# FOSSEE Summer Internship Report

On

## Development of Chemical PFD Builder

Submitted by

**Abhinav Verma**

3rd Year B.Tech Student, CSE
VIT Bhopal University

Under the Guidance of

**Prof. Prabhu Ramachandran**

Department of Aerospace Engineering
Indian Institute of Technology Bombay

Mentor:

**Chirag Koyande**

June 29, 2026

# Acknowledgments

I would like to express my sincere gratitude to the FOSSEE team at IIT Bombay for the opportunity to participate in the Internship program. This experience has been intellectually rewarding, allowing me to contribute to a significant technical project alongside a talented and dedicated team.

As a member of the **Desktop Frontend Team** for the Chemical PFD Builder, I contributed to the PyQt5-based desktop canvas application. My work spanned multiple facets of the codebase: implementing a graph-based PFD validation engine, building a high-performance A* routing engine for connections, developing a complete component resize system, introducing a unit testing infrastructure, and refining the overall UI/UX. Across 18 merged pull requests, I gained deep hands-on experience in Python, PyQt5, graph algorithms, software testing, and open-source collaboration workflows.

I am deeply grateful to **Prof. Prabhu Ramachandran** for his leadership and vision in promoting technical excellence through the FOSSEE initiative.

Special thanks to my mentor, **Chirag Koyande**, for his invaluable guidance, timely reviews, and technical clarity throughout the internship. I also appreciate the support and collaboration of my fellow team members.

This internship has been a defining chapter in my journey, equipping me with the skills and clarity for a career in software engineering.

# Table of Contents


**Acknowledgments** 1

**Chapter 1 Introduction** 2

1.1 FOSSEE Project 2

1.1.1 Projects and Activities 2

1.2 Chemical PFD Builder 3

1.2.1 Overview and Purpose 3

1.2.2 System Architecture 3

1.2.3 Key Features 4

1.2.4 Evolution from Legacy Tool 4

1.2.5 Conclusion 4

**Chapter 2 FOSSEE Selection Task: Chemical Equipment Visualizer** 5

2.1 Problem Statement 5

2.1.1 Project Overview 5

2.1.2 Key Features Required 6

2.2 Tasks Done 6

2.2.1 Django REST API Backend 6

2.2.2 React Web Frontend 7

2.2.3 PyQt5 Desktop Workstation 7

2.3 Code Snippet: Django Ingestion View 8

2.4 Outcome 8

**Chapter 3 Task 1: Graph-Based Process Validation Engine** 8

3.1 Problem Statement 8

3.2 Tasks Done 9

3.2.1 Initial Validation Engine (PR #115) 9

3.2.2 Code Snippet: DFS Loop Detection Logic 9

3.2.3 Validation UI Refinement (PR #116) 9

3.2.4 Rule Relaxation (PR #119) 10

3.2.5 Validation Engine Restoration (PR #128) 10

3.3 Outcome 10

**Chapter 4 Task 2: A* Routing Engine and Connection Rendering** 11

4.1 Problem Statement 11

4.2 Tasks Done 11

4.2.1 Core Algorithm: A* Heuristic Grid Search 11
4.2.2 Code Snippet: A* Shortest Orthogonal Path Search 12
4.2.3 Obstacle Avoidance 12
4.2.4 Visual Rendering 13
4.2.5 Performance Optimizations 13
4.3 Outcome 13

**Chapter 5 Task 3: Local Legend Fallbacks for Flow Line Components 14**
5.1 Problem Statement 14
5.2 Tasks Done 14
5.2.1 Code Snippet: Inflow/Outflow Local Fallbacks 14
5.3 Outcome 15

**Chapter 6 Task 4: Synchronize Component Sizing Across Desktop and Web 16**
6.1 Problem Statement 16
6.2 Tasks Done 16
6.2.1 Unified Sizing Formula 16
6.2.2 1:1 Native Rendering from Database 16
6.2.3 Minimum Size Fallback 17
6.2.4 Visual Polish 17
6.3 Outcome 17

**Chapter 7 Task 5: Help Modal for the Add New Symbol Dialog 18**
7.1 Problem Statement 18
7.2 Tasks Done 18
7.2.1 Code Snippet: Interactive QDialog Initialization 18
7.3 Outcome 19

**Chapter 8 Task 6: Desktop Frontend Unit Testing Infrastructure 20**
8.1 Problem Statement 20
8.2 Tasks Done 20
8.2.1 Initial Test Suite (PR #146) 20
8.2.2 Code Snippet: Cycle Validation Test Case 21
8.2.3 Grip Placement Bug Fix (PR #147) 21
8.2.4 Documentation (PR #155) 21
8.2.5 Component Resize Test Suite (PR #171) 22
8.3 Outcome 22

**Chapter 9 Task 7: Component Resize System** 23
9.1 Problem Statement 23
9.2 Tasks Done 23
9.2.1 Core Resize Implementation (PR #166) 23
9.2.2 Bug Fixes: Label Clipping and Collision Detection (PR #167) 23
9.2.3 Independent Width and Height Resizing (PR #178) 24
9.2.4 Code Snippet: Resizing Control & Collision Checks 24
9.3 Outcome 24
**Chapter 10 Task 8: Desktop Application UI/UX Polish** 25
10.1 Problem Statement 25
10.2 Tasks Done 25
10.2.1 Save Options for New Projects (PR #106) 25
10.2.2 Recent Projects Card Styling (PR #107) 25
10.2.3 Landing Page and UI Styling (PR #175) 25
10.3 Outcome 26
**Chapter 11 Conclusions** 27
11.1 Tasks Accomplished 27
11.2 Skills Developed 27
11.2.1 Technical Skills 27
11.2.2 Professional Skills 28
11.3 Future Scope 28
**Chapter 12 Bibliography** 29

# Chapter 1 Introduction

## 1.1 FOSSEE Project

The FOSSEE (Free/Libre and Open Source Software for Education) project promotes the use of FLOSS tools in academia and research. It is part of the National Mission on Education through Information and Communication Technology (NMEICT), Ministry of Education (MoE), Government of India.

