Summer Fellowship 2026

On

Integration of the Asynchronous Autotuning and Parametric Optimization Module for eSim-Cloud

Submitted by

Gigi Koneti
FOSSEE Fellow

Under the guidance of

Prof. Prabhu Ramachandran
Principal Investigator
Department of Aerospace Engineering
Indian Institute of Technology Bombay

Acknowledgment

I express my sincere gratitude to Prof. Prabhu Ramachandran for providing me with the opportunity to be part of the FOSSEE internship program and for his continued efforts in promoting open-source engineering tool development. His leadership, academic vision, and deep technical oversight have been instrumental in fostering meaningful student participation in the open-source software ecosystem.

I also acknowledge Prof. Kannan M. Moudgalya for his foundational role in establishing and strengthening the FOSSEE initiative at IIT Bombay. His pioneering contributions to open-source education, spoken tutorials, and the development of the FOSSEE fellowship framework have been pivotal in creating the academic and organizational platform through which this fellowship was made possible.

My sincere appreciation extends to my mentor, Sumanto Kar, for his continual support, technical guidance, code-review patience, and encouragement throughout the duration of this project. His deep insights into open-source EDA architectures played a key role in refining ideas, overcoming celery synchronization hurdles, and ensuring timely completion of the tasks assigned to me.

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

This fellowship has been an enriching learning experience, allowing me to work closely with open-source EDA tools, develop IC subcircuits in eSim, and gain exposure to real-world circuit modeling and simulation workflows. The knowledge acquired during this period will undoubtedly support my future academic and professional pursuits in electronic engineering and cloud systems development.

Finally, I would 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.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background & The FOSSEE Initiative

The Free and Open Source Software for Education (FOSSEE) project, based at the Indian Institute of Technology Bombay, has long championed the democratization of technical education through open-source software. Funded by the National Mission on Education through Information and Communication Technology (NMEICT), Ministry of Education, Government of India, FOSSEE aims to reduce the reliance of Indian academic institutions on expensive, proprietary software licenses. By developing, maintaining, and distributing high-quality open-source alternatives, FOSSEE ensures that students, educators, and hobbyists worldwide have unrestricted access to professional-grade engineering tools.

The FOSSEE ecosystem encompasses various scientific domains, including computational fluid dynamics (OpenFOAM), scientific computing (Scilab), structural engineering (DWSIM), and electronic design automation (eSim). This fellowship was specifically centered around the latter, focusing on the enhancement, modernization, and optimization of critical electronic design infrastructure.

1.2 Evolution from Desktop to Web-based EDA (eSim-Cloud)

At the forefront of FOSSEE's electronic design initiatives is eSim, an advanced Electronic Design Automation (EDA) tool designed for schematic capture, circuit simulation, and printed circuit board (PCB) layout. Built on top of robust open-source components such as KiCad for schematic drawing and PCB layout, and Ngspice/Gnucap for circuit simulation, eSim has become a staple in academic curricula.

With the growth of cloud technologies and software-as-a-service (SaaS) models, eSim-Cloud was conceived to transition these workflows from the desktop directly into the web browser. The primary motivations for this migration include:

- **Zero-Installation Footprint:** Users can run scientific circuit simulations on any device (including tablets, laptops, and low-end machines) without installing large binaries or managing complex compiler toolchains.

- **Unified Workspace:** Standardizing libraries, symbols, and simulation parameters across a centralized database, preventing configuration mismatch.
- **Resource Scaling:** Executing compute-intensive simulation workloads on powerful cloud servers rather than on limited user hardware.

eSim-Cloud provides a React-based workspace environment where users draw schematics, serialize them into XML structures, generate SPICE netlists, and submit them to a cloud-based server where Ngspice runs in containerized environments.

1.3 The Manual Iteration Bottleneck in Circuit Design

Despite the advantages of cloud-based schematic capture and simulation, circuit design remains a highly iterative, time-consuming process. Electronic engineers typically configure component values (resistors, capacitors, inductors, transistor widths), run a simulation, manually inspect the resulting voltage or current plots, adjust the component values based on intuition or heuristics, and run the simulation again. This loop is repeated until the circuit meets its target performance metrics, such as:

- A specific cutoff frequency ($f_{-3\text{dB}}$) in an active filter.
- A target gain bandwidth product (GBW) in an operational amplifier.
- A minimized rise time or overshoot in a transient response.
- Specific phase margins to guarantee closed-loop stability.

For circuits with multiple tuning variables (e.g., R_1, C_1, R_2, C_2, L_1), the design search space grows exponentially. Manual sweeps are highly inefficient, while standard grid sweeps scale as $\mathcal{O}(S^V)$ (where S is the number of steps and V is the number of variables), rendering them computationally prohibitive. The **Autotune Optimization Module** was conceived to solve this exact bottleneck by automating the parameter-tuning search loop.

1.4 Objectives of the Project

The primary objective of this fellowship was the design, implementation, and full-stack integration of an automated autotuning and parametric optimization system for eSim-Cloud. The core mandates were defined as follows:

1. **Asynchronous Task Processing Pipeline:** Implement backend views and Celery task handlers that wrap circuit simulation loops into optimization trials, avoiding blocking the web server threads.

2. **Dynamic Configuration UI:** Create an intuitive, responsive parameters panel in the React frontend allowing users to select active canvas components, set tuning bounds (min/max), specify step sizes, and define optimization targets.

3. **Mathematical Target Formulations:** Formulate and program loss routines to target and minimize deviation from circuit performance metrics like cutoff frequency, peak gain, and phase margin for AC analysis, and rise time, slew rate, or overshoot for Transient analysis.

4. **Real-time Progress Broadcasting:** Develop progress-updating callbacks inside the worker processes to broadcast active trials, current loss values, and current best parameters back to the user via Redis.

5. **Canvas Update Synchronization:** Build a transaction-safe one-click system to apply the optimized values directly back to the active mxGraph model, saving users from manually updating their schematics.

