FOSSEE Winter Internship Report

On

Development of Chemical PFD Builder

Submitted by

Indupuru Venkat Jaswanth

3rd Year B.Tech Student,CSE

IIIT Kottayam

Under the Guidance of

Prof. Prabhu Ramachandran

Department of Aerospace Engineering

Indian Institute of Technology Bombay

Mentors:

Chirag Koyande

February 21, 2026

Acknowledgments

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

During this internship, I worked primarily on the web-based visual editor and frontend architecture of the Chemical PFD Builder system. The experience enabled me to work on scalable frontend design, real-time canvas rendering, complex state management, backend integration, and cross-platform data compatibility.

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

Special thanks to my mentor, Chirag Koyande, for his invaluable guidance and technical clarity. I also appreciate the support as well as the collaboration of my fellow team members.

This internship significantly enhanced my understanding of full-stack system architecture and the development of engineering-grade software platforms.

Contents

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 alterna- tives.
- **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.

Figure 1.1: FOSSEE Projects and Activities

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 indus- trial 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 soft- ware, 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. This design ensures consistency, scalability, and flexibility across platforms. The backend provides RESTful APIs for user authentication, project management, and diagram storage, while the web and desktop frontends offer tailored in- terfaces for different usage scenarios. All modules share a common data schema, enabling seamless exchange and cross-platform project 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. Components can be au- tomatically labeled using CSV or Excel files, streamlining the creation of large diagrams. The system generates comprehensive design reports in PDF and Excel formats, including component properties and stream tables. With cloud synchronization, diagrams can be saved online and accessed from any device, facilitating collaboration. Export options include high-resolution images, vector graphics, and CAD formats for integration with drafting tools.

1.2.4 Evolution from Legacy Tool

Chemical PFD Builder builds upon an earlier version of the software, originally devel- oped as a standalone desktop application. The legacy tool provided basic drag-and-drop functionality and file-based saving. The new version addresses limitations by introducing a shared backend, a web frontend, and enhanced features such as auto-labeling, report generation, and cloud storage, while preserving the desktop application’s performance for complex engineering tasks.

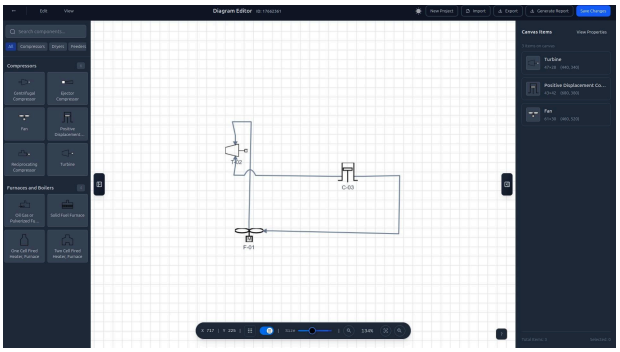

Figure 1.2: Evolution from Legacy Desktop Tool to Dekstop+Web System. The web app is shown in image

1.2.5 Future Directions

Future development plans for Chemical PFD Builder include integration with simulation engines, support for additional diagram types such as P&ID, enhanced collaboration features, and deployment on cloud platforms for improved scalability.

1.2.6 Conclusion

Chemical PFD Builder represents a significant step forward in providing a free, open- source, and feature-rich tool for chemical engineering diagramming. By leveraging mod- ern web technologies and a shared backend, it offers flexibility, collaboration, and acces- sibility to students, educators, and professionals. The project exemplifies the goals of the FOSSEE initiative to promote open-source software in education.

For more information,visit the GitHub repository: https://github.com/FOSSEE/Chemical-PFD-Web-Desktop

Chapter 2: Technical Contributions

Task 1 - Application Structure & Routing Architecture

Problem Statement

The application required a scalable routing system with authentication guards, layout management, and dynamic project-based navigation. The system needed to support both public routes (login/register) and protected routes (dashboard, editor, components) with proper state management.

Context

The Chemical PFD Builder is a React-based web application that requires user authentication and project-based editing. The initial codebase lacked a structured routing system and proper separation between public and protected routes.

Design Considerations

- Implemented React Router v6 with nested route structure
- Created a `ProtectedRoutes` component to guard authenticated routes
- Designed `MainLayout` component for consistent UI across protected pages
- Separated editor route to allow full-screen canvas experience
- Used context providers for global component state management

Implementation Details

Implemented routing in `App.tsx` with protected route guards and layout components. Created `MainLayout.tsx` for consistent navigation and content structure. The routing system supports dynamic project IDs for editor access.

