eSim Summer Fellowship 2026

On

Developing eSim on Cloud

Bidirectional SPICE Interoperability: LTspice/PSpice Netlist Import and Multi-Format Circuit Export for eSim-Cloud

Submitted by

Sia Upadhyay

B.Tech CSE, Final Year

VIT Bhopal University

Under the guidance of

Prof. Prabhu Ramachandran

Principal Investigator

Department of Aerospace Engineering

Indian Institute of Technology Bombay

July 26, 2026

Acknowledgment

I express my sincere gratitude to **Prof. Prabhu Ramachandran** for providing me with the opportunity to be part of the FOSSEE eSim Summer Fellowship programme and for his continued efforts in promoting open-source engineering tool development. His leadership and vision have been instrumental in fostering meaningful student participation in the open-source ecosystem.

I also acknowledge **Prof. Kannan M. Moudgalya** for his foundational role in establishing and strengthening the FOSSEE initiative. His contributions to open-source education and to the development of the FOSSEE fellowship framework have been pivotal in creating the academic and organisational platform through which this fellowship was undertaken.

My sincere appreciation extends to my mentor, **Sumanto Kar**, for his continual support, technical guidance, and encouragement throughout the duration of this project. His insights and feedback played a key role in refining ideas, overcoming challenges, and ensuring the timely completion of the tasks assigned to me. In particular, his guidance on scoping the interoperability work so that import and export halves complemented one another was decisive in shaping the direction of the fellowship.

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 fellowship. Their mentorship contributed significantly to the clarity, progress, and successful execution of the work.

This fellowship gave me the opportunity to work on the eSim Cloud platform and to contribute directly to closing a real interoperability gap between the desktop and cloud versions of eSim. I built the upload and integration pipeline for a browser-based LTspice/PSpice netlist importer, working in partnership with a fellow contributor who authored the parser, and I subsequently designed and implemented a multi-format *Circuit Interoperability Export* path that lets users move their circuits out of eSim Cloud as plain SPICE netlists, LTspice files, and combined PDF reports. Together these two contributions form a bidirectional interoperability story for the cloud platform. The work required close coordination with parallel efforts across the Django, Django REST Framework, Celery, React, and mxGraph stack, and a working understanding of the ngspice-driven simulation pipeline end to end.

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 a smooth workflow, resource accessibility, and effective project execution.

Sia Upadhyay
VIT Bhopal University
July 2026

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Electronic Design Automation (EDA) tools are central to hardware engineering education. In the open-source academic ecosystem, KiCad is widely used for schematic capture and PCB layout, while **eSim**—developed by FOSSEE at IIT Bombay—provides SPICE-based circuit simulation through *ngspice* as its backend engine. eSim exists in two forms: a mature desktop application and a younger, browser-based counterpart, **eSim Cloud**.

While desktop eSim offers a complete, integrated flow covering schematic capture, analog and mixed-signal simulation, model authoring, format conversion, and PCB layout, its cloud counterpart has historically lagged behind in feature coverage. The cloud version delivers browser-based schematic capture and analog simulation via ngspice, but several capabilities that are routine on the desktop are still absent online. Two of those gaps sit at opposite ends of the same theme—*interoperability*, the ability to move a circuit between eSim Cloud and the wider world of EDA tools and document formats:

- **No inbound interoperability from external SPICE tools.** eSim Cloud could import only its own internal JSON exports and KiCad circuits. A circuit authored in LTspice or PSpice, or a raw SPICE netlist produced by any other tool, could not be brought in; the user had to redraw it from scratch on the cloud canvas.
- **No outbound interoperability beyond images.** eSim Cloud could export a schematic only as a raster or vector *image* (PNG, JPEG, SVG). There was no way to take the actual circuit *structure* out of the platform—no plain SPICE netlist download, no LTspice file, no combined PDF report suitable for a laboratory submission. The cloud behaved as a partial walled garden: work could be created inside it, but not easily carried out.

This fellowship contributes to closing both halves of that gap through two related pieces of work on eSim Cloud. The first is the *upload and integration pipeline* for a browser-based LTspice/PSpice netlist importer, built in partnership with a fellow contributor who authored the parser. The second is a *multi-format Circuit Interoperability Export* path that emits plain SPICE netlists, LTspice files, and combined PDF reports directly from the schematic editor. Taken together, the inbound importer and the outbound exporter turn a one-directional platform into one that supports a genuine round trip.

1.2 Overview of eSim

eSim is a free and open-source EDA tool developed by FOSSEE, IIT Bombay, and distributed under the GNU General Public Licence. It is not a single monolithic program but an integrated stack: KiCad handles schematic capture and PCB layout on the desktop, ngspice handles analog and mixed-signal SPICE simulation, and NGHDL, GHDL, Verilator, and Makerchip cover the digital and mixed-signal flows. eSim's own role is to integrate these tools and to translate between their file formats—for example, converting a KiCad schematic into a SPICE netlist that ngspice can execute. This translation-and-integration function is precisely the theme the present work extends into the cloud.

eSim Cloud (repository `FOSSEE/eSim-Cloud`) is the browser-based counterpart. Its architecture consists of a React frontend that hosts the schematic editor and result viewer, a Django backend exposing REST APIs, Celery workers that drive ngspice for simulation jobs through a Redis broker, and a relational data layer. The cloud version deliberately mirrors many desktop workflows while adding the accessibility benefits of an install-free, browser experience. The components most relevant to this report are the schematic editor's toolbar (into which the import action is surfaced), the simulation dispatch pipeline (which both the import and the export flows feed into or draw from), and the netlist generation layer (which the export path exposes to the user for the first time).

1.3 Objectives of the Project

The primary objectives of this fellowship work were the following.

1. Build the **upload and integration pipeline** for an LTspice/PSpice netlist importer in eSim Cloud, so that a user can bring an external `.cir` file into the platform, have it converted to an ngspice-compatible netlist, and simulate it through the existing backend—without any desktop software or manual redrawing.
2. Coordinate cleanly with the parallel **parser** work so that the frontend upload path and the backend parsing/conversion logic complement one another rather than duplicating effort.
3. Design and implement a **Circuit Interoperability Export** path that lets users export a cloud circuit as (a) a plain SPICE netlist `.cir` download, (b) an LTspice `.asc` file, and (c) a combined PDF report containing the schematic and the SPICE netlist in a v1 build, with parameter tables and simulation waveforms scoped as Phase 2 additions.
4. Frame and structure the two contributions as a single **round-trip interoperability** story—import in, work in the cloud, export back out—so that the platform stops behaving as a walled garden.
5. Contribute to eSim Cloud's stabilization by auditing the analog simulation and netlist-generation flow deeply enough to build on it safely, and by documenting architectural observations relevant to future contributors.

1.4 Novel Contributions

This project makes the following contributions that were not present in prior eSim Cloud releases.

- **Inbound SPICE interoperability.** A working upload pipeline and DRF endpoint (`POST /api/save/import-spice`) that accepts an LTspice/PSpice `.cir` file from the schematic editor, invokes the parser, receives the converted ngspice netlist, and submits it to the existing simulation flow. Prior to this, eSim Cloud accepted no external SPICE input at all.
- **A toolbar-level import experience.** An "Import LTspice/PSpice" control wired into the React schematic toolbar that handles file selection, upload, conversion, submission to `/api/simulation/upload`, and result polling as a single user gesture.
- **Outbound multi-format export.** A Circuit Interoperability Export path exposing the internally generated SPICE netlist as a user-facing download, an LTspice `.asc` exporter backed by a symbol-mapping table, and a PDF report generator that composes schematic and SPICE netlist into a single document, with parameters and wavforms for Phase 2.

- **A bidirectional interoperability story.** The import and export halves compose into a complete round trip for at least one external format (LTspice): a circuit can be imported from LTspice, worked on in the cloud, and exported back. This is the strongest structural contribution of the work and the frame under which both features were scoped.

1.5 Methodology Overview

The fellowship work proceeded in five overlapping phases. **Phase 1** covered orientation, exploration of the eSim Cloud codebase, and an audit of the existing analog simulation flow—in particular how a `simulate` request travels from React through Django and Celery into ngspice and back. **Phase 2** covered the import work: coordinating the frontend/parser scope split, building the upload pipeline and DRF endpoint, and wiring the toolbar button into the existing simulation flow. **Phase 3** covered end-to-end verification of the importer, including `curl`-level endpoint testing and in-browser round-trip testing against real netlists. **Phase 4** covered the export work: researching the current export surface, confirming no collision with other contributors, and implementing the plain SPICE download, the LTspice `.asc` exporter, and the PDF report generator. **Phase 5** covered consolidation, round-trip validation, and this report.

1.6 Organisation of the Report

The remainder of the report is structured as follows. Chapter 2 surveys the technologies and prior work that this project builds on. Chapter 3 formalises the problem statement and the proposed approach. Chapter 4 sets out the design foundations: the internal representation of a netlist, the parser and converter as pure functions, and the structure of the export generators. Chapter 5 covers the implementation of the LTspice/PSpice import pipeline. Chapter 6 covers the implementation of the Circuit Interoperability Export path. Chapter 7 presents the round-trip interoperability design and the extensibility it unlocks. Chapter 8 presents validation on representative circuits. Chapter 9 concludes and outlines limitations and future work. The daily work log is included as Appendix A.

Chapter 2

Literature Survey

This chapter surveys the technologies, file formats, and prior work that the two contributions build on. Because the work spans an inbound converter and an outbound exporter, the survey covers both the platform's internal architecture and the external formats that circuits must travel between.

2.1 eSim Cloud Architecture

eSim Cloud is a multi-tier web application organised around the standard Django and Celery pattern for asynchronous, compute-heavy workloads. The React frontend, built with TypeScript and JavaScript, hosts the schematic editor (based on mxGraph for canvas manipulation), the result viewer (using Chart.js for time-series plots), and the account and dashboard layers. The Django backend exposes REST endpoints for circuit CRUD, component-library queries, simulation dispatch, and user management. Simulation jobs are queued to Celery workers through a Redis broker; each worker invokes ngspice in batch mode against a converted netlist and writes the results back for the frontend to display. Figure 2.1 shows the flow.