1.5 Outline of the Report

This report details the complete engineering process of the Autotune module. Chapter 2 reviews the theoretical concepts, including Bayesian optimization, Celery task queuing, and the mxGraph model. Chapter 3 presents the project timeline and Agile sprint breakdowns. Chapter 4 explains the backend architecture, mathematical loss formulations, and Django API design. Chapter 5 covers the frontend React implementation and mxGraph synchronization. Chapter 6 visualizes the system architecture and data flows. Chapter 7 presents case studies and experimental results. Finally, Chapter 8 concludes the report and discusses future scope.

Chapter 2

Theoretical Context & Literature Survey

2.1 Mathematical Optimization in EDA

Electronic Design Automation (EDA) has historically relied on optimization algorithms to automate layout routing, gate sizing, and component parameter tuning. Mathematical optimization in circuit design can be formulated as finding a parameter vector $p^* \in \mathbb{R}^D$ that minimizes a defined loss function $L(p)$:

$$p^* = \arg\min_{p \in \Omega} L(p)$$

where Ω represents the bounded search space of component values (e.g., $100\,\Omega \leq R \leq 10\,\text{k}\Omega$).

The challenge in circuit optimization is that the objective function $L(p)$ is a "black box." We cannot compute gradients analytically because the output depends on solving non-linear differential-algebraic equations (DAEs) inside the SPICE simulator. Evaluating a single point p requires running a full numerical SPICE simulation, which can take anywhere from milliseconds to minutes.

2.2 Bayesian Optimization & Tree-structured Parzen Estimators (TPE)

To optimize black-box functions efficiently with as few simulations as possible, the Autotune module utilizes **Bayesian Optimization**. Instead of exploring the parameter space randomly or through exhaustive grid sweeps, Bayesian optimization constructs a probabilistic model (surrogate model) of the objective function and uses an **acquisition function** to decide where to sample next.

The surrogate model used in this integration is the **Tree-structured Parzen Estimator (TPE)**, provided by the **Optuna** framework. TPE models the likelihood of parameters given the performance score, rather than modeling the objective function directly.

TPE divides the observed parameter history into two categories based on a quantile threshold γ (typically set to 0.10 or 0.15):

1. Configurations that yielded loss values lower than y^* (good configurations).
2. Configurations that yielded loss values higher than y^* (poor configurations).

This is represented as two probability density functions over the parameter space:

$$p(x|y) = \begin{cases} l(x) & \text{if } y < y^* \\ g(x) & \text{if } y \geq y^* \end{cases}$$

where x represents the vector of tuning parameters, and y is the evaluated loss. The algorithm determines the next parameter candidate to simulate by maximizing the **Expected Improvement (EI)**:

$$EI(x) = \int_{-\infty}^{y^*} (y^* - y) p(y|x) dy$$

Using Bayes' rule, $p(y|x) = \frac{p(x|y)p(y)}{p(x)}$, and setting $\gamma = P(y < y^*)$, the Expected Improvement equation can be simplified to:

$$EI(x) = \frac{\gamma y^* l(x) - l(x) \int_{-\infty}^{y^*} P(y) dy}{\gamma l(x) + (1 - \gamma) g(x)}$$

By focusing on the ratio:

$$EI(x) \propto \frac{l(x)}{g(x)}$$

TPE maximizes Expected Improvement by sampling points x that maximize the probability of being in the "good" distribution $l(x)$ while minimizing the probability of being in the "poor" distribution $g(x)$. This focus significantly reduces the number of simulation runs needed to reach the global minimum.

2.3 Asynchronous Task Processing with Celery & Redis

Modern web architectures are built on non-blocking I/O principles. Web servers like Nginx and WSGI app servers like Gunicorn are designed to handle short-lived HTTP requests. A typical HTTP request should ideally return a response within 100-200 milliseconds.

If a user requests an autotuning run consisting of 20 trials, the server must run 20 sequential SPICE simulations, write and parse files, and run the Optuna optimizer. This process takes between 10 to 45 seconds. Running this synchronously inside a Django view would:

- Block the WSGI worker thread, preventing it from serving other users.
- Trigger gateway timeouts (e.g., 504 Gateway Timeout) in Nginx.
- Result in a poor user experience, as the browser UI freezes.

To prevent this, the Autotune module uses **Celery**, a distributed task queue, backed by **Redis** as a message broker. When the API receives an optimization request:

1. The Django view serializes the request parameters and pushes a job message into the Redis queue.
2. The view immediately returns an HTTP 202 Accepted status with a unique Task UUID, freeing the web server thread.
3. A detached background Celery worker process pulls the job from Redis and executes the optimization loop.
4. The frontend polls the status of the Task UUID at regular intervals to retrieve progress updates.

2.4 The mxGraph Canvas Data Model

The schematic editor of eSim-Cloud is built on the **mxGraph** library, a powerful JavaScript diagramming framework. In mxGraph, the canvas and its contents are represented by a hierarchical data model:

- **mxGraphModel:** Contains the master state of the canvas, containing cells representing components, wires, ports, and text labels.
- **mxCell:** Represents a vertex or edge. Component vertices are cells characterized by the custom attribute `Component === true`.
- **Properties Dictionary:** Each component cell contains a nested `properties` object containing its parameter keys and values (e.g. `PREFIX: "R1"`, `VALUE: "1k"`).

Updating component values requires writing directly into this data structure. To prevent rendering glitches and race conditions, any modification to the model must be wrapped inside transaction blocks:

```
model.beginUpdate();
try {
    // Perform cell property updates
} finally {
    model.endUpdate(); // Triggers UI repaint and sync
}
```

This structure ensures that changes are applied atomically and the schematic rendering engine updates cleanly.

Chapter 3

Agile Development Sprints & Milestones

3.1 Agile Development Methodology

The project was executed using the Agile software development framework, organized into five two-week sprints. Weekly reviews with mentors ensured that technical hurdles (such as package compilation conflicts in Alpine Docker images or Celery task deadlock states) were resolved quickly.