### 1.1.1 Projects and Activities

The FOSSEE Project supports the use of various FLOSS tools to enhance education and research. Key activities include:

- **Textbook Companion:** Porting solved examples from textbooks using FLOSS.
- **Lab Migration:** Facilitating the migration of proprietary labs to FLOSS alternatives.
- **Niche Software Activities:** Specialized activities to promote niche software tools.
- **Forums:** Providing a collaborative space for users.
- **Workshops and Conferences:** Organizing events to train and inform users.

For more details, visit the official FOSSEE website at https://fossee.in/

## 1.2 Chemical PFD Builder

Chemical PFD Builder is a cross-platform software system developed under the FOSSEE project for creating, editing, and managing Process Flow Diagrams (PFDs) commonly used in chemical engineering. The tool is designed to cater to both academic and industrial needs, providing a professional-grade diagramming environment with cloud-based project storage and seamless access across web and desktop platforms.

### 1.2.1 Overview and Purpose

Process Flow Diagrams are essential in chemical engineering for visualizing processes, equipment, and material flows. Traditional PFD creation often relies on proprietary software, which can be expensive and restrict collaboration. Chemical PFD Builder addresses this by

offering a free and open-source alternative that runs on multiple platforms and supports modern workflows.

### 1.2.2 System Architecture

Chemical PFD Builder follows a hybrid architecture with a common backend serving both web and desktop frontends:

- **Backend (Django REST):** Handles user authentication, project management, and diagram storage via RESTful APIs.
- **Web Frontend (React + Konva.js):** Provides a browser-based canvas editor.
- **Desktop Frontend (PyQt5):** Provides a native desktop application with a custom canvas, built-in validation, and an A*-based routing engine.

All modules share a common data schema, enabling seamless project exchange and cross-platform continuity.

### 1.2.3 Key Features

The software enables users to:

- Create PFDs through an intuitive drag-and-drop interface with support for a wide range of chemical engineering symbols.
- Automatically label components using CSV or Excel files.
- Validate diagram logic in real time (loop detection, isolated components, missing inlets/outlets).
- Export high-resolution images, vector graphics, and structured project files.
- Synchronize projects with the cloud and access them from any device.

### 1.2.4 Evolution from Legacy Tool

Chemical PFD Builder builds upon an earlier standalone desktop application that provided basic drag-and-drop functionality and file-based saving. The new version introduces a shared backend, a web frontend, and enhanced features such as real-time validation, A*-based connection routing, component resizing, and cross-platform project continuity.

### 1.2.5 Conclusion

Chemical PFD Builder represents a significant step forward in providing a free, open-source, and feature-rich tool for chemical engineering diagramming. For more information, visit the GitHub repository: https://github.com/FOSSEE/Chemical-PFD-Web-Desktop

# Chapter 2 FOSSEE Selection Task: Chemical Equipment Visualizer

## 2.1 Problem Statement

The screening task required building a hybrid web and desktop application for chemical equipment data visualization and analytics.

### 2.1.1 Project Overview

Create a Web + Desktop application that allows users to upload a CSV file containing chemical equipment data (Equipment Name, Type, Flowrate, Pressure, Temperature). The Django backend parses the data, performs analysis, and provides summary statistics via API. Both React (Web) and PyQt5 (Desktop) frontends consume this API to display data tables, charts, and summaries.

### 2.1.2 Key Features Required

- **CSV Upload:** Both interfaces allow CSV upload to backend
- **Data Summary API:** Returns count, averages, and equipment type distribution
- **Visualization:** Charts using Chart.js (Web) and Matplotlib (Desktop)
- **History Management:** Store last 5 uploaded datasets
- **PDF Report Generation:** Generate analysis reports
- **Basic Authentication:** User login/registration

## 2.2 Tasks Done

### 2.2.1 Django REST API Backend

A robust Django backend was implemented to serve as the single source of truth for both web and desktop clients:

- **Data Model:** Designed a relational schema mapping Dataset entities (representing uploaded CSV files) to Equipment records with numeric sensor fields.
- **CSV Ingestion:** Integrated pandas for server-side parsing and structure verification of uploaded files.

- **Bulk Operations:** Configured Django's database manager to use bulk_create for high-throughput writes.
- **Cleanup Policy:** Implemented an automatic database vacuum to keep only the five most recent datasets, preventing storage bloat on the host SQLite database.
- **PDF Exporter:** Utilized ReportLab to compile summary statistics into downloadable PDF reports directly from database datasets.

### 2.2.2 React Web Frontend

A modern dashboard was created using React and Vite:

- **Drag-and-Drop Ingestion:** Implemented a user-friendly drop zone with file type validation.
- **Dynamic Data Grid:** Created filterable tables to browse ingested equipment records.
- **Analytical Visualizations:** Integrated Chart.js for showing distribution breakdowns of equipment categories.

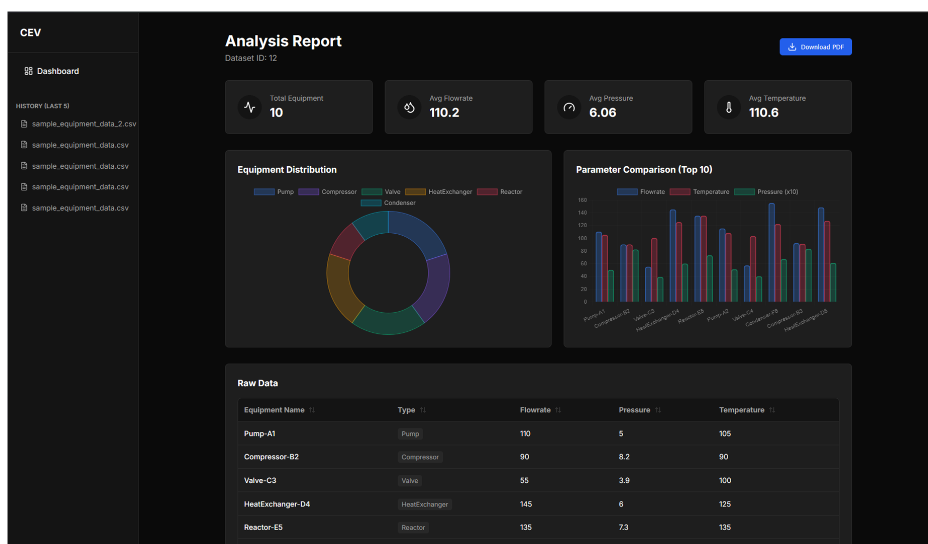


Figure 2.1: Selection Task Web Frontend Dashboard

### 2.2.3 PyQt5 Desktop Workstation

A high-performance workstation client was developed for engineering analysis:

- **Frameless UI Design:** Engineered a dark-themed visual window style (#0a0a0a) with custom window title bar buttons.
- **Asynchronous UI Worker Threads:** Wrapped API request operations in subclassed QThread workers, allowing the UI to remain responsive during query latency.

