eSim Summer Fellowship 2026

on

Tool Manager Enhancements for eSim

Submitted by

Akanksha Shrivastava

B.Tech in Computer Science & Engineering (AI ML)

Vellore Institute of Technology Bhopal

Under the guidance of

Prof. Prabhu Ramachandran

Principal Investigator

Department of Aerospace Engineering

Indian Institute of Technology Bombay

July 27, 2026

Abstract

This report documents the recent Tool Manager work carried out for the eSim Summer Fellowship 2026. The branch was updated to make eSim easier to install and maintain by improving backend portability, centralizing command execution and status reporting, and strengthening version-aware package handling on Windows.

The implementation brings together three practical improvements. First, common helper functions were extracted into a reusable utility layer for safe command execution, path discovery, and status output. Second, the Windows package manager now maintains explicit version catalogs and detection logic for KiCad, Ngspice, LLVM, and eSim-related installers. Third, the PyQt6 updater presents a clearer interface for selecting package versions and monitoring installation progress. These changes reduce setup friction, make the codebase easier to extend, and provide a more dependable workflow for users who rely on the eSim tool chain.

The work is especially relevant for users who need a predictable setup experience on Windows, where tool locations and installation methods can vary widely. By consolidating the Tool Manager behavior in one place, the branch reduces the chance of version mismatch, repeated manual configuration, and hidden installation errors. For a fellowship project focused on infrastructure and usability, these improvements matter as much as visible feature additions because they directly shape how consistently eSim can be installed and used.

Acknowledgment

I would like to express my sincere gratitude to the FOSSEE (Free/Libre and Open Source Software for Education) Project, IIT Bombay, for providing me with the opportunity to undertake this internship. The experience of contributing to an open-source Electronic Design Automation (EDA) project has been both challenging and rewarding, providing valuable practical exposure to software development and tool integration.

I am deeply grateful to Prof. Prabhu Ramachandran for providing the opportunity to participate in the FOSSEE Summer Fellowship Program and for supporting open-source software development in engineering education and research.

I would like to express my sincere appreciation to Mr. Kannan M. Moudgalya, Principal Investigator of the FOSSEE Project, whose vision and leadership have established a strong open-source ecosystem that enables students to contribute to nationally significant software projects.

I extend my heartfelt thanks to my mentor, Mr. Sumanto Kar, for his constant guidance, technical expertise, encouragement, and constructive feedback throughout the internship. His mentorship was instrumental in overcoming technical challenges and successfully carrying out the Tool Manager enhancements in eSim.

I am also grateful to my internal mentors, Mr. Varad Patil and Ms. Shanthi Priya K, for their continuous support, valuable suggestions, and technical guidance during the course of this internship. Their assistance greatly contributed to the successful completion of this project.

I would also like to thank the entire FOSSEE team and the open-source community for their support, coordination, and valuable interactions throughout the internship. Their collaborative efforts created an excellent environment for learning and open-source development.

During this fellowship, I worked on the Tool Manager branch of eSim, focusing on improving how the application detects, installs, and updates its external dependencies on Windows. This involved extracting common backend helpers into a reusable utility layer, building explicit version catalogs and detection logic for tools such as

KiCad, Ngspice, GHDL, Verilator, and LLVM, and redesigning the PyQt6 updater interface so that users can see installed versions, choose target versions, and track installation progress in real time. Working on this layer gave me practical exposure to backend utilities, installer workflows, version detection, and package-management logic within a real open-source codebase.

Finally, I sincerely thank everyone who contributed directly or indirectly to this internship. Working on the Tool Manager enhancements for eSim has significantly deepened my understanding of open-source software development, Windows-based dependency management, and the practical concerns of building a reliable, user-friendly installer and updater experience. The knowledge and experience gained during this internship will remain invaluable in my academic and professional journey.

Contents

Chapter 1

Introduction

1.1 Project Context

eSim is an open-source electronic design automation platform developed by FOSSEE, IIT Bombay. It supports circuit design, simulation, and related educational workflows for students, educators, and researchers. Because eSim depends on external tools such as KiCad, Ngspice, GHDL, LLVM, and related supporting utilities, the reliability of the surrounding tool chain has a direct impact on the usability of the platform.