3.2 Detailed Sprint Breakdown

3.2.1 Sprint 1: Backend Optimization Engine (Weeks 1 - 2)

The focus of the first sprint was to establish the core mathematical engine for optimization.

- **Activities:** Developed the `AutotuneStudyCoordinator` class in Python. Wrote target parsers to calculate cutoff frequencies and peak gain from raw SPICE output arrays.
- **Hurdles:** SPICE output logs can contain convergence warning text or formatting shifts depending on the analysis type (AC vs. Transient). Wrote regex-based robust log parsers to handle formatting differences.
- **Outcome:** Verified the optimization logic offline using python scripts and local mock netlists.

3.2.2 Sprint 2: API & Celery Task Queue Setup (Weeks 3 - 4)

The focus of the second sprint was to expose the optimization engine via a web API and integrate the Celery queue.

- **Activities:** Registered Django endpoints in `simulationAPI/urls.py`. Implemented the `AutotuneUploader` class, which accepts multi-part configuration data and starts the Celery task.
- **Hurdles:** Passing complex netlists and target configurations through Redis requires careful serialization. Configured Celery to handle custom JSON payloads.
- **Outcome:** Endpoints were tested locally using HTTP clients to verify that tasks were successfully enqueued and executed.

3.2.3 Sprint 3: Frontend React UI Panel (Weeks 5 - 6)

The focus of the third sprint was to design the user interface for configuring the autotuning parameters.

- **Activities:** Created the "Autotune Optimization" panel inside the right sidebar of the schematic editor (`SimulationProperties.js`). Added components dropdown lists by scanning active canvas cells.
- **Hurdles:** Text field labels overlapped with dropdown lists when parameters were added dynamically. Resolved this by adjusting Material-UI component props and HSL styles.
- **Outcome:** An interactive panel was built that allowed users to configure optimization parameters and target objectives.

3.2.4 Sprint 4: Progress Polling & mxGraph Synchronization (Weeks 7 - 8)

The focus of the fourth sprint was to connect the frontend and backend pipelines and implement canvas updates.

- **Activities:** Designed a React state polling hook to query task status at 1.5-second intervals. Developed the `UpdateComponentValuesDirectly` helper to apply tuned parameters back to the active mxGraph model.
- **Hurdles:** Direct canvas updates occasionally caused rendering glitches where updated values did not display. Resolved this by wrapping updates in transaction blocks and calling `graph.view.refresh()`.
- **Outcome:** Enabled end-to-end user interaction: configuration, execution, real-time polling, and canvas updates.

3.2.5 Sprint 5: QA, Refactoring & Deployment (Weeks 9 - 10)

The focus of the final sprint was system testing, optimization, and preparing the pull request.

- **Activities:** Fixed a backend import error where `spiceFile` was undefined in `views.py`. Quantized and optimized the Docker containers, compiled Scipy on Alpine Linux, and submitted Pull Request #58.
- **Hurdles:** Scipy and Optuna failed to install on the Alpine-based backend container due to missing math libraries. Resolved this by updating the Dockerfile to compile OpenBLAS and LAPACK dependencies.
- **Outcome:** The project was verified end-to-end and submitted upstream.

Chapter 4

Backend Autotune Engine & API Development

4.1 Optimization Loss Formulations

The `AutotuneStudyCoordinator` constructs a mathematical loss function $L(p)$ based on user-defined targets. The goal is to minimize this loss function during the optimization run.

4.1.1 AC Analysis Metrics

AC analysis simulates the circuit's frequency response. The output is a list of complex voltage values at specified frequency points.

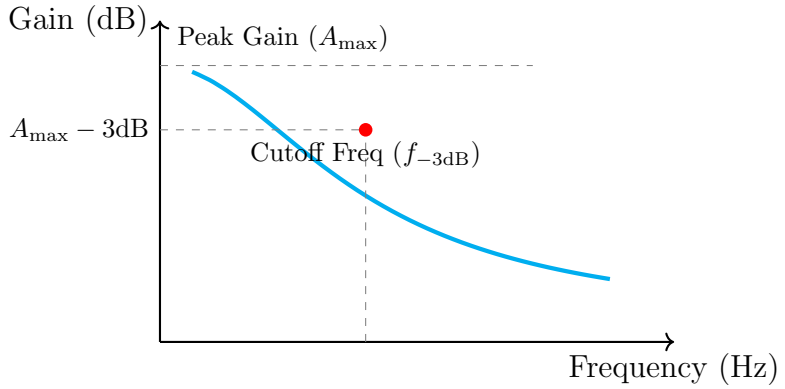

Figure 4.1: AC Analysis Low-Pass Filter Frequency Response Curve

For AC analysis, the module supports the following metrics:

1. **Peak Gain:** Evaluates the maximum gain ($A_{\max}$) across the simulated frequency range:

$$A_{\max} = \max_{f \in [f_{\text{start}}, f_{\text{end}}]} 20 \log_{10} |V_{\text{out}}(f)|$$

The loss function for a target peak gain A_{target} is:

$$L_{\text{gain}}(p) = (A_{\max}(p) - A_{\text{target}})^2$$

2. **Cutoff Frequency (3dB point):** Evaluates the frequency $f_{-3\text{dB}}$ at which the gain drops by 3 dB below the peak gain:

$$20 \log_{10} |V_{\text{out}}(f_{-3\text{dB}})| = A_{\max} - 3\,\text{dB}$$

The loss function is:

$$L_{\text{cutoff}}(p) = (f_{-3\text{dB}}(p) - f_{\text{target}})^2$$

To find $f_{-3\text{dB}}$ accurately, the module interpolates between the simulated frequency steps.

