eSim Summer Fellowship 2026

On

Backend Architecture of eSim Tool Manager

Submitted by

George S Thuruthippillil

Department of Computer Science and Engineering
College of Engineering, Trivandrum

Under the guidance of

Prof. Prabhu Ramachandran
Principal Investigator

Department of Aerospace Engineering
Indian Institute of Technology Bombay

July 2026

Acknowledgements

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 backend infrastructure for eSim, and gain exposure to real-world software architecture, cross-platform development, and tool management 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.

Abstract

This report documents the work carried out during a summer semester-long internship with the FOSSEE team at IIT Bombay, focused on the architectural redesign and implementation of the backend infrastructure for the eSim Tool Manager. eSim is an open-source EDA tool for circuit design, simulation, analysis, and PCB design, developed by the FOSSEE team. The Tool Manager subsystem is responsible for installing, updating, verifying, and removing external dependencies such as KiCad, Ngspice, GHDL, Verilator, LLVM, and Chocolatey across Windows and Linux platforms.

The existing implementation consisted of a monolithic 1,434-line script with deeply coupled Windows-specific logic, duplicated tool metadata across four separate files, and Linux support was fragmented across several standalone shell scripts with duplicated, stale data, deeply coupled to specific distributions with no abstraction and weak concurrency support.

The internship encompassed a complete architectural overhaul of the backend. Key contributions include: a platform abstraction layer providing unified OS detection, subprocess management, and administrator privilege elevation; an abstract base class (ABC) based backend architecture with separate Windows and Linux implementations; a centralized tool metadata registry consolidating all tool definitions, download URLs, install scripts, and search paths into a single source of truth; a reusable worker thread framework for asynchronous installation and command execution; and a Linux package manager abstraction supporting apt, dnf, pacman, zypper, and other package managers.

The refactoring produced a clean, modular architecture with a consolidated tool function registry and a unified backend abstraction, while simultaneously adding full Linux backend support and improving code maintainability, testability, and cross-platform consistency. The work was carried out from May 2026 to July 2026.

Contents

Chapter 1

Introduction

1.1 National Mission on Education through ICT (NMEICT)

The National Mission on Education through ICT (NMEICT) is a flagship scheme under the Department of Higher Education, Ministry of Education, Government of India. Its primary objective is to leverage the transformative potential of Information and Communication Technologies to enhance the quality, reach, and equity of higher education across India. The mission operates on the three cardinal principles of access, equity, and quality, seeking to empower learners and educators in both urban and rural contexts.

1.2 The FOSSEE Project

The FOSSEE (Free/Libre and Open Source Software for Education) project is a key initiative under NMEICT, headquartered at IIT Bombay. Its mission is to promote the adoption of Free/Libre and Open Source Software tools across academic and research institutions in India, reducing dependence on expensive proprietary software. FOSSEE supports a wide range of activities including Textbook Companions, Lab Migrations, and domain-specific software development. It offers semester-long, summer, and winter internship programmes to students across disciplines.

1.3 Overview of eSim

eSim is an open-source EDA (Electronic Design Automation) tool developed by the FOSSEE team at IIT Bombay. It provides an integrated platform for circuit design, simulation, analysis, and PCB layout design. eSim is built using a suite of open-source software components including KiCad for schematic capture and PCB design, Ngspice for analog circuit simulation, and GHDL for VHDL-based digital simulation. The tool supports mixed-signal simulations, microcontroller co-simulation, and subcircuit modelling, making it a versatile platform for electronics education and research.

1.4 The eSim Tool Manager

The eSim Tool Manager is a subsystem responsible for managing the lifecycle of external software dependencies required by eSim. These dependencies include core EDA tools such as KiCad, Ngspice, GHDL, Verilator, and LLVM, as well as system-level package managers like Chocolatey on Windows. The Tool Manager provides a graphical interface for users to install, update, verify, and remove these tools, and a backend that handles the platform-specific operations of downloading, extracting, and configuring each component.

The existing implementation of the Tool Manager backend consisted of a single monolithic Python script—`tool_manager_windows.py`—spanning over 1,400 lines. This script contained deeply coupled Windows-specific logic, hardcoded download URLs, duplicated tool metadata scattered across four separate files, and Linux support was fragmented across several standalone shell scripts carrying duplicated and outdated data, deeply coupled to specific distributions with no abstraction and weak concurrency support. These limitations made the code difficult to maintain, extend, or test.

