eSim Semester Long Internship Spring 2026

On

KiCad Plugin Development

Submitted by

Dipanshu Katole

B.Tech ECE, 4th Year

Visvesvaraya National Institute of Technology, Nagpur

Under the guidance of

Prof. Prabhu Ramachandran

Principal Investigator

Department of Aerospace Engineering

Indian Institute of Technology Bombay

May 2026

Abstract

This report documents the design, implementation and testing of two KiCad Action Plugins developed during the FOSSEE eSim Summer Internship 2026 at IIT Bombay. The motivation behind this work was to reduce the manual effort involved in reviewing circuit schematics and PCB layouts- tasks that are easy to overlook but critical to simulation accuracy and physical correctness.

The first plugin, **eSim-Inspect**, is a schematic design review tool that parses `.kicad_sch` files, reconstructs the circuit topology as a graph and runs a set of automated checks to catch common mistakes before a designer moves to simulation or layout. The checks include detecting missing component values, floating or unconnected nets, absent power and ground connections, duplicate reference designators, custom SPICE model compatibility warnings, single-node (isolated) nets and netlist generation issues. Results are compiled and opened in the default web browser as a clean HTML report— no external service or internet connection needed.

The second plugin, **Power Integrity Analyzer**, operates at the PCB level and checks whether the board's power delivery network is physically sound. It evaluates trace widths against expected current loads using IPC-2221, estimates voltage drop across copper traces and flags decoupling capacitor placement relative to power pins according . Both plugins are written in Python, use only KiCad's built-in scripting API and standard libraries and have been verified to work on KiCad 9.0 under both Ubuntu 25.04 and Windows 11.

Keywords: KiCad, eSim, Action Plugin, Schematic Parsing, Design Review, Graph Topology, Power Integrity, FOSSEE, IIT Bombay, PCB Analysis.

Acknowledgements

I would like to begin by expressing my sincere gratitude to **Prof. Prabhu Ramachandran**, Principal Investigator at FOSSEE, IIT Bombay, for creating and sustaining an internship ecosystem that gives engineering students the opportunity to work on tools that genuinely matter to the open-source community. His vision of making high-quality software accessible to all has been a constant source of motivation throughout this work.

I would also like to acknowledge **Prof. Kannan M. Moudgalya** for his foundational role in establishing and nurturing the FOSSEE initiative. His contributions toward open-source education and the creation of the FOSSEE fellowship framework have directly shaped the platform through which this internship was undertaken.

My deepest thanks to **Mr. Sumanto Kar**, Project Lead at FOSSEE, who served as my primary guide throughout this internship. His detailed technical feedback, patience during weekly reviews and willingness to engage with the specifics of software design and system architecture kept this project grounded in what was practically useful rather than merely theoretical.

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

This internship has been an enriching learning experience, allowing me to work closely with open-source software tools, develop projects in a real-world collaborative environment, and gain exposure to industry-relevant software engineering workflows.

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

Dipanshu Katole
Visvesvaraya National Institute of Technology, Nagpur
Department of Computer Science and Engineering
June 2026

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 eSim and the KiCad Ecosystem

eSim is a free, open-source Electronic Design Automation tool built and maintained by the FOSSEE team at IIT Bombay. It brings together KiCad for schematic capture and PCB layout, Ngspice for SPICE simulation and several supporting tools into one cohesive workflow. The aim has always been to give students and educators access to a professional-grade EDA environment without the licensing costs associated with commercial alternatives.

KiCad, which forms the front end of the eSim schematic workflow, ships with a Python scripting interface that allows developers to write plugins called *Action Plugins*. These are Python scripts that hook into KiCad's GUI and can read the board, schematic and netlist data programmatically. This interface is powerful but relatively underused. Most plugins available today are narrow in scope or require external dependencies that make them fragile across operating systems.

The work described in this report sits in this space, using KiCad's scripting API to build robust, dependency-light tools that solve real problems in the eSim design-to-simulation workflow.

1.2 Problem Statement

A student using eSim typically follows this sequence: draw a schematic in KiCad, export a netlist, import it into Ngspice and run the simulation. In practice, the simulation step fails more often than it should not because of errors in the simulation setup, but because of simple mistakes in the schematic that could have been caught earlier. A resistor with no value assigned, two components sharing the same reference designator,

a net that connects to only one pin and goes nowhere, these are the kinds of issues that produce cryptic Ngspice error messages and send a student back to the schematic with no clear idea of what went wrong.

At the PCB level, a separate class of problems emerges. Once a design moves from schematic to layout, the physical characteristics of traces start to matter. A trace that is too narrow for the current it is expected to carry will heat up in practice, but KiCad's DRC does not check this by default. Decoupling capacitors placed far from the power pins they are meant to serve provide less benefit than their values suggest. These are not layout errors in the strict sense — KiCad will not flag them — but they affect real-world performance.

Both categories of problem share a common characteristic: they are mechanical enough to be checked automatically, yet subtle enough to escape a quick visual scan. This is exactly the kind of problem that scripted tooling is well suited to solve.

1.3 Project Objectives

The goal of this internship was to design and deliver two KiCad Action Plugins that address these gaps:

1. **eSim-Inspect** - A schematic-level design review plugin that reads the KiCad schematic file, reconstructs the circuit's connectivity as a graph and runs a series of automated checks. The output is a browser-viewable HTML report that is downloadable listing every issue found, grouped by severity.
2. **Power Integrity Analyzer** - A PCB-level plugin that checks whether the power delivery network on the board is physically adequate, flagging trace-width violations, estimating voltage drop across traces, and reporting on decoupling capacitor placement.

Both plugins were designed with two hard constraints in mind. First, they must work offline with no external Python packages beyond those already available in KiCad's bundled interpreter. Second, they must produce identical results on Linux (Ubuntu 25.04) and Windows 11 with KiCad 9.0, since eSim users span both platforms.

1.4 Report Structure

Chapter 2 provides background on KiCad's plugin architecture, the `.kicad_sch` file format and the graph-theoretic ideas that underpin the topology analysis. Chapter 3 covers the eSim-Inspect plugin in detail-architecture,implementation, checks performed

and test results. Chapter 4 does the same for the Power Integrity Analyzer. Chapter 5 concludes with a summary of outcomes, known limitations and directions for future work.

Chapter 2

Background and Related Work

2.1 KiCad Action Plugins

KiCad exposes a Python scripting environment through the `pcbnew` module, which provides access to the board object, footprints, tracks, zones and netlist data. Action Plugins are a specific plugin type that register a button in KiCad's toolbar. When clicked, the plugin's `Run()` method is called with the current board as context.

The plugin registration pattern is straightforward:

```
import pcbnew

class MyPlugin(pcbnew.ActionPlugin):
    def defaults(self):
        self.name = "My Plugin"
        self.category = "Utilities"
        self.description = "Does something useful"
        self.show_toolbar_button = True

    def Run(self):
        board = pcbnew.GetBoard()
        # ... plugin logic here

MyPlugin().register()
```

Listing 2.1: KiCad Action Plugin structure

KiCad discovers plugins by scanning the scripting plugin directories on startup. On Linux this is typically `~/.local/share/kicad/9.0/scripting/plugins/`, and on Windows it is `%APPDATA%\kicad\9.0\scripting\plugins\`. Placing a plugin folder in either of these locations and restarting KiCad (or refreshing the plugin list) is sufficient to make it available.