The Tool Manager branch exists to manage this dependency layer. Rather than requiring users to install, configure, and update each external component manually, the branch provides a more controlled workflow for detection, installation, and update. The recent changes focus on improving this workflow for Windows users and on making the codebase more portable and maintainable.

This context is important because an application like eSim is judged not only by its core simulation features but also by how easily a new user can get started. If the supporting tools are hard to install or maintain, the value of the application is reduced before the user even begins to work. Tool Manager therefore acts as the practical bridge between the eSim source code and the environment in which students and developers actually use it.

1.2 Goals of the Work

The primary goal of the work was to make eSim easier to set up and maintain for end users. A practical setup tool must detect what is already installed, identify the correct version, fetch the missing components, and report progress clearly. The branch work therefore targets three broad goals: reduce manual effort, improve

portability, and make package management predictable.

Another goal was to make the setup process easier to understand for someone who may not be deeply familiar with the underlying tools. Clear package names, visible version choices, and progress feedback all reduce confusion during installation. In that sense, the work is not only technical but also usability-focused, because it helps translate a complex dependency chain into a manageable user workflow.

1.3 Scope

This report covers the Tool Manager changes that are visible in the current branch. The scope includes backend helper functions, version detection logic, Windows package resolution, download handling, and the PyQt6 updater interface. It does not attempt to document the entire eSim application; instead, it focuses on the part of the system that prepares and maintains the supporting software stack.

The report intentionally stays within this scope because the branch work itself is concentrated there. Expanding into unrelated parts of eSim would make the report less accurate and dilute the value of the Tool Manager contribution. Keeping the discussion focused on the installer and updater flow makes it easier to connect the code changes with the actual fellowship work completed by the author.

1.3.1 Target Users

The Tool Manager is primarily meant for students, developers, and first-time users who want to set up eSim without spending a large amount of time on dependency troubleshooting. These users benefit most from clear package names, visible version choices, and a simple installation workflow that does not require them to understand every backend detail before they can begin using the application.

1.3.2 Branch Focus

The branch focus is intentionally narrow. Rather than spreading effort across unrelated parts of eSim, the work concentrates on the package-management layer that directly supports installation and updates. That narrow focus made it easier to improve one part of the codebase thoroughly and document the results clearly in this report.

1.3.3 Reporting Approach

The report presents the Tool Manager work in the same order in which a user or maintainer would encounter it: why the manager exists, what problems it solves, how it is implemented, and what results it delivers. This approach keeps the discussion practical and makes it easier to follow the connection between the code and the user-facing workflow.

1.3.4 Work Boundaries

The chapter also makes the boundaries of the fellowship contribution explicit. It does not claim to cover every eSim module or every installer script in the repository. Instead, it focuses on the Tool Manager branch and the portion of the system that was actually updated during the fellowship.

Chapter 2

Background and Motivation

2.1 Why Tool Management Matters

A circuit design environment depends on several external executables and libraries. If one component is missing or incompatible, the user experience degrades quickly. For example, KiCad version mismatches can affect symbol libraries and schematic compatibility. Ngspice version issues can change simulation behavior. GHDL and LLVM mismatches can disrupt digital design workflows. A package manager for eSim therefore has to coordinate several moving pieces, not just one application installer.

The recent branch work responds to this reality by concentrating installation logic in one place. This approach reduces duplication, makes version rules explicit, and provides a clearer path for users and maintainers when external dependencies change.

It also makes future updates easier to manage. When a tool version changes upstream, the maintainers can update the package catalog or detection logic instead of rewriting the whole workflow. That separation of concerns is a major advantage in a project that must support multiple tools with different release cycles.

2.1.1 Dependency Complexity

eSim does not depend on a single binary. It depends on a chain of tools that must work together for the overall workflow to be useful. The manager therefore has to treat dependency handling as a first-class problem rather than as a side task, because a failure in one tool can affect the usefulness of the entire environment.

2.1.2 Version Sensitivity

Version sensitivity is one of the hardest parts of tool management. A version that works in one context may not be compatible with another, and a user may not know which release is correct without guidance. The Tool Manager helps by making supported versions visible and by linking installation choices to known targets.

2.1.3 Maintainability

When dependency logic is duplicated across scripts, maintenance becomes fragile. Centralizing the workflow in the Tool Manager branch reduces that risk because future updates can be made in one place. That is especially important for an open-source project that may need to evolve over multiple release cycles.