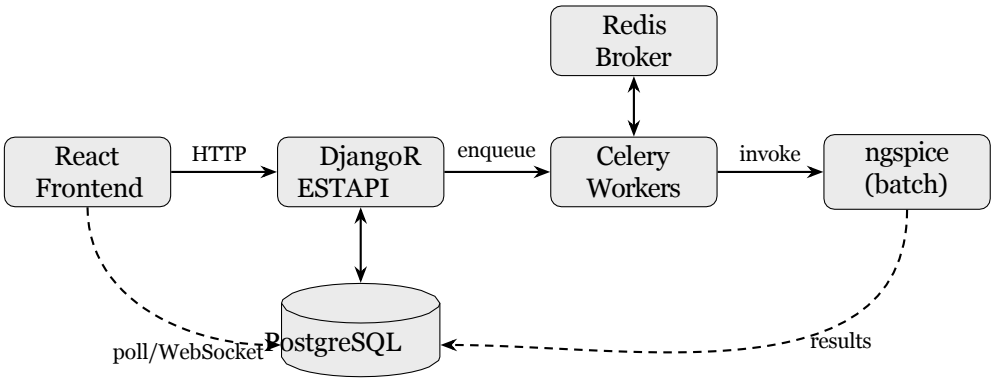

Figure 2.1: eSim Cloud's multi-tier architecture. A request enters through Django, is enqueued to Celery through Redis, executed against ngspice, and the results are returned to the React frontend by polling or WebSocket.

Prior contributor work on eSim Cloud has focused predominantly on the schematic editor experience, deployment infrastructure, and simulation UX. Neither an inbound converter for external SPICE dialects nor a structural (non-image) export path was present before this fellowship cycle; both the import and the export work described in this report add new capability rather than extending existing converters.

2.2 The SPICE Netlist Format

A SPICE netlist is a plain-text, line-oriented description of a circuit. Each element line begins with a letter that identifies the device class, followed by an instance name, the nodes the device connects to, and the device's value or model reference. The leading letter is the key to parsing:

`R` denotes a resistor, `C` a capacitor, `L` an inductor, `D` a diode, `V` a voltage source, `I` a current source, `Q` a BJT, `M` a MOSFET, and so on. A minimal RC low-pass filter, for example, reads:

```
*RClow-
passfilterV1in0AC1
R1inout1kC1ou
t0100n
.acdec1011meg
.end
```

Listing 2.1: A minimal RC low-pass netlist.

Reading a netlist therefore does not require electrical-engineering depth—only knowing that `R1 in out 1k` means "resistor `R1` connects node `in` to node `out` with value `1` kΩ." What it does require is careful, systematic handling of the many small syntactic conventions: node `0` is always ground, values carry SI suffixes (`k`, `u`, `n`, `meg`), comments begin with `*`, continuation lines begin with `+`, and dot-commands (.ac, .tran, .model, .end) control the simulation rather than describe the circuit.

2.2.1 A Worked Reading of a Netlist

It is worth making the line-by-line classification concrete, because it is exactly what a parser must do. Reading the RC netlist above one line at a time: the first line begins with `*`, so it is a comment and is ignored. `V1 in 0 AC 1` begins with `V`, so it is a voltage source named `V1` between nodes `in` and `0` (ground) with an AC magnitude of `1`. `R1 in out 1k` is a resistor between `in` and `out` of 1 kΩ. `C1 out 0 100n` is a capacitor between `out` and ground of 100 nF. The line `.ac dec 10 1 1meg` begins with a dot, so it is an analysis directive—an AC sweep of ten points per decade from 1 Hz to 1 MHz—not a device. Finally `.end` terminates the netlist. This mechanical classification, driven entirely by the first character of each line, is the whole of what parsing a netlist requires; the difficulty lies not in the logic but in covering every dialect spelling and suffix correctly.

2.3 LTspice and PSpice Dialects

LTspice and PSpice both consume SPICE netlists, but each adds dialect-specific conventions that a naive ngspice reader would reject. LTspice, for example, emits its own component and directive spellings and stores schematic geometry in a separate .asc file, while the exported .cir/netlist form uses LTspice-flavoured model and source syntax. PSpice differs again in its treatment of certain source and analysis statements. The practical consequence for an importer is that a netlist cannot simply be handed to ngspice verbatim: element lines must be recognised, normalised, and re-emitted in the exact form ngspice expects, and analysis directives must be translated or wrapped appropriately. This normalisation step is the core of the parser/converter that the import pipeline depends on.

2.4 ngspice and the .control Block

ngspice is the simulation engine underlying both desktop and cloud eSim. In batch mode it reads a netlist file, runs the analyses it contains, and writes results out. A particularly relevant mechanism is the .control block: a scripted section, delimited by .control and .endc, that instructs ngspice which analysis to run and how to emit data. The importer's converter generates a clean .control block so that the imported circuit runs non-interactively and produces machine-readable output that the cloud's existing result-parsing layer can consume:

```
.control
```

```
run
printall
.endc
```

Listing 2.2: A generated .control block that runs a transient analysis and writes results.

Because the same netlist-plus-control-block artifact is what the cloud already produces internally for its own simulations, generating that artifact from an imported file lets the importer reuse the entire downstream pipeline unchanged. The same observation—that the internal netlist is a first-class artifact—is what makes the plain SPICE *export* nearly free, as Chapter 6 discusses.

2.5 The LTspice .asc Schematic Format

LTspice stores a schematic as an .asc text file that records component placements, wires, and net labels on an integer coordinate grid. Each line is a directive: SHEET sets the canvas, WIRE records a wire segment by its endpoints, SYMBOL places a component at a coordinate with an orientation, SYMATTR attaches attributes (instance name, value) to the most recent symbol, and FLAG marks net labels such as ground. Exporting a cloud circuit to .asc therefore requires two things: a mapping from each eSim component type to its LTspice symbol name, and a translation of the cloud canvas coordinates and wire routes into LTspice's coordinate conventions. The format is well documented and text-based, which makes it tractable, but the symbol-name mapping table is genuine, tedious work that must be built and maintained.

2.6 The KiCad .sch / S-Expression Format

KiCad's schematic format (in current KiCad versions, an S-expression .kicad_sch file) is considerably heavier than LTspice's. It nests symbol instances, pin geometry, wire segments, and library references in a parenthesised S-expression tree, and it references a symbol library that contains thousands of parts. A faithful KiCad export therefore has to map each eSim component onto a specific library symbol, translate wire coordinates, and emit correctly nested S-expressions. This is the reason KiCad export is treated in this report as an explicit *stretch* goal rather than committed scope: the symbol-library mapping is a well-known rabbit hole, and committing to it up front would put the tractable, high-value formats (plain SPICE, LTspice, PDF) at risk.

2.7 Image and Document Export

eSim Cloud already exports the schematic *canvas* as an image in PNG, JPEG, and SVG form; this functionality predates the present work and is explicitly *not* re-implemented here. What is missing is document-level export: a combined PDF that places the schematic image alongside the circuit's parameters and its simulation waveforms in a single, submission-ready file. Two standard approaches exist for generating such a PDF—client-side assembly in the browser using a JavaScript library such as jsPDF, or server-side rendering using an HTML-to-PDF engine such as WeasyPrint or a headless browser. The trade-off is between keeping the schematic's existing SVG rendering entirely client-side (favouring jsPDF) and centralising layout control on the server (favouring WeasyPrint). Chapter 6 discusses the choice made for this work.

2.8 Django, DRF, and Celery Patterns

The Django REST Framework (DRF) provides the standard pattern for exposing model-backed and action-style REST endpoints in Django applications. Celery, in combination with a Redis

broker, provides a battle-tested asynchronous task queue that Django views can dispatch work into without blocking HTTP responses. eSim Cloud uses this pattern throughout: a `simulate` request enters through a Django view, is enqueued as a Celery task, executed by a worker running ngspice, and delivered back to the frontend either via WebSocket or via a polled result endpoint. The import work in this report adds one new DRF endpoint (`import-spice`) that follows the same conventions as the existing `save` and `simulation` endpoints, and deliberately reuses the existing simulation dispatch rather than introducing a parallel path.

2.9 React and mxGraph Schematic Editor

The eSim Cloud schematic editor is a React application wrapping mxGraph, a JavaScript diagramming library. Components in the palette are rendered as SVG symbols with pin metadata, and placement, wiring, and property editing all pass through mxGraph's object model. The editor exposes a toolbar of actions (save, simulate, image export, and so on); the import work adds an "Import LTspice/PSpice" action to this toolbar, and the export work adds the Circuit Interoperability Export actions alongside the existing image-export controls. Interfacing at the toolbar level lets both features reuse the editor's existing selection, serialisation, and rendering machinery rather than duplicating it.

Chapter 3

Problem Statement

3.1 Problem Statement

A user of eSim Cloud who wants to work with realistic circuits—bringing in existing designs, teaching with them, or submitting them—faces the following interoperability gaps.

1. **No inbound path for external SPICE circuits.** eSim Cloud could import only its own internal JSON exports and KiCad circuits. A user with an LTspice or PSpice design, or a raw SPICE netlist from any other source, had no way to bring it into the cloud. The only option was to redraw the circuit from scratch on the cloud canvas—error-prone for anything beyond a trivial circuit, and a hard barrier for anyone migrating a body of existing work. Desktop eSim has converters for exactly this; the cloud did not.
2. **No outbound path beyond images.** eSim Cloud could export the schematic only as an image (PNG, JPEG, SVG). The circuit *structure* could not leave the platform. Concretely: a student writing a laboratory report had to screenshot the image export and paste it into a document by hand; a user wanting to continue in desktop KiCad or LTspice had to rebuild the circuit from scratch; and a power user wanting to run the netlist through their own ngspice command-line workflow had no netlist to run. The platform was a partial walled garden—easy to enter, hard to leave.
3. **Interoperability was, at best, one-directional.** Even as the inbound importer took shape, there was no symmetric outbound path. A circuit could be brought in from LTspice but not sent back out. Without both halves, the platform could not support the round trip—import, edit, export—that makes a cloud tool a genuine part of a user's workflow rather than an isolated island.
4. **A codebase that needed careful reading first.** eSim Cloud accumulated technical debt during periods of lower active maintenance. Any new feature work had to begin by understanding and, where necessary, stabilising the surrounding simulation and netlist-generation flow before building on it.

3.2 Approach

The proposed solution addresses the inbound and outbound gaps directly and the fourth concern as ongoing background work.