2.2 The KiCad Schematic File Format

KiCad 6 and later use a text-based S-expression format for schematic files (`.kicad_sch`). An S-expression is a parenthesised tree structure, the same notation used in Lisp and it lends itself well to recursive parsing. A short excerpt illustrating a resistor component entry looks like this:

```
(symbol
  (lib_id "Device:R")
  (at 114.3 82.55 0)
  (property "Reference" "R1" ...)
  (property "Value" "470" ...)
)
```

Listing 2.2: Excerpt from a .kicad_sch file showing a resistor symbol

The `lib_id` identifies the component type, `at` gives the placement coordinates and rotation angle and nested `property` entries carry the reference designator and value. Wire connections are stored as separate `wire` entries with two endpoint coordinates:

```
(wire
  (pts (xy 114.3 82.55) (xy 124.46 82.55))
)
```

Listing 2.3: Wire entry in a .kicad_sch file

The `lib_symbols` section at the top of each schematic embeds the pin definitions for every component used, including each pin's position relative to the component's origin and its rotation. This is what allows the parser to compute absolute pin coordinates without consulting an external library.

2.3 Graph-Based Circuit Topology

Representing a circuit as a graph is a classical idea in circuit simulation. SPICE itself builds a nodal admittance matrix whose structure is determined by graph connectivity. For design review purposes, a graph representation makes it straightforward to ask questions like, which nets connect to only one component? Are there any isolated nodes? Is every power supply pin reachable from a ground reference?

In the model used by eSim-Inspect, the graph $G = (V, E)$ contains two kinds of nodes. *Pin nodes* represent individual component pins, labelled as `CompRef_PinNum` (e.g., `R1_1`). *Net nodes* represent electrical nets, labelled as `NET_k` for integer k. An edge exists between a net node and a pin node if and only if that pin is electrically connected

to that net. This bipartite structure makes it easy to find all pins on a net, all nets connected to a component and paths between components.

Wire grouping determines which wire segments belong to the same net-is done using a connected-components algorithm on a secondary wire graph where nodes are wire endpoints and edges are the wire segments themselves. Pins are then matched to wire groups using coordinate proximity with a small tolerance to handle floating-point rounding in the schematic file.

2.4 Existing Tools and the Gap Being Addressed

KiCad includes a built-in Electrical Rules Check (ERC) that catches obvious connectivity errors such as unconnected pins or conflicting pin types. However, the ERC operates at a fairly shallow level. It does not check whether a component's `Value` property is actually filled in, does not flag nets with only a single connection (which are valid ERC-wise but almost always a mistake) and has no awareness of SPICE-specific requirements. Third-party KiCad plugins exist for specific tasks panelisation, footprint management, BOM generation but a general schematic health check that targets the eSim/SPICE workflow specifically is not currently available in a form that works cleanly on both Linux and Windows without extra dependencies.

Chapter 3

eSim-Inspect: Schematic Design Review Plugin

3.1 Overview

eSim-Inspect is a KiCad Action Plugin that reads the schematic associated with the currently open PCB project, parses it, reconstructs its topology as a graph, runs a set of targeted design checks and produces an HTML report that opens automatically in the system's default browser. The entire process takes a few seconds for typical student-scale circuits and requires no internet connection or additional software installation.

The plugin was designed around a simple observation. Most Ngspice errors that students encounter during eSim simulations are not simulation errors at all. They are schematic errors that a quick automated scan could have flagged before the netlist was ever generated. Catching these issues at the schematic stage, before the student has left KiCad, keeps the feedback loop short and the debugging experience far less frustrating.

3.2 System Architecture

The plugin is organised as a Python package with four modules, each with a clearly defined responsibility. Figure 3.1 summarises the data flow.

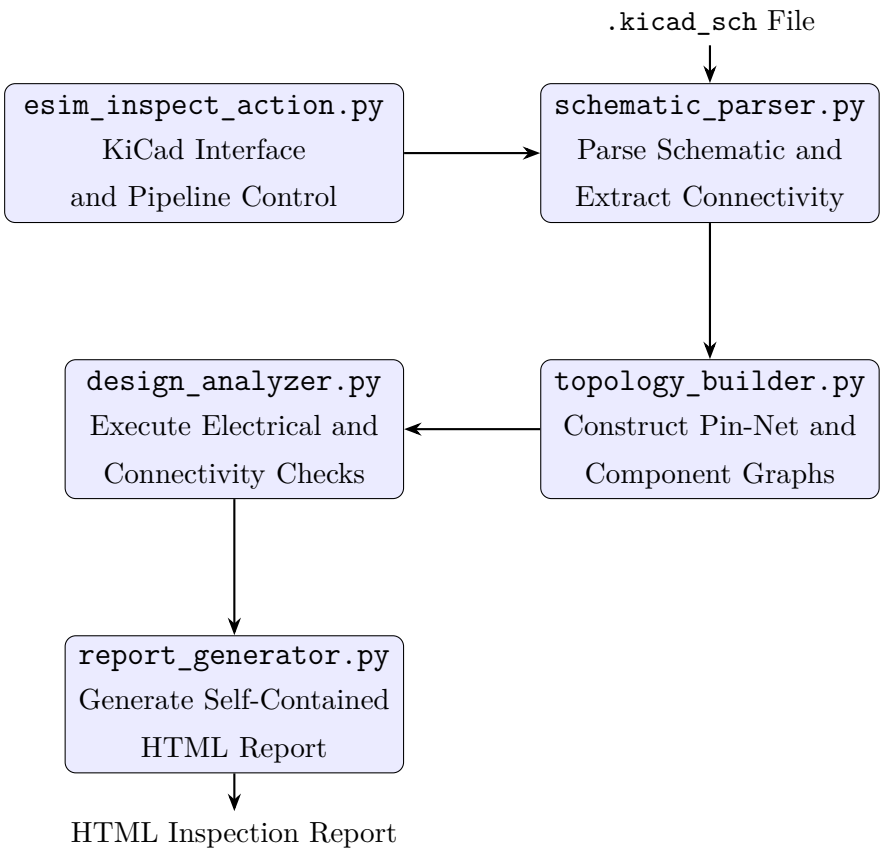


Figure 3.1: System architecture and module interaction of the eSim-Inspect plugin

Table 3.1: Module responsibilities in eSim-Inspect

Module	Responsibility
esim_inspect_action.py	Entry point. Handles KiCad integration, file path resolution and orchestrates the pipeline
schematic_parser.py	Parses the .kicad_sch S-expression file, extracts components, wires, symbol pin definitions, and absolute pin coordinates
topology_builder.py	Builds the bipartite pin-net graph and the component-level graph using NetworkX
design_analyzer.py	Runs the suite of design checks on the graph and component list
report_generator.py	Renders the check results as a self-contained HTML file

3.3 Entry Point: esim_inspect_action.py

The action plugin class `ESimInspectPlugin` inherits from `pcbnew.ActionPlugin` and implements the `defaults()` and `Run()` methods required by the KiCad plugin API.