2.1.4 User Confidence

A good tool manager also builds user confidence. If installation progress, package detection, and error handling are all visible, the user has a much better sense of what the system is doing. That matters because setup problems often feel worse when the application gives no meaningful feedback.

2.2 Portability Concerns

The earlier installer workflow had to account for different installation paths, tool locations, and bundled environments. On Windows, some systems provide KiCad and ngspice through standard Program Files locations, while others install them through package managers or custom directories. MSYS2 can also appear in more than one layout. A robust tool manager must therefore search multiple candidate paths and fall back gracefully when a tool is not found.

The current branch addresses these concerns by extracting common path-resolution helpers and by introducing clearer constants for the default locations used during detection and execution.

Those changes are valuable because they reduce the number of assumptions hidden inside the scripts. A path that is written once as a constant or helper is easier to audit and adjust than the same path repeated across multiple install files. For users, that means fewer surprises; for maintainers, it means fewer places where a Windows installation change can break the workflow.

2.2.1 Windows Path Variation

Windows installations are not uniform, especially when applications are installed from different sources. The Tool Manager has to consider standard Program Files directories, custom folders, and bundled environments. That flexibility is necessary if the same report and codebase are expected to work across multiple machines.

2.2.2 Fallback Discovery

The branch uses fallback discovery so that a missing default path does not immediately stop the workflow. If one location does not contain the expected executable, the search can continue in other common locations or through executable lookup utilities. This is a practical way to make the manager more resilient without overcomplicating the interface.

2.2.3 Environment Assumptions

Reducing hidden environment assumptions makes the code safer to maintain. Instead of assuming a single install layout, the branch collects likely paths and checks them explicitly. That makes the behavior easier to reason about when a user reports a setup issue.

2.2.4 Bundled Support

Bundled support is especially useful when a tool manager must work in environments that do not have every dependency preinstalled. By accounting for bundled or alternate setups, the branch can support more than one installation style while still keeping the logic readable and predictable.

Chapter 3

Problem Statement

The core problem is that eSim depends on a tool chain that is both version-sensitive and platform-sensitive. Users should not need to remember where each package was installed, which version is compatible, or how to update a tool safely. The project therefore needs a management layer that can detect installed versions, choose valid targets, run installers without exposing the user to unnecessary console clutter, and keep the system state understandable.

In practical terms, this means the Tool Manager must solve several smaller problems at once. It must identify executables reliably, execute commands safely, stream output without freezing the interface, show the current and target versions in a readable form, and handle download caching or repeated installs without corrupting the environment. The current branch is shaped around these requirements.

The problem is therefore not limited to installation alone. It is also about visibility, recoverability, and repeatability. A good manager should make it obvious what it found, what it is doing, and what to do if something fails. The current branch moves toward that standard by making the control flow more explicit and by surfacing progress in the UI.

3.0.1 Manual Setup Friction

Without a dedicated manager, users would need to download tools one by one, identify compatible versions, and configure paths manually. That process creates friction even before the first simulation is run. The Tool Manager exists to remove that burden and to make the setup sequence more predictable.

3.0.2 Version Mismatch Risk

One incorrect version choice can break part of the workflow or produce confusing behavior later. The manager addresses that risk by exposing supported versions and by tying installation paths to explicit version catalogs. This reduces the likelihood of mismatched combinations being installed accidentally.

3.0.3 Execution Reliability

Installers and helper scripts must run reliably because they are part of the setup path, not the main application runtime. If a script hangs or fails silently, the user is left with an incomplete environment. The branch therefore emphasizes safe command execution, timeout handling, and visible logging.

3.0.4 Feedback and Recovery

The system should also make recovery easier. If something fails, the user should know whether the issue came from download, detection, or execution. That kind of clear feedback turns a vague installation failure into a solvable problem and is a major part of the Tool Manager's value.

Chapter 4

Implementation

4.1 Backend Utility Layer

A major structural change in the branch is the extraction of common backend helpers into a dedicated utility module. The `utils.py` file centralizes command execution, executable lookup, status printing, and MSYS2 discovery. This reduces duplication across installer scripts and allows the other modules to depend on a stable set of helper functions.

