eSim Summer Fellowship 2026

Internship Report on

Engineering eSim 2.6: A System-Wide Modernization of Simulation, Mixed-Signal Workflows, User Interface, Reliability, and Cross-Platform Delivery

Submitted by

Varadharam R S
Electrical and Computer Engineering
Shiv Nadar University IOE, Delhi-NCR

Under the guidance of

Prof. Prabhu Ramachandran
Principal Investigator, FOSSEE
Department of Aerospace Engineering, IIT Bombay

With mentorship from

Mr. Sumanto Kar
Assistant Project Manager, FOSSEE, IIT Bombay

August 2026

Acknowledgment

I thank **Prof. Prabhu Ramachandran** for the opportunity to return to **FOSSEE, IIT Bombay** and contribute to its open-source engineering work. His leadership gives student developers the unusual chance to work on software that is used beyond the duration of an internship. I am also grateful to **Prof. Kannan M. Moudgalya** for the educational and open-source vision on which FOSSEE and eSim are built.

My sincere thanks go to my mentor, **Mr. Sumanto Kar**, Assistant Project Manager, FOSSEE, for his guidance throughout the fellowship. His knowledge of eSim, careful reviews, and practical advice helped me understand the wider implications of individual engineering decisions. I am grateful for the time he devoted to technical discussions and for the trust he placed in me to take responsibility for work across the application.

I am sincerely grateful to **Mr. Varad Patil** for his guidance and support throughout the fellowship. His regular presence in the lab and involvement in coordinating its activities helped sustain a productive working environment for the interns. I particularly appreciate his attention to the difficulties interns encountered and his efforts to bring people together to resolve them. His support contributed meaningfully to my experience at FOSSEE.

I also thank **Ms. Shanthi Priya K** for her support during the programme and for helping maintain a welcoming environment for the interns.

Returning for a second internship was personally important. During my first fellowship, I learned how to enter a mature codebase and complete a focused plotting refactor. This time I could take responsibility for work across simulation, interface architecture, HDL workflows, installers, testing, and release engineering. I am grateful for the chance to see how much more I could contribute after one year of experience.

I thank **Akshat Sharma** for the ideas we exchanged and for the parts of the interface and mixed-signal work that we explored together. His curiosity made our discussions enjoyable and useful. I am equally grateful to the other interns whose questions, experiments, and encouragement made the fellowship a richer experience.

Abstract

This report documents the engineering work completed for eSim 2.6, an open-source electronic design automation suite developed by FOSSEE at IIT Bombay. The internship began with production issues in the plotting subsystem created during my previous fellowship. Following those issues through the application exposed related problems in configuration, project identity, process ownership, PyQt6 migration, HDL workflows, netlisting, installation, and release packaging. The scope therefore expanded from one window to the complete path by which a design is created, translated, simulated, inspected, and delivered to a user.

The resulting work stabilizes waveform extraction and rendering, completes the PyQt6 migration, introduces a shared interface framework, improves project recovery and editing, and adds integrated Verilog verification and Icarus-based digital co-simulation. It also hardens NGHDL and Makerchip integration, replaces the fragile KiCad export path, prepares and verifies SKY130 assets, and produces reproducible Ubuntu and Windows distributions. Across these areas, implicit paths and duplicated state were replaced with explicit owners, typed data boundaries, background jobs, focused tests, and build procedures that can be repeated on a clean machine.

The report is organized by engineering problem rather than by development chronology. Each chapter starts from an observable failure or maintenance risk, identifies the responsible code boundary, explains the implemented design, and presents evidence from tests, generated artifacts, waveform comparisons, or installer verification. The aim is to show how the individual changes combine into a dependable release while keeping the reasoning open to technical review.

Keywords: eSim, FOSSEE, PyQt6, KiCad, Ngspice, Icarus Verilog, GHDL, mixed-signal co-simulation, reproducible packaging, electronic design automation.

Contents

Acknowledgment i

Abstract ii

1 The Mission 1
- 1.1 Returning to a production codebase . . . 1
- 1.2 What eSim coordinates . . . 1
- 1.3 How the scope expanded . . . 2
- 1.4 Method and standard of proof . . . 3
- 1.5 Scale, without using scale as proof . . . 3
- 1.6 How to read the report . . . 4

I Stabilizing the Core 5

2 Hardening the Plotting System 6
- 2.1 Starting point . . . 6
- 2.2 Failures found after integration . . . 6
- 2.3 Parsing Ngspice output . . . 7
- 2.4 Architecture after the rewrite . . . 8
- 2.5 Three views over one dataset . . . 9
- 2.6 Interaction and performance . . . 11
- 2.7 Validation . . . 11

3 Completing the PyQt6 Migration 13
- 3.1 Why a successful launch was insufficient . . . 13
- 3.2 The semantic changes that mattered . . . 13
- 3.3 One owner for every running process . . . 15
- 3.4 Dialogs and exceptions return to the GUI thread . . . 15
- 3.5 Shutdown is an ordered operation . . . 16
- 3.6 Evidence and regression boundaries . . . 16

II Modernizing the Experience 18

4 Aurora: A Design System for eSim 19
- 4.1 Design goals from the application, not a mock-up . . . 19
- 4.2 One visual vocabulary, several renderers . . . 19
- 4.3 A home screen that uses the real actions . . . 21

4.4 Live theme changes are a transaction . . . 22
4.5 Preferences are live and reversible . . . 23
4.6 Scaling, motion, and custom surfaces . . . 24
4.7 Working tools, not only the main window . . . 25
4.7.1 The Subcircuit Builder owns a visible selection . . . 26
4.7.2 The Model Editor uses the whole dock and protects table edits 27
4.8 Regression coverage . . . 29

5 Project Identity and Lifecycle 30
5.1 When a folder name became identity . . . 30
5.2 One project identity contract . . . 30
5.3 Snapshots with an undoable restore . . . 32
5.4 State belongs to the user, the project, or the process . . . 33
5.5 Asynchronous work retains its origin . . . 34
5.6 Conversion as a state machine . . . 34

6 A Built-in Code Editor 37
6.1 Why editing belongs inside eSim . . . 37
6.2 Ownership from project row to file bytes . . . 37
6.3 A SPICE editor that understands line roles . . . 38
6.4 Language selection is explicit . . . 39
6.5 Saving and external changes . . . 41

III New Capabilities 43

7 Maker and the HDL Workflow 44
7.1 Why the workflow needed an architectural change . . . 44
7.2 One navigator, two language paths . . . 44
7.3 A design that has a safe home . . . 46
7.4 Makerchip without surrendering file ownership . . . 47
7.5 Removal is part of model creation . . . 47

8 Verilog Verification Inside eSim 49
8.1 The missing step between source and model . . . 49
8.2 A layered implementation . . . 49
8.3 The verification workspace . . . 50
8.4 Structural parsing and testbench ownership . . . 51
8.5 From Simulate to a waveform . . . 51
8.6 Failure containment and lifecycle . . . 53

9 Digital Co-simulation 55
9.1 The user-facing path . . . 55
9.2 From source to one mixed-signal run . . . 56
9.2.1 Model creation . . . 56

9.2.2 Schematic conversion 56
9.3 The one-engine constraint 57
9.4 Correctness at the analog–digital boundary 58
9.4.1 A ramp must produce one digital edge 58
9.4.2 Time precision and the operating point 58
9.5 Toolchain resolution and run control 59
9.6 End-to-end validation 60

10 NGHDL and Makerchip Integration **62**
10.1 Two integrations, two responsibilities 62
10.2 How one VHDL entity becomes a circuit block 63
10.2.1 Port information is a protocol contract 63
10.2.2 Generated files and installed ownership 64
10.3 The live socket exchange 64
10.4 Selecting a GHDL backend by capability 65
10.5 Makerchip as a revisioned external editor 67
10.6 Backend boundaries in the shared workspace 68

11 KiCad 7–10 Netlisting **70**
11.1 Why the old export stopped being usable 70
11.2 A structured boundary instead of a text patch 71
11.2.1 Electrical identity 71
11.2.2 Component semantics 71
11.3 Failure must preserve the last usable circuit 72
11.4 The conversion workspace after netlisting 73
11.5 How the compatibility claim was tested 75
11.6 Practical project behaviour 76

IV Shipping eSim 2.6 **78**

12 Ubuntu 26.04 Installation **79**
12.1 From release scripts to installation profiles 79
12.2 One action list, eleven ordered steps 80
12.3 The installer interface under a safe terminal demo 80
12.4 The one-line bootstrap has narrow ownership 81
12.5 Mandatory, optional, and advisory outcomes 82
12.6 Clean-room equivalence audit 83

13 Reproducible Windows Packaging **85**
13.1 The packaging problem was reproducibility 85
13.2 One command defines the product 85
13.2.1 Pinned inputs and deliberate exceptions 86
13.2.2 The stage is a product, not a copied checkout 87
13.3 A real installer, not a zip with a shortcut 88

13.3.1 Bundled KiCad without taking over the machine 88
13.4 Machine payload and user state 89
13.5 Windows faults became permanent gates 89
13.6 Result 90

14 PDK and Toolchain Closure **91**
14.1 Closure begins after file existence 91
14.2 SKY130: from archive to electrical evidence 91
14.3 IHP: enabling only the models that exist 92
14.4 One capability model for every simulation flow 93
14.4.1 Dependency subsets prevent false blocking 95
14.5 KiCad registration with explicit ownership 95
14.6 Release closure as a test ladder 97

15 Performance Engineering **99**
15.1 A performance contract for the desktop 99
15.2 Startup: first frame before optional tools 100
15.3 Responsiveness: work outside the paint loop 100
15.4 Rendering: less screen work, complete evidence 101
15.5 Evidence and observed gains 102

16 Conclusion and Future Work **104**
16.1 From isolated fixes to one maintained system 104
16.2 What eSim 2.6 now guarantees 105
16.3 Impact for users and maintainers 106
16.4 Next engineering boundaries 106
16.5 Closing perspective 107

A Work Index and Metrics **108**
A.1 Measured scope 108
A.2 Engineering work index 108
A.3 How the history was made reviewable 110
A.4 Reproduction and evidence map 110
A.5 Companion audit artifacts 112

List of Figures **113**

List of Tables **116**

Bibliography **118**

Chapter 1

The Mission

I returned to FOSSEE to finish the production work around eSim's waveform viewer. The first failures led beyond plotting into process management, project state, interface architecture, mixed-signal backends, and distribution. This chapter explains why the internship grew into a system-wide modernization and how the work is evaluated in the chapters that follow.

1.1 Returning to a production codebase

My 2025 fellowship focused on replacing eSim's legacy waveform viewer. The refactor was successful on the circuits available during development and was later integrated into the main codebase. Integration changed the standard of proof. The plotter now had to handle the full example library, output produced by different Ngspice versions, long node names, mixed analog and digital traces, high-DPI displays, theme changes, and projects whose paths contained spaces.

The first 2026 tasks were therefore corrective. They recovered complete signal names, separated numeric extraction from the interface, repaired stacked-pane geometry, and made the canvas respond safely to theme and display changes. These failures were not isolated. The plotter received data from simulation processes; those processes relied on configuration and project identity; every window inherited assumptions from the unfinished PyQt6 migration; and none of the improvements would reach a student unless the Ubuntu and Windows packages reproduced the same environment.

This changed the assignment from a plotter follow-up into one question:

How can eSim's design-to-result path be made explicit, testable, and reproducible without breaking the circuit workflows that users already depend on?

1.2 What eSim coordinates

eSim is an orchestrator rather than a single simulator. A project may move through KiCad, a netlist translator, Ngspice, Icarus Verilog, GHDL, Verilator, model libraries, and the waveform viewer. The desktop application must preserve identity and state while those tools create files and subprocesses outside Qt's event loop. Release packages must then install compatible versions of the same tools.

Figure 1.1 shows the runtime path used as the system model for this work. The solid arrows are data or control flow. The outer delivery boundary is equally

important: source code, simulator binaries, models, and launch configuration must arrive together for the inner path to work on a clean machine.

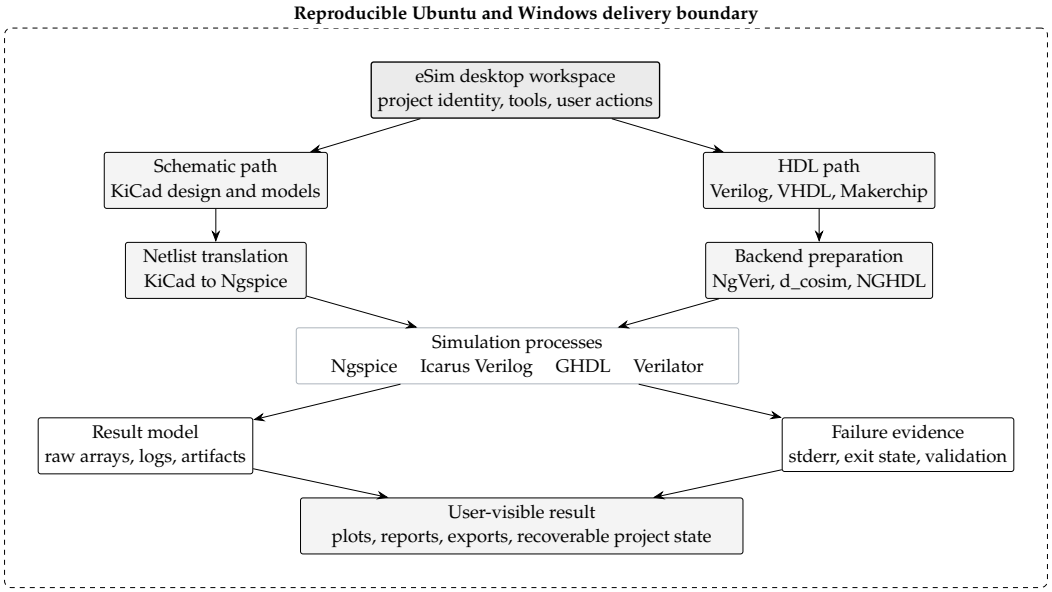


Figure 1.1: The eSim design-to-result path used during the modernization. A defect in a translation, process, or packaging boundary can surface several stages later, so each stage needs an explicit contract and observable failure state.

1.3 How the scope expanded

The work was organized around four engineering programmes. They are presented here as responsibility areas, not as a development timeline.

Table 1.1: The four programmes and the user-facing condition each one had to establish.

Programme	Main code boundaries	Required user outcome
Core reliability	Plot extraction and rendering, PyQt6 lifetimes, configuration, dialogs, process ownership	Simulations start, stop, report errors, and remain usable through theme, scale, and window-state changes.
Application experience	Shared visual framework, projects, snapshots, editor, docks, fullscreen tools	The same project identity and interaction rules hold across tools, instead of each window maintaining its own conventions.
Mixed-signal capability	Verilog Verifier, Icarus `d_cosim`, NgVeri, NGHDL, Makerchip, KiCad netlisting	Analog and HDL elements can be prepared, simulated, and inspected through one understandable workflow.
Release engineering	Ubuntu installer, native Windows distribution, SKY130 assets, toolchain audit, documentation	A clean machine can install and run the same product that was tested in the development tree.

Several examples explain why the boundaries cannot be treated separately. A KiCad field change may alter the SPICE line consumed by Ngspice. A simulator line-width setting may truncate the node name displayed by the plotter. Selecting an incompatible GHDL backend may appear to the user as a socket failure. Omitting one runtime file from an installer can make correct source code impossible to launch. The work therefore followed faults to the layer that owned them, even when that layer belonged to a later part of the application.

1.4 Method and standard of proof

Screenshots can confirm a visible state. They cannot establish parser correctness, electrical equivalence, process cleanup, or installability. Evidence therefore had to match the risk. Figure 1.2 summarizes the method used across the report.

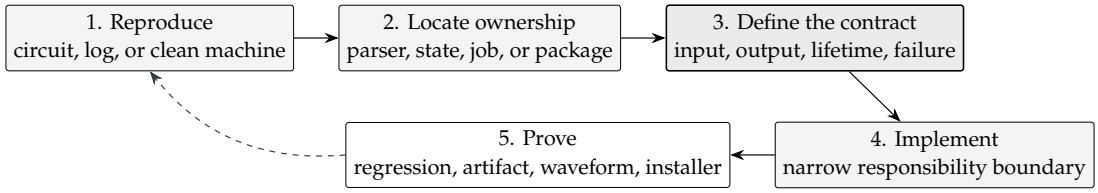


Figure 1.2: The engineering proof loop used in this report. The dashed return path means that failed evidence reopens the observed boundary. Parsers receive fixture tests, translators receive golden outputs, backends receive waveform checks, and release work receives clean-machine runs.

This approach produced several kinds of evidence:

- focused tests for parser states, identities, dialogs, jobs, and Qt lifetimes;
- topology and waveform comparisons for netlisting and mixed-signal backends;
- electrical smoke tests for packaged PDK assets;
- installer self-checks and launch tests on clean Ubuntu and Windows environments;
- final-tree comparison after the development history was consolidated for review.

1.5 Scale, without using scale as proof

The audited history contains 265 commits relative to the selected upstream base. The final repository delta covers 416 files, with 97,336 insertions and 27,912 deletions across implementation, tests, documentation, packaging, data, and removal of obsolete generated material. These figures establish the breadth of the work. They do not establish whether a circuit is correct or an installer works, which is why later chapters use boundary-specific evidence instead of relying on activity counts.

Table 1.2: Audited scale and release evidence for the completed work.

Measure	Result
Audited development history	265 commits: 262 non-merge changes and 3 merges
Final repository delta	416 files; 97,336 insertions and 27,912 deletions
Review structure	11 dependency-ordered logical changes with an identical final tree
Changed automated-test surface	114 `test_*.py` files; 126 files in test directories
Recorded Windows checkout run	1,235 passed and 26 skipped in the earlier full-suite audit; not rerun for this source refresh
Release targets	Ubuntu archive and native Windows 10/11 installer

1.6 How to read the report

The report follows the product rather than the commit sequence. Each chapter starts with the failure or maintenance risk, traces it to the responsible code, explains the replacement design, and records the validation used to close the work. The separate ledger preserves the full development history. The main report keeps commit identifiers out of the technical narrative so that the reasoning can be read as a connected system.

Part I

Stabilizing the Core

Chapter 2

Hardening the Plotting System

The plotting module from my previous fellowship became the starting point for this internship. Once it was integrated and exercised with eSim's wider example set, failures appeared in data extraction, redraw behavior, digital timing, cursor handling, theme changes, and export. Resolving them required more than visual cleanup: the parser, data ownership, rendering paths, and interaction model were all reworked and tested as a production subsystem.

2.1 Starting point

eSim 2.5 used a monolithic waveform viewer in `src/ngspiceSimulation/python Plotting.py`. One `QMainWindow` owned the parser, trace list, controls, matplotlib canvas, and export path. All selected signals were placed on a single axes. Voltage and current traces could therefore share a label that described only one of them, and node names longer than fifteen characters were truncated before they reached the user.

The 2025 refactor introduced the modern viewer, but integration exposed cases that the original development circuits did not cover. Different Ngspice versions emitted repeated headers and column groups. Large transient runs made a full figure rebuild visible on every selection change. Live theme and DPI changes invalidated assumptions made when the canvas was first created. Mixed-signal examples also demanded clearer timing and measurement tools than a single overlaid axes could provide.

2.2 Failures found after integration

The follow-up used eleven eSim example circuits spanning transient, AC, and DC analyses. Each issue was reproduced from simulator output, traced to its owner, and then covered at the parser, model, renderer, or interface boundary. Table 2.1 groups the main failures. The count is useful only as context; the important result is that related symptoms were closed by one responsible design change.

Table 2.1: Defect groups addressed during plotting production hardening.

Boundary	Verified issues	Representative cause
General rendering	12	Visibility, zoom, legend, and input paths could leave the trace model and matplotlib artists in different states.
Digital timing	9	Mid-rail samples were classified inconsistently, traces used the wrong normalization reference, and the view lacked useful level and frequency annotations.
Cursor measurement	4	Cursor artists were destroyed by redraws, dragging required expensive full-canvas work, and coordinate handling failed after some axis transforms.
Data extraction	Parser rewrite	Column groups, repeated page headers, AC complex fields, malformed rows, and long or duplicate node names were interpreted incorrectly.
Process and input safety	3	Simulator errors could be hidden, paths with spaces could break invocation, and function traces used unrestricted Python evaluation.

2.3 Parsing Ngspice output

The files `plot_data_v.txt` and `plot_data_i.txt` are formatted reports, not stable rectangular tables. A long run may repeat its header at a page boundary. A new group may begin with an asterisk marker or simply with different column names. AC rows contain real and imaginary fields, while a malformed row may have fewer fields than the header. Treating every line as numeric data cannot distinguish these cases.

The replacement in `data_extraction.py` performs two passes. The first pass is a state machine that records headers and groups without converting numbers. The second pass sends each complete group to `numpy.loadtxt`. The first valid group owns the shared time or frequency axis; later groups contribute only signal arrays. Identical headers continue the current group after a page break, while changed headers open a new group. Figure 2.1 shows the actual classification order.

The parser core is independent of the plot widgets and is exercised with synthetic Ngspice fixtures. Tests cover transient blocks, AC fields, ragged rows, raw-file node name recovery, event-stream resampling, missing files, and complete project round trips. Vectorized conversion measured 2.4 times faster than the previous Python float loop on large transient data.

Function traces received a separate safety correction. Signal names such as `v(out)` are first replaced with controlled identifiers. The expression is parsed as an

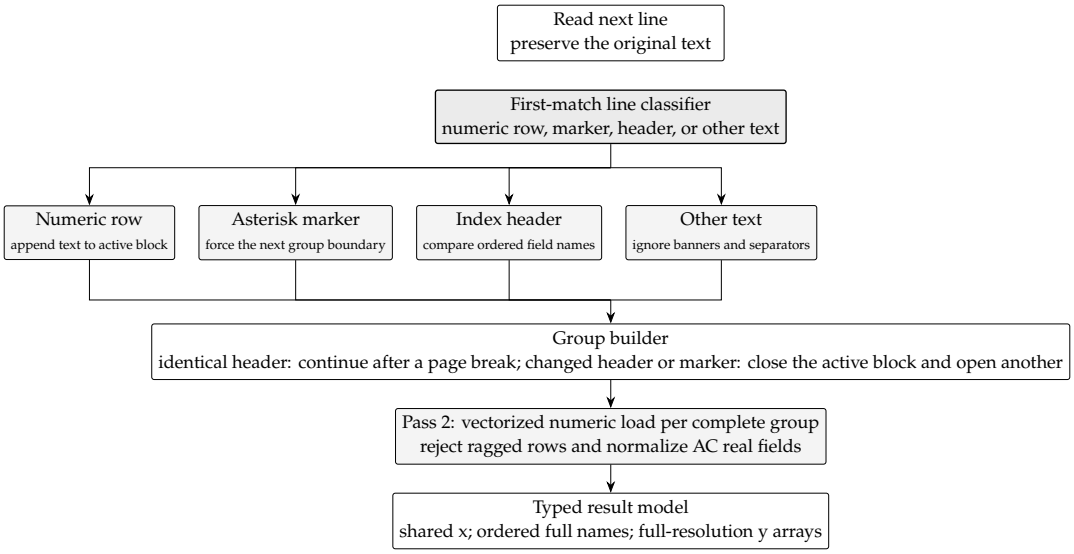


Figure 2.1: The two-pass extraction path in `data_extraction.py`. The first pass classifies text and establishes group boundaries; only complete groups reach numerical conversion. Repeated headers therefore mean page continuation, while changed headers create a new signal group.

abstract syntax tree and evaluated against a small set of arithmetic operators and numerical functions. Attribute access, unknown identifiers, and unapproved calls are rejected. This removes unrestricted `eval()` while retaining useful expressions such as `v(in)-v(out)`.

2.4 Architecture after the rewrite

Correctness work caused `plot_window.py` to grow to about 2,600 lines. The file was then decomposed by responsibility. The remaining window creates widgets and owns session state; five mixins implement rendering, panes, cursors, the waveform list, and derived function traces. Data extraction, trace models, palette selection, and numeric utilities remain usable without constructing the full interface.

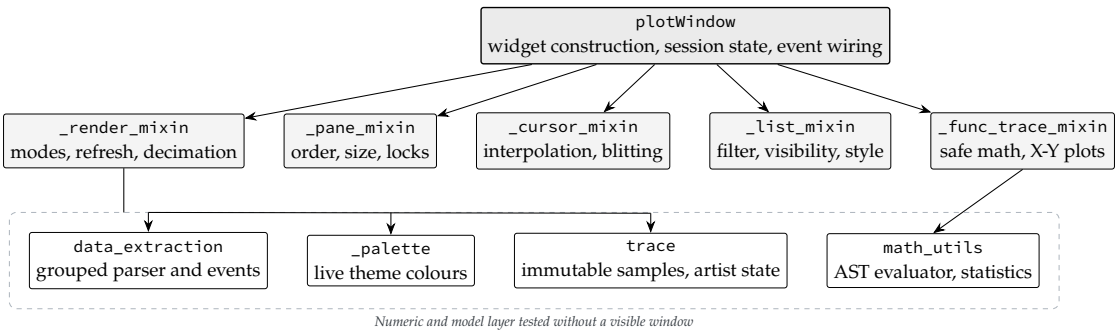


Figure 2.2: The current plotting architecture. The window remains the state owner, while interaction responsibilities are separated into mixins and numerical work is kept in small modules with direct tests.

This structure also makes state transitions inspectable. The trace model owns name, full-resolution samples, visibility, colour, line style, and its current matplotlib artist. The pane mixin owns order, height, and per-pane locks. The renderer owns the composition signature that determines whether an event can update existing artists or must rebuild the figure. Cursor code can therefore move measurement artists without changing trace or pane state.

2.5 Three views over one dataset

The viewer provides three presentations of the same arrays. Standard view overlays selected traces and is useful for analog relationships. Stacked view gives each signal an aligned pane with its own scale and statistics. Digital timing view converts samples to logic levels using the selected threshold and lays them out as a timing diagram. Changing views does not reparse the simulation or overwrite the samples.

Figures 2.3 to 2.5 were captured specifically for this report from the production eSim 2.6 `plotWindow`. The input is one deterministic transient containing a clock, reset, two inputs, three digital outputs labelled by their mixed-signal backends, and an analog RC response. Keeping the dataset fixed makes the different presentation contracts directly comparable.

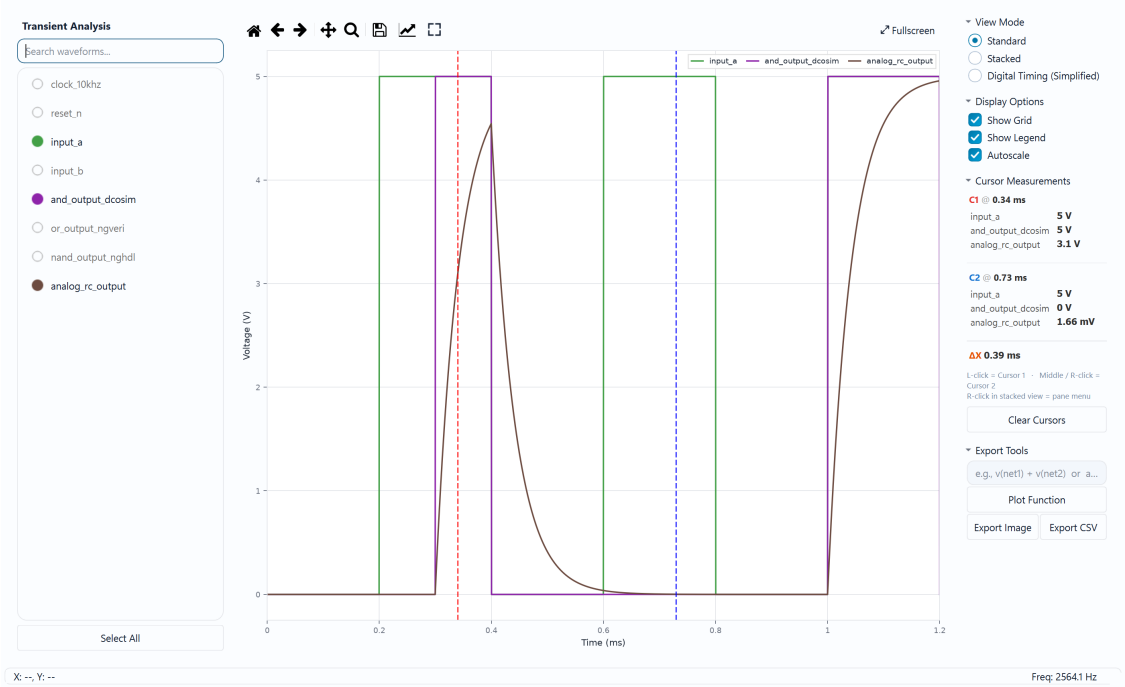


Figure 2.3: Standard view in the current eSim 2.6 interface. Three signals are overlaid while the complete waveform inventory remains available on the left. The measurement panel reports both cursor positions, interpolated values, and the time difference.

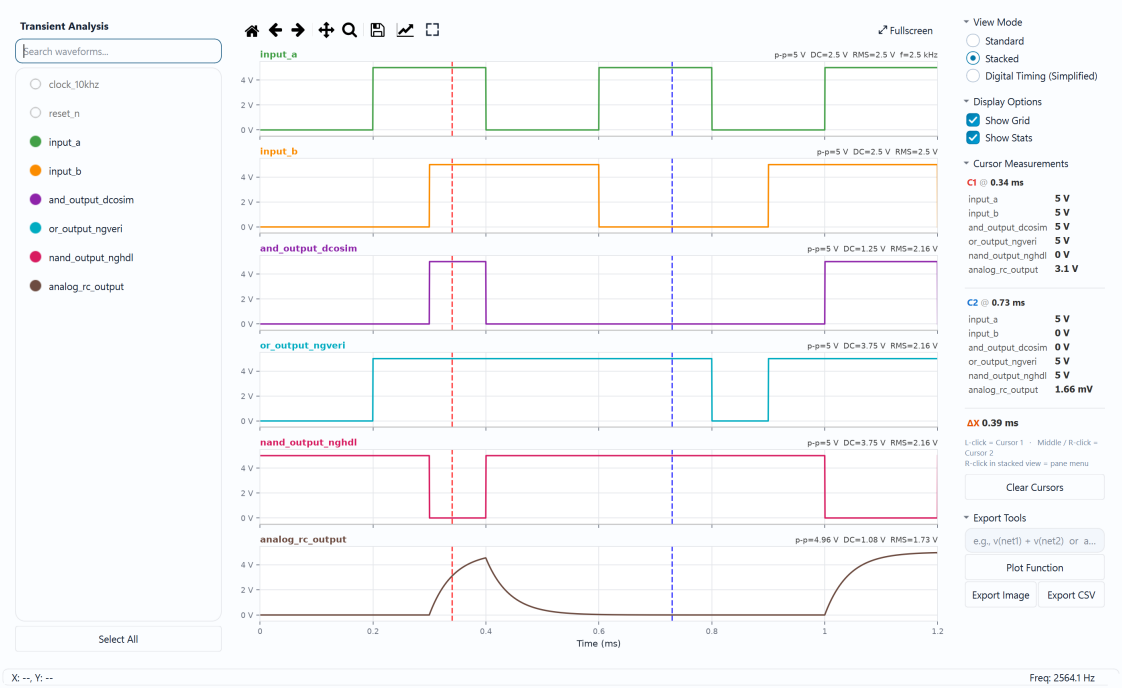


Figure 2.4: Stacked view of the same transient. Independent voltage scales make the analog response readable beside digital signals, while shared time, aligned cursors, pane titles, and peak-to-peak, DC, RMS, and frequency statistics preserve comparison.

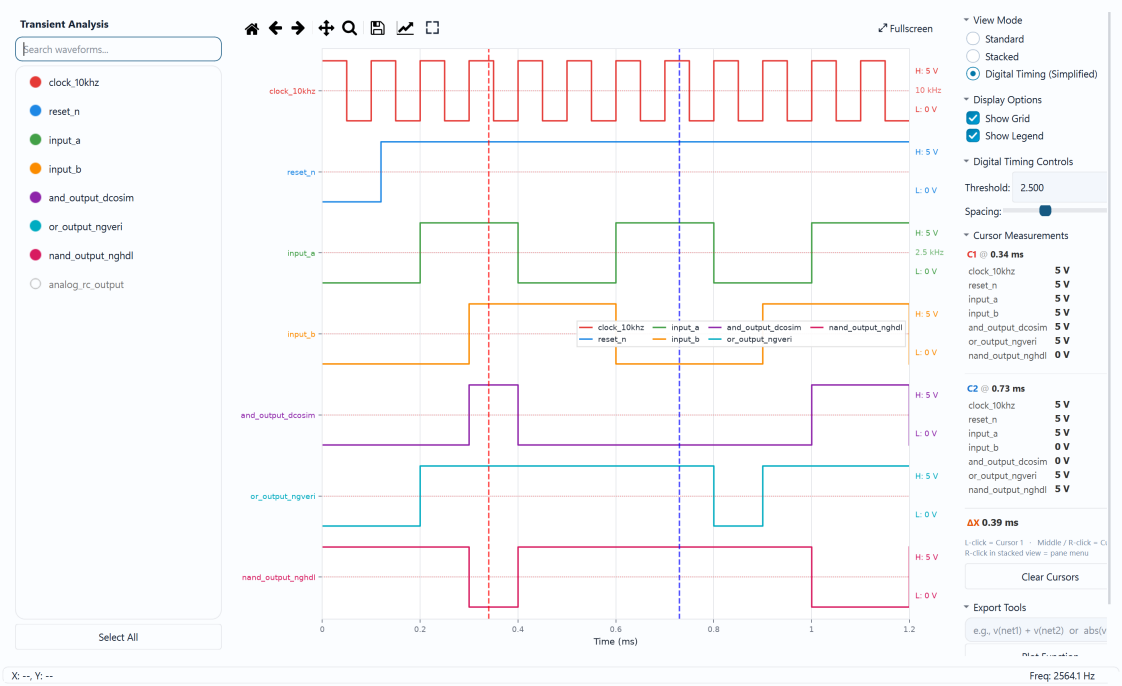


Figure 2.5: Digital timing view derived from the same source arrays at a 2.5 V threshold. Level and frequency annotations, adjustable row spacing, a shared legend, and cursor readouts support inspection of mixed-backend timing relationships.

2.6 Interaction and performance

A full matplotlib figure rebuild is appropriate when the mode or pane geometry changes, but it is wasteful for a visibility toggle or cursor movement. The current renderer computes a composition signature from the properties that determine axes and artist structure. If the signature is unchanged, standard mode updates line visibility in place. Stacked mode adds or removes only the affected panes and retains the axes, zoom, titles, and lines of surviving panes. Changes to statistics, pane order, or view mode take the full rebuild path because they alter structure.

Rapid selection events are coalesced by a single-shot timer. Long monotonic traces are reduced at draw time with a min/max envelope, preserving local extrema while limiting the number of points sent to matplotlib. Zooming schedules a new slice from the raw array, so detail returns as the visible interval narrows. Cursor dragging uses a saved canvas background and redraws only the cursor artists. Together, these paths remove the repeated work that made the earlier 464 ms stacked toggle noticeable.

The distinction between source data and display data is deliberate. Figure 2.6 shows which interactions may use derived or decimated representations and which ones must read the full arrays.

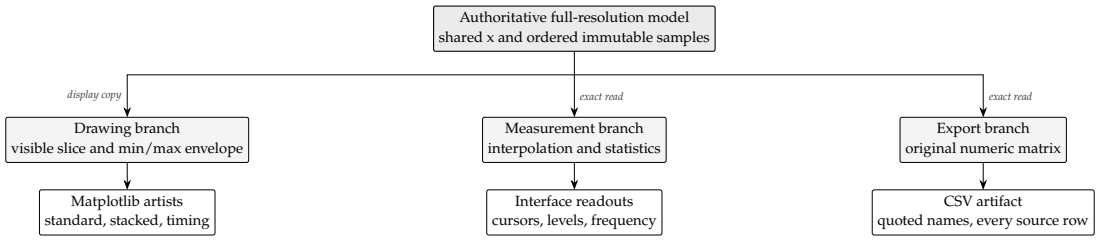


Figure 2.6: The plotting data contract. Only the drawing branch may decimate. Cursor values, statistics, and CSV rows are computed from the authoritative simulation arrays, so a faster canvas cannot change an engineering result.