- **Advanced Analytics Canvas:** Integrated Matplotlib within PyQt5 layouts to draw equipment parameter distributions, profile charts, and Top-10 ranking bar graphs.

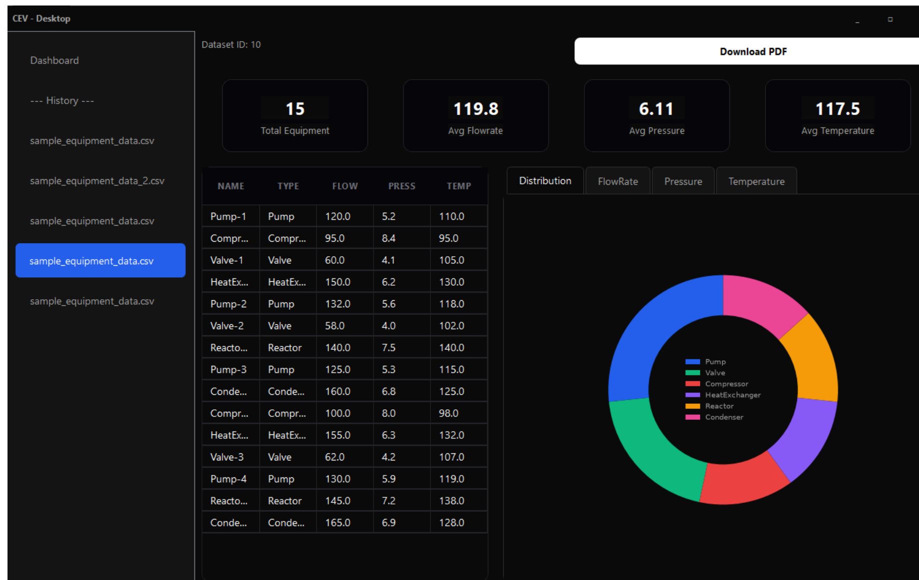


Figure 2.2: Selection Task Desktop Application Workstation

## 2.3 Code Snippet: Django Ingestion View

The backend endpoint uses a Pandas DataFrame to parse and validate CSV headers, and performs bulk writes:

```
class UploadCSVView(APIView):
    parser_classes = [MultiPartParser]

    def post(self, request):
        if 'file' not in request.FILES:
            return Response({"error": "No file uploaded"},
status=status.HTTP_400_BAD_REQUEST)
        file = request.FILES['file']
        try:
            df = pd.read_csv(file)
        except Exception as e:
            return Response({"error": f"Failed to parse
CSV: {str(e)}"}, status=status.HTTP_400_BAD_REQUEST)

        # Validate columns
        required_cols = ['Equipment Name', 'Type',
'Flowrate', 'Pressure', 'Temperature']
        if not all(col in df.columns for col in
required_cols):
            return Response({"error": f"Invalid columns.
```

```
Required: {required_cols}"},
status=status.HTTP_400_BAD_REQUEST)

        # Create Dataset
        dataset =
Dataset.objects.create(filename=file.name)

        # Bulk Create Equipment
        equipment_list = [
            Equipment(
                dataset=dataset,
                name=row['Equipment Name'],
                type=row['Type'],
                flowrate=row['Flowrate'],
                pressure=row['Pressure'],
                temperature=row['Temperature']
            )
            for _, row in df.iterrows()
        ]
        Equipment.objects.bulk_create(equipment_list)

        # Cleanup old datasets (Keep last 5)
        ids_to_keep =
list(Dataset.objects.order_by('-uploaded_at')[:5].values_l
ist('id', flat=True))
        if ids_to_keep:

Dataset.objects.exclude(id__in=ids_to_keep).delete()

        return Response({"id": dataset.id, "message":
"Upload Successful"}, status=status.HTTP_201_CREATED)
```

## 2.4 Outcome

The selection task successfully demonstrated a fully integrated hybrid analytics client-server application. All functional requirements were met, enabling the selection of the candidate for the main Chemical PFD Builder development team.

# Chapter 3 Task 1: Graph-Based Process Validation Engine

---

Pull Requests: PR #115 · PR #116 · PR #119 · PR #128

## 3.1 Problem Statement

The desktop canvas had no mechanism to detect logical errors in a Process Flow Diagram before a user attempted to run or export it. Users could inadvertently create circular loops, leave components isolated without any connections, or build diagrams with no inlet or outlet, violating fundamental PFD rules. A real-time validation engine was required that would identify these errors automatically as the diagram evolved.

## 3.2 Tasks Done

### 3.2.1 Initial Validation Engine (PR #115)

The first implementation introduced validation.py, a dedicated module containing the GraphValidator class. The validator converts the live canvas state into an in-memory adjacency-list representation of the PFD graph and performs two graph traversals:

- **DFS (Depth-First Search) — Circular Loop Detection:** A classic DFS with a recursion stack is used to detect back-edges, which correspond to circular material loops in the process flow.
- **BFS (Breadth-First Search) — Flow Continuity:** Starting from identified Inflow/Inlet components, the BFS verifies that all components are reachable, catching broken process flows.
- **Isolated Component Detection:** Any node with both in-degree and out-degree equal to zero is flagged as isolated.