The command execution helpers are designed with Windows usability in mind. `run_cmd_safe` executes a command with window suppression and a timeout, which helps prevent hanging subprocesses from blocking the tool. `run_cmd_stream` streams output in real time, which is important for long-running package builds or installs because users can see progress and error messages as they occur. The `print_status` helper standardizes status output so that the surrounding UI or launcher can parse it consistently.

The utility module also defines default path constants for Windows installations. These include candidate locations for MSYS2, eSim, KiCad, Ngspice, and LLVM. By collecting these values in one place, the branch makes future maintenance easier because a change in one installation assumption does not require editing multiple scripts.

This design also makes the code more readable for contributors. Instead of searching through multiple scripts to understand where a tool is expected to live, a maintainer can inspect one module and see the supported locations immediately. That is a practical improvement for long-term maintenance, especially in a project that may continue to evolve after the fellowship ends.

4.1.1 Reusable Backend Design

The backend helpers were designed to be reusable across multiple scripts, which reduces duplication and keeps the control flow consistent. This is useful because installer scripts often need the same basic operations: locating tools, running commands, and reporting status. A shared utility layer makes those operations easier to maintain.

4.1.2 Command Safety

The command execution helpers are written to avoid unnecessary failures and to keep the user environment stable. Timeouts, hidden windows on Windows, and captured output all reduce the chance that a long-running installation step will leave the system in an unclear state. That makes the tooling safer to use during installation and update operations.

4.1.3 Path Resolution

Path resolution is one of the most important practical tasks in Tool Manager. The branch defines candidate locations for common tools and checks them in a structured way so that the rest of the workflow can rely on the results. This keeps the manager robust without forcing users to manually locate every dependency.

4.1.4 Status Communication

The status output format is important because it gives the UI and launcher a predictable way to understand what the backend is doing. That consistency is part of what makes the branch usable as a manager rather than just as a collection of scripts.

4.2 Version Detection and Package Catalogs

The Windows package manager logic in `tool_manager_windows.py` introduces explicit version catalogs for the supported tools. KiCad, Ngspice, LLVM, and eSim are listed with known versions or direct download URLs. This makes the supported matrix visible in the code and gives the updater a clear source of truth.

The version detection functions use a layered approach. First, they check known filesystem locations and bundled paths. If that fails, they can inspect system-level information such as registry entries or command-line version output. For KiCad, the

code looks through multiple Windows installation directories and can also consult the Windows registry. For Ngspice and LLVM, it falls back to executable discovery and version parsing.

This layered approach is important because it avoids a single point of failure. Users can install software in different ways, and the manager still has a chance to identify what is present and whether it matches the requested version.

It also allows the updater to present more meaningful choices. Rather than showing an arbitrary list of versions, the interface can be tied to known targets that are actually supported by the install scripts. That keeps the experience closer to real deployment conditions and reduces the risk of choosing a version that cannot be resolved cleanly.

4.2.1 Supported Versions

Supported versions are listed explicitly so that the manager can present a controlled set of choices. This reduces ambiguity for the user and reduces the chance that an incompatible release will be selected during installation.

4.2.2 Detection Flow

The detection flow is layered so that the branch can still identify installed software when the first check fails. This matters because users do not all install software in the same way, and the manager must remain flexible enough to adapt to those differences.

4.2.3 Registry and File Checks

For Windows-specific tools such as KiCad, the branch can check both common file locations and registry-based installation data. That dual approach helps the manager find tools that were installed through different mechanisms while still keeping the code understandable.

4.2.4 Catalog Stability

Keeping the package catalog in code gives the updater a stable reference point. If the supported versions are visible and maintained together, then future updates are easier to audit and less likely to introduce mismatched combinations.

4.3 Supported Package Matrix

The current branch makes the supported package combinations explicit in code. The table below captures the most visible version information used by the Windows updater and installer logic.

This table is useful because it turns a scattered set of version rules into a compact reference. For users, it shows what the manager is prepared to handle; for maintainers, it documents the practical boundaries of the current branch. That kind of explicitness is useful in a report because it connects the implementation work with a concrete operational matrix.

Package	**Supported Versions or Targets**	**Notes**
KiCad	6.0.11, 7.0.11, 8.0.9, 9.0.x candidates	Windows path detection checks standard Program Files locations and registry entries.
Ngspice	35 through 43	The updater includes version selection and shell-script based installation support.
GHDL	3.0.0, 4.0.0, 4.1.0, nightly	Used with digital design flows and installer orchestration through the updater GUI.
Verilator	4.228, 5.020, 5.026, 5.030	Added to the package manager so digital simulation dependencies stay aligned.
eSim	2.2, 2.3, 2.4	Installer URLs are defined explicitly so the manager can fetch the correct release.

