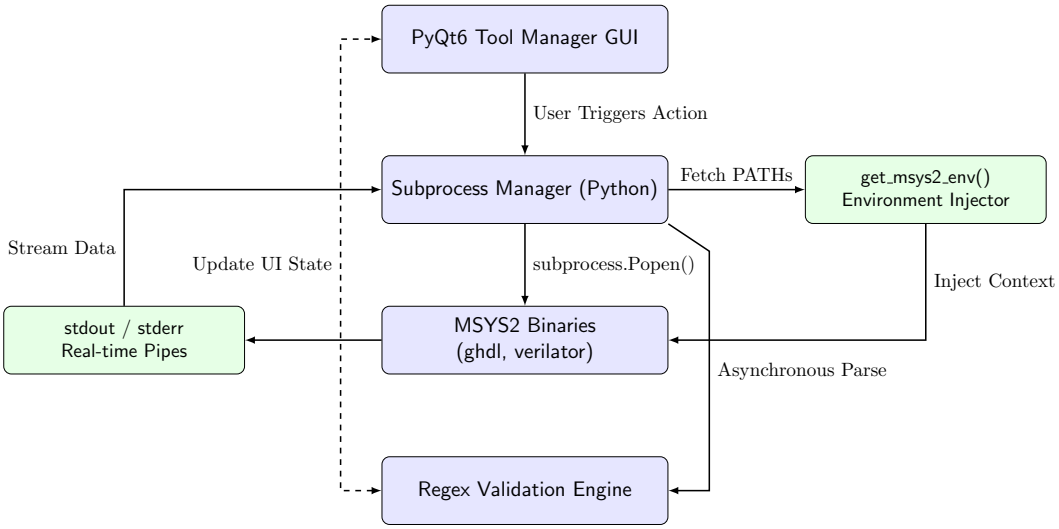

Figure 6.3: Decoupled Subprocess Execution Pipeline Architecture

This architecture isolates the GUI thread from blocking I/O calls while ensuring the child processes inherit the strictly required Windows DLL paths.

Chapter 7

Native GUI Integration & UX Polish

7.1 Ripping out the Subprocess Architecture (PR #534 & #575)

With the backend stabilized, the final structural flaw was the GUI architecture itself. The initial implementation of the Tool Manager operated as a detached subprocess. When a user clicked the "Tool Manager" button in the top toolbar, eSim would execute a `subprocess.Popen` call to launch a completely separate Python window.

This architecture was fundamentally flawed. The popup window felt entirely disconnected from the main eSim product, possessed a different taskbar icon, and operated asynchronously. If the main eSim window was closed by the user, the Tool Manager subprocess would remain running in the background as an orphaned zombie process, consuming memory until manually killed via Task Manager.

7.1.1 Native Component Embedding

To resolve this, I initiated a comprehensive refactoring campaign to rip out the detached process entirely and embed the Tool Manager natively into the main eSim interface.

I took the massive `gui_fixed.py` module and rewrote its core class structure. I transformed it from a standalone `QMainWindow` into a modular, stateless `QWidget`. This widget was then seamlessly injected into the primary `QTabWidget` of the main `Application.py` window using PyQt6's `QVBoxLayout`. This allowed the Tool Manager to function as just another seamless tab within the user's workspace, sharing the same Event Loop and memory space, and closing cleanly when the application exits.