1.5 Internship Objectives

The primary objectives of this internship were:

1. To redesign the monolithic Tool Manager backend into a modular, extensible architecture using object-oriented design principles.

2. To introduce an abstract backend interface supporting both Windows and Linux platforms.

3. To consolidate all tool metadata—versions, download URLs, install scripts, and search paths—into a single, maintainable registry.

4. To implement a reusable worker thread framework for asynchronous tool operations.

5. To provide a Linux backend with package manager abstraction supporting apt, dnf, pacman, zypper, and other package managers.

6. To reduce code duplication and overall codebase size while improving maintainability.

1.6 Report Organization

This report is organized as follows. Chapter 2 describes the screening task and project onboarding. Chapter 3 details the system architecture and technology stack. Chapter 4 presents the backend implementation in detail, organized by architectural layer. Chapter 5 discusses the ongoing frontend integration work. Chapter 6 describes key technical challenges and their solutions. Chapter 7 presents the development workflow and timeline. Chapter 8 concludes with a summary of contributions and future scope.

Chapter 2

Screening Task and Project Onboarding

2.1 Screening Task

Prior to being accepted for the semester-long internship, I was required to complete a screening task that demonstrated my proficiency with tool management concepts and cross-platform development. The task involved building a tool manager that could pull resources from remote repositories such as Git and SourceForge, using command-line utilities including `wget` and `curl` for file downloads. The solution was expected to handle version checking, file extraction, and basic dependency resolution. Successful completion of this screening task led to the offer for the eSim Tool Manager internship.

2.2 Initial Repository Setup and Environment Configuration

Following acceptance, the initial phase of the internship was spent on onboarding and environment setup. The eSim repository was cloned and its project structure was studied. The Tool Manager subsystem resides in `src/toolManager/` within the eSim codebase. The existing codebase was reviewed thoroughly, with particular attention to:

- `tool_manager_windows.py` — the 1,434-line monolithic backend script

- `updater_gui.py` — the graphical update interface
- `gui_fixed.py` — the main GUI implementation
- `main.py` — the application entry point
- Various bash shell scripts for Linux package installation

The development environment was configured with Python 3.10+, PyQt6, and the eSim application on a Linux workstation. Git was used for version control, with all changes tracked against the `feature/tool-manager-integration` branch.

Chapter 3

System Architecture and Technology Stack

3.1 Limitations of the Existing Monolithic Design

The original Tool Manager backend in `tool_manager_windows.py` suffered from several architectural shortcomings:

- **Single-file monolith:** All backend logic—package installation, executable search, file extraction, version checking—was contained in a single 1,434-line file with no clear separation of concerns.
- **Windows-only:** The implementation was tightly coupled to Windows-specific APIs, paths, and conventions (MSYS2, Chocolatey, Windows Registry), with no abstraction for other platforms.
- **Duplicated metadata:** Tool labels, versions, download URLs, and search paths were redundantly defined across `main.py`, `gui_fixed.py`, `updater_gui.py`, and `tool_manager_windows.py`. Any change to a tool's version or URL required updates in multiple locations.
- **No threading abstraction:** Long-running operations directly invoked subprocess calls without a standardised worker pattern, making the UI unresponsive during installations.
- **Fragmented Linux support:** Linux installation logic was scattered across several standalone shell scripts carrying duplicated and often stale data. These scripts were deeply coupled to specific distributions with no abstraction layer,

and had weak concurrency support—concurrent invocations of package managers would race for lock files and fail unpredictably.

3.2 Proposed Modular Architecture

To address these limitations, a layered modular architecture was proposed, consisting of the following components:

- **Platform Abstraction Layer (`pm_platform.py`):** A single module providing unified OS detection, subprocess execution helpers, and administrator privilege elevation, encapsulating all platform-specific logic.
- **Abstract Backend Interface (`backends/base.py`):** An abstract base class defining the contract for all backends—methods for finding executables, installing packages, downloading files, extracting archives, and running installers.
- **Platform Backends (`backends/windows/`, `backends/linux/`):** Concrete implementations of the backend interface for Windows (using Chocolatey, registry scanning, MSYS2) and Linux (using package managers, bash scripts).
- **Tool Function Registry (`backends/tools/tools.py`):** A unified dispatch table mapping tool IDs to their check, install, and uninstall functions, with OS-conditional branches.
- **Tool Metadata Registry (`registry.py`):** A centralized dataclass-based registry holding all tool specifications—labels, versions, download URLs, install scripts, search paths.
- **Worker Thread Framework (`worker.py`):** A set of QThread-based worker classes for asynchronous installation, command execution, and script running.

