eSim Summer Fellowship 2026

On

Windows QA, Functional Validation and Tool Manager Testing for the eSim Tool Manager

Submitted by

Yash Savai

Bharatiya Vidya Bhavan's Sardar Patel Institute of Technology (SPIT), Mumbai

Under the guidance of

Prof. Prabhu Ramachandran
Principal Investigator
Indian Institute of Technology Bombay

July 2026

Declaration

I hereby declare that the work presented in this report, titled *"Windows QA, Functional Validation and Tool Manager Testing for the eSim Tool Manager"*, is a genuine record of the work carried out by me during the eSim Summer Fellowship, undertaken as part of my academic and technical engagement while studying at Bharatiya Vidya Bhavan's Sardar Patel Institute of Technology (SPIT), Mumbai.

The eSim Tool Manager was built through the collaborative effort of the full fellowship team. My individual contribution and area of primary responsibility was Windows Quality Assurance (QA): functional validation, installation testing, regression testing, bug discovery and documentation, and verification of fixes following Pull Requests. Wherever implementation work is described in this report, it is explicitly identified as collaborative. This report has not been submitted, in part or in full, for any other purpose, and all sources referred to have been duly acknowledged.

Yash Savai
Bharatiya Vidya Bhavan's Sardar Patel Institute of Technology (SPIT), Mumbai

Acknowledgement

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, Mr. **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 Mr. **Varad Patil** and Ms. **Shanthi Priya K** for their support and encouragement throughout the course of this internship, which was genuinely helpful in navigating the challenges of the project.

I would also like to express my sincere gratitude to my teammate, Mr. **Mausam Kar**, for his valuable support, collaboration, and contribution throughout the internship. His cooperation and discussions during the project were helpful in addressing challenges and ensuring the smooth progress of the work.

This fellowship has been an enriching learning experience, allowing me to work closely with open-source EDA tools and to develop a structured, professional approach to software quality assurance. I would also like to thank the entire FOSSEE team for their coordination, assistance, and timely responses at various stages of this work; their collective efforts ensured a smooth workflow, ready access to resources, and effective project execution throughout the fellowship.

Finally, I thank the Department of Electronics and Telecommunication Engineering at Bharatiya Vidya Bhavan's Sardar Patel Institute of Technology (SPIT), Mumbai, for encouraging participation in external research and open-source fellowships of this kind, and for fostering an environment that supports learning beyond the classroom.

Abstract

This report documents my contribution to the FOSSEE eSim Summer Fellowship, focused on Windows Quality Assurance (QA) for the eSim Tool Manager, a package-management utility that installs and tracks the analog and digital EDA toolchains (KiCad, Ngspice, GHDL, Verilator and LLVM) used by eSim. While the Tool Manager itself was designed and implemented collaboratively by the fellowship team, my primary responsibility was to validate its behaviour on Windows 11: exercising the Analog and Digital Mode installation workflows, verifying that package-detection and version-reporting logic reflected the true state of the host system, testing the graphical interface for layout and rendering defects, and running regression passes after fixes were merged.

Over two structured QA cycles, fourteen distinct defects were identified and documented, spanning package-detection failures (GHDL, Verilator, LLVM, and eSim reported incorrectly after successful installation), a non-functional Refresh action, an unreliable Bulk Uninstall operation, and Windows-specific GUI layout corruption caused by DPI and font-metric handling. Each defect was logged with expected and actual behaviour, a probable root cause, and severity, and was tracked through to verification once a corresponding fix was merged via Pull Request. This report presents the Tool Manager architecture as I came to understand it through testing, the QA methodology and test environment used, a full account of the bugs discovered, the collaborative GitHub-based development process the team followed, and the learning outcomes and future work arising from the fellowship.

Keywords: eSim, FOSSEE, Tool Manager, Windows QA, Regression Testing, GUI Validation, Package Detection, Open-Source Contribution

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background of eSim and FOSSEE

FOSSEE (Free and Open Source Software for Education) is an initiative of the Ministry of Education, hosted at IIT Bombay, that promotes the use and development of open-source tools in technical education. eSim is FOSSEE's flagship Electronic Design Automation (EDA) platform: it brings together circuit schematic capture, SPICE-based analog simulation, digital/HDL simulation, and PCB design into a single, freely available application aimed at students, educators and hobbyist circuit designers.

Historically, eSim relied on manually installed dependencies – KiCad for schematic capture, Ngspice for analog simulation, and GHDL/Verilator for digital and mixed-signal simulation. This created a high barrier to entry, particularly on Windows, where dependency installation is less standardised than on Linux. The *Tool Manager* was introduced to solve this problem: a package-management front end, built into eSim, that can install, detect, update and remove these EDA tools through a single graphical interface, offering "Analog Mode" and "Digital Mode" one-click installation bundles in addition to individual tool management.

1.2 The FOSSEE Summer Fellowship

The FOSSEE Summer Fellowship invites students from across India to contribute to FOSSEE's suite of open-source tools over a structured, mentor-guided internship period. Fellows are assigned to a specific sub-project, work alongside a small team of peers and one or more mentors, and are expected to make demonstrable, reviewed contributions to the codebase – whether through feature development, bug fixing, testing, or documentation – culminating in a final report and project review.

I was assigned to the **Tool Manager** sub-project within eSim, working as part of a small team alongside my teammate Mausam Kar, under the guidance of Dr. Reena Sonkusare and the FOSSEE Tool Manager mentors. While development of the Tool Manager's installation and detection logic was carried out collaboratively by the team, my own focus for the duration of the fellowship was **Windows Quality Assurance**: functional and regression testing of the Tool Manager on Windows 11, verification of installation and package-detection behaviour, GUI testing, bug discovery and documentation, and re-verification of fixes once they were merged.