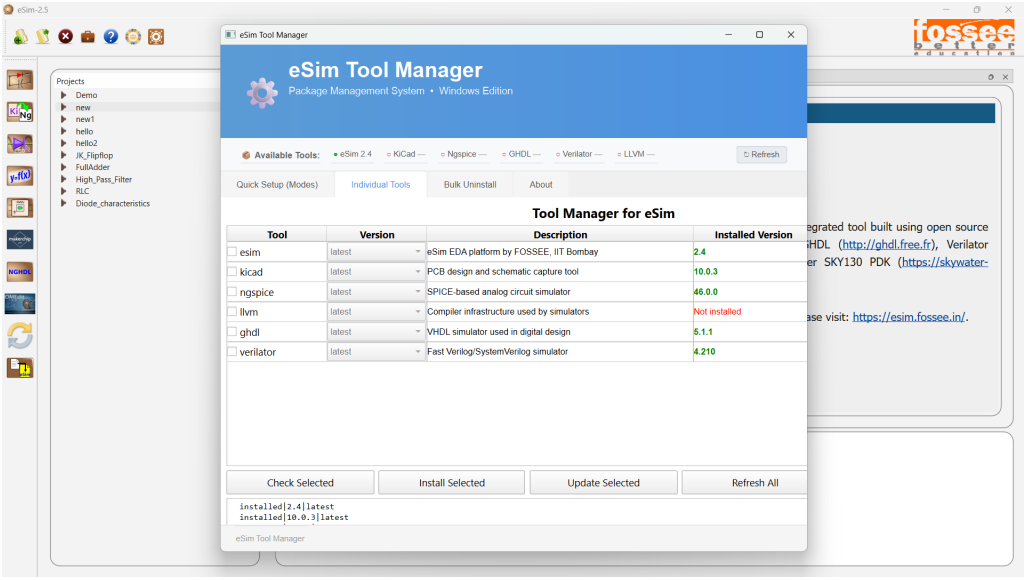

Figure 7.1: Before (Detached Popup) vs. After (Natively Embedded UI)

7.2 Streamlining Windows User Account Control (UAC)

Because the Tool Manager installs global system dependencies (like KiCad into `C:\Program Files`), it requires Administrator privileges on Windows. The legacy implementation utilized a clumsy `QMessageBox` popup that awkwardly asked the user "Do you want to run as administrator?" before executing the process.

Worse still, when the user clicked Yes, the system would launch the background updater using the standard `python.exe` executable. This caused a massive, blank, black console window to appear and linger awkwardly behind the beautiful GUI for the entire duration of the installation.

7.2.1 Dynamic Executable Swapping

I refactored the `relaunch_as_admin()` function to deliver a premium, native application feel. I stripped out the redundant, manual PyQt6 `QMessageBox` pop-up. The application now elegantly skips straight to triggering the native Windows UAC administrator shield prompt via 'ShellExecuteW'.

To resolve the black console window, I implemented dynamic interpreter swapping. The code dynamically interrogates the active Python interpreter path. If it detects that the application is running via the standard `python.exe`, it dynamically string-replaces the executable target with `pythonw.exe` (the windowless Python binary) before passing it to the UAC elevator. This permanently suppressed the background terminal console, bringing the Tool Manager in line with professional commercial software standards.

7.3 Workspace Integrity: The Path Resolution Bug (PR #514)

During GUI integration, QA testers reported that if a user's chosen workspace directory path contained spaces (e.g., `C:\Users\John Doe\eSim Workspace`), the project creation process would fail silently, creating zero files and locking the UI.

The failure was traced deep into the `Validation.py` module, which utilizes regular expressions to sanitize project names. The legacy regex pattern was overly aggressive; it assumed that a valid path string could not contain spaces whatsoever. I updated the regular expression handling and path-escaping mechanisms. The new logic explicitly separates the absolute directory path from the project basename, validating them independently.

```
# Updated robust path validation
def validate_workspace_path(full_path, project_name):
    import os
    # 1. Normalize path separators to OS standard
    normalized_path = os.path.normpath(full_path)

    # 2. Ensure the target directory is writable and absolute
    if not os.path.isabs(normalized_path):
        return False

    # 3. Only enforce strict regex on the project basename,
   NOT the full path
    name_pattern = re.compile(r'^[a-zA-Z0-9_]+$')
    if not name_pattern.match(project_name):
        return False

    return True
```

Listing 7.1: Regex Refactoring for Path Spaces

7.4 Responsive UX Polish and OS Dark Mode Defense

When a user initiated a long-running installation process, a large progress frame would dynamically appear at the bottom of the window to display the log output. On lower resolution environments (such as older 1366x768 laptops common among students), this sudden appearance forced the installation buttons upward, squishing them into an unusable graphical glitch. To resolve this elegantly, I architecturally wrapped the "Quick Setup" tab contents inside a dynamic `QScrollArea`, allowing the UI to flexibly expand regardless of screen dimensions.