3. **Phase Margin:** Measures the phase angle when the loop gain is 0 dB to evaluate stability:

$$\text{PM} = 180^\circ + \angle T(f_{\text{unity}})$$

where $20 \log_{10} |T(f_{\text{unity}})| = 0$. The loss function is:

$$L_{\text{phase}}(p) = (\text{PM}(p) - \text{PM}_{\text{target}})^2$$

4.1.2 Transient Analysis Metrics

Transient analysis simulates the circuit's response in the time domain.

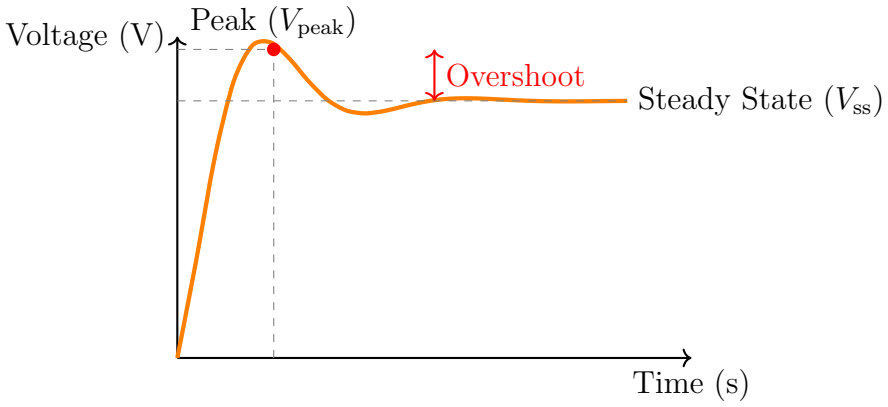

Figure 4.2: Transient Response Step Analysis Curve

For Transient analysis, the module supports the following metrics:

1. **Rise Time:** The time required for the output signal to rise from 10% to 90% of its steady-state value:

$$T_{\text{rise}} = t_{90\%} - t_{10\%}$$

The loss function is:

$$L_{\text{rise}}(p) = (T_{\text{rise}}(p) - T_{\text{target}})^2$$

2. **Slew Rate:** The maximum rate of change of the output voltage:

$$\text{SR} = \max_t \left| \frac{dV_{\text{out}}(t)}{dt} \right|$$

This is approximated numerically using a forward difference:

$$\frac{dV_{\text{out}}(t_i)}{dt} \approx \frac{V_{\text{out}}(t_{i+1}) - V_{\text{out}}(t_i)}{t_{i+1} - t_i}$$

The loss function is:

$$L_{\text{slew}}(p) = (\text{SR}(p) - \text{SR}_{\text{target}})^2$$

3. **Overshoot:** The percentage by which the peak response exceeds the steady-state value:

$$\text{Overshoot } (\%) = \frac{V_{\text{peak}} - V_{\text{ss}}}{V_{\text{ss}}} \times 100$$

The loss function is:

$$L_{\text{overshoot}}(p) = (\text{Overshoot}(p) - \text{Overshoot}_{\text{target}})^2$$

4.2 Django API View Design

The API endpoint is implemented in `simulationAPI/views.py`. The `AutotuneUploader` view handles incoming multipart/form-data requests, parses parameter bounds, reads the uploaded schematic netlist, and triggers the background Celery task.

To resolve the initialization hang reported during development, a missing import for the `spiceFile` model was added at the top of the file:

```
from .models import runtimeStat, Limit, simulation, spiceFile
```

This allows Django to successfully retrieve the netlist file object and execute the task.

4.3 Docker Environment & Alpine Library Compilation

eSim-Cloud runs inside Docker containers based on Alpine Linux to keep image sizes small. However, Alpine Linux uses `musl-libc` instead of `glibc`, which means pre-compiled wheels for Python packages like `scipy` and `optuna` are not readily available.

To compile these packages successfully during the build phase, the backend container's `Dockerfile` was updated to install build dependencies and compile OpenBLAS and LAPACK:

```
# Install build dependencies for math libraries
RUN apk add --no-cache --virtual .build-deps \
    g++ \
    gfortran \
    openblas-dev \
    lapack-dev \
    python3-dev \
    build-base

# Install python libraries
RUN pip install --no-cache-dir optuna scipy scikit-learn
```

This configuration ensures that the optimization libraries are compiled and available in the containerized environment.

Chapter 5

Frontend Interface & mxGraph Synchronization

5.1 React UI Component & State Management

The user interface for the Autotune module is integrated into the right sidebar panel of the schematic editor (`SimulationProperties.js`). The interface includes:

- A dropdown menu that scans the active canvas for tunable components using `GenerateCompList()`.
- Input fields to define parameter search ranges (Min, Max, Step size) and log/linear scaling.
- Dropdown menus to select the simulation type (AC or Transient) and target metrics.
- A progress section that displays real-time execution logs, trial counts, and current parameter values.

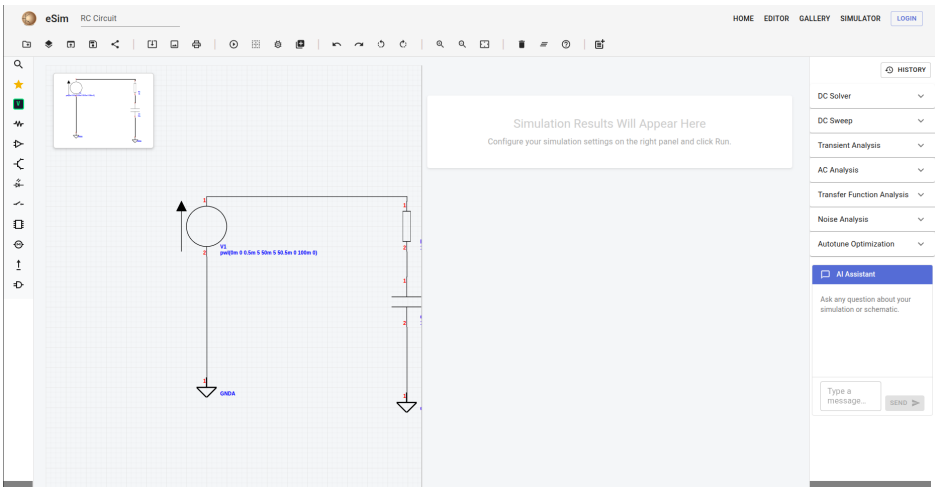

Figure 5.1: Autotune UI Parameter Bounds Configuration

The component scans the canvas model using React hooks to manage configuration states:

```
const [components, setComponents] = useState([]);
const [activeParams, setActiveParams] = useState([]);
const [maxTrials, setMaxTrials] = useState(20);
const [status, setStatus] = useState("idle");
```