1.3 Objectives of the Internship

The specific objectives I set out to achieve during the fellowship were:

- Validate the Tool Manager's Analog Mode and Digital Mode installation workflows on a clean Windows 11 environment.
- Verify that package detection and version reporting (for eSim, KiCad, Ngspice, GHDL, Verilator and LLVM) accurately reflect the true state of the host system.
- Exercise the graphical interface – the Quick Setup, Individual Tools, and Bulk Uninstall tabs – to identify layout, rendering and responsiveness defects specific to Windows.

- Document every defect found in a structured, reproducible format so that developers could triage and fix issues efficiently.
- Re-test and verify each fix once it was merged through a Pull Request, and run regression passes to confirm no new defects were introduced.
- Collaborate with the development team through GitHub – raising issues, reviewing linked Pull Requests, and discussing root causes with the implementers.

1.4 My Role in the Project

The Tool Manager was, by design, a team effort: architecture decisions, the PyQt-based GUI, and the underlying installation/detection logic were built collaboratively by the fellowship cohort working on this sub-project. My individual role within that effort was that of the **Windows QA engineer** – I did not author the detection or installation logic myself, but I was responsible for independently exercising it, uncovering where its behaviour diverged from the intended behaviour on Windows, and providing the development team with the evidence and reproduction steps needed to fix it. This report is deliberately scoped to that contribution: it describes the Tool Manager as I encountered and tested it, not as a claim of sole authorship over its implementation.

1.5 Report Organisation

The remainder of this report is organised as follows. Chapter 2 describes the Tool Manager's architecture and workflow as understood through the testing process. Chapter 3 details the QA methodology, test environment, and the categories of testing performed. Chapter 4, the core of this report, documents every bug discovered during the fellowship, including expected versus actual behaviour, root cause, impact, and resolution status. Chapter 5 describes the collaborative, GitHub-based development process the team followed. Chapter 6 reflects on the technical and professional learning outcomes of the fellowship, and Chapter 7 concludes the report with a summary of achievements and proposed future work.

Chapter 2

Tool Manager Architecture and Understanding

2.1 Purpose and Position within eSim

The Tool Manager sits between the end user and the collection of third-party EDA tools that eSim depends on. Rather than requiring a user to separately locate, download and configure KiCad, Ngspice, GHDL, Verilator and LLVM, the Tool Manager exposes a single interface through which these can be installed, verified and removed. Understanding this architecture was a prerequisite for effective QA: to test the Tool Manager meaningfully, it was necessary to understand which sub-systems were responsible for installation, which were responsible for status detection, and how the two were expected to stay in sync.

2.2 High-Level Architecture

At a high level, the Tool Manager can be described as three cooperating layers: a *presentation layer* (the PyQt-based GUI, organised into the Quick Setup, Individual Tools and Bulk Uninstall tabs), an *orchestration layer* that sequences installation, update and removal operations against underlying package managers (principally Chocolatey on Windows), and a *detection layer* that queries the host system – via installed executables, PATH entries, or registry/version markers – to determine which tools are present and which versions are installed. Figure 2.1 summarises this arrangement as I came to understand it through testing.

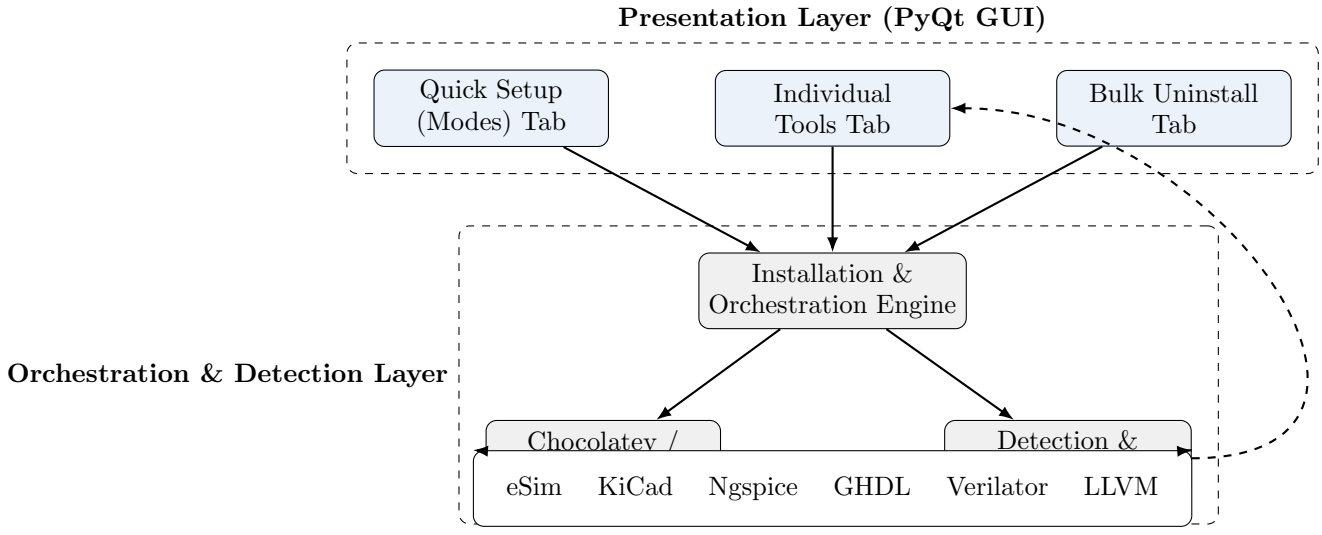

Figure 2.1: High-level Tool Manager architecture as understood from testing. The dashed arrow indicates the status-refresh path that was found, during testing, to be unreliable (see Chapter 4).