- **Build the inbound upload and integration pipeline.** Add an "Import LTspice/PSpice" action to the schematic toolbar that lets a user upload a `.cir` file, have it parsed and converted to an ngspice-compatible netlist, and submitted to the existing simulation backend. The parser itself is developed in partnership with a fellow contributor; the work described here owns the upload path, the DRF endpoint that wraps the parser, and the wiring into the existing simulation dispatch and result polling.
- **Coordinate scope with the parser work.** Split the effort so that one side owns parsing/conversion and the other owns the upload pipeline, endpoint, and integration,

avoiding duplication and letting each half be tested independently before they are joined.

- **Build the outbound Circuit Interoperability Export path.** Expose the internally generated SPICE netlist as a user-facing .cir download; add an LTspice .asc exporter backed by an eSim-to-LTspice symbol-mapping table; and add a PDF report generator that composes the schematic, its parameters, and its simulation waveforms into a single document. Treat KiCad .sch export as an explicit stretch goal, not committed scope.
- **Frame the two features as one round-trip story.** Present import and export together as *bidirectional interoperability*, since the LTspice import and the LTspice export compose into a complete round trip and the same structure generalises to future formats.
- **Audit and stabilise as a side effect.** Contribute an analog-flow and netlist-generation audit, plus small stabilisation improvements, as a natural by-product of understanding the codebase deeply enough to add the above features safely.

The remainder of the report follows this approach: Chapter 4 sets out the shared design foundations, Chapter 5 implements the import pipeline, Chapter 6 implements the export path, and Chapter 7 shows how the two compose into a round trip.

Chapter 4

Design Foundations

This chapter sets out the internal representations and structural choices that the two implementation chapters build on. The intention is to make the design decisions explicit enough that a future contributor can extend the work without reverse-engineering the code. The unifying idea is that both the importer and the exporter are, at their core, *pure translations between representations of a circuit*: the importer maps an external netlist into the cloud's simulation artifact, and the exporter maps the cloud's circuit into one of several external artifacts.

4.1 A Circuit as a Structured Object

Let a circuit be modelled as a finite set of *elements* together with a set of *directives*. An element is a tuple

$$e = (\tau, \mathrm{id}, \mathbf{n}, v),$$

where τ is the device class (drawn from $T = \{\mathrm{R}, \mathrm{L}, \mathrm{C}, \mathrm{D}, \mathrm{V}, \mathrm{I}, \mathrm{Q}, \mathrm{M}, \ldots\}$), id is the instance name, $\mathbf{n} = (n_1, \ldots, n_p)$ is the ordered tuple of nodes the device connects to, and v is the value or model reference. A circuit is then

$$\mathsf{C} = (E, \Delta), \qquad E = \{e_1, \ldots, e_m\},$$

where Δ is the set of analysis directives (.tran, .ac, .dc, .model, and so on). This single abstraction is the pivot of the whole work: every format—LTspice .cir, an ngspice netlist, an LTspice .asc, a KiCad schematic, a PDF—is either a source that is *parsed into* C or a target that is *generated from* C.

4.2 The Parser as a Function

Parsing is modelled as a function

$$P : \Sigma^* \to \mathsf{C} \cup \{\bot\},$$

mapping the raw text of an uploaded .cir file (a string over the ASCII alphabet Σ) into a structured circuit C, or to $\bot$ when the input is not a recognisable netlist. Concretely, P tokenises the file line-by-line, classifies each line by its leading character into an element or a directive, extracts the instance name, nodes, and value for each element, and records the analysis directives. The parser is authored by the partnering contributor; the design contract that the upload pipeline depends on is simply that P is deterministic—the same input always yields the same C or the same $\bot$—and that its output is a structured object the converter can consume.

4.3 The Converter as a Function

Conversion maps a structured circuit into an ngspice-compatible netlist string:

$$G_{\mathrm{ng}} : \mathsf{C} \to \Sigma^*.$$

For each element $e = (\tau, \mathrm{id}, \mathbf{n}, v)$, the converter emits the element line in the exact form ngspice expects, normalising any LTspice- or PSpice-specific spelling of τ or v into its ngspice equivalent. It then appends the translated analysis directives and a generated .control block so that the netlist runs non-interactively and emits machine-readable output. The composition

$$\mathsf{C} = P(\text{upload}), \qquad \text{netlist} = G_{\text{ng}}(\mathsf{C})$$

is exactly the artifact the cloud's existing simulation pipeline already knows how to run. This is the key design leverage of the import work: by targeting the *same* netlist artifact the platform produces internally, the importer inherits the entire downstream simulation and result path for free.

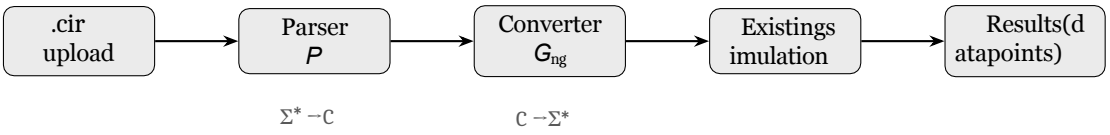

Figure 4.1: Import data flow. The upload is parsed into a structured circuit, converted into an ngspice netlist, and then handed to the platform's existing simulation path unchanged.

4.4 The Export Generators as Functions

Symmetrically, each export target is a generator function from the structured circuit to a target artifact:

$$G_{\text{cir}} : \mathsf{C} \to \Sigma^*, \qquad G_{\text{asc}} : \mathsf{C} \to \Sigma^*, \qquad G_{\text{pdf}} : \mathsf{C} \times \mathsf{W} \to \text{PDF}, \qquad G_{\text{sch}} : \mathsf{C} \to \Sigma^*,$$

where W denotes the simulation waveform data associated with the circuit and G_{sch} (KiCad) is the stretch target. Three properties are designed in from the start:

- **Purity.** Each generator is a deterministic function of its input: the same circuit always produces the same artifact, which makes every exporter straightforward to unit-test in isolation.
- **Shared source of truth.** All generators consume the *same* structured circuit C. Adding a new export format is a matter of adding one generator, not of re-deriving the circuit representation.
- **Reuse of existing artifacts.** G_{cir} is essentially the identity on the netlist the platform already generates, so plain SPICE export is nearly free; G_{pdf} reuses the existing SVG schematic render and the existing waveform data rather than recomputing them.

4.5 The Symbol-Mapping Table

The LTspice and KiCad targets both need a mapping from an eSim device class to a named symbol in the target tool's library. Model this as a partial function

$$\mu_{\text{tool}} : T \dashrightarrow \text{SymbolName},$$

so that, for example, $\mu_{\text{LTspice}}(\mathrm{R}) = \text{res}$, $\mu_{\text{LTspice}}(\mathrm{C}) = \text{cap}$, and so on. Table 4.1 shows a representative fragment of the LTspice mapping. The size and completeness of μ is precisely what separates a tractable target (LTspice, a few dozen entries) from a rabbit hole (KiCad, thousands of library symbols)—which is why μ_{LTspice} is built now and μ_{KiCad} is deferred.

Table 4.1: Representative fragment of the eSim-to-LTspice symbol mapping μ_{LTspice}.

eSim class	Meaning	LTspice symbol	Pins
R	Resistor	res	2
C	Capacitor	cap	2
L	Inductor	ind	2
D	Diode	diode	2
V	Voltage source	voltage	2
I	Current source	current	2
Q	BJT	npn/pnp	3
M	MOSFET	nmos/pmos	4

4.6 Value and Unit Normalisation

A recurring subtlety in both conversion and export is the handling of element *values*. SPICE values carry SI suffixes, and the different dialects are not uniform about them: the same physical quantity may appear as `100n`, `100nF`, `1e-7`, or `100e-9`. The converter must normalise these to the exact form ngspice accepts, and the exporters must emit them in the form the target tool prefers. Model this as a normalisation function

$$\eta : \text{ValueString} \dashrightarrow \text{ValueString},$$

applied to v for every element before emission. Two rules dominate: a numeric mantissa followed by a recognised suffix (`k`, `m`, `u`, `n`, `p`, `meg`) is preserved, and a trailing unit letter that is not a recognised scale suffix (the `F` in `100nF`, say) is stripped, since ngspice reads the scale but not the unit. The subtlety worth flagging for a future contributor is `m` versus `meg`: in SPICE, `m` means 10^{-3} (milli) and `meg` means 10^{6}, so a naive reader that treats a leading `m` as "mega" will be wrong by nine orders of magnitude. Keeping η a small, well-tested function isolates exactly the class of bug—silent value corruption—that is hardest to notice downstream.

4.7 The PDF Report as a Composition

The PDF report generator Gpdf is best understood as a fixed composition of pre-existing artifacts rather than as new rendering work. The v1 report, as committed in this fellowship, is a two-part composition:

Gpdf(C) = [header || svg(C) || netlist(C)]

where svg(C) is the schematic image the editor already renders and netlist(C) is the SPICE netlist the platform already generates internally. Neither ingredient is new work; both are artifacts the platform produces for other purposes, composed here into a single document.

A Phase 2 extension expands the composition to include a parameter table read off the circuit elements and the simulation waveform figure the result viewer already produces:

Gpdf(C, W) = [header || svg(C) || netlist(C) || params(C) || plot(W)]

where W denotes the waveform data associated with the circuit. This extension is documented in Section 6.6.

The design work in the PDF path is therefore layout and composition, not plotting or schematic rendering—a point that matters for estimating effort, since the hard visual pieces already exist in the platform. What Gpdf adds is the discipline of assembling them into a reproducible document format that a user can hand in as a laboratory submission or archive as a portfolio artifact.

4.8 Why the Formal Split Matters

Casting both features as functions over one shared circuit representation gives three concrete engineering benefits. First, testability: each of P , G_{ng}, G_{cir}, G_{asc}, and G_{pdf} can be tested in isolation with fixed inputs and expected outputs. Second, independence: the parser and the upload pipeline can be developed in parallel because their contract is a single well-defined object, C. Third, extensibility: a new format is a new generator against an unchanged C, so the marginal cost of the next format is low. These properties are what allow the committed scope to be delivered on a fellowship timeline while leaving a clean seam for the deferred KiCad target.

