**Technical Guide**

**For**

**SWIP--SDTO-2025-13**

|  |  |  |  |
| --- | --- | --- | --- |
| **SWIP ID** | SWIP--SDTO-2025-13 | **Unit / Dept.** | MI |

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Editor(s) / Author(s)** | **Date** | **Change description** | **Revision** |
| Qian Peisheng | 25 June 2025 | Fill in the SWIP technical information | V0.1 |
| Qian Peisheng | 28 July 2025 | Minor revisions. | V0.2 |
|  |  |  |  |

[1. Introduction 4](#_Toc10325964)

[1.1 Acronyms 4](#_Toc10325965)

[1.2 References 4](#_Toc10325966)

[2. Release Note 5](#_Toc10325967)

[2.1 Software Release Information 5](#_Toc10325970)

[2.2 List of open source/ 3rd party libraries with version and copyright information 5](#_Toc10325971)

[2.3 List of features/functionalities supported by the software 5](#_Toc10325972)

[3. Getting Started 5](#_Toc10325973)

[3.1 System configurations and minimum requirements 5](#_Toc10325974)

[3.2 How to build, install and run the software 5](#_Toc10325975)

[3.2.1 Step by step to build, install and run 5](#_Toc10325976)

[3.2.2 Quick start 5](#_Toc10325977)

[4. User Guide 6](#_Toc10325978)

[4.1 Step by Step user guide 6](#_Toc10325979)

[5. Technical details 6](#_Toc10325980)

[5.1 Software Architecture and design 6](#_Toc10325981)

[6. Test results 6](#_Toc10325982)

[6.1 Performance results 6](#_Toc10325989)

[6.2 Benchmarking 6](#_Toc10325990)

[Appendix A: Supporting Diagrams 7](#_Toc10325991)

# Introduction

<Give a brief of the software and the technology with the context>

**This document describes a Python/Flask‐based platform for rapid development and deployment of AI‐driven prediction models in semiconductor epitaxy process optimization. The software integrates design-of-experiment (DOE) management, interactive parameter editing, and multiple machine-learning inference engines (LSTM neural networks and physics-informed forward models) to predict film thickness and doping uniformity for SiC and Ge wafer processes. By combining experimental and TCAD-derived simulation data with real-time visualization and validation, the platform streamlines traditional manual DOE cycles—enabling process engineers and developers to accelerate model training, evaluate “what-if” scenarios directly in the browser, and export results for downstream analysis.**

## Acronyms

* **API**: Application Programming Interface
* **CSV**: Comma-Separated Values
* **DOE**: Design of Experiment
* **Flask**: (micro) Framework for Python Web Development
* **HTML**: HyperText Markup Language
* **HTTP**: HyperText Transfer Protocol
* **JSON**: JavaScript Object Notation
* **LSTM**: Long Short-Term Memory (neural network)
* **ML**: Machine Learning
* **REST**: Representational State Transfer
* **SiC**: Silicon Carbide
* **TCAD**: Technology Computer-Aided Design
* **UI**: User Interface
* **URL**: Uniform Resource Locator

## References

* SWIP submission form
* <Reference documents if any> NA

# Release Note



## Software Release Information

|  |  |
| --- | --- |
| **Version of the released software** | v0.1 |
| **List of known bugs and any workaround** | NA |
| **Limitations** | **In-memory caching only**  *Limitation:* All DOE tables and edits reside in RAM and are lost on server restart; memory usage grows with table size and user count.  *Solution:* Persist caches in Redis or a relational database (e.g. PostgreSQL) to survive restarts, share across instances, and throttle memory growth.  **Blocking inference and plotting**  *Limitation:* Heatmap generation and model inference execute synchronously on the Flask thread, risking request time-outs and poor responsiveness under load.  *Solution:* Delegate long-running tasks to a background queue (e.g. Celery or RQ) and return immediate acknowledgements, with status polling or WebSocket updates.  **Hard-coded process parameters**  *Limitation:* Mesh resolutions, DOE cell mappings, and supported wafer sizes are embedded in code; extending to new geometries or chemistries requires code changes.  *Solution:* Externalize all geometry definitions, DOE templates, and mapping rules into JSON/YAML configuration files, loaded at runtime.  **No authentication or authorization**  *Limitation:* The application lacks user login and role-based access control, exposing sensitive process data to any visitor.  *Solution:* Integrate an authentication layer (e.g. Flask-Login or OAuth2/JWT) and enforce role-based permissions to restrict data access and editing. |
|  |  |