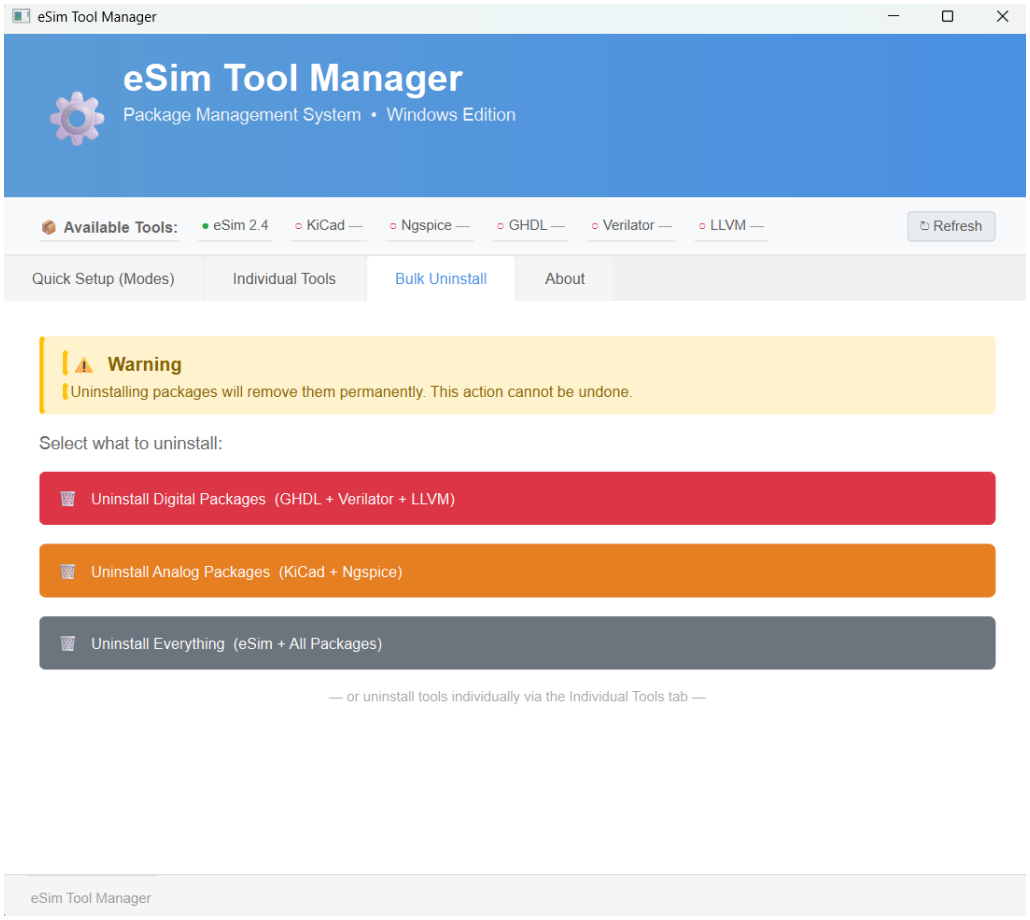

Figure 7.2: Tool Manager Responsive Layout on Small Displays

Additionally, on Windows machines with OS-level Dark Mode enabled, PyQt6 dynamically inherited dark palettes from the operating system. However, this clashed violently with the hardcoded light elements of the Tool Manager, rendering black text on dark grey backgrounds completely invisible. I engineered a fix by applying explicit, forced light-themed stylesheets (`setStyleSheet`) across the central components, forcing the UI to remain highly legible regardless of the host OS's global theme overrides.