2.3 Package Management Workflow

The orchestration engine is responsible for sequencing an installation: resolving which package manager commands to invoke for a selected tool, tracking progress, and reporting completion or failure back to the GUI. On Windows, this layer primarily drives Chocolatey package installs. Because installation is a multi-minute, multi-step process (download, verification, unpacking, registration, and any post-install configuration), the orchestration layer also owns the Installation Progress panel shown to the user, including its percentage indicator and final success/failure message.

2.4 The Windows Tool Manager and Detection Layer

The detection layer is logically separate from the orchestration layer: its role is to independently query the host system and answer the question "is tool X installed, and at what version?" On Windows this can involve checking the system `PATH`, probing for known executable names, or reading installer-recorded state. This separation of concerns – install now, detect later – is a reasonable architecture, but it also introduces the central risk that QA testing was designed to surface: if the detection layer's assumptions about *where* and *how* a tool announces itself on Windows do not match what the installer for that tool actually does, the two layers fall out of sync, and the Tool Manager reports incorrect status even though the underlying installation succeeded.

2.5 Backend Overview

The GUI communicates with the orchestration and detection layers through internal callbacks that are expected to fire on events such as "installation complete" or "refresh requested." In principle, a successful install should trigger a detection re-scan, and a manual refresh should force the same re-scan regardless of any cached state. During testing, this expected data flow – summarised by the dashed arrow in Figure 2.1 – became the single most consequential architectural detail, since a large share of the defects found during this fellowship (documented in Chapter 4) trace back to this refresh path either not firing, or firing against a stale cached state rather than the live system.

Chapter 3

QA Testing and Validation

3.1 Testing Methodology

Testing was carried out over two structured QA cycles during the fellowship, each producing a standalone report that fed into the consolidated bug list presented in Chapter 4. The methodology followed for both cycles was consistent:

1. **Preparation** – reset the test machine to a clean state (or a known baseline) so that installation behaviour would not be masked by leftover files or registry entries from a previous run.
2. **Scripted functional pass** – exercise each defined test area (launch, installation, detection, GUI, refresh, uninstall) against the checklist in Section 3.3.
3. **Independent verification** – for every status the Tool Manager reported, independently confirm it using the Windows Command Prompt (e.g. `ghdl --version`, `kicad --version`), so that GUI-reported state could never be taken at face value.
4. **Documentation** – for every discrepancy, capture a screenshot, record expected vs. actual behaviour, assign a severity, and where possible propose a likely root cause.
5. **Regression pass** – after fixes were merged via Pull Request, re-run the relevant subset of tests to confirm the fix, and re-run the full checklist to confirm no new regressions had been introduced.

Figure 3.1 illustrates this testing lifecycle.

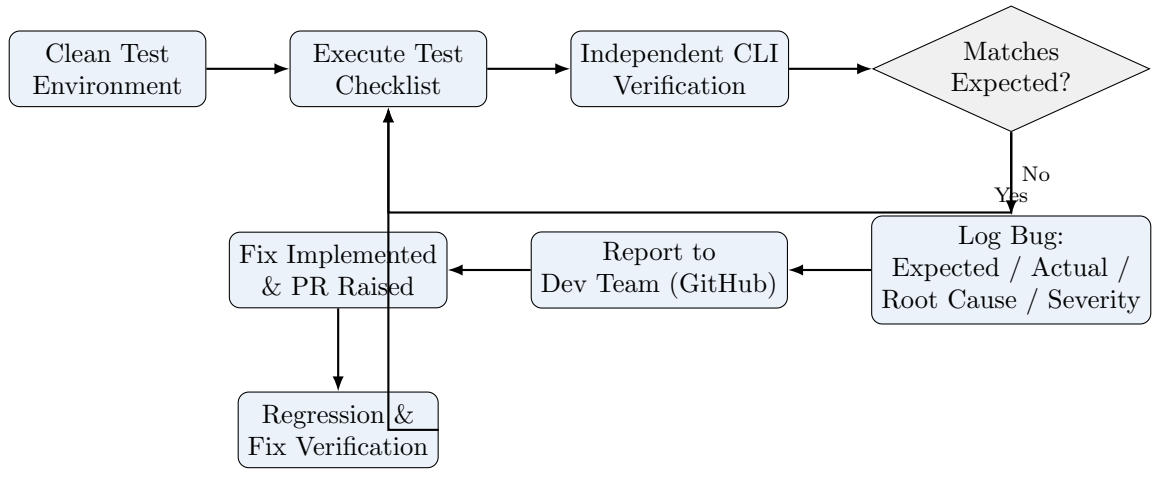

Figure 3.1: The QA testing lifecycle followed during the fellowship: from environment setup through independent verification, bug logging, and regression testing after a fix is merged.

3.2 Test Environment

All testing described in this report was performed on the following environment, matching the environment used for both QA cycles:

Table 3.1: Test environment used for Windows QA validation

Attribute	Detail
Operating System	Windows 11 Pro
Project	eSim Tool Manager
Branch	`feature/tool-manager-integration`
Package Manager	Chocolatey
Testing Types	Functional, Installation, GUI, Regression, Dependency Verification
Tester	Yash Savai

3.3 Functional Testing

Functional testing covered the core user journeys the Tool Manager is expected to support: launching the application, opening the Individual Tool Manager (referred to internally during earlier cycles as the "Advanced" view), and confirming that every visible control produces the effect its label promises. Ten distinct test areas were exercised: Tool Manager launch, Tool & Package Manager launch, Analog Mode installation, Digital Mode installation, package detection, version detection, GHDL validation, status synchronisation, GUI rendering, and general Windows compatibility.

3.4 Analog Mode Installation Testing

Analog Mode bundles eSim, KiCad and Ngspice as the minimum toolchain required for schematic capture and SPICE simulation. Testing this workflow involved triggering the "Install Analog Mode" action from a clean state and observing the full installation lifecycle: progress reporting, completion messaging, and the resulting status shown for each installed component. This installation consistently completed, but was found to stall visually near completion and to under-report its own success for at least one component (see Bug 4 and Bug 5 in Chapter 4).