## List of open source/ 3rd party libraries with version and copyright information

<Provide list, version numbers, the copyright and licensing information for the open source/3rd party software used in this SWIP>

|  |  |  |  |
| --- | --- | --- | --- |
| Name/Title of the open-source software used, source URL and version # | License type (e.g., BSD, MIT, Apache, GPL etc.) | Has the open source used as a library (Y/N)? | Has the source code of the open source modified (Y/N)? |
| PyTorch | BSD | Y | N |
| Flask | BSD | Y | N |
| SciPy | BSD | Y | N |
| Hyperopt | BSD | Y | N |
| Scikit-learn | BSD | Y | N |

## List of features/functionalities supported by the software

<Describe the features / functionalities the Software IP provides>

|  |  |  |
| --- | --- | --- |
| **SN** | **Feature/Functionality Name** | **Description of the functionality** |
| 1 | DOE Table Loading & Caching | Parses Excel/CSV source files for SiC and Ge DOE parameters and response data at startup; stores them in in-memory caches (cell\_data\_cache, ge\_response\_cache) and serves them to clients via AJAX endpoints. |
| 2 | Editable Parameter Grid | Renders DOE tables as <td contenteditable> grids with data-row/col metadata, supports inline editing, undo/redo stack, and persists individual cell edits via the /update\_cell endpoint. |
| 4 | Dynamic Model Selection | Queries /get\_thickness\_models, /get\_doping\_models, and /get\_ge\_thickness\_models to populate ML-method and model-file dropdowns based on available checkpoints. |
| 5 | Inference Execution | Submits DOE parameters and model choices to /predict\_from\_table (SiC) or /predict\_ge (Ge), executes either PyTorch LSTM or serialized forward models, and returns JSON with metrics and base64-encoded plots. |
| 6 | Heatmap Generation | Uses SciPy griddata and Matplotlib to interpolate prediction values onto precomputed wafer meshes, generate PNG heatmaps, encode them in base64, and embed directly in the UI. |
| 7 | Training Job Orchestration | Exposes /start\_training\_skip1 and /start\_training\_ge to launch external training scripts via subprocess.run, with the frontend displaying an elapsed-time counter and completion status. |
| 8 | Data Export & Reporting | Provides /save\_table to stream any cached table or prediction result as .txt or .xlsx, embedding Matplotlib-generated images into Excel sheets using XlsxWriter. |
| 10 | Bulk Table Update | Supports full-table POST updates via /update\_entire\_table to overwrite the server cache with a user-edited 2D array, used prior to inference or export operations. |

# Getting Started

## System configurations and minimum requirements

**3.1 System Configurations and Minimum Requirements**

* **Operating System:**
  + Linux (Ubuntu 18.04+), Windows 10+, or macOS 10.14+
* **CPU:**
  + Quad-core x86\_64 processor
* **Memory:**
  + 8 GB RAM
* **Disk Space:**
  + 5 GB free (to accommodate code, datasets, and model caches)
* **Python:**
  + Python 3.8 interpreter
* **Python Dependencies:**
  + Flask
  + Pandas
  + NumPy
  + SciPy
  + Matplotlib
  + PyTorch
  + scikit-learn
  + joblib
  + xlsxwriter
* **Network:**
  + HTTP/HTTPS access on the chosen server port (default 8080)
* **Web Browser (Client):**
  + Chrome 80+, Firefox 75+, Edge 80+, or Safari 13+
* **Optional (GPU Acceleration):**
  + CUDA-capable NVIDIA GPU with corresponding CUDA/cuDNN if using PyTorch GPU builds