4.4 Download and Installation Flow

The branch also standardizes how packages are downloaded and cached. The `Download` directory inside the Tool Manager area is used to store fetched installers and archives.

If a package already exists locally, the manager can reuse it rather than downloading it again. This saves time and reduces network dependency.

The same mechanism also helps with repeatability. If a package must be installed again on another machine, or if a user reruns the installer after a failed attempt, caching avoids unnecessary downloads and keeps the workflow consistent. That is especially valuable for larger packages where repeated network fetches would slow down the process.

The download helper performs the fetch operation with visible progress messages. If the download fails, the code removes the partially downloaded file to avoid leaving behind a corrupt package. This is a small but important reliability improvement because broken downloads can otherwise cause difficult-to-diagnose install failures later in the workflow.

4.4.1 Caching Behavior

Caching is important because it avoids repeated downloads and makes the workflow faster on repeated runs. It also reduces dependency on the network when the same package needs to be installed more than once.

4.4.2 Installer Flow

The download and installation flow are kept closely connected so that the manager can fetch a package, run the relevant script, and report status in one predictable sequence. This makes the overall experience feel integrated instead of fragmented.

4.4.3 Failure Cleanup

When a download fails, the branch removes incomplete files rather than leaving them behind. That cleanup step prevents a partially downloaded archive from causing a later install failure that would be harder to trace.

4.4.4 Repeatability

The same package can be installed again on another machine or after a previous failure without changing the logic. That repeatability is one of the practical strengths of the current Tool Manager design.

4.5 Updater User Interface

The PyQt6 updater in `updater_gui.py` provides the user-facing installation and update experience. The interface is built around a table containing four rows for the major packages: KiCad, Ngspice, GHDL, and Verilator. Each row includes a checkbox, the package name, the currently installed version, and a version selector.

The interface is deliberately simple because its job is to guide a user through a technical process without overwhelming them. A table layout keeps the package selection visible, and the version dropdowns make it clear what is available. That matters for a tool manager because the UI should support decision-making rather than hide the underlying installation logic.

The GUI is not only decorative; it acts as an interactive control surface for package management. The user can select one or more packages, choose a target version from the available list, and then start an installation process. The supported version lists are explicit, which helps prevent accidental selection of unsupported combinations.

A background `InstallerThread` handles the actual installation work. It launches the selected shell scripts and listens to their output line by line. The thread converts key log words such as installing, configuring, compiling, or extracting into progress updates. That keeps the interface responsive while still giving the user meaningful feedback on what stage the installer has reached.

This approach also makes long-running operations less stressful for the user. Instead of seeing a frozen window or waiting without feedback, the user receives status updates that describe the install process in human terms. For a tool manager, that kind of responsiveness is a major part of usability.

4.5.1 Table Layout

The table layout presents all major packages in a single view, which makes it easier for the user to compare current and target versions. That keeps the installation controls compact while still showing the information needed to make a choice.

4.5.2 Threaded Execution

The installer runs in a background thread so the GUI remains usable while the installation is in progress. This is important because users need visible feedback and should not be forced to wait on a frozen interface.

4.5.3 Progress Feedback

Progress feedback is translated from script output into user-facing status updates. That helps bridge the gap between technical installer logs and the simpler information a user actually needs during setup.

4.5.4 Selection Controls

Checkboxes and version selectors make the interface practical for multi-package installation. The user can select only the packages they need, which keeps the process flexible and avoids unnecessary installation steps.

4.6 User Experience Improvements

The branch makes several small user experience improvements that matter in practice. Installed packages are shown in green, missing packages in red, and the refresh action updates the table after detection is rerun. The window also uses a cleaner table layout and explicit version selection controls so the user can understand the current state quickly.

These details reduce the cognitive load during setup. Instead of asking the user to infer what the manager is doing, the interface exposes the information directly: what is installed, what is available, what is being installed, and whether the process completed successfully.