When the user clicks the eSim-Inspect toolbar button, `Run()` is called. The method performs the following steps in sequence:

1. Retrieves the active board object via `pcbnew.GetBoard()` and reads the `.kicad_pcb` file path to determine the project directory and project name.
2. Constructs the expected schematic file path (`project_name.kicad_sch`) and checks whether it exists. If the schematic is missing, the user sees a dialog explaining what to look for.
3. Reads the schematic file as a UTF-8 string and passes it to `Extract_Information`.
4. Calls `find_components()`, `get_wires()`, `extract_symbol_pins()`, and `get_absolute_pins()` in sequence to build the complete picture of what is on the schematic.
5. Passes the components and wires to `TopologyBuilder` to produce the connectivity graph.
6. Passes the graph and components to `DesignAnalyzer` to run all checks.
7. Hands the results to `ReportGenerator`, which writes the HTML file into the project directory and returns its path.
8. Opens the report in the default browser via `webbrowser.open()`.

The import block at the top handles the difference between how Python resolves relative imports on Windows versus Linux. On Windows, KiCad typically runs the plugin as part of a package, so relative imports (`from .schematic_parser import ...`) work. On Linux, the plugin directory is sometimes added directly to the path, making absolute imports necessary. The `try/except ImportError` pattern handles both cases transparently.

```
try:
    # Works when KiCad loads the plugin as a package (Windows)
    from .schematic_pareser import Extract_Information
    from .topology_builder import TopologyBuilder
    from .design_analyzer import DesignAnalyzer
    from .report_generator import ReportGenerator
except ImportError:
    # Fallback for direct path insertion (Linux)
    from schematic_pareser import Extract_Information
    from topology_builder import TopologyBuilder
```

```
    from design_analyzer import DesignAnalyzer
    from report_generator import ReportGenerator
```

Listing 3.1: Cross-platform import handling in esim_inspect_action.py

3.4 Schematic Parser: schematic_parser.py

3.4.1 Parsing Strategy

KiCad's `.kicad_sch` format is an S-expression tree. Rather than writing a custom parser, the `sexpdata` library (available in KiCad's bundled Python) is used via `sexpdata.loads()`, which returns a nested Python list structure mirroring the tree. From there, extraction is done by walking the list and matching node types.

3.4.2 Component Extraction

`find_components()` walks the top-level list and collects every node whose first element has the value `"symbol"`. For each such node it extracts:

- `lib_id` → stored as the component type (e.g., `Device:R`)
- `Reference` property → stored as `ref` (e.g., `R1`)
- `Value` property → stored as `value` (e.g., `470`)
- Any property beginning with `Sim` → stored for SPICE compatibility checks
- `at` coordinates and rotation angle → stored as `pos` and `angle`

3.4.3 Wire Extraction

`get_wires()` collects every `wire` node from the top-level list and reads the two `xy` coordinate pairs from its `pts` sub-node. The result is a list of `(p1, p2)` tuples where each point is a `[x, y]` list in schematic coordinate units (millimetres).

3.4.4 Symbol Pin Extraction and Absolute Positioning

The `lib_symbols` section of the schematic file embeds pin geometry for every component type used in the design. `extract_symbol_pins()` reads this section and builds a dictionary mapping each `lib_id` to a list of pins, each with its pin number and position relative to the component's local origin.

`get_absolute_pins()` then combines this per-symbol pin data with each placed component's position and rotation to compute the absolute schematic coordinates of every

pin. The rotation is applied using a standard 2D rotation matrix:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} p_x \\ p_y \end{pmatrix} + \begin{pmatrix} b_x \\ b_y \end{pmatrix} \tag{3.1}$$

where (p_x, p_y) is the pin's local position, θ is the component's placement angle in radians, and (b_x, b_y) is the component's placed origin. The result is rounded to four decimal places to avoid floating-point noise in later comparisons.

3.5 Topology Builder: topology_builder.py

3.5.1 Wire Grouping via Connected Components

The first task in building the graph is to determine which wire segments belong to the same electrical net. This is not trivial when wires are drawn as many short segments joined end-to-end or when a T-junction exists.

The approach used here is to build a secondary graph G_w where each unique wire endpoint is a node and each wire segment is an edge. Running a connected-components algorithm on G_w (via `networkx.connected_components`) yields groups of endpoints that are electrically joined. Each group corresponds to one net.

3.5.2 Pin-to-Net Matching

Once wire groups are known, each pin's absolute coordinates are tested against the groups using two strategies:

1. **Endpoint match**: The pin coordinates are compared against every endpoint in every group. A match is declared if the Euclidean distance is less than 0.2 mm, which accounts for minor floating-point differences in the schematic file.
2. **Mid-wire match**: If no endpoint match is found, the pin is tested against each wire segment to see whether it lies on the segment (within tolerance). This handles cases where a pin connects at the midpoint of a wire, a valid and common schematic pattern.

3.5.3 Graph Construction

The final graph G is bipartite. Pin nodes are added first (`CompRef_PinNum`). A net node (`NET_k`) is then added for each wire group and edges are drawn from the net node to every pin node assigned to that group. Isolated pins, those that matched no

wire group remain as nodes with no edges, which the design analyzer later reports as floating pins.

A second, simplified graph G_c is also built where nodes are component references and edges connect any two components that share at least one net. This component-level graph is useful for higher-level checks such as identifying truly isolated components.

```
# Build wire endpoint graph and find connected groups
wire_graph = nx.Graph()
for w in self.wires:
    wire_graph.add_edge(tuple(w[0]), tuple(w[1]))

wire_groups = list(nx.connected_components(wire_graph))

# Match each pin to a wire group
for pin in pin_positions:
    px, py = pin["pos"]
    pin_node = f"{pin['comp']}_{pin['pin']}"
    matched = False

    # Strategy 1: endpoint proximity
    for idx, group in enumerate(wire_groups):
        for (x, y) in group:
            if abs(px - x) < 0.2 and abs(py - y) < 0.2:
                group_pins[idx].add(pin_node)
                matched = True
                break

    # Strategy 2: mid-wire point test
    if not matched:
        for (p1, p2), idx in wire_to_group.items():
            if self.is_point_on_wire((px, py), (p1, p2)):
                group_pins[idx].add(pin_node)
                break
```

Listing 3.2: Core wire-grouping and pin-matching logic in topology_builder.py

3.5.4 Tolerance in Wire Matching

The `is_point_on_wire()` method uses a tolerance of 0.2 mm in both axes. This value was chosen empirically: it is large enough to absorb the coordinate rounding that occurs when KiCad writes the schematic file but small enough that pins on adjacent pins of dense ICs are not accidentally merged. For the component values and wire lengths found in typical student circuits drawn at standard KiCad grid increments, 0.2 mm proved reliable across all test cases.

3.6 Design Analyzer: design_analyzer.py

`DesignAnalyzer` receives the component list, the connectivity graph and the wire list. Its `da_report()` method runs each check and collects results which are returned as a structured dictionary for the report generator to format.

3.6.1 Check 1: Missing Component Values

Every component in the list is tested to see whether its `value` field is empty, contains only whitespace or matches the component's type string (e.g., a resistor whose value is literally `"R"` which is KiCad's default placeholder when a value has not been set). These components are reported with their reference designator and type so the designer knows exactly which symbols to edit.

