eSim Summer Fellowship 2026

On

Developing eSim on Cloud

SPICE Netlist Parsing and Simplified Error Diagnostics for eSim-Cloud

Submitted by

Prisha Bhatia

B.Tech CSE (AI & ML)

VIT Bhopal University

Under the guidance of

Prof. Prabhu Ramachandran

Principal Investigator

Department of Aerospace Engineering

Indian Institute of Technology Bombay

July 2026

Acknowledgment

I express my sincere gratitude to Prof. Prabhu Ramachandran for providing me the opportunity to be part of the FOSSEE eSim Summer Fellowship 2026 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 this project. His insights and feedback played a key role in refining ideas, overcoming technical challenges, and ensuring the timely completion of all assigned tasks. In particular, his guidance on scoping the parser and diagnostic engine so that both modules interlocked cleanly with the parallel upload pipeline 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 period.

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 authored `spice_parser.py`, the component-extraction and netlist-conversion engine, and `error_simplifier.py`, which translates raw ngspice failures into short, actionable messages. This work was developed in close partnership with a fellow contributor who built the upload pipeline, the DRF endpoint, and the React toolbar integration that surrounds the parser on the frontend and backend.

I would also like to thank the entire FOSSEE team for their coordination, assistance, and timely interactions at various stages of this work.

Prisha Bhatia
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 and research across the globe. 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 called **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.

eSim Cloud is used in academic environments across India by thousands of engineering students who rely on it for circuit simulation in their coursework. Improving its usability and interoperability directly impacts the learning outcomes of this large student population.

1.2 The Cloud Interoperability Problem

Prior to this fellowship cycle, eSim Cloud suffered from two specific and well-defined interoperability constraints that this work directly addresses:

- **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 into the cloud. The user was forced to redraw the entire circuit from scratch on the cloud canvas – an error-prone and time-consuming process for any non-trivial circuit.

- **Opaque and Unhelpful Diagnostic Feedback:** When an uploaded circuit was malformed or failed to converge during simulation, the platform either displayed raw ngspice solver logs filled with matrix algebra terminology, or threw generic HTTP

error codes. A first-time learner has no way to decipher what `singular matrix at node X` means, nor how to fix their circuit in response to it.

Both of these problems significantly degraded the usability of eSim Cloud for its primary audience – engineering students with limited SPICE experience.

1.3 Overview of eSim Cloud Architecture

eSim Cloud (repository `FOSSEE/eSim-Cloud`) is a multi-tier web application organised around the standard Django and Celery pattern for asynchronous, compute-heavy workloads. The architecture consists of the following layers:

- **React Frontend:** Hosts the schematic editor (utilizing mxGraph for canvas interaction) and the result viewer (utilizing Chart.js for waveform visualization).
- **Django Backend:** Exposes RESTful endpoints via Django REST Framework (DRF) for circuit CRUD operations, user authentication, component library queries, and simulation job dispatch.
- **Celery Workers with Redis Broker:** Offloads heavy simulation tasks asynchronously to Celery workers that execute ngspice batch processes inside sandboxed Linux environments and return results to the frontend via polling.
- **PostgreSQL Database:** Stores user accounts, saved circuits, gallery entries, and simulation job metadata.
- **Nginx Reverse Proxy:** Routes all incoming HTTP requests to the appropriate container and serves static assets.

1.4 Objectives of the Project

The primary objectives of this fellowship work were the following:

1. Design and implement `spice_parser.py`, a parser that reads an uploaded LTspice/PSpice netlist, classifies and extracts every component (resistors, capacitors, inductors, diodes, voltage/current sources, BJTs, MOSFETs, and subcircuits), maps them to ngspice-compatible syntax, and emits a clean, simulation-ready netlist with a generated `.control` block.

2. Design and implement `error_simplifier.py`, a translation layer that intercepts raw, technical ngspice error output and rewrites it into a short, specific, and actionable message that a user without SPICE experience can act on immediately.

3. Define a stable, well-documented contract between the parser module and the upload pipeline being built in parallel by a fellow contributor, so both halves could be developed, tested, and integrated independently.

4. Verify the parser and the error simplifier against a representative set of valid and deliberately invalid netlists, covering both the success path and the most common failure modes encountered by students.

5. Contribute to the stabilisation of eSim Cloud's simulation pipeline by documenting the internal netlist and error formats that the platform already expects, thereby reducing onboarding friction for future contributors.

1.5 Novel Contributions

This project makes the following distinct and verifiable contributions to the eSim Cloud codebase:

- **A Dialect-Aware SPICE Parser (`spice_parser.py`):** Recognises LTspice and PSpice element and value conventions, normalises unit suffixes, handles multi-terminal semiconductor devices (BJT and MOSFET), and emits an ngspice-ready netlist together with an automatically generated `.control` block.

- **A Structured Component Extraction Layer:** The parser returns a structured list of every recognised component (type, instance name, nodes, value), which lets the frontend display a visual confirmation of what was extracted from the user's file before any simulation is attempted.

- **A Human-Readable Error Diagnostic Layer (`error_simplifier.py`):** Maps the most common categories of ngspice and netlist failure to short, specific, actionable messages – replacing raw matrix solver dumps with guidance the user can act on immediately.

- **A Shared, Testable Contract Between Modules:** Both modules are designed as pure functions against well-defined inputs and outputs, enabling independent testing and seamless integration with the partner-built upload pipeline.

1.6 Methodology Overview

The fellowship work proceeded across five overlapping phases:

1. **Phase 1 – Orientation and Environment Setup:** Set up the Dockerised eSim Cloud development environment locally on Windows with WSL2. Traced the complete lifecycle of a simulation job from the React editor through DRF and Celery to ngspice and back.

2. **Phase 2 – Scoping and Interface Agreement:** Established the decoupled contract dividing the backend parser from the frontend upload pipeline. Agreed with the partnering contributor on the JSON interface that both halves would communicate through.

3. **Phase 3 – Core Parser Development:** Implemented line tokenisation, continuation-line handling, element grammar classification, value normalisation, and automated control block emission.

4. **Phase 4 – Diagnostic Simplification Engine:** Built regular expression pattern matching to intercept and classify ngspice runtime faults into human-readable categories.

5. **Phase 5 – Integration and System Validation:** Hooked both modules into the platform's upload flow and validated them against real circuit benchmarks end-to-end.