3.5 Digital Mode Installation Testing

Digital Mode extends Analog Mode with GHDL, Verilator and LLVM for HDL and mixed-signal simulation. This is a heavier install, and testing focused specifically on whether the additional digital toolchain components were correctly detected once installation reported success – an area where multiple high-severity defects were found (Bugs 1, 3 and 4 in the consolidated list).

3.6 Version Verification and Package Detection

For every tool the Tool Manager claimed to have installed, the reported version string and installed status were independently checked from the Windows Command Prompt using each tool's own `--version` flag. This cross-check was the single most effective technique used during the fellowship: several of the highest-severity bugs (incorrect GHDL version, Verilator and LLVM reported "Installed" but not runnable, KiCad reported installed but not on `PATH`) were only detectable by comparing the GUI's claim against the command line's ground truth.

3.7 Regression Testing

After each fix was merged, the corresponding test case was re-run to confirm the fix, and the full functional checklist was re-executed to confirm that the fix had not introduced a new defect elsewhere in the interface – for example, confirming that changes to the status-refresh logic for eSim did not affect the already-working detection of KiCad, Ngspice or Ngspice-adjacent components.

3.8 UI Testing

UI testing focused on the Quick Setup, Individual Tools and Bulk Uninstall tabs, specifically looking for layout defects introduced by Windows-specific rendering behaviour (font metrics, DPI scaling, and dark-mode style bleed). Overlapping text, clipped labels, and a non-functional Active Installations panel were all identified during this pass (see Bug 7 and related findings in Chapter 4).

3.9 Bug Reporting Process

Every defect identified was documented using a consistent template – description, expected behaviour, actual behaviour, impact, and severity – and, where the underlying cause could be inferred from black-box testing, a proposed root cause. This structure, applied consistently throughout, is what made the findings in Chapter 4 directly actionable for the development team, and is described further, alongside the GitHub workflow used to communicate it, in Chapter 5.

3.10 Verification of Fixes after Pull Requests

Once a Pull Request addressing a reported defect was merged into `feature/tool-manager-integration`, the corresponding bug was re-tested against the updated branch. A defect was only marked *Resolved* in this report once independent command-line verification (not just the Tool Manager's own GUI) confirmed that the reported status matched the true state of the system.

Chapter 4

Bug Findings and Improvements

This chapter consolidates every defect identified across both QA cycles into a single, de-duplicated list of eleven confirmed bugs. Each entry follows the same structure used during testing: description, expected behaviour, actual behaviour, probable root cause, user impact, and resolution status at the time of writing.

4.1 Bug 1: GHDL Detection and Version-Reporting Failure

Severity: High **Area:** Package Detection

Description	The Tool Manager fails to correctly detect GHDL, in two distinct ways observed across the two QA cycles: it was seen reporting GHDL as "Not Installed" despite a working GHDL binary on the system, and separately reporting an installed *version* (5.1.1) that did not match the version actually present (0.37, Dunoon edition).
Expected Behaviour	GHDL should be detected as installed, and the version shown in the Tool Manager should match the output of `ghdl --version` on the command line.
Actual Behaviour	The Individual Tool Manager either shows GHDL as not installed, or shows an installed version that disagrees with the command-line ground truth.
Root Cause	The detection logic likely queries the wrong location for GHDL's version string on Windows, or resolves `PATH` entries in a way that is inconsistent with how GHDL registers itself outside of a POSIX shell.
Impact	Users cannot trust the GHDL status shown in the Tool Manager, and may be misled into believing a newer or non-existent GHDL build is present.
Resolution Status	Reported; fix pending re-verification.

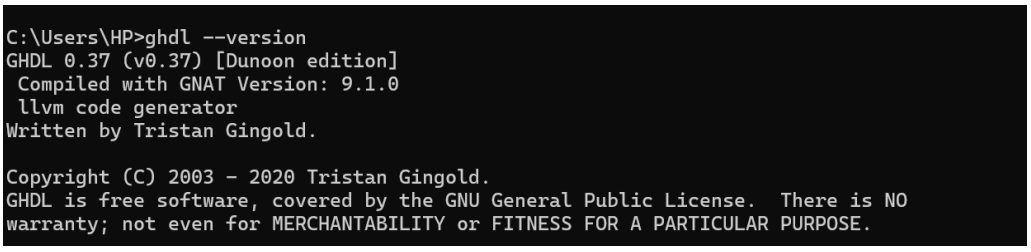

Figure 4.1: Command-line output confirming the actual installed GHDL version (0.37, Dunoon edition), used to cross-check the version reported by the Tool Manager.

4.2 Bug 2: False GHDL DLL Runtime Validation Error

Severity: High **Area:** Runtime Validation

Description	During GHDL runtime validation, the Tool Manager reports that `libgcc_s_seh-1.dll` was not found, even though GHDL executes correctly from the Command Prompt.
Expected Behaviour	Runtime validation should pass, since the required DLL is present and GHDL is functional outside the Tool Manager.
Actual Behaviour	A false-positive DLL error is raised inside the Tool Manager while the same binary runs without issue from a standalone terminal.
Root Cause	The Tool Manager likely invokes GHDL as a child process without inheriting the full system `PATH`, in particular the MinGW/MSYS2 runtime directories that provide the required DLL.
Impact	Users may be told installation or validation has failed when the tool is, in fact, fully functional, leading to unnecessary re-installation attempts.
Resolution Status	Reported to the development team; recommended fix is to ensure the child process inherits the full system `PATH` before validation is run.