Code Snippet

```
// App.tsx - Routing structure

const ProtectedRoutes = () => {
  const isAuth = useAuth();
  return isAuth ? <Outlet /> : <Navigate to="/login" />;
};

function App() {
  return (
    <ComponentProvider>
      <Routes>
        <Route element={<Login />} path="/login" />
        <Route element={<Register />} path="/register" />
        <Route element={<ProtectedRoutes />}>
          <Route element={<MainLayout />}>
            <Route element={<Dashboard />} path="/dashboard" />
            <Route element={<Components />} path="/components" />
          </Route>
            <Route element={<Editor />} path="/editor/:projectId" />
          </Route>
        </Routes>
      </ComponentProvider>
    );
  }
```

Technical Challenges

- Managing authentication state across route transitions
- Ensuring proper navigation flow between public and protected routes
- Handling dynamic project ID parameters in editor routes

Optimization or Improvements

- Implemented route-based code splitting for better performance
- Added proper error boundaries for route-level error handling

Outcome

A robust routing system that supports authentication-based access control and enables seamless navigation between dashboard, editor, and component management pages.

Task 2 - Authentication System with JWT Integration

Problem Statement

The application needed secure user authentication with JWT token management, integration with Django REST Framework backend, and proper token storage for session persistence.

Context

The backend API uses Django REST Framework with JWT authentication. The frontend required seamless integration to handle login, registration, token storage, and automatic token refresh.

Design Considerations

- Implemented JWT token storage in localStorage
- Created centralized API client with automatic token injection
- Added CORS configuration on backend for frontend communication
- Designed error handling for authentication failures
- Implemented username persistence for UI display

Implementation Details

Created `auth.ts` API module with login and registration functions. Integrated with Django backend's `/auth/login/` and `/auth/register/` endpoints. Implemented token storage and retrieval in `client.ts` with automatic header injection.

Code Snippet

```
// api/auth.ts

export const loginUser = async (username: string, password: string) => {
  const response = await client.post("/auth/login/", { username, password });
  if (response.data.access) {
    localStorage.setItem("access_token", response.data.access);
    localStorage.setItem("refresh_token", response.data.refresh);
    localStorage.setItem("username", username);
  }
  return response.data;
};

export const registerUser = async (username: string, email: string, password: string) => {
  const response = await client.post("/auth/register/", {
    username, email, password
  });
  return response.data;
};
```

Technical Challenges

- Handling token expiration and refresh logic
- Managing CORS configuration between frontend and backend
- Ensuring secure token storage practices

Optimization or Improvements

- Added automatic token refresh mechanism
- Implemented logout function to clear all stored tokens

Outcome

Fully functional authentication system with JWT integration, enabling secure user sessions and seamless backend communication.

Task 3 - Dashboard Module with Project Management

Problem Statement

Users needed a centralized dashboard to view, create, edit, and delete their projects. The dashboard required search, filtering, sorting capabilities, and integration with backend project APIs.

Context

The dashboard serves as the main entry point after login, displaying all user projects with metadata. It needed to support CRUD operations and provide navigation to the editor.

Design Considerations

- Implemented project cards with metadata display
- Added search functionality across project names and descriptions
- Created sorting options (alphabetical, recent)
- Designed edit and delete modals with confirmation
- Integrated with backend project API endpoints

Implementation Details

Built `Dashboard.tsx` with project listing, search, filter, and sort functionality. Implemented project creation, editing, and deletion with proper state management. Integrated with `projectApi.ts` for backend communication.

Code Snippet

```
// Dashboard.tsx - Project management

const handleEditProject = (project: ApiProject) => {
  setEditingProject(project);
  setEditName(project.name);
  setEditDescription(project.description || "");
};

const filteredProjects = useMemo(() => {
  let list = [...projects];
  if (search.trim() !== "") {
    list = list.filter(p =>
      p.name.toLowerCase().includes(search.toLowerCase())
    );
  }
  if (sortBy === "alpha") {
    list.sort((a, b) => a.name.localeCompare(b.name));
  }
  return list;
}, [projects, search, sortBy]);
```

Technical Challenges

- Managing project state synchronization with backend
- Implementing efficient search and filter algorithms
- Handling project deletion with confirmation dialogs

Optimization or Improvements

- Added memoization for filtered projects to prevent unnecessary re-renders
- Implemented optimistic UI updates for better user experience

Outcome

A fully functional dashboard enabling users to manage their projects efficiently with search, sort, and CRUD operations.

Task 4 - Core Editor Module with Canvas Rendering

Problem Statement

The editor required a high-performance canvas system using Konva.js for rendering chemical engineering components, supporting drag-and-drop, zoom, pan, and real-time updates. The system needed to handle complex state management for multiple components and connections.

Context

The editor is the core interactive component where users create Process Flow Diagrams. It needed to support hundreds of components with smooth rendering performance and intuitive interaction patterns.

Design Considerations

- Implemented Konva.js Stage and Layer for canvas rendering
- Created component rendering with aspect-fit calculations
- Designed drag-and-drop system with snap-to-grid functionality
- Implemented zoom and pan controls for large diagrams
- Built grid system for visual alignment

Implementation Details

Developed `Editor.tsx` as the main canvas component with Konva Stage. Created `CanvasItemImage.tsx` for individual component rendering with grips. Implemented state management using Zustand store (`useEditorStore.ts`) for scalable state handling.