The validation is triggered by connecting to PyQt5's QUndoStack.indexChanged signal, ensuring every canvas action (add, delete, connect, disconnect) fires a revalidation without requiring an explicit user action.

The ComponentWidget was updated to display a customized red dashed halo and specific per-error tooltips. A dynamic global warning indicator

button was added to the OverlayContainer in canvas_screen.py to list all active canvas errors inline.

### 3.2.2 Code Snippet: DFS Loop Detection Logic

```
def _find_loops_dfs(self):
    """Identify components involved in circular loops
using DFS."""
    visited = set()
    rec_stack = set()
    loop_components = set()

    def dfs(node, path):
        visited.add(node)
        rec_stack.add(node)
        path.append(node)

        for neighbor in self.adj_list[node]:
            if neighbor not in visited:
                dfs(neighbor, path)
            elif neighbor in rec_stack:
                cycle_start_index = path.index(neighbor)
                for n in path[cycle_start_index:]:
                    loop_components.add(n)

        rec_stack.remove(node)
        path.pop()

    for comp in self.components:
        if comp not in visited:
            dfs(comp, [])

    return list(loop_components)
```

### 3.2.3 Validation UI Refinement (PR #116)

Following user feedback, the visual design of the validation UI was refined:

- Removed the intrusive red dashed border halo; replaced with a clean badge-style red ! icon overlay.
- Improved error message clarity and specificity.
- Fixed the SVG aspect ratio scaling of ComponentWidget so that components rendered at correct proportions after the validation pass updated their state.

#### 3.2.4 Rule Relaxation (PR #119)

The validation rules were further relaxed to reduce false positives:

- Components are only flagged as isolated if both in-degree and out-degree are zero. A component with at least one connection in either direction is no longer marked as an error.
- The circular loop detection (DFS) was retained unchanged.

This change was introduced after observing that legitimate process diagrams often contain terminal components (e.g., a final heat exchanger with only one incoming stream) that the stricter rule incorrectly flagged.

#### 3.2.5 Validation Engine Restoration (PR #128)

During an earlier routing refactor, the validation engine was accidentally overwritten. This PR restored the correct engine with three targeted fixes:

- **Logical Graph Validation Restored:** Removed an incorrect _is_physically_connected function that was performing strict physical coordinate matching, causing valid connections to fail validation. The engine was reverted to purely logical adjacency-list based graph construction.
- **Aspect Ratio Sizing Restored:** The component_widget.py dynamic sizing logic was restored. Components now accurately preserve their SVG aspect ratios through calculate_svg_rect() instead of being stretched to fit strict bounding boxes.
- **UI Badges Restored:** The red ! validation badges for isolated components were re-enabled after the routing refactor had unintentionally disabled them.

### 3.3 Outcome

The validation engine provides real-time PFD logical correctness checking without requiring any explicit user action. It catches circular loops, broken process flows, isolated components, and missing inlets/outlets, displaying targeted error tooltips on each offending component.

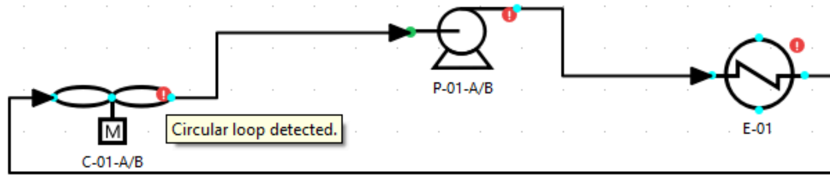


Figure 3.1: Circular loop validation error displayed on the PyQt5 desktop canvas

# Chapter 4 Task 2: A* Routing Engine and Connection Rendering

Pull Requests: PR #132

## 4.1 Problem Statement

The desktop canvas lacked an intelligent connection routing system. Connections were drawn as straight lines that could pierce through component bodies, cross each other without any visual distinction, and failed to respect the logical departure and arrival sides of connection ports. A high-performance, orthogonal routing engine was required to produce clean, professional-looking PFD connection paths.

## 4.2 Tasks Done

### 4.2.1 Core Algorithm: A* Heuristic Grid Search

The routing engine implements a grid-based A* pathfinder specifically tuned for orthogonal PFD layouts:

- **Grid Mapping:** The canvas is discretized into a 10 px logical grid. Every node in the search space corresponds to one grid cell.
- **Cost Function:** Turns are penalized (+60 cost) to favor long straight runs; segment crossovers with existing connections are penalized (+25 cost) to keep the diagram clean.
- **Region of Interest (ROI):** To maintain 60 fps performance during live dragging, the search space is limited to a dynamically calculated ROI around the start and end grip points.

### 4.2.2 Code Snippet: A* Shortest Orthogonal Path Search

```
while pq:
    f, cost, cc, cr, dir_idx = heapq.heappop(pq)

    if cc == eg_c and cr == eg_r:
        found_state = (cc, cr, dir_idx)
        break

    if best_cost.get((cc, cr, dir_idx), float('inf')) <
cost:
        continue
```

```
    for dc, dr, n_dir_idx in dirs:
        nc, nr = cc + dc, cr + dr

        # Check ROI bounds
        if nc < roi_col_lo or nc > roi_col_hi or nr <
roi_row_lo or nr > roi_row_hi:
            continue

        # Calculate incremental cost
        move_cost = 1.0
        if dir_idx != n_dir_idx:
            move_cost += TURN_PENALTY

        nb = (nc, nr)
        if nb in line_cells:
            move_cost += CROSSOVER_PENALTY

        if nb in obstacles and nb != eg:
            move_cost += 50000.0  # Soft obstacle penalty

        new_cost = cost + move_cost
        state = (nc, nr, n_dir_idx)

        if state in best_cost and new_cost >=
best_cost[state]:
            continue

        best_cost[state] = new_cost
        parent[state] = (cc, cr, dir_idx)

        f_score = new_cost + abs(nc - eg_c) + abs(nr -
eg_r)
        heapq.heappush(pq, (f_score, new_cost, nc, nr,
n_dir_idx))
```

#### 4.2.3 Obstacle Avoidance

- **Soft Obstacles:** Every component's bounding rectangle is treated as a soft obstacle with a cost of +50,000. This is high enough to prefer routing around components, but still finite so the algorithm can reach a grip point inside a bounding box.