This architecture enables clean separation of concerns, platform-independent tool logic, and straightforward addition of new tools or platform backends.

3.3 Technology Stack

The following technologies and tools were used during the internship:

- **Python 3.10+** — Primary programming language for backend development

- **PyQt6** — GUI framework for the Tool Manager interface and worker threads
- **Chocolatey** — Package manager for Windows dependency management
- **apt, dnf, pacman, zypper** — Linux package managers supported through the abstraction layer
- **7-Zip, tar** — Archive extraction utilities
- **Git** — Version control and collaboration
- **pkexec, sudo** — Linux privilege elevation mechanisms

Chapter 4

Backend Implementation

This chapter describes the implementation of each architectural layer in the redesigned Tool Manager backend.

4.1 Platform Abstraction Layer

The `pm_platform.py` module serves as the foundation of the cross-platform backend, providing:

- **OS Detection:** Boolean flags (`IS_WINDOWS`, `IS_LINUX`, `IS_MAC`) and a `get_os_id()` function derived from `sys.platform`, used throughout the codebase for conditional branching without repeating platform checks.
- **Subprocess Helpers:** Platform-aware subprocess execution with appropriate flags—Windows uses `STARTF_USESHOWWINDOW` and `CREATE_NO_WINDOW` to suppress console windows, while Linux uses default subprocess settings.
- **Distribution Detection:** `distro_label()` reads `/etc/os-release` (or uses `freedesktop_os_release()` where available) to identify the Linux distribution and version, enabling distribution-specific package manager selection.
- **Package Manager Detection:** `detect_package_manager()` scans for available package managers (`apt-get`, `dnf`, `pacman`, `zypper`, `apk`) using `shutil.which()`.
- **Admin Privilege Utilities:** On Windows, `is_admin()` checks if the process is elevated and `relaunch_as_admin()` re-invokes the script with administrator privileges via `ShellExecute`. On Linux, admin privileges are handled through `sudo`.

- **Streaming Subprocess:** `run_cmd_stream()` provides real-time output streaming with timeout support, used by the installer thread to display progress.

4.2 Backend Abstraction and Cross-Platform Backends

4.2.1 Abstract Base Class

The `backends/base.py` module defines `Backend` as an abstract base class (ABC) that establishes a consistent interface for all platform-specific operations:

- `run_cmd()` — Execute a command and capture output
- `run_stream()` — Execute a command with real-time output streaming
- `which()` — Locate an executable in the system PATH
- `find_executable()` — Locate a tool using platform-specific search strategies
- `find_executable_with_version()` — Locate a tool and determine its version
- `install_package()` — Install a package using the platform package manager
- `uninstall_package()` — Remove a package
- `download_file()` — Download a file from a URL with progress reporting
- `extract_zip()` / `extract_7z()` — Extract compressed archives
- `run_installer()` — Execute a platform installer
- `print_status()` — Output a structured status string for GUI consumption

4.2.2 Windows Backend

The `backends/windows/` package provides a full Windows implementation:

- `backend.py`: `WindowsBackend` class implementing all ABC methods. Handles MSYS2 path resolution, Chocolatey integration, and Windows-specific process creation flags. Maintains download cache directory and manages UTF-8 output encoding for console compatibility.

- `search.py`: Executable discovery for each tool on Windows. Uses multiple strategies: well-known installation paths (e.g., `C:\ProgramFiles\KiCad`), Chocolatey package listing, Windows Registry scanning via `winreg`, and PATH lookup. Each tool has a dedicated search function with fallback chains.
- `choco.py`: Chocolatey package manager wrapper providing `choco_install()`, `choco_uninstall()`, and `choco_list()` with proper version handling and force flags.
- `file_ops.py`: File operations including HTTP download with progress callbacks, ZIP extraction, 7-Zip archive extraction (with automatic `7z.exe` discovery), and silent installer execution.