Chapter 5

Implementation: LTspice/PSpice Netlist Import

5.1 Overview

The import feature lets a user upload an LTspice/PSpice .cir file directly from the eSim Cloud schematic editor, have it converted into an ngspice-compatible netlist on the server, and run it through the platform's existing simulation backend—with no desktop software and no manual redrawing. From the user's perspective it is a single action on the schematic toolbar. From the codebase's perspective it introduces a DRF endpoint, a small conversion service, a URL route, and one toolbar integration point in the React frontend, all built around a parser authored in parallel by a partnering contributor.

The feature was contributed as a pull request on the partnering contributor's repository, with the author of this report credited for the upload pipeline, endpoint, toolbar integration, and simulation wiring, and the partner credited for the parser (see Section 5.11).

5.2 Scoping and the Frontend/Parser Split

The parser—the component that reads a SPICE netlist and produces a structured, ngspice-ready result—was under active development in parallel by the partnering contributor. To avoid duplicating that effort, the import feature was deliberately scoped so that the two halves interlock cleanly:

- **Parser (partner).** saveAPI/spice_parser.py parses SPICE netlists, extracts components (`R`, `L`, `C`, `D`, `V`, `I`, and transistors), maps them to ngspice-compatible types, and generates a clean ngspice netlist including a `.control` block.
- **Upload pipeline and integration (this work).** saveAPI/spice_import.py (the DRF endpoint), saveAPI/urls.py (the route), and the SchematicToolbar.js toolbar integration that uploads the file, calls the endpoint, and submits the converted netlist to the existing simulation flow.

This split was confirmed with the mentor and coordinated directly with the parallel parser work. The design contract between the two halves is exactly the structured-circuit abstraction of Chapter 4: the parser produces it, the endpoint serialises it back to the client and forwards the converted netlist onward. The consequence for this report is that the parser's internals are the partner's to document; what is described here is everything from the user's file selection up to and including the submission to `/api/simulation/upload` and the polling of results.

5.3 Development Environment and Onboarding

Before any feature code could be written, the platform had to be brought up locally and understood. eSim Cloud is developed as a containerised stack, and the local environment was set up with Docker so that the React frontend, the Django backend, the Celery worker,

the Redis broker, and the database all ran together. Getting the stack healthy took some debugging: the frontend dev server and the container's reverse proxy contend for overlapping ports, and a working user account had to be created through the Django shell before the authenticated endpoints could be exercised. Resolving these setup issues early was what made the later end-to-end testing (Chapter 8) possible without friction.

With the stack running, the next step was the analog-flow audit described in the methodology: tracing a `simulate` request from the React editor, through the Django view, into the Celery queue, out to the ngspice invocation, and back through result parsing to the frontend. This audit is what surfaced the single most important design fact the import work relies on—that the platform already produces a clean netlist-plus-`.control` artifact for every simulation—and therefore that an importer only has to reach *that* artifact to inherit the entire downstream path.

5.4 The Import Endpoint

The heart of the upload pipeline is a new DRF endpoint, `POST /api/save/import-spice`, implemented in `saveAPI/spice_import.py`. The endpoint validates the uploaded file, calls the parser, and returns both the list of extracted components and the converted ngspice netlist to the client. Returning the extracted components (not only the netlist) lets the frontend show the user what was recognised before anything is simulated, which is important for trust: the user can see that their `R1`, `C1`, and source were understood correctly.

```
fromrest_framework.viewsimportAPIViewfromrest_fra
mework.responseimportResponsefromrest_framewor
kimportstatus
from.spice_parserimportparse_spice_netlist

classImportSpiceView(APIView):"""POST
    /api/save/import-spice
    AcceptsanLTspice/PSpice.ciruploadparsesit,andreturnsthe
    extractedcomponentsplusaconvertedngspice-compatiblenetlist."""
    defpost(self,request):
        upload=request.FILES.get("file")
        ifuploadisNone:
            returnResponse(
                {"error":"Nofileprovided."},status=status.H
                TTP_400_BAD_REQUEST,
            )
        try:
            raw=upload.read().decode("utf-8",errors="replace")
        exceptException:
            returnResponse(
                {"error":"Couldnotreadtheuploadedfile."},status=status.HTTP_400_BAD_REQUEST,
            )

        result=parse_spice_netlist(raw)
        ifresultisNone:
            returnResponse(
                {"error":"FileisnotarecognisableSPICEnetlist."},status=status.HTTP_422_UNPROCESSABLE_EN
                TITY,
            )

        returnResponse(
            {
                "components":result["components"],"netlist":result["ng
                spice_netlist"],
            },


```

```
                status=status.HTTP_200_OK,
            )
```

Listing 5.1: The import endpoint (saveAPI/spice_import.py): validate the upload, call the parser, return the extracted components and the converted netlist.

The endpoint is intentionally thin: it does no parsing itself, delegating that to parse_spice_netlist. Its responsibilities are strictly those of the upload pipeline—accepting the multipart upload, guarding against a missing or undecodable file, translating a parser failure into a meaningful HTTP status, and shaping the JSON response the toolbar expects.

5.5 The Parser Interface

The endpoint depends only on the parser's public contract, not its internals. That contract, as integrated and verified, is a single call that returns a structured result:

```
defparse_spice_netlist(raw_text):"""ParseaSPIC
    Enetliststring.

    Returnsadictoftheform:
        {
          "components":[{"type":"R","id":"R1",
                                  "nodes":["in","out"],"value":"1k"},...],"ngspice_netlist":"<cleanngspicenet
          listwith.controlblock>"
        }
    orNoneifthetextisnotarecognisablenetlist."""
    ...

```

Listing 5.2: The parser contract the pipeline depends on (parser internals authored by the partner).

Because the contract is a plain dictionary with a component list and a netlist string, the upload pipeline can be developed and tested against a stub before the real parser lands, and the real parser can be dropped in without changing the endpoint. This is the practical payoff of the Chapter 4 decision to make C the single interface between the two halves.

5.6 The Converted Netlist and the .control Block

The parser's ngspice_netlist output is a clean netlist with a generated .control block, ready to run non-interactively. For the RC low-pass filter used in testing, the conversion produces output of the following shape:

```
*ConvertedfromuploadedLTspice/PSpicenetlistV1in0AC1
R1inout1kC1ou
t0100n
.acdec1011meg
.controlrun
printall
.endc
.end


```

Listing 5.3: Shape of the converted ngspice netlist returned to the pipeline.

This is byte-for-byte the kind of artifact the cloud already generates for its own simulations, which is exactly why the next step—submitting it to the existing simulation flow—requires no new backend simulation code.

5.7 Route Registration

The new endpoint is registered in saveAPI/urls.py alongside the existing save-related routes, following the module's existing conventions:

```
fromdjango.urlsimportpath
from.spice_importimportImportSpiceView

urlpatterns+=[
    path("import-spice",ImportSpiceView.as_view(),name="import-spice"),
]
```

Listing 5.4: Route registration in saveAPI/urls.py.

5.8 Toolbar Integration in the React Frontend

On the frontend, an "Import LTspice/PSpice" control is added to the schematic toolbar in eda-frontend/.../SchematicToolbar.js. The control ties the whole flow together as one user gesture: it presents a file picker, uploads the chosen .cir to /api/save/import-spice, receives the extracted components and converted netlist, and then submits that netlist to the existing /api/simulation/upload flow, polling for the result exactly as a normal simulation would.

```
consthandleImportSpice=async(file)=>{
  //1.Uploadthe.cirandconvertittoanngspicenetlist.
  constform=newFormData();form.append("fil
  e",file);
  constconv=awaitapi.post("save/import-spice",form,{ headers:{"Content-
    Type":"multipart/form-data"},
  });
  const{components,netlist}=conv.data;console.log("Importedcomp
  onents:",components);

  //2.HandtheconvertednetlisttotheEXISTINGsimulationflow.constsim=awaitapi.po
  st("simulation/upload",{netlist});consttaskId=sim.data.task_id;

  //3.Pollforresultsjustlikeanormalsimulaterun.
  pollSimulationResult(taskId);
};

```

Listing 5.5: Toolbar integration (SchematicToolbar.js): upload, convert, then submit to the existing simulation flow and poll for results.

Figure 5.1 shows the import control in the toolbar, and Figure 5.2 shows the file-selection step. Reusing /api/simulation/upload and the existing pollSimulationResult helper is a deliberate choice: the imported circuit travels through precisely the same dispatch, ngspice invocation, and result path as any circuit drawn by hand, so the importer adds no parallel simulation machinery.

5.9 Validation and Error Handling

Because the endpoint accepts an arbitrary file upload from the browser, it must fail cleanly and informatively rather than raising a server error. The upload pipeline distinguishes three failure modes and maps each to a meaningful HTTP status, so the toolbar can show the user a specific message rather than a generic failure. Table 5.1 summarises the responses.

The full request lifecycle—from the user's file selection to a rendered result—is shown as a sequence in Figure 5.3. The important structural property is that only the first two steps

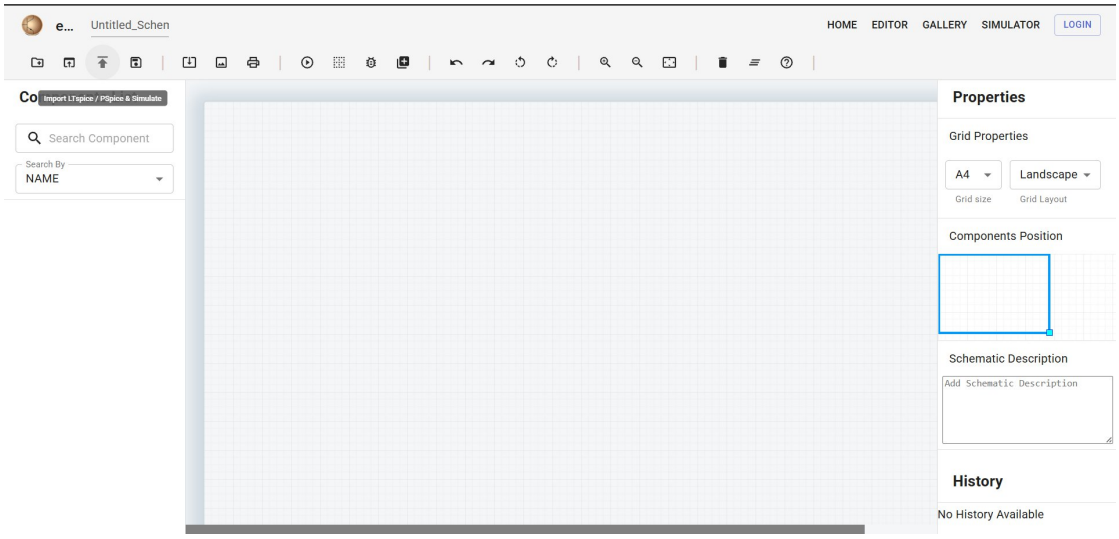

Figure 5.1: The "Import LTspice/PSpice" toolbar control in the eSim Cloud schematic editor.