## How to build, install and run the software

1. **Prerequisites**
   * Python 3.8 installed and on your $PATH
   * Git (to clone the repository)
   * (Optional) CUDA/cuDNN and an NVIDIA GPU if you plan to accelerate PyTorch
2. **Clone the Repository**
3. git clone https://<your-git-server>/path/to/epitaxy-platform.git
4. cd epitaxy-platform
5. **Create and Activate a Virtual Environment**
6. python3 -m venv venv
7. source venv/bin/activate # Linux/macOS
8. venv\Scripts\activate.bat # Windows
9. **Install Python Dependencies**
10. pip install --upgrade pip
11. pip install -r requirements.txt
12. **Prepare Configuration and Data**
    * Populate the config/ folder with the provided JSON and Excel files (model mappings, highlight rules, range definitions).
    * Place your DOE source and response files under data/SiC/… and data/GE/… exactly as in the repository structure.
13. **Initialize Caches (Optional)**  
    The first time you run, the app will read all Excel/CSV tables into memory. To verify, start the server and inspect the console logs.
14. **Run the Application**
15. export FLASK\_APP=app.py
16. export FLASK\_ENV=development # enables debug mode
17. flask run --host=0.0.0.0 --port=8080

Or simply:

python app.py

1. **Access the UI**  
   Open your browser to http://localhost:8080/. You will see the landing page with links to the SiC and Ge modules.
2. **(Optional) Production Deployment**  
   For a production setup, use a WSGI server such as Gunicorn:
3. gunicorn --workers 4 --bind 0.0.0.0:8080 app:app

And put it behind an Nginx or Apache reverse proxy with HTTPS termination.

1. **Stopping the Server**  
   Press Ctrl+C in the terminal where Flask or Gunicorn is running.

Once running, you can navigate to /sic\_data, /sic\_model, /ge\_data, and /ge\_model to interact with DOE tables, configure models, run predictions, and launch training jobs.

### Step by step to build, install and run

1. **Install System Prerequisites**
   * Ensure **Python 3.8+** is installed:
   * python3 --version
   * (Optional) For GPU support, install NVIDIA drivers, CUDA 11+, and cuDNN.
2. **Clone the Repository**
3. git clone https://<git-server>/epitaxy-platform.git
4. cd epitaxy-platform
5. **Create a Python Virtual Environment**
6. python3 -m venv venv
7. # Activate:
8. source venv/bin/activate # Linux/macOS
9. venv\Scripts\activate.bat # Windows
10. **Install Python Dependencies**
11. pip install --upgrade pip
12. pip install -r requirements.txt
13. **Prepare Configuration Files**
    * Copy or verify the presence of JSON and Excel configs in the config/ directory:
      + model\_input\_config.json
      + highlight\_config.json
      + ranges.json or Input\_Range\_and\_Step\_Size\_SiC\_6inch.xlsx
    * Ensure any DOE source files (e.g. Recipes.xlsx, GeEpiDOESourceParameters.xlsx) and response data reside under data/SiC/... and data/GE/... with the expected filenames.
14. **Initialize and Verify Data Caches (Optional)**
    * Start the server briefly to confirm that all DOE tables load without errors:
    * python app.py
    * Inspect console logs for successful cache messages, then stop with Ctrl+C.
15. **Run in Development Mode**
16. export FLASK\_APP=app.py
17. export FLASK\_ENV=development
18. flask run --host=0.0.0.0 --port=8080
    * Or simply:
    * python app.py
19. **Access the Web Interface**
    * Open a browser and navigate to:
    * http://localhost:8080/
    * Use the “DOE Pathfinder SiC-E” and “DOE Pathfinder Ge-E” links to enter the respective modules.
20. **Production Deployment (Optional)**
    * Install Gunicorn: pip install gunicorn
    * Launch with multiple workers:
    * gunicorn --workers 4 --bind 0.0.0.0:8080 app:app
    * Configure Nginx or Apache as a reverse proxy with SSL termination.
21. **Shut Down**
    * Press **Ctrl+C** in the terminal running Flask or Gunicorn to stop the service.

### Quick start

1. **Clone & enter project**
2. git clone https://<repo>/epitaxy-platform.git
3. cd epitaxy-platform
4. **Create & activate venv**
5. python3 -m venv venv && source venv/bin/activate
6. **Install requirements**
7. pip install -r requirements.txt
8. **Run the app**
9. python app.py
10. **Open in browser**  
    Visit http://localhost:8080/ and choose “DOE Pathfinder SiC-E” or “DOE Pathfinder Ge-E.”