- **Side-Entry Enforcement:** The router projects a 20 px stub outward from each grip before the full A* search, ensuring lines never pierce component bodies immediately upon leaving a port.

### 4.2.4 Visual Rendering

- **Layered Painting:** Arrowheads are drawn on a separate ConnectionOverlay widget always painted on top of all component widgets, solving Z-order issues.
- **Crossing Jumps:** A post-path scan detects intersections between connections and dynamically injects semi-circular jump arcs to visually distinguish crossing lines.

### 4.2.5 Performance Optimizations

- **Static Obstacle Caching:** During active dragging, component bounding rectangles are pre-compiled into a bit-mask, converting obstacle lookup from O(N) per A* expansion to O(1).
- **Lazy Jump Generation:** Intersection detection is deferred to mouseReleaseEvent rather than recalculated every frame.

## 4.3 Outcome

The A* routing engine produces clean, orthogonal connection paths that route around components, respect port sides, and visually distinguish crossing lines with jump arcs, while maintaining 60 fps performance during live dragging through ROI limiting and static obstacle caching.

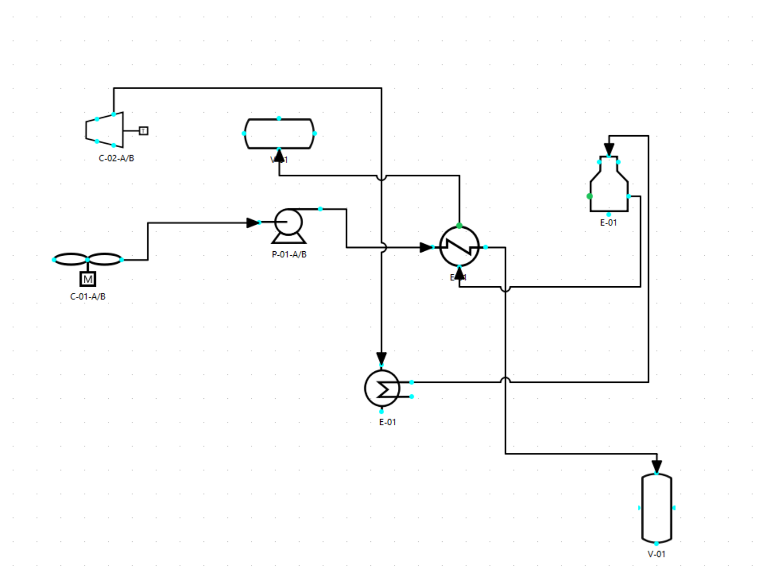


Figure 4.1: Orthogonal connection routing with crossing jump arcs on the desktop canvas

# Chapter 5 Task 3: Local Legend Fallbacks for Flow Line Components

---

Pull Requests: PR #134

## 5.1 Problem Statement

Two special component types — InflowLine and OutflowLine — did not have legend entries in the component database. When these components were loaded onto the canvas, their label was left blank because the legend lookup failed silently. This made it impossible for users to identify inlet and outlet streams visually in the PFD.

## 5.2 Tasks Done

The fix implemented local legend fallbacks in the component loading logic. When the legend lookup for a component's type returns None or an empty string, a hard-coded fallback is applied:

- Components matching the 'Inflow' or 'InflowLine' type receive the legend label "Inflow Stream".
- Components matching the 'Outflow' or 'OutflowLine' type receive the legend label "Outflow Stream".

The fallback is applied at the point of component construction in component_widget.py, so it works both for freshly dropped components and for components loaded from a saved project file.

### 5.2.1 Code Snippet: Inflow/Outflow Local Fallbacks

```
if key not in self.label_data:
    # Local Fallback for specific components with empty
API legends
    if not legend:
        if key == "inflowline":
            legend = "IN"
        elif key == "outflowline":
            legend = "OUT"

    if legend:
        self.label_data[key] = {
            "legend": legend,
            "suffix": suffix,
```

```
            "count": 0
        }
```

## 5.3 Outcome

InflowLine and OutflowLine components now always display their correct legend label on the canvas, regardless of the state of the component database.

# Chapter 6 Task 4: Synchronize Component Sizing Across Desktop and Web

---

Pull Requests: PR #141

## 6.1 Problem Statement

Components dragged onto the desktop canvas rendered at a different physical size than the same components on the web frontend. This caused visual inconsistency when switching between platforms and problems when loading a project saved on one platform into the other.

## 6.2 Tasks Done

### 6.2.1 Unified Sizing Formula

The desktop was migrated from an arbitrary 100 px bounding box to the web frontend's targetArea = 1500 geometric formula:

```
target_area = 3500.0
aspect_ratio = float(width) / float(height)
logical_height = math.sqrt(target_area / aspect_ratio)
logical_width = logical_height * aspect_ratio
```

This ensures that a freshly dropped component starts at the same physical size on both platforms.

### 6.2.2 1:1 Native Rendering from Database

The desktop's export.py contained an artificial size multiplier that distorted component dimensions when loading from the database. This multiplier was removed. The loader now accurately respects the exact Width/Height values stored in the database.

### 6.2.3 Minimum Size Fallback

When loading a project originally saved from the web frontend with very small component sizes, the desktop loader now snaps them up to a minimum area of 3500 $px^2$ to guarantee components remain interactive and visible.

#### 6.2.4 Visual Polish

Connection node radii and margins were scaled down proportionally to stay consistent with the newly updated baseline component scale.

### 6.3 Outcome

Components now render at identical sizes on both platforms. Round-trip project save/load preserves component dimensions accurately, and web-sourced micro-components are automatically corrected to remain usable on the desktop.

# Chapter 7 Task 5: Help Modal for the Add New Symbol Dialog

Pull Requests: PR #143

## 7.1 Problem Statement

The 'Add New Symbol' dialog allows users to create custom chemical engineering components from SVG files, but had no in-application guidance. New users frequently created components with incorrectly placed grips, missing metadata fields, or wrong coordinate system orientations.