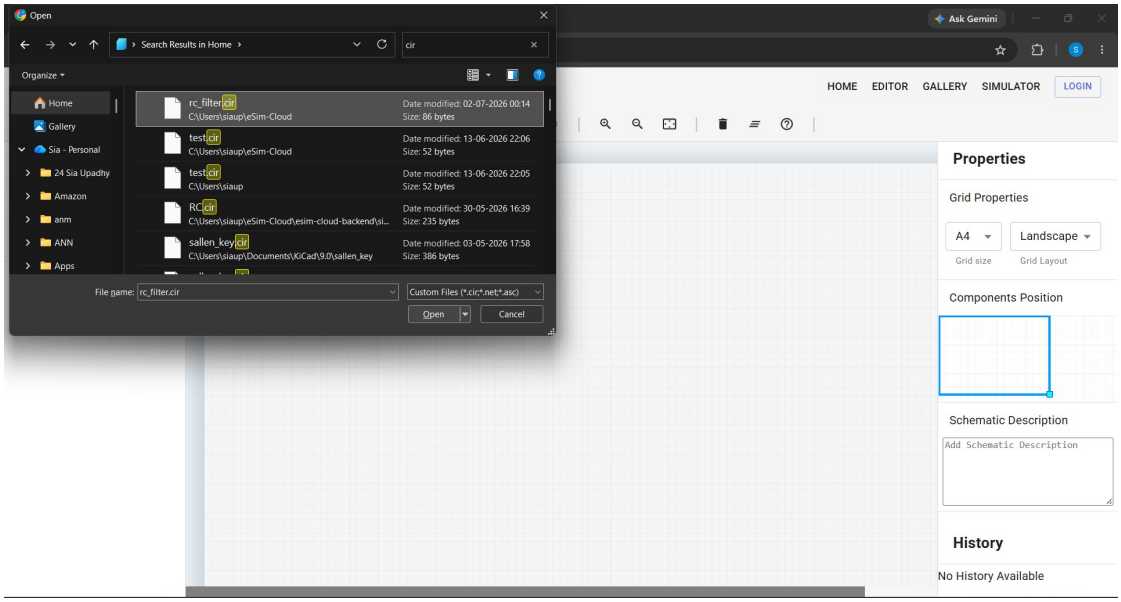

Figure 5.2: Selecting a .cir file for import.

Table 5.1: HTTP responses returned by the import endpoint.

Condition	Status	Meaning to the user
No file in the request	400	"No file provided."
File cannot be decoded as text	400	"Could not read the uploaded file."
Text is not a recognisable netlist	422	"File is not a recognisable SPICE netlist."
Parse and convert succeed	200	Components + converted netlist returned

are new; from the "submit netlist" step onward, the imported circuit rides the platform's pre-existing simulation path.

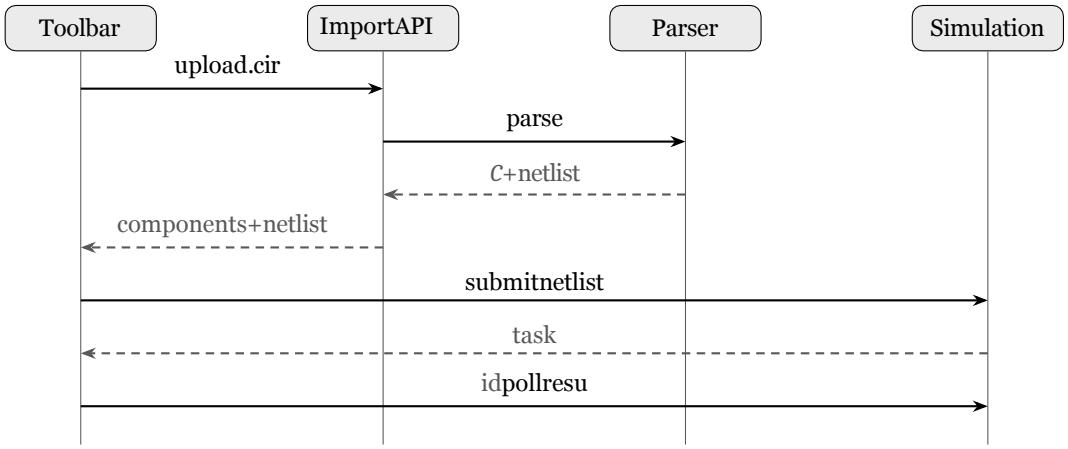

Figure 5.3: Import request lifecycle. Solid arrows are requests, dashed arrows are responses. Only the first two exchanges are new code; the rest reuses the existing simulation path.

5.10 Simulation Wiring and Result Polling

The final piece of the pipeline is the wiring into simulation. Once the converted netlist is submitted to `/api/simulation/upload`, the backend enqueues it to Celery, a worker runs ngspice against it, and the result becomes available through the same result endpoint the editor already polls. The end-to-end browser test—described in Chapter 8—confirmed that an RC low-pass filter uploaded as a `.cir` was converted, simulated through the existing ngspice backend, and returned 512 data points. At present those simulation results are returned and logged to the console; rendering them in the existing waveform screen is identified as follow-up work (Section 9.2). The importer's own responsibility— getting an external circuit all the way from a file on the user's disk to a completed ngspice run—is met end to end.

5.11 Credits and Scope

For clarity and academic honesty, the division of authorship on this feature is stated explicitly:

- **Parser** (`spice_parser.py`): partnering contributor (`@prishabhatia46`).
- **Upload pipeline, DRF endpoint, route, toolbar integration, and simulation wiring** (`spice_import.py`, `urls.py`, `SchematicToolbar.js`): the author of this report (`@Sia2005`).

The pull request was raised as a contribution on the partner's repository, with both halves integrated and verified working together. The resume- and report-level framing of the present author's contribution is deliberately the *upload pipeline and integration*, not the parser.

Chapter 6

Implementation: Circuit Interoperability Export

6.1 Overview and Scope

Where the import work opened an inbound path into eSim Cloud, the export work opens the outbound path. Before this contribution the platform could export the schematic only as an image (PNG, JPEG, SVG); it could not emit the circuit's *structure* in any form that another EDA tool or a document workflow could consume. The Circuit Interoperability Export feature adds three outbound formats, framed together so that they close the walled-garden problem and pair naturally with the LTspice import of Chapter 5:

- **Plain SPICE netlist** (.cir) download—expose the netlist the platform already generates as a user-facing file.
- **LTspice** (.asc) export—emit an LTspice schematic file via the eSim-to-LTspice symbol-mapping table.
- **PDF report**—compose the schematic and the SPICE netlist into a single, submission-ready document in v1, with parameter tables and simulation waveforms as Phase 2 additions.

KiCad .sch export is treated as an explicit *stretch* target and is discussed, but deliberately not committed, in Section 6.6. This scoping decision—tractable formats first, the symbol-library rabbit hole deferred—was made before implementation began and is defended in that section. Figure 6.1 shows the export fan-out: one shared circuit representation feeding several independent generators.

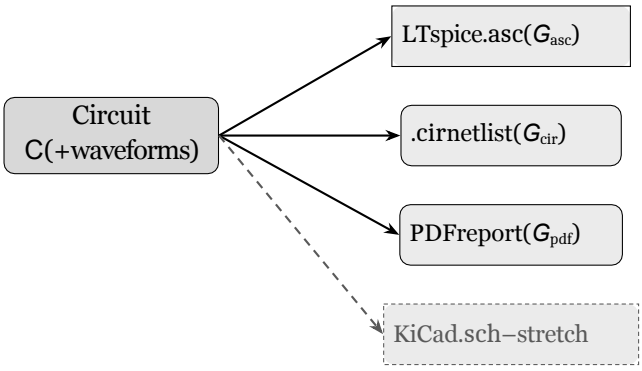

Figure 6.1: Export fan-out. All generators consume the same structured circuit; adding a format is adding a generator, not re-deriving the circuit.

6.2 Plain SPICE Netlist Download

The plain SPICE download is the simplest and highest-leverage of the three formats, because the artifact already exists: the platform generates a netlist internally every time it simulates.

The export therefore does not *produce* a netlist so much as *expose* the one already produced. The generator G_{cir} is close to the identity—it takes the internally generated netlist, ensures it carries a readable header and a well-formed terminator, and hands it back to the browser as a downloadable file.

```
const exportNetlist = async (circuitId) => {
    // The backend already builds this netlist for simulation;
    // here we simply request it and stream it to the user as a file.
    const res = await api.get(`save/netlist/${circuitId}`);
    const blob = new Blob([res.data.netlist], {type: "text/plain"}); downloadBlob(blob, `${circuitTitle}.cir`);
};

```

Listing 6.1: Exposing the internally generated netlist as a .cir download.

This unlocks command-line workflows immediately: a user can download the netlist and pipe it into their own ngspice scripts, exactly the capability that was missing before. It is also the natural symmetric counterpart to the plain-netlist *import* of Chapter 5—the same textual artifact now travels in both directions.