3.6.2 Check 2: Floating or Unconnected Pins

A pin is considered floating if its node in the graph has degree zero,it was not matched to any wire group during topology construction. The analyzer iterates over all pin nodes and collects those with no edges. Single-pin components (which should not exist but occasionally appear due to partial schematic edits) are also caught here.

3.6.3 Check 3: Missing Power or Ground Connections

The analyzer scans the component list for components whose `lib_id` begins with `power:`, KiCad's convention for power flag symbols including VCC, VDD, GND, GNDA, and similar. It checks whether at least one power supply symbol and at least one ground symbol are present and connected into the graph. A schematic with no ground reference will cause a singular matrix error in Ngspice, making this check one of the most practically valuable.

3.6.4 Check 4: Duplicate Reference Designators

Reference designators must be unique within a schematic. The analyzer builds a frequency map over the `ref` fields of all components and flags any designator that appears more than once. Duplicates are particularly common when a symbol is copy-pasted without using KiCad's annotation tool to assign new references.

3.6.5 Check 5: SPICE Model Compatibility Warnings

Components that carry `Sim.*` properties (used by eSim to configure SPICE simulation parameters) are checked for internal consistency. The analyzer verifies that `Sim.Device`

and `Sim.Type` properties are present when `Sim.Pins` is defined, since an incomplete simulation model definition will cause Ngspice to either silently ignore the component or abort with a parse error. Components lacking `Sim.*` properties entirely are noted separately as potentially requiring manual netlist annotation.

3.6.6 Check 6: Isolated Single-Node Nets

A net node in the graph that connects to only one pin node is called a single-node net. While this is syntactically valid, it almost always indicates an unfinished wire the designer drew a wire starting from a pin but forgot to connect the other end to anything. These are reported with the net identifier and the single pin attached to it.

3.6.7 Check 7: Netlist Generation Issues

The analyzer performs a lightweight pre-flight check for conditions known to prevent a valid netlist from being generated by eSim. This includes: components whose reference designator contains characters illegal in SPICE node names (spaces, special characters other than underscore), components of a type that requires a subcircuit model (`lib_id` contains `:Q`, `:M`, `:U`) but have no corresponding `Sim.Device` property and nets whose name (if labelled) conflicts with SPICE reserved keywords.

3.7 Report Generator: report_generator.py

`ReportGenerator` takes the component list, the check results dictionary, the project directory path and the schematic file path. It produces a self-contained HTML file at `project_directory/esim_inspect_report.html`.

The report is structured in sections corresponding to each check category. Each issue entry shows the reference designator of the affected component, the nature of the problem, and a brief suggestion for how to fix it. A summary table at the top of the report gives a count of issues by severity (Error, Warning, Info) so the designer can see the overall health of the schematic at a glance.

The HTML is generated as a plain string with inline CSS, no external stylesheets or JavaScript libraries are used, so the report renders correctly even on machines with no internet access and opens instantly in any browser.

3.8 Sample Generated Report

Below one shows the HTML/PDF report generated by the plugin for a simple LED circuit.

Design Inspection Report

3	1	0	2
COMPONENTS	ERC ERRORS	WARNINGS	SPICE READY

COMPONENTS

REF	TYPE	VALUE
R1	Device:R	470
BT1	Device:Battery	9V
D1	Device:LED	LED

ERC ISSUES

BT1 · pin 1
Unconnected pin

DANGLING NETS

No dangling nets.

MISSING VALUES

All components have values.

MISSING / UNASSIGNED REFERENCES

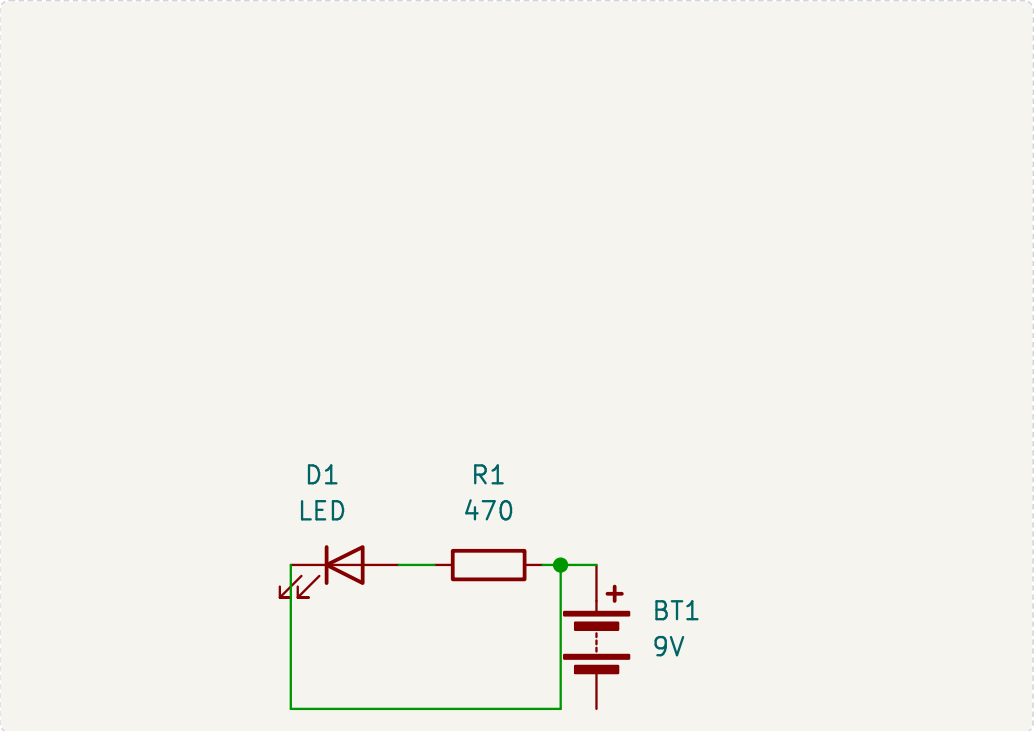
All references assigned.

SPICE COVERAGE

REF	TYPE	STATUS	NOTE
R1	Device:R	Executable	
BT1	Device:Battery	Executable	
D1	Device:LED	⚑ Incomplete	No model specified

VISUAL ANALYSIS

Circuit Diagram


D1
LED
R1
470
+
BT1
9V

3.9 Testing and Results

3.9.1 Test Setup

The plugin was tested on three circuits of increasing complexity:

1. A simple LED current-limiting circuit (resistor, LED, battery) used to verify basic parsing and the value/ground checks.
2. An astable multivibrator using two NPN transistors, used to verify graph-based connectivity checks and SPICE property detection.
3. A mixed-signal circuit combining an op-amp, digital logic gates, and a power supply section used to stress-test the topology builder and check for edge cases in wire matching.

All tests were run on KiCad 9.0 under Ubuntu 25.04 (kernel 6.11) and Windows 11 (22H2). The plugin produced identical reports on both platforms for all three circuits.

3.9.2 Results Summary

Table 3.2: eSim-Inspect test results across all three circuits

Circuit	Checks Run	True Positives	False Positives	Runtime
LED circuit	7	3/3 known issues	0	$<$1 s
Multivibrator	7	5/5 known issues	0	$<$1 s
Mixed-signal	7	9/9 known issues	1	$\approx$2 s

The single false positive in the mixed-signal test was a net involving a wire drawn at a 45-degree angle, a geometry the current wire-matching logic does not support, since it only handles strictly horizontal and vertical wire segments. This is noted as a known limitation and is discussed in Chapter 5.