4.3 Bug 3: Ambiguous and Stalling Installation Progress Reporting

Severity: Medium **Area:** Installation Workflow

Description	Analog Mode installation appears to stall for an extended period near completion, and ultimately finishes with a generic "finished with some issues" message rather than a clear success or failure outcome.
Expected Behaviour	Installation progress should advance smoothly to completion, with a clear, specific status message describing the outcome.
Actual Behaviour	Progress stalled at approximately 99.5% for roughly 15 minutes before completing, and the final message did not indicate which component(s), if any, had issues.
Root Cause	A long-running post-install step (such as library indexing or a finalisation task) is likely not reflected in the progress calculation, and the completion handler does not surface per-component outcomes.
Impact	Users may assume the installation has hung and forcibly terminate it, risking a partial or corrupted install; the vague completion message also undermines confidence in the tool.
Resolution Status	Reported; recommended fix is a granular sub-task progress indicator and a specific, per-component completion summary.

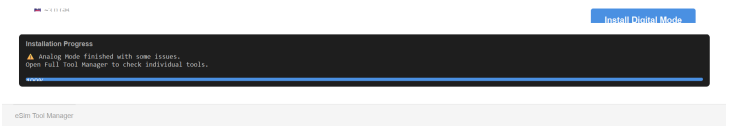

Figure 4.2: Installation Progress panel showing the generic "finished with some issues" completion message, with no indication of which component was affected.

4.4 Bug 4: Refresh Action and Available Tools Panel Do Not Reflect Live State

Severity: High **Area:** Status Synchronisation

Description	The top "Available Tools" status strip and the Refresh control both fail to update package status to reflect the true, current state of the system.
Expected Behaviour	Triggering Refresh should re-query the system and immediately update all displayed statuses; the Available Tools strip should always reflect live availability.
Actual Behaviour	Both the Refresh action and the Available Tools strip continue to display outdated status after an installation completes, and require a full application restart to show correct information.
Root Cause	There is likely no post-installation status-refresh callback wired to these two elements, or the refresh path reads from a cached state object that is never invalidated.
Impact	Users are shown outdated information and cannot rely on either control for an accurate, real-time status check, undermining confidence in the tool overall.
Resolution Status	Reported; recommended fix is to force a live re-query on every refresh and to invalidate cached state immediately after any installation event.

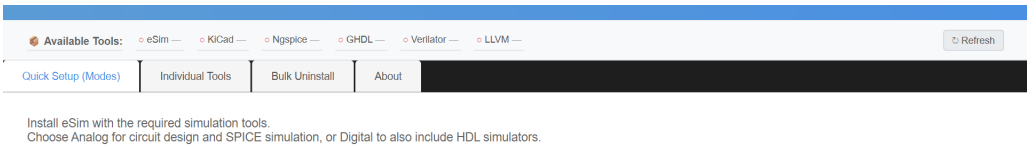

Figure 4.3: The "Available Tools" status strip, which was found not to update to reflect actual tool availability.

4.5 Bug 5: eSim Not Detected After a Successful Analog Installation

Severity: High **Area:** Package Detection

Description	Following a reported successful Analog Mode installation, the Individual Tool Manager continues to list eSim itself as "Not Installed."
Expected Behaviour	Once Analog Mode installation completes successfully, eSim should be listed as "Installed" in the Individual Tool Manager.
Actual Behaviour	eSim is shown as "Not Installed" despite the installer reporting success moments earlier.
Root Cause	eSim detection most likely relies on a registry key, install path, or executable name that differs from what the Windows installer actually creates, so the check silently fails.
Impact	This directly contradicts the reported installation outcome and is highly likely to cause users to attempt unnecessary reinstalls, or to distrust the Tool Manager's status reporting altogether.
Resolution Status	Reported; recommended fix is to align the eSim detection logic with the actual installation artefacts produced on Windows.

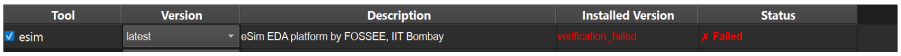

Figure 4.4: Individual Tool Manager row showing eSim as "Not Installed" despite a completed Analog Mode installation.

4.6 Bug 6: Verilator Not Detected After a Successful Digital Installation

Severity: High **Area:** Package Detection

Description	Verilator is shown as "Installed" in the Individual Tool Manager after a Digital Mode installation, but is not recognised at all from the command line.
Expected Behaviour	If the Tool Manager reports Verilator as installed, running `verilator --version` in the Windows Command Prompt should succeed.
Actual Behaviour	`verilator --version` fails with "is not recognized as an internal or external command."
Root Cause	Verilator's install location or executable name on Windows likely differs from the Linux-centric detection path used by the Tool Manager's status checks.
Impact	The tool is effectively unusable outside the application despite being marked installed, undermining trust in every other status the Tool Manager reports.
Resolution Status	Reported; recommended fix is a Windows-specific detection routine, e.g. using `where verilator` or checking the expected Windows install directory directly.

4.7 Bug 7: LLVM Reported Installed but Not Available on `PATH`

Severity: High **Area:** Package Detection

```
C:\Users\HP>verilator --version
'verilator' is not recognized as an internal or external command,
operable program or batch file.
```

Figure 4.5: Command Prompt confirming that Verilator is not actually available despite being marked "Installed" in the Tool Manager.

Description	LLVM is shown as "Installed" in the Advanced/Individual Tool Manager, but neither `llvm --version` nor `LLVM --version` resolves on the command line.
Expected Behaviour	If LLVM is reported installed, its executable should be discoverable and callable from the system command line.
Actual Behaviour	Both command variants fail with "is not recognized as an internal or external command."
Root Cause	The same class of PATH/registry-mismatch issue observed for Verilator and eSim appears to affect LLVM's detection logic as well.
Impact	The status panel does not reflect the real system state, compounding the pattern of untrustworthy status reporting seen across the Digital Mode toolchain.
Resolution Status	Reported; grouped with Bugs 5 and 6 for a unified Windows detection-logic audit.

```
C:\Users\HP>verilator --version
'verilator' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\HP>llvm --version
'llvm' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\HP>LLVM --version
'LLVM' is not recognized as an internal or external command,
operable program or batch file.
```

Figure 4.6: Command-line checks confirming that neither Verilator nor LLVM is actually detectable on the system, despite both being marked "Installed."