They also make the tool easier to demonstrate and support. When someone asks what the manager is doing, the UI already exposes the relevant state. That makes troubleshooting more straightforward and gives the project a cleaner, more polished feel.

4.6.1 Visual Clarity

The use of color and compact table rows makes the interface easier to scan quickly. Users can identify installed and missing tools without reading through a large amount of text.

4.6.2 Lower Cognitive Load

The interface reduces the amount of setup knowledge the user must carry in their head. Rather than remembering package names, versions, and commands separately, the user can work directly from the manager's on-screen choices.

4.6.3 Support Readiness

Because the interface surfaces package state clearly, it is easier to support and explain. That matters when the same report and the same tool need to be understandable to both users and maintainers.

4.6.4 Polished Workflow

The refreshed UI behavior gives the manager a more polished feel overall. The user sees a clearer workflow from detection to selection to installation, which makes the tool feel more complete.

Chapter 5

Testing and Validation

The branch changes were validated by checking detection logic, installer script wiring, and GUI behavior in the expected tool combinations. The key checks were straightforward: confirm that known tools can be found in standard Windows locations, verify that version parsing works for installed executables, and ensure that the updater can launch the correct script for each selected package.

This style of validation is appropriate for an infrastructure-heavy branch because the most important failures are usually in discovery, wiring, and version matching rather than in a visible algorithmic output. By checking those paths directly, the branch demonstrates that the manager can support the intended install flow with less manual intervention.

Reliability was also improved through defensive behavior in the backend. Commands are run with explicit timeouts, output is captured in a controlled way, and failures are reported rather than ignored. This means a broken installation path or a missing executable produces a visible problem instead of silently failing.

Another useful validation point is package selection. Because the GUI restricts the available versions per tool, users are less likely to choose a version that the installer scripts do not support. That version catalog is especially important for the packages that need to remain compatible with eSim-specific workflows.

Together, these checks suggest that the branch is moving toward a more stable release-ready workflow. The code does not merely add buttons or scripts; it connects the installation logic, the version data, and the interface behavior into one operational path that is easier to reason about and maintain.

In practice, the validation process also helps reveal where the tool manager can be improved further. For example, if an expected executable is missing from the standard locations, the detection logic can be extended without changing the rest of the workflow. If a script reports an error, the progress output makes it easier

to trace whether the problem came from download handling, version selection, or execution permissions. That kind of traceability is useful in both development and support.

The validation work therefore goes beyond simple success checks. It confirms that the branch can fail in understandable ways, which is important for a tool intended to be used by students and other non-specialist users. Clear failures are easier to debug, easier to document, and easier to recover from than silent or ambiguous ones.

5.0.1 Stability Checks

The most important tests are the ones that confirm the manager can consistently locate tools and launch the correct scripts. Those checks show whether the installation flow is stable enough for practical use.

5.0.2 Error Traceability

When something goes wrong, the progress and log output make it easier to identify the failing stage. That traceability is useful because it turns a broad installation failure into a more specific debugging task.

5.0.3 User-Oriented Validation

The validation process also checks whether the system fails in a way that users can understand. A tool manager is only helpful if it can explain its own behavior clearly enough for non-specialist users to act on it.

5.0.4 Release Readiness

These checks indicate that the branch is moving closer to a release-ready state. The code now behaves like a structured manager rather than a loose collection of scripts, which is exactly what an installation tool needs.

Chapter 6

Results

The most visible result of this work is a more coherent Tool Manager flow for eSim. The backend logic is now organized around reusable helpers, the Windows manager can detect and classify installed packages more cleanly, and the updater interface provides a direct path from package selection to installation progress.

The deeper result is that the Tool Manager now behaves more like a managed system and less like a collection of disconnected scripts. That shift matters because it improves confidence in the setup process and makes the branch more sustainable for future contributors.

From a maintenance point of view, the code is also easier to extend. Adding a new detection rule, a new download URL, or a new package script is now more local because the branch separates common behaviors from package-specific logic. That reduces the risk of accidental regressions when future updates are made.

From a user point of view, the process is simpler and more predictable. The manager tells the user what it found, what it is about to install, and how the installation is progressing. That is the main practical improvement delivered by the branch.

From a fellowship perspective, this is also a solid example of infrastructure work with direct impact. The branch may not change the core simulation engine, but it makes the ecosystem around eSim more reliable, which is what allows the core application to be used effectively.