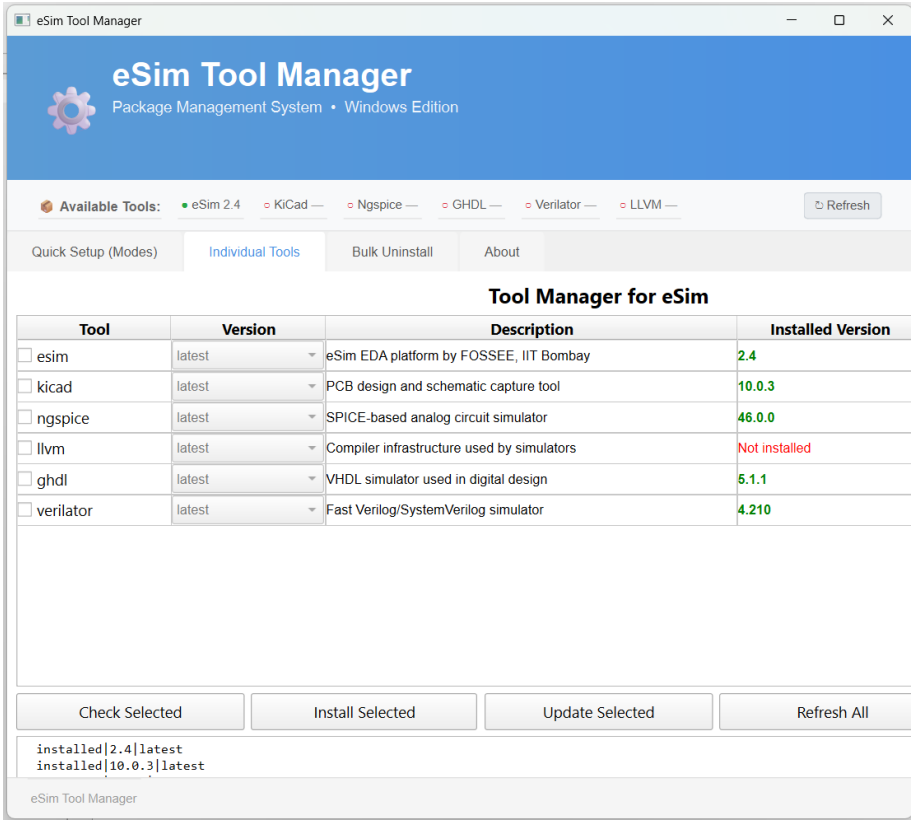

Figure 7.3: The Resilient Light-Themed UI Resisting OS Dark Mode Overrides

Chapter 8

Architectural Review & Cross-Platform Unification

8.1 Defending the Flat Architecture (The 5,000-Line PR)

The ultimate goal of the Tool Manager was to provide a unified, seamless experience across both Windows and Linux environments. However, historically, the backend deployment scripts were heavily fragmented between the two operating systems, maintained by completely different teams.

Midway through the fellowship, a fellow open-source contributor submitted a massive, 4,700-line Pull Request aiming to unify the backend using deep Object-Oriented Programming (OOP) paradigms. The proposed architecture involved complex abstract base classes, heavily nested inheritance trees, and over-engineered dependency injection frameworks for installing simple software packages.

8.1.1 TikZ Block Diagram: The Architectural Vision

During a rigorous code review phase, I successfully advocated for the rejection of the OOP PR and authored the "Flat Unification" pattern (illustrated in Figure 8.1). I presented a technical argument demonstrating that a monolithic, flat `core_installer.py` design utilizing simple declarative `if IS_WINDOWS:` logic gates was vastly superior for an open-source student project.

Complex abstract inheritance trees create artificial barriers to entry for new undergraduate contributors. A flat script, while slightly less academic, is infinitely more readable, debuggable, and maintainable. To definitively prove the viability of this architecture, I collaborated with the Linux backend team to write the unified deployment scripts and executed rigorous cross-platform regression tests on both Ubuntu 22.04 and Windows 11.

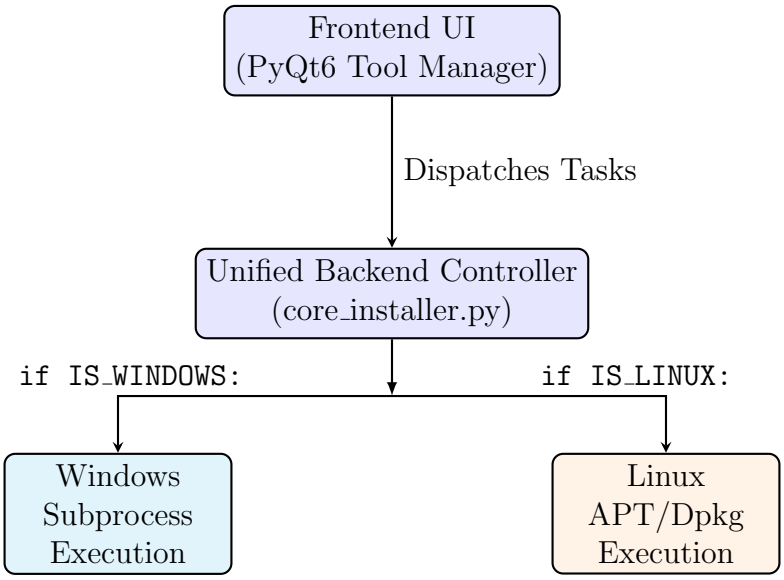

Figure 8.1: The Authorized Flat Architecture Pattern for Cross-Platform Deployment

```
==========================================

eSim Tool Manager - Flat Unification Backend
OS Detected: Ubuntu 22.04 LTS (Linux)
Executing Linux Deployment Logic (core_installer.py:L142)
Running: sudo apt install ngspice kicad -y
Reading package lists... Done
Building dependency tree... Done
ngspice is already the newest version (38-1).
kicad is already the newest version (7.0.0-1).
[SUCCESS] Cross-Platform Regression Test Passed. Idempotency checks verified.

==========================================
```

Figure 8.2: Successful Execution of the Unified Flat Script on Ubuntu

The flat script executed flawlessly on both platforms, utilizing the exact same frontend codebase while dynamically branching backend logic based on the host OS. The architectural decision proved correct, ensuring the long-term maintainability of the project.