## 7.2 Tasks Done

An interactive Help / Info modal was added to the 'Add New Symbol' dialog, triggered by a ? help button in the dialog's header bar. The modal provides:

- **Step-by-step guide:** A numbered walkthrough of the custom component creation process from importing an SVG to defining grips and saving.
- **Grip coordinate system explanation:** Explains that grip coordinates use a bottom-left origin (Y=0 at bottom, Y=100 at top), the inverse of the canvas coordinate system, with examples for correct inlet/outlet grip placement.
- **Metadata field descriptions:** Each form field (object name, legend, suffix, grips) is described with its purpose and expected format.
- **Common pitfalls:** Lists the most frequent mistakes with their corrections.

The modal is implemented as a QDialog subclass with a QScrollArea to handle variable content length on smaller display resolutions.

### 7.2.1 Code Snippet: Interactive QDialog Initialization

```
def show_help(self):
    help_dialog = QDialog(self)
    help_dialog.setWindowTitle("Help / Info")
    help_dialog.setWindowFlags(help_dialog.windowFlags() &
~Qt.WindowContextHelpButtonHint)
```

```
    help_dialog.setModal(True)
    help_dialog.setMinimumWidth(400)
    help_dialog.setStyleSheet(self.styleSheet())

    layout = QVBoxLayout(help_dialog)
    layout.setSpacing(15)

    help_text = """
    <h3 style="margin-top:0; margin-bottom:5px;
font-weight:600;">Required Files</h3>
    <ul style="margin-top:0;">
        <li><span style="font-weight:600;">PNG</span> :-
toolbar display</li>
        <li><span style="font-weight:600;">SVG</span> :-
canvas rendering</li>
    </ul>
    <h3 style="margin-bottom:5px; font-weight:600;">Grip /
Connector Info</h3>
    <p style="margin-top:0;">Grips are connection
points.<br>
    Defined with bottom-left origin: Y=0 at bottom, Y=100
at top.</p>
    """
    label = QLabel(help_text)
    label.setWordWrap(True)
    label.setTextFormat(Qt.RichText)
    label.setStyleSheet("font-family: 'Segoe UI',
system-ui, sans-serif; font-size: 14px;")
    layout.addWidget(label)

    close_btn = QPushButton("Close")
    close_btn.setObjectName("submitBtn")
    close_btn.clicked.connect(help_dialog.accept)
    layout.addWidget(close_btn)
    help_dialog.exec_()
```

## 7.3 Outcome

New users have immediate, context-sensitive guidance inside the dialog itself, reducing onboarding friction and the frequency of incorrectly defined custom components.

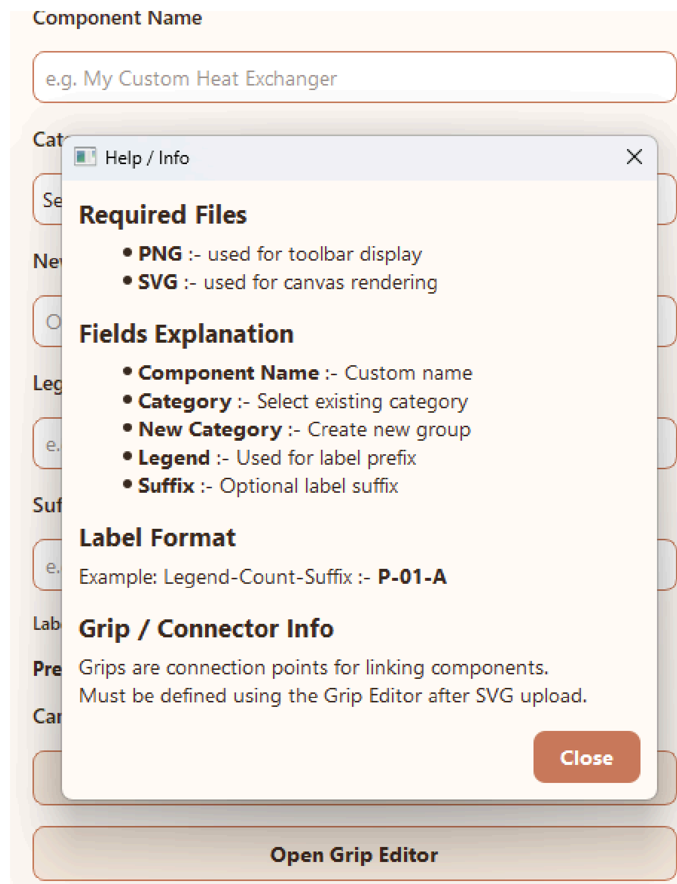


Figure 7.1: Interactive Help / Info modal on the Add New Symbol dialog

# Chapter 8 Task 6: Desktop Frontend Unit Testing Infrastructure

Pull Requests: PR #146 · PR #147 · PR #155 · PR #171

## 8.1 Problem Statement

The desktop frontend codebase had no automated tests. Regressions in critical subsystems (validation, routing, label generation, grip placement) were only discovered after the fact during manual testing. A structured unit testing framework was needed to provide confidence during future development.

## 8.2 Tasks Done

### 8.2.1 Initial Test Suite (PR #146)

Three focused test modules were created in a new desktop-frontend-UnitTests/ directory:

| Test Module | Coverage |
|---|---|
| test_validation.py | Tests the GraphValidator with synthetic adjacency lists: cycle detection, isolated node detection, inlet/outlet presence validation, and edge cases (empty graph, single node). |
| test_label_generation.py | Tests component label formatting and normalization: legend + suffix concatenation, zero-padded count generation, Unicode names, and metadata CSV loading. |
| test_routing.py | Tests orthogonal routing helpers: stub projection, grid snapping, ROI calculation, and A* path validity (no component collisions, correct orthogonality). |