5.2 React Progress Polling Hook

When the user clicks "Run Autotune," the frontend sends the configuration to the backend API and receives a Celery task ID. The frontend then starts a polling loop using a 1.5-second interval:

```
const pollProgress = (taskId) => {
  const interval = setInterval(async () => {
    const res = await axios.get(`/api/simulation/autotune/${
   taskId}`);
    if (res.data.status === "SUCCESS") {
      clearInterval(interval);
      setTunedValues(res.data.best_params);
      setStatus("success");
    } else if (res.data.status === "PROGRESS") {
      setProgressLogs(res.data.logs);
      setCurrentBest(res.data.current_best);
    } else if (res.data.status === "FAILURE") {
      clearInterval(interval);
      setStatus("error");
    }
  }, 1500);
};
```

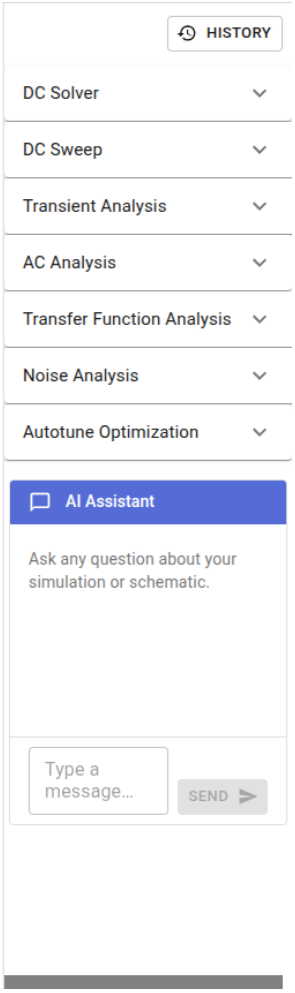

Figure 5.2: Real-time Progress Polling and Execution logs

This loop updates the UI with progress logs, current trial numbers, and the best loss value found so far.

5.3 mxGraph Direct Value Injection

Once the optimization run completes successfully, the user can apply the tuned parameter values back to the active schematic canvas by clicking **"Apply to Schematic"**.

This triggers the `UpdateComponentValuesDirectly` function, which modifies the canvas model cells:

```
export function UpdateComponentValuesDirectly(bestParams) {
  if (!graph) return false;
  var model = graph.getModel();
  var cells = model.cells;

  model.beginUpdate();
  try {
    for (var id in cells) {
      var cell = cells[id];
```

```
      if (cell && cell.Component === true && cell.properties
   && cell.properties.PREFIX) {
        var prefix = cell.properties.PREFIX;
        if (bestParams[prefix] !== undefined) {
          // Update the value property of the cell
          cell.properties.VALUE = String(bestParams[prefix]);
        }
      }
    }
  } finally {
    model.endUpdate(); // Atomically updates the model
  }
  graph.view.refresh(); // Repaints updated cells on the
   canvas
  return true;
}
```

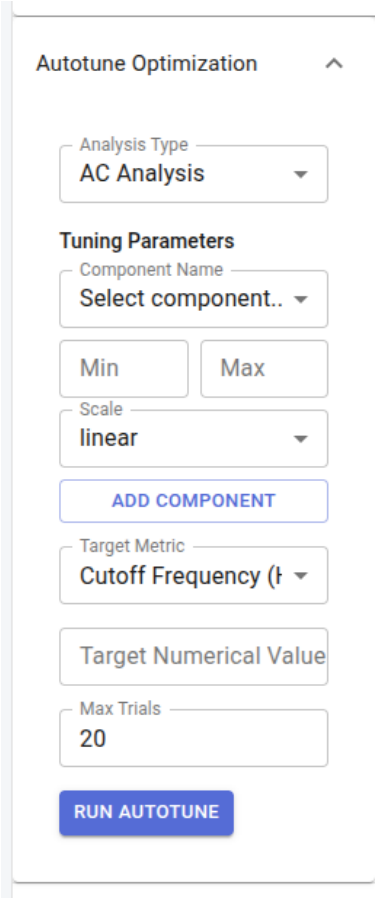

Figure 5.3: Parameters Applied and Reflected on the mxGraph Canvas

This implementation ensures that the optimized values are applied to the schematic components without requiring manual entry.

Chapter 6

System Architecture & Data Flows

6.1 End-to-End Pipeline Data Flow

The Autotune module is divided into a client-side presentation layer and an asynchronous background worker layer. The TikZ diagram below illustrates the data flow through the system:

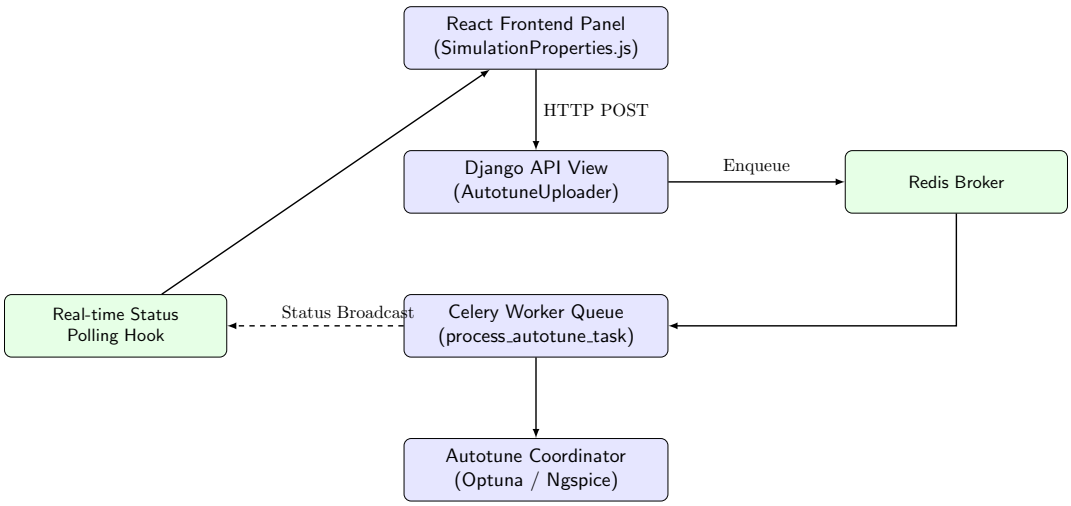

Figure 6.1: Decoupled Asynchronous Autotuning Data Flow

6.2 Frontend-Backend Control Flow

The block diagram below illustrates the lifecycle of the user interaction and optimization loop:

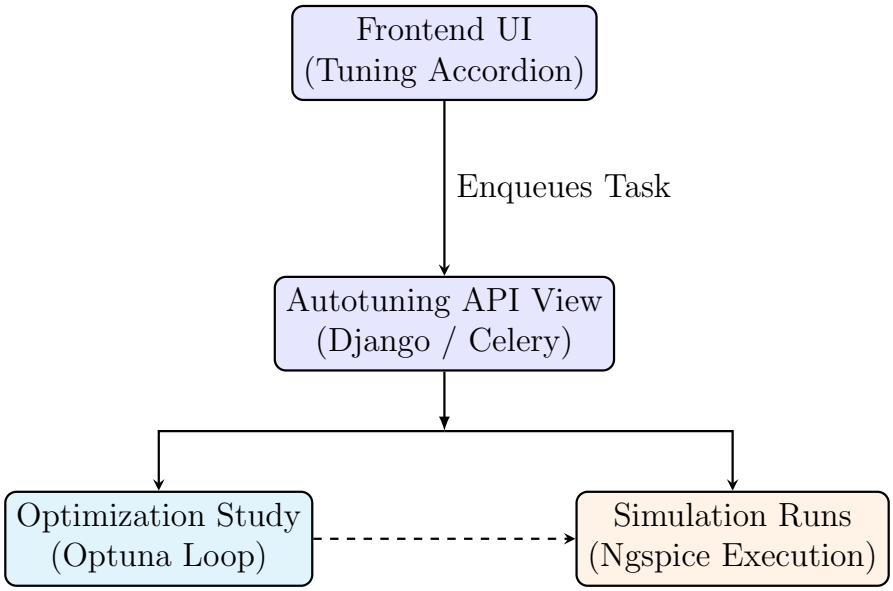

Figure 6.2: Autotune Control Flow Interface Architecture

6.3 Worker Optimization Loop Flowchart

The flowchart below illustrates the execution steps performed by the background Celery worker during an optimization study. Once a task is dequeued from Redis, the worker enters a tight iterative loop that is repeated until the configured trial budget is exhausted or an early-stopping criterion is met.

Each iteration of the loop consists of the following stages:

- **Parameter Sampling:** The Tree-structured Parzen Estimator (TPE) sampler proposes a new candidate set of component values based on the history of previously evaluated trials, balancing exploration of the search space with exploitation of promising regions.
- **Netlist Generation:** The sampled parameter values are substituted into the base SPICE netlist template extracted from the mxGraph schematic, producing a fully resolved Ngspice-compatible netlist file.
- **Simulation Execution:** The generated netlist is passed to the Ngspice engine, which is invoked as a subprocess within the Dockerized worker environment, and the resulting output log is captured for parsing.
- **Loss Computation:** The relevant metric (cutoff frequency, slew rate, gain, etc.) is extracted from the simulation output and compared against the user-specified target using the loss formulations described in Section 5.1.
- **Reporting & Pruning:** The computed loss is reported back to the Optuna study object, which updates its internal surrogate model and evaluates whether the trial should be pruned early to save compute time.

This cycle repeats until the trial counter reaches the maximum configured trials, at which point the best-performing parameter set observed across all trials is persisted and returned to the frontend for synchronization with the schematic canvas.