# User Guide

## Step by Step user guide

<Provide step by step user guide on how user will use the software. If you have a separate user guide document, just refer the document name in this section and attach the guide>

**4.1 Step-by-Step User Guide**

1. **Launch the Application**
   * Start the Flask server (python app.py), then open your browser to http://localhost:8080/.
2. **Select the Module**
   * On the home page, click **“DOE Pathfinder SiC-E”** to work with SiC processes or **“DOE Pathfinder Ge-E”** for Ge processes.
3. **Browse DOE Tables**
   * In the SiC (or Ge) Data page, the left pane displays DOE rows (1–47) with colored cells indicating available data.
   * Click any filled cell (e.g. the orange “E-DB” cell for DOE 1) to load its parameter table into the right pane.
4. **Edit DOE Parameters**
   * The right-pane table is fully editable.
   * Hover a cell to see its valid range and step size; on blur, invalid entries will revert and show a warning.
   * To undo an edit, click **Undo**.
5. **Save or Delete Table**
   * Click **Save** to download the current table as .xlsx or .txt.
   * Click **Delete** to remove it from server cache (the left-pane cell will revert to un-loaded).
6. **Switch to Predictive Model**
   * In the SiC Data page, click the **Predictive model (SiC)** tab link or navigate directly to **Predictive model (SiC)** via the top navigation.
7. **Configure Prediction**
   * Under **Inference Configuration**, choose your **ML Method** (LSTM or Neural Network), then select the corresponding **Thickness** and **Doping** model files from the dropdowns.
8. **Run Prediction**
   * Click the **→** arrow button between the left (input) and right (predictions) tables.
   * Wait a few seconds for the server to compute.
9. **View Results**
   * The right panel updates with:
     + **Numeric metrics**: average, non-uniformity, STD, skewness
     + **Heatmap**: a color-mapped wafer plot of predicted thickness (and doping for SiC)
10. **Export Prediction**
    * Click **Save Results** to download an Excel file containing metrics and embedded heatmap images.
11. **Train New Models (Optional)**
    * In the **Training Configuration** tab, set wafer size, data type (Experimental or Experimental + Simulation), and choose your training algorithms.
    * Click **Start data training**. A timer appears; wait for “Training completes.”
    * Retrained model files will be available in the models directory and automatically listed in the inference dropdowns.
12. **Exit**
    * Close the browser window. In your server terminal, press **Ctrl + C** to stop the Flask application.

# Technical details

## Software Architecture and design

<Provides details of the software components, interactions, input/output and processing and the design approach to implement the software>

The platform is organized into three primary layers—Data, Application, and Presentation—following a lightweight MVC pattern to separate concerns and facilitate maintenance and extension.