The same ownership rules support practical features without creating parallel data stores. Search changes which rows are shown, not trace selection. Theme changes restyle live axes and artists. Function panes store their derived arrays explicitly. CSV export quotes signal names safely and writes the numeric body with `numpy.savetxt`. These operations share one model, which prevents a view change from silently changing saved engineering data.

2.7 Validation

Plotting tests are divided by responsibility. Pure numerical tests run without a visible interface. Qt-backed tests construct the window only when artist lifetimes, signals, scaling, or live theming are the behavior under test.

Table 2.2: Representative plotting risks and the evidence used to close them.

Risk	Implemented boundary	Validation
Grouped or irregular simulator output	Two-pass parser with one vectorized load per group	Transient, AC, ragged-row, raw-name, event-resampling, and full round-trip tests
Unsafe function expression	AST whitelist and controlled signal-name substitution	Arithmetic, precedence, allowed-function, mismatched-length, attribute-access, and unknown-call tests
Slow stacked visibility changes	Composition signature and surviving-axes reuse	Tests confirm removal reuses axes, addition creates only one pane, and pane heights persist
Loss of detail on large data	Draw-only min/max envelope with zoom-time re-slicing	Tests verify reduction, preserved extrema, constant signals, and recovery from raw data after zoom
Theme and display changes	Live palette reapplication and DPI-aware matplotlib sizing	Theme-artist and high-DPI regression suites
Stale or expensive cursors	Persistent cursor state with background blitting	Interaction tests plus direct verification in standard, stacked, timing, and logarithmic axes

Impact

The plotting subsystem now has an explicit parser boundary, one state-owning window, responsibility-specific interaction modules, and a clear separation between raw engineering data and display optimization. Roughly thirty integration defects were closed across eleven example circuits. Extraction measured 2.4 times faster, stacked refresh avoids rebuilding unaffected panes, cursor dragging uses blitting, and user-entered functions no longer pass through unrestricted Python evaluation. These properties make the viewer suitable as the shared result surface for the mixed-signal workflows introduced later in the report.

Chapter 3

Completing the PyQt6 Migration

The starting tree already contained a broad mechanical port from PyQt5 to PyQt6. The main window could be constructed, but that was only the first checkpoint. I completed the migration at the places where a desktop application is most likely to fail: signals, process ownership, modal dialogs, repeated runs, and shutdown. The result is a PyQt6 runtime contract shared by the simulator, converters, editors, and external tools.

3.1 Why a successful launch was insufficient

Constructing the main window exercises imports, widget constructors, and an initial paint. It does not launch Ngspice, wait for KiCad, return a dialog result, deliver a queued signal, or destroy a dock while work is active. Those operations are where Qt owns native objects and Python owns wrappers around them. A wrong enum, an unretained `QThread`, or a callback that reaches a deleted widget may remain invisible until the user performs a specific action.

I therefore treated each user workflow as a runtime contract. A contract begins at an action in the main window, crosses a process, thread, or dialog boundary, and ends only when the GUI has received a typed result and restored a usable state. Figure 3.1 shows the three boundary types that recur across eSim.

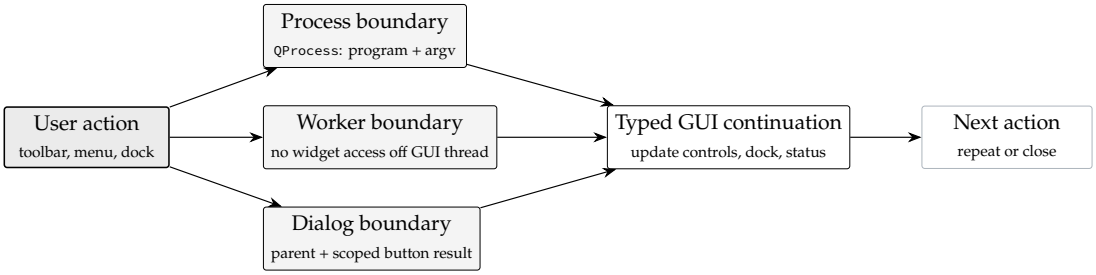


Figure 3.1: The migration boundary used for workflow validation. Opening a window covers only the first node; the useful behavior lies beyond it.

3.2 The semantic changes that mattered

PyQt6 changed more than import names. Flat constants became scoped enums, several methods were removed or renamed, and signal overloads became less forgiving. The mechanical changes were simple in isolation. Their consequences were distributed across long workflows.

Table 3.1: PyQt6 changes and the eSim behavior affected by each one.

Boundary	Migration issue	Runtime consequence and correction
Scoped enums	Flat values such as `QMessageBox.Yes` and `QProcess.NotRunning` no longer exist	Button masks, process-state tests, selection modes, palettes, and file dialogs now use their PyQt6 enum owners.
Signal meaning	`activated` reports a user choice; `currentTextChanged` also fires after programmatic index changes	Converter and model forms were reviewed field by field so initialization could not overwrite values or execute work early.
Process completion	Exit status and exit code carry different information	The application signal is typed as `QProcess.ExitStatus, int`; plotting starts only after a normal exit with code zero.
Removed APIs	`exec_()`, `pid()`, and the Qt5 matplotlib backend were stale	Modal calls use `exec()`, process tracking uses `processId()`, and matplotlib uses `backend_qtagg`.
Command launch	A command line is not a portable substitute for a program and an argument list	QProcess invocations were separated into executable and arguments; shell commands use parsed argument vectors.

One subtle example occurred in analysis and model forms. It seems reasonable to replace `activated` with `currentTextChanged`: both carry text. They do not have the same trigger. During form restoration, `setCurrentIndex()` emits `currentTextChanged`; a handler intended for a human selection can then clear AC, DC, or transient fields before the dialog appears. The correction was based on the required interaction, not a global signal rename.

The simulation return path required the same precision:

```
simulationEndSignal = pyqtSignal(QProcess.ExitStatus, int)

@pyqtSlot(QProcess.ExitStatus, int)
def plotSimulationData(self, exit_status, exit_code):
    if exit_status == QProcess.ExitStatus.NormalExit and exit_code == 0:
        self.obj_Mainview.obj_dockarea.plottingEditor()
```

Keeping the parameters in this order matters. A process can terminate normally and still return a nonzero error code. Plotting that run would replace a useful simulator failure with a second, misleading plotting error.

3.3 One owner for every running process

eSim launches two kinds of children. Ngspice is a `QProcess` owned by its simulation widget. KiCad and other external applications are launched through a `WorkerThread` using `subprocess.Popen`. Both paths now retain process handles instead of bare process identifiers. A handle describes the same child for its lifetime; an integer identifier may be reused by the operating system after the original process exits.

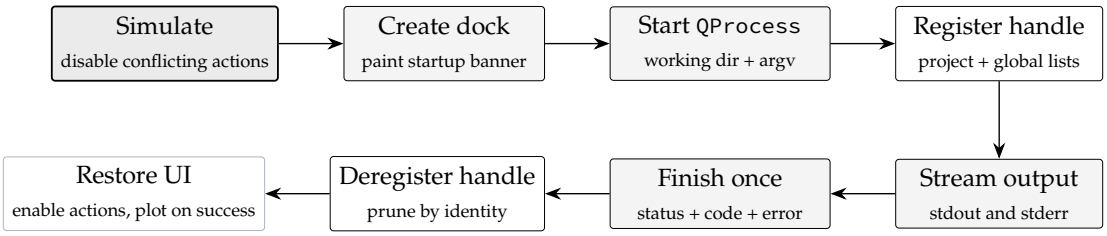


Figure 3.2: Current Ngspice process lifecycle in `NgspiceWidget.py`. Clean exit, process error, and dock destruction converge on one completion and cleanup path.

The dock-destruction branch in Figure 3.2 handles a failure that ordinary success tests miss. If a user closes the Simulation tab while Ngspice is active, Qt destroys the widget and its child process. A small reporter, connected to `destroyed`, closes over plain Python state rather than the deleted widget. It emits a crash result once, removes the process by identity, and allows the main window to re-enable Simulate, Convert, Close Project, and Workspace.

The external-tool path follows the same ownership rule. Each `WorkerThread` retains itself while its child is alive, reports launch errors through a queued signal, and deregisters the `Popen` handle after `wait()` reaps the process. No message box is created on the worker thread.

3.4 Dialogs and exceptions return to the GUI thread

Modal dialogs were consolidated in `configuration/Dialogs.py`. The helper builds a parented `QMessageBox`, applies the required standard-button mask, and returns the scoped result from `exec()`. Parenting is operational: it keeps the dialog above the window that requested it and ensures that closing the owner also closes the dialog.

Unhandled exceptions from Qt slots use a separate path. The installed exception hook writes the traceback first, then posts one deduplicated message to the GUI reporter. The dialog is created on a later event-loop turn. This avoids constructing widgets on a worker thread and avoids entering a nested modal loop from inside a paint, close, or teardown callback.

3.5 Shutdown is an ordered operation

PyQt wraps C++ objects whose destruction order is controlled partly by Qt. At exit, a late timer or animation can invoke Python code after its target widget has been freed. The application now quiesces active work before Qt releases the widget tree.

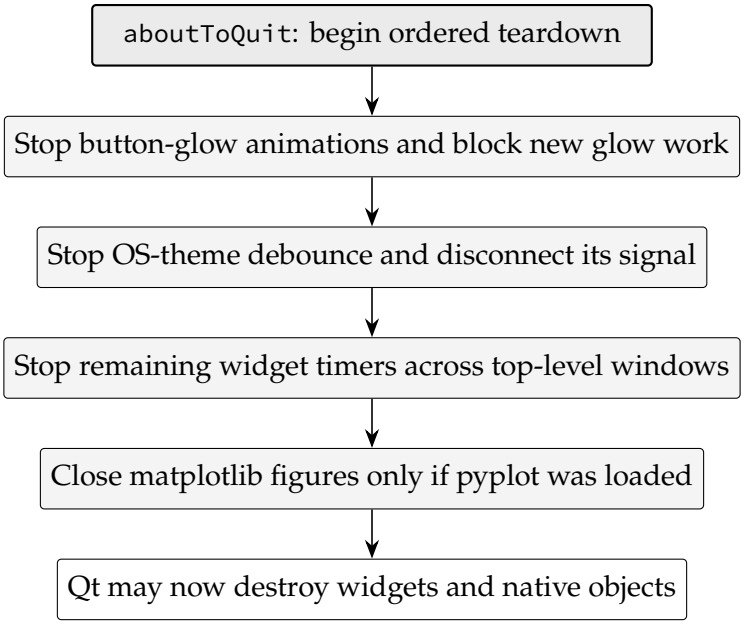


Figure 3.3: The teardown order implemented by `Application.py`. Each stage is guarded so shutdown cannot be aborted by a cleanup error.

Application exit also terminates all tracked children within one shared wait budget. Every child receives a graceful terminate request first; only the children still alive after the common deadline are killed. The total delay therefore stays bounded as the number of open tools grows.

3.6 Evidence and regression boundaries

The final audit combined source inspection with focused tests. A tree-wide search of the runtime source finds no PyQt5 import, Qt5 matplotlib backend, or `exec_()` call. The remaining "PyQt5" text appears only in regression-test documentation that forbids those patterns.

Table 3.2: Verification layers retained in the current source tree.

Layer	Question	Concrete guard
Source	Can a stale Qt5 API re-enter?	Dialog-hygiene and tree-wide text audits
Construction	Can widgets be created without a display server?	Headless Qt widget fixtures and import tests
Action	Are program, arguments, enum values, and signal types valid?	Console migration and workflow-specific tests
Repeat	Does a second run reuse valid state?	Process reuse, converter-state, and single-instance dock tests
Close	Are handles reaped and GUI controls restored?	Process lifecycle, batch termination, and dock-destruction tests
Exit	Can callbacks outlive their widgets?	Application teardown and motion-filter tests

Impact

- Completed the PyQt6 migration across real workflows, including simulation, converters, model tools, dialogs, repeated runs, and shutdown.
- Replaced ambiguous process bookkeeping with retained handles and one-shot completion paths.
- Enforced GUI-thread ownership for dialogs and error reporting.
- Established the Qt6 runtime foundation used by Aurora, the native editor, Ubuntu packaging, and the bundled Windows release.

Part II

Modernizing the Experience

Chapter 4

Aurora: A Design System for eSim

Aurora is the visual and interaction layer developed for eSim 2.6. It gives the application a coherent light and dark interface, a useful home screen, scalable controls, consistent dialogs, and safe live theme changes. The work extends beyond two stylesheets: it defines shared tokens, custom-painted widgets, motion rules, icon recoloring, plotting colors, preferences, and lifecycle guards.

4.1 Design goals from the application, not a mock-up

eSim contains several kinds of interface in one process: Qt widgets, docked tools, text editors, terminal output, matplotlib canvases, SVG icons, and native window frames. Styling only the main window would leave a patchwork once the user opened a converter or a plot. Aurora was therefore designed around five requirements:

1. one semantic palette for every surface;
2. light, dark, and operating-system-following modes;
3. readable controls across different logical screen sizes;
4. immediate preference changes that can be cancelled safely;
5. animations and effects that stop cleanly during repolish and shutdown.

These requirements shaped the architecture. Visual values flow down from named tokens, while state changes flow through one serialized application-level theme function. Individual tools do not invent a second theme mechanism.

4.2 One visual vocabulary, several renderers

The 84-line `frontEnd/tokens.py` file names the stable visual roles used by the rest of the application. The dark palette, for example, distinguishes the application background, raised surfaces, borders, primary and secondary text, accent, success, and danger states.

```
DARK = {
    "bg":         "#05070F",
    "surface":    "#0E1728",
    "text":       "#F4F8FF",
    "text_muted":"#94A8C3",
    "accent":     "#53D7FF",
    "accent_2":   "#9B7CFF",
    "success":    "#42E6A4",
    "danger":     "#FB7185",
}
```

The token names are more useful than raw colors because each renderer needs the same meaning in a different form. QSS needs selectors and pixel metrics. Custom widgets need `QColor` values. Matplotlib needs figure, axes, grid, line, and annotation colors. Native Windows frames need a dark-mode flag and DWM attributes. Figure 4.1 follows those paths through the current source.

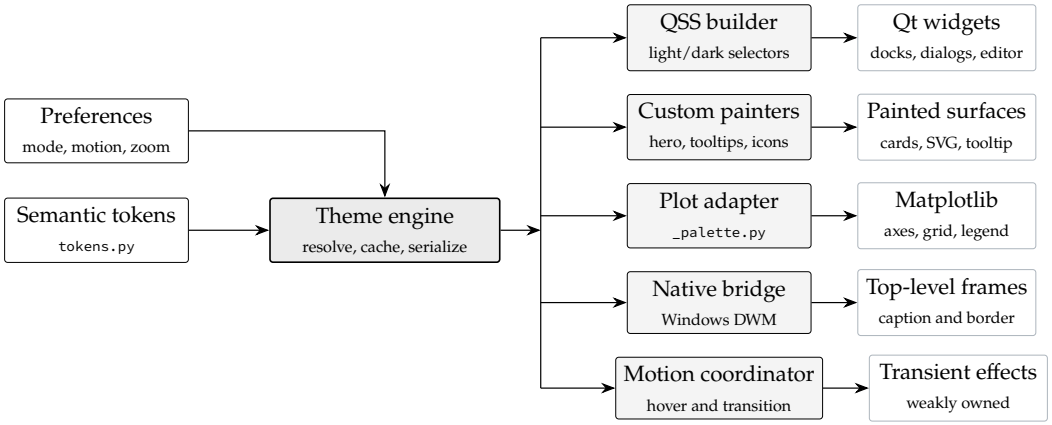


Figure 4.1: Aurora's rendering architecture in the current source. The theme engine resolves one state and publishes it through a single fan-out bus. Each adapter then translates that state for the surface it owns, keeping renderer-specific details out of theme selection.

Table 4.1: Main Aurora modules in the audited eSim 2.6 source tree.

Module	Lines	Responsibility
`tokens.py`	84	Semantic color roles and conversion helpers
`theme_utils.py`	1,484	Mode resolution, QSS generation and cache, zoom metrics, transitions, icon refresh, native frame integration
`motion.py`	812	Application event filter, optional hover effects, effect ownership, cleanup
`widgets.py`	632	Custom-painted labels, hero surfaces, and visual helpers
`style_dark.qss`	1,792	Dark selectors and scalable control metrics
`style_light.qss`	1,803	Light selectors and scalable control metrics
`_palette.py`	226	Live Aurora-to-matplotlib palette adapter

These line counts describe the audited source snapshot, not a target. They also explain why calling Aurora a stylesheet is incomplete. More than half of the difficult work concerns state, ownership, scaling, and interaction around the stylesheet.

4.3 A home screen that uses the real actions

The Welcome dock in `browser/Welcome.py` is the first complete view a user sees. It groups the workflow into Quick Start and Tools and remains scrollable on a short display.

The cards resolve an attribute on the main application and trigger its existing `QAction`. New Project from the dashboard, toolbar, or File menu therefore reaches the same handler. The left rail groups Design, Simulate, Model, and Convert actions, while the project explorer keeps project navigation separate from tool launch.

The center of the application is a `DockArea`, not a stack of unrelated windows. Its registry is keyed by tool kind and project. A repeated launch focuses or reloads the live dock; a first launch constructs, tabifies, and registers one. Figure 4.2 follows that ownership path through `Application.py`, `Welcome.py`, and `DockArea.py`.

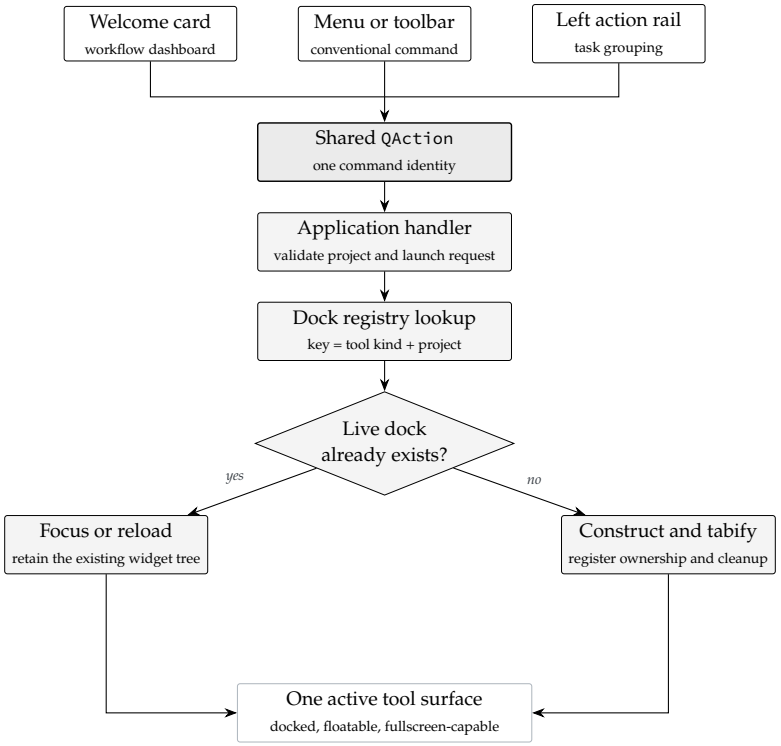
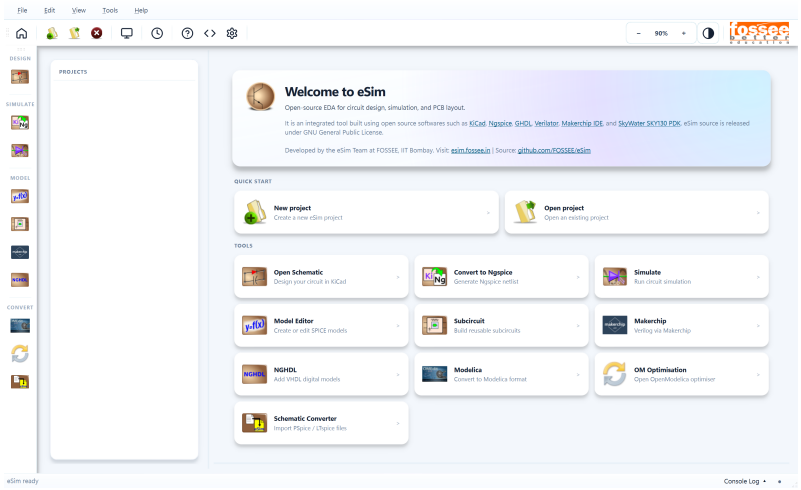
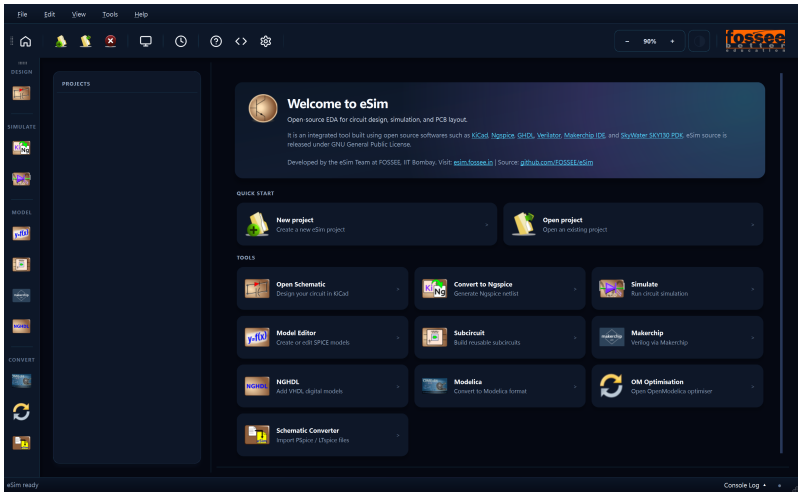


Figure 4.2: Command identity and dock ownership in the current main window. Three launch surfaces converge on one action and handler; the registry then reuses a live tool or creates exactly one owned dock for that tool and project.



Aurora light



Aurora dark

Figure 4.3: Fresh captures from the current eSim 2.6 application. Both modes use the same dashboard, action identity, dock layout, zoom control, and information hierarchy.

Figure 4.3 compares the same production window in both modes. The captures came from the current modernization tree with an isolated settings directory. The screen was not recreated for the report, and no image from an earlier report was used.

4.4 Live theme changes are a transaction

A running theme change touches existing widgets, icons, plots, graphics effects, and native frames. It can also arrive twice: a user may click quickly, or the operating system may emit several color-scheme notifications. `theme_utils.py` serializes the operation with an applying flag and one pending bit. A request received during a repolish is coalesced and performed after the first one settles.

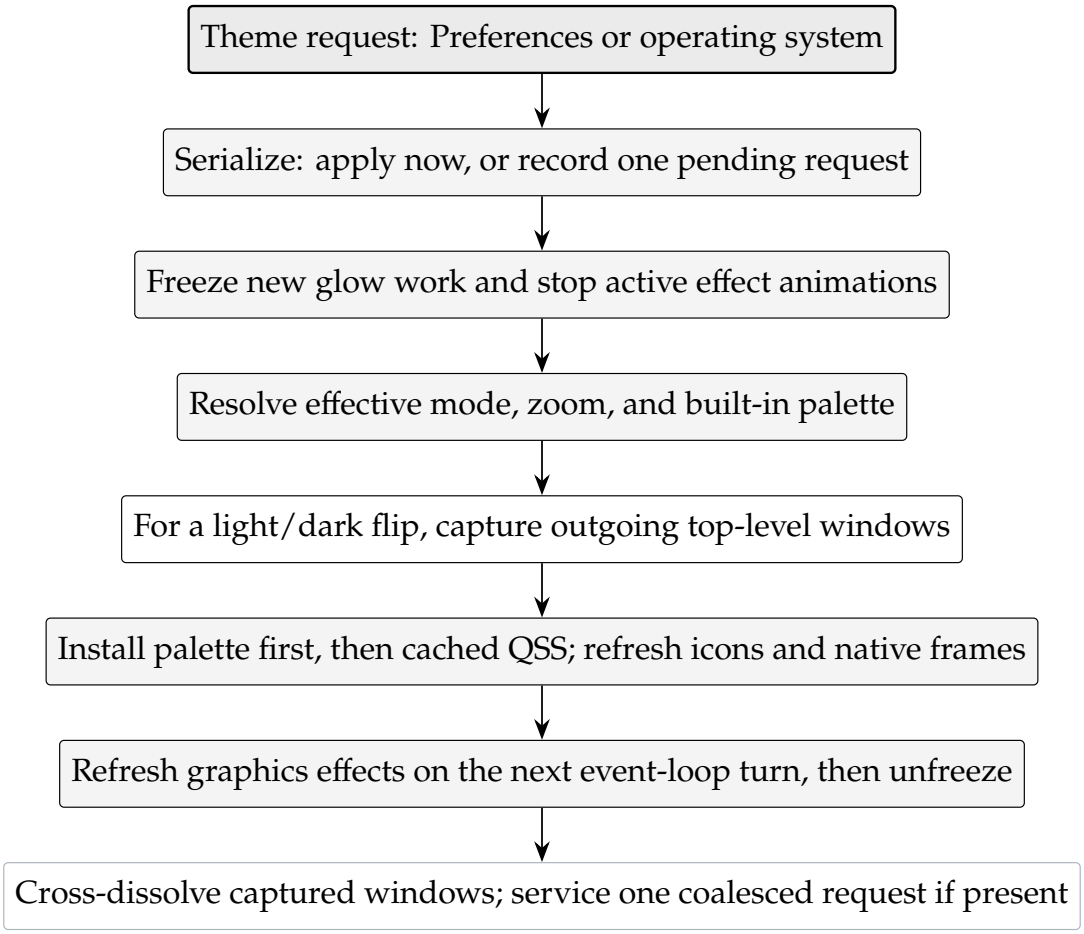


Figure 4.4: The live theme transaction implemented by `theme_utils.py`. Ordering prevents half-themed frames and animations that target replaced graphics effects.

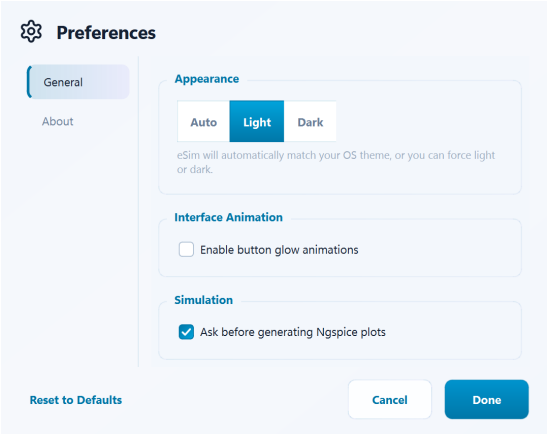
The palette is installed before the application stylesheet. Qt does not reliably propagate a new palette through a tree that is already controlled by QSS; setting the palette first lets the stylesheet repolish every widget against one state. The QSS builder caches each theme and zoom combination, so switching back to a previously used mode avoids reading and transforming the same sheet again.

The transition uses snapshots rather than attempting to animate thousands of live widgets independently. Aurora captures the visible top-level windows in the outgoing theme, applies the new theme underneath, then fades the captured layer away. Painting is frozen only for the handover. If motion is disabled, the same state transaction runs without the dissolve.

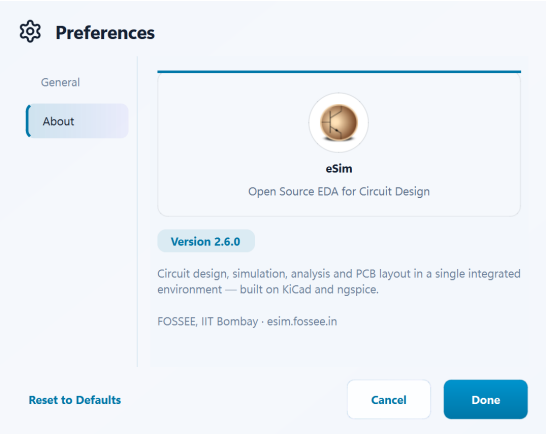
4.5 Preferences are live and reversible

The current Preferences dialog exposes three decisions that users can understand: follow the operating system or force Light or Dark, enable button-glow animation, and choose whether eSim asks before generating native Ngspice plots. Earlier experimental accent keys are normalized to the built-in palette so all surfaces

remain coordinated.



General: live appearance and simulation choices



About: product identity and running version

Figure 4.5: Both pages of the current production Preferences dialog, captured directly from eSim 2.6. General changes preview in the running application; Cancel restores the opening snapshot. The About page reports the product, release, and FOSSEE identity.

`PreferencesDialog.py` records the normalized settings when the dialog opens. Changes are debounced and applied to the running `QApplication`. Done writes the new state; Cancel compares the on-disk state with the opening snapshot and reapplies only when a real difference exists. Preference files use atomic replacement, so a process interruption cannot leave a partially written JSON document.

Keeping About in the same compact dialog is intentional. It provides the installed version and project identity without adding another independently styled window, and it is rendered by the same token, scaling, and navigation rules as General.

4.6 Scaling, motion, and custom surfaces

Aurora had to remain usable on displays with different logical work areas. On first run, `calibrate_default_zoom()` measures the available logical width and height, selects a value on the same ten-percent grid as the toolbar control, and clamps it to a usable range. Later starts preserve the user's choice. Font scaling follows its own curve and floor so compact chrome does not force labels below a readable size.

QSS cannot express every visual. `widgets.py` paints the Welcome hero and gradient text; `icon_paths.py` recolors SVG assets from the active palette; `ngspiceSimulation/_palette.py` derives matplotlib colors from the live application. Tooltips use an application-level event filter so their rounded card style is consistent with docks and dialogs.

Motion is optional and bounded. `motion.py` installs one idempotent application filter, keeps weak references to effect targets, and creates hover feedback only when needed. Before a theme repolish, active animations stop and new ones are

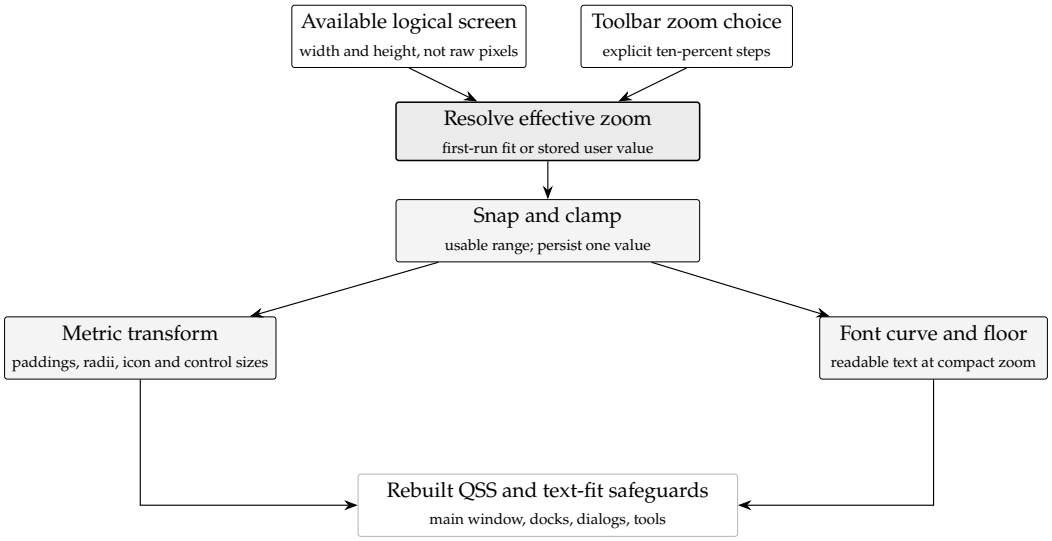


Figure 4.6: Aurora's scaling path. Automatic calibration runs only when no user choice exists; both inputs converge on the same bounded zoom state, after which control metrics and font sizing are derived separately before the interface is repolished.

frozen. The same ownership rules are used during application teardown, linking the visual system back to the lifecycle work in Chapter 3.

Table 4.2: Aurora invariants and their visible effect.

Invariant	Implementation rule	User-visible result
Semantic color	Renderers consume named roles	Main window, dialogs, plots, and icons agree in both modes
Action identity	Welcome cards trigger existing actions	Dashboard, menus, and toolbars cannot diverge
Single tool instance	Dock registry reuses a live tool per project	Repeated clicks focus the existing tab
Atomic preference state	Write a complete replacement file	A failed write cannot leave truncated settings
Bounded motion	One filter, weak ownership, explicit freeze and cleanup	Feedback does not survive its widget or a theme change
Adaptive metrics	First-run calibration plus user-controlled zoom	Controls remain usable across logical display sizes

4.7 Working tools, not only the main window

The Schematic Converter is only one consumer of Aurora. Two substantial tool panels had been missing from the visual account of the redesign: the Subcircuit Builder and the Model Editor. They matter because their changes were not a coat of paint. Both tools had layouts that obscured state and left most of a dock unused; both now make the object being edited and the next safe action explicit.

4.7.1 The Subcircuit Builder owns a visible selection

The legacy builder put four fixed-size buttons in one horizontal row. It did not name the selected subcircuit, distinguish schematic actions from netlist actions, or prevent Convert from being pressed before a target existed. Fixed widths also clipped translated or scaled labels while the remainder of the dock stayed empty.

The current panel groups New and Edit under Schematic, groups Convert and Upload under Netlist, and explains the three-stage workflow beside the controls. More importantly, the selection strip is backed by panel-local folder and stem state. Raising a project's tab reasserts that tab's selection, so Convert cannot silently act on a subcircuit last touched in another project. Convert remains disabled until a target exists.

The visible state corresponds to a deeper repair. New and Edit resolve a folder and one canonical subcircuit stem; ambiguous folders use a picker instead of attempting to open a file named `None`. Convert captures that identity before starting its background netlist export, ignores a second click while the job is active, and continues with the captured target even if the user changes tabs. Modern KiCad schematics regenerate their netlist automatically, while the shipped legacy corpus can continue through an existing `.cir` file. Upload remains a separate path for a validated, already-authored `.subckt` definition.

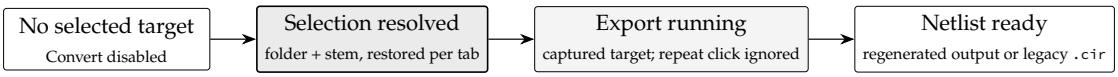
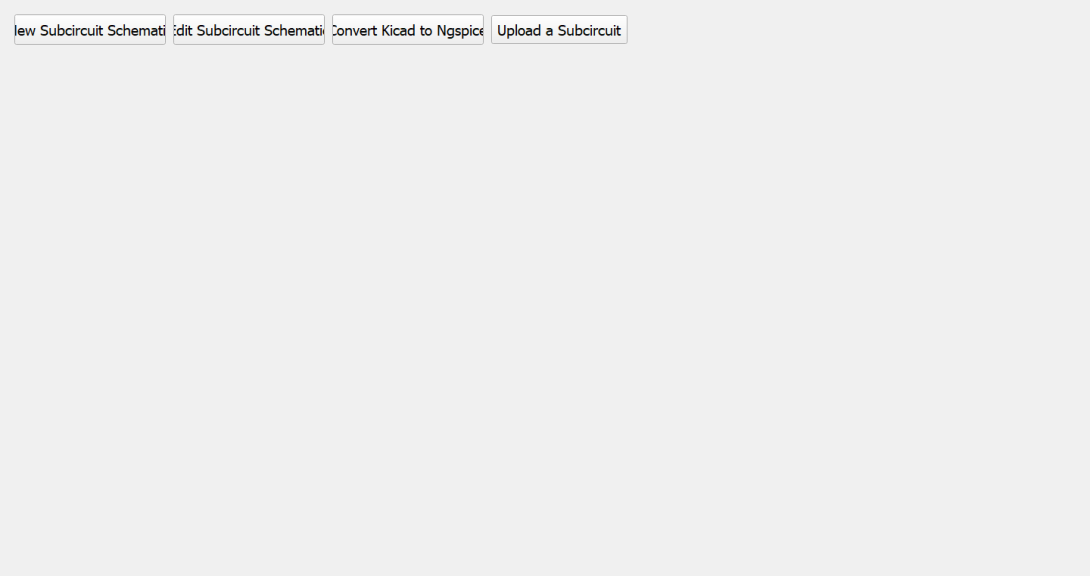
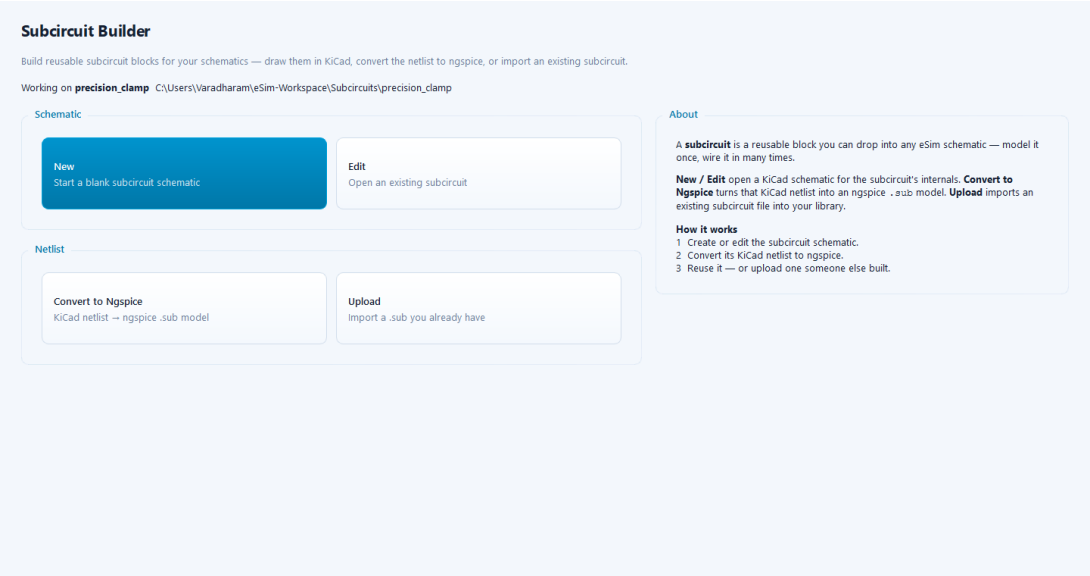


Figure 4.7: Subcircuit conversion state. The worker receives an immutable target snapshot, so tab changes cannot redirect an in-flight conversion and a second click cannot start a competing export.