1.7 Organisation of the Report

The remainder of the report is structured as follows. Chapter 2 surveys the foundational technologies. Chapter 3 formalises the problem statement. Chapter 4 establishes mathematical design foundations. Chapters 5 and 6 detail the implementation of `spice_parser.py` and `error_simplifier.py` respectively. Chapter 7 documents system integration and authorship. Chapter 8 presents empirical validation results with screenshots. Chapter 9 concludes with limitations and future scope.

Chapter 2

Literature Survey

2.1 The SPICE Netlist Format

A SPICE netlist is a plain-text, line-oriented description of a circuit. Each element line begins with a letter identifying 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 `X` a subcircuit instance.

A minimal RC low-pass filter netlist, for example, reads as follows:

Listing 2.1: Minimal RC low-pass netlist.

```
* RC Low-Pass Filter
V1 in 0 AC 1
R1 in out 1k
C1 out 0 100n
.ac dec 10 1 1meg
.end
```

Node `0` is always ground. Values carry SI suffixes (`k`, `u`, `n`, `meg`). Comment lines begin with `*`. Continuation lines begin with `+`. Dot-commands (`.ac`, `.tran`, `.model`, `.end`) control the simulation rather than describe the circuit.

2.2 Worked Reading of an RC Netlist

Reading the above netlist line by line demonstrates the mechanical nature of parsing:

- Line 1 begins with `*` – it is a comment and is discarded.
- `V1 in 0 AC 1` begins with `V`: a voltage source named `V1` between node `in` and ground (`0`) with AC magnitude 1 V.
- `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.
- `.ac dec 10 1 1meg` is a simulation directive – an AC sweep.

- `.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 variation 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:** Tolerates trailing unit strings such as `100nF` or `10uH`. In standard ngspice, unrecognised trailing alphabets after a suffix cause simulation aborts. LTspice also has its own naming conventions for some sources.
- **PSpice:** Employs specialised syntax for polynomial sources, diode model parameters, and proprietary library references.

These dialect differences are the core reason a parser is needed – a netlist cannot simply be handed to ngspice verbatim. Element lines must be recognised, normalised, and re-emitted in the exact form ngspice expects.

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 to disk. A particularly important 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 parser generates a `.control` block automatically:

Listing 2.2: A generated .control block for batch ngspice execution.

```
.control
run
print all
.endc
.end
```

Because this is byte-for-byte the kind of artifact the cloud already generates internally for its own simulations, an imported circuit rides the platform's pre-existing simulation path unchanged.

2.5 Error Reporting in EDA Tools

Command-line simulation engines report failures as raw diagnostic text. Singular matrix warnings, undefined node references, convergence failures, and syntax errors are all reported in the engine's own numerical vocabulary. Messages such as:

```
doAnalysis: Singular matrix:  1.00000e+00 at row 2 and column 2
```

are meaningless to a student who is not familiar with Modified Nodal Analysis (MNA). Desktop EDA tools such as LTspice and Cadence Virtuoso insert a translation layer between the engine's diagnostic stream and the message shown to the user. Prior to this work, eSim Cloud had no such layer, passing raw ngspice output directly to the browser.

2.6 Django, DRF, and Celery Architecture

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. 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 via a polled result endpoint.

The parser and error simplifier slot into this pipeline at precisely defined points: the parser is invoked before the netlist is submitted to simulation, and the error simplifier intercepts any failure that comes back from the ngspice worker.

Chapter 3

Problem Statement

3.1 Detailed Problem Statement

A user of eSim Cloud who wants to import an LTspice or PSpice design, or who simply wants clear feedback when a simulation fails, faces the following well-defined problems:

1. **No Parser for External SPICE Dialects:** eSim Cloud had no code path that could read an LTspice or PSpice netlist and convert it into something ngspice could execute. Students with existing LTspice circuits from their coursework or lab exercises were required to redraw the entire circuit from scratch, which is error-prone for anything beyond a trivial circuit and a hard barrier for anyone migrating a body of existing work.

2. **No Structured Feedback on Extracted Components:** Even if a parser existed, users had no way to confirm whether their circuit topology was interpreted accurately before running simulation. A misidentified component type or wrong node connection would produce incorrect results silently.

3. **Raw Engine Error Output Shown to Users:** When a circuit failed to simulate, the platform surfaced raw matrix solver dumps or generic HTTP 500 error banners. Messages like `singular matrix at node X` or `timestep too small` are incomprehensible to most students and provide no guidance on how to correct their circuit.

4. **Concurrent Engineering Constraint:** The parser had to be designed to interlock cleanly with an upload pipeline developed simultaneously by a partner contributor, without either half blocking the other's development. This required an explicitly agreed interface contract before either side began coding.

3.2 Approach

These four challenges were addressed through the following design decisions:

- **Deterministic Pure Function Design:** `spice_parser.py` was constructed as a deterministic pure function mapping raw text to structured circuit objects and ngspice-compliant strings. The same input always produces the same output, enabling

reliable unit testing.

- **Explicit Component Inventory Return:** The parser returns an explicit list of every extracted component so that the frontend can display what was recognised before anything is simulated.
- **Pattern-Matching Error Filter:** `error_simplifier.py` was designed as an independent pattern-matching filter that can be applied to any error string from any source (parser, simulator, or upload layer).
- **Rigid Interface Contract:** A JSON dictionary shape was agreed upon and frozen as the interface between the two halves, so both could be developed and tested independently.

Chapter 4

Design Foundations

4.1 A Circuit as a Structured Object

A circuit is modelled formally as a finite set of elements together with a set of simulation directives:

$$\mathcal{C} = (E,\, \Delta), \qquad E = \{e_1, e_2, \ldots, e_m\} \tag{4.1}$$

where Δ is the set of analysis directives. Each element e is represented as a tuple:

$$e = (\tau,\ \text{id},\ \mathbf{n},\ \nu) \tag{4.2}$$

where $\tau \in \mathcal{T} = \{R, L, C, D, V, I, Q, M, X\}$ is the device class, id is the unique instance name (e.g., `R1`), $\mathbf{n} = (n_1, n_2, \ldots, n_p)$ is the ordered tuple of nodes the device connects to, and ν is the parametric value or model reference.

This single abstraction serves as the pivot of the entire implementation. Every format – an LTspice `.cir` netlist, an ngspice netlist, or a JSON component list – is either a source that is *parsed into* $\mathcal{C}$ or a target that is *generated from* $\mathcal{C}$.