4.8 Bug 8: KiCad Reported Installed but Not Recognised on `PATH`

Severity: High **Area:** Package Detection

Description	The Tool Manager reports KiCad as installed, and the Advanced Tool Manager view lists a specific installed version, but the `kicad` executable cannot be verified from the command line.
Expected Behaviour	If the Tool Manager reports KiCad as installed, `kicad --version` should succeed from the Windows Command Prompt.
Actual Behaviour	Running `kicad --version` fails with "is not recognized as an internal or external command," even though the Tool Manager's own detection table lists KiCad as installed with a specific version.
Root Cause	Indicates the same detection/PATH/installation-validation inconsistency seen elsewhere: the detection layer is likely checking for an installer-recorded marker rather than confirming the executable is actually reachable.
Impact	The tool is unusable from outside the application despite being marked installed, and is particularly damaging to user trust because KiCad is core to eSim's schematic-capture workflow.
Resolution Status	Reported; recommended fix is to validate detected tools against an actual command execution, not just an installer-recorded marker.

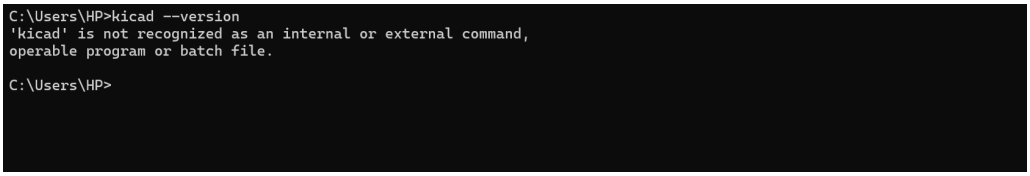

Figure 4.7: Windows Command Prompt failing to recognise the `kicad` command despite the Tool Manager reporting KiCad as installed.

Figure 4.8: Advanced Tool Manager view showing KiCad, Ngspice, GHDL and Verilator all marked "Installed," several of which were subsequently found not to be reachable from the command line.

4.9 Bug 9: GUI Rendering and Layout Corruption on Windows

Severity: High **Area:** GUI / Windows Compatibility

Description	The Quick Setup screen exhibits overlapping text and icons in the Analog and Digital Mode panels, and the installation log page shows clipped and overlapping UI elements during package installation.
Expected Behaviour	All UI elements – mode descriptions, checklists, size estimates, action buttons, and installation log text – should render in a clean, non-overlapping layout regardless of Windows display settings.
Actual Behaviour	Text and icons overlap in the Quick Setup tab, and the installation log page shows a Chocolatey error message overlapping nearby action buttons during a KiCad install attempt.
Root Cause	Fixed-size layouts are not accounting for Windows DPI scaling and font-metric differences relative to the platform the interface was originally designed against.
Impact	Reduces usability and readability, and gives an unpolished first impression that undermines confidence in the Tool Manager, particularly during error conditions when clear text matters most.
Resolution Status	Reported; recommended fix is a layout/responsive-design audit at 100%, 125% and 150% DPI, and replacing fixed pixel sizing with scaled units.

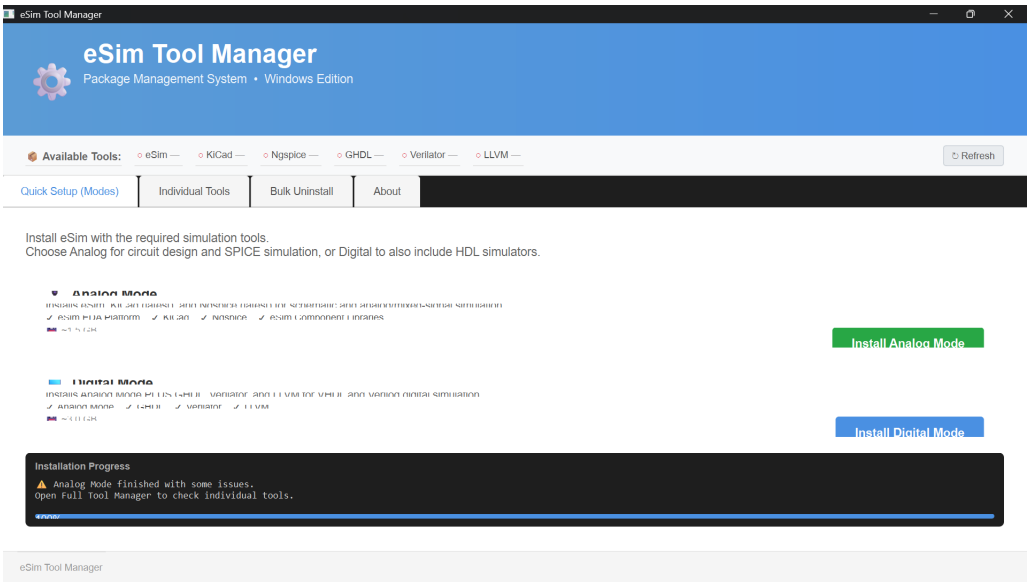

Figure 4.9: Quick Setup (Modes) tab showing overlapping text and UI elements in the Analog and Digital Mode panels.

Figure 4.10: Installation log showing a Chocolatey error together with overlapping and clipped UI elements during a KiCad installation attempt.