4.2.3 Linux Backend

The `backends/linux/` package implements full Linux support:

- `pm.py`: An abstraction layer over multiple Linux package managers. `LinuxPM` class detects the available package manager and exposes a uniform interface:
  - `update_package_lists()` — Runs the distribution's package list update command
  - `install_packages()` — Installs packages using the detected manager
  - `remove_packages()` — Removes packages
  - `is_installed()` — Checks if a package is installed
  - `get_version()` — Retrieves the installed version of a package

  Supports `apt-get`/`apt` (Debian/Ubuntu), `dnf`/`yum` (Fedora/RHEL), `pacman` (Arch), `zypper` (openSUSE), and `apk` (Alpine).
- `search.py`: Linux executable discovery scanning standard PATH directories as well as `/opt`, Snap, and Flatpak installation paths.
- `file_ops.py`: Linux-specific file operations including `tar` archive extraction, shell script execution (`.sh`), and binary installer handling (`.run`).
- `__init__.py`: `LinuxBackend` class implementing the ABC interface. Integrates sudo credential caching via a `sudo -S -v` validity check followed by `pkexec` for GUI password prompts. Implements `threading.Lock` serialization for bash script execution to prevent concurrent package manager conflicts.

4.3 Centralized Tool Metadata Registry

The `registry.py` module consolidates all tool metadata into a single source of truth, eliminating the duplication that previously existed across four separate files. The core data structure is the `ToolSpec` dataclass:

```
@dataclass(frozen=True)
class ToolSpec:
id: str
label: str
category: str          # 'analog', 'digital', '
    core', 'system'
versions: List[str]
default_version: str
executables: Dict      # OS id -> executable
    name
download_urls: Dict    # version -> download URL
install_scripts: Dict  # OS id -> bash script
    path
description: str
```

The registry defines complete specifications for seven managed tools: eSim, KiCad, Ngspice, GHDL, Verilator, LLVM, and Chocolatey. Each specification includes:

- Supported versions with exact download URLs (e.g., KiCad 6.0.11 through 9.0.7; GHDL 4.0.0 through 5.1.1)
- Executable names for each supported operating system
- Linux install script paths for tools that require bash-based compilation
- Category classification (analog, digital, core, system)

Convenience dictionaries (`TOOL_LABELS`, `TOOL_VERSIONS`, `ANALOG_TOOLS`, `DIGITAL_TOOLS`) provide backward compatibility with existing UI code, while Windows-specific search paths (`WIN_KICAD_PATHS`, `WIN_NGSPICE_PATHS`, `WIN_LLVM_PATHS`) and version mappings (`VERILATOR_VERSIONS`, `KICAD_VERSIONS`, `NGSPICE_VERSIONS`, `LLVM_VERSIONS`) are also centralized here.

4.4 Worker Thread Framework

The `worker.py` module provides a reusable set of QThread-based worker classes for asynchronous operations:

- `BaseWorker(QThread):` Abstract base worker with cancellation support, timeout handling via `threading.Timer`, and process lifecycle management. Provides `_make_popen()` for creating subprocesses with automatic timeout cleanup and `_iter_stream()` for real-time output iteration.

- `InstallWorker(BaseWorker):` Orchestrates installation of multiple tools sequentially. Emits `progress` signals with status messages and a `finished` signal upon completion. Each tool is installed by invoking the CLI adapter with the tool ID and version.

- `CommandWorker(BaseWorker):` Executes an arbitrary command for a specific tool, optionally providing a sudo password. Emits per-line progress and a final `finished` signal containing the complete output. Used by the GUI for ad-hoc operations.

- `InstallerThread(BaseWorker):` Runs bash installation scripts for a list of packages with intelligent progress estimation. Maps keywords in script output to estimated progress states (e.g., "downloading" → download phase, "compiling" → build phase), providing meaningful progress feedback to the user during long-running builds.

4.5 Unified Tool Function Registry

The `backends/tools/tools.py` module consolidates all per-tool check, install, and uninstall logic into a single file with a dispatcher pattern. Previously implemented as six separate files (one per tool), the merged implementation uses a `TOOL_FUNCS` dictionary:

```
TOOL_FUNCS = {
        "esim": {"check": check_esim, "install":
            install_esim, ...},
        "kicad": {"check": check_kicad, "install
           ": install_kicad, ...},
        "ngspice": {"check": check_ngspice, "
           install": install_ngspice, ...},
        "ghdl": {"check": check_ghdl, "install":
            install_ghdl, ...},
        "verilator": {"check": check_verilator,
           "install": install_verilator, ...},
        "llvm": {"check": check_llvm, "install":
            install_llvm, ...},
```

```
                    "chocolatey": {"check": check_chocolatey
                        , "install": install_chocolatey,
                        ...},
                }
```

Each tool function uses the backend interface for all platform-specific operations, with `IS_LINUX` branches where Linux and Windows paths diverge. This eliminated a substantial amount of duplicated boilerplate across the original six tool files, centralizing all per-tool logic into a single, maintainable dispatch table.