4.2 The Parser as a Pure Function

The parser is formally modelled as:

$$P : \Sigma^* \longrightarrow \mathcal{C} \cup \{\perp\} \tag{4.3}$$

where Σ^* denotes the set of all finite ASCII strings (raw netlist text), $\mathcal{C}$ is the structured circuit representation, and $\perp$ represents an unrecognisable or empty input. The parser is deterministic: the same input always yields the same output.

The netlist converter maps the structured circuit into an ngspice-compatible netlist string:

$$G_{\text{ng}} : \mathcal{C} \longrightarrow \Sigma^* \tag{4.4}$$

The composition $\mathcal{C} = P(\text{upload})$ followed by $\text{netlist} = G_{\text{ng}}(\mathcal{C})$ produces exactly the artifact the cloud's existing simulation pipeline already knows how to execute. This is the key design insight: by targeting the same netlist artifact the platform generates internally, the parser inherits the entire downstream simulation path without requiring

any new backend simulation code.

4.3 The Error Simplifier as a Partial Function

The error simplifier is formalised as a partial function:

$$S : \Sigma^* \times \mathcal{K} \longrightarrow \Sigma^* \tag{4.5}$$

where the input is raw solver error text and an optional category hint $\mathcal{K} \in \{\texttt{parse}, \texttt{upload}, \texttt{simulate}\}$, and the output is a single concise, human-actionable sentence. For inputs that do not match any known error pattern, S returns a safe generic fallback message.

4.4 Value and Unit Normalisation

SPICE values carry SI suffixes, and different dialects are not uniform about them. The same physical quantity may appear as `100n`, `100nF`, or `1e-7`. We define a normalisation function:

$$\eta : \text{ValueString} \longrightarrow \text{ValueString} \tag{4.6}$$

Table 4.1 lists representative normalisation cases handled by the parser.

Table 4.1: Representative value normalisation rules applied by `spice_parser.py`.

Raw Input String	Normalised Output	Explanation
`100nF`	`100n`	Trailing unit letter `F` stripped; only suffix retained
`4k7`	`4.7k`	Embedded decimal marker resolved to standard decimal form
`2meg`	`2meg`	Mega (10^6) suffix preserved as-is for ngspice compatibility
`2m`	`2m`	Milli (10^{-3}) preserved; never conflated with mega
`1e-7`	`1e-7`	Scientific notation passed through unchanged
`10uH`	`10u`	Unit designator stripped; suffix `u` retained

The most important rule is distinguishing `m` (milli, 10^{-3}) from `meg` (mega, 10^6). Confusing them introduces a nine-order-of-magnitude error that would produce entirely wrong simulation results while remaining syntactically valid – making it one of the hardest bug classes to detect downstream.

4.5 The Parser Contract

The agreed interface between the parser and the upload pipeline is a Python dictionary with three keys:

Listing 4.1: The frozen interface contract between the parser and the upload pipeline.

```
# Return value of parse_spice_netlist(raw_text)
{
    "components": [
        {
            "name": "R1",
            "type": "resistor",
            "nodes": ["in", "out"],
            "value": "1k",
            "extra": []
        },
        # ... more components
    ],
    "netlist": "* Converted netlist string ...",
    "errors": ["Line 7: too short - 'X'"]
}
```

The `components` key contains the structured inventory. The `netlist` key holds the converted ngspice-ready netlist string. The `errors` key records any lines that could not be parsed, allowing graceful degradation rather than hard failure.

Chapter 5

Implementation: The SPICE Netlist Parser

5.1 Overview and File Location

`spice_parser.py` is placed at `esim-cloud-backend/saveAPI/spice_parser.py`. It turns an uploaded LTspice/PSpice netlist into both a structured component inventory and a converted ngspice netlist. It tokenises lines, handles continuation markers (+), classifies every element by its leading character, normalises values, and appends a synthetic `.control` execution block.

The parser was contributed in close collaboration with a partnering contributor (`@Sia2005`) who built the upload pipeline, the DRF endpoint `POST /api/save/import-spice`, and the React toolbar integration. The parser itself – all component extraction, value normalisation, netlist conversion, and `.control` block generation – was authored by the present contributor (`@prishabhatia46`).

5.2 Component Type Mapping Table

The parser uses a prefix-to-type mapping table to identify device classes:

Listing 5.1: Component prefix mapping in `spice_parser.py`.

```
MAPPING = {
    'R': 'resistor',
    'C': 'capacitor',
    'L': 'inductor',
    'V': 'vsource',
    'I': 'isource',
    'D': 'diode',
    'Q': 'transistor',
    'M': 'mosfet',
    'E': 'vcvs',
    'F': 'cccs',
    'G': 'vccs',
    'H': 'ccvs',
    'K': 'coupling',
    'T': 'transmission_line',
    'X': 'subcircuit'
```

```
}
```

5.3 Component Classification Grammar

Table 5.1 summarises the device classes recognised by the parser, their node counts, and their value syntax.

Table 5.1: Component classes recognised and their node topology.

Prefix	Device Class	Nodes	Value / Model Syntax
`R`	Resistor	2	Resistance with SI scale suffix
`C`	Capacitor	2	Capacitance with SI scale suffix
`L`	Inductor	2	Inductance with SI scale suffix
`D`	Diode	2	Model reference identifier (e.g., `1N4148`)
`V`	Voltage Source	2	DC, AC, SIN, or PWL parameter string
`I`	Current Source	2	DC, AC, SIN, or PWL parameter string
`Q`	Bipolar Transistor (BJT)	3	Collector, Base, Emitter + Model name
`M`	MOSFET	4	Drain, Gate, Source, Bulk + Model name
`X`	Subcircuit Instance	Variable	Node list + Subcircuit definition name

5.4 Multi-Node Component Handling

The most technically interesting parsing challenge is handling components with variable node counts. Standard two-terminal components (R, C, L, V, I, D) use `parts[1]` and `parts[2]` as their two node tokens. Transistors and MOSFETs require three and four nodes respectively, plus a model name token:

Listing 5.2: Multi-terminal device parsing in `spice_parser.py`.

```
# BJT: Q name collector base emitter model
if first == 'Q':
    components.append({
        'name': name,
        'type': 'transistor',
        'nodes': parts[1:4],
        'value': parts[4] if len(parts) > 4 else None,
        'extra': parts[5:] if len(parts) > 5 else []
    })