4.10 Bug 10: Active Installations Panel Does Not Function

Severity: Medium **Area:** GUI / Installation Tracking

Description	The Active Installations panel, intended to let users monitor ongoing installation tasks, does not update or otherwise function as expected.
Expected Behaviour	The panel should accurately reflect which tools currently have an installation in progress.
Actual Behaviour	The panel remains static and does not reflect ongoing installation activity.
Root Cause	The panel is likely not wired to the same progress-reporting events used elsewhere by the orchestration engine (see Chapter 2), or is reading from state that is never updated during an active install.
Impact	Users cannot monitor ongoing installation progress through this panel, reducing visibility into long-running operations.
Resolution Status	Reported; recommended fix is to connect the panel to the orchestration engine's existing progress events.

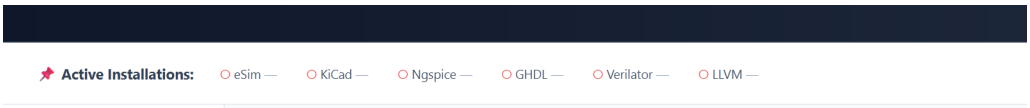

Figure 4.11: Active Installations panel, which does not update or function as expected during an installation.

4.11 Bug 11: Bulk Uninstall Instability

Severity: Medium **Area:** Uninstallation

Description	The Bulk Uninstall operation is unreliable when removing Analog Mode tools as a group.
Expected Behaviour	Bulk Uninstall should consistently remove all selected tools within a reasonable, predictable time frame.
Actual Behaviour	Bulk Uninstall occasionally fails to complete correctly, or takes significantly longer than expected and appears unresponsive during Analog Mode removal.
Root Cause	Likely missing timeout handling and retry logic around the underlying package-removal calls, with no progress feedback to distinguish a slow operation from a hung one.
Impact	Undermines confidence in cleanup operations and may leave the system in a partially uninstalled state, requiring manual intervention.
Resolution Status	Reported; recommended fix is to add timeout handling, retries, and progress feedback to the Bulk Uninstall operation.

4.12 Summary of Findings

Table 4.1 summarises all eleven confirmed defects. Package-detection failures dominate the list, reflecting the central architectural risk identified in Chapter 2: the orchestration and detection layers falling out of sync on Windows.

Table 4.1: Consolidated bug list from Windows QA testing of the eSim Tool Manager

ID	Title	Area	Severity
1	GHDL detection and version-reporting failure	Package Detection	High
2	False GHDL DLL runtime validation error	Runtime Validation	High
3	Ambiguous / stalling installation progress reporting	Installation Workflow	Medium
4	Refresh action and Available Tools panel out of sync	Status Synchronisation	High
5	eSim not detected after successful Analog install	Package Detection	High
6	Verilator not detected after Digital install	Package Detection	High
7	LLVM reported installed but not on PATH	Package Detection	High
8	KiCad reported installed but not on PATH	Package Detection	High
9	GUI rendering and layout corruption on Windows	GUI / Compatibility	High
10	Active Installations panel non-functional	GUI / Tracking	Medium
11	Bulk Uninstall instability	Uninstallation	Medium

Chapter 5

Collaborative Development

5.1 Overview

Although this report is scoped to my own contribution as the fellowship's Windows QA tester, none of that testing happened in isolation. The Tool Manager was, and remains, a collaboratively developed component, and the workflow described in this chapter reflects how QA findings were fed back into development, reviewed, and closed out as a team.

5.2 GitHub Workflow

All work on the Tool Manager was tracked against the `feature/tool-manager-integration` branch in the eSim GitHub repository. My own workflow, from finding a defect to seeing it resolved, followed the cycle shown in Figure 5.1.

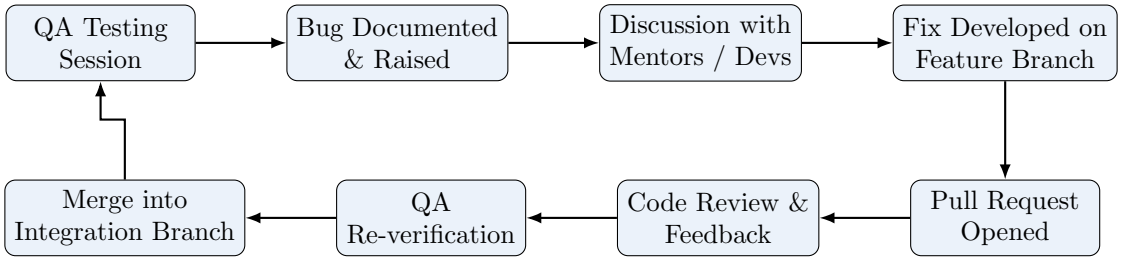

Figure 5.1: The collaborative GitHub-based cycle followed for each defect: testing, reporting, discussion, fix, review, QA re-verification, and merge – feeding back into further testing.

5.3 Pull Requests and Code Review

Every fix related to a bug documented in Chapter 4 was implemented on its own branch and submitted as a Pull Request against `feature/tool-manager-integration`, rather than being pushed directly. This gave the wider team, including myself as the reporting tester, the opportunity to review the proposed change before it was merged – checking not just whether the fix compiled and ran, but whether it addressed the root cause identified during testing rather than only the symptom observed in the GUI.

5.4 Team Collaboration

Because the Tool Manager's installation and detection logic was implemented jointly by the fellowship team, close collaboration with my teammate, Mausam Kar, was essential to producing bug reports that were actually useful to them. In several cases, an initial black-box observation (for example, "eSim shows Not Installed") was refined together with the implementers into a more specific root-cause hypothesis (for example, "the detection check is looking for an install marker that the Windows installer does not create") before a fix was attempted. This report deliberately avoids claiming ownership over that implementation work; my contribution was in

surfacing, characterising, and verifying these issues, not in writing the underlying detection or installation code myself.

5.5 Verification After Fixes

Once a Pull Request was merged, I re-tested the corresponding scenario independently, using the same command-line cross-checks described in Chapter 3, before considering a bug closed. This step was treated as non-negotiable: a fix was not considered complete on the basis of a code review alone, but only once the Tool Manager's reported status could be shown to match the actual state of the system again.