Another result of the work is that the branch now leaves a clearer path for future improvements. Once the package detection and updater structure are organized, adding support for another version, another installer source, or another path convention becomes much easier. The result is not only a better tool manager today but also a more maintainable foundation for the next iteration of the project.

This also has value for the report itself because it shows that the fellowship

contribution was not isolated to one small bug fix. It involved shaping the support layer of the application in a way that can keep paying off later. In an open-source project, that kind of architecture-oriented improvement often has a longer lifespan than a single feature change.

6.0.1 Operational Stability

The Tool Manager now behaves more predictably because the backend, download flow, and GUI all participate in the same workflow. That stability is valuable because setup tools must be dependable before they can be useful.

6.0.2 Maintainability Outcome

The changes reduce duplication and make future edits easier to place in the right file. That is an important result because maintainable infrastructure tends to matter more over time than a short-term convenience fix.

6.0.3 User Impact

The user sees clearer progress, clearer package choices, and fewer manual steps. That makes the setup process more approachable, especially for first-time users and students who are trying eSim for the first time.

6.0.4 Project Value

The branch adds value to eSim even though it sits outside the simulation engine itself. It improves the ecosystem that users need in order to reach the core functionality, which makes the work meaningful at the project level.

Chapter 7

Conclusion and Future Scope

The Tool Manager branch work strengthens eSim by turning dependency setup into a guided and reproducible workflow. Instead of relying on scattered manual installation steps, the project now has a clearer backend for detecting tools, handling downloads, and reporting installation status. That makes the setup process more dependable for students and other users who need eSim to work consistently across different Windows environments.

This kind of work is especially valuable in an open-source project because the same reportable behavior helps both new users and future maintainers. A clearer Tool Manager means fewer setup barriers today and less technical debt tomorrow.

The most important result of the work is not just that the tools can be installed, but that the installation story is now easier to understand and maintain. The backend helpers reduce duplication, the version catalogs make compatibility visible, and the updater GUI gives users a straightforward way to choose and install packages. Future improvements can extend the same structure with broader cross-platform support, persistent user preferences, richer failure diagnostics, and automated tests for version detection and installer selection.

In summary, the fellowship work turned Tool Manager into a more structured and maintainable part of eSim. That is a good outcome for a CSE student contribution because it demonstrates practical software engineering beyond feature implementation: path handling, state management, user feedback, and maintainability all had to be addressed together.

If the branch continues to evolve, the same design direction can support more ambitious improvements without forcing a rewrite. The manager could eventually use a richer configuration model, remember user-selected locations, and offer clearer recovery steps when tools are not installed correctly. Those future extensions would build naturally on the structure established by the current work.

Overall, the project demonstrates that infrastructure work is not secondary work. For eSim, the ability to install and update the surrounding tool chain is a prerequisite for successful use of the application. Improving that layer was therefore a meaningful fellowship contribution, especially for a CSE student working in a focused branch of the codebase.

7.0.1 Summary of Contribution

The fellowship contribution is best understood as a support-layer improvement. It does not replace the core eSim application, but it makes the application easier to reach, install, and maintain. That is a practical result with long-term value.

7.0.2 Future Improvements

The same structure can support broader platform support, richer configuration storage, and more detailed error reporting. Those improvements would fit naturally into the current design without needing to change the basic workflow.

7.0.3 Open-Source Value

Because this is an open-source project, the Tool Manager work helps the community as much as the individual user. A better setup path lowers the barrier to entry for contributors and learners, which is important for the health of the project overall.

7.0.4 Learning Outcome

From a student perspective, the work demonstrates how software engineering includes not only building visible features but also making the surrounding system easier to use and maintain. That makes the fellowship experience valuable beyond the immediate code changes.

Chapter 8

Bibliography

- FOSSEE, IIT Bombay, eSim project documentation and user guides.
- eSim Tool Manager source files in the current branch.
- KiCad, Ngspice, GHDL, and LLVM official release and installation documentation.
- PyQt6 documentation for the updater interface and thread-based UI updates.
- Windows and MSYS2 path and command-line behavior references used during Tool Manager development.
- Git and GitHub documentation for managing source changes and branch-based development.
- Python documentation for subprocess handling, path utilities, and package execution support.
- Qt documentation for table widgets, combo boxes, and threaded GUI updates.