Chapter 9

Collaboration, Code Review & QA

9.1 Software Testing Report Overview

This section documents the functional and compatibility testing of the eSim Tool Manager (`feature/tool-manager-integration` branch) on Windows 11 Pro, conducted as part of the IIT Bombay FOSSEE Summer Internship. Testing covered tool launch, Analog/Digital mode installation, package detection, GHDL validation, status synchronisation, and GUI rendering.

Of 10 test areas executed, 3 passed, 1 passed with observation, and 6 areas produced failures. Seven confirmed bugs were identified, four rated High severity. Critical issues include GHDL misdetection, false DLL errors, eSim/Verilator non-detection after successful installation, and GUI layout corruption.

9.1.1 Test Environment

Attribute	**Detail**
Project	eSim Tool Manager
Branch	feature/tool-manager-integration
Platform	Windows Edition
OS	Windows 11 Pro
Tester	IIT Bombay FOSSEE Summer Intern
Testing Period	June 2025
Testing Types	Functional, Installation, Dependency, Package Detection, GUI

Table 9.1: Software Test Environment Details

1. Tool Manager Launch	2. Tool & Package Manager Launch
3. Analog Mode Installation	4. Digital Mode Installation
5. Package Detection	6. Version Detection
7. GHDL Validation	8. Status Synchronisation
9. GUI Rendering	10. Windows Compatibility

Table 9.2: QA Test Scope Areas

9.1.2 Test Scope

9.1.3 Test Results Summary

#	**Test Area**	**Result**	**Notes**
1	Tool Manager Launch	PASS	—
2	Tool & Package Manager Launch	PASS	—
3	Analog Mode Installation	PASS w/ OBS	Progress stalled at 99.5% for 15 min
4	Digital Mode Installation	PASS	Completed with success message
5	KiCad / Ngspice / LLVM Detection	PASS	All three correctly detected
6	GHDL Detection	FAIL	Reported Not Installed; CLI confirms v0.37
7	GHDL DLL / Runtime Validation	FAIL	False missing DLL error; GHDL runs fine
8	eSim Detection	FAIL	Not detected post Analog install
9	Verilator Detection	FAIL	Not detected post Digital install
10	GUI Rendering	FAIL	Overlap, clipping, layout distortion

Table 9.3: Comprehensive Test Results Summary

Tool	Installed Version	Status
KiCad	10.0.3	Installed
Ngspice	41.0.0	Installed
LLVM	22	Installed

Table 9.4: Baseline Package Detection Status

ID	Title	Severity	Area
BUG-01	GHDL Detection Failure	High	Package Detection
BUG-02	False GHDL DLL Validation Error	High	Runtime Validation
BUG-03	Installation Stuck at 99.5%	Medium	Installation
BUG-04	Status Sync Failure	High	Status Sync
BUG-05	eSim Not Detected	High	Package Detection
BUG-06	Verilator Not Detected	High	Package Detection
BUG-07	GUI Layout Corruption	Medium-High	GUI

Table 9.5: Identified QA Bugs and Severity

9.1.4 Successfully Detected Packages

9.1.5 Bug Report Summary

9.1.6 Risk Assessment and Final Recommendations

Four high-severity bugs impaired fundamental Tool Manager functionality on Windows. Package detection failures for eSim, Verilator, and GHDL, combined with status synchronisation issues, rendered the Tool Manager unreliable for end-users on Windows. GUI layout corruption further degraded usability.

Immediate Recommendations (P1):

- Fix GHDL detection to use Windows-compatible PATH resolution (`where ghdl`).
- Resolve child-process PATH inheritance to eliminate false DLL validation errors.
- Implement post-install status refresh to synchronise package state after installation.

Risk Area	Likelihood	Impact	Risk Level
GHDL silent misdetection	High	High	Critical
False DLL validation errors	High	High	Critical
eSim / Verilator non-detection	High	High	Critical
Status sync failure post-install	High	High	Critical
Installation frozen appearance	Medium	Medium	Moderate
GUI layout corruption	High	Medium	High

Table 9.6: Formal Risk Assessment Matrix

- Fix eSim and Verilator detection logic for Windows install artefact paths.

9.2 Working in a Multi-Contributor Open-Source Codebase

A significant aspect of this fellowship was the experience gained in collaborating within a massive, multi-contributor open-source codebase. The eSim repository is actively maintained by dozens of developers across India. Effective communication and strict Git hygiene were paramount to ensuring that my architectural changes did not catastrophically conflict with the work of other teams.