1. **Data Layer**
   * **Source Files:**
     + Experimental and simulated DOE parameters and response tables stored in Excel (.xlsx) and CSV under data/SiC/… and data/GE/….
     + Configuration files in JSON (config/\*.json) or Excel for input mappings, highlight rules, and validation ranges.
   * **Caches:**
     + cell\_data\_cache and ge\_response\_cache hold parsed tables in memory keyed by DOE identifiers, avoiding repeated disk I/O.
2. **Application Layer (Flask Backend)**
   * **Route Handlers:**
     + Data retrieval (/get\_cached\_data, /get\_cached\_ge\_data), uploads (/upload\_data), updates (/update\_cell, /update\_entire\_table), and exports (/save\_table).
     + Model management (/get\_thickness\_models, /get\_ge\_thickness\_models, /get\_doping\_models).
     + Inference endpoints (/predict\_from\_table for SiC; /predict\_ge for Ge) that orchestrate data extraction, scaling, model loading, prediction, statistics computation, and plot generation.
     + Training orchestration (/start\_training\_skip1, /start\_training\_ge) that spawns external Python scripts via subprocess.run.
   * **Utilities:**
     + Data preprocessing (time‐string conversion, input‐configuration mapping).
     + Scalers (StandardScaler) fitted at startup for consistent normalization.
     + Mesh generation routines for wafer‐map interpolation.
     + Table‐to‐HTML conversion with highlighting logic in generate\_table\_html.
3. **Presentation Layer (Frontend)**
   * **Templates:** Jinja2 HTML pages (index.html, sic\_data.html, sic\_model.html, ge\_data.html, ge\_model.html) provide structure, CSS placeholders, and dynamic injection points.
   * **Static Assets:** CSS for layout and color schemes; JavaScript for interactive behaviors.
   * **Client Logic:**
     + AJAX calls to backend endpoints to fetch or submit table HTML and JSON payloads.
     + Contenteditable tables with focus/blur listeners for change detection and undo/redo.
     + Tooltip and validation logic driven by range data fetched from /get\_ranges.
     + Dynamic dropdown population for model selection.
     + Modal dialogs for image galleries.
     + Arrow‐icon–driven prediction triggers and real‐time training timers.
4. **5.1.2 Interactions & Data Flow**
5. **Application Startup**
   * Read configuration files; load and cache all DOE tables; precompute wafer meshes.
   * Fit input/output scalers for ML models.
6. **User Requests**
   * **Page Load:** Client requests /sic\_data or /ge\_data → server renders initial HTML with minimal data.
   * **Table Fetch:** Clicking a DOE cell invokes /get\_cached\_data?cell\_id=… → server returns HTML fragment for the table; client injects it into the DOM.
   * **Editing:** User edits a cell → on blur, client POSTs to /update\_cell; cache is updated.
   * **Bulk Update:** Before inference, client may POST full table to /update\_entire\_table.
   * **Inference:** Client sends form‐encoded or JSON to /predict\_from\_table or /predict\_ge; server processes and returns JSON with metrics + base-64 image strings; client updates UI.
   * **Training:** Clicking “Start data training” hits /start\_training\_\*; server runs training scripts synchronously, returning status on completion; client shows elapsed time.
   * **Export & Images:** Client requests /save\_table or /get\_images; server streams files or JSON lists; client triggers download or displays image grid.
7. **5.1.3 Inputs, Processing, and Outputs**

* **Inputs:** DOE parameter tables (Excel/CSV), user‐edited values, selected model filenames, ML‐method flags.
* **Processing:**
  + Table parsing and sanitization → in-memory caching
  + Input extraction per configuration → numeric array
  + Normalization → model inference (LSTM or forward model)
  + Statistical computation (average, non-uniformity, STD, skewness)
  + Spatial interpolation on fixed wafer mesh → Matplotlib plot
* **Outputs:**
  + HTML fragments of editable tables with highlight annotations
  + JSON payloads containing numeric metrics and base64-encoded plots
  + Downloadable .xlsx and .txt files embedding tables and images

1. **5.1.4 Design Approach**

* **Modularity:** Each functional area (DOE management, inference, training, export) resides in discrete route handlers and utility functions.
* **Configuration-Driven:** Highlight rules, input mappings, and range constraints are externalized, allowing non-code updates.
* **Stateless Frontend:** The client relies solely on server-provided HTML/JSON; no persistent client state outside of localStorage stacks for undo and navigation.
* **Blocking vs. Asynchronous:** Core inference runs synchronously for simplicity; training scripts block by design but could be refactored to an asynchronous task queue.
* **Scalability Considerations:** In-memory caches optimize latency but can be migrated to persistent stores when scaling beyond a single instance.

This architecture balances rapid development and clear separation of concerns while providing a straightforward path for future enhancements—such as database integration, asynchronous processing, or role-based security—without upheaval of existing modules.

# Test results



## Performance results

<Describe the performance results achieved by the software e.g. accuracy, speed etc.>

**Dataset Description and Model Selection**

The SIC dataset comprises 47 samples, each representing a complete sequence of 9 operational steps. Given the ordered, time-dependent nature of these steps, the dataset is inherently sequential. To effectively model this temporal structure, we adopt a Long Short-Term Memory (LSTM) network, which is well-suited for capturing dependencies across time steps in sequential data. The model outputs the predicted thickness and doping at key process points; related plots have been omitted here to preserve confidentiality.

**Model Justification**

Compared to traditional feedforward neural networks such as Multi-Layer Perceptrons (MLPs), LSTMs are explicitly designed to model temporal correlations by maintaining and updating an internal memory state over time. This capability allows LSTMs to learn patterns that depend on the order and timing of events, which is crucial for the SIC data. While LSTMs offer superior performance on sequential tasks, their architecture is more complex than that of MLPs, incorporating gating mechanisms to control information flow. As such, effective training requires careful tuning of parameters such as hidden size, number of layers, learning rate, and dropout rate.