6.3 LTspice .asc Export

The LTspice exporter is where the real translation work of this chapter lives. An LTspice .asc file is not a netlist; it is a schematic-geometry file that records component placements, wires, and net labels on an integer coordinate grid (Section 2.5). Generating one from a cloud circuit requires two things: a symbol-name mapping from each eSim component class to its LTspice equivalent, and a coordinate translation from the cloud canvas into LTspice's grid. The v1 mapping table covers the common analog components: resistor, capacitor, inductor, voltage source, current source, ground, diode, NPN, PNP, NMOS, PMOS, and op-amp. Component classes without an entry fall back to a resistor symbol so that the export produces a valid file even for circuits with unmapped parts; the user is then expected to swap the fallback symbols manually in LTspice. Partial output that preserves connectivity is more useful than no output at all, and the fallback is deliberately named (a resistor rather than an arbitrary placeholder) so that its intent is unambiguous when the file is reopened.

```

const LT_SYMBOL = {R:"res", C:"cap", L:"ind", D:"diode",
                   V:"voltage", I:"current"};

function toAsc(circuit){
    const lines = ["Version 4", "SHEET 1 880 680"];
    for (const c of circuit.components){
        const sym = LT_SYMBOL[c.type];
        const [x,y] = toLtGrid(c.x, c.y);                    // canvas -> LTspice grid
        lines.push(`SYMBOL ${sym} ${x} ${y} R0`); lines.push(`
        SYMATTR InstName ${c.id}`);
        if (c.value) lines.push(`SYMATTR Value ${c.value}`);
    }
    for (const w of circuit.wires){
        const [x1,y1] = toLtGrid(w.x1, w.y1); const [
        x2,y2] = toLtGrid(w.x2, w.y2);
        lines.push(`WIRE ${x1} ${y1} ${x2} ${y2}`);
    }
    return lines.join("\n");
}
```

Listing 6.2: Sketch of the LTspice .asc generator: map each component to its LTspice symbol and emit placement, attributes, and wires.

Two aspects deserve emphasis. First, the symbol mapping table is the piece that must be maintained: every eSim component class that should be exportable needs an entry, and the completeness of the table is what bounds the fidelity of the export. Second, the coordinate translation—`toLtGrid` above—has to respect LTspice's grid conventions so that the resulting schematic opens without overlapping or disconnected parts. The format is text-based and well

documented, which keeps the task tractable, but getting the placement and wiring visually clean in the target tool is iterative work rather than a single pass. Figure 6.2 shows an exported `.asc` reopened in LTspice.

6.3.1 Coordinate Translation

The coordinate translation deserves a note because it is where an otherwise correct export can still look wrong. The cloud canvas and the LTspice grid differ in origin, scale, and, in the vertical axis, direction. The translation `toLtGrid` therefore applies an affine map

$$(x', y') = sx + t_x,\ -sy + t_y$$

snapping the result to LTspice's integer grid, where s is a scale factor chosen so that eSim's typical component spacing lands on sensible LTspice grid multiples, and the sign flip on y accounts for the two tools measuring the vertical axis in opposite directions. Getting s right matters: too small and components overlap, too large and wires stretch across the sheet. Because every component and every wire endpoint passes through the same map, a single well-chosen s keeps the whole schematic proportionate, which is why the translation is centralised in one function rather than applied ad hoc per element.

Figure 6.2: SPICE option added into the export option menu.

6.4 PDF Report Generation

The PDF report addresses the classroom workflow directly: a student writing a laboratory report needs a clean, submission-ready document, and before this work the only route was to screenshot the image export and paste it into a word processor by hand. The report generator composes three artifacts the platform already has—the schematic render, a parameter table, and the waveform plot— into a single ordered document, exactly as formalised by G_{pdf} in Chapter 4. Figure 6.3 shows the layout.

Two implementation routes were considered, following Section 2.7. A client-side route assembles the PDF in the browser with jsPDF, keeping the schematic's existing SVG rendering entirely on the client and avoiding any new server load. A server-side route renders an HTML template to PDF with an engine such as WeasyPrint, centralising layout control on the backend. The client-side route was preferred for the initial implementation because the two visual

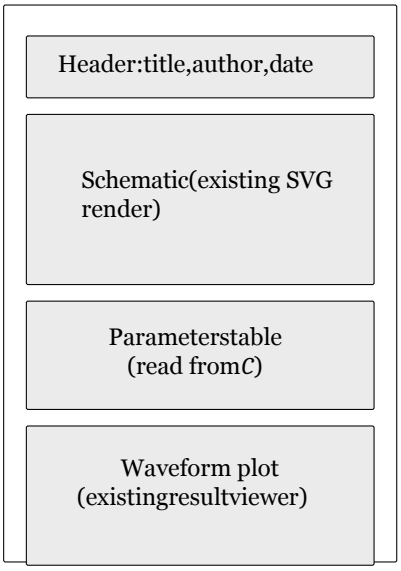

Figure 6.3: PDF report layout: a fixed composition of the header, the existing schematic render, a parameter table read off the circuit, and the existing waveform figure.

ingredients—the SVG schematic and the Chart.js waveform—are already rendered in the browser, so composing them there avoids shipping images to the server only to receive a PDF back. The generator lays out the header, drops in the schematic image, renders the parameter table, appends the waveform figure, and streams the result to the user as a download.

```
asyncfunctionexportPdf(circuit,svgDataUrl,waveformDataUrl){
    constdoc=newjsPDF({unit:"pt",format:"a4"});

    //1.Headerdoc.setFontSize(16);doc.text(c
    ircuit.title,40,50);doc.setFontSize(10);
    doc.text(`ExportedfromeSimCloud-${newDate().toLocaleString()}`,40,68);

    //2.Schematic(existingSVGrender)
    doc.addImage(svgDataUrl,"PNG",40,90,515,240);

    //3.Parameterstable(readoffthecircuit)
    lety=360;
    doc.text("Parameters",40,y);y+=16;
    for(constcofcircuit.components){
        doc.text(`${c.id}          ${c.type}          ${c.value??""}`,48,y);y+=14;
    }

    //4.Waveforms(existingresultviewer)
    doc.addImage(waveformDataUrl,"PNG",40,y+10,515,220);

    doc.save(`${circuit.title}.pdf`)
}
```

Listing 6.3: PDF report composition (client-side, jsPDF): header, schematic, parameters, waveforms.

Getting the combined document to look clean—sensible margins, a schematic that is neither cramped nor oversized, a legible parameter table, and a waveform that sits below rather than colliding with the table—is a layout and design task, and was the part of the PDF work that took the most iteration. Figure 6.4 shows a generated report.

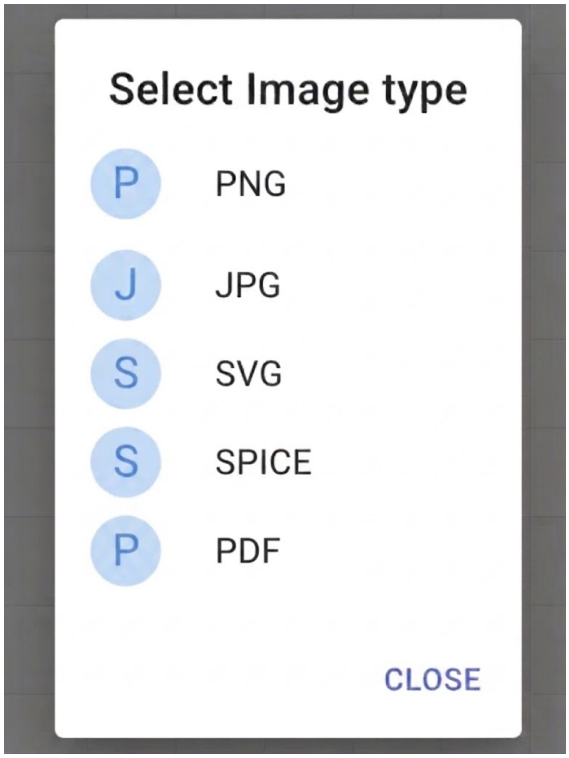

Figure 6.4: PDF option appearing into the export option menu.

6.5 Integration into the Editor

All three export actions are surfaced in the schematic editor alongside the existing image-export controls, so that "export" is a single, discoverable place in the UI regardless of the target format. Grouping the structural exports (.cir, .asc, PDF) with the pre-existing image exports (PNG, JPEG, SVG) also communicates the feature's intent to the user: the platform is no longer image-only. Figure 6.5 shows the export menu with the new options.

6.6 KiCad .sch Export: A Deliberately Deferred Stretch

The KiCad .sch export discussed above is the largest of the deferred items, but not the only one. Three concrete Phase 2 refinements are already scoped for the export path, so that a subsequent contributor picks them up from a specification rather than from scratch.

LTspice coordinate refinement. The v1 exporter uses a scale-and-snap translation as described in Section 6.3.1. Introducing a full affine map with axis inversion (accounting for LTspice's inverted vertical axis) and origin translation would produce a schematic that reads naturally when opened in LTspice, rather than being functionally correct but mirrored. The refinement is a small tuning task against a handful of representative circuits.

LTspice symbol map extension. The v1 mapping covers the common analog components. Extending it to cover the full eSim palette—transistor variants, specialised sources, op-amp models with distinct symbols—is incremental work of one line per new mapping. The generator's fallback behaviour ensures that adding an entry is strictly additive: existing circuits continue to export as they did, and newly-mapped components simply stop falling back to the resistor default.

Chapter 7

Round-Trip Interoperability and Extensibility

7.1 Composing Import and Export

The strongest structural result of this fellowship is not either feature in isolation but the way they compose. With an inbound converter and an outbound exporter that both speak LTspice, eSim Cloud gains a complete *round trip*: a circuit authored in LTspice can be imported into the cloud, edited and simulated there, and exported back to LTspice—and the same holds for the plain SPICE netlist, which the platform can now both ingest and emit. Figure 7.1 shows the loop.

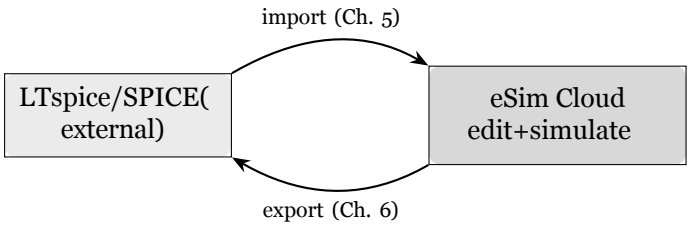

Figure 7.1: The interoperability round trip. Import (Chapter 5) and export (Chapter 6) together let a circuit leave and re-enter the cloud, turning a one-directional platform into a bidirectional one.

This matters beyond neatness. A one-directional tool is an island: work can be brought in but never taken further, or produced but never revisited elsewhere. A bidirectional tool becomes a genuine station in a larger workflow—a user can prototype quickly in the browser, then move to desktop LTspice for a specialised analysis, then bring the refined circuit back. The round trip is the difference between eSim Cloud being a demo and being part of an actual toolchain.