4.6 Linux Package Manager Abstraction

The `backends/linux/pm.py` module implements a distribution-agnostic package manager abstraction. The `LinuxPM` class detects the available package manager at runtime and maps common operations to distribution-specific commands:

- **apt/apt-get (Debian, Ubuntu):** `apt-get update`, `apt-get install -y`, `apt-get remove -y`, `dpkg -l`
- **dnf (Fedora, RHEL 8+):** `dnf check-update`, `dnf install -y`, `dnf remove -y`, `rpm -q`
- **pacman (Arch Linux):** `pacman -Sy`, `pacman -S -noconfirm`, `pacman -Rs -noconfirm`, `pacman -Qi`
- **zypper (openSUSE):** `zypper refresh`, `zypper install -y`, `zypper remove -y`, `rpm -q`
- **pkg (FreeBSD):** `pkg update`, `pkg install -y`, `pkg delete -y`, `pkg info`
- **xbps (Void Linux):** `xbps-install -S`, `xbps-install -y`, `xbps-remove -y`, `xbps-query`

This abstraction allows eSim to be installed and maintained across the full spectrum of Linux distributions without distribution-specific code paths.

4.7 CLI Adapter and Privilege Elevation

The `tool_manager_linux.py` module provides a command-line interface adapter for the Linux backend, mirroring the existing Windows adapter. It handles:

- **PyQt6 Import Safety:** Inserts the tool manager directory into `sys.path` at runtime to resolve import paths correctly, preventing crashes when launched from different working directories.
- **Sudo Credential Caching:** On Linux, package management commands typically require root privileges. The implementation first checks if valid sudo credentials exist via `sudo -n -v`. If not, it falls back to `pkexec` (GUI password prompt) for the initial elevation, then caches credentials for subsequent operations using `sudo -S -v`.
- **Lock Serialization:** Linux package managers do not support concurrent invocations. A `threading.Lock` ensures that bash scripts and package manager commands are serialized, preventing race conditions and lock file conflicts.

Chapter 5

Frontend Integration (Ongoing Work)

While the backend architecture forms the core contribution of this internship, significant frontend work has been carried out in parallel by other team members to integrate the Tool Manager into the eSim user interface. This chapter provides an overview of the ongoing frontend work to contextualize how the backend interfaces with the user-facing components.

5.1 PyQt6 Migration and UI Refactoring

The eSim codebase was originally built on PyQt5. A systematic migration to PyQt6 was undertaken across the Tool Manager UI components to stay current with the Qt framework and take advantage of improved Python 3 type safety. This involved:

- Updating all PyQt5 enum references to their PyQt6 equivalents (e.g., `Qt.AlignLeft` → `Qt.AlignmentFlag.AlignLeft`)
- Replacing deprecated API calls (`QDesktopWidget.primaryScreen()` → `QScreen`)
- Updating signal-slot signatures for PyQt6 strict matching
- Migrating checkbox, list view, and edit trigger patterns to PyQt6 conventions

5.2 UI/UX Enhancements and Toolbar Integration

The Tool Manager interface was enhanced with improved visual design and integrated into the main eSim application toolbar for quicker access. Enhancements include:

- A three-tab layout for the individual tool manager, organizing tools by category
- A modern custom scroll bar implementation for smoother navigation
- Responsive design adjustments preventing layout distortion on smaller screens
- Dedicated taskbar icon for improved OS-level window management
- Windows Dark Mode support with corrected contrast styling
- Polished typography, spacing, and section structuring

5.3 Current Status

As of July 2026, the backend refactoring is complete and functional. The frontend UI components are in an active state of integration and refinement. The three-tab layout for the individual tool manager has been implemented, and the main toolbar integration is functional. Further UI polish, user experience testing, and cross-platform validation are ongoing as part of continued development efforts by the eSim team.