The architecture of our model is described in an accompanying diagram, which is not included here to meet confidentiality requirements. First, a fully connected layer is applied to each step’s input features to produce an embedded representation. These embedded features are then processed sequentially by the LSTM layer, which captures temporal dependencies. Finally, another fully connected layer is used to map the LSTM output to the desired target dimensions (doping and thickness).

**Experimental Setup**

In our dataset, only 5 out of the 9 operational steps contain variable features, while the remaining steps remain constant. Therefore, we selected the features from these 5 variable steps as the input to the LSTM model. The target outputs are **doping** and **thickness**, which the model is trained to predict.

All experiments were conducted on a high-performance workstation equipped with an **NVIDIA RTX 4090 GPU**, ensuring efficient training and evaluation of the model. The implementation was done using **PyTorch**.

This configuration was chosen to balance model complexity and generalization ability. The training process also incorporated data normalization and early stopping according to the performance on validate dataset to improve convergence and prevent overfitting.

## Benchmarking

<Benchmarking information respect to performance of the software solution Vs solution created by others>

**Cross-Validation**

To robustly assess the model’s generalization performance, we employ 10-fold cross-validation. In this approach, the dataset is randomly divided into 10 equal-sized subsets. For each fold, the model is trained on 8 subsets, validated on 1 subset, and the remaining 1 subset is used to test and record results only if the validation subset achieves the best performance during training. This process is repeated 10 times, ensuring that each subset serves as the validation set exactly once. This strategy helps mitigate variance in evaluation and provides a more reliable estimate of model performance, which is particularly important given the small dataset size.

**Evaluation Strategy and Metrics**

To assess the performance of the model, we use two complementary metrics: the R² score and the Mean Absolute Percentage Error (MAPE).

R² Score measures how well the model's predictions approximate the actual values. It indicates the proportion of the variance in the target variable that is predictable from the input features.

Advantage: R² provides an intuitive sense of model performance, especially for regression tasks. A score close to 1 indicates a strong fit, making it useful for comparing models or assessing improvements.

MAPE (Mean Absolute Percentage Error) calculates the average absolute difference between the predicted and actual values, expressed as a percentage of the actual values.

Advantage: MAPE is scale-independent, making it easy to interpret across different datasets or target ranges. It is particularly useful when understanding the relative size of prediction errors is more important than their absolute magnitude.

By using both metrics, we gain a balanced understanding of the model’s predictive accuracy and robustness, capturing both absolute performance (via MAPE) and relative fit (via R²). The results are shown in the table below.

| **Experiment Name** | **R² Score** | **MAPE** |
| --- | --- | --- |
| LSTM\_model\_lr0.001 | **0.38** | **4.05** |
| LSTM\_model\_lr0.0005 | 0.34 | 4.11 |
| LSTM\_model\_lr0.0001 | 0.34 | 4.20 |
| LSTM\_model\_lr5e‑05 | 0.37 | 4.05 |
| LSTM\_model\_lr1e‑05 | 0.37 | 4.01 |
| UnetFc\_model\_lr0.001 | 0.36 | 4.08 |
| UnetFc\_model\_lr0.0005 | **0.39** | **3.93** |
| UnetFc\_model\_lr5e‑05 | 0.36 | 3.96 |
| UnetFc\_model\_lr0.0002 | 0.36 | 3.99 |
| UnetFc\_model\_lr0.0001 | 0.34 | 4.14 |
| UnetFc\_model\_lr1e‑05 | 0.13 | 4.47 |

Table 1. Experimental results of thickness prediction using LSTM and UnetFc models.

# 

# Appendix A: Supporting Diagrams

<This is applicable for supporting items e.g. use case diagrams, flow charts etc.>

|  |  |  |  |
| --- | --- | --- | --- |
|  | memmory | Training time /epoch (38 training samples) | Testing time/ sample |
| LSTM | 263MB | 329.7 ms | 1.15ms |
| UnetFc | 227.25MB | 298.24ms | 1.11ms |

Table 2. Computation efficiency of proposed LSTM and UnetFc models.

A graph of different colored lines

AI-generated content may be incorrect.

Fig.1: model with different learning rate, coverage speed is different