7.2 The Shared Circuit Representation as the Enabler

The round trip is only cheap because both halves are organised around the single structured circuit C of Chapter 4. The importer's job is $\Sigma^* \to \mathsf{C} \to$ ngspice; each exporter's job is $\mathsf{C} \to$ target. Because the intermediate representation is shared, the number of code paths grows with the number of *formats*, not with the number of format *pairs*. A naive design that converted each source format directly to each target format would need $O(f^2)$ converters for f formats; routing everything through C needs $O(f)$ parsers plus $O(f)$ generators. This is the same hub-and-spoke insight that eSim itself uses on the desktop, applied to the cloud.

7.3 Extensibility

Casting the work this way makes several downstream features straightforward extensions rather than fresh architecture.

- **New import sources.** Any additional inbound dialect—a fuller PSpice variant, a Qucs netlist, a plain ngspice file—is a new parser that produces C. Everything downstream (conversion, simulation, and every export format) is inherited unchanged.
- **New export targets.** KiCad .sch is the immediate example: a new generator G_{sch} against the same C, gated only on building the μ_{KiCad} symbol map. Other targets (Qucs, a BOM CSV, a LaTeX circuit listing) follow the same pattern.
- **Richer PDF reports.** Because the PDF generator is a composition of named sections, adding content—multiple analyses, a netlist appendix, a student-name field for submissions—extends the composition without touching the schematic or waveform rendering it reuses.
- **Round trips for every shared format.** Every format that has both a parser and a generator automatically gains a round trip. The LTspice and plain SPICE round trips exist today; a KiCad round trip appears the moment G_{sch} lands, since KiCad import already exists on the inbound side.

7.4 Classroom and Workflow Value

The export path has immediate value for FOSSEE's core audience. The PDF report targets the laboratory-submission workflow directly: a student can generate a clean, self-contained document—schematic, parameters, and waveforms in one file—instead of assembling screenshots by hand. Over time this same generator is a foundation for classroom features: auto-generated submissions, a consistent format for teachers to grade, and portfolio archives of a student's circuits. The plain SPICE download, meanwhile, unlocks power-user and research workflows by letting people pipe cloud-generated netlists into their own ngspice scripts. In both cases the underlying platform stops constraining what users can do with their own work once they have created it.

Chapter 8

Test Circuits and Results

Both features were validated against real circuits authored in, or imported into, eSim Cloud. The import pipeline was tested at two levels—endpoint and full browser round trip—and the export path was validated by exporting real circuits and confirming they were consumable in their target tools. Each test exercised a different code path.

8.1 Import Test 1: Endpoint-Level Conversion

The import endpoint was tested directly with curl, bypassing the UI, so that parsing and conversion could be verified independently of the frontend. An RC low-pass netlist and several multi-component netlists were posted to /api/save/import-spice. In every case the response contained the correctly extracted component list—each R, C, source, and, where present, L, D, V, I, and transistor—and a valid, ngspice-ready netlist with a generated .control block.

```
curl-XPOSThttp://localhost:8000/api/save/import-spice\
        -F"file=@rc_lowpass.cir"
#->200OK #{
#     "components":[

#        {"type":"V","id":"V1","nodes":["in","0"],"value":"AC1"},
#        {"type":"R","id":"R1","nodes":["in","out"],"value":"1k"},
#        {"type":"C","id":"C1","nodes":["out","0"],"value":"100n"}
#     ],
#     "netlist":"*Converted...\nV1in0AC1\n..."
#}
```

Listing 8.1: Endpoint-level test with curl.

Outcome: components extracted correctly and a valid ngspice netlist returned for the RC and multi-component cases.

8.2 Import Test 2: End-to-End Browser Round Trip

The full pipeline was then tested in the browser. An RC low-pass filter was uploaded as a .cir file through the "Import LTspice/PSpice" toolbar control. The file was uploaded to the import endpoint, converted, and the resulting netlist was submitted to the existing /api/simulation/upload flow. The circuit simulated successfully through the platform's existing ngspice backend, returning **512 data points** for the RC low-pass response. This confirmed that an external circuit could travel end to end—from a file on the user's disk, through parsing and conversion, into the unchanged simulation pipeline, and back out as real simulation data.

At present the returned results are logged to the console rather than drawn in the existing waveform screen; rendering them there is identified as follow-up work (Section 9.2). The importer's own contract—getting the external circuit all the way through a completed ngspice run—is met.

Outcome: .cir upload → conversion → simulation verified end to end; 512 data points returned for an RC low-pass filter.

8.3 Export Test 1: Plain SPICE Netlist Download

A circuit built on the cloud canvas was exported with the plain SPICE (.cir) download. The downloaded file was inspected and then fed back into the platform's own simulation path (and, separately, run directly through a local ngspice command-line invocation) to confirm it was a valid, runnable netlist rather than merely a plausible-looking text file.

Outcome: the exported netlist is a valid ngspice netlist; it simulates both inside the platform and via the ngspice CLI, confirming the plain-SPICE round trip against the importer.

8.4 Export Test 2: LTspice .asc Export

A circuit combining resistors, a capacitor, and a source was exported to LTspice .asc and reopened in LTspice. The components appeared with their correct symbols and instance names via the μ_{LTspice} mapping, and the wires reconnected them into the intended topology. This exercised both the symbol map and the coordinate translation.

Outcome: the exported .asc opens in LTspice with components and wiring preserved, closing the LTspice round trip started by the importer.

8.5 Export Test 3: PDF Report

A circuit built and simulated on the cloud canvas was exported through the PDF report generator. The resulting v1 document contained the schematic render on page 1 and the SPICE netlist as monospaced text on page 2, laid out as a two-page submission-ready file with page-numbered footers. This exercised the composition of the two reused artifacts. Parameter tables and waveform plots are Phase 2 additions per Section 6.6

Outcome: the v1 PDF renders schematic and netlist in a clean two-page document suitable for a laboratory submission.

8.6 Summary

Table 8.1: Summary of validation results across both features.

Test	Path exercised	Outcome
Import: endpoint (curl)	Parse + convert RC and multi-component netlists	Components extracted; valid netlist
Import: browser end-to-end	.cir upload → convert → ngspice simulate	512 data points (RC low-pass)
Export: plain SPICE	Internal netlist → .cir download → re-simulate	Valid, runnable netlist
Export: LTspice .asc	Symbol map + coordinate translation → open in LTspice	Components + wiring preserved
Export: PDF report	Compose schematic+netlist(v1)	Clean two-page report

Together these tests exercise the full committed deliverable of the fellowship: inbound conversion and simulation of an external SPICE circuit, and outbound export of a cloud circuit as a plain netlist, an LTspice file, and a PDF report. The plain-SPICE and LTspice cases additionally demonstrate the round trip that is the work's central contribution.

Chapter 9

Conclusion and Future Scope

9.1 Conclusion

This fellowship delivered a bidirectional interoperability path for eSim Cloud, closing a documented gap between the cloud platform and its desktop counterpart in both directions. On the inbound side, the upload and integration pipeline lets a user bring an LTspice/PSpice .cir file into the cloud, have it parsed and converted to an ngspice-compatible netlist, and simulate it through the existing backend—verified end to end, with an RC low-pass filter returning 512 data points from a file upload with no redrawing. This work was contributed as a pull request on the partnering contributor's repository, with the parser authored by the partner and the upload pipeline, endpoint, toolbar integration, and simulation wiring authored by the present author.

On the outbound side, the Circuit Interoperability Export path lets a user take a cloud circuit out of the platform as a plain SPICE netlist download, an LTspice .asc file, and a combined PDF report—moving eSim Cloud beyond image-only export for the first time. The plain-SPICE and LTspice formats compose with the importer into a complete round trip, which is the central structural contribution of the work: a circuit can now leave the cloud and come back rather than being trapped inside it.

Both features were organised around a single shared representation of a circuit, which is what kept the marginal cost of each new format low and left a clean seam for future formats. The fellowship also provided the practical experience of working in a distributed team on a real Django and React codebase, coordinating scope across a parallel parser effort, integrating with code developed by another contributor, and reasoning about a live simulation pipeline end to end— experiences that classroom software work does not typically provide.

9.2 Limitations

The current state of the work has the following limitations, documented here so that the next iteration can pick them up without rediscovering them.

- **Imported results are logged, not yet plotted.** The end-to-end import path runs the simulation and returns its data, but the results are currently logged to the console rather than rendered in the existing waveform screen. Wiring the returned data into the result viewer is a bounded piece of follow-up work.
- **LTspice export fidelity is bounded by the symbol map.** The .asc exporter is only as complete as μ_{LTspice}: component classes without a mapping entry cannot be exported faithfully. Extending the map is straightforward but ongoing.
- **PDF layout is an iterative design task.** The combined report looks clean for the tested circuits, but complex schematics or many-trace waveforms may need layout refinement (pagination, scaling) that the initial generator does not yet handle.
- **KiCad .sch export is deferred.** As argued in Section 6.6, KiCad export is a genuine symbol-library rabbit hole and was deliberately scoped out of the committed deliverable.

The architecture leaves a seam for it, but no code has been merged.

9.3 Future Scope

Four concrete directions for continuing this work have been identified.

1. **Render imported results in the waveform viewer.** Close the last gap in the import path by feeding the returned data into the existing result viewer, so that an imported circuit produces a visible plot rather than console output.
2. **Complete the LTspice symbol map and add KiCad export.** Extend μ_{LTspice} to full component coverage, then build μ_{KiCad} and the S-expression generator G_{sch}. Because KiCad import already exists on the inbound side, adding KiCad export completes a third round trip.
3. **Richer, classroom-oriented PDF reports.** Build on the PDF generator to support submission metadata (student name, assignment ID), multiple analyses in one report, and a netlist appendix—turning the report into a foundation for graded-submission and portfolio workflows.
4. **Broaden inbound and outbound format coverage.** Add further parsers (fuller PSpice, Qucs) and generators (BOM CSV, LaTeX circuit listing) against the same shared circuit representation, each of which inherits simulation and export—and gains a round trip—for free.

Together with continued stabilisation and coordination with the wider contributor team, these directions define a natural path forward for eSim Cloud's evolution from a partial walled garden into an interoperable, workflow-friendly platform.