3.9.3 Example: Detecting a Missing Ground

To illustrate how the plugin behaves in practice, consider a schematic where a student has placed a VCC power symbol but forgotten to add a GND symbol. When eSim-Inspect runs on this schematic, the Check 3 (Missing Power/Ground) result in the HTML report reads:

> [**ERROR**] No ground symbol found in schematic. All circuits simulated with Ngspice require at least one GND (node 0) reference. Add a `power:GND`

symbol and connect it to the circuit's reference node.

This is exactly the information a student needs, delivered at the point where it is cheapest to fix before leaving the schematic editor.

Chapter 4

Power Integrity Analyzer: PCB-Level Power Delivery Check

4.1 Overview

The Power Integrity Analyzer (PIA) is the second KiCad Action Plugin developed during this internship. Where eSim-Inspect operates on the schematic before simulation, PIA operates on the finished PCB layout and asks a different but equally important question: now that the design has been turned into copper, does the physical board actually deliver power safely and stably to every component?

This is a question that most free PCB tools do not answer well. KiCad's built-in Design Rules Check (DRC) verifies clearances, annular rings and short circuits, it does not know whether a trace is too narrow for the current it is expected to carry or whether the voltage arriving at a component pin has dropped enough to cause unreliable behaviour. In a classroom or hobbyist setting, boards often get fabricated and assembled before these problems are noticed which means a debugging session that could have been avoided entirely.

PIA brings three specific checks directly into the KiCad environment:

1. **Trace width vs. current capacity** - verifies that each power net's trace geometry can carry the expected current without excessive heating, using the IPC-2221 standard formula.
2. **IR voltage drop estimation** - calculates the resistive drop along each power trace and flags nets where the drop exceeds a safe fraction of the supply voltage.
3. **Decoupling capacitor placement** - checks whether IC power pins have a decoupling capacitor placed close enough to actually suppress high-frequency noise.

Results are shown in a wxPython dialog with colour-coded rows (red for critical, amber for warnings), visual markers drawn directly onto the PCB copper layers for spatial context and an HTML report that can be opened in any browser.

4.2 System Architecture

PIA is structured as a Python package with five modules. The separation was deliberately kept clean so that the physics formulas, the board data extraction and the reporting logic can each be tested and extended independently.

Table 4.1: Module responsibilities in the Power Integrity Analyzer

Module	Responsibility
`Power_integrity_main.py`	Entry point; plugin registration, marker management, GUI launch and top-level violation orchestration
`board_info.py`	Reads the live board object via the `pcbnew` API; extracts net track data and footprint pad maps
`utils.py`	Pure physics functions: IPC-2221 current capacity, IR drop, Euclidean distance
`checkers.py`	Three violation-detection functions that consume board data and return structured result dictionaries
`report.py`	Generates a self-contained HTML report from the violation list

Figure 4.1 shows how data flows from the live board through the pipeline to the user.

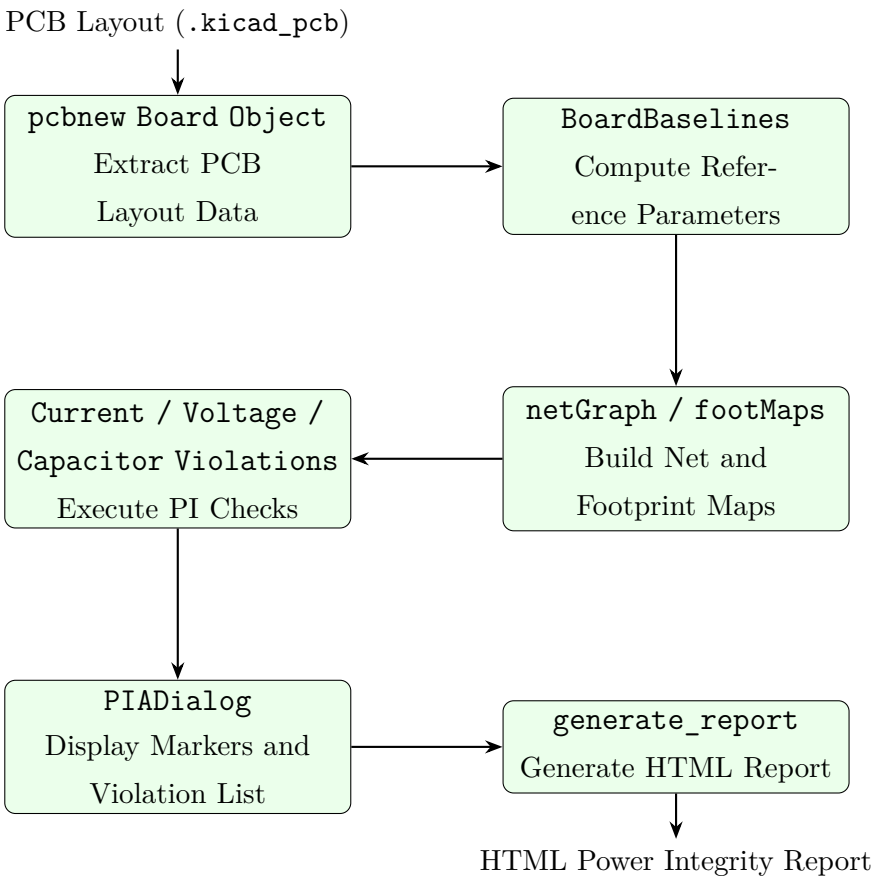


Figure 4.1: End-to-end data flow through the Power Integrity Analyzer

4.3 Board Data Extraction: board_info.py

`BoardBaselines` is a thin wrapper around the live `pcbnew.Board` object. It is responsible for translating KiCad's internal units (nanometres, with coordinates expressed as `VECTOR2I` integers) into millimetres and organising the data into plain Python dictionaries that the rest of the pipeline can work with without importing `pcbnew`.

4.3.1 Net Track Data

`_get_nets()` iterates over `board.GetTracks()` and groups track objects by net name. Each entry in the resulting dictionary is a list of `PCB_TRACK` objects belonging to that net. This raw list is kept alongside the aggregated statistics because the checker functions need access to individual track positions when placing markers.

`get_netInfo()` consumes the raw net dictionary and computes two aggregate values per net: the average track width (in mm) and the total routed length (in mm). These are what the current and voltage checkers operate on.

```
1 def get_netInfo(self):
```

```
    nets = self._get_nets()
    netGraph = {}
    for net, tracks in nets.items():
        total_length, total_width = 0, 0
        for t in tracks:
            total_length += t.GetLength()    # nanometres
            total_width  += t.GetWidth()     # nanometres
        netGraph[net]["avg_width"]     = (total_width / len(tracks)) /
            1e6
        netGraph[net]["total_length"]  = total_length / 1e6
    return netGraph
```

Listing 4.1: Net aggregation in board_info.py

4.3.2 Footprint Pad Map

`get_footPrints()` iterates over all footprints on the board and for each one, collects every pad's net name and position in millimetres. The result is a dictionary keyed by reference designator (e.g., `U1`, `C3`) where each value is a list of `((x_mm, y_mm), net)` tuples. Footprints with the placeholder reference `REF**` and pads belonging to unconnected nets are silently skipped, as they have no meaning for power integrity analysis.