5.6 Working with Mentors

Regular check-ins with Dr. Kiran Talele and the FOSSEE Tool Manager mentors provided an opportunity to prioritise which defects to tackle first – in practice, the High-severity package-detection failures documented in Chapter 4 were consistently treated as blocking issues, ahead of the lower-severity GUI and workflow-messaging defects.

Chapter 6

Learning Outcomes

6.1 Technical Learning

The fellowship deepened my practical understanding of software testing well beyond what classroom exposure typically provides. Working through real, reproducible defects on the Tool Manager taught me to distrust a GUI's own claims about system state and to always corroborate them against an independent source of truth – in this case, the Windows command line. This habit of independent verification, formed while testing GHDL, KiCad, Verilator and LLVM detection, is a QA principle I expect to carry into future work well beyond this project.

6.2 Git and GitHub Workflow

I gained hands-on experience with a professional, branch-and-review-based Git workflow: working against a shared integration branch, raising and discussing issues before a fix was attempted, and treating a Pull Request merge as the trigger for re-verification rather than the end of the process.

6.3 QA Methodology

I learned to structure defect reports so that they are immediately actionable – separating what was *expected*, what was *observed*, a plausible *root cause*, and the resulting *impact* – and to scale a testing effort across multiple cycles (an initial functional pass, followed by a targeted regression pass) rather than treating testing as a single, one-off activity.

6.4 Debugging and Root-Cause Analysis

Although I was not implementing the Tool Manager's detection logic myself, formulating credible root causes for each defect (for example, distinguishing a PATH-inheritance issue from a stale-cache issue when both can produce similar symptoms) required me to reason carefully about how a PyQt application on Windows interacts with child processes, environment variables, and the registry.

6.5 Windows Package Management

Testing installation workflows built on Chocolatey gave me practical exposure to Windows package management – and to the specific ways in which tools designed with Linux-first assumptions (fixed executable names, PATH conventions, DLL dependencies) can behave differently once installed on Windows.

6.6 Professional Documentation

Producing bug reports and this consolidated report itself reinforced the value of clear, structured technical writing: a defect that is well-documented, with a screenshot and a precise expected-

versus- actual description, is dramatically faster for a development team to act on than an informal description of "it's broken."

6.7 Open-Source Contribution Experience

Beyond the technical skills, the fellowship gave me a genuine sense of what it means to contribute to an open-source project used by students and educators well beyond my own institution – including the responsibility that comes with reporting a defect accurately, and the satisfaction of seeing a fix I helped surface land in the codebase.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Over the course of the eSim Summer Fellowship, I conducted structured Windows QA on the eSim Tool Manager across two testing cycles, identifying and documenting eleven distinct defects spanning package detection, status synchronisation, installation-progress reporting, GUI rendering, and uninstallation reliability. The consistent theme across these findings – summarised in Chapter 4 – was that the Tool Manager's core installation workflow functioned correctly, while the layer responsible for reporting status back to the user was frequently out of sync with the true state of the system on Windows. By pairing every GUI observation with an independent command-line verification, I was able to provide the development team with precise, reproducible evidence for each defect, several of which were prioritised and resolved through the Pull Request process described in Chapter 5.

This fellowship experience reinforced the value of rigorous, independently verified QA in an open-source setting, and gave me direct, hands-on exposure to collaborative software development practices – branch-based workflows, code review, and regression testing – that I expect to apply well beyond this project.

7.2 Future Work

Based on the patterns observed during testing, I would recommend the following areas for continued work on the Tool Manager:

- **Automated regression suite.** Many of the defects documented in this report (package detection returning stale or incorrect results) would be well suited to an automated test suite that runs the detection logic against a set of known-good and known-bad system states, rather than relying solely on manual QA cycles.
- **Unified Windows detection strategy.** Given that GHDL, Verilator, LLVM, KiCad and eSim itself all exhibited variants of the same underlying problem, a single, well-tested Windows detection utility (checking `PATH` and running each tool's version command directly) could replace several bespoke, inconsistent detection paths.
- **Continuous cross-platform testing.** Extending this QA effort to cover macOS, alongside the existing Windows and Linux testing, would help catch platform-specific assumptions earlier in development.
- **DPI-aware layout testing as a standard CI check.** Since several GUI defects were specific to Windows DPI scaling, incorporating layout screenshots at multiple DPI settings into the review process could catch these regressions before merge.

I look forward to continuing to contribute to the eSim project, and to applying the QA methodology developed during this fellowship to future open-source work.

Bibliography

[1] FOSSEE, IIT Bombay. *eSim Official Website*. https://esim.fossee.in/

[2] FOSSEE, IIT Bombay. *eSim GitHub Repository*. https://github.com/FOSSEE/eSim

[3] KiCad EDA. *KiCad Documentation*. https://docs.kicad.org/

[4] Ngspice. *Ngspice User's Manual*. https://ngspice.sourceforge.io/docs.html

[5] GHDL Project. *GHDL Documentation*. https://ghdl.github.io/ghdl/

[6] Veripool. *Verilator Documentation*. https://verilator.org/guide/latest/

[7] Chocolatey Software. *Chocolatey Package Manager Documentation*. https://docs.chocolatey.org/

[8] FOSSEE, IIT Bombay. *FOSSEE Fellowship Programme*. https://fossee.in/

[9] R. Paknikar, S. Bansode, G. Nandihal, M. P. Desai, K. M. Moudgalya, and A. Jha, "eSim: An Open Source EDA Tool for Mixed-Signal and Microcontroller Simulations," *2021 4th International Conference on Circuits, Systems and Simulation (ICCSS)*, Kuala Lumpur, Malaysia, 2021, pp. 212–217.