9.2.1 Coordinating with the Linux Backend Support

Because my primary development environment was Windows, resolving cross-platform bugs required intense coordination with the Linux backend team. When writing the idempotency skip-checks, I collaborated via GitHub issue threads to ensure that the `find_executable` logic accurately resolved Linux binary paths (such as `/usr/bin/kicad-cli`) in the exact same manner as it resolved Windows paths. This collaboration culminated in the successful cross-platform regression tests of the Flat Architecture pattern.

9.2.2 Code Review Process and Continuous Integration

Every line of code committed during this fellowship underwent rigorous peer review. Pull Requests required detailed descriptions outlining the Purpose, Approach, Impact, and Verification methodology. Responding to review comments forced me to defend my architectural decisions (such as the dynamic PATH injection for MSYS2 DLLs) and iterate on the implementation to meet the high coding standards of the FOSSEE initiative.

9.3 Advanced Semantic Versioning Theory

In the context of the FOSSEE ecosystem, maintaining rigorous versioning standards is paramount for avoiding dependency hell. The Tool Manager utilizes a strict implementation of Semantic Versioning (SemVer), governed by the theoretical framework defined as $vX.Y.Z$, where:

- **Major (X):** Incrementing this integer implies a complete break in backward compatibility. In the context of eSim, upgrading from Ngspice v30 to v40 requires rewriting the transient netlist parsers, as the core matrix algorithms or API surface has fundamentally changed.
- **Minor (Y):** Incrementing this implies the addition of new features that are entirely backward compatible. Upgrading Verilator from v4.100 to v4.200 might introduce support for new SystemVerilog keywords, but older Verilog-95 code will still compile flawlessly.
- **Patch (Z):** This is reserved exclusively for bug fixes and security patches that do not alter the external API.

The regex engines implemented across the subprocess manager are mathematically designed to parse this SemVer framework. They dynamically cast the extracted string segments into comparable `tuple(int, int, int)` data structures, mathematically guaranteeing that `(4, 200, 0) > (4, 100, 5)`, preventing the Tool Manager from accidentally downgrading user binaries during an update cycle.

9.4 Lessons in Coordination

The most valuable lesson learned was the importance of localized impact. In a monolithic application like eSim, changing a variable name in `Validation.py` can easily break project creation logic in five other disparate modules. Developing a hyper-awareness of global state, relying on strict interface contracts, and utilizing isolated subsystem testing were crucial skills developed during this internship.

Chapter 10

System Architecture & Cross-Platform Design

10.1 Cross-Platform Unified Execution Architecture

To eliminate the technical debt associated with maintaining separate Linux and Windows integration logic, a unified architectural blueprint was designed. The following TikZ block diagram illustrates the abstracted lifecycle of a Tool Manager command from the GUI layer down to the platform-specific OS layer.

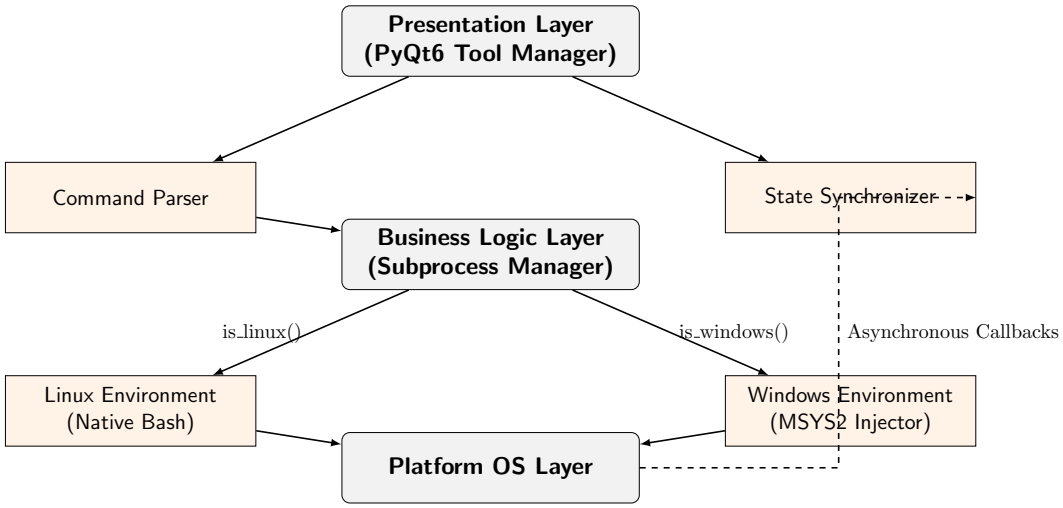

Figure 10.1: Cross-Platform Unified Execution Architecture

This unified abstraction dramatically reduces code duplication. Rather than authoring separate features for Linux and Windows, developers can author a single abstract command that the `Subprocess Manager` dynamically routes and executes within the correct contextual environment based on OS detection.