This footprint map is the data structure used by the capacitor placement checker.

4.4 Physics Layer: utils.py

All the electrical calculations live in `utils.py` as pure functions with no side effects. Keeping them here means they can be unit-tested independently of KiCad being installed.

4.4.1 Maximum Current Capacity: IPC-2221

The industry standard for relating trace cross-section to current-carrying capacity is the IPC-2221 empirical formula, derived from measured thermal data on FR4 boards:

$$I = k \cdot (\Delta T)^{0.44} \cdot A^{0.725} \tag{4.1}$$

where I is the maximum safe current in amperes, ΔT is the allowable temperature rise above ambient in °C, A is the conductor cross-sectional area in square mils and k is a layer coefficient 0.048 for external (outer) layers and 0.024 for internal layers, reflecting the better thermal dissipation of exposed traces.

The cross-sectional area is:

$$A = w_{\text{mils}} \times t_{\text{mils}} \tag{4.2}$$

where w is the trace width converted from millimetres to mils (1 mm = 39.37 mils) and t is the copper thickness in mils (1 oz/ft^2 of copper = 1.378 mils).

```
def max_current(width_mm, copper_oz=1, dT=10, layer='ext'):
    k = 0.048 if layer == 'ext' else 0.024
    width_mils     = width_mm  * 39.37
    thickness_mils = copper_oz * 1.378
    A = width_mils * thickness_mils
    return k * (dT ** 0.44) * (A ** 0.725)
```

Listing 4.2: IPC-2221 current capacity in utils.py

A default temperature rise of $\Delta T = 10\,^{\circ}\text{C}$ is used, which is a conservative but widely accepted threshold for signal-integrity-sensitive designs. The function returns the maximum current in amperes that the trace can carry without exceeding this thermal budget.

4.4.2 IR Voltage Drop

The resistive voltage drop along a copper trace is calculated from first principles. Copper has resistivity $\rho = 1.72 \times 10^{-8}\ \Omega \cdot \text{m}$. The trace resistance is:

$$R = \frac{\rho \cdot L}{A} \tag{4.3}$$

where L is the total routed length in metres and A is the cross-sectional area in square metres. The voltage drop follows directly from Ohm's law:

$$V_{\text{drop}} = I \cdot R \tag{4.4}$$

```
rho = 1.72e-8  # copper resistivity, Ohm m

def ir_drop(length_mm, width_mm, current_A, copper_mm=0.035):
    length_m    = length_mm / 1000
    width_m     = width_mm  / 1000
    thickness_m = copper_mm / 1000
    A = width_m * thickness_m
    R = rho * length_m / A
    return current_A * R
```

Listing 4.3: IR drop calculation in utils.py

The default copper thickness of 0.035 mm corresponds to 1 oz/ft^2, matching the default copper weight exposed in the GUI. If the user changes the copper weight input, this parameter updates accordingly.

4.4.3 Euclidean Distance

`distance(p1, p2)` is a straightforward two-dimensional Euclidean distance function operating on `(x_mm, y_mm)` tuples. It is used exclusively by the capacitor placement checker.

4.5 Violation Detection: checkers.py

Each of the three checker functions accepts pre-processed board data dictionaries and returns a list of violation dictionaries. Every violation has at minimum a `net`, `type`, `severity` (`"red"` or `"amber"`), `message`, and, where physically meaningful, a `position` in millimetres that is used to place a marker on the PCB canvas.

4.5.1 Check 1: Trace Width vs. Current (Current_violations)

For every net present in `CURRENT_CONFIG` (the user-supplied expected current map), the checker compares the net's average trace width against the maximum current that width can safely carry, computed via `max_current()`.

If `safe_current < required_current`, the check fails. The position attached to the violation is the midpoint of the *thinnest* track segment on that net the exact location where the thermal risk is highest giving the designer an immediate visual anchor on the board.

```
for net, data in netGraph.items():
    if net not in CURRENT_CONFIG:
        continue
    required_current = CURRENT_CONFIG[net]
    actual_width     = data["avg_width"]
    safe_current     = max_current(actual_width, COPPER_OZ)

    if safe_current < required_current:
        thinnest = min(nets[net], key=lambda t: t.GetWidth())
        mid      = thinnest.GetStart()
        violations.append({
            "type":               "width",
            "severity":           "red",
            "net":                net,
            "actual_width_mm":    actual_width,
            "safe_current_A":     safe_current,
```

```
17              "required_current_A": required_current,
18              "message": f"{net}: supports {safe_current:.2f}A but needs
                    {required_current}A",
19              "position": (mid.x / 1e6, mid.y / 1e6),
20          })
```

Listing 4.4: Current violation detection in checkers.py

Trace width violations are always classified as `"red"` because the consequence trace heating or, in extreme cases, copper delamination is a physical damage risk rather than a performance degradation.

4.5.2 Check 2: IR Voltage Drop (Voltage_violations)

For nets present in both `CURRENT_CONFIG` and `VOLTAGE_CONFIG`, the checker calls `ir_drop()` using the net's total routed length and its minimum track width (using the minimum rather than the average is deliberately conservative the narrowest segment dominates the resistance). The resulting drop is expressed as a percentage of the nominal supply voltage:

$$\text{drop}\% = \frac{V_{\text{drop}}}{V_{\text{supply}}} \times 100 \tag{4.5}$$

Two thresholds are applied:

- drop% $> 5\%$ → `"red"` (critical): most digital ICs have a $\pm 5\%$ supply tolerance, so a 5% drop at the trace alone consumes the entire budget.
- $3\% <$ drop% $\leq 5\%$ → `"amber"` (warning): the design is borderline and should be reviewed, especially if there is additional drop across connectors or solder joints.

The marker for this violation is placed at the start of the *longest* track segment on the net, because longer segments contribute the most to total resistance and are the most productive targets for widening.

4.5.3 Check 3: Decoupling Capacitor Placement

This check works on the footprint pad map rather than the track data. The logic is:

1. Separate footprints into ICs (references starting with `U`) and capacitors (references starting with `C`).
2. For each IC, find every pad connected to a power net (any net in `POWER_NETS`).
3. For each such power-net pad on an IC, search all capacitor pads for the closest one on the same net.

4. If no capacitor is found on that net at all, flag a `"red"` violation: the IC has no decoupling whatsoever on that supply.

5. If a capacitor is found but its distance exceeds 3 mm, flag a `"red"` violation with the distance reported: the capacitor is too far away to suppress high-frequency transients effectively.