Before: four fixed controls, no visible target, and an otherwise empty dock

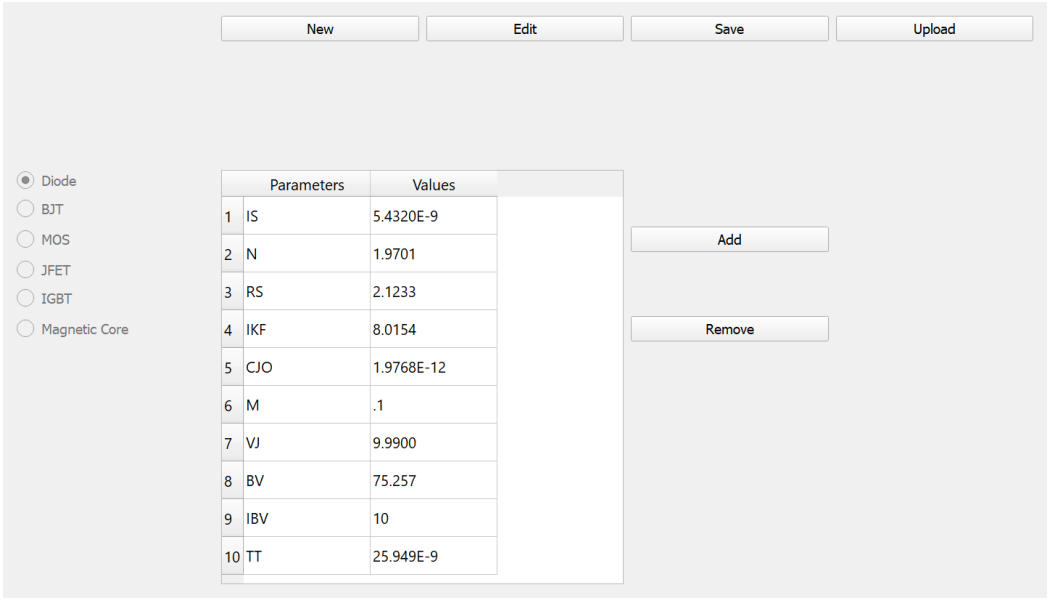


After: task groups, selection state, guidance, and a gated conversion action

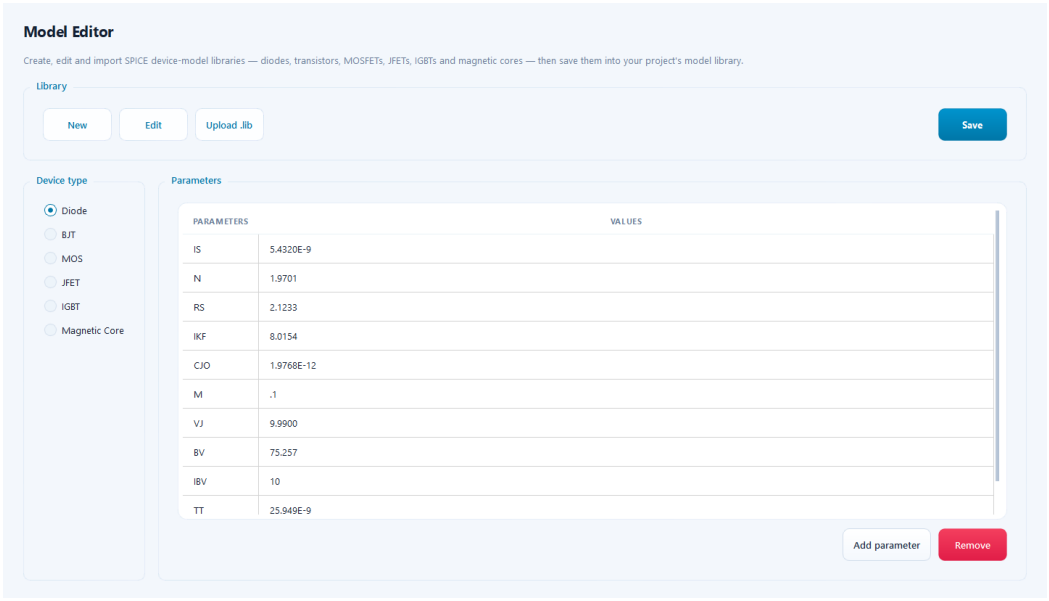
Figure 4.8: The Subcircuit Builder before and after modernization. Both panels were launched directly from their corresponding eSim source trees: the legacy PyQt5 implementation and the current PyQt6 implementation with Aurora. The current capture selects `precision_clamp` so the state that drives Convert is visible rather than implicit.

4.7.2 The Model Editor uses the whole dock and protects table edits

The legacy Model Editor scattered four file actions, a device list, a fixed `200 px by 200 px` table, and Add/Remove buttons across a grid. On a large dock, the editable data remained a small island surrounded by dead space. Figure 4.9 uses the same 1N4148 model in both captures, making the layout change directly comparable.



Before: fixed table and disconnected controls in a sparse grid



After: library toolbar, device rail, expanding parameter table, and explicit actions

Figure 4.9: The same diode model in the legacy and current Model Editor, launched directly from the respective PyQt5 and PyQt6 source implementations. The current table expands with the dock, keeps the active device visible, and gives Save and Remove distinct visual roles.

The redesign also fixes editing behavior. A table delegate gives the in-cell editor the full cell rectangle and preserves a readable caret state. Saving rebuilds the parameter map from both table columns, so renaming a parameter no longer drops its value or leaves a stale key. Library paths now pass through the shared resource resolver, which keeps the same editor usable from a source tree, an Ubuntu installation, and the staged Windows bundle.

Specialized interfaces remain with the chapters that explain their contracts: plotting in Chapter 2, project and editor tools in Chapters 5 and 6, digital verification and co-simulation in their workflow chapters, NGHDL and Makerchip in Chapter 10, and installation diagnostics later in the report. This avoids repeating one screenshot as evidence for unrelated work.

4.8 Regression coverage

Focused tests cover theme snapshots, QSS caching, motion-filter reuse, graphics effects, zoom, text fit, dock reuse, plot theming, and teardown. The cases remain separate because palette correctness, transition safety, and effect destruction are independent concerns.

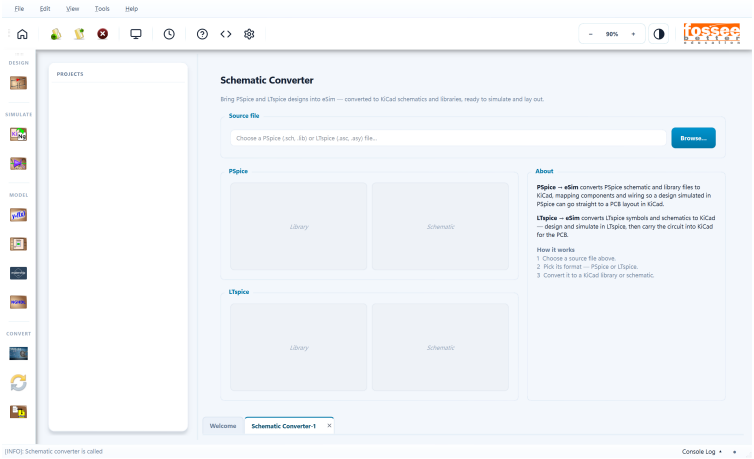


Figure 4.10: Aurora applied to the current Schematic Converter. The same tokens and metrics cover full-width input controls, disabled empty-state actions, explanatory content, the project rail, and the single-instance dock tab.

Impact

- One light/dark vocabulary now covers windows, docks, dialogs, editors, plots, icons, and native frames.
- Dashboard cards use the existing actions, while the dock registry prevents duplicate tool trees.
- Theme changes are serialized, cached, reversible, and safe around effects and native objects.
- Screen calibration and user zoom preserve usable metrics and readable text.
- Current-state UI images in this chapter were captured from the current eSim tree; the two labeled legacy views were captured from the pre-modernization source for direct comparison, not copied from another report.

Chapter 5

Project Identity and Lifecycle

An eSim project is more than the directory that contains it. Its schematic, project anchor, converted netlist, simulation data, model choices, and background jobs must continue to refer to the same design even after the directory is renamed or the user changes projects. eSim 2.6 gives that state explicit owners. A project is resolved from its `.proj` anchor, snapshots are stored outside the live directory, per-user conversion hints stay under `~/.esim`, and asynchronous continuations retain the project that launched them.

5.1 When a folder name became identity

The eSim 2.5 opening path derived the project name from the final component of the directory path and then looked for a `.proj` file with the same stem:

```
projName = os.path.basename(str(projDir))
lookProj = os.path.join(str(projDir), projName + ".proj")
if os.path.exists(lookProj):
    return True
return False
```

That rule made a directory name behave like a primary key. Renaming the folder broke opening even when every project file was intact. The same assumption appeared in schematic lookup, current-project state, conversion caches, and the project tree, so a local fix in one validator could not make the project portable.

> **Before: eSim 2.5**
>
> In eSim 2.5 the folder, `.proj`, and schematic were expected to share one space-free stem.
>
> - Renaming or moving the folder could prevent the project from opening.
> - The new-project validator rejected names containing spaces.
> - IP-Library designs could arrive in `eSim_Project_Files` while their actual anchor was `MACProject.proj`.

5.2 One project identity contract

`src/projManagement/projectPaths.py` is a 280-line, Qt-free boundary for project identity. It normalises path spellings for comparison, searches only the

project directory for anchors, and returns both a stem and a status. The status matters: one anchor is normal, no anchor uses a legacy fallback, and several anchors are ambiguous and must not be guessed.

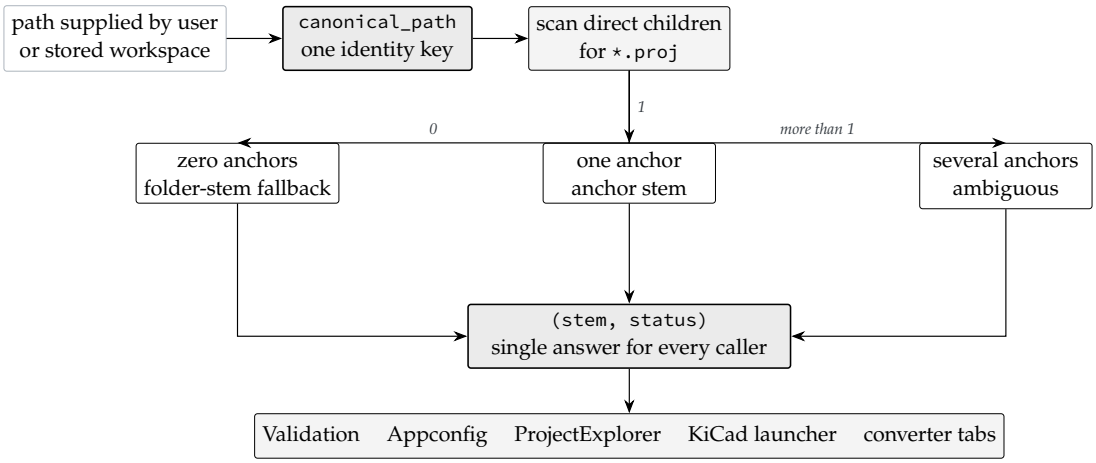


Figure 5.1: Project resolution in the current source. Path normalisation and anchor discovery happen once, and callers receive the resolved stem together with an explicit status instead of reconstructing identity from a folder basename.

The same module resolves the main schematic. It first reads the `schematicFile` record in the anchor, prefers the migrated `.kicad_sch` sibling when a legacy `.sch` has been upgraded, and only then uses naming conventions. The project registry is also replaced atomically through a sibling temporary file, so an interrupted write cannot erase the complete project tree.

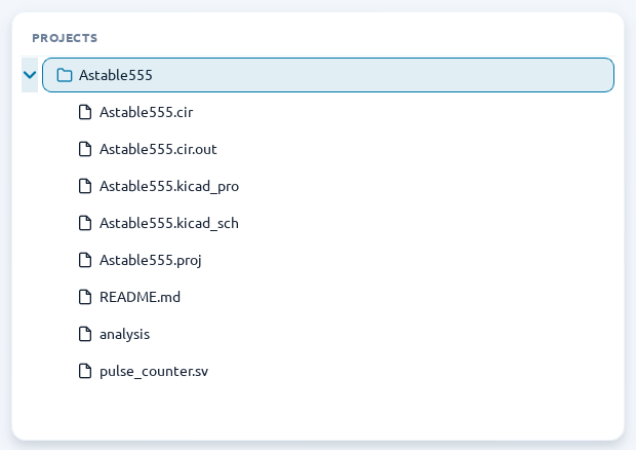


Figure 5.2: Fresh capture of the current Project Explorer. The folder used for this capture is named `Renamed Project With Spaces`, but the tree displays `Astable555`, resolved from `Astable555.proj`. The expanded project also shows the file types that feed conversion, editing, and simulation.

After: eSim 2.6

In eSim 2.6 the anchor owns the stem and the canonical directory owns equality.

- A folder can be renamed or moved without changing the project stem.
- Equivalent path spellings collapse to one project-tree entry and one dock key.
- A migrated KiCad schematic is opened instead of repeatedly returning to its legacy predecessor.

5.3 Snapshots with an undoable restore

Project Snapshots separates storage from presentation. The backend, `src/frontEnd/SnapshotStore.py` (363 lines), contains no Qt. The Aurora panel in `src/frontEnd/TimeExplorer.py` (385 lines) renders the store and translates the user's selection into a full or partial restore.

A snapshot is one directory under `~/.esim/history/<project>/<snapshot-id>`. Its `manifest.json` records the label, kind, time, source project path, and captured filenames; the bytes live under a `files/` child. A restore never consumes the snapshot. Before overwriting any selected live file, the store captures the current version of those same files as an automatic *Before restore* snapshot.

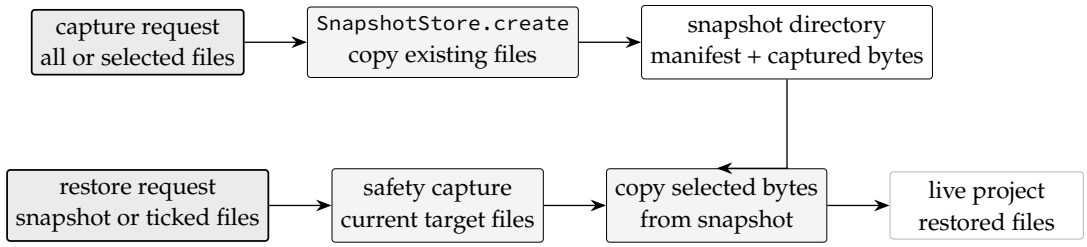


Figure 5.3: Snapshot capture and restore. The lower lane begins with a safety capture, so restoring an older version creates a path back to the state that existed immediately before the restore.

The panel is global rather than tied to one open project. Quick Backup names and targets the active project; Restore uses the destination path recorded in the chosen manifest. Checkboxes under a snapshot select individual files. Legacy loose backups with the form `<file>(<timestamp>)` are grouped by timestamp and migrated into the same directory model on first read.

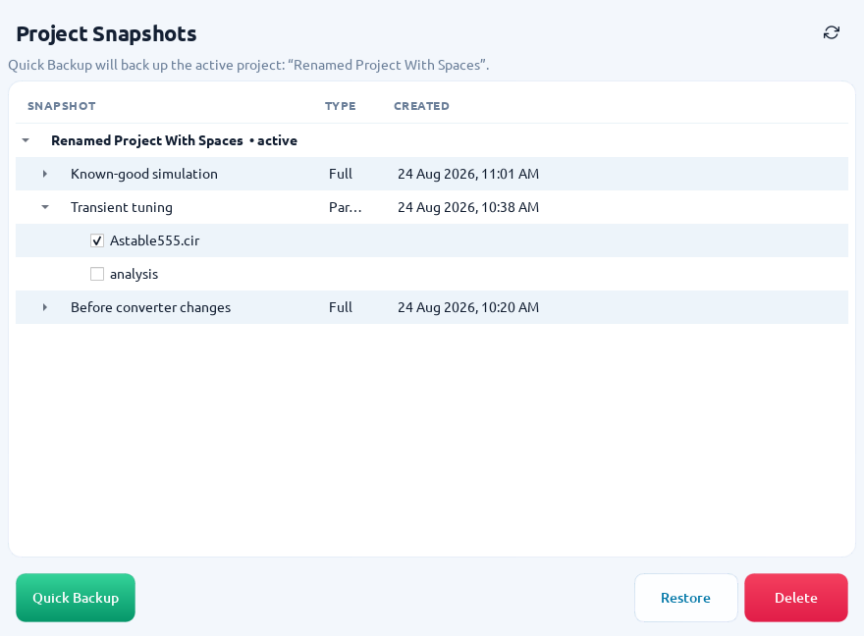


Figure 5.4: Fresh current eSim capture of Project Snapshots. The tree shows two full snapshots and one partial snapshot; only `Astable555.cir` is ticked for the illustrated partial restore. The header names the project targeted by Quick Backup.

5.4 State belongs to the user, the project, or the process

Several failures came from keeping correct data in the wrong place. A library path that helps one user repeat a conversion is not part of a shareable circuit. A dialog is not application state, but it needs a visible owner. A child process belongs to the project that launched it and must be stopped through its live handle. The current modules make those boundaries explicit.

Table 5.1: State ownership after the project-management rebuild.

State	Owner and location	Contract
Project identity	`projectPaths.py`	Stable through folder renames, path aliases, and schematic migration
Conversion choices	`~/.esim/prevvalues`	Per-user values remain outside a shared project; legacy values migrate once
Cross-project model hint	`modelCache.py`	A remembered path must still exist and never overrides a project's own choice
Dialogs	`Dialogs.py`	Modal prompts remain parented to the visible window that raised them
Install and user paths	`paths.py`	Resource lookup is independent of the process working directory
External processes	`Worker.py`	Live handles stay grouped by project; shutdown uses one deadline and avoids recycled PIDs

5.5 Asynchronous work retains its origin

KiCad netlist export can take several seconds on a cold Windows start. The current launcher therefore uses `BackgroundJob` from `src/maker/hdl/jobs.py`, then returns to the GUI thread through Qt signals. The project directory and resolved stem are captured before the job starts and bound into both success and failure callbacks. A later project switch changes the visible application state, but it cannot redirect the earlier continuation.

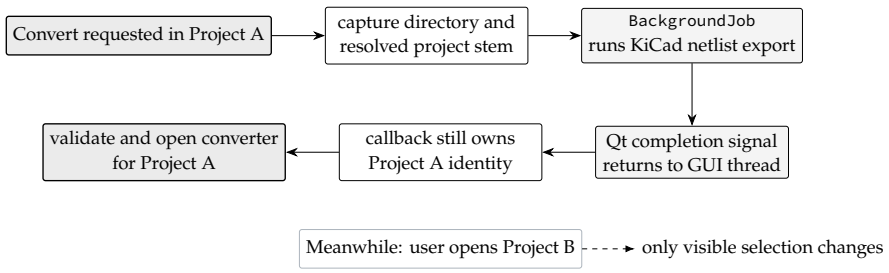


Figure 5.5: Project-pinned netlist generation. The callback closes over the identity captured at launch, so the converter resumes against Project A even if Project B becomes active while KiCad is running.

The external-process worker in `src/projManagement/Worker.py` solves a different lifecycle problem. It launches applications, retains each worker while its process is alive, stores handles under the originating project, and removes exited handles from every registry. Keeping these two worker roles separate makes the report match the code: one prepares a netlist result for a continuation; the other owns long-lived third-party processes.

5.6 Conversion as a state machine

The converter must turn several component rows into one consistent set of model assignments. `ModelGrouping.py` parses device and subcircuit instances and groups them by case-insensitive resolved model identity. `ModelGroupWidget.py` then exposes one group path, while preserving the original per-reference line edits consumed by the existing conversion code.

An instance inherits later group changes until the user edits that instance's field. That edit becomes an override; Reset reattaches it to the group. The distinction is implemented with `textEdited`, which represents user input, instead of `textChanged`, which also fires during programmatic fan-out. Removing a path clears the corresponding tracked assignment, so hidden stale values cannot reappear at Convert.

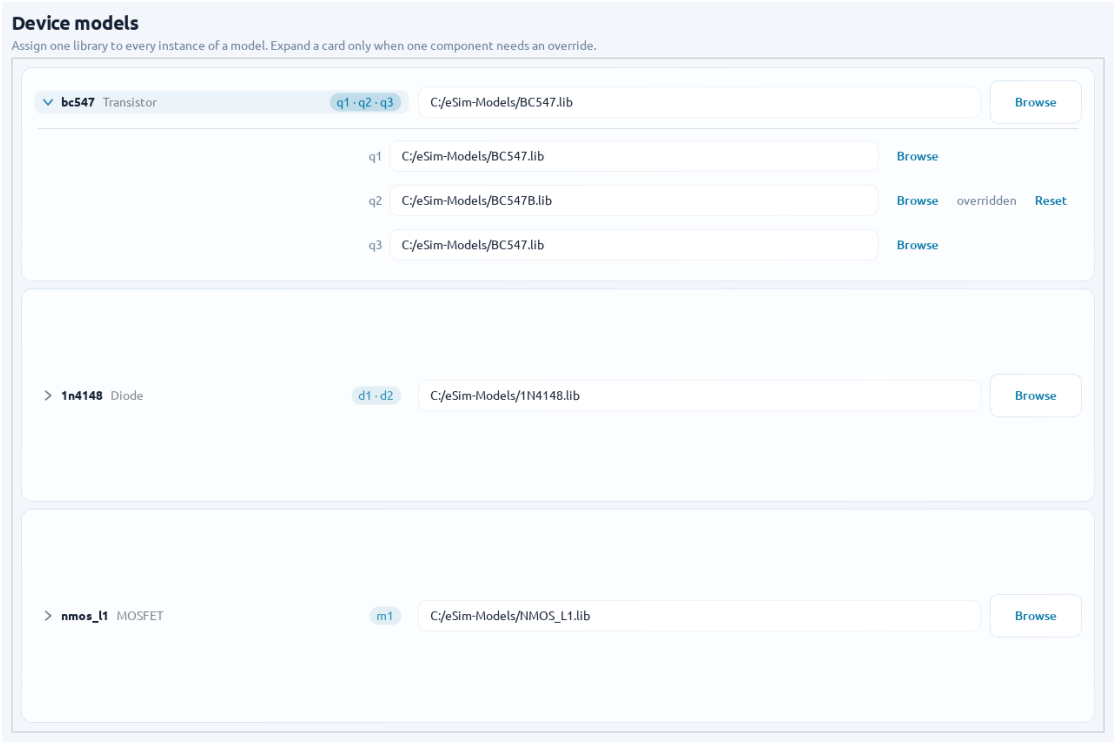


Figure 5.6: Fresh capture of the production Device Models tab. Three `BC547` instances receive one library assignment; `q2` has an explicit override and can be returned to the group with Reset. Diode and MOSFET groups remain collapsed without hiding their resolved paths.

Subcircuits add a structural check. The resolver reads the `.SUBCKT` declaration, preserves its port order, validates the selected directory against each instance's port count, and rejects a missing or ambiguous declaration. A cross-project cache may prefill a group only when the path still exists and validates for every member.

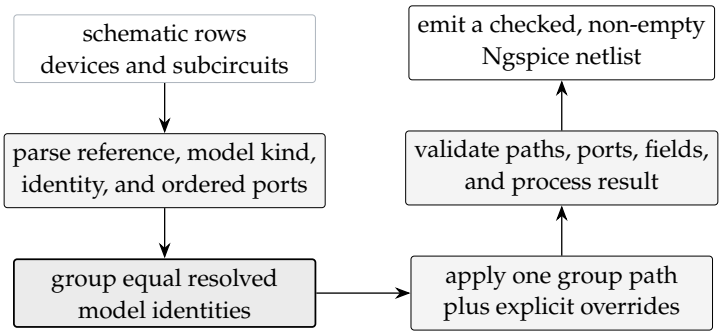


Figure 5.7: The conversion contract. Model grouping changes the interaction layer while the per-reference storage remains intact. Conversion advances only after current paths, subcircuit ports, required fields, process status, and generated output have passed their own checks.

A conversion is reported as successful only when the process succeeds, the expected file exists, and the result contains usable circuit content rather than whitespace or captured diagnostics. This keeps an empty netlist from reaching simulation and preserves the stage that owns the failure.

Table 5.2: Conversion gates and where a failed operation remains.

Gate	Acceptance condition	Failure remains with
Model source	Selected path exists and resolves for the whole model group	The affected group or per-instance override
Subcircuit structure	One declaration is found and its ordered port count matches the instance	The subcircuit selector, with the resolver's diagnostic
Converter process	The external conversion stage exits successfully	The conversion stage, with its process status and captured diagnostic
Output artifact	The expected netlist exists and contains usable circuit text	The converter tab; simulation is not started

Impact

- Project identity comes from the `.proj` anchor and canonical directory, not the folder name.
- Snapshots support full and per-file restore, with an automatic safety snapshot before any live file is replaced.
- Shared projects no longer carry another user's remembered model paths.
- Netlist preparation and external-process ownership have separate, explicit lifecycle boundaries.
- Grouped model assignment reduces repeated choices without replacing the converter's per-instance data contract.

Chapter 6

A Built-in Code Editor

SPICE decks, device models, subcircuits, Verilog, VHDL, project metadata, and generated netlists are part of an eSim workflow. Release 2.6 adds a 2,596-line editor subsystem that opens those files from the Project Explorer, keeps several of them in one tabbed window, applies language-specific syntax rules, and protects the file lifecycle when eSim or another tool changes the same path.

6.1 Why editing belongs inside eSim

In eSim 2.5, changing a generated netlist or inspecting a model meant leaving the application for an external editor. The user lost the project context, the file opened without eSim-specific syntax cues, and a general-purpose save could change its encoding or line endings. There was also no shared point at which conversion or simulation could ask open buffers to write their newest text before reading from disk.

The current editor is connected to the Project Explorer. One editor window is reused per project, and a normalised absolute path identifies each tab, so opening the same file again focuses its existing buffer. Before conversion, a shared hook saves modified buffers from every open editor window; the source exposes it as `flush_all_dirty`. The converter therefore reads the user's current text rather than the older file on disk.

6.2 Ownership from project row to file bytes

`src/codeEditor/EditorWindow.py` owns tabs, commands, search, window settings, and close decisions. Each rich tab is a `CodeEditor.py` buffer that owns loading, editing state, syntax, file watching, and save. If QScintilla cannot be imported, `PlainEditor.py` supplies the same basic file lifecycle through `QPlainTextEdit`; a missing optional dependency does not prevent the Project Explorer from starting.

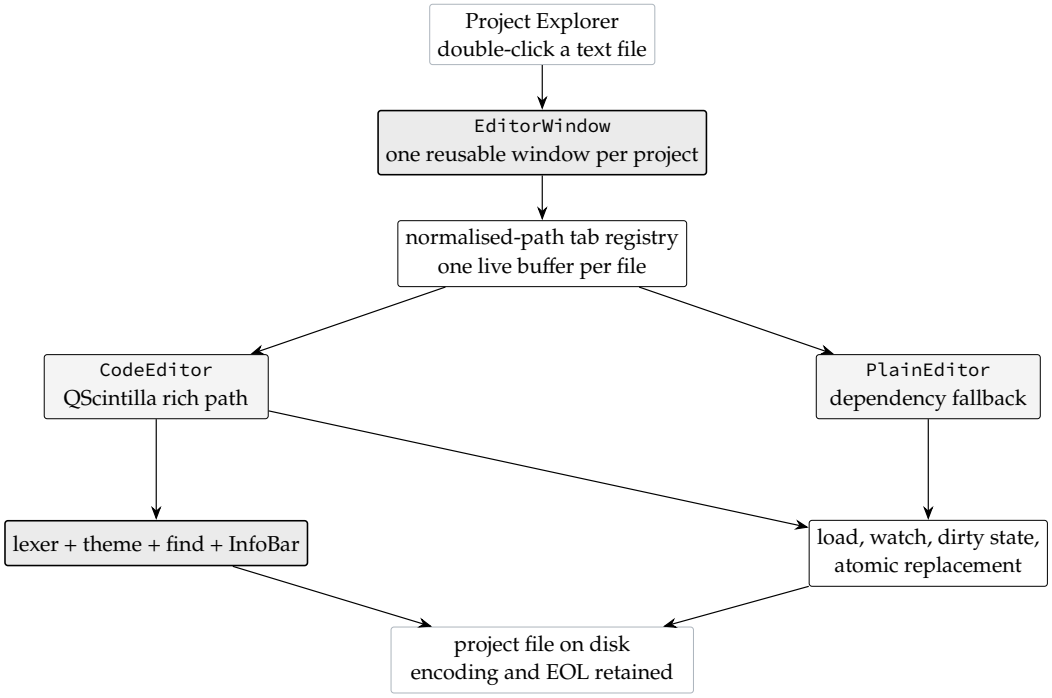


Figure 6.1: Editor ownership in the current source. Window navigation, per-file buffer state, language services, and disk replacement are separate layers. The fallback joins the same file-lifecycle boundary without pretending to provide QScintilla features.

Table 6.1: Current editor module inventory in the audited eSim 2.6 tree.

Module	Lines	Responsibility
`EditorWindow.py`	765	Tab registry, menus, shared commands, find overlay, status bar, reload prompts, and window lifecycle
`CodeEditor.py`	508	Rich buffer, atomic save, signatures, generated and oversize guards, search, folding, and context actions
`lexers.py`	535	Language detection, custom SPICE grammar, fold levels, SystemVerilog keywords, and stock lexer dispatch
`FindBar.py`	324	Debounced search and replace, match count, regex, case, whole-word, and keyboard navigation
`theme.py`	286	Aurora palette mapping for editor content and chrome
`InfoBar.py`	95	Inline notification for a conflicting external change
`PlainEditor.py`	83	QScintilla-free editing fallback
Total	2,596	Complete `src/codeEditor` subsystem

6.3 A SPICE editor that understands line roles

SPICE cannot be highlighted reliably by coloring every word from a keyword list. The first non-space character can make a line a comment, directive, continuation, device instance, or command inside a `.control` block. `SpiceLexer` therefore classifies the line before scanning its remaining tokens as numbers, parameters,

strings, expressions, operators, functions, or nodes.

The lexer also understands context. The first line of a top-level deck is styled as its title, while the generated `analysis` fragment is exempt because its first line is a real directive. A stored per-line state tracks whether styling is inside `.control`; editing `.control` or `.endc` propagates only until downstream state converges. Fold headers are produced for `.subckt`, `.control`, `.if`, and `.macro` regions. The full fold pass is skipped above 20,000 lines, which prevents a large deck from paying an unbounded cost for a visual convenience.

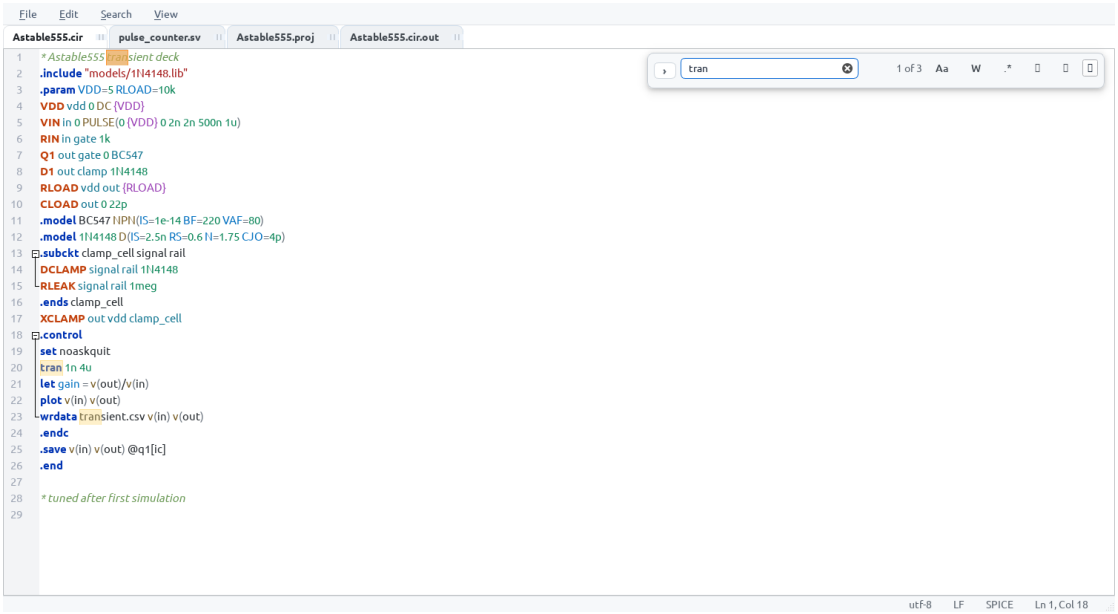


Figure 6.2: Fresh capture of the current editor with a writable `.cir` deck. The custom lexer distinguishes directives, instances, nodes, values, parameters, comments, and commands; the gutter exposes foldable subcircuit and control blocks. The live search overlay reports three matches, while the status bar identifies UTF-8, LF, and SPICE.

6.4 Language selection is explicit

The editor first uses the raw final extension to decide whether the file is generated. This is why `design.cir.out` is protected as an output even though its effective language is SPICE. It then maps the effective extension to a language and chooses the corresponding lexer. File size is a separate guard: a file above the configured 2,000,000-byte threshold opens read-only without syntax or folding.

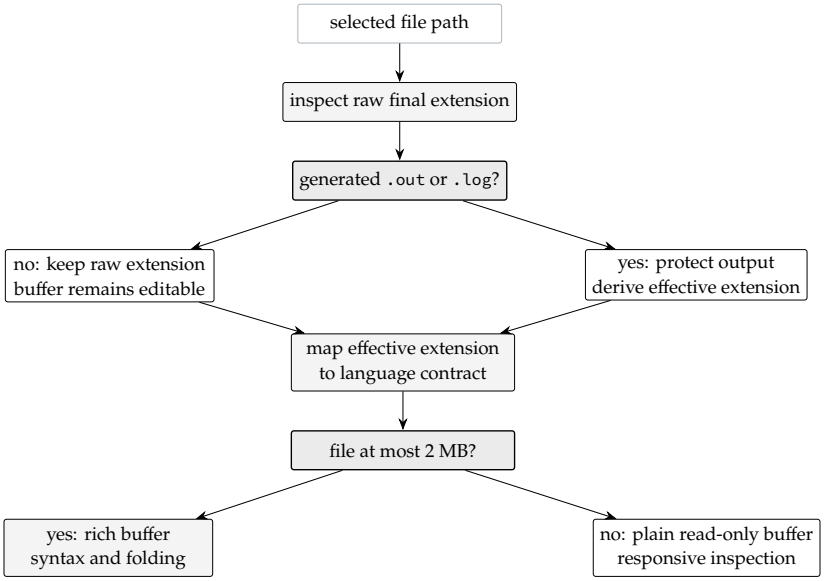


Figure 6.3: Language and safety dispatch. Generated ownership, language semantics, and file size answer different questions, so the editor evaluates them as separate policies.