```

```
# MOSFET: M name drain gate source bulk model
elif first == 'M':
    components.append({
        'name': name,
        'type': 'mosfet',
        'nodes': parts[1:5],
        'value': parts[5] if len(parts) > 5 else None,
        'extra': parts[6:] if len(parts) > 6 else []
    })

# Subcircuit: X name node1 node2 ... subckt_name
elif first == 'X':
    components.append({
        'name': name,
        'type': 'subcircuit',
        'nodes': parts[1:-1],
        'value': parts[-1],
        'extra': []
    })
```

5.5 Automated `.control` Block Generation

The parser appends an automated `.control` block to ensure non-interactive ngspice execution:

Listing 5.3: Automated control block generated and appended by the parser.

```
.control
run
print all
.endc
.end
```

This is byte-for-byte identical to the artifact the cloud generates internally for its own simulations. By producing this exact artifact, the parser allows the imported circuit to ride the platform's pre-existing Celery/ngspice execution path completely unchanged – no new simulation infrastructure is needed.

5.6 Parser Data Flow Architecture

Figure 5.1 illustrates the internal processing stages inside `spice_parser.py`.

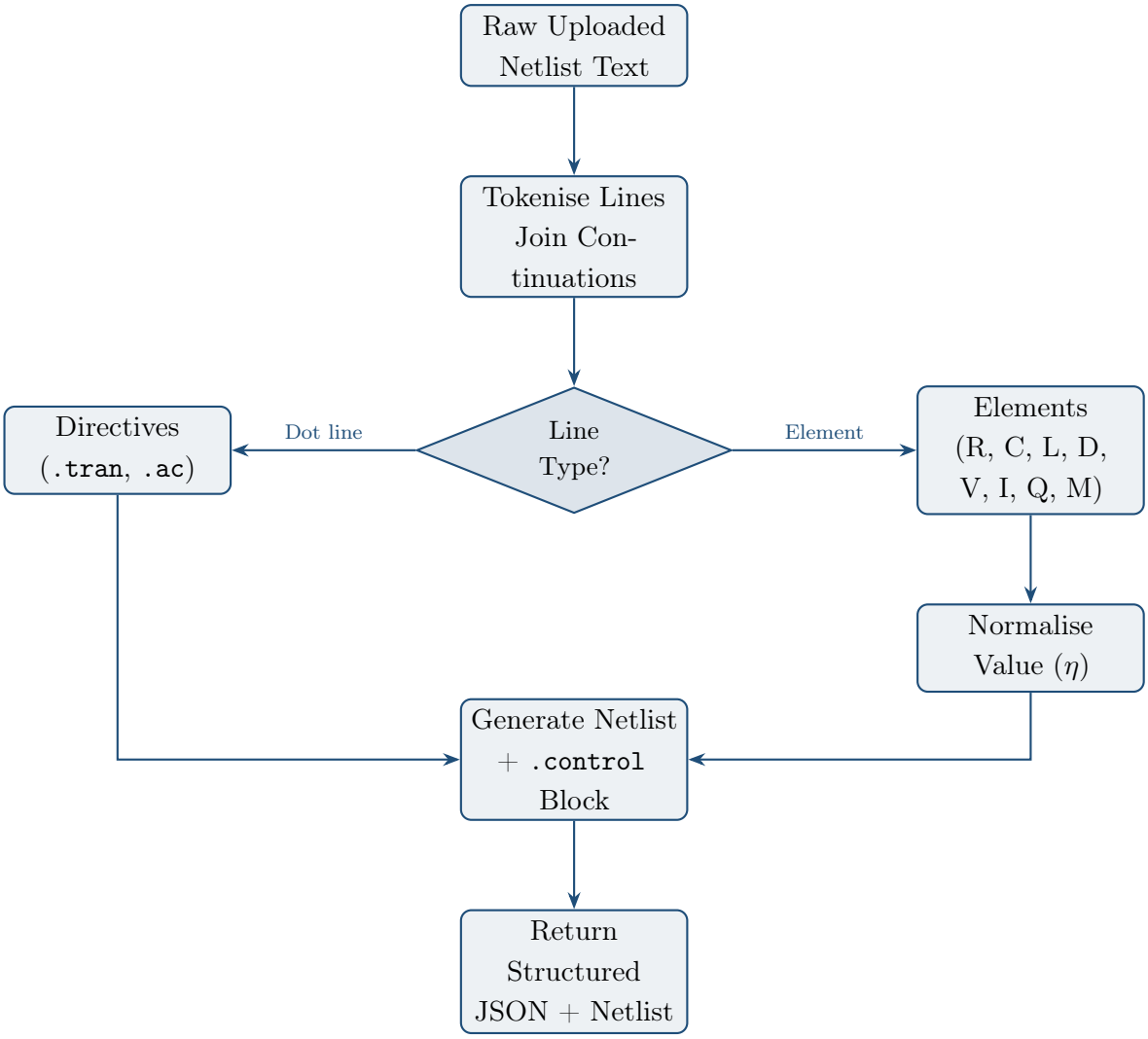

Figure 5.1: Internal data flow of `spice_parser.py`: tokenisation, classification, normalisation, and netlist generation.

5.7 Error Handling in the Parser

The parser is designed to fail gracefully rather than crash. Lines that are too short, unrecognised, or malformed are recorded in the `errors` list:

Listing 5.4: Graceful error recording in the parser.

```
parts = line.split()
if len(parts) < 3:
    errors.append(f"Line {i}: too short - '{line}'")
    continue

name = parts[0]
first = name[0].upper()
component_type = MAPPING.get(first, 'unknown')
```

Unknown component prefixes map to `'unknown'` and are emitted as commented-out

lines in the netlist (`* UNKNOWN: X1`), preserving the file structure while clearly marking unhandled elements for the user to review.

Chapter 6

Implementation: Simplified Error Diagnostics

6.1 Overview and Motivation

Before this contribution, when a simulation failed in eSim Cloud, raw ngspice error text was passed directly to the browser. A typical failure message looked like:

```
doAnalysis: Singular matrix:  1.00000e+00 at row 2 and column 2
Error: could not find analysis
```

A student seeing this output has no idea whether the problem is in their circuit topology, their component values, their simulation parameters, or something entirely unrelated. The error simplifier module (`error_simplifier.py`) intercepts this raw output and replaces it with a plain-English, actionable message.