5.4 Planned Frontend Enhancements

The frontend integration roadmap includes further refinements: completing the unification of the individual and full tool manager views, polishing the three-tab layout with persistent tool state indicators, adding real-time installation progress visualization using the worker framework's signal emission, and conducting cross-platform UI testing across Windows and Linux. These enhancements are being carried out in coordination with other team members as part of the broader eSim development cycle.

Chapter 6

Technical Challenges and Solutions

This chapter discusses the key technical challenges encountered during the internship and the solutions adopted.

6.1 Breaking the 1,434-Line Monolith

Challenge: The original `tool_manager_windows.py` contained 1,434 lines of deeply coupled logic spanning executable search, Chocolatey integration, file downloading, archive extraction, version parsing, and installer execution. Every function was interdependent, making it impossible to test or modify any single concern independently.

Solution: A phased decomposition approach was adopted. First, an abstract base class (`Backend`) was designed to define the interface for all platform operations. Then, the monolithic file was systematically refactored into the `backends/` package: `base.py` for the ABC, `backends/windows/` for Windows-specific operations (search, Chocolatey, file ops), and `backends/linux/` for Linux operations. The original 1,434-line file was reduced to a 40-line CLI adapter that delegates to the new architecture. Each new module was verified independently before integration.

6.2 Cross-Platform Executable Discovery

Challenge: Finding installed executables on Windows and Linux requires fundamentally different approaches. On Windows, executables may be found via Chocolatey, the Windows Registry, well-known installation directories, or PATH. On Linux, the PATH is the primary mechanism, but tools may also be installed via Snap, Flatpak,

or in non-standard locations under `/opt`.

Solution: A multi-strategy search pattern was implemented for each platform. On Windows, the search chain is: Chocolatey listing → well-known installation paths → Windows Registry → PATH → common fallback locations. On Linux, the search chain is: PATH → `/opt` subdirectories → Snap → Flatpak. Each tool has a dedicated search function that implements the optimal discovery strategy for its typical installation pattern, with graceful fallbacks at each step.

6.3 Sudo Credential Caching on Linux

Challenge: Package management operations on Linux require root privileges. Using `pkexec` for every operation prompts the user for a password each time, creating a poor user experience during multi-tool installations. However, storing or reusing passwords is a security concern.

Solution: A two-tier approach was implemented. First, the system checks for valid cached sudo credentials using `sudo -n -v`. If valid, subsequent operations proceed without prompting. If not, `pkexec` is used once to launch a helper that runs `sudo -S -v` to cache credentials for the session duration. The cached credential state is tracked within the `LinuxBackend` instance, and all bash script executions are serialized with a `threading.Lock` to prevent concurrent package manager operations.

6.4 PyQt6 Compatibility Issues

Challenge: The migration from PyQt5 to PyQt6 introduced breaking changes in multiple areas: enum values became strict enum members instead of integers, selection behavior APIs changed, and several methods were deprecated or renamed.

Solution: Each compatibility issue was addressed systematically. Enum comparisons were updated to use the member directly rather than integer flags—for example, `flags=Qt.ItemFlag.ItemIsSelectable`. Subprocess execution paths were fixed to use the `PATH` variable for environment resolution. Deprecation warnings were resolved by updating to the recommended PyQt6 API. A comprehensive audit of all Qt method calls across the Tool Manager codebase was performed to fix all deprecated usage.

Chapter 7

Development Workflow and Timeline

The internship followed a phased development approach, with each phase building on the previous one:

Table 7.1: Development Phases and Deliverables

Phase	Timeline	Deliverables
1	Week 1	Cleanup: Removal of dead shell scripts and orphaned bytecode; repository familiarization
2	Week 2	Platform abstraction layer (`pm_platform.py`): Unified OS detection, subprocess helpers, admin utilities
3	Week 3–4	Modular architecture (`backends/` package): ABC base class, Windows backend, worker framework (`worker.py`), tool metadata registry (`registry.py`)
4	Week 5–6	Linux backend (`backends/linux/`): Package manager abstraction, executable search, file operations; `IS_LINUX` branches in tool functions
5	Week 7	CLI adapter, sudo improvements, lock serialization, path fixes
6	Week 8	Code consolidation: Merged all per-tool files into a single unified dispatch table, eliminating redundancy and centralizing tool logic

All development was tracked via Git on the `feature/tool-manager-integration` branch of the eSim repository, with commit messages following the Conventional Commits specification.