Table 6.2: Syntax treatment is selected from the file contract.

Family	Treatment	Reason
SPICE and Ngspice	Purpose-built incremental lexer	Line role, element prefix, node position, expressions, control commands, and fold blocks are domain specific
Verilog and SystemVerilog	Extended QScintilla Verilog lexer	Repairs the stock keyword set and adds current synthesizable SystemVerilog constructs
VHDL, XML, JSON, Python, Markdown, Intel HEX	Stock QScintilla lexers	Existing grammar matches the file formats used inside a project
Modelica	C-family lexer with Modelica vocabulary	Its comments, strings, operators, and numbers fit the host lexer while its keywords remain domain specific
KiCad schematic and board S-expressions	Plain text	Approximate parenthesis coloring would imply semantic understanding that the editor does not have
Unsupported or oversize file	Plain buffer; read-only when oversize	The file remains inspectable without freezing the interface

The SystemVerilog path is visible rather than merely listed. The current lexer repairs a joined keyword defect in the stock set and adds constructs such as `logic`, `typedef`, `enum`, `always_ff`, `always_comb`, `unique`, and `interface`.

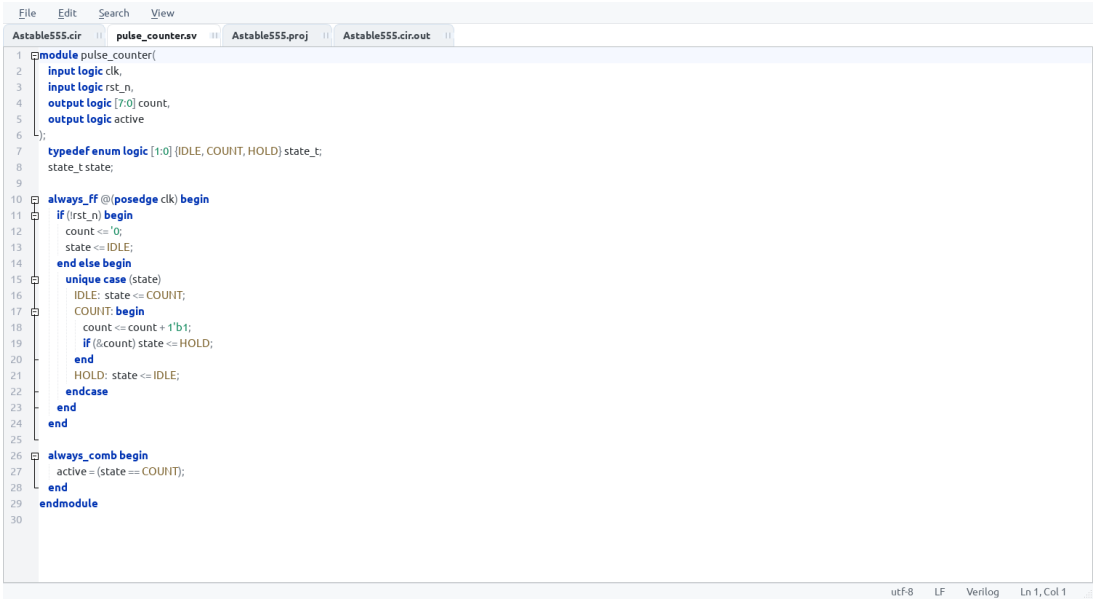


Figure 6.4: The same production editor displaying SystemVerilog. Tabs, fold markers, theme, encoding, line endings, and cursor metadata remain shared, while the lexer and language label change with the active file.

6.5 Saving and external changes

A save creates a temporary file in the same directory, writes the complete buffer with its detected encoding and line-ending convention, flushes it, calls `fsync`, and replaces the original with `os.replace`. If any step fails, the temporary path is removed and the original remains in place.

```
fd, tmp = tempfile.mkstemp(dir=directory, suffix=".tmp")
with os.fdopen(fd, "w", encoding=self.encoding, newline="") as handle:
    handle.write(data)
    handle.flush()
    os.fsync(handle.fileno())
os.replace(tmp, self.file_path)
self._last_signature = self._sig()
```

The recorded signature is updated before the filesystem watcher can report the atomic replacement. A matching event is the editor's own echo and is ignored. A genuine external change follows one of two paths: a clean buffer reloads silently; a dirty buffer keeps its text and receives an inline InfoBar offering *Discard Changes and Reload*. No modal dialog interrupts typing, and no external write silently destroys local edits.

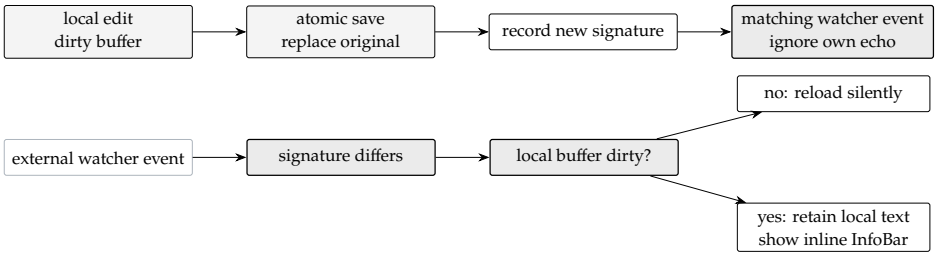


Figure 6.5: Save and watcher ordering. The signature separates an echoed notification from a real external write; local dirty state then decides between silent reload and an inline conflict choice.

Generated `.out` and `.log` files begin read-only because another eSim stage owns their next rewrite. The context menu exposes *Edit Anyway* when the user deliberately takes ownership. This is a reversible guard rather than an attempt to hide the file.

Table 6.3: Editor responses are selected from both file ownership and buffer state.

Event or file contract	Local state	Response
Watcher reports the saved signature	Any	Ignore the editor's own echoed event
External signature differs	Clean buffer	Reload from disk without interrupting the user
External signature differs	Dirty buffer	Retain local text and show the inline reload choice
Generated `.out` or `.log`	Newly opened	Start read-only; expose *Edit Anyway*
File above 2 MB	Newly opened	Use the plain read-only buffer without syntax or folding

Impact

- Project files open in one reusable, multi-tab editor, with a defined plain-editor fallback when QScintilla is unavailable.
- SPICE uses an incremental domain lexer; current HDL and metadata formats select their own explicit treatments.
- Atomic replacement preserves encoding and line endings. Signature-based watching separates self-saves from external changes, while generated and oversize files begin under explicit read-only guards.

Part III

New Capabilities

Chapter 7

Maker and the HDL Workflow

> **Impact**
> The Maker subsystem changed from a collection of conversion dialogs into a coherent HDL workspace. A design can begin inside eSim, remain available between sessions, move freely between authoring, verification, and conversion, and be removed later without leaving a half-installed model behind.

7.1 Why the workflow needed an architectural change

The eSim 2.5 Maker directory contained seven production Python files and 2,764 lines. Its 2.6 counterpart contains 27 files and 14,553 lines. The change in size is useful only because it reflects a change in responsibility: code that had been entangled in large widgets was separated into design ownership, HDL parsing, tool discovery, process control, model registration, and teardown.

The old path assumed that the user already had a Verilog file. A file was selected, different dialogs read it independently, and conversion became the first serious test of whether the source and toolchain were usable. Starting a design in eSim did not have a dependable persistence path, switching views could provoke file-watch prompts, and a long build could make the application appear frozen. Removing a generated model was equally fragmented: each backend knew only part of what it had created.

7.2 One navigator, two language paths

`maker/FlowNavigator.py` is the visible shell. The language control separates the Verilog path from the self-contained VHDL/NGHDL path. In Verilog mode, Author, Verify, and Convert are ordinary tabs by design: they can be visited in any order. The strip explains the normal direction of work without locking an experienced user into a wizard. Expensive panels are created only when first opened, and a failure while constructing one panel is contained behind an actionable placeholder rather than taking down the whole dock.

The Author page now supports a genuine inside-out workflow. *New Verilog Module* creates a named, compilable stub; opening an existing file imports a working copy into the library; and the editor streams changes into the shared

Table 7.1: The ownership boundaries introduced in the Maker subsystem.

Concern	Owner	Practical consequence
Current Verilog design	`DesignBus`	Author, Verify, and Convert see one source instead of passing temporary files between views.
Persistent source library	`verilog_library`	A parseable design receives a workspace-local home named after its top module.
HDL structure	`maker/hdl/ports.py`	Module identity, dependencies, port widths, and testbench generation use one structural interpretation.
Compiler and processes	`CosimConfig`, `hdl.icarus`, `hdl.procs`	Verifier and co-simulation resolve and stop the same Icarus toolchain consistently.
Generated artifacts	`model_registry`, `model_teardown`	A model is listed and removed as an owned set, even when an earlier run left only fragments.

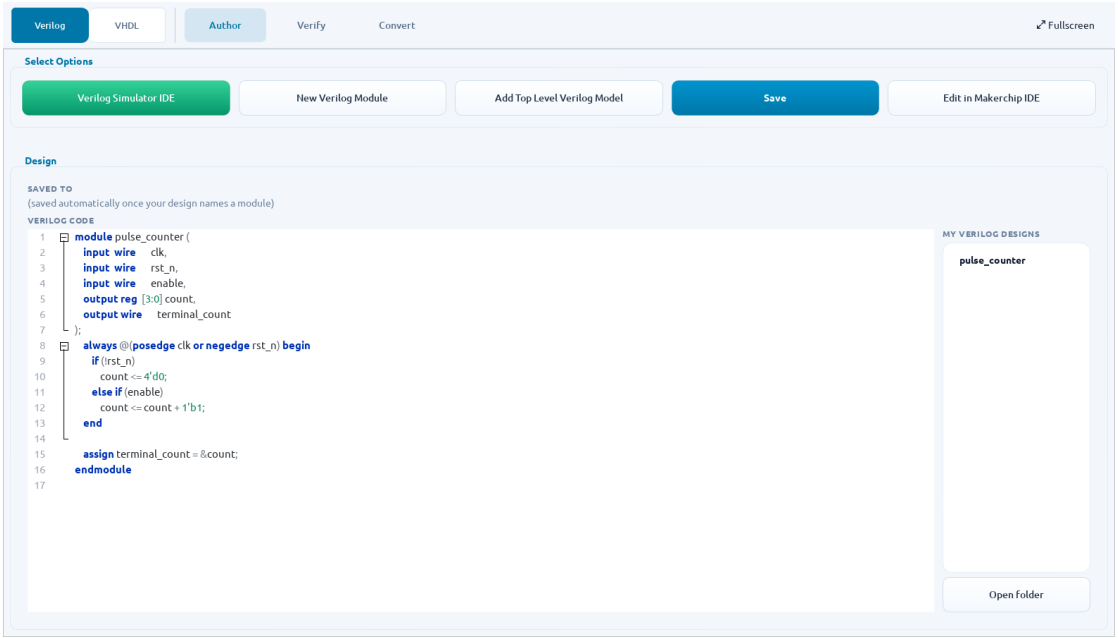


Figure 7.1: The current production Author stage. The source is edited with Verilog syntax highlighting, its workspace-local library is visible beside the editor, and the same header provides direct access to Verify and Convert. This screenshot was captured from the current `FlowNavigator` and `Maker` widgets.

design. The user can still save explicitly, but Save is no longer the operation that makes an in-application design survive.

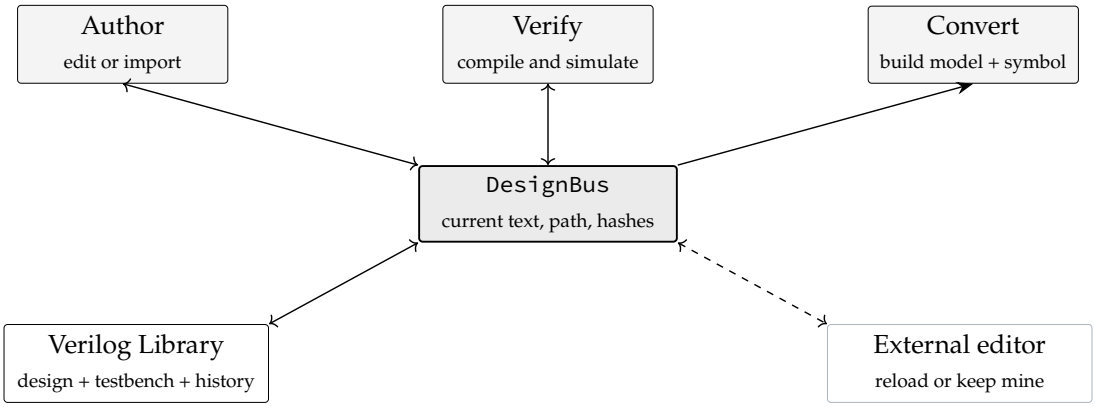


Figure 7.2: The Verilog stages are views around one state owner. Navigation synchronizes memory; the library persists the design; Convert asks for a materialized path only when its file-based toolchain needs one. No connector crosses a node or label.

7.3 A design that has a safe home

`maker/DesignBus.py` keeps the current text, its working path, the optional original import path, and the hash of the last eSim write. Calling `set_content()` is idempotent and starts a short quiet-period timer. Once the text contains complete, parseable modules, `flush_autosave()` writes it under `<workspace>/VerilogLibrary/<top>/<top>.v`. Half-typed module names and incomplete pastes remain safely in memory instead of creating a trail of junk folders.

An imported file is treated differently from a file born in the library. eSim copies the import to its private working library and performs background edits there; the original is written only when the user explicitly presses Save. A pure top-module rename can move an untouched library folder and its generated testbench. Replacing the source with an unrelated design is conservative: the old folder is retained, because deleting work on the strength of a similarity guess would be worse than leaving a harmless copy.

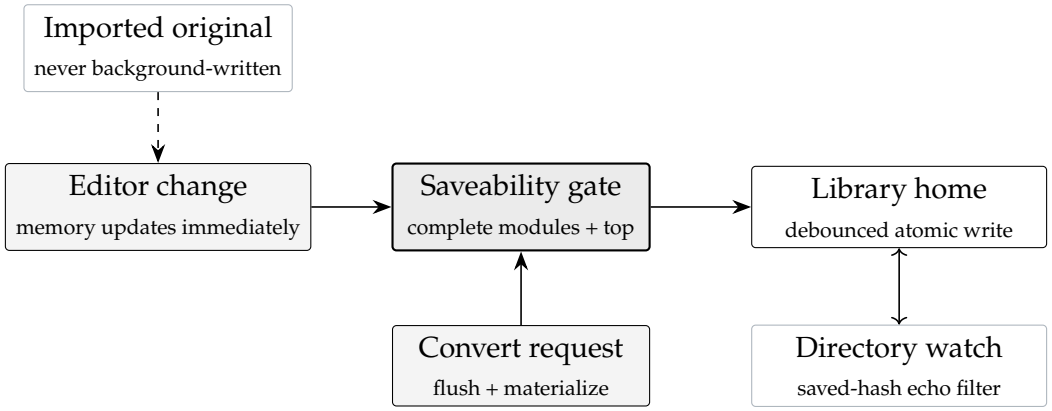


Figure 7.3: Persistence rules in `DesignBus`. The design is responsive in memory, autosave is gated by structural completeness, imports retain a protected origin, and atomic-replace events from editors are detected at the directory boundary.

Watching a directory is necessary because many editors save by replacing the file rather than modifying it in place. Before eSim writes, the bus records the SHA-256 of the bytes it is about to store. A watcher event carrying that same content is an echo and is ignored. Different bytes produce the non-modal *Reload / Keep mine* bar. This small rule removes both noisy self-notifications and silent outside overwrites.

7.4 Makerchip without surrendering file ownership

Makerchip is a browser application, so it cannot directly read and write an arbitrary local design. `maker/MakerchipBridge.py` supplies the missing boundary. It starts a server on `127.0.0.1` with an operating-system-selected port and an unguessable session path, serves one file to the supported Makerchip browser plugin, and accepts debounced edits only through that session. Each read carries a SHA-256 revision. A stale browser write receives a conflict instead of overwriting a newer eSim or external edit.

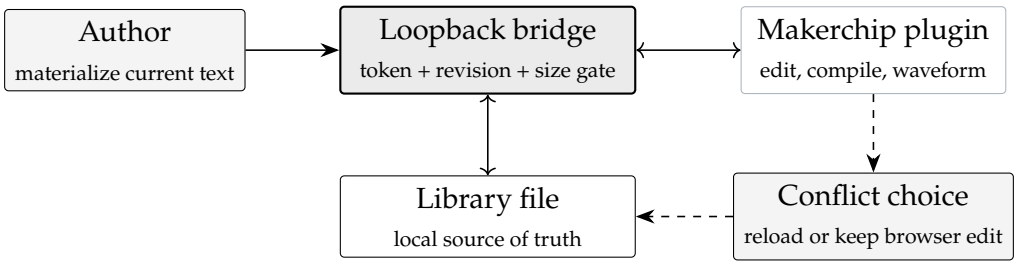


Figure 7.4: Makerchip integration. The remote IDE runs in the browser, while a narrow, token-protected loopback session preserves the local file and detects revision conflicts. The bridge is stopped when its Author panel is destroyed.

The wrapper also gives ordinary Verilog a useful Makerchip simulation harness: scalar inputs are driven from counter bits, common clock and reset names are recognized, and the generated top ends the run after a bounded number of cycles. Compilation status and VCD callbacks open Makerchip's waveform pane automatically, while the local bridge continues to own persistence.

7.5 Removal is part of model creation

Generation is not complete unless its output can later be identified and removed. `maker/RemoveItemsDialog.py` provides search, multi-selection, and backend badges for NgVeri, `d_cosim`, and NGHDL models. The dialog is intentionally only a chooser; irreversible work belongs to the standard-library-only `maker/model_teardown.py`, where it can be tested without Qt.

A single registry file is not trusted as the complete inventory. An interrupted or older installation may leave any subset of five traces: a `modpath.lst` entry, a source build directory, a release build directory, parameter XML, or a KiCad

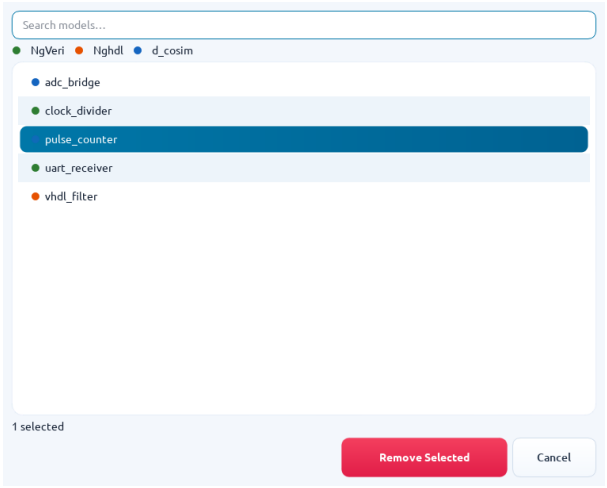


Figure 7.5: The current production removal dialog. Models from all three digital backends are searchable in one list, visibly attributed to their engine, and removable in one operation. The capture uses the real `RemoveItemsDialog`.

symbol. Discovery takes the union of those traces, so an orphan still appears in the dialog. Backend resolution follows deterministic evidence—`d_cosim`, then NGHDL, then NgVeri—and uses case-exact directory listings so Windows and Linux agree about model identity.

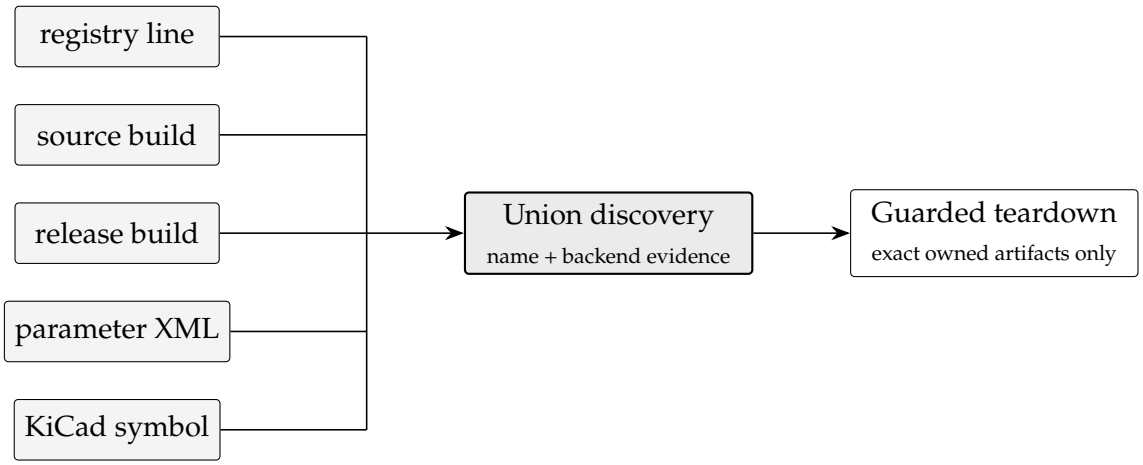


Figure 7.6: Orphan-aware model discovery. Any surviving trace is enough to make a model visible; teardown validates every model name and path before deleting its exact artifact set, then atomically prunes stale or duplicate registry entries.

The path guard rejects blank names, separators, `.`, `..`, and any resolution outside the expected model root. Teardown is absence-tolerant and idempotent: running it again does not broaden the deletion boundary. This is the less visible half of a reliable modeling workflow, but it is what makes repeated classroom experiments and backend switches safe.

Chapter 8

Verilog Verification Inside eSim

> **Impact**
>
> Functional verification is now a first-class stage rather than work that must happen in another application. The same eSim panel holds the design hierarchy, source and testbench editors, compiler diagnostics, run progress, and the route to a full-width waveform view.

8.1 The missing step between source and model

Earlier eSim releases could convert Verilog into a mixed-signal model, but they did not provide a disciplined way to prove the Verilog first. Syntax errors, width mistakes, and testbenches that no longer matched the design often appeared during a heavier conversion path. The failure looked like a model-generation problem even when the model generator had never received valid HDL.

The new Verify stage keeps Icarus Verilog as the execution engine and makes eSim responsible for everything around it: assembling the right sources, explaining structural problems, running the process without freezing the interface, mapping diagnostics back to the correct editor, decoding the VCD off the GUI thread, and opening the result in eSim's plotting surface.

8.2 A layered implementation

`maker/VerilogVerifier.py` is a 3,018-line production widget, but it does not own the compiler algorithms. Most of the mechanics live in the Qt-free `maker/hdl` package: `ports.py` (894 lines) parses structure and generates testbenches; `icarus.py` (498 lines) compiles and simulates; `vcd.py` (284 lines) decodes waveforms; and `procs.py` (56 lines) owns portable process-tree termination. The 59-line `jobs.py` is the narrow Qt boundary that runs a blocking callable on a worker thread and returns progress, success, or failure by queued signals.

The resolver is shared with the `d_cosim` backend of Chapter 9. Environment overrides, the NGHDL configuration, the bundled toolchain, `PATH`, and platform fallbacks are consulted in one order. A Verifier run and a later mixed-signal build therefore cannot silently choose different Icarus installations.

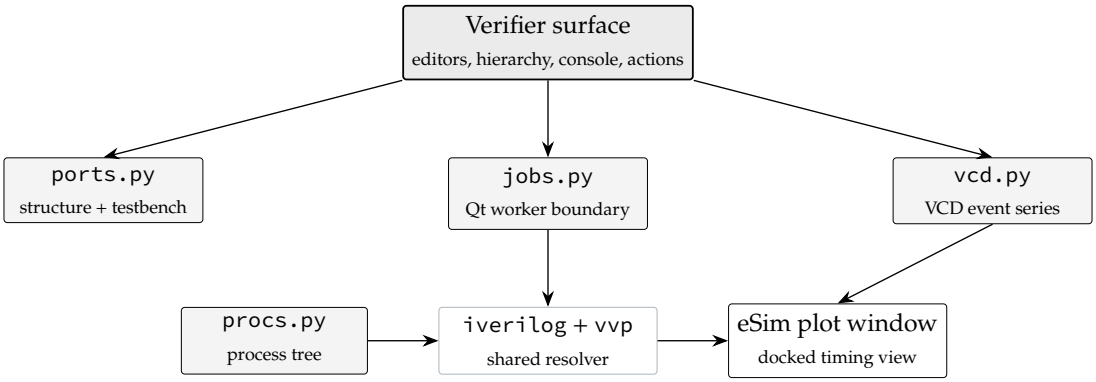


Figure 8.1: Verifier architecture. The widget coordinates user state; the HDL core owns structure, execution, cancellation, and waveform semantics. Only the small job adapter depends on Qt, so the hard parts remain independently testable.

8.3 The verification workspace

The upper workspace combines a reorderable module hierarchy with QScintilla editors. Design tabs may be dragged or moved by keyboard; their order is also the compilation order. The testbench is pinned last and cannot be closed accidentally. File names written into the temporary compile directory are sanitized and de-duplicated, but a reverse map to the owning editor is retained so diagnostics still reach a tab whose visible label contains spaces or parentheses.

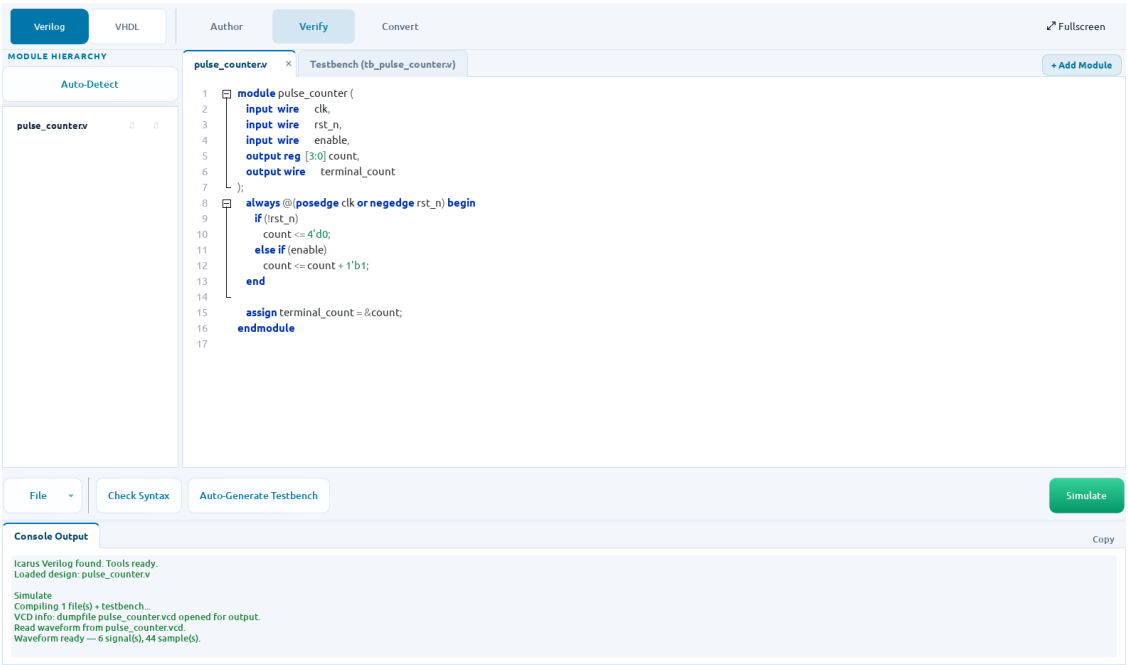


Figure 8.2: A successful run in the current production Verify stage. The design, testbench, module hierarchy, build actions, and the real Icarus result remain visible together. The console records the VCD file and the decoded signal and sample counts.

Verify binds to the `DesignBus` from Chapter 7. Entering the stage renders

the current design; leaving it collects the ordered design tabs into the bus. The testbench has separate ownership and is saved beside the design as `tb_<top>.v`. This prevents the act of concatenating design modules from overwriting verification stimulus.

8.4 Structural parsing and testbench ownership

Verilog is not safely understood by searching for the first word `module`. Comments and strings can contain HDL-looking text; declarations may use ANSI or non-ANSI ports; dimensions may be parameterized; and a file may define several modules in an order that does not match their dependencies. The structural parser first masks comments and strings, then finds module boundaries, ports, widths, parameters, instantiations, and a topological module order. This one interpretation drives the hierarchy, generated testbench, and the model name used by Convert.

Testbench generation is governed by ownership rather than convenience. An empty testbench, or one still marked as generated by eSim, may be regenerated when the module changes. A hand-written testbench is never replaced merely because it does not match; the Verifier explains the mismatch and leaves the text untouched. A self-contained, port-less testbench is recognized and run directly.

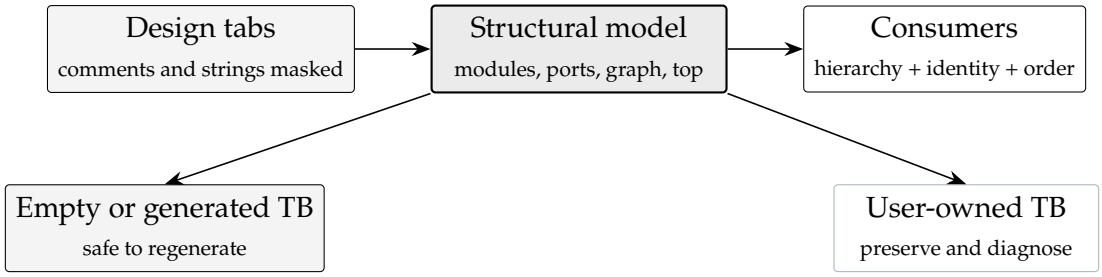


Figure 8.3: One structural model serves the interface and the run. Testbench handling has an explicit ownership branch: eSim may replace its own scaffold, but a user's stimulus is preserved even when it has become stale.

Generated stubs declare width-correct registers and wires, connect the selected module, and add dump and termination scaffolding. Outside testbenches are also accepted. If dump control is absent, a small companion source captures all signals; if termination is absent too, the companion adds a bounded stop. The user's file is not rewritten.

8.5 From Simulate to a waveform

The worker relays compiler and simulator output while it runs, but caps the number of queued live lines so a noisy `$display` loop cannot flood the event queue. Full output remains available to summarize after completion. A cancellation token binds the live child process and also records an early cancellation that arrives

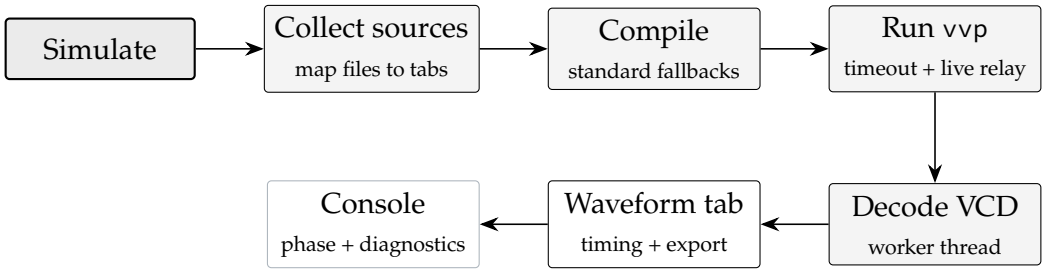


Figure 8.4: The simulation pipeline. Compilation, execution, and VCD parsing remain on the worker side. The GUI receives bounded progress and a completed plot-ready result rather than a blocking subprocess or a large dump to parse.

before spawning; timeouts and Cancel terminate the process tree, not just its immediate shell.

Icarus diagnostics are parsed for file and line information, displayed as clickable console entries, and marked in the owning editor. Syntax checking first compiles the design alone. Only after the design is known good does it check the design with the testbench, which lets the interface say whether the fault belongs to the HDL or to its stimulus.

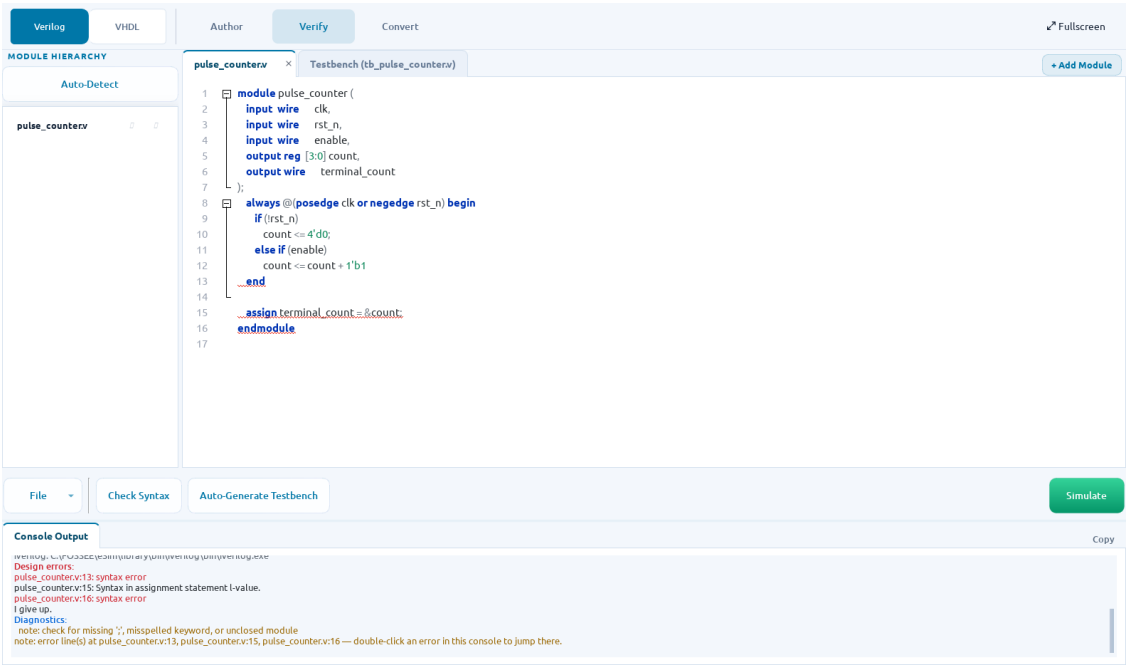


Figure 8.5: A real failed Icarus syntax check in the current Verify stage. The compiler lines identify the source tab and line, the editor marks the affected statements, and the final note tells the user that a console error can be opened directly.

On success, VCD parsing also stays on the worker thread. Sparse value changes become aligned, forward-filled signal series. Raw states remain available for CSV export and warnings about undefined values. The completed plot opens as a full-width eSim tab.

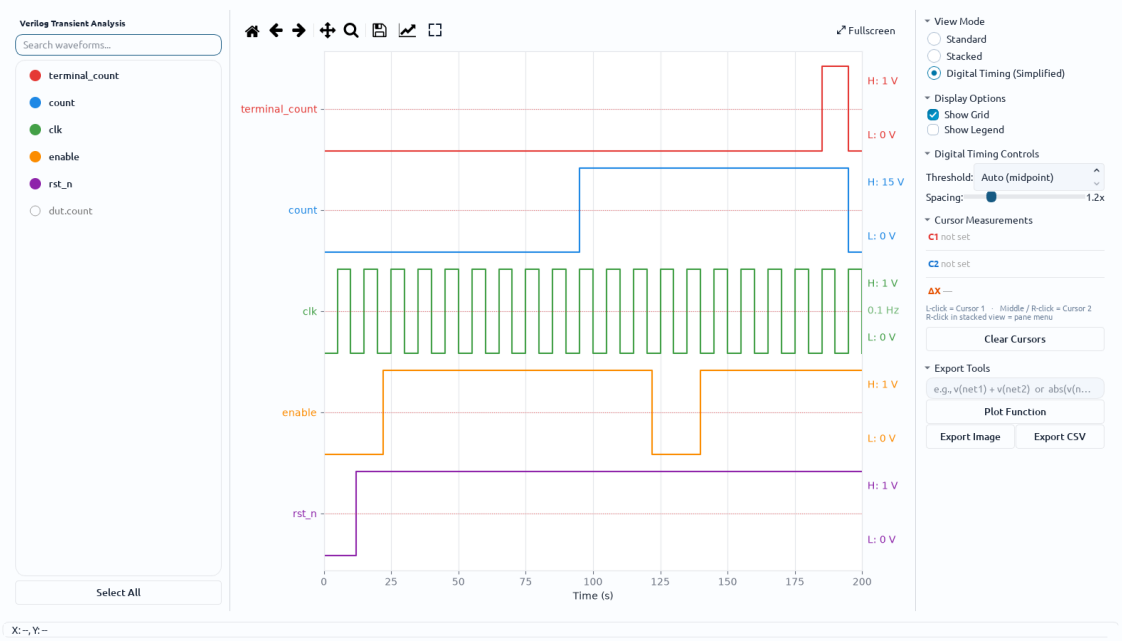


Figure 8.6: The waveform produced by the run in Figure 8.2, displayed by eSim's current production plot window in digital-timing mode. The counter bus, terminal count, clock, enable, and reset can be searched, measured, and exported with the same controls used for circuit results.