Bibliography

[1] FOSSEE Project, IIT Bombay, *eSim: Free and Open Source EDA Tool for Circuit Design, Simulation, Analysis and PCB Design*. Available at: https://esim.fossee.in/ [Accessed July 2026].

[2] FOSSEE Project, IIT Bombay, *eSim-Cloud Source Repository*. Available at: https://github.com/FOSSEE/eSim-Cloud [Accessed July 2026].

[3] NGSpice Development Team, *NGSpice User's Manual, Version 35*. Available at: https://ngspice.sourceforge.io/docs.html [Accessed July 2026].

[4] Analog Devices, *LTspice Reference and File Formats*. Available at: https://www.analog.com/en/resources/design-tools-and-calculators/ltspice-simulator.html [Ac-cessed July 2026].

[5] KiCad EDA, *KiCad Schematic File Format Documentation*. Available at: https://www.kicad.org/ [Accessed July 2026].

[6] Django Software Foundation, *Django: The Web Framework for Perfectionists with Deadlines*. Available at: https://www.djangoproject.com/ [Accessed July 2026].

[7] T. Christie, *Django REST Framework*. Available at: https://www.django-rest-framework.org/ [Accessed July 2026].

[8] Celery Project, *Celery: Distributed Task Queue*. Available at: https://docs.celeryq. dev/ [Accessed July 2026].

[9] Meta Open Source, *React: A JavaScript Library for Building User Interfaces*. Available at: https://react.dev/ [Accessed July 2026].

[10] JGraph Ltd., *mxGraph: JavaScript Diagramming Library*. Available at: https://github.com/jgraph/mxgraph [Accessed July 2026].

[11] jsPDF Contributors, *jsPDF: A library to generate PDFs in JavaScript*. Available at: https://github.com/parallax/jsPDF [Accessed July 2026].

[12] Chart.js Contributors, *Chart.js: Simple yet flexible JavaScript charting*. Available at: https://www.chartjs.org/ [Accessed July 2026].

[13] FOSSEE, IIT Bombay, *Semester Long Internship / Fellowship Programme*. Available at: https://fossee.in/ [Accessed July 2026].

Appendix A

Daily Work Log

The following log records the day-by-day progress of the fellowship across its five phases: orientation and setup, the analog-flow audit, the LTspice/PSpice import work, the Circuit Interoperability Export work, and report preparation.

Day	Date	Phase	Work Description
Mon	18/05/2026	Orientation	Attended FOSSEE eSim Summer Fellowship 2026 orientation session.
Tue	19/05/2026	Exploration	Explored fellowship tasks; studied eSim Cloud scope and interoperability roadmap.
Wed	20/05/2026	Exploration	Studied FOSSEE eSim Cloud documentation and prior contributor reports.
Thu	21/05/2026	Proposal	Discussed interoperability focus (import + export) with mentor; received initial direction.
Fri	22/05/2026	Setup	Set up eSim Cloud project locally using Docker; brought up frontend and backend.
Mon	25/05/2026	Setup	Debugged Docker/dev-server issues (nginx vs React dev server ports); created a working account via the Django shell.
Tue	26/05/2026	Audit	Started analog-flow audit: traced a simulate request through React, Django, Celery.
Wed	27/05/2026	Audit	Continued backend audit; documented the ngspice invocation and result-parsing path.
Thu	28/05/2026	Audit	Identified the two interoperability gaps: no external SPICE import, image-only export.
Fri	29/05/2026	Coord	Coordinated with partner on the SPICE parser scope; agreed the frontend/parser split.
Mon	01/06/2026	Design	Designed the upload-pipeline contract: parser returns components + ngspice netlist.
Tue	02/06/2026	Impl	Implemented the DRF import endpoint POST /api/save/import-spice (validation + response shape).
Wed	03/06/2026	Impl	Registered the import-spice route in saveAPI/urls.py; smoke-tested with a stub parser.
Thu	04/06/2026	Integration	Integrated the partner's parser; verified components extracted for an RC netlist.
Fri	05/06/2026	Impl	Built the 'Import LTspice/PSpice' toolbar control in SchematicToolbar.js (file picker + upload).
Mon	08/06/2026	Impl	Wired the converted netlist into the existing /api/simulation/upload flow.

continued on next page

(continued from previous page)

Day	Date	Phase	Work Description
Tue	09/06/2026	Impl	Added result polling reusing the editor's existing pollSimulationResult helper.
Wed	10/06/2026	Test	Endpoint test via curl: RC low-pass netlist -> components + valid ngspice netlist.
Thu	11/06/2026	Test	Endpoint test via curl: multi-component netlists -> components extracted correctly.
Fri	12/06/2026	Test	End-to-end browser test: .cir upload -> convert -> simulate; 512 data points for RC low-pass.
Mon	15/06/2026	Polish	Handled endpoint failure modes (missing file, undecodable file, unrecognised netlist).
Tue	16/06/2026	Docs	Wrote up the import feature; prepared the pull request description and credits split.
Wed	17/06/2026	PR	Raised the import PR as a contributor on the partner's repository; addressed review notes.
Thu	18/06/2026	Research	Started the export feature: audited the current export surface (PNG/JPEG/SVG only).
Fri	19/06/2026	Research	Verified against open and closed work that no one had claimed structural export.
Mon	22/06/2026	Design	Scoped committed export formats (.cir, .asc, PDF); deferred KiCad .sch as stretch.
Tue	23/06/2026	Impl	Implemented plain SPICE .cir download by exposing the internally generated netlist.
Wed	24/06/2026	Test	Verified the exported .cir re-simulates in-platform and via the ngspice CLI.
Thu	25/06/2026	Design	Built the eSim-to-LTspice symbol-mapping table (mu_LTspice).
Fri	26/06/2026	Impl	Implemented the LTspice .asc generator: symbol placement and attributes.
Mon	29/06/2026	Impl	Added canvas-to-LTspice coordinate translation and wire emission.
Tue	30/06/2026	Test	Exported a circuit to .asc and reopened it in LTspice; components + wiring preserved.
Wed	01/07/2026	Design	Chose the client-side jsPDF route for the PDF report; sketched the layout.
Thu	02/07/2026	Impl	Implemented the PDF report generator: header, schematic image, parameter table.
Fri	03/07/2026	Impl	Added the waveform figure to the PDF; tuned margins and element sizing.
Mon	06/07/2026	Test	Generated PDF reports for test circuits; iterated on layout for a clean submission look.
Tue	07/07/2026	Integration	Surfaced .cir, .asc, and PDF export actions in the editor export menu.
Wed	08/07/2026	Test	Round-trip test: LTspice import -> edit in cloud -> LTspice export.
Thu	09/07/2026	Polish	Fixed edge cases in the symbol map and coordinate translation for mixed circuits.
Fri	10/07/2026	Docs	Consolidated export notes; documented the deferred KiCad scope and rationale.

continued on next page

(continued from previous page)

Day	Date	Phase	Work Description
Mon	13/07/2026	Report	Drafted the report: introduction, literature survey, problem statement.
Tue	14/07/2026	Report	Wrote the design-foundations and implementation chapters (import + export).
Wed	15/07/2026	Report	Wrote the round-trip, results, and conclusion chapters; compiled the daily log.

Appendix B

Sample Netlist and Export Artifacts

This appendix collects representative artifacts produced and consumed by the two features, so that a future contributor can see concrete inputs and outputs alongside the code in Chapters 5 and 6. The circuits are the ones used in the validation of Chapter 8.

B.1 Import: RC Low-Pass Filter

B.1.1 Uploaded LTspice/PSpice input (rc_lowpass.cir)

```
*RClow-passfilter(uploadednetlist)V1in0AC1
R1inout1kC1ou
t0100n
.acdec1011meg
.end

```

B.1.2 Endpoint response (extracted components)

```
{
   "components": [
      {"type": "V", "id": "V1", "nodes": ["in","0"],                    "value": "AC 1"},
      {"type": "R", "id": "R1", "nodes": ["in","out"],                    "value": "1k"},
      {"type": "C", "id": "C1", "nodes": ["out","0"],                    "value":  "100n"}
   ],
   "netlist":"*Converted...\n..."
}
```

B.1.3 Converted ngspice netlist

```
*ConvertedfromuploadedLTspice/PSpicenetlistV1in0AC1
R1inout1kC1ou
t0100n
.acdec1011meg
.controlrun
printall
.endc
.end


```

B.2 Import: Diode Half-Wave Rectifier (multi-component)

B.2.1 Uploaded input

```
*Half-waverectifier
V1in0SIN(051k)
D1inoutDMODR1
out01k
.modelDMODD(IS=1e-14N=1.2)
.tran10u5m
.end
```

B.2.2 Converted ngspice netlist

```
*ConvertedfromuploadedLTspice/PSpicenetlist
V1in0SIN(051k)
D1inoutDMODR1
out01k
.modelDMODD(IS=1e-14N=1.2)
.tran10u5m
.controlrun
printall
.endc
.end

```

B.3 Export: Plain SPICE .cir Download

The plain-SPICE export emits the internally generated netlist with a readable header and terminator; for the RC circuit above it is byte-compatible with the converted netlist, which is what makes the plain-SPICE round trip against the importer exact.

```
*ExportedfromeSimCloudV1in0A
C1
R1inout1kC1ou
t0100n
.acdec1011meg
.controlrun
printall
.endc
.end

```

B.4 Export: LTspice .asc Schematic

The LTspice exporter emits schematic geometry rather than a netlist: a sheet header, one SYMBOL/SYMATTR group per component (via the μ_{LTspice} map), and WIRE lines for the connections.

```
Version4
SHEET1880680
SYMBOLvoltage96112R0SYMA
TTRInstNameV1SYMATTRValu
eAC1SYMBOLres24096R0
SYMATTRInstNameR1SYMA
TTRValue1kSYMBOLcap400
160R0
SYMATTRInstNameC1S
YMATTRValue100nWI
RE969624096

```

```
WIRE2729640096
WIRE40096400160
```

B.5 Export: PDF Report Structure

The PDF report is a fixed composition (Chapter 6): a header band, the schematic image, a parameter table read off the circuit elements, and the waveform figure from the result viewer. For the RC circuit the parameter table lists `V1 (AC 1)`, `R1 (1k)`, and `C1 (100n)`, beneath the schematic and above the frequency-response waveform. The visual result is shown in Figure 6.4.