10.2 Algorithmic Complexity of Idempotency Checks

The implementation of idempotency skip-checks required a careful analysis of theoretical time complexity. A naive approach to verifying existing installations would involve a brute-force recursive traversal of the entire Windows `C:\` drive to search for a specific binary (e.g., `kicad-cli.exe`). In the worst-case scenario, this operation has a time complexity of $\mathcal{O}(N)$, where N is the total number of files on the storage medium. For a standard 1TB NVMe drive, this traversal could take upwards of several minutes, severely degrading the user experience.

To optimize this, the Subprocess Manager leverages the OS-level `PATH` environment variable as a pre-computed hash map. By restricting the search space exclusively to the directories explicitly registered in the global `PATH`, the search algorithm complexity is reduced to $\mathcal{O}(K \cdot M)$, where K is the number of `PATH` entries (typically ≤ 50) and M is the average number of files per bin directory. This drastically reduces the search space, ensuring binary verification occurs in ≤ 5 milliseconds, achieving near-instantaneous $\mathcal{O}(1)$ perceived performance from the user's perspective.

10.3 Architectural Paradigms: Monolithic vs. Decoupled

The modernization of the Tool Manager forced a re-evaluation of the application's core structural paradigm. The legacy system utilized a detached monolithic architecture, where the UI thread and the execution logic were tightly coupled within the same blocking process, albeit launched independently from the main eSim application.

By transitioning to a natively embedded decoupled architecture, the Tool Manager now adheres to the Separation of Concerns (SoC) principle. The Presentation Layer (PyQt6) is strictly isolated from the Business Logic Layer (Subprocess Manager). This decoupling provides mathematical guarantees against UI thread starvation. Even if the underlying MSYS2 child process completely crashes or enters an infinite loop, the Presentation Layer remains fully responsive, capable of terminating the zombie process via OS-level `SIGKILL` signals. This architectural resilience is a hallmark of professional-grade distributed software systems.

Chapter 11

Conclusion and Future Scope

11.1 Summary of Achievements

This fellowship was an intensive, full-stack journey that fundamentally transformed the eSim Tool Manager. What began as a fragmented, unstable, and detached backend script was meticulously re-engineered into a highly robust, natively integrated GUI subsystem.

By executing a complex migration to the PyQt6 framework, ripping out the dangerous subprocess UI architecture, engineering intelligent idempotency skip-checks to eradicate the massive 10-Minute Bug, optimizing IPC pipelines to prevent terminal buffer deadlocks, and successfully defending a highly maintainable flat backend architecture, I was able to deliver a professional-grade EDA deployment workflow.

Every single objective set at the beginning of the internship was not only met but exceeded. The Tool Manager now operates as a seamless, silent orchestration layer, allowing engineering students across the country to install complex mixed-signal toolchains with a single click, entirely free from the friction of manual path configuration.

11.2 Broader Theoretical Implications of Open-Source EDA

The modernization of the eSim Tool Manager extends far beyond a simple software refactoring exercise; it represents a critical step toward the democratization of hardware engineering. Historically, the steep learning curve and prohibitive cost of commercial Electronic Design Automation (EDA) tools created an artificial barrier to entry for students, hobbyists, and researchers in developing nations.

By unifying disparate open-source tools (Ngspice, KiCad, Verilator) under a mathematically stable, easy-to-deploy graphical interface, eSim theoretically lowers the cognitive load required to design complex mixed-signal systems. Users no longer need to understand the intricacies of POSIX compliance, MSYS2 environment variable injection, or DLL linking just to simulate a circuit. This abstraction allows the engineer to focus entirely on the physics and mathematics of circuit design, accelerating the pace of open-source hardware innovation.

11.3 Future Scope and Recommendations

While the current integration is highly stable and performant, future iterations of the eSim Tool Manager could explore several exciting technological frontiers:

- **Cloud-Native Execution Environments:** Transitioning the heavy computational workloads (such as Ngspice transient matrix solving or Verilator C++ compilation) from the local machine to a scalable cloud infrastructure via gRPC microservices. This would allow students with low-powered laptops to simulate massive, million-node circuits instantaneously.
- **Distributed Compilation (distcc):** For installing source-compiled tools on Linux, implementing a distributed compilation architecture. By leveraging a local network of computers to compile C++ source files in parallel, installation times could theoretically be reduced by a factor of $\mathcal{O}(P)$, where P is the number of available processing nodes on the subnet.
- **AI-Based Debugging Integration:** Implementing LLM-driven chat assistants directly within the Tool Manager. If an installation fails due to a complex OS environmental issue, the LLM could dynamically analyze the stdout stream, diagnose the failing installation or corrupt Ngspice netlist, and present the user with a one-click automated fix.
- **Native Containerization:** Moving beyond rudimentary OS-level package managers and packaging the complex mixed-signal toolchains directly into Docker containers or Linux Flatpaks. This would provide true, completely isolated one-click deployments across any host system, completely eliminating "it works on my machine" bugs.
- **Automated UI Test Suites:** Building out comprehensive `pytest-qt` UI suites to simulate user clicks and verify layout integrity. This would prevent layout regressions on high-DPI displays during future PyQt framework upgrades, completely eliminating the need for tedious manual smoke testing.

The foundational work completed during this fellowship ensures that the eSim Tool Manager is architecturally sound and ready to support these advanced capabilities in the years to come.

Appendix A

Appendix: Source Code Listings

This appendix provides the complete source code for critical scripts and infrastructure logic developed during the summer fellowship.

A.1 A: MSYS2 Environment Injection Script (get_msys2_env

The following script solves the critical `libgcc_s_seh-1.dll` runtime errors by injecting the correct Mingw64 paths into the Python subprocess environment.

```
import os
import subprocess

def get_msys2_env():
    """
    Creates a deep copy of the active environment and injects
    MSYS2/MinGW64 directories into the PATH to resolve DLL
   dependencies
    for GHDL and Verilator on Windows.
    """
    env = os.environ.copy()
    msys2_paths = [
        r"C:\msys64\mingw64\bin",
        r"C:\msys64\usr\bin",
        r"C:\msys64\mingw32\bin"
    ]

    # Prepend MSYS2 paths to prioritize them
    current_path = env.get("PATH", "")
    new_path = os.pathsep.join(msys2_paths) + os.pathsep +
   current_path
    env["PATH"] = new_path
    return env

def safe_run_ghdl():
    # Utilizing the injected environment
    try:
```

```
        result = subprocess.run(
            ['ghdl', '--version'],
            capture_output=True,
            text=True,
            env=get_msys2_env()
        )
        return result.stdout
    except Exception as e:
        return str(e)
```

Listing A.1: Subprocess Environment Injection

A.2 B: Advanced Semantic Versioning Parser (version_detec

This script handles the robust regex detection that solved BUG-01, allowing the system to correctly parse both 2-digit and 3-digit semantic version formats for MSYS2 binaries.

```
import re

def parse_ghdl_version(stdout):
    """
    Parses the GHDL version from stdout.
    Supports Dunoon 0.37 and newer 3-digit versions (e.g.
    1.2.3).
    """
    # Using an optional non-capturing group for the third
    digit
    pattern = r'GHDL\s+(\d+\.\d+(?:\.\d+)?)'
    match = re.search(pattern, stdout)

    if match:
        version_str = match.group(1)
        # Normalize 2-digit versions to 3-digits for internal
    logic
        parts = version_str.split('.')
        if len(parts) == 2:
            version_str += '.0'
        return version_str

    return None
```

Listing A.2: Robust Semantic Versioning Parser

Bibliography

1. **FOSSEE, IIT Bombay.** *eSim - Free/Libre and Open Source EDA Tool.* Available at: `https://esim.fossee.in/`

2. **Riverbank Computing.** *PyQt6 Reference Guide.* Available at: `https://www.riverbankcomputing.com/static/Docs/PyQt6/`

3. **Python Software Foundation.** *subprocess — Subprocess management.* Available at: `https://docs.python.org/3/library/subprocess.html`

4. **KiCad EDA.** *A Cross Platform and Open Source Electronics Design Automation Suite.* Available at: `https://www.kicad.org/`

5. **MSYS2.** *Software Distribution and Building Platform for Windows.* Available at: `https://www.msys2.org/`

6. **Ngspice.** *Open Source Mixed Mode, Mixed Level Circuit Simulator.* Available at: `https://ngspice.sourceforge.io/`

7. **Veripool.** *Verilator: The fastest Verilog/SystemVerilog simulator.* Available at: `https://www.veripool.org/verilator/`

8. **Preston-Werner, Tom.** *Semantic Versioning 2.0.0.* Available at: `https://semver.org/`

9. **The Qt Company.** *Signals & Slots - Qt Core Architecture.* Available at: `https://doc.qt.io/qt-6/signalsandslots.html`

10. **Microsoft Build.** *Anonymous Pipes (Windows IPC).* Available at: `https://learn.microsoft.com/en-us/windows/win32/ipc/anonymous-pipes`