8.6 Failure containment and lifecycle

The successful path is only half the engineering problem. A panel may close during a compile, a simulator may never reach `$finish`, a child process may outlive its shell, or a VCD may be huge or malformed. Teardown-critical state is therefore mirrored outside the dying Qt widget: the active worker, cancellation token, and temporary directory can be stopped and cleaned from a destruction hook even when a conventional close event is skipped. Plotting imports remain lazy, so users who never request a waveform do not pay the import cost at startup.

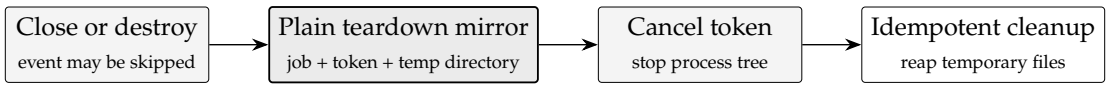


Figure 8.7: Destruction-safe run teardown. Cleanup reads plain mirrored state rather than dereferencing a Qt widget whose native object may already have been deleted.

Table 8.1: Failure surfaces and the response visible to the user.

Failure surface	Containment in code	User-visible result
Missing Icarus tool	Shared resolver and capability lock	Editing remains visible; run actions explain how to locate the executable.
Reserved primitive name	Structural preflight before compile	The module collision is named directly instead of reported as an unexplained syntax error.
Stale generated testbench	Provenance check and regeneration	eSim refreshes only its own scaffold; hand-written stimulus is preserved.
Hung or noisy run	Timeout, process-tree cancellation, bounded relay	The panel stays responsive, shows its phase and elapsed time, and offers Cancel.
Dock destroyed mid-run	Plain teardown mirror and idempotent cleanup	No late callback targets a deleted widget; temporary files and children are reaped.
Sparse or undefined VCD	Worker-side parser and X/Z inspection	Signals align on one timeline; an undriven design receives a specific diagnostic.

Impact

The Verifier creates a reliable boundary between editable HDL and external tools. Source identity is structural, user-owned testbenches are protected, compiler work is cancellable, diagnostics return to the correct tab, and the waveform arrives through a reusable plotting contract. The same boundaries now support pure-digital checking and the mixed-signal work that follows.

Chapter 9

Digital Co-simulation

> **Impact**
>
> eSim 2.6 adds a Verilog co-simulation path in which ngspice solves the analog circuit while Icarus Verilog evaluates the digital design. The finished path handles several digital blocks in one schematic, preserves pin order and time semantics, evaluates valid outputs at the operating point, refreshes rebuilt models before every run, and reports missing tools before launching ngspice.

9.1 The user-facing path

The feature appears in the same Convert stage as the established NgVeri backend. A design written in Author can be built as a native NgVeri code model or as a Dual Co-sim model backed by Icarus and ngspice's `d_cosim` adapter. The two buttons produce the same schematic-level contract: a generated KiCad symbol, a parameter record, and a model that can be placed in an analog circuit. Their compiler and runtime ownership remain separate.

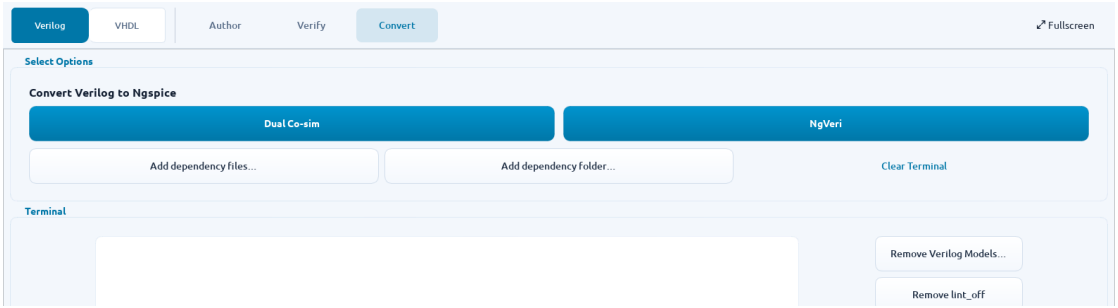


Figure 9.1: The current eSim Convert stage captured from the production application. Dual Co-sim and NgVeri receive equal visual weight, while dependency controls and model maintenance stay in the same workspace. The image is a focused crop of the live interface; unused terminal space below the controls has been omitted.

The compact interface hides substantial backend work. A usable mixed-signal block carries its Verilog source, ordered port manifest, compiled runtime, KiCad symbol, and the metadata needed to rebuild its connections. It also needs the ngspice-to-Icarus adapter and a compatible Icarus runtime. The validation machine

resolved ngspice 45.2 and Icarus 14.0 through the same configuration layer as the GUI.

9.2 From source to one mixed-signal run

The implementation is divided at a deliberate boundary. Model creation records what a digital design *is*; schematic conversion decides how every placed instance participates in this particular simulation. Figure 9.2 shows both phases without collapsing them into one opaque "build" step.

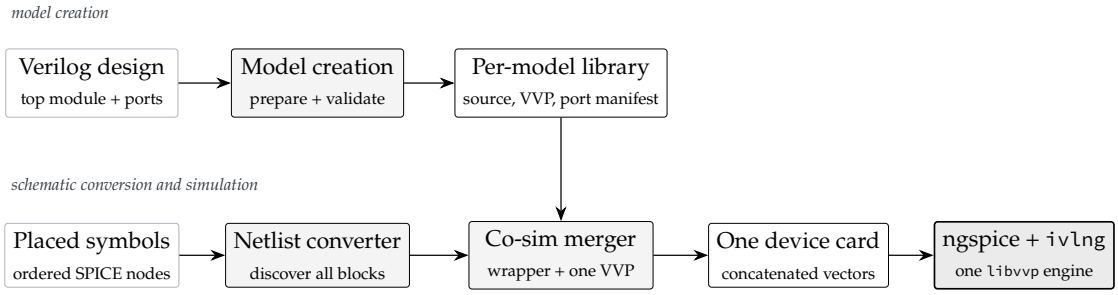


Figure 9.2: The production `d_cosim` path. A model is stored once, but every schematic conversion reconstructs a run from the models actually placed. The converter compiles those instances into one wrapper and rewrites the netlist to one co-simulation device. No arrow crosses a processing node, and each artifact is shown at the stage that owns it.

9.2.1 Model creation

`ModelGeneration.py` parses the selected top module, rejects unsupported `inout` ports, writes `connection_info.txt` in declaration order, and prepares the source for Icarus. A generated top wrapper presents exactly one input vector and one output vector to `ivlng`. Every user port becomes a deterministic slice of those vectors, so VVP's internal port enumeration cannot reorder schematic pins between builds.

The model is stored under a backend-specific root, `DIGITAL_MODEL/NgVeriCosim/<model>`. That directory is a sibling of the legacy NgVeri tree, not a child of it. A model rebuilt or removed through one backend therefore cannot overwrite the other backend's compiled files. The same separation is reflected in the KiCad symbol library and parameter XML.

9.2.2 Schematic conversion

`cosim_merge.py` reads every placed co-simulation block, checks the current port widths against its symbol, copies each distinct source once, and generates `esim_cosim_top`. Two placements of the same module receive different instance labels and disjoint vector slices. The compiled artifact, `esim_cosim_merged`, is staged beside the circuit netlist because `ivlng` resolves `sim_args` relative to ngspice's working directory.

The rewritten circuit contains one digital device regardless of the number of placed blocks. Its essential form is:

```
a_esim_cosim [all_input_nodes] [all_output_nodes] esim_cosim_model
.model esim_cosim_model d_cosim simulation="ivlng" \
    lib_args=["libvvp", "ivlng"] sim_args=["esim_cosim_merged"]
```

The long node vectors are not arbitrary packing. Inputs from every block are concatenated in schematic order, followed by outputs in the same order used by the generated wrapper. A width mismatch is treated as a stale-symbol error and stops conversion. Guessing the missing width would create a netlist that runs with the wrong wiring, which is more dangerous than a visible failure.

9.3 The one-engine constraint

The first design treated every placed Verilog block as a separate `d_cosim` device. That mapping appears natural, but `ivlng` loads Icarus through `libvvp`, whose simulator state is process-global and single-use. The first device consumes the engine. A second device attempts to reuse it, reports that the VVP simulation has already run, and can terminate ngspice without producing output. Renaming a second copy of the library does not create an independent engine because the VPI adapter still binds to the first library by name.

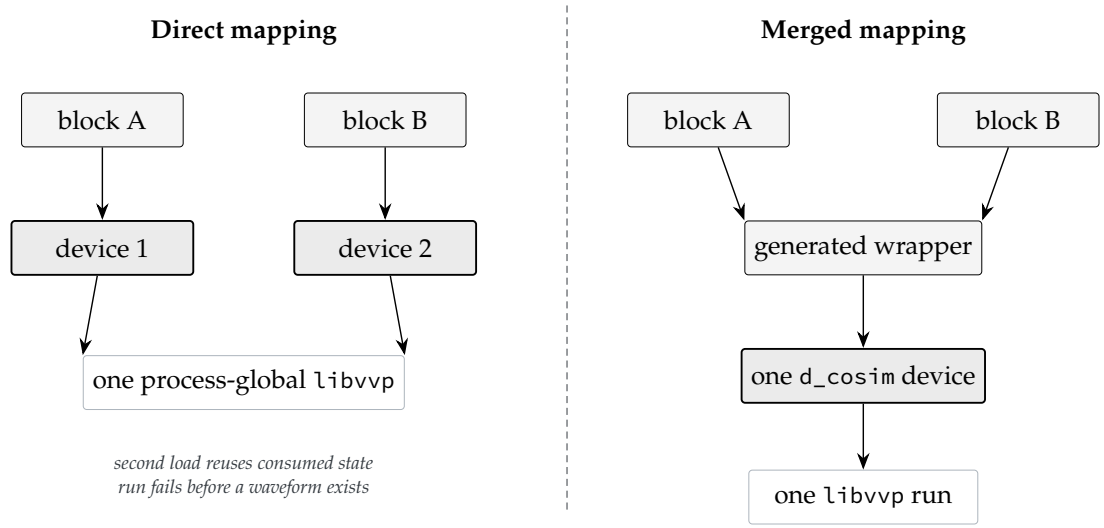


Figure 9.3: Why the converter merges blocks. The original one-device-per-block mapping asks a single-use runtime to start twice. The current mapping moves multiplicity into generated Verilog, where one engine can instantiate any number of designs while SPICE nodes remain separate.

This correction is architectural rather than a special case for "two blocks." The generated wrapper can contain two instances of one module, many different modules, or any mixture of the two. Duplicate schematic instance names and stale port manifests are rejected before compilation. Source-library search paths are

retained for dependency modules, so merging does not reduce the Verilog that an individual model may use.

9.4 Correctness at the analog–digital boundary

A mixed-signal simulator can complete successfully and still be wrong. Three such cases were found in the boundary between ngspice time and Verilog events: threshold uncertainty, time precision, and initialization.

9.4.1 A ramp must produce one digital edge

The standard ngspice `adc_bridge` emits `x` while its input lies between `in_low` and `in_high`. A four-state Verilog simulator can interpret a rising sequence `0 -> x -> 1` as two positive-edge events. That made counters advance twice on one analog clock ramp. NgVeri's generated C already treats every state other than definite zero as one, so the Icarus wrapper applies that rule explicitly, bit by bit, with case equality.

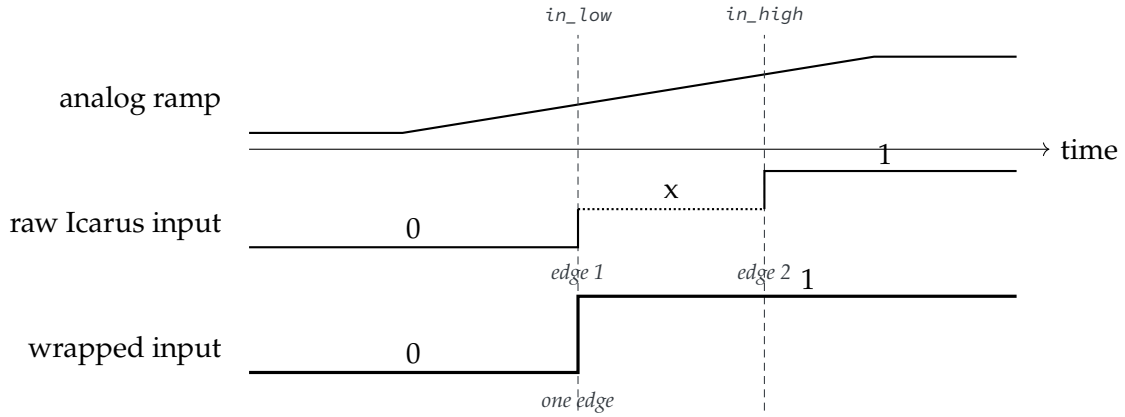


Figure 9.4: Threshold handling in the generated wrapper. The uncertain band is real electrical information, but exposing it as a second edge changes the digital design. Mapping each uncertain bit to one collapses the transition to the single edge already used by the NgVeri backend.

9.4.2 Time precision and the operating point

`ivlng` converts elapsed SPICE time into VVP ticks. If a source declares a precision coarser than the analog step, that division truncates to zero and the digital model never advances. eSim preserves the design's time unit, because changing it would alter every `#delay`, but sharpens only the precision to at least 1 ps. A missing directive receives `‘timescale 1ns/1ps`. The wrapper carries the same normalized directive as the design so both share one time base.

Initialization required a simulator-side correction. The original `d_cosim` code model transferred inputs at time zero without evaluating the co-simulator, so a

combinational high output or an `initial` value was reported as zero until the first transient step. The bundled ngspice patch evaluates the digital model during the operating point. A mixed-signal inverter therefore has the correct output at $t = 0$, and that output still responds to a later input transition.

Table 9.1: Boundary failures and the layer that owns each correction.

Failure	Observable result	Correction and owner
ADC uncertainty	one analog ramp clocks a sequential model twice	per-bit `x`-to-one mapping in the generated Verilog wrapper
Coarse Verilog precision	ngspice finishes while digital outputs remain frozen	normalize precision before Icarus compilation; retain the source time unit
Port enumeration drift	the same source can reconnect bits differently after a rebuild	one input vector and one output vector, sliced arithmetically
Unevaluated operating point	correct high outputs appear as zero at $t = 0$	evaluate `d_cosim` during ngspice initialization
Stale staged VVP	rebuilding a model appears to have no effect	compare modification times and restage at every simulation start and redo

9.5 Toolchain resolution and run control

`CosimConfig.py` is the shared resolver for the Verilog verifier, Dual Co-sim builder, converter, simulator, and doctor. Resolution is performed at call time, so a developer can change an override without restarting the Python process. The search order is consistent across tools:

1. an explicit environment override such as `ESIM_IVERILOG`;
2. the installed `\textasciitilde/.nghdl/config.ini`;
3. an eSim-bundled executable or library;
4. the operating-system `PATH`, followed by guarded Windows fallbacks.

Capability checks go beyond finding executable names. A Dual Co-sim run requires the ngspice code-model directory to contain `ivlng`, and the Icarus prefix to contain `libvvp`. On Windows, the simulator process also receives the ngspice code-model directory and the Icarus library directory on `PATH`; on POSIX, `libvvp` is added through `LD_LIBRARY_PATH`.

`NgspiceWidget.py` recognizes a `d_cosim` netlist before starting the process. It omits the normal raw-file switch because the co-simulation analysis lives in the netlist's `.control` block, adds the loader paths, and restages any VVP newer than the copy beside the netlist. The redo action performs the same refresh. Output is streamed to the simulation terminal while `CosimLogger.py` mirrors structured phases to a rotating diagnostic log.

9.6 End-to-end validation

The validation run used for Figure 9.5 is intentionally harder than a single truth table. It places two instances of a four-bit counter and one two-bit toggle in one Icarus engine. Counter A leaves reset at 2 ms; Counter B leaves reset at 5 ms. If instance slices are crossed, the counters become identical. If the uncertain ADC band creates two edges, their values advance twice as fast. If the second instance still owns a second VVP engine, the run produces no waveform.

Table 9.2: Selected decoded states from the real three-block run.

Sample time	Counter A	Counter B	Two-bit toggle
2.05 ms	1	0	1
3.05 ms	2	0	2
5.05 ms	4	1	0
8.05 ms	7	4	3
11.05 ms	10	7	2

The early samples show Counter B remaining at reset while Counter A and the toggle advance; the last sample confirms that the three-millisecond counter offset persists across the run. The automated gate decodes these buses as integers instead of accepting a successful process exit. Other tests check the wrap strobe, startup values, time normalization, repeated or mixed placements, rebuilt-model restaging, backend isolation, and dependency errors. Each protects against a failure that could otherwise produce a plausible but incorrect waveform.

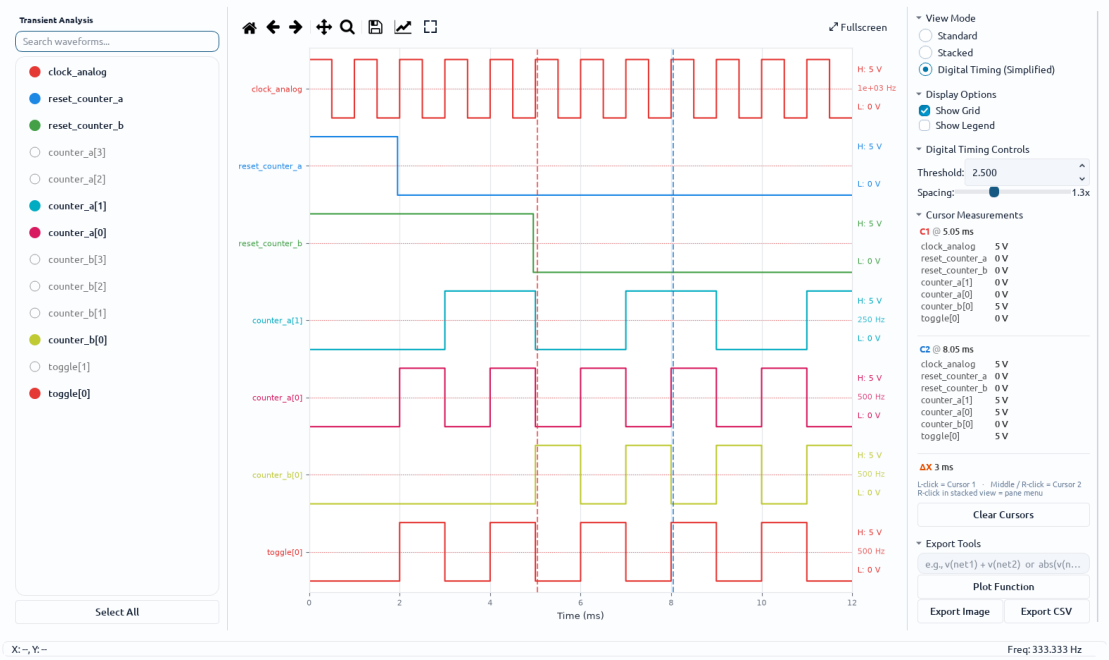


Figure 9.5: A fresh production capture of the real merged co-simulation. The installed Icarus toolchain compiled two counter instances and one toggle into `esim_cosim_merged`; ngspice 45.2 executed it through `ivlng`; and eSim's current Digital Timing view displayed the resulting nodes. Cursor readouts at 5.05 ms and 8.05 ms expose the three-millisecond offset between the counters instead of relying on visual resemblance alone.

Impact

The completed path is not a thin button over Icarus. It is a reproducible model contract, a converter that respects a single-use runtime, an explicit analog–digital boundary, and a run controller that keeps artifacts and loader paths current. Those layers make several Verilog blocks behave as ordinary components in one analog simulation while retaining enough diagnostics to explain a failure before it becomes a silent result.

Chapter 10

NGHDL and Makerchip Integration

> **Impact**
> The HDL workspace now gives VHDL and browser-based authoring explicit system boundaries. NGHDL owns its generated code model, socket server, compiler processes, symbols, and teardown. Makerchip remains an external editor connected through a short-lived loopback session with revision checks. Both paths return to eSim's model and waveform workflows without sharing build directories or silently replacing the local design.

10.1 Two integrations, two responsibilities

NGHDL and Makerchip appear together in the HDL workspace, but they solve different problems. NGHDL turns a VHDL entity into an ngspice code model whose runtime talks to a GHDL testbench server. Makerchip provides a browser IDE for Verilog and TL-Verilog; eSim owns the file and synchronization boundary around that editor. Treating both as a generic "HDL tool" would hide the decisions that determine correctness: compiler backend and socket lifecycle for NGHDL, revision and source ownership for Makerchip.

The top language control therefore switches between the established Verilog Author–Verify–Convert path and the dedicated VHDL model builder shown in Figure 10.1. The NGHDL interface is embedded lazily. A toolchain preflight runs before construction, so a missing compiler or code model produces an actionable placeholder instead of a half-built panel.

This is a real integration change, not only a restyled NGHDL window. The earlier toolbar action validated an external `nghdl` executable and launched `nghdl -e` as a separate process. eSim did not own that window, its project identity, or its teardown. The current NGHDL and Makerchip launchers enter the same single-instance Model Creation dock: NGHDL selects the VHDL path, Makerchip selects the Verilog path, and repeated clicks focus and switch the existing dock instead of stacking another tool window. Closing the project now reaps the dock and its associated worker state.

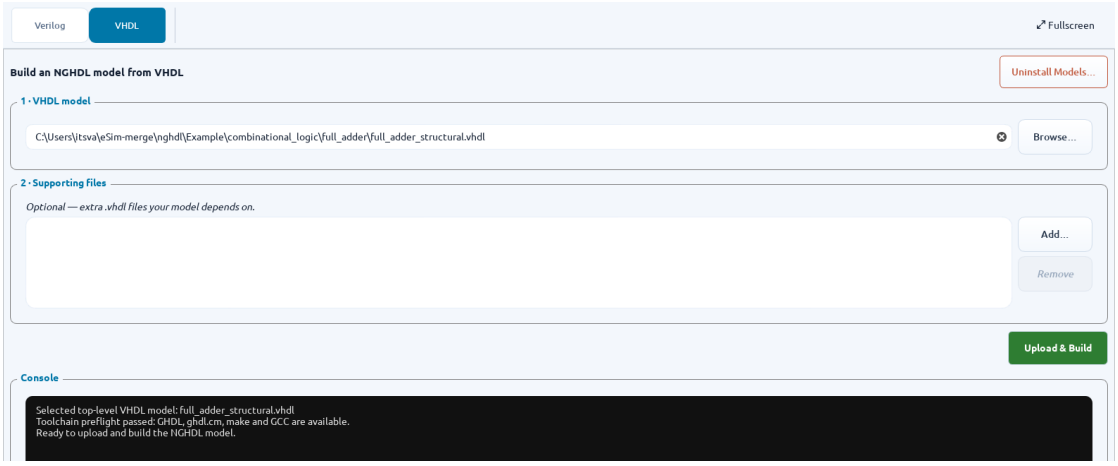


Figure 10.1: The current NGHDL builder captured from eSim 2.6 with a repository VHDL example selected. The page follows the actual task: choose a top-level model, add optional dependencies, upload and build, then inspect the console. Model uninstall is visibly separate from removing a dependency from the staging list. The console records a successful production toolchain preflight on the capture machine.

10.2 How one VHDL entity becomes a circuit block

An NGHDL build does not feed the user's VHDL directly to ngspice. It generates the two halves of a protocol: an XSPICE client compiled into `ghdl.cm` and a GHDL testbench that hosts the user's entity. The client lives inside ngspice; the server lives in a separate process. Their messages carry the current input bits and the resulting output bits over loopback.

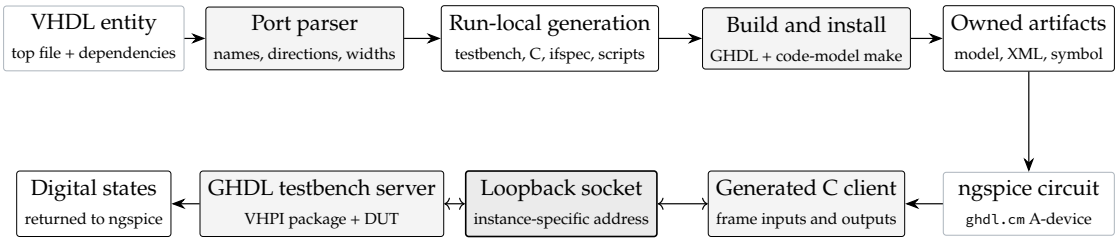


Figure 10.2: NGHDL build and runtime ownership. Generation proceeds left to right on the upper lane. Simulation returns right to left on the lower lane, where a compiled XSPICE client and a GHDL testbench exchange digital state over loopback. Build artifacts and live processes are kept distinct.

10.2.1 Port information is a protocol contract

`nghdl/src/model_generation.py` derives one canonical model stem from the final filename extension and writes every generated file into a temporary run directory. This removed two sources of drift: dotted filenames no longer produce different names in the testbench and model tree, and a read-only launch directory can no longer break generation before the build begins.

Each connection record has three fields: port name, direction, and bit width. The parser classifies the direction field itself rather than searching the whole line for the text `in` or `out`. A port named `sout` is therefore not mistaken for both directions. That ordered manifest drives four outputs:

- the generated VHDL testbench and its DUT port map;
- the C client's input and output framing;
- the XSPICE `ifspec.ifs` port tables; and
- the KiCad symbol and parameter XML used by the schematic converter.

One parse feeds all four consumers. Changing the order in only one output would rewire the schematic without a compiler error, so the shared manifest is part of the simulation protocol rather than incidental metadata.

10.2.2 Generated files and installed ownership

The build produces more than a testbench. `cfunc.mod` implements the ngspice-side client, `ifspec.ifs` declares the code-model interface, `<model>_tb.vhdl` hosts the design, and `start_server.sh` analyzes, elaborates, and starts the testbench. A second script writes an instance-specific VHDL package containing the loopback address and port.

Successful installation registers an exact model line, copies sources into the NGHDL model tree, rebuilds `ghdl.cm`, writes the parameter XML, and updates `eSim_Nghdl.kicad_sym`. Removal walks the inverse ownership set: registry line, symbol, XML, source tree, and release tree. Every step tolerates an already-absent artifact, while guarded path construction prevents an empty or malformed model name from becoming a broad recursive deletion.

10.3 The live socket exchange

At simulation initialization, the generated C model selects a port from the instance identifier and starts the matching testbench server. Inputs are serialized as named bit strings, sent to the server, applied to VHDL signals, and followed by a reply containing the output strings. The XSPICE event state keeps current and previous output values so rise and fall delays can be applied without confusing a port tag with a historical time point.

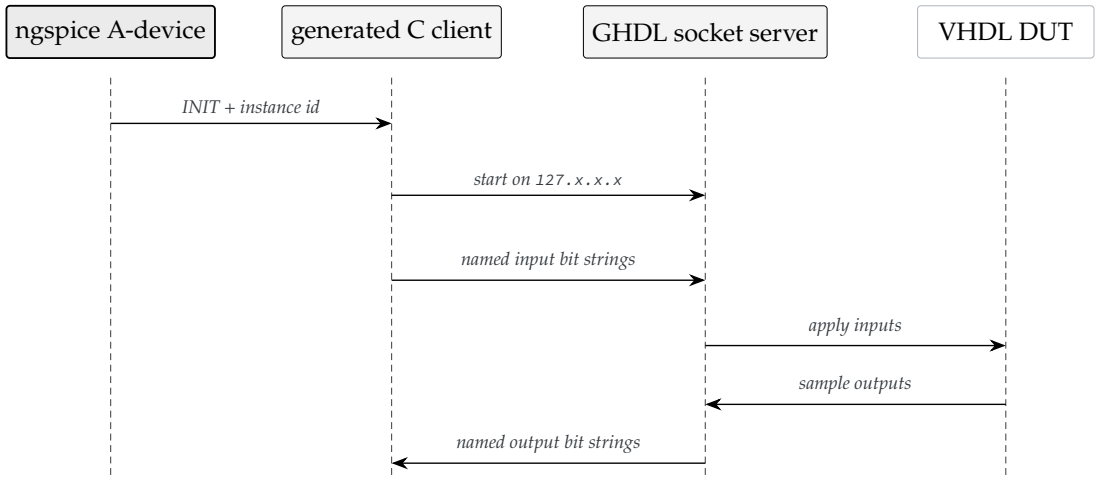


Figure 10.3: One NGHDL simulation exchange. The server is reachable only through loopback. Messages retain port names as well as bits, while the generated testbench performs the VHDL signal conversion around the DUT.

The client and server buffers are now sized from the real port framing. Before generation begins, the input message is checked against the server's 4096-byte receive capacity and the output reply against its 2048-byte capacity. A model that cannot fit is refused with a width-specific explanation. Previously, a fixed 1024-byte client buffer could truncate a wide interface and allow a simulation to continue with partial values.

The server binds to loopback rather than every network interface. This is a safety boundary and an architectural statement: the protocol links two local processes; it is not a network service. Verbose iteration traces go to the run's client log rather than flooding the GUI console, so compiler and simulator failures remain visible.

10.4 Selecting a GHDL backend by capability

NGHDL elaborates its generated testbench with `-Wl,ghdlserver.o`. The GHDL LLVM and GCC backends can pass that object to a linker; the mcode JIT backend cannot. On Ubuntu 26.04, installing the generic `ghdl` meta-package can select mcode first. On Ubuntu 24.04, the LLVM driver can print a version even when its backend library is missing. A version probe alone is therefore insufficient in both cases.

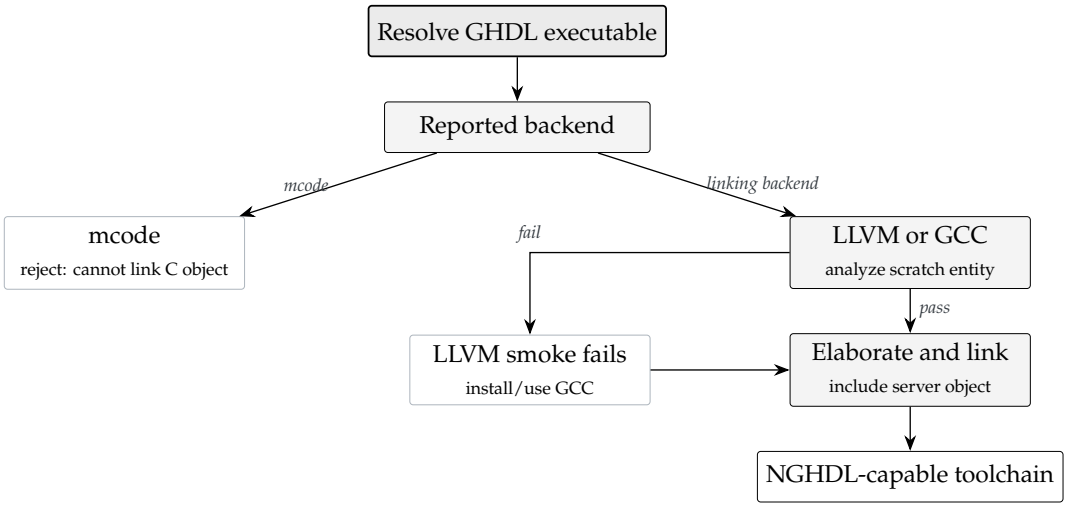


Figure 10.4: Capability-based GHDL selection. The gate tests the analyze, elaborate, and linker operations NGHDL actually needs. A working GCC backend is an accepted fallback when the installed LLVM backend cannot compile.

The same rule is enforced in installation, the command-line doctor, the Help dialog, and the VHDL stage preflight. A machine cannot be declared ready by one surface and rejected later by another.

Table 10.1: NGHDL failure modes converted into explicit contracts.

Old failure mode	Why it was difficult to diagnose	Current contract
Generation in process CWD	packaged or read-only launch directories fail on the first write	every run writes into an owned temporary directory
Dotted model filenames	generator outputs disagree on the model identity	one canonical stem from the final extension
Substring direction parsing	names containing “in” or “out” enter the wrong port list	classify only the manifest’s direction field
Fixed client buffers	wide interfaces truncate into plausible wrong values	size from framing and refuse messages beyond server limits
Tag/timepoint confusion	later output ports freeze after their first event	each port tag reads current timepoint 0 and previous timepoint 1
Wildcard network bind	local simulation becomes an exposed service	bind the server to loopback only
GUI-thread teardown	removing models freezes the interface	guarded worker, queued progress, and bounded close wait

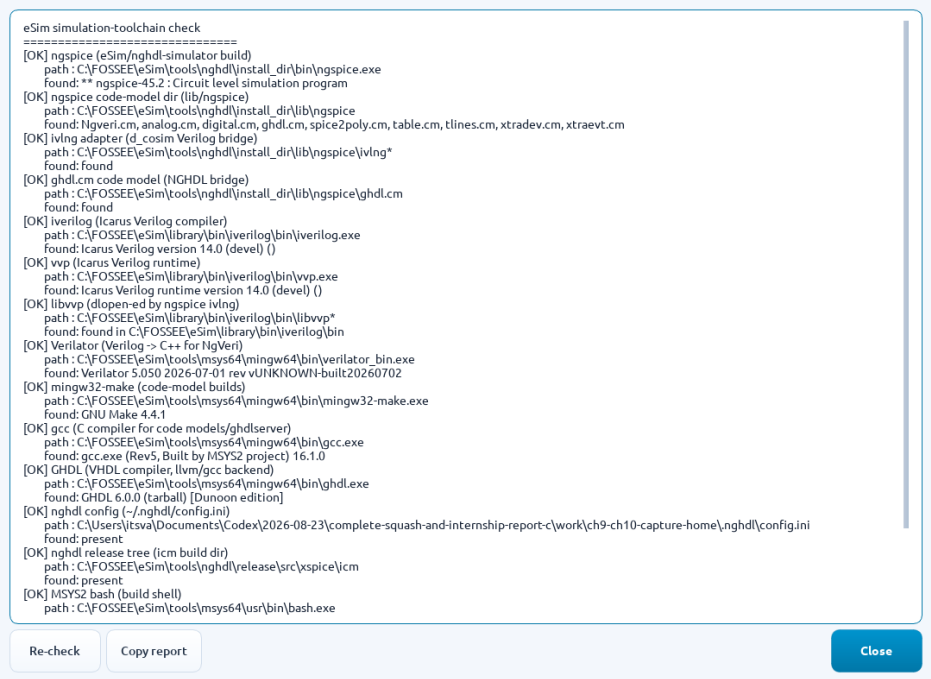


Figure 10.5: The production toolchain doctor on the report machine. The visible report comes from the same probe functions used by flow preflights. It checks the executable, ngspice code models, `ivlng`, `libvvp`, Verilator, build tools, GHDL, the NGHDL release tree, and the Windows MSYS2 runtime.

10.5 Makerchip as a revisioned external editor

Chapter 7 covers the Author workspace and its design bus. The integration boundary matters here because Makerchip is deliberately not an embedded copy of that editor. When the user chooses Makerchip, eSim materializes the current design, starts a loopback bridge for that file, and opens a session-bearing URL in the supported browser plugin. The bridge owns a random token and a revision counter; the browser never receives an unrestricted local file path or a persistent server endpoint.