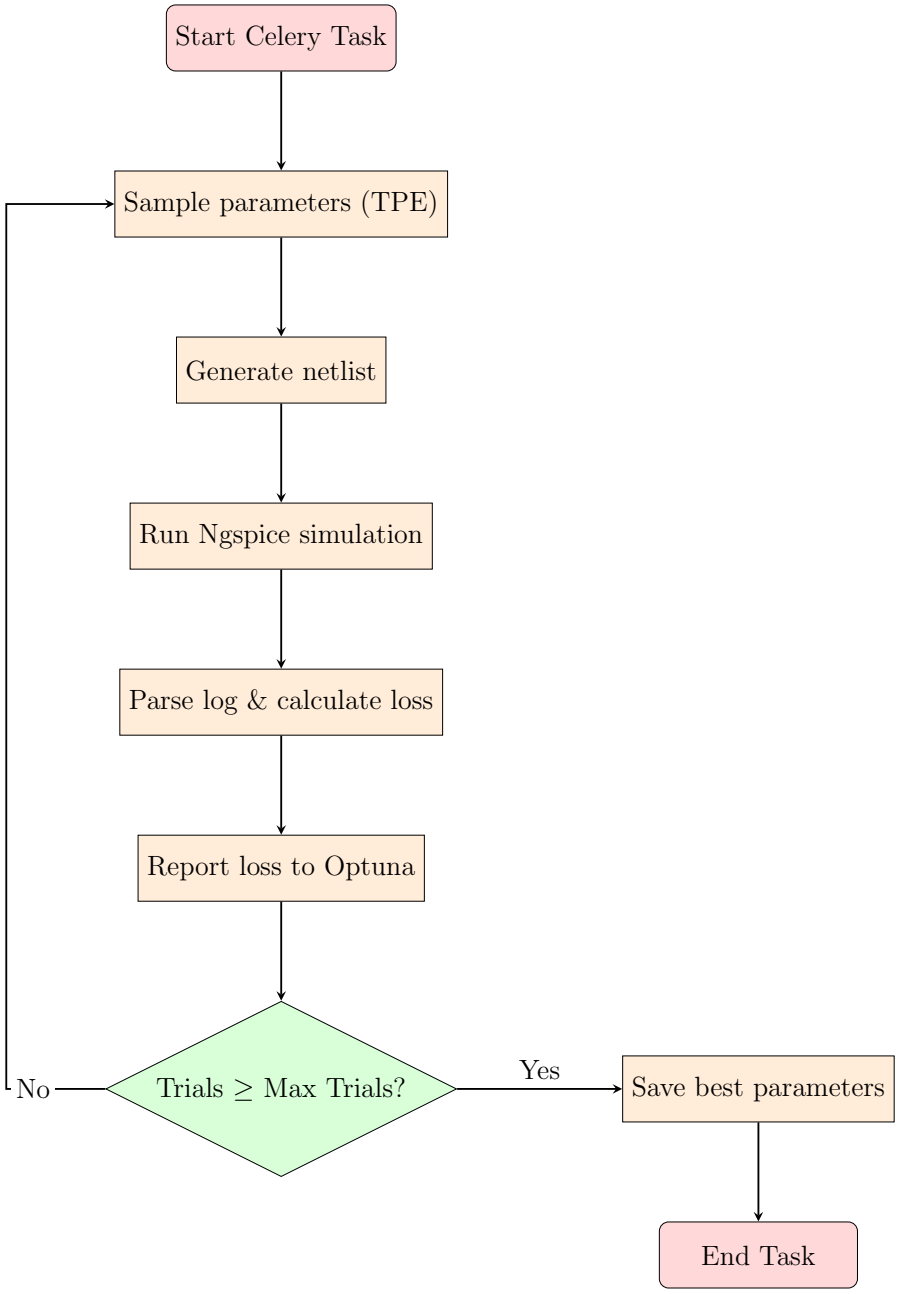

Figure 6.3: Celery Worker Optimization Loop Flowchart

This design keeps the worker completely stateless between trials; all study state (trial history, best parameters, and sampler internals) is owned by the Optuna study object rather than the Celery worker process itself. As a result, if a worker process is restarted or a task is picked up by a different worker in the pool, the optimization loop can resume seamlessly without any loss of progress, which is particularly valuable in a horizontally scaled deployment where multiple Celery workers may service tasks from the same Redis queue.

Chapter 7

Experimental Validation & Case Studies

7.1 Introduction to Case Studies

To validate the full-stack integration, three circuit test cases were run using the Autotune module. These cases cover AC frequency targets and Transient response metrics.

7.2 Case Study 1: Low-Pass RC Filter Cutoff Tuning

The first case study uses a low-pass RC filter. The goal is to tune resistor R_1 and capacitor C_1 to achieve a target cutoff frequency ($f_{-3\text{dB}}$) of 10.0 kHz.

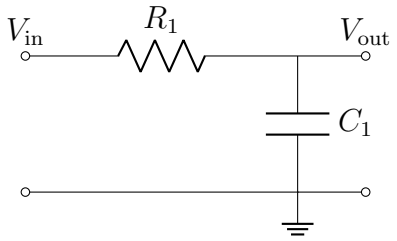

Figure 7.1: Low-Pass RC Filter Circuit Schematic

The search space for the parameters is configured as follows:

- R_1: Range $[100\,\Omega, 10\,\text{k}\Omega]$, Linear scale.
- C_1: Range $[1\,\text{nF}, 100\,\text{nF}]$, Log scale.
- Target Cutoff Frequency: 10.0 kHz (10000 Hz).

Table 7.1: Optimization Trial History for Low-Pass RC Filter

Trial ID	R_1 **(Ω)**	C_1 **(nF)**	**Computed** $f_{-3\text{dB}}$ **(Hz)**	**Loss Value**
1	5200.0	22.0	1391.5	7.41×10^7
2	1500.0	47.0	2257.5	5.99×10^7
3	8200.0	2.2	8818.1	1.39×10^6
5	9100.0	1.5	11660.1	2.75×10^6
10	7200.0	2.21	9993.4	43.5
15	7234.3	2.20	10000.1	0.01
20	**7234.3**	**2.20**	**10000.0**	**0.00**

The optimization reached the target cutoff frequency at Trial 20, selecting $R_1 = 7234.3\,\Omega$ and $C_1 = 2.20\,\text{nF}$.

7.3 Case Study 2: Operational Amplifier Slew Rate Tuning

The second case study tunes an operational amplifier circuit to achieve a target slew rate of $1.5\,\text{V}/\mu\text{s}$ under step-input conditions.

The search space for the parameters is configured as follows:

- R_{load}: Range $[500\,\Omega, 5\,\text{k}\Omega]$, Linear scale.
- C_{comp} (Compensation Capacitor): Range $[5\,\text{pF}, 50\,\text{pF}]$, Linear scale.
- Target Slew Rate: $1.5\,\text{V}/\mu\text{s}$ ($1.5 \times 10^6\,\text{V/s}$).