The 3 mm threshold reflects standard practice for digital ICs operating at frequencies above a few megahertz, where the inductance of the PCB trace between the capacitor and the power pin begins to significantly reduce the capacitor's effectiveness. For slower or purely analog designs this threshold is more lenient, but 3 mm is a reasonable default for a general-purpose check.

```
for ref, pads in ics.items():
    for net, positions in net_positions.items():
        nearest, nearest_cref = float("inf"), None
        for cref, cpads in caps.items():
            for (cpos, cnet) in cpads:
                if cnet == net:
                    d = distance(positions[0], cpos)
                    if d < nearest:
                        nearest, nearest_cref = d, cref

        if nearest == float("inf"):
            violations.append({"severity": "red",
                "message": f"{ref}: no capacitor on {net}", ...})
        elif nearest > 3:
            violations.append({"severity": "red",
                "message": f"{ref}: nearest cap {nearest_cref} at {
                    nearest:.2f}mm", ...})
```

Listing 4.5: Capacitor placement check in checkers.py (core loop)

4.6 User Interface: PIADialog

The plugin presents results through a `wx.Dialog` rather than directly modifying the board which keeps the interaction non-destructive the user can look at the results, dismiss the dialog, make layout changes, and rerun the analysis as many times as needed.

4.6.1 Input Panel

A four-field grid at the top of the dialog collects the parameters that vary by design:

- **Copper Weight (oz)** = defaults to 1, covers the vast majority of standard PCB fabrication options.
- **Power Net Name** = the net to analyse; the user types the exact net name as it appears in KiCad (e.g., `+9V`, `VCC`).
- **Supply Voltage (V)** = used to compute the drop percentage threshold.
- **Expected Current (A)** = the load current the net is expected to carry.

All four inputs are parsed from string to float when the Run button is clicked. Invalid inputs produce a `wx.MessageBox` error rather than a crash.

4.6.2 Results Display

The `wx.ListCtrl` below the input fields shows one row per violation with four columns: Type, Severity, Net, and Message. Rows are coloured red (`RGB 255, 220, 220`) for critical violations and amber (`RGB 255, 240, 180`) for warnings, giving an immediate visual triage even before reading the message text.

A summary label below the list shows the total count of issues broken down by severity level.

4.6.3 Visual Markers on the PCB Canvas

For every violation that carries a `"position"` field, `add_marker()` draws a circle onto the PCB canvas. Critical violations (red) are drawn on the `F_Cu` layer, which is always visible and will appear in Gerber output as a reminder. Warning violations (amber) are drawn on `B_Cu`. Each circle has a radius of 1 mm and a line thickness of 0.2 mm, which is large enough to be visible at typical zoom levels without obscuring the underlying routing.

`pcbnew.Refresh()` is called after all markers are added so they appear immediately without requiring the user to scroll or zoom. The *Clear Markers* button calls `clear_markers()`, which removes every marker object from the board and calls `Refresh()` again. This means the markers exist only in the live session they are not saved to the `.kicad_pcb` file unless the user explicitly saves after a run, which is intentional.

```
def add_marker(self, board, position_mm, severity="red"):
    x_nm = int(position_mm[0] * 1e6)
    y_nm = int(position_mm[1] * 1e6)
    seg = pcbnew.PCB_SHAPE(board)
    seg.SetShape(pcbnew.SHAPE_T_CIRCLE)
    seg.SetCenter(pcbnew.VECTOR2I(x_nm, y_nm))
    seg.SetEnd(pcbnew.VECTOR2I(x_nm + int(1e6), y_nm))  # 1 mm radius
```

```
    seg.SetWidth(int(0.2e6))
    seg.SetFilled(False)
    seg.SetLayer(pcbnew.F_Cu if severity == "red" else pcbnew.B_Cu)
    board.Add(seg)
    self.markers.append(seg)
```

Listing 4.6: Marker drawing in Power_integrity_main.py

4.6.4 HTML Report

After each run, `generate_report()` in `report.py` writes an HTML file into the same directory as the `.kicad_pcb` file and returns its path. The *Open Report* button (disabled until a run completes) calls `webbrowser.open()` with a `file://` URI, mirroring the same browser-based reporting approach used by eSim-Inspect. The report lists all violations grouped by type, with the board file name and a timestamp in the header.

4.7 Power Integrity Analyzer Results

Figure 4.2 shows the graphical interface of the Power Integrity Analyzer integrated inside KiCad. The plugin highlights detected violations directly on the PCB and displays the corresponding analysis results.

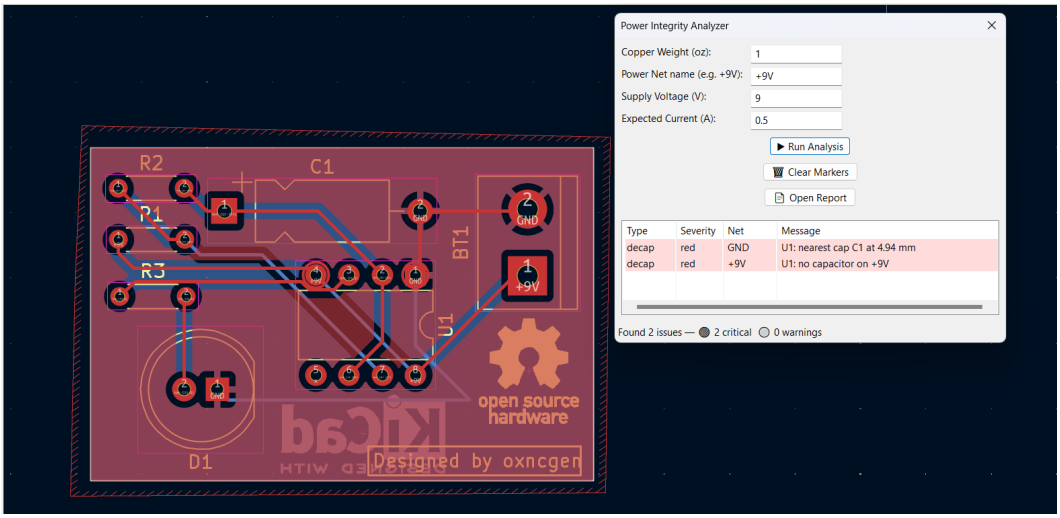


Figure 4.2: Power Integrity Analyzer running inside KiCad

The generated HTML/PDF inspection report for the analyzed PCB is shown below.

Power Integrity Report

Generated: 2026-05-11 22:07:11 | Board: LED Blinker.kicad_pcb

2	0	0
Errors	Warnings	Passed

Net	Check Type	Status	Message
GND	decap	● ERROR	U1: nearest cap C1 at 4.94 mm
+9V	decap	● ERROR	U1: no capacitor on +9V

4.8 Testing and Results

4.8.1 Test Setup

The plugin was tested on two boards:

1. A simple single-supply board (9 V regulator, a handful of resistors and an LED driver IC) used to verify the current and voltage checks on a board where violations could be deliberately introduced by narrowing traces in the layout editor.
2. A small microcontroller board with multiple ICs, decoupling capacitors and a mixed power/signal net structure used to verify the capacitor placement checker with both compliant and non-compliant layouts.

All tests were run on KiCad 9.0 on Ubuntu 25.04 and Windows 11. The dialog, markers, and HTML report were verified to behave identically on both platforms.