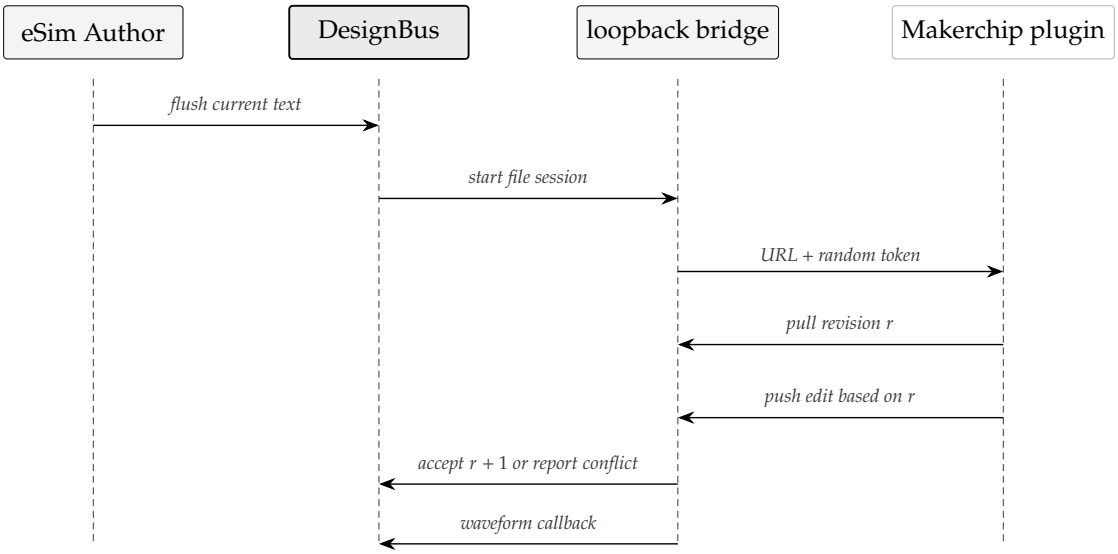


Figure 10.6: Makerchip session lifecycle. A browser edit is accepted only against the revision it read. If eSim and the browser change the design concurrently, the bridge reports a conflict instead of selecting a winner. Simulation data returns through an explicit callback to eSim's waveform destination.

The bridge listens on `127.0.0.1`, validates the token on every request, and rejects stale revisions. Author shutdown calls `stop()` explicitly; it does not wait for Python garbage collection to release a listening socket. If the operating system cannot open the browser, the newly created bridge is stopped immediately. The local design remains authoritative throughout this lifecycle, and accepted browser edits re-enter through the same design bus used by Author, Verify, and Convert.

Makerchip-generated waveforms also have a named return path. The plugin posts the waveform payload to the session endpoint, the bridge validates it, and the Author integration emits a request for eSim's waveform tab. Closing the browser does not strand simulation data in an external tab, and opening Makerchip does not create a second model registry.

10.6 Backend boundaries in the shared workspace

The common interface is intentionally narrower than the tools behind it. Each backend keeps its own compiler, build root, runtime artifact, and teardown; only design identity, capability reporting, symbol registration, and waveform destination are shared.

Table 10.2: Contracts exposed by the HDL workspace.

Path	Engine	Owned result	Purpose
Verify	Icarus + VVP	temporary executable, VCD, diagnostics	fast functional proof before model creation
Dual Co-sim	Icarus + ngspice `d_cosim`	backend-specific VVP, symbol, parameter XML	Verilog inside a mixed-signal run
NgVeri	Verilator + ngspice	native code-model sources and library entry	established compiled Verilog model path
NGHDL	GHDL + socket server + ngspice	linked testbench server, code-model entry, symbol, XML	VHDL mixed-signal co-simulation
Makerchip	supported browser plugin	revisioned edit session and waveform handoff	external Verilog and TL-Verilog authoring

These boundaries keep a convenient front end from becoming shared mutable state. NGHDL teardown cannot delete a Verilog model, and a Makerchip session cannot replace a newer local edit. VHDL users get capability checks, owned artifacts, exact cleanup, and a bounded client–server runtime; browser users retain the maintained Makerchip editor while eSim preserves source ownership, detects revision conflicts, and routes every accepted waveform back to its plotting workspace.

Chapter 11

KiCad 7–10 Netlisting

> **Impact**
>
> eSim no longer depends on KiCad's lossy SPICE exporter. It asks KiCad for a connectivity-preserving XML netlist, reconstructs eSim's SPICE semantics, and keeps the last usable circuit whenever regeneration fails. The same parsed circuit then feeds a modernized conversion workspace with per-window state and grouped model assignment.

11.1 Why the old export stopped being usable

KiCad remains eSim's schematic editor, but modern KiCad does not know the conventions encoded in eSim's generated symbols. In particular, its direct SPICE export can reduce such parts to placeholder rows with no electrical terminals. The ordinary resistor and capacitor rows survive, which makes the defect easy to misread as a model problem. It is actually a loss of connectivity at the export boundary.

Figure 11.1 follows one JFET amplifier through both routes. The broken row `J1 __J1` has a reference and nothing else; the corrected row restores drain, gate, source, and model in their electrical order. Sources and plot probes recover their nodes for the same reason.

Direct KiCad SPICE export

```
.title KiCad schematic
C6 Net-_J1-S_ GND 0.1u
R3 Net-_v2-+_ out 3k
v2 __v2
U2 __U2
J1 __J1
v1 __v1
.end
```

eSim structured conversion

```
* FET_Amplifier
c6 net__j1_s 0 0.1u
j1 out net__j1_g net__j1_s eSim_NJF
r3 net__v2 out 3k
u1 in plot_v1
u2 out plot_v1
v1 in 0 sine
v2 net__v2 0 DC
.end
```

Figure 11.1: The same schematic at the compatibility boundary. The direct export loses the terminals of eSim-specific symbols; the structured route preserves enough information to reconstruct a simulatable SPICE row.

Keeping the old bundled KiCad hid this defect; it avoided the changed boundary instead of making that boundary dependable. Supporting four modern KiCad generations required eSim to own the translation rules specific to eSim.

11.2 A structured boundary instead of a text patch

The replacement lives in `src/kicadtoNgspice/KicadNetlister.py`. It deliberately does not request `--format spice`. Instead it runs `kicad-cli sch export netlist --format kicadxml`, parses components, fields, pins, and nets from XML, and emits the flat `.cir` file expected by the existing converter. Figure 11.2 separates the jobs clearly: KiCad owns schematic decoding; eSim owns electrical meaning.

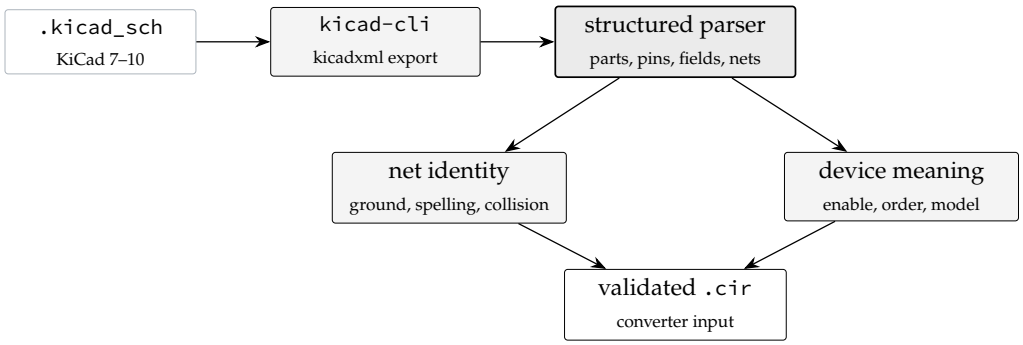


Figure 11.2: The maintained netlisting boundary. XML preserves structure; two explicit translation stages recover electrical identity and component semantics before the working circuit is replaced.

11.2.1 Electrical identity

Node spelling is not cosmetic in ngspice. Characters that KiCad uses in automatic names, including `+`, `-`, and `/`, are operators inside expressions such as `v(node)`. The netlister lower-cases names and maps characters outside `[a-z0-9_]` to underscores. If two original names would then become identical, their KiCad net codes disambiguate them. This prevents sanitization from introducing an electrical short. Ground is handled separately: `GND`, `esim_gnd`, and the literal `0` intentionally converge on SPICE node zero.

11.2.2 Component semantics

The XML fields carry decisions that a generic exporter cannot infer. The translator honours a valid `Spice_Node_Sequence`; otherwise, numbered pins are emitted in numeric order and non-numeric pins retain their encounter order. Field lookup is case-insensitive. A disable value of `n`, `no`, `false`, or `0` removes the component rather than depending on one exact capitalization.

Subcircuit classification is equally deliberate. A matching model description

takes precedence over a same-named `.sub` file. Otherwise the project directory is searched before the system library, and subcircuit instances receive the required `X` prefix. KiCad's own simulation metadata is also recognised when it explicitly marks a device as SPICE type `X`.

Table 11.1: Translation contracts enforced by the structured netlister.

Contract	Risk at the boundary	Rule used by eSim
Ground identity	Ground becomes a floating named node	Canonical ground labels map to node zero.
Safe node spelling	A plotted node is parsed as an expression	Unsupported characters become underscores.
Collision freedom	Two distinct wires become one node	Collisions are disambiguated with KiCad net codes.
Terminal order	A device is connected to the wrong terminals	Numeric pin order is the default; a valid symbol override takes precedence.
Enablement	Helper or disabled symbols become devices	Field names and common false values are matched case-insensitively.
Model ownership	A model card is mistaken for a subcircuit	Model XML takes precedence, followed by project and system subcircuit lookup.

11.3 Failure must preserve the last usable circuit

Regeneration is an improvement layered over a project that may already contain a working `.cir`. Destroying that file because KiCad is absent, an export times out, XML is malformed, or a schematic contains no components would turn a recoverable problem into data loss. The implemented rule is therefore simple: every pre-write failure returns without touching the existing circuit. The intermediate XML file is removed on both the success and failure paths.

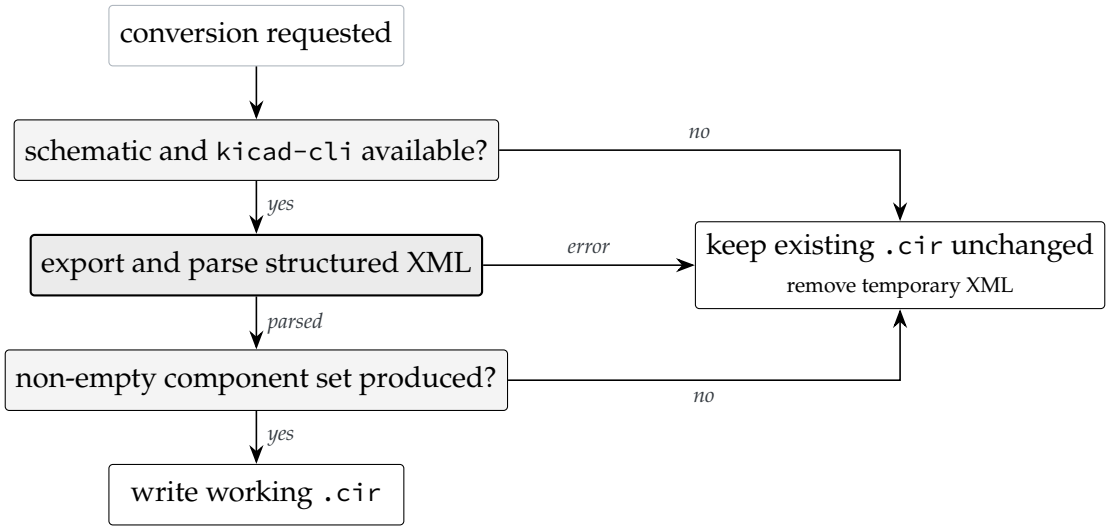


Figure 11.3: Preserved-on-failure output. Only a successfully parsed, non-empty component set reaches the write path; every earlier failure leaves the user's previous netlist intact.

This is intentionally described as *preserved on failure*, not as an atomic file replacement. Validation happens before the write, but the current implementation writes the accepted text directly. That distinction matters in a technical report because it states exactly what the code guarantees.

11.4 The conversion workspace after netlisting

The generated `.cir` still passes through eSim's established conversion workspace, where analysis commands, sources, ngspice models, device libraries, subcircuits, and microcontrollers are completed before `.cir.out` is written. The workspace remains familiar, but its state model has changed substantially.

Each open converter now owns a separate `TrackWidget`; parsed rows and user entries are injected into its tabs rather than shared through class-level state. The window records the source file's modification time and reloads if KiCad regenerates the circuit while the converter is open. An empty or half-parsed model is never serialized. Conversion errors are collected and shown with a corrective hint instead of leaking a partial result.

Repeated subcircuits no longer require the same directory to be selected instance by instance. The converter groups rows by model, lets one validated path fan out to all members, and still supports an override for a single placement. Port count is checked for each instance before the group assignment is accepted. Internally, a separate entry is retained for each reference. The output writer and saved values therefore receive the same precise instance mapping.

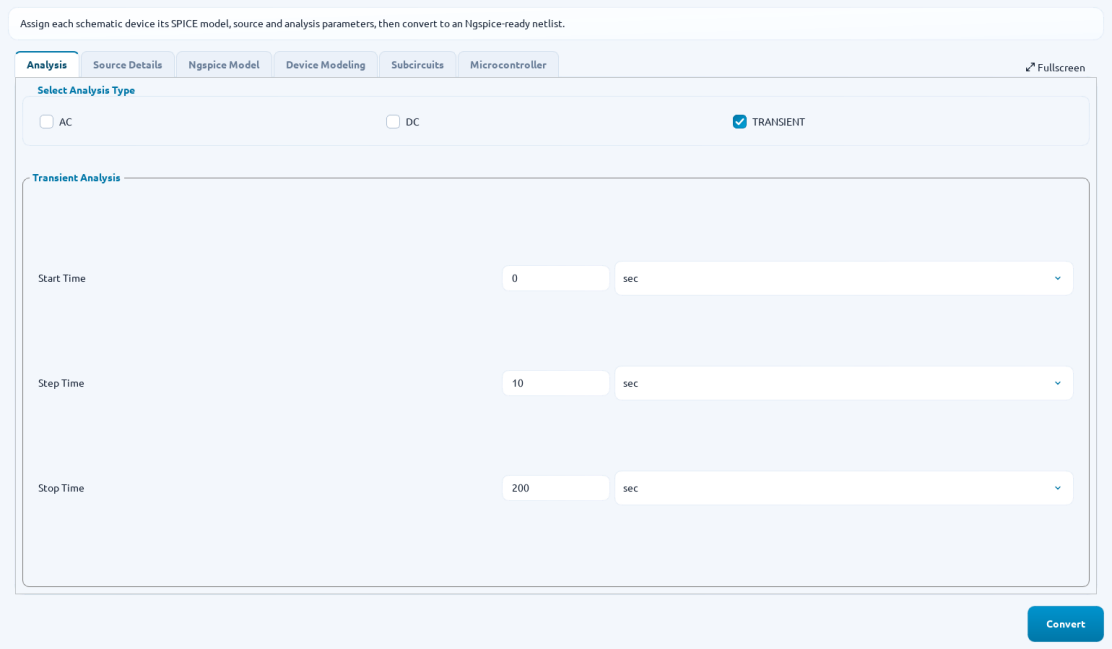


Figure 11.4: The current production converter running against a copied mixed-signal example. The transient analysis is configured in the same workspace that owns the source, model, subcircuit, and microcontroller tabs. This image was captured from the maintained eSim source, not inherited from an earlier report.

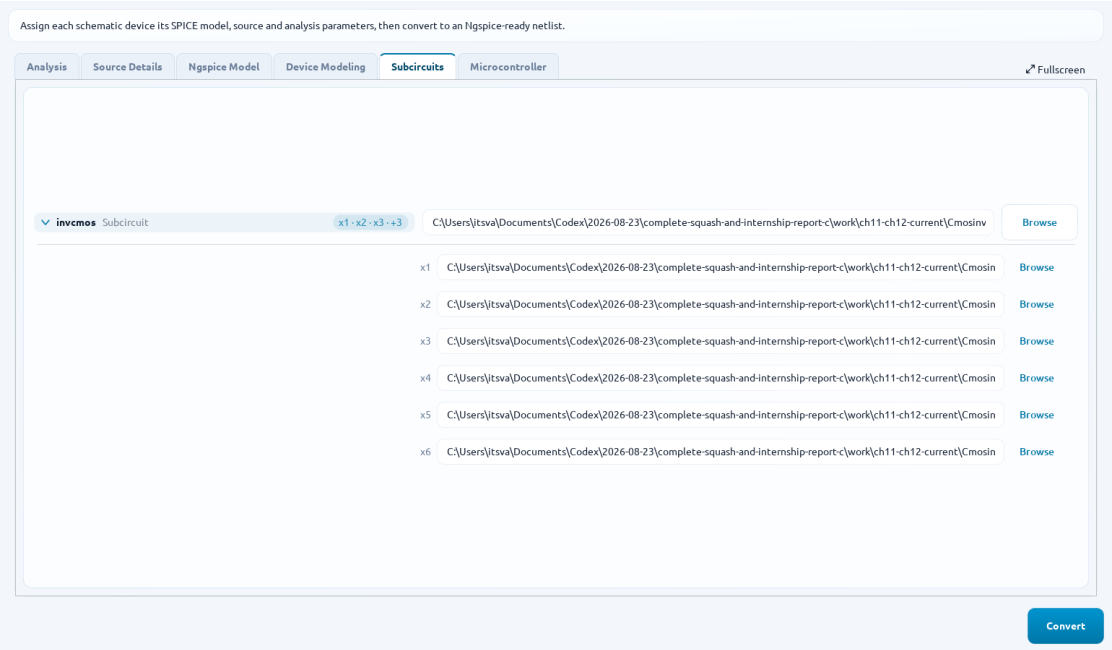


Figure 11.5: A current production capture of six `INVCMOS` placements grouped by model. One directory supplies the shared default while every row remains independently addressable. The compact header records all six references without duplicating six separate assignment panels.

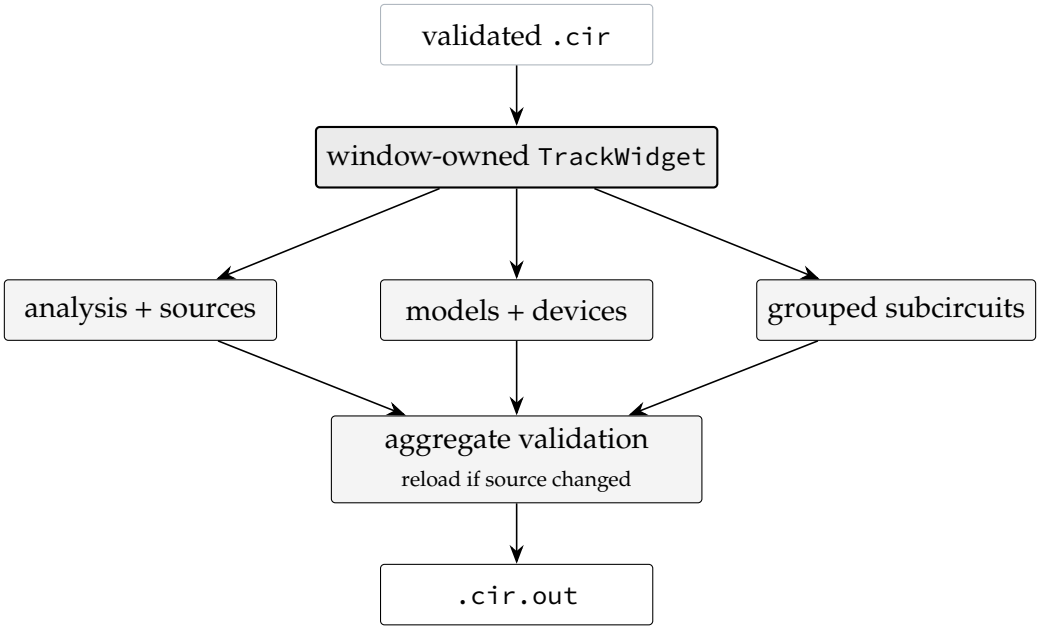


Figure 11.6: State ownership in the current converter. Tabs share one window-local data bus; validation reconverges before the output file is produced.

11.5 How the compatibility claim was tested

The shipped tests under `src/kicadtoNgspice/tests` cover netlist generation, failure paths, converter reloads, model and subcircuit grouping, cached assignments, output writing, and mixed-signal handling. These tests protect individual contracts and the integration between the parser, tabs, and writer.

During development, a broader characterization corpus compared generated circuits against expected circuits as graphs rather than raw text. This ignored harmless line ordering while still detecting changed connectivity. The corpus grew from 52 fixtures through a further 33 subcircuit-focused cases to 101 passing fixtures, including maintained example and student circuits. Those bulky projects were development evidence; the focused regression tests remain in the shipped source tree.

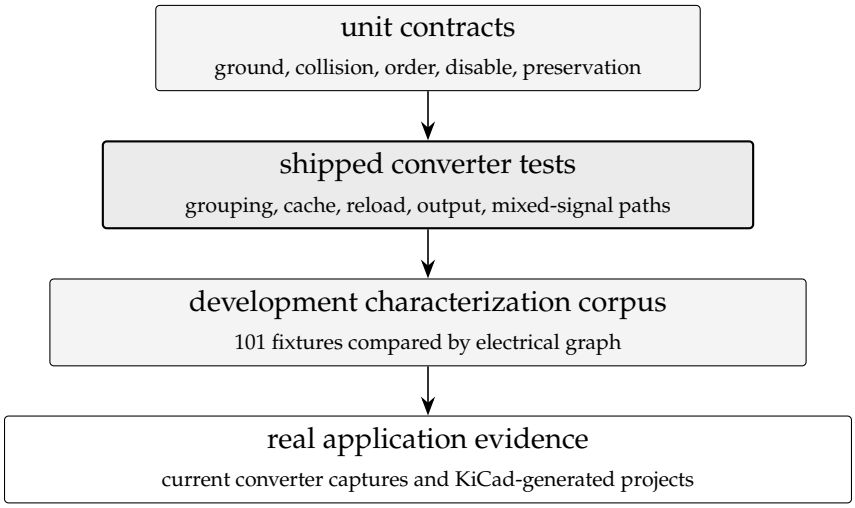


Figure 11.7: The compatibility proof is layered. Small semantic tests localize defects; integration tests protect the maintained workspace; circuit-level comparison checks the electrical result rather than its formatting.

11.6 Practical project behaviour

The compatibility work is deliberately quiet in normal use. A project still opens the converter and ends with `.cir.out`; the additional decisions happen at the boundary where a project can otherwise lose electrical intent.

Table 11.2: Observable behaviour at common project boundaries.

Project condition	Maintained behaviour
Legacy `.sch` and migrated `.kicad_sch` both exist	Prefer the modern schematic that current KiCad actually edits.
No modern schematic is available	Leave an existing hand-made or historical `.cir` untouched.
KiCad CLI export or XML parsing fails	Surface the failure and preserve the last usable circuit input.
The schematic regenerates while the converter is open	Detect the changed modification time and reload before conversion.
Several placements use one subcircuit model	Validate one group default, retain a separate override and output entry for every reference.

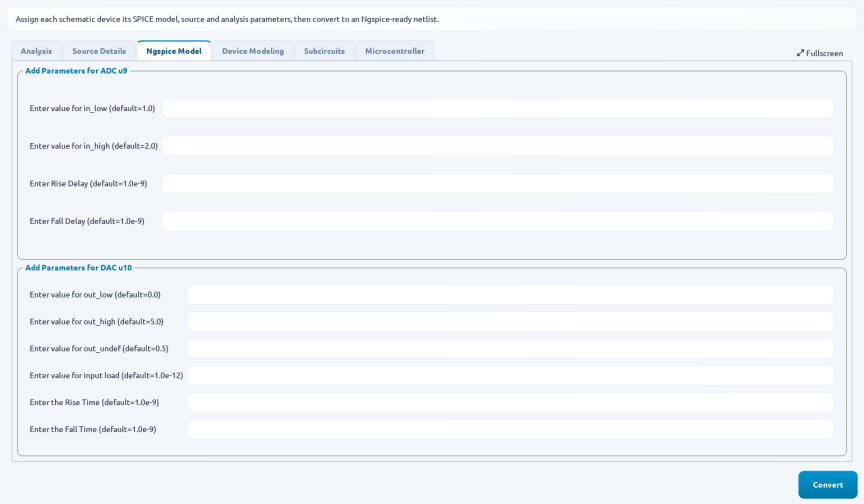


Figure 11.8: The current ngspice-model tab for the same mixed-signal project. ADC and DAC parameters are generated from the registered model descriptions, keeping model-specific input in the shared converter rather than scattering it across separate dialogs.

The result is more than opening eSim beside a newer KiCad. The schematic boundary has explicit ownership, failure does not erase the last usable input, and instance-level intent survives all the way to ngspice.

Part IV

Shipping eSim 2.6

Chapter 12

Ubuntu 26.04 Installation

> **Impact**
>
> Ubuntu support is now one installation system rather than a collection of drifting release scripts. A profile selects distribution-specific packages, one ordered action list drives both installation and dry-run, the bootstrap owns only trees it created, and mandatory build failures cannot fall through to a success banner.

12.1 From release scripts to installation profiles

The earlier Ubuntu path kept a separate shell script for each distribution release. That structure duplicated package lists and failure handling, so only one copy tended to remain current. Ubuntu 26.04 could not be added responsibly by making one more copy. The replacement, `Ubuntu/install-eSim.sh`, reads `/etc/os-release` and selects a profile inside a single installer.

The profile contains only the choices that genuinely vary: KiCad source, expected major version, and release status. Everything else follows the same ordered installation path. Ubuntu 22.04 is rejected with a concrete reason—its standard Verilator and KiCad are below the supported floors—while the end-of-life 23.04 profile remains best effort and is excluded from the supported release metadata.

Table 12.1: Release behavior selected by the current `detect_profile()` implementation.

Ubuntu	KiCad source and floor	Installer decision
22.04	Repository KiCad 6; Verilator 4	Reject: tool versions are below the supported floors.
23.04	KiCad PPA; minimum 7	Best effort, with an end-of-life warning.
24.04	KiCad 9 PPA; minimum 9	Noble profile and its Qt 6 package set.
25.04	Universe; minimum 8	Avoid the unreliable KiCad 9 PPA on Plucky.
26.04	Universe; minimum 9	Distribution packages for the fellowship target.

The command line is explicit: exactly one of `--install`, `--uninstall`, or `--dry-run` must be supplied. This removes the ambiguity of an argument-free script whose default action changes the machine.

12.2 One action list, eleven ordered steps

The installer defines its work once in `INSTALL_STEPS`. The install path executes that array; the dry-run path prints it. A new step therefore cannot quietly appear in one mode and be absent from the other. Figure 12.1 shows the control structure around the list.

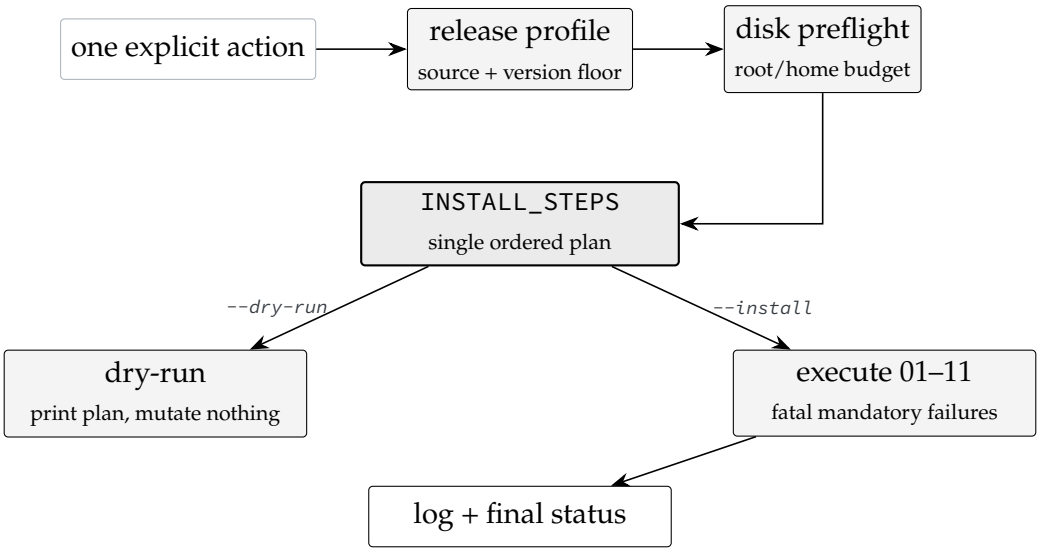


Figure 12.1: Installer control flow. Detection and capacity checks happen before mutation; install and dry-run share the same source of truth.

Capacity is checked before package work. When root and home share a filesystem, the installer budgets roughly 5 GB; when they are separate, it checks about 3.5 GB on root and 1.5 GB on home. A fresh installation fails the preflight when that space is unavailable. An existing installation can continue after a warning because much of the reserved space may already be occupied by reusable artifacts.

The eleven steps cover legacy cleanup, configuration, system packages and the virtual environment, PyQt6 and QScintilla, KiCad, symbol installation, NGHDL and ngspice, SKY130, optional IHP, launchers, and a final toolchain check. Heavy binary dependencies come from the distribution where possible; selected pure-Python dependencies are pinned. Setting `NEEDRESTART_MODE=l` prevents a package upgrade from stopping the terminal for an interactive service-restart prompt.

12.3 The installer interface under a safe terminal demo

Figure 12.2 comes from `Ubuntu/tests/demo.sh`. The harness runs the real installer main block, numbered step loop, logging pipeline, progress display, warning style, and terminal capability detection. Only the package and build functions are replaced with short, harmless stubs. It redirects `HOME` to a temporary directory and uses neither `sudo` nor the network, so the interface can be exercised without

altering the machine.

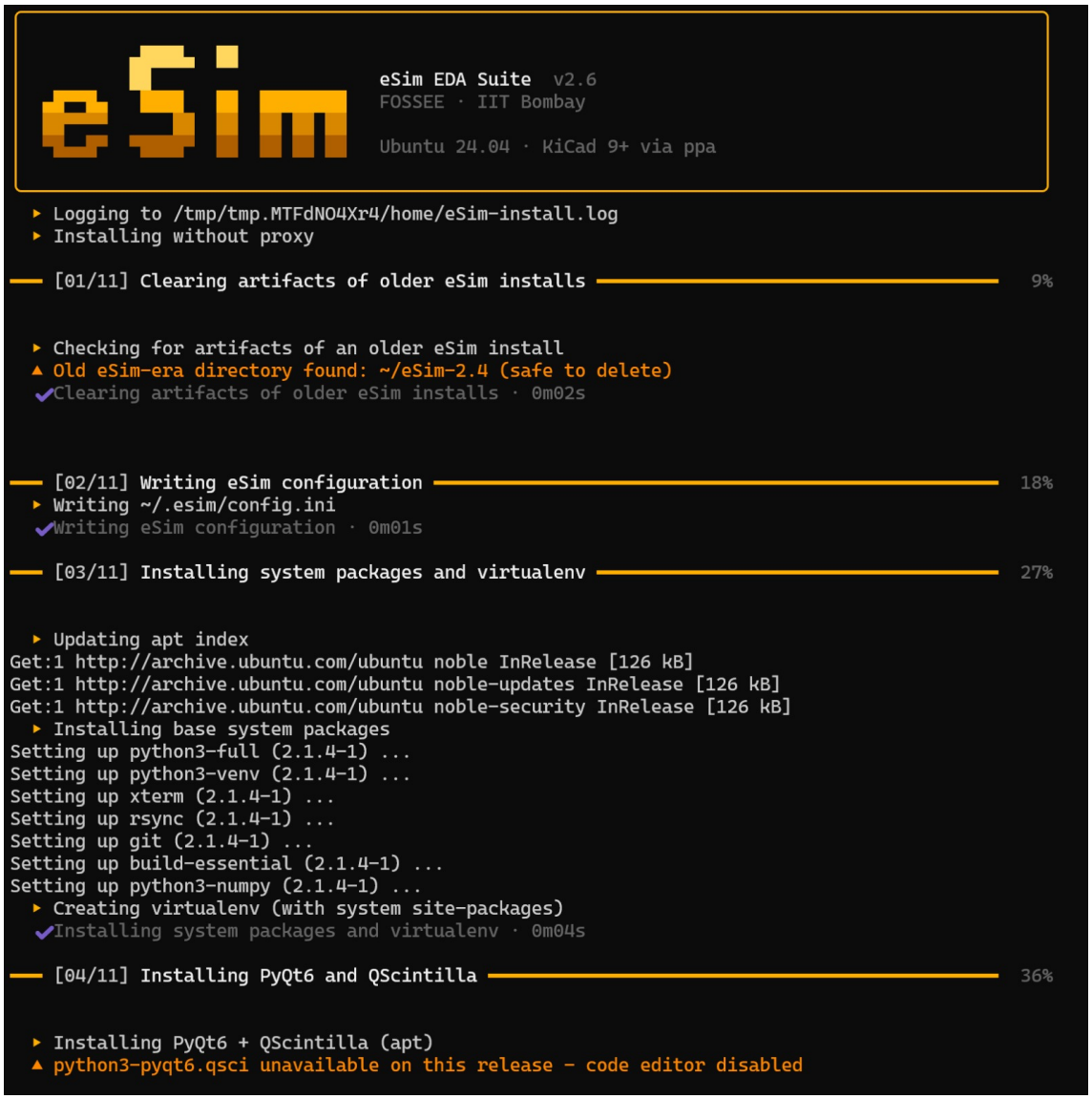


Figure 12.2: The current Ubuntu installer terminal interface running through its repository demo harness. The capture shows the branded header, persistent eleven-step progress, per-step timing, detailed package output, and warnings that remain visible in the log. The harness changes no real installation state.

The repository also retains `--dry-run` for an exact, non-mutating plan. Running it while preparing this report resolved the Ubuntu 24.04 profile and listed the same eleven steps, eSim root, virtual environment, symbol destinations, PDK state, and log path. The demo and dry run serve different checks: one validates terminal behavior; the other validates the resolved installation plan.

12.4 The one-line bootstrap has narrow ownership

After FOSSEE publishes the `v2.6` tag, the stable installation entry is:

```
curl -fsSL https://raw.githubusercontent.com/FOSSEE/eSim/v2.6/Ubuntu/bootstrap.
    sh | ESIM_BRANCH=v2.6 bash
```

The bootstrap is intentionally not a second installer. It validates that the session is a non-root Ubuntu terminal with `curl`, locates or fetches an eSim source tree, and dispatches to that tree's `Ubuntu/install-eSim.sh`. Tags are attempted before branches; development defaults to `FOSSEE/eSim` on `master`. The downloaded source tree is also the installed application; there is no hidden second payload.

Ownership is recorded with `.esim-bootstrap`. A marked tree may be updated safely on a rerun, while generated NGHDL, NgVeri, and co-simulation model descriptions are preserved. A git checkout or a hand-extracted tree has no marker and is reused without being overwritten. Uninstallation still works after the application directory has disappeared: the bootstrap reads the configured eSim home and can fetch a temporary dispatcher solely for the removal operation.

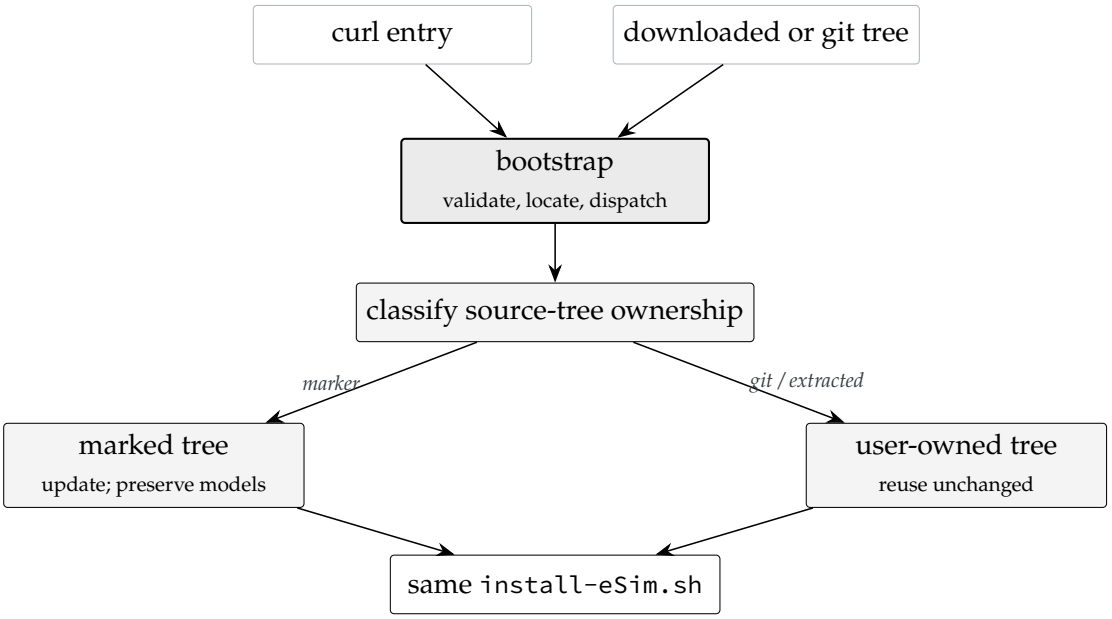


Figure 12.3: Bootstrap ownership and convergence. Only a tree stamped by the bootstrap is updated; both managed and user-owned source trees dispatch to the same installer.