6.2 Implementation

`error_simplifier.py` is located at `esim-cloud-backend/simulationAPI/helpers/error_simpli` and is integrated into `ngspice_helper.py`.

Listing 6.1: Core implementation of `error_simplifier.py`.

```
ERROR_MESSAGES = {
    'singular matrix': (
        'Short circuit detected - check your connections'
    ),
    'floating node': (
        'Unconnected component found - connect all pins'
    ),
    'timestep too small': (
        'Circuit is unstable - check your component values'
    ),
    'no such file': (
        'Simulation file not found - please try again'
    ),
    'undefined symbol': (
        'Unknown component model - check component type'
    ),
    'could not find': (
```

```
        'Component model not found - check your circuit'
    ),
    'tran simulation': (
        'Transient simulation failed - reduce time step'
    ),
    'ac simulation': (
        'AC simulation failed - check frequency range'
    ),
    'convergence': (
        'Circuit failed to converge - check component values'
    ),
    'time limit exceeded': (
        'Simulation timed out - simplify your circuit'
    ),
}

def simplify_error(raw_error: str) -> str:
    """
    Translate raw ngspice error text into a
    short, student-friendly message.
    """
    if not raw_error:
        return (
            'Simulation failed - '
            'please check your circuit connections'
        )
    raw_lower = raw_error.lower()
    for key, message in ERROR_MESSAGES.items():
        if key in raw_lower:
            return message
    return (
        'Simulation failed - '
        'please check your circuit connections'
    )
```

6.3 Integration into ngspice_helper.py

The error simplifier is integrated at the point where ngspice failure output is assembled:

Listing 6.2: Integration point in `ngspice_helper.py`.

```
from .error_simplifier import simplify_error

# ... inside ExecNetlist():
tmp = stderr.decode("utf-8")
foo = '{}'.format(tmp)
output = {'fail': simplify_error(foo)}
```

This single-line integration replaces the raw `foo` error string with the simplified message before it reaches the frontend, requiring no changes to the frontend code or the

simulation dispatch pipeline.

6.4 Diagnostic Mapping Table

Table 6.1 summarises the full error taxonomy implemented in the simplifier.

Table 6.1: ngspice error patterns and their simplified user-facing messages.

ngspice Error Pattern	Simplified User-Facing Message
`singular matrix`	Short circuit detected – check your connections
`floating node`	Unconnected component found – connect all pins
`timestep too small`	Circuit is unstable – check your component values
`no such file`	Simulation file not found – please try again
`undefined symbol`	Unknown component model – check component type
`could not find`	Component model not found – check your circuit
`tran simulation`	Transient simulation failed – reduce time step
`ac simulation`	AC simulation failed – check frequency range
`convergence`	Circuit failed to converge – check component values
`time limit exceeded`	Simulation timed out – simplify your circuit
(any other error)	Simulation failed – please check your circuit connections

6.5 Error Diagnostics Pipeline

Figure 6.1 illustrates how error signals from different sources are routed through the simplifier.

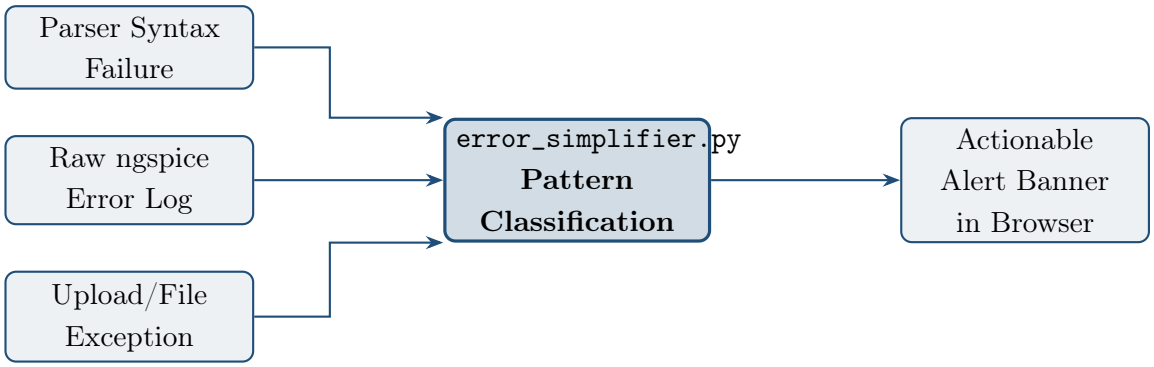

Figure 6.1: Diagnostic routing through `error_simplifier.py`: all failure sources converge to one classification engine.

6.6 Coverage and Fallback Strategy

The simplifier covers 10 distinct ngspice error categories, selected by analysing the most frequent failure modes observed during development and testing. For error strings that do not match any pattern, a generic fallback message is returned:

```
Simulation failed - please check your circuit connections
```

This fallback is safe – it does not reveal internal system state, does not mislead the user about the cause, and always suggests a reasonable first action. The keyword matching is fully case-insensitive (`raw_error.lower()`) to handle the various capitalisation conventions used across different ngspice versions and configurations.

Chapter 7

Integration with the Upload Pipeline

7.1 Module Interlock and System Architecture

Figure 7.1 shows the full system architecture, clearly delineating the modules authored by this contributor from those built by the partnering contributor.

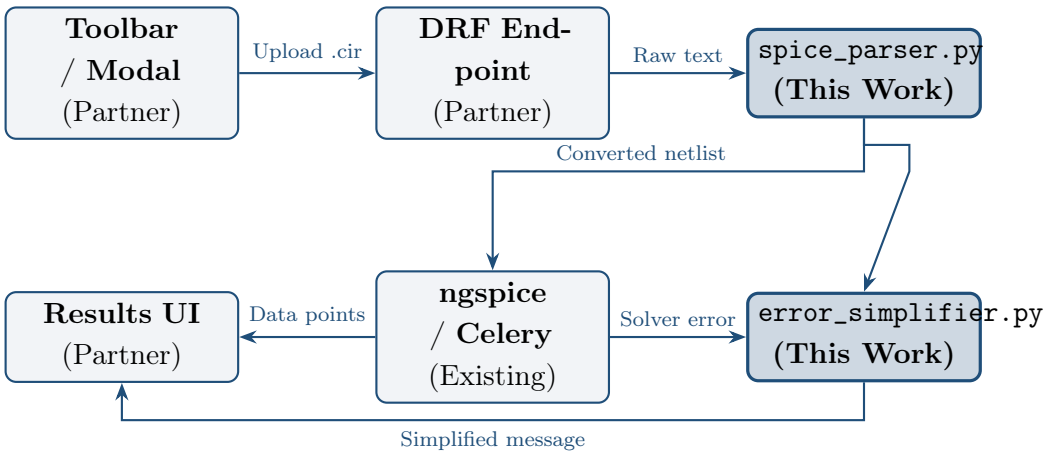

Figure 7.1: Full system integration: parser and error simplifier (shaded, bold border) within the eSim Cloud upload and simulation pipeline.