### 8.2.2 Code Snippet: Cycle Validation Test Case

```
def test_loop_components_are_reported(make_component):
    comp_a = make_component("pump")
    comp_b = make_component("heater")
    comp_c = make_component("cooler")

    connections = [
        DummyConnection(comp_a, comp_b),
```

```
        DummyConnection(comp_b, comp_c),
        DummyConnection(comp_c, comp_a),
    ]

    result = GraphValidator([comp_a, comp_b, comp_c],
connections).validate()

    assert set(result["loops"]) == {comp_a, comp_b,
comp_c}
```

### 8.2.3 Grip Placement Bug Fix (PR #147)

While writing unit tests for the routing module, a pre-existing bug in grip placement for newly dropped components was discovered:

- **Root Cause:** The toolbar drag/drop payload serialized the component blueprint but omitted the grips list. Dropped components were reconstructed with zero grips.
- **Fix:** The grips list is now explicitly included in the drag payload and preserved throughout the drop pipeline.
- **Y-Axis Convention Fix:** The Grip Editor was serializing grip Y coordinates using a top-left origin, but the renderer expected a bottom-left origin. The serialization was corrected to flip the Y axis consistently.

### 8.2.4 Documentation (PR #155)

A README.md was added to desktop-frontend-UnitTests/ covering:

- Prerequisites and dependencies (pytest, PyQt5)
- How to run the full test suite and individual modules
- Brief description of each test file and what it covers

### 8.2.5 Component Resize Test Suite (PR #171)

Following the implementation of the component resize feature, a dedicated test_component_resize.py module was added:

- Corner handle scaling: Verifies proportional (aspect-ratio-locked) scaling.
- Mid-point handle stretching: Verifies independent width or height stretching.
- Minimum size enforcement: Verifies clamping at minimum component size (50×50 px).

- Collision boundary: Verifies a component cannot be resized into a neighbouring component.

## 8.3 Outcome

| Module | Focus | Tests |
| --- | --- | --- |
| test_validation.py | Graph validation logic | 12 |
| test_label_generation.py | Label formatting and metadata | 10 |
| test_routing.py | A* routing helpers | 14 |
| test_component_resize.py | Resize handles and collision | 22 |
| Total (after PR #178) | | 58 |

Note: The 4 modules authored during this internship established the core testing foundation. This framework was subsequently expanded by other team members into a comprehensive suite of 9 modules containing over 80 tests covering API serialization, canvas resources, and widget interactions.

# Chapter 9 Task 7: Component Resize System

---

Pull Requests: PR #166 · PR #167 · PR #178 · PR #175

## 9.1 Problem Statement

Components on the desktop canvas could not be resized after being placed. Users working on large diagrams needed to adjust component sizes to improve layout clarity, and standard proportional scaling was insufficient for elongated process equipment that needed independent width or height adjustment.

## 9.2 Tasks Done

### 9.2.1 Core Resize Implementation (PR #166)

The initial resize system added eight interactive resize handles around each selected component:

- **Four corner handles:** Enforce aspect-ratio-locked (proportional) scaling.
- **Handle hit testing:** Each handle is a small QRectF tested against mousePressEvent coordinates.
- **Live resize behavior:** The component's bounding rectangle updates in real time, with the SVG image re-rendered at the new dimensions. The operation is committed to the undo stack on mouseReleaseEvent.

### 9.2.2 Bug Fixes: Label Clipping and Collision Detection (PR #167)

Label Horizontal Clipping: When a component had a long name, the label text was clipped to the component's bounding box width. The fix dynamically expands the effective bounding box horizontally using QFontMetrics.boundingRect().

Label Detachment on Top-Handle Resize: When resizing from the top handle, the label anchor was recomputed incorrectly and visually detached. The fix recomputes the label anchor from the new bounding box on every resize event.

Collision Detection: During a resize drag, the proposed new bounding box is tested against all other components' sceneBoundingRect(). If it intersects any neighbour, the resize delta is clamped to the maximum non-overlapping size.

### 9.2.3 Independent Width and Height Resizing (PR #178)

Four additional mid-point handles were added:

- **Top and Bottom handles:** Stretch component height independently without changing width.
- **Left and Right handles:** Stretch component width independently without changing height.

Unlike corner handles, mid-point handles do not preserve the SVG aspect ratio. The SVG is stretched using independent scale_x and scale_y transforms. Connection pipe routing was updated to correctly recalculate grip world positions after an independent-axis resize.

The resize test suite was extended with new tests covering independent-axis stretching, bringing the total to 58 passing tests.

### 9.2.4 Code Snippet: Resizing Control & Collision Checks

```
if self.resize_handle in ["tl", "tr", "bl", "br"]:
    # Proportional corner scaling
    width_driver = abs(dw) / old_w >= abs(dh) / old_h
    if width_driver:
        new_w = max(min_size, raw_w)
        new_h = max(min_size, new_w / aspect)
    else:
        new_h = max(min_size, raw_h)
        new_w = max(min_size, new_h * aspect)
else:
    # Independent width/height midpoint stretching
    new_w = max(min_size, raw_w) if sx != 0 else old_w
    new_h = max(min_size, raw_h) if sy != 0 else old_h

new_rect = QRectF(new_left, new_top, new_w, new_h)

# Collision Boundary Check
has_collision = False
for other in parent.components:
    if other != self:
        if new_rect.adjusted(-35, -35, 35,
35).intersects(other.logical_rect):
```

```
            has_collision = True
            break
```

## 9.3 Outcome

The complete resize system provides:

- Four corner handles — aspect-ratio-locked proportional scaling.
- Four mid-point handles — independent width or height stretching.
- Collision detection — prevents overlapping during resize.
- Live label tracking — label stays attached during all resize operations.
- Connection routing update — grips re-snap correctly after independent-axis resize.
- 58 automated tests — full regression coverage for all resize behaviors.