Install, uninstall, and dry-run arguments reach the same implementation. Release CI checks the tag against `VERSION`, names the release `eSim-2.6`, and lists Ubuntu 24.04, 25.04, and 26.04 as supported targets.

12.5 Mandatory, optional, and advisory outcomes

An installer should distinguish a failed product from a missing enhancement. NGHDL is mandatory because it supplies the ngspice build used by eSim; a missing source archive or a failed build stops installation. SKY130 and IHP are useful PDK additions, but their absence does not make ordinary schematic simulation

impossible. The final doctor run is advisory: it reports affected flows and their fixes without retroactively treating every optional capability as a failed base install.

Table 12.2: Failure semantics in the current Ubuntu installer.

Stage	Outcome class	Behaviour
Base installation	Mandatory	Profile, capacity, system packages, KiCad, and NGHDL/ngspice must succeed; otherwise the success banner is unreachable.
Python extras and PDKs	Optional	Warn when an extra or selected PDK is unavailable while preserving the usable base application.
Toolchain doctor	Advisory	Print actionable, flow-specific findings without masking successful mandatory steps.

All terminal output is mirrored to `~/eSim-install.log`; ANSI styling is stripped only from the file, leaving the interactive display readable and the saved log portable. The uninstaller removes eSim-owned launchers, package registrations, generated symbols, PDKs, and NGHDL artifacts, but does not search outside `~/.esim` for user projects or hand-edited files.

The toolchain doctor itself is covered in Chapter 10. Reusing that single diagnostic surface is important: the installer, command line, Help menu, and runtime gates all report the same underlying capability checks instead of maintaining four lists.

12.6 Clean-room equivalence audit

The architecture says the curl and downloaded-source paths converge. A clean-room audit checked the resulting systems rather than trusting that diagram. Two pristine Ubuntu 24.04 containers were prepared with the same minimal desktop baseline. One used the one-line entry; the other used the downloaded release source. A twelve-section manifest then compared source identity, installed files, resolved tools, and lifecycle behaviour.

The installations shared 11,300 files. None differed after generated Python bytecode was excluded. Their staged source trees matched, and they produced the same doctor report. Archive and release identity were checked separately, preventing two equivalent installs from being accepted when both started from the wrong payload.

The audit was useful because it also found failure-path defects: capacity was checked too late, an NGHDL build failure could be treated as non-fatal, and the script could reach a success message without a working simulator. The current preflight, fatal NGHDL step, and ordered success path are the direct corrections.

Table 12.3: What the clean-room comparison established.

Audit dimension	Evidence compared	Claim protected
Entry path	Curl bootstrap versus downloaded source	Public installation routes converge before installation logic.
Source identity	Release input, staged source, and manifest identity	Both machines install the intended application source.
Filesystem	11,300 shared files, excluding bytecode	The entry route does not change the installed payload.
Capability	Toolchain doctor output	Matching files resolve to matching simulator capabilities.
Failure honesty	Low-space and missing-NGHDL cases	A non-simulatable mandatory state cannot report success.
Lifecycle	Dry-run, reinstall, and uninstall behaviour	Repeated operation remains predictable and ownership stays bounded.

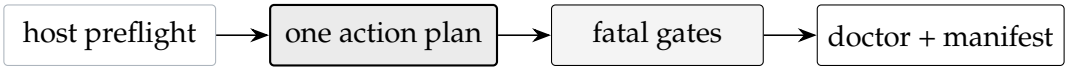


Figure 12.4: The installation assurance chain: choose the correct host profile, expose one plan, stop on required failures, and verify the resulting capabilities and payload.

The comparison therefore closes at behaviour, not at file count. Both entry paths resolve the same plan, stop at the same mandatory gates, expose the same simulator capabilities, and leave the same ownership boundaries after reinstall or removal. That is the condition the release process now checks before an Ubuntu build is treated as reproducible.

Chapter 13

Reproducible Windows Packaging

Windows support already existed, but its build process did not. The earlier installer was an off-repository binary assembled through manual steps. For eSim 2.6, the Windows product became a repository-owned system: downloads are pinned, the simulator is built from source, KiCad is bundled without its unused payload, the install tree is cleaned before release, and the resulting executable is accompanied by a checksum.

13.1 The packaging problem was reproducibility

The eSim 2.5 Windows installer was useful to users, but difficult to maintain. Its roughly 478 MB NSIS executable could not be reconstructed from the source tree. When a dependency changed, there was no manifest to review. When a runtime file was missing, the failure appeared on a user machine rather than at the packaging boundary. The work for 2.6 therefore began with a stricter question: can another maintainer take the tagged source, run one documented command, and obtain the same product structure with every required tool proven before publication?

Table 13.1 summarizes the answer. The figures in the final row are taken from the completed local artifacts, not an estimate: the current installer is 379.0 MiB and its sha256 file is emitted beside it.

13.2 One command defines the product

The build entry point is `windows/build-windows.ps1`. It resolves its own directories, reads `VERSION`, creates a filtered application stage, provisions a private Python runtime, expands and prepares the SKY130 PDK, builds the mixed-signal toolchain in MSYS2, prunes KiCad, compiles the native launcher, removes development state, and invokes Inno Setup. The order is significant. A later stage is allowed to consume only an output that the preceding stage has already checked.

Figure 13.1 shows the actual control structure at report scale. It is not a list of downloaded programs. The central object is the staged product tree. Every source and binary input enters that tree through a named preparation function, and the tree passes through normalization and independent assertions before the installer can read it.

Table 13.1: The Windows deliverable before and after the repository-owned build.

Property	Earlier delivery	eSim 2.6 delivery
Build definition	Manual process outside the repository	`windows/build-windows.ps1` and `windows/installer.iss`
Dependency custody	Versions embedded in the builder's working environment	Seven third-party inputs recorded by URL, version, filename, and sha256
Simulation runtime	Opaque prebuilt combination	ngspice 45.2 and a libvvp-enabled Icarus build assembled from pinned sources
KiCad	External or manually repackaged dependency	KiCad 9.0.3 payload extracted, pruned, and smoke-tested inside the build
User configuration	Coupled to installation-time assumptions	Idempotent bootstrap executed for the current user at every launch
Published artifact	Approximately 478 MB NSIS package	379.0 MiB Inno Setup executable plus sha256

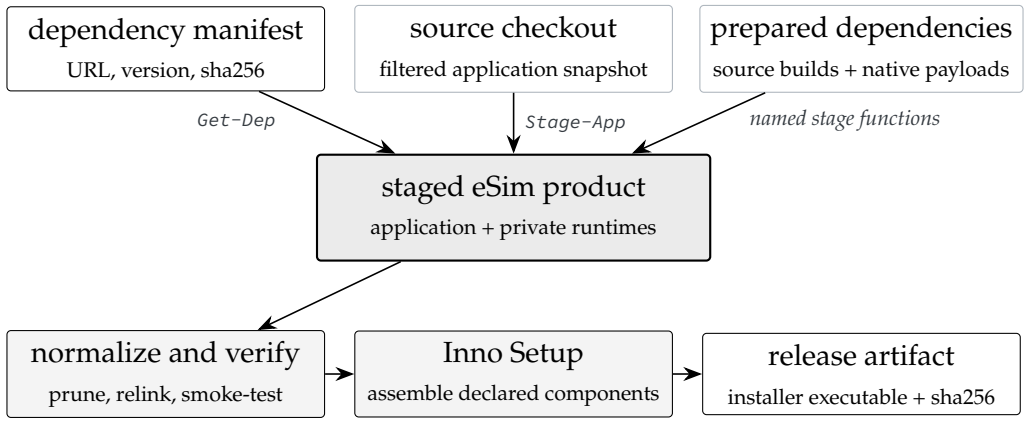


Figure 13.1: The repository-owned Windows build. The installer compiler never reads the developer checkout directly; it consumes a prepared and independently checked product tree.

13.2.1 Pinned inputs and deliberate exceptions

`deps-manifest.json` contains seven downloadable inputs. `Get-Dep` uses a cached file only after recomputing its sha256. A blank hash is rejected by default. The maintainer-only `-AcceptNewHashes` override records the observed value and instructs the maintainer to compare it with the publisher's checksum. A mismatch stops the build and prints both values.

The manifest also records why each source was chosen. The Icarus Windows

binary, for example, is only a compact-build fallback because it lacks `libvvp`; the Full installer builds the pinned Icarus source with `-enable-libvvp`. Inno Setup is a build-machine tool and does not enter the product. This distinction matters because a manifest should describe dependency roles, not merely collect URLs.

Table 13.2: Pinned inputs and the boundary each one owns.

Input	Pinned version	Role in the Windows product
CPython	3.12.10	Private interpreter into which the locked PyQt6 wheel set is installed
ngspice	45.2	Compact fallback and version reference for the source-built simulator
Icarus source	pinned post-v12 revision	Full build with `libvvp` for `d_cosim`
Icarus binary	dated v12 build	Verifier-capable fallback when simulator builds are skipped
MSYS2	20250622	Isolated gcc, make, Verilator, GHDL, and source-build environment
KiCad installer	9.0.3	Payload source; the installer itself is extracted and never run
Inno Setup	6.3.3	Compiler for the final setup executable

13.2.2 The stage is a product, not a copied checkout

`Stage-App` mirrors the repository incrementally with `robocopy`, but explicit filters define what is eligible to ship. Junction traversal is disabled so a developer's convenience link to an installed toolchain cannot cause the build to package itself. Generated traces, caches, platform-only folders, build downloads, and simulation outputs are excluded. Runtime files from `windows/` are then copied back one by one because the packaging directory itself is not part of the application.

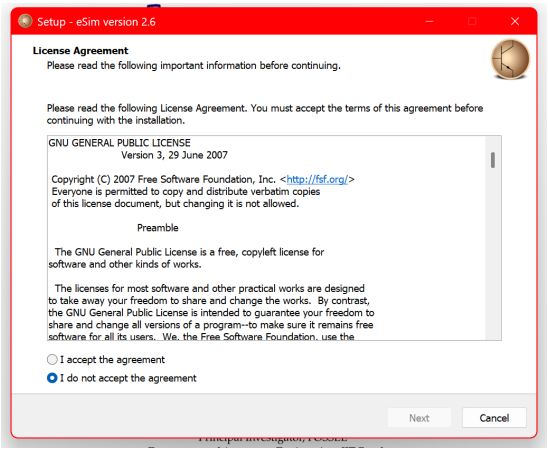
The distinction between a Git checkout and a linked worktree produced one final leak. A normal checkout has a `.git` directory, while a worktree has a `.git` file that contains an absolute pointer to the parent repository. The copy filter now rejects both representations. `Assert-CleanStage` separately checks for any surviving `.git` path, so a future change to the copy command cannot silently reintroduce the disclosure.

Cleaning goes further than VCS metadata. User-created NgVeri, NGHDL, and `d_cosim` models are removed from the parameter registries and source trees. The corresponding code-model libraries are relinked even when the source folders already look empty, because a deleted model can remain compiled inside a `.cm` binary. The assertion then scans those binaries for removed model names and checks that the generated KiCad symbol seeds contain no developer symbols.

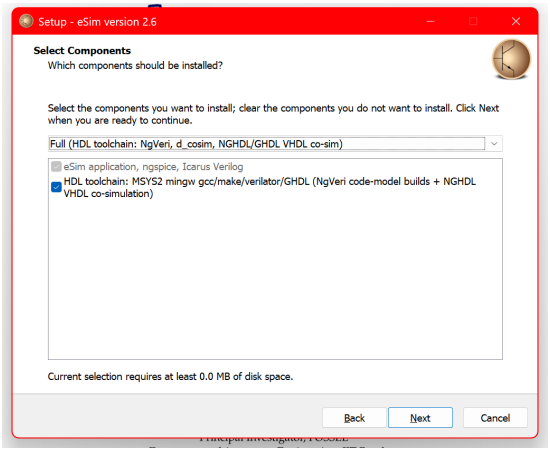
13.3 A real installer, not a zip with a shortcut

Inno Setup assembles the checked stage into a single executable. The product has one icon from download to uninstall, a GPLv3 license page, a space-free default root at `C:\FOSSEE\eSim`, and explicit Full, Compact, and Custom installation types. Core simulation always includes eSim, ngspice, and Icarus. The Full choice adds MSYS2, gcc, make, Verilator, and GHDL for runtime model generation.

Figure 13.2 is a fresh render of the current `installer.iss` definition with the repository's production branding and component text. The capture harness carries no payload and performs no installation; it exists only to render the same wizard pages safely for this report.



(a) Product identity and GPLv3 acceptance.



(b) Full installation with the HDL toolchain selected.

Figure 13.2: The eSim 2.6 Windows installer as defined by the current Inno Setup source. The component page exposes the actual runtime boundary instead of hiding it in an undocumented bundle.

13.3.1 Bundled KiCad without taking over the machine

The build extracts the official KiCad installer as data. It removes the 784 MB 3D model tree, demos, translations, and plugin extras that eSim does not invoke, then stores the remaining payload under `tools/kicad`. A user's separate KiCad installation is not registered, modified, or placed behind the bundled copy globally. Only the eSim launcher prepends the private `tools/kicad/bin` directory to its process environment.

Pruning is followed by behavior. The staged `kicad-cli` must report the pinned version, the number of stock symbol files must match the template library table, and a real example schematic must export a non-empty netlist. These checks catch a structurally valid but unusable KiCad tree before it enters the installer.

13.4 Machine payload and user state

The elevated installer owns the application payload and shortcuts. It does not guess which accounts will use the machine. The native `eSim.exe` launcher therefore calls `windows/windows_bootstrap.py` for the current user on every start. Each operation is idempotent and independent, so one failed repair is reported without skipping the rest.

The bootstrap writes the user-side `.esim/config.ini`, seeds the three generated symbol libraries without overwriting user models, records the NGHDL and Icarus paths in `.nghdl/config.ini`, relocates ngspice code-model paths in `spinit`, supplies the unversioned `libvvp.dll` name expected by `ivlng`, and registers eSim-owned KiCad rows. Figure 13.3 makes the ownership split explicit.

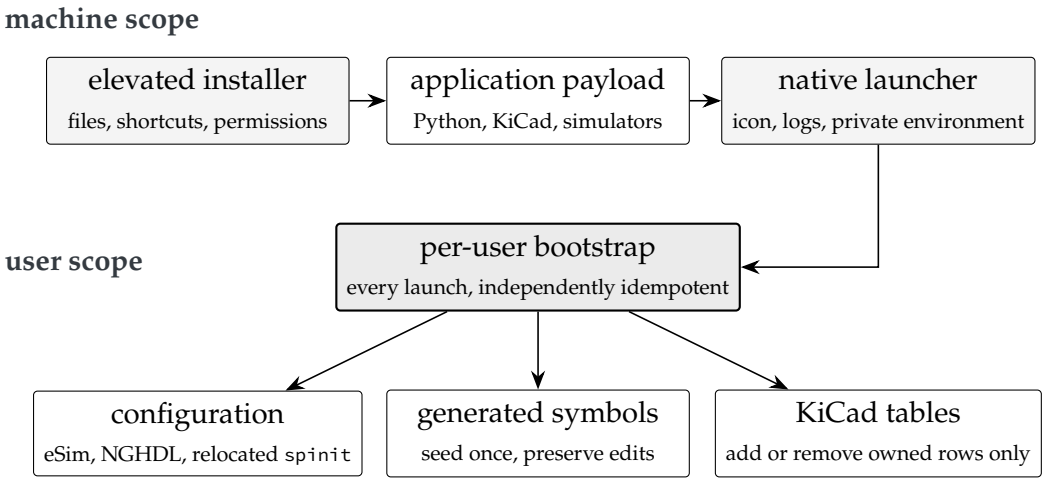


Figure 13.3: Windows ownership by lifetime. Setup owns the machine payload; launch-time bootstrap owns only the current user's paths, generated libraries, and registrations.

The installer follows the same rule for permissions. Runtime write access is scoped to `tools/nghdl` and `library/modelParamXML`, the two trees where model generation actually writes. Python, KiCad, MSYS2, source modules, and the native launcher remain read-only application assets. Uninstall removes runtime-built files that Inno could not have recorded at install time, while the separate user-data prompt controls whether the current account's configuration is retained.

13.5 Windows faults became permanent gates

Building the complete stack natively exposed failures that a Linux-only package could not show. The useful outcome was not the number of defects; it was converting each one into a check at the boundary where it can recur. Table 13.3 records the main classes and their containment.

Table 13.3: Windows-specific failure classes and the checks that now contain them.

Boundary	Failure observed during porting	Permanent containment
Compiler and ABI	Strict-aliasing optimization corrupted simulator behavior	Source flags plus a batch analog simulation
Socket runtime	The NGHDL client lacked the Winsock link contract	Explicit native libraries and a link-capable GHDL probe
Dynamic loading	VPI modules existed while their dependent DLLs or `libvvp` did not resolve	Recursive DLL closure and bare-environment compile/run tests
Process model	Console ngspice output and interactive Windows plotting required different binaries	Console runtime for eSim plus a co-located `ngspice_gui.exe` twin
Relocation	Build-machine paths survived in `spinit` and configuration	Launch-time relocation and doctor path reporting
Payload hygiene	Local models, audit files, or worktree metadata entered the stage	Normalization followed by an independent rejecting assertion

13.6 Result

The completed build produced `eSim-2.6-installer.exe` at 379.0 MiB and a matching sha256 sidecar. The packaging suite covers hash policy, bootstrap idempotence, uninstall ownership, and the Windows-specific script contracts. More importantly, the build itself checks executable versions, code-model presence, a plain ngspice simulation, a libvvp-backed Icarus compile and run, a KiCad netlist export, the SKY130 electrical test, and the final stage contents before ISCC is allowed to publish.

> **Impact**
>
> The Windows release is now rebuildable from repository source. Its dependencies have reviewable identities, its installer exposes the real component boundary, its per-user state repairs itself at launch, and its publication step consumes a verified product tree rather than a developer checkout.

Chapter 14

PDK and Toolchain Closure

A release can contain every expected file and still fail at the first useful simulation. eSim 2.6 treats packaging as a behavioral contract. The SKY130 deck must survive relocation and drive a CMOS inverter, IHP OSDI entries must refer only to compiled models, and every simulation flow must be able to explain the exact capability it is missing.

14.1 Closure begins after file existence

The mixed-signal stack crosses several boundaries: compressed model archives, nested SPICE includes, dynamically loaded code models, compiler backends, runtime libraries, per-user configuration, and KiCad library tables. Checking that a directory exists proves almost nothing about those relationships. A copied PDK can contain a malformed include. An executable can print its version while the DLL it loads at run time is absent. A library row can be registered while pointing to another installation.

The closure work follows one rule: test the behavior at the boundary where the user needs it. PDK validation reaches an electrical measurement. Toolchain checks follow resolved paths into companion libraries. Registration records ownership so install, upgrade, and uninstall do not damage unrelated tools.

14.2 SKY130: from archive to electrical evidence

The repository stores SKY130 as `library/sky130_fd_pr.tar.xz`. Windows must ship the expanded directory because eSim opens `library/sky130_fd_pr/models` directly. `Stage-Sky130` therefore rebuilds that directory from the archive on each staging run, invokes `src/configuration/Sky130Prepare.py`, and removes both archive layers only after preparation succeeds.

The preparation module handles one known defect in the pinned 2022 snapshot. A model contains Spectre-like `include` text where ngspice requires a relative `.include` directive and the correct `.spice` target. The repair is purposely narrow. It accepts exactly one broken occurrence or exactly one corrected occurrence. Missing entry points, duplicate matches, or an unfamiliar third state are rejected. This prevents a future PDK revision from being rewritten through an assumption that was valid only for the vendored snapshot.

Figure 14.1 shows the full state machine. The final gate uses the ngspice executable from the staged product, loads the complete `tt` corner, and simulates a

1.8 V inverter. At 2 ns the output magnitude must be below 0.1 V; at 4 ns it must exceed 1.7 V. Both measurements are parsed from ngspice output, so a successful process with a missing or nonsensical result still fails the build.

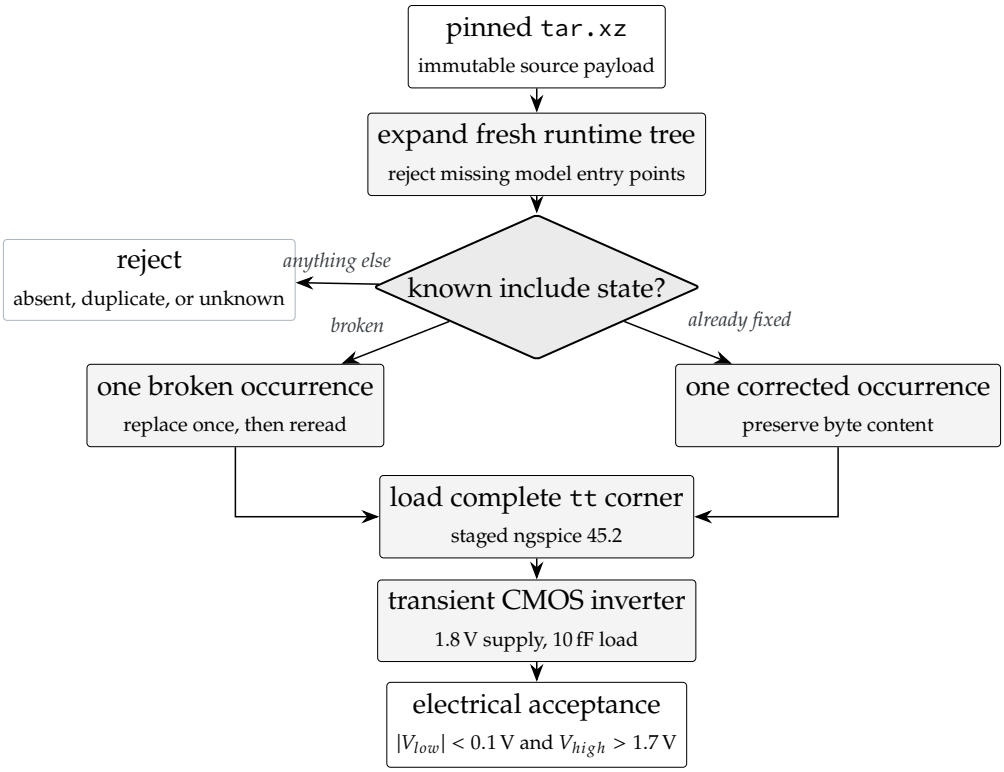


Figure 14.1: The SKY130 release gate follows an exact repair state into a full corner load and an electrical measurement. Unknown deck states stop before packaging.

This same preparation module is portable Python and is used by the supported installers. Its focused tests cover repair, idempotent re-entry, missing files, and unexpected deck content. The release build then adds the simulator-level proof that a unit test cannot provide.

14.3 IHP: enabling only the models that exist

IHP OpenPDK follows a different delivery model. It is an optional Ubuntu installation, not part of the Windows payload. The installer obtains the SG13G2 tree, installs OpenVAF, compiles the Verilog-A sources into OSDI libraries, and refuses to continue when no `.osdi` output is produced. It then adds one marked IHP block to the NGHDL `spinit` file.

Enabling OSDI exposed a less obvious defect. ngspice's stock `spinit` contains directives for reference models that ngspice does not ship as compiled libraries. They remain harmless while `osdi_enabled` is unset. Turning the switch on for IHP also activated those unrelated entries, so every simulation began with eight missing-library errors even though the IHP models themselves loaded correctly.

The correction is based on the filesystem rather than a hardcoded model list.

`sync_osdi_lines` examines every active OSDI directive. If the target does not exist, it comments the line with an eSim-owned tag. If the file appears on a later run, the tagged line is restored. Uninstallation removes only the marked IHP section and returns to `unset osdi_enabled` when no active OSDI entries remain. The process is therefore repeatable in both directions.

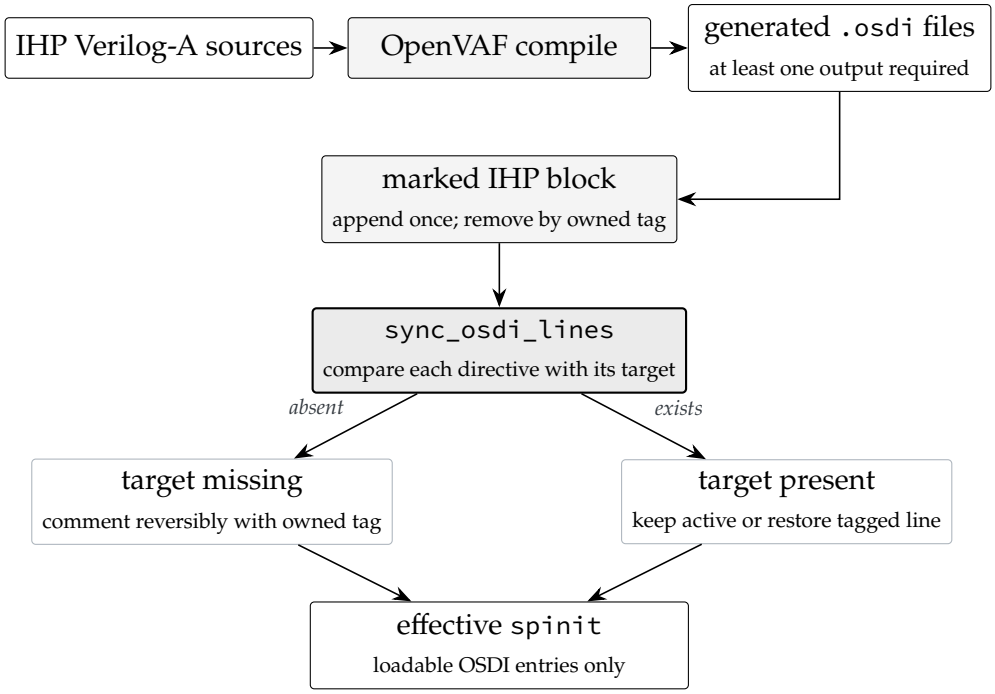


Figure 14.2: IHP OSDI configuration is derived from compiled outputs. Missing stock directives are disabled reversibly, while eSim's own IHP block remains identifiable for upgrade and uninstall.

14.4 One capability model for every simulation flow

A generic "tool not found" message is insufficient for this stack. `d_cosim` needs ngspice, its code-model directory, `ivlng`, Icarus, `vvp`, and `libvvp`. NGHDL needs a link-capable GHDL backend, `ghdl.cm`, gcc, make, and the configured release tree. NgVeri uses Verilator instead of the Icarus runtime. A single global pass or fail would either block unrelated work or let an incomplete flow start.

`src/maker/ToolchainCheck.py` models every probe as a `Check` record with a stable key, label, result, resolved path, detail, repair hint, and the flows that depend on it. `run_checks(flow)` filters that graph for one workflow. The probing core is Qt-free, reads the filesystem at call time, and suppresses console flashes on Windows. Only the dialog imports Qt, and it does so lazily.

Figure 14.3 shows how one probe graph serves four contexts. The headless command prints a report and returns a failing exit status when necessary. The Help menu wraps the same report in a copyable dialog. Flow preflight calls `failure_message(flow)` before starting a compiler. The Ubuntu installer runs

the headless form as its post-install self-check, while the Windows log collector records it for support.

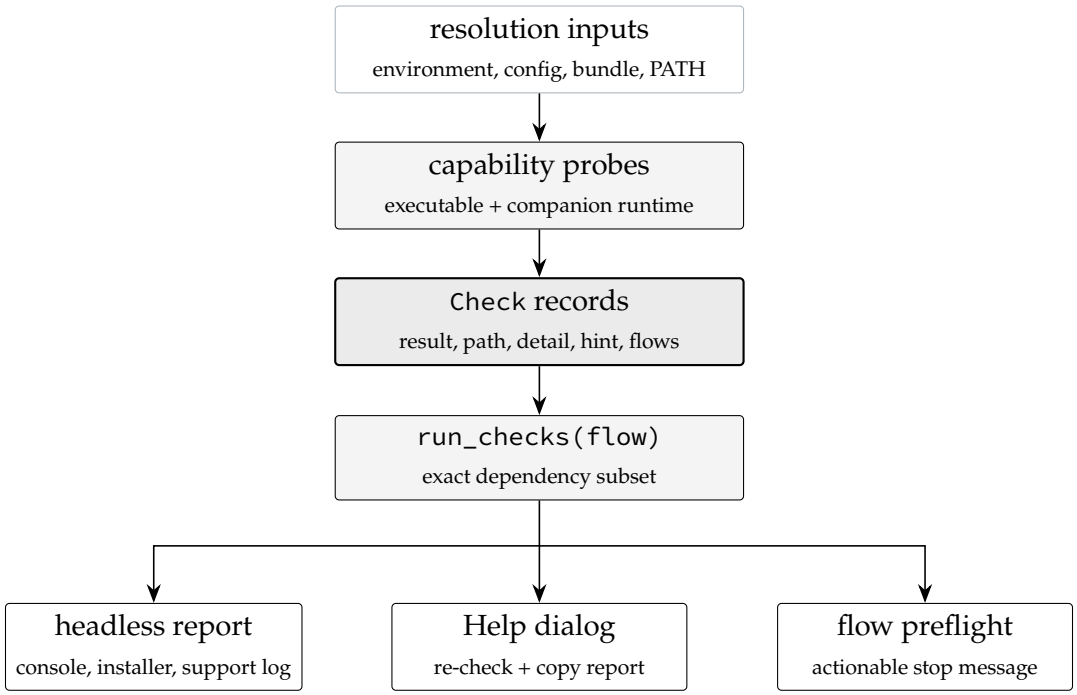


Figure 14.3: Toolchain diagnosis has one source of truth. User surfaces format the same records; they do not reimplement path resolution or the definition of a runnable flow.

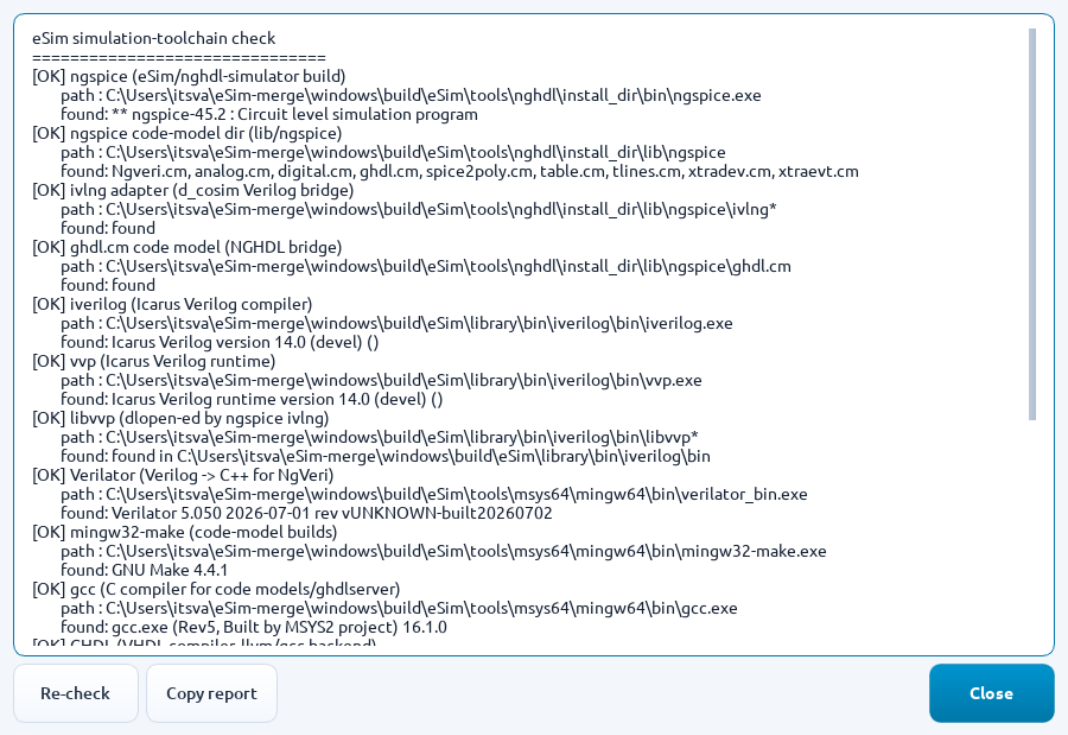


Figure 14.4: Fresh capture of the current eSim 2.6 toolchain dialog against the staged Full Windows product. The report names the exact ngspice, code-model, Icarus, Verilator, make, and compiler paths that were probed.

The capture demonstrates an important reporting choice. A green row still carries its resolved path and version or contents. This lets a user distinguish the bundled tool from an unrelated program found earlier on `PATH`. A failed row uses the same path field to show where eSim looked and adds one repair action.

14.4.1 Dependency subsets prevent false blocking

Table 14.1 is derived from the flow assignments in `ToolchainCheck.py`. A plain analog simulation is not blocked because Verilator is absent, and the Verilog Verifier does not require GHDL. Conversely, finding `iverilog` is not enough to approve `d_cosim`; the shared library and ngspice adapter are separate capabilities.

The GHDL probe deserves special mention. Printing a version is not enough. The `mcode` backend is rejected because NGHDL links its socket server during elaboration, an operation that mcode cannot perform. The report asks for `ghdl-llvm` or `ghdl-gcc` and explains why, turning a late silent failure into a preflight decision.

14.5 KiCad registration with explicit ownership

Windows bootstrap preserves existing stock KiCad rows and inserts missing ones as disabled. Static eSim libraries, including 15 restored legacy converter libraries, use owned absolute paths: the installed application on Windows and `~/.esim/kicad_static` on Ubuntu. The three generated libraries remain in

Table 14.1: Representative capability groups used by each simulation flow. • means the flow is gated on that capability.

Capability	Analog	Verifier	`d_cosim`	NgVeri	NGHDL
ngspice executable and code-model directory	•		•	•	•
Icarus compiler		•	•	•	
`vvp` runtime		•	•		
`libvvp` and `ivlng`			•		
Verilator				•	
gcc and make				•	•
GHDL with a link-capable backend and `ghdl.cm`					•
NGHDL configuration and build tree				•	•
Windows MSYS2 shell or terminal where required	•			•	•

`~/.esim/kicad_symbols`. Packaged legacy files match their upstream originals; template entries for unavailable LTspice libraries were removed.

Table 14.2: Ownership rules for KiCad symbol registration.

Observed state	Bootstrap action	Reason
Stock library already registered	Leave the row byte-for-byte unchanged	The user or KiCad owns that choice
Stock file present, template row missing from user table	Insert an absolute row in the disabled state	Make the bundled library discoverable without enabling it
eSim static symbol library	Register its eSim-owned absolute path as active	It is part of the application workflow
eSim generated symbol library	Seed once in the user directory and register that copy	Model builders may extend it; upgrades must not erase the additions
Row points inside an eSim-owned private KiCad tree during uninstall	Remove that owned row only	A separate system KiCad remains untouched

The Windows build verifies a one-to-one match between the retained stock symbol files and the private KiCad template rows. This closes the other half of the contract: the bootstrap cannot register a library that pruning accidentally

removed. Chapter 13 describes the same ownership rule for worktree metadata and user-generated models at the larger installer boundary.

Figure 14.5 shows why registration is implemented as a merge rather than a replacement. The bootstrap reads both the user's current table and the inventory shipped with eSim. It then chooses the least invasive action for each nickname and source class.