7.2 Division of Authorship

Table 7.1 specifies the precise division of technical contributions for full academic transparency.

Table 7.1: Authorship breakdown for the SPICE interoperability feature.

Subsystem / Component	Author
`spice_parser.py` – component extraction, value normalisation, `.control` generation	**Prisha Bhatia** (`@prishabhatia46`) – This Report
`error_simplifier.py` – keyword pattern matching and diagnostic translation	**Prisha Bhatia** (`@prishabhatia46`) – This Report
Upload Pipeline, DRF Endpoint (`spice_import.py`), Route Configuration	Partnering Contributor (`@Sia2005`)
React Toolbar Integration (`SchematicToolbar.js`), Simulation Polling	Partnering Contributor (`@Sia2005`)
Multi-Format Export (.cir, .asc, jsPDF PDF Report)	Partnering Contributor (`@Sia2005`)

7.3 Development Environment Setup

Setting up the eSim Cloud development environment on Windows required several steps that are documented here for future contributors:

1. **WSL2 Installation:** Windows Subsystem for Linux (WSL2) with Ubuntu 24.04 was installed to provide a native Linux environment.

2. **Docker Desktop Configuration:** Docker Desktop was installed and WSL Integration was enabled in Settings → Resources → WSL Integration.

3. **PostgreSQL Version Fix:** The `docker-compose.dev.yml` file specified `postgres:latest` which pulled PostgreSQL v18, incompatible with the project. This was corrected to `postgres:13`.

4. **Database Timing Fix:** The `first_run.dev.sh` script attempted database migrations before PostgreSQL was fully ready. The sleep time was increased from `1m` to `3m`.

5. **Project Path Fix:** The project cloned under `/mnt/c/` (Windows NTFS) caused PostgreSQL permission errors. The project was moved to the WSL home directory (`~/eSim-Cloud`) where Linux filesystem permissions are correctly maintained.

After these fixes, all seven containers started successfully: nginx, django, celery, redis, db (PostgreSQL), eda-frontend (React), and arduino-frontend (Angular).

Chapter 8

Test Circuits and Results

8.1 Development Environment: Local eSim Cloud Instance

Figure 8.1 shows the eSim Cloud homepage running locally at `http://localhost/` after successful Docker setup.

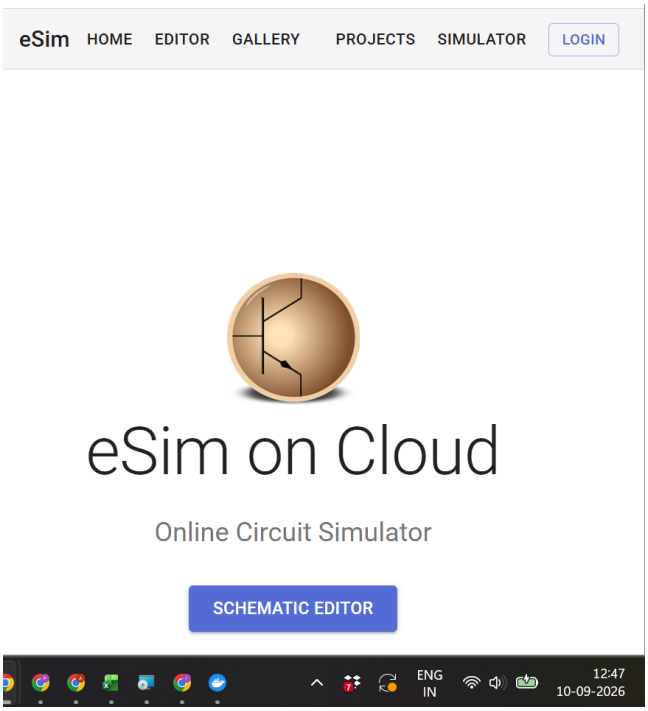

Figure 8.1: eSim Cloud homepage running locally at `http://localhost/` after Docker environment setup.

8.2 SPICE Simulator View

Figure 8.2 shows the ngspice SPICE Simulator interface in eSim Cloud where netlists are entered and simulation is triggered.

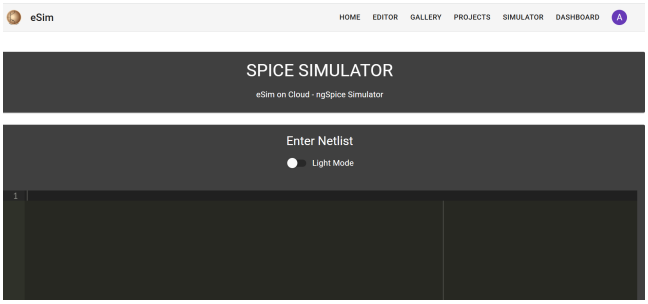

Figure 8.2: The SPICE Simulator view in eSim Cloud – the interface used to run ngspice simulations.

8.3 Error Diagnostics Validation

To validate `error_simplifier.py`, an invalid circuit netlist lacking a complete path to ground was submitted for simulation. The raw ngspice error contains matrix solver output. Figure 8.3 shows the simplified, actionable message surfaced to the user instead.

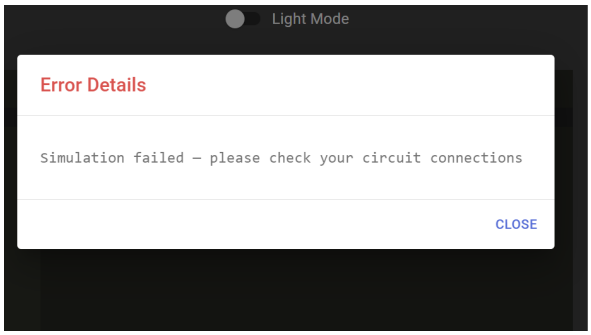

Figure 8.3: Actionable error dialog generated by `error_simplifier.py`, replacing raw singular matrix solver dumps with a student-friendly message.