4.8.2 Injecting Known Violations

To verify the checkers, violations were injected deliberately:

- For current violations, the `+9V` power trace was narrowed to 0.2 mm in the layout editor with a 0.5 A expected current. The IPC-2221 formula predicts $I_{\max} \approx 0.28$ A at this width, so a violation should be flagged. The plugin correctly reported it.
- For voltage drop, a 200 mm trace at 0.2 mm width carrying 0.5 A was used. The calculated drop is approximately $R = \rho L/A = 1.72 \times 10^{-8} \times 0.2/(0.2 \times 10^{-3} \times 0.035 \times 10^{-3}) \approx 0.049\,\Omega$, giving $V_{\text{drop}} = 0.024$ V, or about 0.27% of 9 V — well below the amber threshold. The trace was then extended to 2000 mm (simulating a very long power distribution trace) to push the drop above 3%, at which point the amber violation appeared as expected.
- For capacitor placement, a decoupling capacitor was moved progressively further from its IC's VCC pin. The red violation appeared at the correct moment when the distance crossed 3 mm.

4.8.3 Results Summary

Table 4.2: Power Integrity Analyzer test results

Check	**Injected Fault**	**Detected**	**Severity**	**Marker Placed**
Trace width	0.2 mm trace for 0.5 A	Yes	Red	Yes (thinnest track)
Voltage drop	Extended trace length	Yes	Amber	Yes (longest track)
Decap missing	No capacitor on VCC	Yes	Red	Yes (IC pad)
Decap far	Cap moved to 4 mm	Yes	Red	Yes (IC pad)
No fault	Compliant layout	No violation reported	—	—

No false positives were observed on the compliant layout. The markers appeared on the correct locations of the PCB canvas on both Linux and Windows immediately after clicking Run, and cleared correctly when the Clear Markers button was used.

Chapter 5

Conclusion and Future Work

5.1 Summary

This internship produced two KiCad Action Plugins that together address problems at two different stages of the eSim design workflow. eSim-Inspect sits at the schematic stage, before the designer has generated a netlist or left KiCad, and catches the kind of connectivity and annotation mistakes that produce cryptic Ngspice errors or silent simulation failures. The Power Integrity Analyzer sits at the PCB layout stage and asks whether the physical copper on the board can actually deliver power reliably, a question that KiCad's built-in DRC does not address.

The two plugins are complementary rather than overlapping. A designer using eSim would naturally run eSim-Inspect while still in the schematic editor, fix whatever it flags, generate the netlist, complete the layout and then run the Power Integrity Analyzer before sending the board for fabrication. Together they close a gap that has existed in the eSim toolchain since its inception.

Both plugins were written to work without any dependencies beyond those already present in KiCad's bundled Python interpreter, and both have been verified to produce identical results on Ubuntu 25.04 and Windows 11 with KiCad 9.0. The modular structure of each plugin separating data extraction, physics computation, violation detection and reporting into distinct files means that individual components can be updated or extended without touching the rest of the pipeline.

5.2 Known Limitations

eSim-Inspect

- **Diagonal wires**: The wire-matching logic handles only horizontal and vertical wire segments. KiCad allows wires at 45 degrees, and these will not be correctly grouped into nets by the current topology builder.
- **Hierarchical schematics**: The parser reads only the top-level `.kicad_sch` file. Components and connections defined in sub-sheets of a hierarchical design will be missed entirely.
- **Bus notation**: KiCad bus entries (grouped multi-bit signals) are not parsed by `get_wires()`, so designs that make heavy use of buses will show spurious floating-pin warnings.
- **Value format validation**: The value check tests only for emptiness or KiCad's default placeholder string. It does not verify that a resistance value is a valid number, that a capacitor value includes units, or that a voltage source value is in a format Ngspice will accept.

Power Integrity Analyzer

- **Manual net configuration**: The user must type the power net name and expected current manually for each run. If the net name in the schematic does not match exactly (including any leading `+` or trailing space), the net will be silently skipped.
- **Single-net analysis per run**: The GUI analyses one user-specified power net per run. Boards with multiple power rails (e.g., 3.3 V and 5 V simultaneously) require multiple separate runs.
- **Uniform copper weight assumption**: The IPC-2221 calculation uses a single copper weight value for the entire board. Boards with different copper weights on different layers are not handled accurately.
- **Capacitor identification by reference prefix**: The decoupling check identifies capacitors by looking for references starting with `C`. Components with non-standard reference prefixes will be missed, and non-decoupling capacitors (e.g., filter capacitors on signal nets) will be included unnecessarily.
- **Markers written to copper layer**: Violation markers are drawn on `F_Cu` or `B_Cu` rather than a dedicated user layer, which means they will appear in Gerber output if the user saves after a run without clearing them first.

5.3 Future Work

1. **Diagonal wire support in eSim-Inspect**: Extending `is_point_on_wire()` to use the general point-to-line-segment distance formula would make the topology builder handle all KiCad wire geometries correctly, eliminating the only known source of false negatives.

2. **Hierarchical schematic parsing**: Recursively loading and merging sub-sheet files using the `sheet` and `sheet_pin` entries in the top-level schematic would make eSim-Inspect applicable to realistic multi-sheet designs used in upper-year courses.

3. **SPICE value format validation**: A regular-expression pass over component values at parse time could catch common mistakes, a resistor written as `"1k ohm"` instead of `"1k"`, or a capacitor written as `"100n"` when Ngspice expects `"100n"` only without a space before they cause simulation parse errors.

4. **Multi-net support in PIA**: Extending the GUI to accept a table of net names, voltages, and currents would allow the Power Integrity Analyzer to check all power rails in one pass, which is far more practical for any board with more than one supply.

5. **Per-layer copper weight in PIA**: Reading the layer stackup from the `.kicad_pcb` file and applying the correct copper weight per layer would make the IPC-2221 calculations accurate for multi-layer boards with different copper weights on inner layers.

6. **Dedicated marker layer**: Drawing violation markers onto `Cmts_User` or another non-copper user layer would avoid any risk of markers appearing in fabrication output, making the tool safer to use during active layout work.

7. **Integration between the two plugins**: A combined entry point that runs eSim-Inspect on the schematic and PIA on the PCB in sequence, then produces a unified report covering both layers of the design, would give the user a single action that covers the complete pre-fabrication checklist.

8. **Contribution to the FOSSEE eSim repository**: Both plugins are ready in principle for upstream contribution. Making them part of the official eSim distribution would ensure they receive ongoing maintenance and reach the full eSim user base without requiring separate installation.

Bibliography

[1] FOSSEE Team, "eSim Project Documentation," IIT Bombay, India, 2023. https://esim.fossee.in

[2] KiCad Development Team, "KiCad EDA Suite — Version 9.0 Documentation," 2024. https://docs.kicad.org

[3] KiCad Developers, "KiCad Python Scripting Reference," 2024. https://docs.kicad.org/doxygen-python/

[4] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring Network Structure, Dynamics, and Function using NetworkX," in *Proc. 7th Python in Science Conference (SciPy2008)*, G. Varoquaux, T. Vaught, J. Millman (Eds.), pp. 11–15, 2008.

[5] T. Ikeda, "sexpdata: S-expression parser for Python," 2012. https://github.com/tkf/sexpdata

[6] IPC, "IPC-2221B: Generic Standard on Printed Board Design," IPC International, 2012.

[7] FOSSEE, "Free and Open Source Software for Education," IIT Bombay, India. https://fossee.in

[8] L. W. Nagel and D. O. Pederson, "SPICE: Simulation Program with Integrated Circuit Emphasis," EECS Department, University of California, Berkeley, Memorandum No. ERL-M382, April 1973.