The complete set of backend changes has been published as Pull Request #580

on the FOSSEE/eSim GitHub repository. As of July 2026, the PR remains open and unmerged due to conflicts with parallel backend refactoring work by another team member on the same codebase. The architectural improvements are available for review and testing on the `feature/tool-manager-integration` branch. Also, similar refactoring changes are in development upon the frontend portion, which will be opened in the near future.

Chapter 8

Conclusion and Future Scope

8.1 Summary of Contributions

This internship delivered a comprehensive architectural overhaul of the eSim Tool Manager backend. The key contributions are:

1. **Platform Abstraction Layer:** A unified module providing OS detection, subprocess management, and privilege elevation, serving as the foundation for cross-platform support.

2. **Modular Backend Architecture:** An abstract base class design with separate Windows and Linux implementations, enabling clean separation of platform-specific concerns.

3. **Centralized Tool Registry:** A single source of truth for all tool metadata, eliminating the duplication that previously required changes in four separate files.

4. **Worker Thread Framework:** Reusable QThread-based workers for asynchronous operations, improving UI responsiveness during long-running installations.

5. **Linux Package Manager Abstraction:** Support for six package managers across the Linux ecosystem, making eSim maintainable on any major distribution.

6. **Code Reduction:** The refactored codebase is approximately 60% smaller (3,964 to 1,439 lines), with improved readability, testability, and maintainability.

8.2 Skills Developed

8.2.1 Technical Skills

- Cross-platform software architecture and design patterns (Abstract Base Classes, Adapter pattern, Registry pattern)
- Platform-specific development (Windows subprocess management, Linux privilege elevation, package manager internals)
- PyQt6 GUI framework (QThread workers, signal-slot patterns, migration from PyQt5)
- Python packaging, subprocess management, and threading
- Version control and collaborative development workflows

8.2.2 Professional Skills

- Code review and architectural decision-making
- Technical documentation and communication
- Incremental refactoring and backward compatibility
- Open-source development practices and community workflows

8.3 Future Scope

While the current implementation provides a robust foundation, several areas offer opportunities for future enhancement:

- **Configuration File Format:** Migrating tool metadata from Python dataclasses to YAML, JSON, or TOML configuration files, allowing non-developers to add or update tool definitions without modifying Python code.
- **Automated Testing:** Establishing a comprehensive test suite for each backend layer, including unit tests for individual tool functions and integration tests for end-to-end installation workflows.

- **Containerized Backends:** Exploring Docker-based tool management as an alternative to native installation, providing fully isolated and reproducible environments.
- **macOS Support:** Extending the backend abstraction to support macOS via Homebrew and MacPorts, following the same architectural patterns established for Windows and Linux.
- **Offline Mode:** Supporting installation from pre-downloaded archives for environments without internet connectivity.
- **PR Integration:** Resolving merge conflicts with the parallel frontend refactoring and integrating the modular backend into the main eSim codebase.
- **Continued Frontend Development:** Completing the UI integration of the Tool Manager, including the unified view and real-time progress feedback.

8.4 Final Remarks

The internship provided invaluable exposure to real-world software architecture challenges in an open-source EDA ecosystem. The redesigned Tool Manager backend is now more maintainable, extensible, and cross-platform capable. The modular architecture ensures that future contributors can add support for new tools, platforms, or package managers with minimal friction, supporting FOSSEE's mission of promoting open-source engineering tools in education.

Bibliography

[1] FOSSEE Project. *Free/Libre and Open Source Software for Education*. Indian Institute of Technology Bombay.
https://fossee.in/

[2] eSim. *Open Source EDA Tool*. FOSSEE, IIT Bombay.
https://esim.fossee.in/

[3] KiCad. *Schematic Capture and PCB Design Suite*.
https://www.kicad.org/

[4] Ngspice. *General Purpose Circuit Simulator*.
https://ngspice.sourceforge.io/

[5] GHDL. *VHDL Simulator*.
https://github.com/ghdl/ghdl

[6] Verilator. *Fast Verilog/SystemVerilog Simulator*.
https://verilator.org/

[7] LLVM. *Compiler Infrastructure*.
https://llvm.org/

[8] Chocolatey. *Package Manager for Windows*.
https://chocolatey.org/

[9] PyQt6. *Python Bindings for Qt 6*.
https://www.riverbankcomputing.com/software/pyqt/

[10] FOSSEE Semester-Long Internship Programme.
https://fossee.in/internship