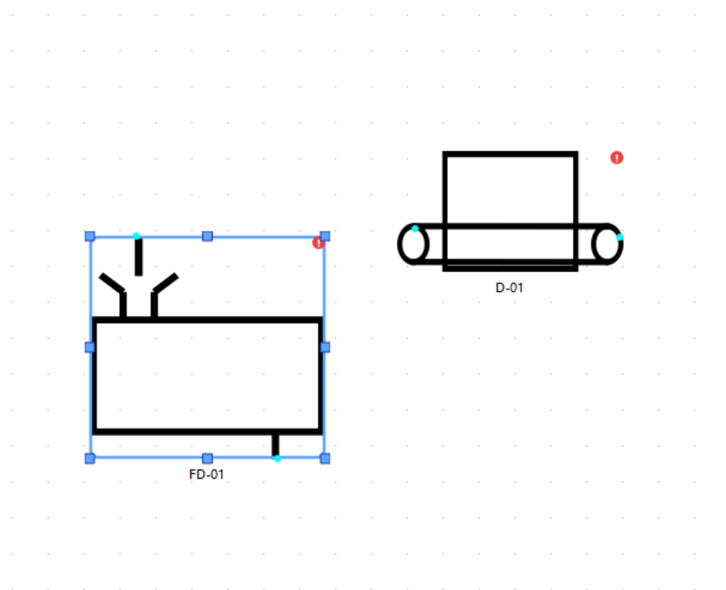


Figure 9.1: Component sizing system showing resize handles and independent width/height stretching

# Chapter 10 Task 8: Desktop Application UI/UX Polish

---

Pull Requests: PR #106 · PR #107 · PR #175

## 10.1 Problem Statement

As the desktop frontend matured, several user experience (UX) and visual styling issues became apparent. Saving new projects lacked safeguards, the recent projects list looked cluttered, and the landing page needed visual refinement to match modern desktop application standards.

## 10.2 Tasks Done

### 10.2.1 Save Options for New Projects (PR #106)

Previously, clicking 'Save' on a completely new (unsaved) project could silently fail or overwrite a default file. The save behavior was intercepted so that the first save of any new project automatically forces the 'Save As' dialogue, ensuring the user intentionally chooses a file location.

### 10.2.2 Recent Projects Card Styling (PR #107)

The list of recent projects was transformed from a plain text list into individually styled, interactive cards. Each card displays the project name, last modified date, and path, significantly improving readability.

### 10.2.3 Landing Page and UI Styling (PR #175)

- Adjusted card and list item paddings for better visual balance.
- Refined scrollbar and tab bar styling for both light and dark themes.
- Fixed emoji rendering in component labels and tab names.

## 10.3 Outcome

These collective UI/UX improvements resulted in a much more polished, professional, and robust application entry point. Users are now guided safely through project creation and saving, while enjoying a visually pleasing landing interface.

# Chapter 11 Conclusions

## 11.1 Tasks Accomplished

Throughout the FOSSEE Internship, I contributed to the Desktop Frontend and full-stack subsystems of the Chemical PFD Builder across several pull requests covering the following major areas:

| # | Contribution | PRs |
|---|---|---|
| 1 | FOSSEE Selection Task (Chemical Equipment Visualizer full-stack Django, React, and PyQt5 application) | Selection Task |
| 2 | Graph-Based Validation Engine (DFS loops, BFS flow, isolated nodes, inlet/outlet enforcement) | #115–#128 |
| 3 | A* Orthogonal Routing Engine (grid search, turn penalties, crossing jump arcs) | #132 |
| 4 | Local legend fallbacks for InflowLine and OutflowLine components | #134 |
| 5 | Cross-platform component sizing synchronization | #141 |
| 6 | Interactive Help Modal for the Add New Symbol dialog | #143 |
| 7 | Unit testing infrastructure (58 passing tests across 4 modules) | #146, #147, #155, #171 |
| 8 | Component resize system: 8-handle proportional + independent-axis stretch with collision detection | #166, #167, #178 |
| 9 | Desktop Application UI/UX Polish (Save Options, Recent Projects, Landing Page Styles) | #106, #107, #175 |

## 11.2 Skills Developed

### 11.2.1 Technical Skills

- **Python and PyQt5:** Event-driven GUI programming, custom widget painting, undo/redo stacks, signal/slot architecture, and drag/drop pipelines.
- **Graph Algorithms:** Applied DFS, BFS, and A* heuristic search in real production code, optimizing for correctness and performance.
- **Software Testing:** Built a structured pytest test suite from scratch covering stateful GUI component behavior, with mocking and parameterized test cases.
- **Geometric Programming:** Hit testing, bounding box operations, coordinate system transformations, and collision detection for a 2D canvas.
- **Open-Source Workflow:** Pull request authoring, code reviews, rebasing, commit hygiene, and writing meaningful PR descriptions.

### 11.2.2 Professional Skills

- Decomposed large features into incremental, reviewable pull requests.
- Maintained backward compatibility while refactoring critical subsystems.
- Investigated and fixed regressions introduced by parallel team contributions.
- Wrote documentation and test README files for future contributors.
- Communicated technical decisions clearly in PR descriptions and commit messages.

## 11.3 Future Scope

- **Real-Time Collaboration:** Extend the desktop frontend to support multi-user editing sessions through WebSocket-based state synchronization.
- **Simulation Integration:** Connect the validation engine to chemical process simulators (e.g., DWSIM) for real-time mass balance and energy balance checks.
- **Advanced Routing:** Explore Rectilinear Steiner Tree algorithms to further minimize total wire length while maintaining orthogonality.

- **Extended Test Coverage:** Add integration tests simulating full user workflows (drop, connect, validate, export) using pytest-qt.
- **P&ID Support:** Extend the symbol library and validation rules to support Piping and Instrumentation Diagrams.

# Bibliography

[1] FOSSEE Project, FOSSEE Website, https://fossee.in/, Accessed: 2026-06-27.

[2] FOSSEE/Chemical-PFD-Web-Desktop GitHub Repository, https://github.com/FOSSEE/Chemical-PFD-Web-Desktop, Accessed: 2026-06-27.

[3] Peter E. Hart, Nils J. Nilsson, Bertram Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", IEEE Transactions on Systems Science and Cybernetics, 1968.

[4] PyQt5 Documentation, Qt for Python, https://doc.qt.io/qtforpython/, Accessed: 2026-06-27.

[5] Python Software Foundation, unittest — Unit testing framework, https://docs.python.org/3/library/unittest.html, Accessed: 2026-06-27.