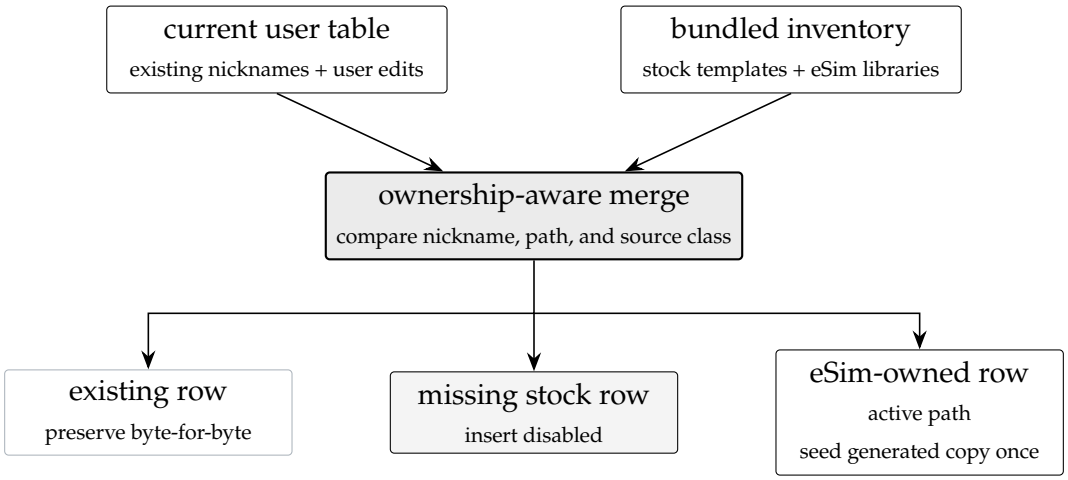


Figure 14.5: KiCad registration is an ownership-aware merge. Existing choices survive, missing stock libraries remain disabled, and only eSim-owned paths are activated or removed later.

14.6 Release closure as a test ladder

The resulting checks form a ladder from static structure to useful behavior. Each higher step catches a class of failure the lower step cannot see.

Table 14.3: Release checks ordered by the strength of the evidence they provide.

Layer	Evidence	Failure caught
Identity	Pinned URL, revision, and sha256	Silent replacement or drift of a downloaded input
Structure	Required file, directory, and companion library checks	Incomplete extraction or pruning
Loadability	ngspice loads code models; `vvp` resolves `libvvp`; GHDL backend can link	Dynamic-library and backend traps
Integration	KiCad exports a real netlist; preflight resolves the exact flow subset	Tools that exist but cannot cooperate
Behavior	SKY130 inverter crosses low and high voltage thresholds	A loadable yet electrically broken PDK
Hygiene	Normalized stage and independent rejection of VCS or local model state	Developer-machine data entering the release

Impact

Release closure now reaches the user's first useful result. SKY130 is repaired only in a known state and then simulated, IHP enables only OSDI files that exist, the toolchain doctor explains capabilities per flow, and KiCad registration changes only rows that eSim owns.

Chapter 15

Performance Engineering

Performance in a desktop engineering tool is a chain of small contracts. The first window must appear without waiting for every optional tool. A long compiler or netlister must not own the event loop. Plotting must limit screen work without discarding the samples used for measurement. This chapter follows those contracts through startup, background execution, rendering, and distribution.

15.1 A performance contract for the desktop

The work did not rely on one global "make eSim faster" patch. It moved cost to the layer that could own it safely. Imports that are unnecessary for the first frame are deferred. Programs that may block run on workers. Repeated screen updates are coalesced. Large traces retain their raw arrays while the screen receives a bounded envelope. Each change therefore has both a speed argument and a correctness boundary.

Table 15.1: The performance contract used across eSim 2.6. Each cost has one owner and one condition that the optimization may not violate.

Cost	Responsible mechanism	Preserved property
Cold imports	Lazy tool imports and delayed prewarm	The main window reaches the event loop before optional modules are loaded
External programs	`BackgroundJob`, queued signals, cancellation, and process-tree cleanup	Qt widgets remain owned by the GUI thread
Repeated UI events	Single-shot debounce timers and incremental artist reuse	The final state after a burst is rendered exactly once
Long waveforms	View-local min/max envelope with raw arrays retained	Cursor values, statistics, export, and zoom continue to use complete simulation data
Visual effects	One hover effect and fixed theme snapshots	Geometry, focus, hit targets, and reduced-motion behavior remain unchanged

15.2 Startup: first frame before optional tools

The earlier startup path imported plotting, Maker, converters, editors, and their native dependencies before the application became interactive. This was especially visible on a cold Windows launch, where module loads could trigger antivirus scanning. The splash screen appeared, but it could not repaint while the import chain occupied the main thread.

The new boundary is implemented in `src/frontEnd/DockArea.py`: every substantial tool is imported inside the method that opens its dock. After the application has entered its event loop, `src/frontEnd/Application.py` schedules a background prewarm three seconds later. The worker imports modules only; it never constructs a `QWidget`. If a user opens a tool while prewarm is running, Python's import lock serializes the two requests safely. An import failure is contained, so optional warmup can never prevent eSim from starting.

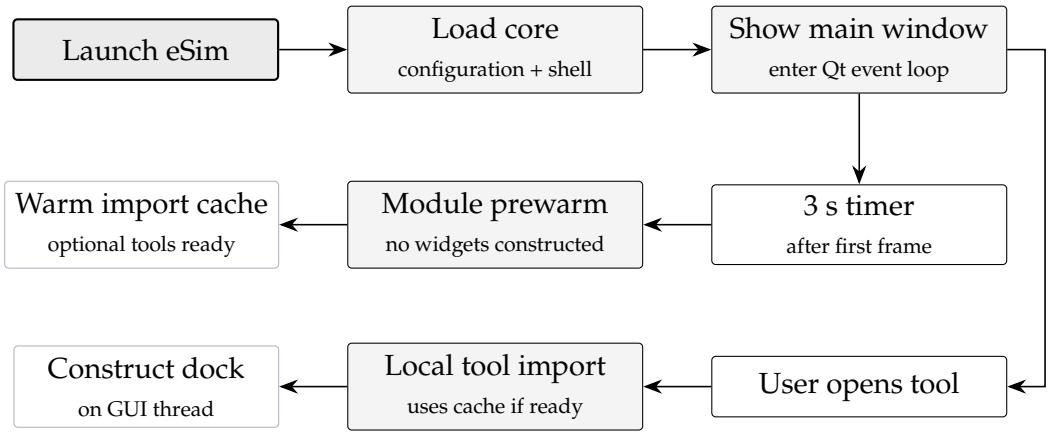


Figure 15.1: Startup critical path and deferred prewarm. The first interactive frame depends only on the application shell; optional tools move to the post-start path.

15.3 Responsiveness: work outside the paint loop

Several operations can take seconds even when they are behaving correctly. A cold `kicad-cli` export, an Icarus compile, a simulation, a model build, or a converter subprocess must therefore be treated as a job rather than a function call from a button handler. `src/maker/hdl/jobs.py` provides that common boundary. A `BackgroundJob` executes the backend call on a `QThread`; progress, success, and failure return through queued Qt signals, so the receiving slot runs on the GUI thread.

Cancellation is an ownership problem as well as a button. The backend binds its live process to a `CancelToken`. A cancellation request terminates the complete descendant tree, not only the driver process. This matters because tools such as `iverilog` and `mingw32-make` create children that can keep output files open after their parent is killed. The shared process helper snapshots descendants before

terminating them, then falls back to a direct kill if process inspection is unavailable. The retry therefore begins from a clean workspace instead of colliding with a hidden process.

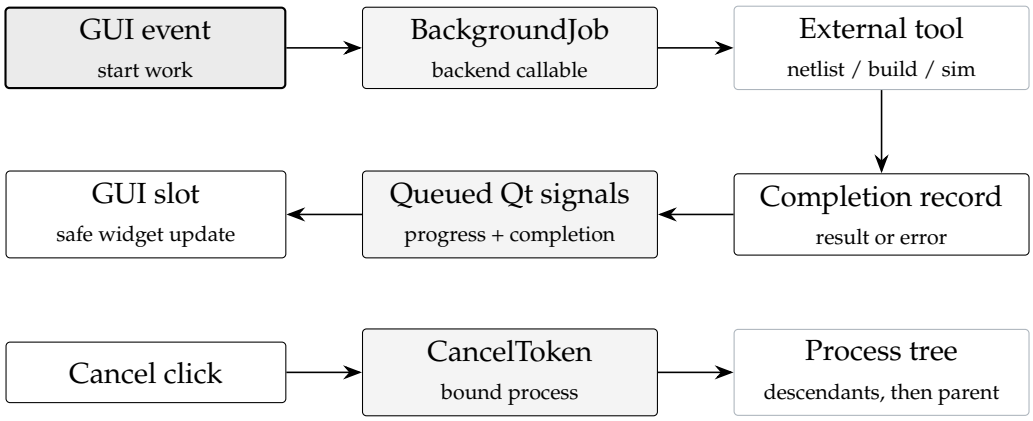


Figure 15.2: Background execution and cancellation ownership. Widgets remain on the GUI thread, while blocking programs and their descendants remain under worker control.

The same rule now covers KiCad netlist generation, the PSpice conversion path, Verilog compile and simulation, NgVeri model builds, and removal of generated models. Long-running jobs can report their current phase without granting a worker access to widgets. This makes responsiveness a reusable contract rather than a collection of one-off thread fixes.

15.4 Rendering: less screen work, complete evidence

Plot rendering has two distinct data products. The scientific product is the complete array parsed from Ngspice output. The screen product is a set of Matplotlib artists sized for the current viewport. Treating those products as the same object forced the renderer to transform every sample on every refresh.

For monotonic traces longer than 8,000 points, eSim now divides the visible range into 2,000 bins and retains each bin's minimum and maximum samples in encounter order. A single-sample spike cannot disappear inside an average. The raw arrays stay attached to the artist, and a zoom or pan starts a 150 ms single-shot timer that slices and decimates the new viewport. Function traces with non-monotonic x values bypass the optimization because reordering them would change their path. Cursor lookup, statistics, and export also bypass the display envelope and read the raw arrays.

Smaller interactions received similarly bounded paths. Visibility changes are delayed by 80 ms in the normal plot and 160 ms in stacked view, so a burst collapses to one refresh. When plot composition is unchanged, existing axes and artists are reused. Cursor drag saves the static canvas background and redraws only cursor lines with Matplotlib blitting. Editor search waits 120 ms after typing before rescanning a large source file.

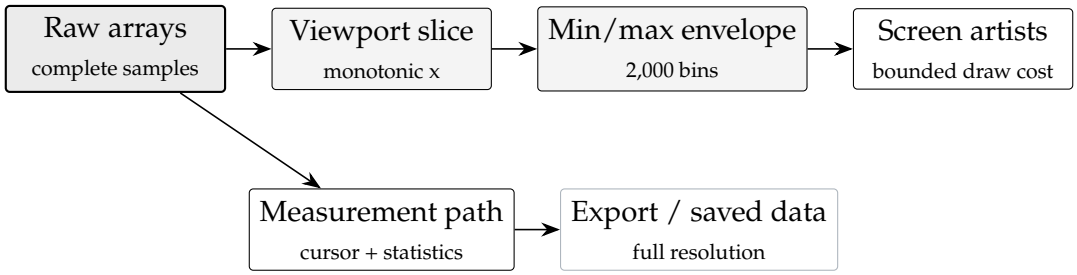


Figure 15.3: Rendering and measurement use different data paths. Decimation bounds drawing cost, while every operation that reports or saves a value retains full-resolution data.

The application shell also avoids permanent compositor work. Release 2.5 attached a live shadow effect to every toolbar button. The current motion service creates one glow only for the hovered control and releases it on leave. Theme changes freeze painting during the repolish, capture the old and new window states, and cross-fade two opaque snapshots. Each animation frame is therefore two image blits, independent of the number of widgets beneath the overlay. Reduced-motion mode uses the same atomic update without the dissolve.

15.5 Evidence and observed gains

Table 15.2 separates measured results from structural evidence. The historical plot benchmarks record the workload observed during implementation; the automated tests protect the invariants that matter across machines, such as envelope preservation, raw-data recovery after zoom, and descendant cleanup after cancellation.

> **Impact**
>
> Performance work in eSim 2.6 is visible as responsiveness, but its deeper result is bounded ownership. Startup has a smaller critical path, workers own slow programs and cancellation, the renderer receives only the detail the screen can use, and measurement continues to read the complete simulation. The speedups do not create a second, less trustworthy version of the user's data.

Table 15.2: Performance results and their evidence. Numeric claims are retained only where a recorded benchmark or release artifact supports them.

Area	Result	Evidence boundary
Raw-data extraction	About **2.4× faster** on representative DC, AC, and transient files	Recorded implementation benchmark for vectorized two-pass parsing
Repeated plot refresh	A roughly **464 ms** full rebuild was removed when the plot composition is unchanged	Incremental-path benchmark plus artist-identity tests
Long transient traces	Drawn from a bounded peak-preserving envelope; full samples restored on zoom	60,000-sample tests cover reduction, isolated peaks, and raw-data recovery
Cursor interaction	Full-figure redraw replaced by background restore, cursor-artist draw, and one blit	Code-level rendering path and cursor state tests
External tools	Window remains interactive; supported jobs expose progress and cancellation	Worker-signal tests, cancellation tests, and recursive process-tree cleanup tests
Startup	Optional tools leave the first-frame path and are warmed after startup	Local dock imports plus delayed module-only prewarm
Windows distribution	Final package about **21% smaller**, from roughly 478 MB to 379 MB	Previous-package record and final release artifact

Chapter 16

Conclusion and Future Work

eSim 2.6 is the delivered result of a system-wide modernization: a stable simulation core, a coherent desktop interface, complete HDL and mixed-signal workflows, version-aware conversion, and reproducible delivery on Ubuntu and Windows. This chapter draws those threads together and identifies the next engineering boundaries without reopening work that the release has already closed.

16.1 From isolated fixes to one maintained system

The internship began with a practical question: could the plotting work from the previous fellowship survive the full application around it? Answering that exposed boundaries well beyond plotting. A waveform window depends on reliable simulator processes, project paths, theme state, conversion output, tool discovery, and an installer that places every runtime where the application expects it. The modernization therefore followed the dependency chain instead of treating each visible feature as an island.

Four engineering campaigns emerged. Core stabilization made parsing defensive, plotting incremental, process handling explicit, and the PyQt6 migration behaviorally complete. Experience modernization introduced a token-driven interface, portable project identity, snapshots, structured conversion state, and an integrated editor. Capability work connected HDL authoring, functional verification, model generation, mixed-signal simulation, and KiCad conversion through defined workspaces. Delivery work then built and checked the same toolchain on Ubuntu and Windows, including the PDK and packaging failures that simple file existence checks had missed.

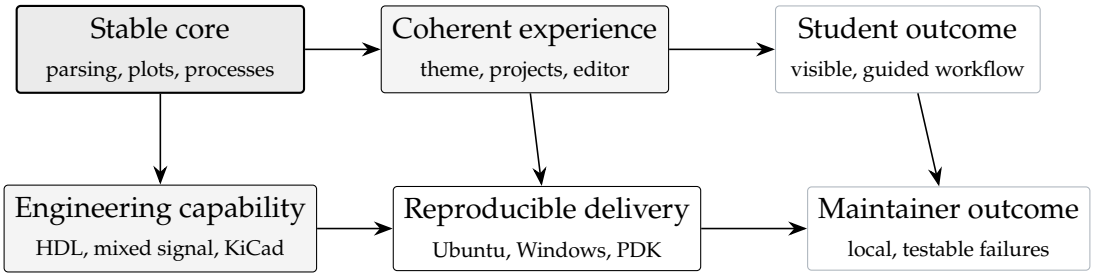


Figure 16.1: The modernization as one dependency graph. Core reliability supports both the desktop experience and engineering workflows; reproducible delivery closes the chain.

16.2 What eSim 2.6 now guarantees

The release is best summarized by the guarantees it provides at its boundaries. A guarantee is stronger than a feature name: it states what the user may rely on and where the maintainer can inspect the evidence.

Table 16.1: Delivered guarantees across the eSim 2.6 system.

Boundary	User guarantee	Engineering evidence
Simulation and plots	Valid output opens in stable normal, stacked, and timing views; measurement retains full-resolution samples	Defensive parser, process ownership, incremental artists, decimation invariants, and plot regression suites
Desktop interface	Light and dark themes, zoom, focus, dialogs, motion, and dock state act as one interface rather than unrelated screens	Central tokens and services, reduced-motion path, ownership rules, and rendered UI tests
Projects and source	A project may move, be renamed, snapshotted, restored, searched, and edited without relying on the current working directory	Anchor-based identity, metadata schema, restore validation, integrated editor, and project-state tests
HDL and mixed signal	Verilog and VHDL work moves through visible stages from source to diagnostics, waveform, model, and co-simulation	Shared workspace contracts, Icarus and GHDL adapters, VCD parser, model builder, and real mixed-backend examples
KiCad conversion	Supported KiCad projects are parsed structurally, model identity is stable, and success requires useful output	Version-aware readers, normalized component model, output gates, and golden conversion fixtures
Distribution	Ubuntu and Windows installations carry a known toolchain and explain a missing capability before a workflow starts	Pinned inputs, checksums, source builds, native launchers, bootstrap tests, and the shared doctor

Boundary	User guarantee	Engineering evidence
PDK and release health	A PDK is accepted only after an electrical probe succeeds; release staging rejects worktree administration metadata	Operating-point probes, controlled library seeding, exact exclusion rules, and clean-stage packaging checks

16.3 Impact for users and maintainers

For a student, the largest change is that complicated work has a visible shape. HDL source does not vanish into a terminal command: compile, run, diagnostics, waveform, and model generation are named stages. Digital backends can enter one Ngspice simulation and be compared on a shared time axis. A conversion failure points to the boundary that rejected the input. A missing compiler or code model is reported by capability instead of surfacing later as an unrelated dialog error.

For a maintainer, the important change is ownership. Project identity no longer belongs to the process working directory. Parser state no longer belongs to a widget. Theme values no longer live as copied color fragments. Subprocesses have a worker, a cancellation path, and a cleanup rule. Build inputs have versions and hashes; success checks inspect behavior as well as files. These boundaries narrow the distance between a symptom and the code that can explain it.

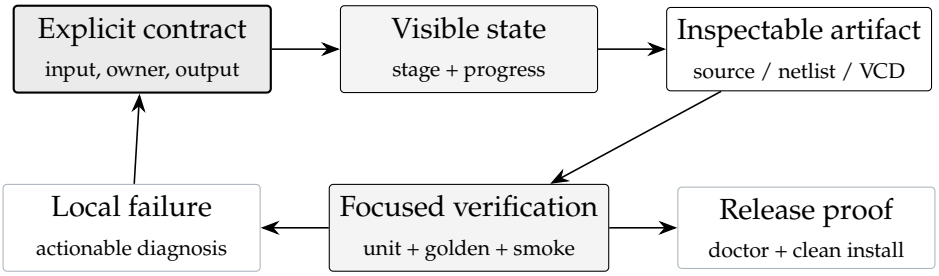


Figure 16.2: The maintenance loop left by eSim 2.6. Explicit contracts produce inspectable artifacts, focused tests, local diagnoses, and release-level proof.

16.4 Next engineering boundaries

Future work should extend the same contracts rather than add parallel implementations. The following items are ordered by the amount of release risk they remove and the number of workflows they strengthen.

Table 16.2: Prioritized future work after the eSim 2.6 release. Each item extends an existing boundary instead of replacing the delivered architecture.

Next boundary	Concrete extension	Completion evidence
Clean-machine Windows CI	Install every release candidate in a fresh virtual machine and exercise KiCad export, Ngspice, Icarus, GHDL, Verilator, PDK probes, and uninstall	Recorded doctor output plus one end-to-end mixed-signal smoke run
Conversion corpus	Add hierarchical sheets, library-table combinations, continuation forms, and third-party subcircuits from more KiCad versions	Golden successful outputs and exact expected failures
Teaching examples	Ship compact NgVeri, NGHDL, and `d_cosim` circuits with exercises and expected waveforms	Re-runnable projects whose saved traces match documented behavior
VHDL verification parity	Bring source-linked diagnostics and a waveform-first testbench loop to NGHDL	The same compile-to-waveform acceptance path used by Verilog
Linux packaging	Turn the reproducible layout into maintained distribution packages while preserving source-built mixed-signal support	Package install, doctor, example simulation, and removal in clean containers
Remaining interface seams	Route legacy dialogs through shared theme, ownership, zoom, and keyboard services	Focus, modality, escape, tab order, and dark-mode UI tests

16.5 Closing perspective

The first fellowship rebuilt one difficult subsystem: plotting. Returning to eSim made it possible to apply the same lesson across the application. State should have an owner; external work should cross an explicit boundary; success should produce an artifact that can be inspected; and a release should prove the environment in which that artifact is used.

> **Impact**
>
> eSim 2.6 is not a collection of disconnected interface changes. It is a maintained path from schematic and HDL source to simulation evidence, with portable project state, visible execution stages, reproducible tools, and failure messages that point back to the responsible boundary. That path is the main contribution of the internship and the foundation on which the next release can build.

Chapter A

Work Index and Metrics

This appendix keeps the audit trail separate from the technical narrative. The development history was classified by engineering concern and consolidated into eleven reviewable changes. The totals include the final library restoration and source handoff, but exclude the fork-only commit that uploads this report.

A.1 Measured scope

Table A.1: Repository-wide scope of the completed modernization. Merge commits are retained in the historical count but excluded from the change-count subtotal.

Measure	Result
Historical development commits	265
Non-merge changes	262
Merge commits	3
Files changed	416
Inserted lines	97,336
Deleted lines	27,912
Changed `test_*.py` files	114
Changed files under test directories	126
Review commits after consolidation	11

These are final-tree differences from upstream, not cumulative commit additions. The 15 packaged legacy libraries contribute 23,593 added lines of unchanged upstream data, not new application code. Restored originals reduce deletions; the development and review heads still resolve to the same Git tree.

A.2 Engineering work index

Table A.2: Where the implemented work is explained and what evidence closes each area.

Area	Principal result	Report evidence
Plotting and simulation	Stable parsing, incremental refresh, stacked traces, and safe process ownership	Chapters 2 and 15; waveform screens and data-path diagrams
PyQt6 and interface	Removed compatibility debt; token-driven light/dark theme, motion, preferences, and parented dialogs	Chapters 3 and 4; before/after interface screens
Project and conversion state	Anchor-based identity, snapshots, per-user caches, grouped models, subcircuit parsing, and non-empty output gates	Chapter 5; state and conversion diagrams
Editor and maker workspace	Integrated source editing and a staged HDL authoring flow with explicit state transfer	Chapters 6 and 7; editor and flow workspace screens
Functional verification	Icarus compile/run, testbench generation, diagnostic navigation, VCD decoding, and waveform launch	Chapter 8; verifier architecture, state model, and live interface
Digital co-simulation	Reproducible `d_cosim` model build and mixed-backend Ngspice execution	Chapter 9; real build log, netlist, converter, and waveform
NGHDL and Makerchip	Shared workspace contracts for VHDL, Verilog, and browser authoring; tool discovery and cleanup	Chapter 10; capability and bridge diagrams
KiCad conversion	KiCad 7–10 schematic and netlist handling, stable model identity, and golden-output protection	Chapter 11; version matrix and conversion model
Ubuntu delivery	Deterministic toolchain build, one-command install, dependency doctor, and release archive	Chapter 12; installation and diagnostic pipeline

Area	Principal result	Report evidence
Windows delivery	Native launcher, pruned private KiCad, source-built simulators, per-user bootstrap, and clean-stage packaging	Chapter 13; build and cross-platform diagrams
PDK and release closure	Electrical SKY130 validation, disabled-library seeding, exact packaging exclusions, and shared health gates	Chapter 14; PDK and doctor diagrams
Performance engineering	Deferred imports, background jobs, process-tree cancellation, bounded waveform drawing, and lower distribution cost	Chapter 15; ownership diagrams and measured-results table
System synthesis	User and maintainer guarantees, release evidence, and prioritized next boundaries	Chapter 16; guarantee matrix and maintenance loop

A.3 How the history was made reviewable

The eleven review changes retain the same engineering boundaries. Relative to upstream `4296828a`, development head `765d3f26` and review head `6ce4c993` have identical trees. The fork-only report upload `d51af337` is excluded. The earlier ledger records 263 entries; library restoration `2b34e737` and final handoff `765d3f26` bring the code history to 265 commits.

The closing changes restore converter libraries, retain upstream AppImage and Snap files without treating them as verified 2.6 release paths, and tighten release naming and tag checks. The standalone `library/CD40107B_test/` project is absent from the final tree but remains in Git history. The review branch contains eleven commits and no merges.

A.4 Reproduction and evidence map

Screenshots establish visible states, unit tests check isolated contracts, and installer runs test complete payloads. Table A.3 identifies the evidence and reproduction route for each claim.

Table A.3: Reproduction map for the report and release evidence.

Claim class	Primary evidence	Reproduction route
Visible interface state	Fresh eSim 2.6 screens captured from the current application	Open the cited workflow with the example shown in its chapter and compare the labeled state
Parser and state invariants	Focused Pytest suites beside the owning module	Run the chapter's subsystem tests from the repository root with the bundled Python environment
Conversion compatibility	Golden inputs, expected normalized models, and useful-output checks	Re-run each fixture through the converter and compare its normalized artifact
HDL and mixed-signal flow	Source, generated model, netlist, build log, VCD, and waveform	Run the documented example from source through compilation and Ngspice plotting
Toolchain health	Shared dependency doctor and electrical PDK probes	Run the doctor from the installed launcher; a missing capability produces a nonzero result
Windows distribution	Manifest hashes, clean staging, installer self-checks, and final release artifact	Build from `windows/build-windows.ps1`, install cleanly, and run the doctor plus an example export
Historical completeness	Earlier CSV ledger, closing commits above, and identical Git trees	Check the 265-commit source range, eleven review changes, and source-to-result tree equivalence
Report layout	Compiled PDF, full-page render set, text extraction, and boundary scan	Compile `report.tex`, render every page, inspect the visual index, and check for clipped content

A.5 Companion audit artifacts

The earlier machine-readable ledger and consolidation report preserve the historical audit. Their source identities, change classes, destinations, and verification records remain useful, but their snapshot totals predate the closing changes recorded above. This appendix gives the current code-only counts and heads. The recorded runtime tests and screenshots remain evidence from their original audits, not fresh runs of the final handoff. The chapters explain engineering decisions; Git identifies the delivered source.

Table A.4: Historical report and audit artifacts retained for reference.

Artifact	Purpose
`eSim_2.6_Internship_Report_Varadharam_RS.pdf`	Original fellowship report and visual evidence
`eSim_2.6_Internship_Report_LaTeX_Source.zip`	Earlier source, figures, and cross-reference files
`ESIM_ORIGINAL_COMMIT_LEDGER.csv`	Machine-readable coverage of the earlier development history
`ESIM_SQUASH_AND_INTERNSHIP_REPORT.md`	Consolidation boundaries, validation, and release audit

> **Impact**
>
> The technical chapters remain readable because chronology is confined to this appendix and its companion artifacts. The same separation gives reviewers both views: a system-level explanation of why eSim 2.6 works and an exact accounting of how the completed source was preserved.

List of Figures

1.1 The eSim design-to-result path used during the modernization. 2
1.2 The engineering proof loop used in this report. 3

2.1 The two-pass extraction path in `data_extraction.py`. 8
2.2 The current plotting architecture. 8
2.3 Standard view in the current eSim 2.6 interface. 9
2.4 Stacked view of the same transient. 10
2.5 Digital timing view derived from the same source arrays at a 2.5 V threshold. 10
2.6 The plotting data contract. 11

3.1 The migration boundary used for workflow validation. 13
3.2 Current Ngspice process lifecycle in `NgspiceWidget.py`. 15
3.3 The teardown order implemented by `Application.py`. 16

4.1 Aurora's rendering architecture in the current source. 20
4.2 Command identity and dock ownership in the current main window. . . . 21
4.3 Fresh captures from the current eSim 2.6 application. 22
4.4 The live theme transaction implemented by `theme_utils.py`. 23
4.5 Both pages of the current production Preferences dialog, captured directly from eSim 2.6. 24
4.6 Aurora's scaling path. 25
4.7 Subcircuit conversion state. 26
4.8 The Subcircuit Builder before and after modernization. 27
4.9 The same diode model in the legacy and current Model Editor, launched directly from the respective PyQt5 and PyQt6 source implementations. . . 28
4.10 Aurora applied to the current Schematic Converter. 29

5.1 Project resolution in the current source. 31
5.2 Fresh capture of the current Project Explorer. 31
5.3 Snapshot capture and restore. 32
5.4 Fresh current eSim capture of Project Snapshots. 33
5.5 Project-pinned netlist generation. 34
5.6 Fresh capture of the production Device Models tab. 35
5.7 The conversion contract. 35

6.1 Editor ownership in the current source. 38
6.2 Fresh capture of the current editor with a writable `.cir` deck. 39
6.3 Language and safety dispatch. 40
6.4 The same production editor displaying SystemVerilog. 41
6.5 Save and watcher ordering. 42

7.1 The current production Author stage. 45

7.2 The Verilog stages are views around one state owner. 46
7.3 Persistence rules in `DesignBus`. 46
7.4 Makerchip integration. 47
7.5 The current production removal dialog. 48
7.6 Orphan-aware model discovery. 48

8.1 Verifier architecture. 50
8.2 A successful run in the current production Verify stage. 50
8.3 One structural model serves the interface and the run. 51
8.4 The simulation pipeline. 52
8.5 A real failed Icarus syntax check in the current Verify stage. 52
8.6 The waveform produced by the run in Figure 8.2, displayed by eSim's current production plot window in digital-timing mode. 53
8.7 Destruction-safe run teardown. 53

9.1 The current eSim Convert stage captured from the production application. 55
9.2 The production `d_cosim` path. 56
9.3 Why the converter merges blocks. 57
9.4 Threshold handling in the generated wrapper. 58
9.5 A fresh production capture of the real merged co-simulation. 61

10.1 The current NGHDL builder captured from eSim 2.6 with a repository VHDL example selected. 63
10.2 NGHDL build and runtime ownership. 63
10.3 One NGHDL simulation exchange. 65
10.4 Capability-based GHDL selection. 66
10.5 The production toolchain doctor on the report machine. 67
10.6 Makerchip session lifecycle. 68

11.1 The same schematic at the compatibility boundary. 70
11.2 The maintained netlisting boundary. 71
11.3 Preserved-on-failure output. 73
11.4 The current production converter running against a copied mixed-signal example. 74
11.5 A current production capture of six `INVCMOS` placements grouped by model. 74
11.6 State ownership in the current converter. 75
11.7 The compatibility proof is layered. 76
11.8 The current ngspice-model tab for the same mixed-signal project. 77

12.1 Installer control flow. 80
12.2 The current Ubuntu installer terminal interface running through its repository demo harness. 81
12.3 Bootstrap ownership and convergence. 82
12.4 The installation assurance chain: choose the correct host profile, expose one plan, stop on required failures, and verify the resulting capabilities and payload. 84

13.1 The repository-owned Windows build. 86

13.2 The eSim 2.6 Windows installer as defined by the current Inno Setup source. 88
13.3 Windows ownership by lifetime. 89

14.1 The SKY130 release gate follows an exact repair state into a full corner load and an electrical measurement. 92
14.2 IHP OSDI configuration is derived from compiled outputs. 93
14.3 Toolchain diagnosis has one source of truth. 94
14.4 Fresh capture of the current eSim 2.6 toolchain dialog against the staged Full Windows product. 95
14.5 KiCad registration is an ownership-aware merge. 97

15.1 Startup critical path and deferred prewarm. 100
15.2 Background execution and cancellation ownership. 101
15.3 Rendering and measurement use different data paths. 102

16.1 The modernization as one dependency graph. 104
16.2 The maintenance loop left by eSim 2.6. 106

List of Tables

1.1 The four programmes and the user-facing condition each one had to establish. 2
1.2 Audited scale and release evidence for the completed work. 4

2.1 Defect groups addressed during plotting production hardening. 7
2.2 Representative plotting risks and the evidence used to close them. 12

3.1 PyQt6 changes and the eSim behavior affected by each one. 14
3.2 Verification layers retained in the current source tree. 17

4.1 Main Aurora modules in the audited eSim 2.6 source tree. 20
4.2 Aurora invariants and their visible effect. 25

5.1 State ownership after the project-management rebuild. 33
5.2 Conversion gates and where a failed operation remains. 36

6.1 Current editor module inventory in the audited eSim 2.6 tree. 38
6.2 Syntax treatment is selected from the file contract. 40
6.3 Editor responses are selected from both file ownership and buffer state. . . 42

7.1 The ownership boundaries introduced in the Maker subsystem. 45

8.1 Failure surfaces and the response visible to the user. 54

9.1 Boundary failures and the layer that owns each correction. 59
9.2 Selected decoded states from the real three-block run. 60

10.1 NGHDL failure modes converted into explicit contracts. 66
10.2 Contracts exposed by the HDL workspace. 69

11.1 Translation contracts enforced by the structured netlister. 72
11.2 Observable behaviour at common project boundaries. 76

12.1 Release behavior selected by the current `detect_profile()` implementation. 79
12.2 Failure semantics in the current Ubuntu installer. 83
12.3 What the clean-room comparison established. 84

13.1 The Windows deliverable before and after the repository-owned build. . . 86
13.2 Pinned inputs and the boundary each one owns. 87
13.3 Windows-specific failure classes and the checks that now contain them. . . 90

14.1 Representative capability groups used by each simulation flow. 96
14.2 Ownership rules for KiCad symbol registration. 96
14.3 Release checks ordered by the strength of the evidence they provide. . . . 98

15.1 The performance contract used across eSim 2.6. 99
15.2 Performance results and their evidence. 103

16.1 Delivered guarantees across the eSim 2.6 system. 105
16.2 Prioritized future work after the eSim 2.6 release. 107

A.1 Repository-wide scope of the completed modernization. 108
A.2 Where the implemented work is explained and what evidence closes each area. 109
A.3 Reproduction map for the report and release evidence. 111
A.4 Historical report and audit artifacts retained for reference. 112

Bibliography

[1] eSim, an open-source EDA tool. FOSSEE, IIT Bombay.
https://esim.fossee.in

[2] FOSSEE eSim repository.
https://github.com/FOSSEE/eSim

[3] PR #416: Modularized plotting interface.
https://github.com/FOSSEE/eSim/pull/416

[4] PR #506: Plotting data-extraction rewrite and correctness fixes.
https://github.com/FOSSEE/eSim/pull/506

[5] PR #520: Stacked view, incremental refresh, architecture cleanup.
https://github.com/FOSSEE/eSim/pull/520

[6] PR #521: PyQt6 migration completion and crash fixes.
https://github.com/FOSSEE/eSim/pull/521

[7] eSim 2.6 source and release lineage.
https://github.com/VaradhaCodes/eSim

[8] Ngspice circuit simulator.
https://ngspice.sourceforge.io

[9] KiCad EDA.
https://www.kicad.org

[10] NGHDL mixed-signal simulation for eSim.
https://github.com/FOSSEE/nghdl

[11] Icarus Verilog.
https://steveicarus.github.io/iverilog/

[12] PyQt6 Reference Guide. Riverbank Computing.
https://www.riverbankcomputing.com/static/Docs/PyQt6/

[13] eSim v2.6 release and platform artifacts.
https://github.com/VaradhaCodes/eSim/releases/tag/v2.6

[14] eSim 2.6 modernization review branch.
https://github.com/VaradhaCodes/eSim/tree/upstream/esim-2.6-modernization

[15] KiCad schematic file format documentation.
https://dev-docs.kicad.org/en/file-formats/sexpr-schematic/

[16] Ngspice mixed-signal and digital co-simulation documentation.
https://ngspice.sourceforge.io/docs.html

[17] GHDL documentation.
https://ghdl.github.io/ghdl/

[18] Verilator documentation.
https://verilator.org/guide/latest/

[19] OpenVAF documentation and source.
https://github.com/pascalkuthe/OpenVAF

[20] SKY130 open-source PDK documentation.
https://skywater-pdk.readthedocs.io/

[21] IHP Open PDK documentation and source.
https://github.com/IHP-GmbH/IHP-Open-PDK

[22] MSYS2 documentation.
https://www.msys2.org/docs/

[23] Inno Setup documentation.
https://jrsoftware.org/ishelp/

[24] Matplotlib documentation.
https://matplotlib.org/stable/