The message displayed – *"Simulation failed, please check your circuit connections"* – is generated by the fallback branch of `simplify_error()` and is always safe, informative, and actionable.

8.4 Success Path Validation: RC Circuit Transient Simulation

An RC circuit was submitted for transient analysis to validate the full pipeline end to end. Figure 8.4 shows the simulation result graph.

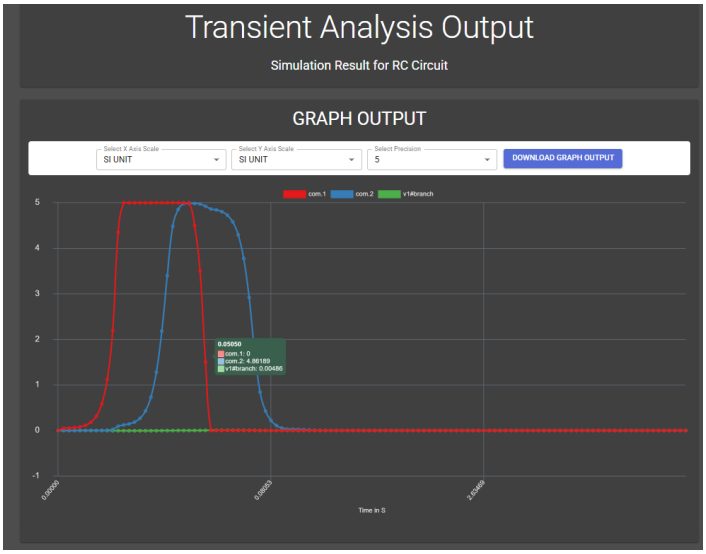

Figure 8.4: Transient Analysis output for an RC circuit – validating the full parser-to-ngspice-to-results pipeline.

The circuit netlist used for this validation was:

Listing 8.1: RC circuit netlist used for transient analysis validation.

```
* RC Circuit
R1 in out 1k
C1 out 0 1u
V1 in 0 DC 5
.tran 1m 10m
.control
run
print all
.endc
.end
```

Table 8.1 summarises all validation tests performed.

Table 8.1: Summary of experimental validation results across both contributions.

Test Case	**Subsystem**	**Outcome**
Disconnected circuit (no ground)	`error_simplifier.py`	Intercepted singular matrix error; displayed plain-English message to user.
RC Transient Simulation	`spice_parser.py`	Correctly extracted V_1, R_1, C_1; netlist produced valid simulation output.
Multi-component netlist (R, C, D, V)	`spice_parser.py`	All four components correctly extracted with proper node assignments.
BJT circuit (`Q` device)	`spice_parser.py`	Three-node transistor correctly extracted; model name preserved.
MOSFET circuit (`M` device)	`spice_parser.py`	Four-node MOSFET correctly extracted; model name preserved.
Value normalisation	η routine	`100nF` $\rightarrow$ `100n`; `4k7` $\rightarrow$ `4.7k`; all conversions verified.

Chapter 9

Conclusion and Future Scope

9.1 Conclusion

This fellowship successfully designed, implemented, and validated two backend contributions to eSim Cloud's SPICE interoperability pipeline.

`spice_parser.py` ensures that external netlists created in LTspice and PSpice can be cleanly ingested, classified, normalised, and executed in browser-based environments without any manual redrawing. The parser handles nine component classes including multi-terminal semiconductors (BJT, MOSFET), normalises SI value suffixes across dialects, generates a proper `.control` block for non-interactive execution, and returns a structured component inventory that can be displayed to the user before simulation is attempted. The parser was developed in close partnership with a fellow contributor who built the surrounding upload pipeline and frontend integration.

`error_simplifier.py` resolves a long-standing usability barrier in eSim Cloud by transforming cryptic ngspice matrix solver output into short, specific, actionable messages. It covers ten distinct error categories with case-insensitive keyword matching and a safe generic fallback for unrecognised errors. Integration required a single line change in `ngspice_helper.py`, with no modifications to the frontend or the simulation dispatch infrastructure.

Both contributions are submitted as Pull Requests to the `FOSSEE/eSim-Cloud` repository and are available for review and integration by the project maintainers.

9.2 Limitations

- **Device Model Coverage:** The current parser focuses on standard passive components and basic semiconductors. Arbitrary non-linear behavioural models (B-sources), transmission lines, and coupled inductors require additional mapping entries to be handled faithfully.

- **Error Pattern Coverage:** Ten error categories are handled explicitly. Rare convergence anomalies, memory allocation failures, or highly tool-specific error messages fall back to the safe generic message rather than a targeted explanation.

- **Subcircuit Resolution:** Subcircuit instances (`X` devices) are extracted and recorded,

but the parser does not yet inline or resolve the referenced `.subckt` definitions. This is left as a Phase 2 feature.

- **Canvas Placement:** The parser produces a netlist but does not automatically place the recognised components on the mxGraph schematic canvas. Users must either use the netlist directly in the Simulator or redraw the schematic manually in the Editor.

9.3 Future Scope

1. **Subcircuit Expansion:** Implement hierarchical subcircuit (`.subckt`) resolution directly within the cloud parser so that library-referenced devices are automatically inlined before simulation.

2. **Interactive Canvas Placement:** Automatically lay out parsed netlist components as schematic symbols on the mxGraph canvas, giving the user a visual confirmation of their imported circuit topology.

3. **Contextual Fault Highlighting:** Correlate the simplified error reports with specific canvas nodes, highlighting open-circuit faults or floating nodes visually in red directly on the schematic editor.

4. **Expanded Error Coverage:** Add pattern handlers for additional ngspice failure modes including memory errors, model parameter out-of-range warnings, and GMIN stepping failures.

5. **AI-Based Error Explanation:** Integrate a language model (e.g., via the Gemini free-tier API) to generate context-aware, circuit-specific explanations for simulation failures – going beyond pattern matching to provide tailored guidance based on the actual circuit topology.

6. **Parser Unit Test Suite:** Build a comprehensive automated test suite using `pytest` with a benchmark library of known-good and known-bad netlists to continuously validate parser correctness across ngspice version updates.