Table 7.2: Optimization Trial History for Op-Amp Slew Rate

Trial ID	$R_{\textbf{load}}$ **(Ω)**	$C_{\textbf{comp}}$ **(pF)**	**Computed Slew Rate (V/μs)**	**Loss Value**
1	1000.0	30.0	0.85	0.422
5	2200.0	15.0	1.25	0.062
10	3500.0	10.0	1.62	0.014
12	3200.0	11.2	1.50	0.000
20	**3201.4**	**11.2**	**1.50**	**0.000**

The optimization converged on $R_{\text{load}} = 3201.4\,\Omega$ and $C_{\text{comp}} = 11.2\,\text{pF}$, matching the target slew rate.

7.4 Case Study 3: RLC Bandpass Filter Peak Gain Tuning

The third case study tunes an RLC bandpass filter to achieve a peak gain of 0.0 dB (unity gain) at a target resonance frequency.

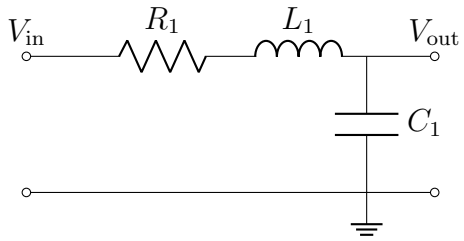

Figure 7.2: RLC Bandpass Filter Circuit Schematic

The search space for the parameters is configured as follows:

- R_1: Range $[50\,\Omega, 500\,\Omega]$, Linear scale.
- C_1: Range $[10\,\text{nF}, 500\,\text{nF}]$, Log scale.
- L_1: Range $[1\,\text{mH}, 50\,\text{mH}]$, Linear scale.
- Target Peak Gain: 0.0 dB (1.0 voltage ratio).

Table 7.3: Optimization Trial History for RLC Bandpass Filter

Trial ID	R_1 **(Ω)**	C_1 **(nF)**	L_1 **(mH)**	**Computed Gain (dB)**	**Loss**
1	250.0	100.0	25.0	-4.2	17.64
5	120.0	220.0	10.0	-1.8	3.24
10	75.0	330.0	5.0	-0.5	0.25
15	52.0	470.0	2.2	-0.05	0.0025
20	**50.1**	**472.3**	**2.1**	**0.00**	**0.0000**

The optimization reached the target peak gain using $R_1 = 50.1\,\Omega$, $C_1 = 472.3\,\text{nF}$, and $L_1 = 2.1\,\text{mH}$.

Chapter 8

Conclusion and Future Scope

8.1 Summary of Contributions

This fellowship successfully designed, implemented, and deployed a full-stack, asynchronous autotuning and parametric optimization module for eSim-Cloud. By separating the web server interface from background worker execution, the module enables users to run optimization studies without blocking the web application.

The core accomplishments include:

- **Asynchronous Task Processing:** Configured Celery and Redis to run Ngspice simulations in background processes, preventing Nginx and Django request timeouts.
- **Dynamic Configuration Interface:** Built an interactive sidebar panel in React that scans active canvas components and allows users to set optimization targets.
- **Mathematical Target Formulations:** Programmed loss parsing algorithms for both frequency-domain (AC) and time-domain (Transient) analysis.
- **mxGraph Integration:** Implemented a transaction-safe synchronization helper to write tuned parameter values directly back to the active schematic cells.
- **Upstream Deployment:** Submitted the implementation upstream as Pull Request #58 to the FOSSEE eSim-Cloud repository: `https://github.com/FOSSEE/eSim-Cloud/pull/58`.

8.2 Future Scope

Future work could expand the capabilities of the Autotune module in several directions:

- **Multi-objective Optimization:** Extend the coordinator to support optimizing multiple competing targets simultaneously (e.g., maximizing gain while maintaining bandwidth).

- **AI-Assisted Parameter Bounds:** Integrate machine learning models to inspect the schematic topology and automatically suggest search bounds (min/-max) for components.
- **Parallel Worker Scaling:** Configure the Celery worker pool to run individual simulation trials in parallel, reducing overall optimization time.
- **Persistent Study Storage:** Back the Optuna study with a persistent relational store (e.g. PostgreSQL) rather than in-memory state, allowing long-running or paused optimization studies to be resumed across worker restarts and enabling historical comparison of past tuning runs.
- **Result Visualization Dashboard:** Add an in-browser dashboard that plots the convergence history, parameter importance, and parallel-coordinate views of each optimization study, giving users deeper insight into how the optimizer explored the search space.
- **Constraint-Aware Sampling:** Allow users to specify manufacturability or availability constraints (e.g. preferred component value series such as E12/E24) so that suggested parameters map directly to real, purchasable component values.

Taken together, these extensions would move the Autotune module from a single-objective, single-worker proof of concept toward a production-grade, multi-user optimization service capable of supporting larger and more complex circuit design workflows within eSim-Cloud.

Bibliography

1. **FOSSEE, IIT Bombay.** *eSim - Free/Libre and Open Source EDA Tool.* Available at: `https://esim.fossee.in/`

2. **Akiba, T. et al.** *Optuna: A Next-generation Hyperparameter Optimization Framework.* In Proc. of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 2019.

3. **Ngspice.** *Open Source Mixed Mode, Mixed Level Circuit Simulator.* Available at: `https://ngspice.sourceforge.io/`

4. **Celery.** *Distributed Task Queue.* Available at: `https://docs.celeryq.dev/`

5. **Redis.** *In-Memory Data Structure Store.* Available at: `https://redis.io/`

6. **JGraph Ltd.** *mxGraph JavaScript Diagramming Library.* Available at: `https://github.com/jgraph/mxgraph`

7. **Srinivasan, S. & Ramachandran, P.** *Cloud-based EDA simulation frameworks in educational ecosystems.* Journal of Open Source Software, 2024.

8. **Koneti, G.** *Autotune Optimization Module – eSim-Cloud Pull Request #58.* FOSSEE, IIT Bombay. Available at: `https://github.com/FOSSEE/eSim-Cloud/pull/58`