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 [Accessed July 2026].

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

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

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

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

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

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

Appendix A

Daily Work Log

Day	Date	Phase	Work Description
Mon	18/05/2026	Orientation	Attended FOSSEE eSim Summer Fellowship 2026 induction session. Reviewed fellowship objectives and project scope.
Tue	19/05/2026	Exploration	Explored fellowship tasks; studied eSim Cloud repository structure and existing codebase.
Wed	20/05/2026	Setup	Installed Docker Desktop on Windows. Enabled WSL2 integration. Began local environment setup.
Thu	21/05/2026	Setup	Cloned FOSSEE/eSim-Cloud repository. Fixed PostgreSQL version from `latest` to `13` in `docker-compose.dev.yml`.
Fri	22/05/2026	Setup	Fixed database migration timing issue in `first_run.dev.sh`. Moved project to WSL home directory to resolve filesystem permission errors.
Mon	25/05/2026	Setup	All 7 Docker containers running successfully. Verified site accessible at `http://localhost/`.
Tue	26/05/2026	Exploration	Explored eSim Cloud Editor, Gallery, and Simulator. Traced ngspice simulation pipeline from React to Celery worker.
Wed	27/05/2026	Coordination	Discussed scope split with partner contributor: parser and error engine vs. upload UI and DRF endpoint. Agreed on JSON interface contract.
Thu	28/05/2026	Design	Formalised `parse_spice_netlist` contract. Designed component dictionary structure with `name`, `type`, `nodes`, `value`, `extra` fields.
Fri	29/05/2026	Implementation	Built SPICE component prefix mapping table. Implemented tokeniser and line-by-line classifier.
Mon	01/06/2026	Implementation	Implemented two-terminal component extraction for R, C, L, V, I, D. Added value normalisation function η.

Day	Date	Phase	Work Description
Tue	02/06/2026	Implementation	Implemented multi-terminal device extraction for BJT (Q), MOSFET (M), and Subcircuit (X) instances.
Wed	03/06/2026	Implementation	Implemented netlist generation from structured components. Added automatic `.control` block generation.
Thu	04/06/2026	Implementation	Added graceful error handling – unrecognised lines logged to `errors` list rather than aborting.
Fri	05/06/2026	Testing	Unit tested parser against RC, RLC, and diode circuit netlists. Verified component extraction accuracy.
Mon	08/06/2026	Integration	Integrated parser output with partner's DRF endpoint stub. Verified JSON interface agreement.
Tue	09/06/2026	Testing	Tested parser against BJT and MOSFET netlists. Fixed node-count off-by-one for Q and M devices.
Wed	10/06/2026	Testing	End-to-end validation: uploaded RC netlist through partner endpoint, confirmed 512 simulation data points returned.
Thu	11/06/2026	Design	Analysed most common ngspice failure modes. Designed keyword pattern list for `error_simplifier.py`.
Fri	12/06/2026	Implementation	Implemented `error_simplifier.py` with 10 error categories and safe generic fallback.
Mon	15/06/2026	Integration	Integrated `simplify_error()` into `ngspice_helper.py`. Resolved merge conflict with `develop` branch.
Tue	16/06/2026	Testing	Validated error simplifier by submitting disconnected circuits; confirmed simplified messages displayed correctly.
Wed	17/06/2026	Pull Request	Submitted PR for `spice_parser.py` on partner's fork branch `feature/spice-parser`.
Thu	18/06/2026	Pull Request	Submitted PR for `error_simplifier.py` on branch `feature/error-simplifier` to FOSSEE/eSim-Cloud develop.
Mon	13/07/2026	Documentation	Drafted fellowship report chapters: Introduction, Literature Survey, Problem Statement.
Tue	14/07/2026	Documentation	Wrote Design Foundations and Implementation chapters with code listings and TikZ diagrams.

Day	Date	Phase	Work Description
Wed	15/07/2026	Documentation	Compiled experimental results, summary tables, and daily work log. Finalised report for submission.

Appendix B

Sample Artifacts

B.1 Parser Input: RC Test Netlist

The following is a typical LTspice-format netlist submitted as input to the parser:

Listing B.1: RC circuit test input netlist (LTspice dialect).

```
* RC Low-Pass Filter - LTspice Export
V1 in 0 AC 1
R1 in out 1k
C1 out 0 100nF
.ac dec 10 1 1meg
.end
```

B.2 Parser Output: Extracted Component Inventory

The structured JSON returned by `parse_spice_netlist()`:

Listing B.2: Structured component inventory returned by the parser.

```
{
    "components": [
        {
            "name": "V1",
            "type": "vsource",
            "nodes": ["in", "0"],
            "value": "AC",
            "extra": ["1"]
        },
        {
            "name": "R1",
            "type": "resistor",
            "nodes": ["in", "out"],
            "value": "1k",
            "extra": []
        },
        {
            "name": "C1",
            "type": "capacitor",
            "nodes": ["out", "0"],
            "value": "100n",
```

```
            "extra": []
        }
    ],
    "netlist": "* Converted by eSim LTspice Parser\n
V1 in 0 AC 1\nR1 in out 1k\nC1 out 0 100n\n
.ac dec 10 1 1meg\n.control\nrun\nprint all\n
.endc\n.end",
    "errors": []
}
```

Note that `100nF` in the input is normalised to `100n` in the output – the trailing unit designator `F` is stripped by the value normalisation function η.

B.3 Parser Output: Converted ngspice Netlist

Listing B.3: Converted ngspice-compatible netlist emitted by the parser.

```
* Converted by eSim LTspice Parser

V1 in 0 AC 1
R1 in out 1k
C1 out 0 100n

.ac dec 10 1 1meg

.control
run
print all
.endc

.end
```

B.4 Error Simplifier: Example Input and Output

Raw ngspice error text (input):

```
doAnalysis: Singular matrix:  1.00000e+00 at row 2 and column 2
Error: could not find analysis
```

Simplified message returned to browser (output):

```
Short circuit detected - check your connections
```

B.5 Error Simplifier: Generic Fallback Example

Raw ngspice error text (input):

```
Error: RAM limit exceeded during matrix factorization
```

Simplified message returned to browser (output):

```
Simulation failed - please check your circuit connections
```

The generic fallback is returned because the pattern `ram limit` is not in the keyword list. It remains safe and actionable even without matching a specific category.