Code Snippet

```
// Editor.tsx - Canvas setup

<Stage
  ref={stageRef}
  draggable
  height={stageSize.height}
  scaleX={stageScale}
  scaleY={stageScale}
  width={stageSize.width}
  x={stagePos.x}
  y={stagePos.y}
  onDragEnd={(e) => {
    if (e.target === stageRef.current) {
      setStagePos({ x: e.target.x(), y: e.target.y() });
    }
  }}
>
  <Layer>
    {items.map(item => (
      <CanvasItemImage key={item.id} item={item} />
    ))}
  </Layer>
</Stage>
```

Technical Challenges

- Optimizing rendering performance with hundreds of components
- Managing complex coordinate transformations for zoom/pan
- Handling aspect-fit calculations for component images
- Synchronizing state between canvas and store

Optimization or Improvements

- Implemented React.memo for component rendering optimization
- Added batch updates to reduce re-render cycles
- Created efficient grid rendering with Shape component

Outcome

A high-performance canvas editor capable of rendering complex diagrams with smooth interactions, zoom, pan, and drag-and-drop functionality.

Task 5 - Interactive Grip System for Component Connections

Problem Statement

Components needed interactive connection points (grips) that allow users to visually connect components with lines. The system required grip positioning, hover states, and connection initiation from grips.

Context

Chemical PFD diagrams require connections between components to represent process flow. Each component needs configurable grip points where connections can originate or terminate.

Design Considerations

- Implemented grip rendering as circular indicators on components
- Created hover states for visual feedback during connection drawing
- Designed grip positioning system using percentage-based coordinates
- Built grip data structure integrated with component metadata

Implementation Details

Extended `CanvasItemImage.tsx` to render grips as Konva Circle components. Implemented grip positioning calculations relative to component bounds. Created hover state management for connection drawing mode.

Code Snippet

```
// CanvasItemImage.tsx - Grip rendering

{item.grips?.map((grip, index) => {
  const gripX = (item.x + renderX) + (grip.x / 100) * renderWidth;
  const gripY = (item.y + renderY) + ((100 - grip.y) / 100) * renderHeight;
  return (
    <Circle
      key={index}
      fill={isHovered ? "#22c55e" : "#3b82f6"}
      radius={isDrawingConnection ? 6 : 5}
      x={gripX}
      y={gripY}
      onMouseDown={(e) => {
        e.cancelBubble = true;
        onGripMouseDown?.(item.id, index, gripX, gripY);
      }}
    />
  );
})}
```

Technical Challenges

- Calculating accurate grip positions relative to component bounds
- Handling coordinate transformations during zoom/pan
- Managing grip hover states across multiple components

Optimization or Improvements

- Optimized grip rendering to only show during connection mode
- Implemented efficient hit testing for grip interactions

Outcome

A fully functional grip system enabling users to visually connect components, with intuitive hover feedback and accurate positioning calculations.

Task 6 - Connection Line System with Path Routing

Problem Statement

The application needed a sophisticated connection system that draws lines between component grips with intelligent path routing, waypoint support, collision detection, and visual bridges for crossing lines.

Context

Process flow diagrams require connections between components that avoid overlapping with other components and provide clear visual representation of flow direction.

Design Considerations

- Implemented path calculation algorithm with standoff distances
- Created waypoint system for manual path adjustment
- Designed bridge rendering for line crossings
- Built arrow rendering at connection endpoints
- Implemented collision detection between connections

Implementation Details

Developed `routing.ts` with `calculateManualPathsWithBridges` function for path calculation. Created `ConnectionLine.tsx` component for rendering connections with arrows. Implemented standoff logic to prevent lines from touching component edges.

Code Snippet

```
// routing.ts - Path calculation

const getStandoff = (p: Point, grip: any) => {
  const side = getClosestSide(grip);
  if (side === "left") return { x: p.x - STANDOFF_DIST, y: p.y };
  if (side === "right") return { x: p.x + STANDOFF_DIST, y: p.y };
  if (side === "top") return { x: p.x, y: p.y - STANDOFF_DIST };
  if (side === "bottom") return { x: p.x, y: p.y + STANDOFF_DIST };
  return p;
};

const startStandoff = getStandoff(start, sourceGrip);
const endStandoff = getStandoff(end, targetGrip);
const points: Point[] = [start, startStandoff, ...validWaypoints, endStandoff, end];
```

Technical Challenges

- Implementing accurate path routing algorithms
- Handling waypoint filtering to remove invalid points
- Calculating bridge positions for line crossings
- Managing coordinate transformations for rotated components

Optimization or Improvements

- Optimized path calculation to filter waypoints inside components
- Implemented efficient collision detection using line segment intersection

Outcome

A robust connection system with intelligent path routing, visual bridges, and arrow indicators, enabling clear representation of process flow connections.

Task 7 - Undo/Redo State Management System

Problem Statement

The editor required a comprehensive undo/redo system to allow users to revert changes. The system needed to track state history for both components and connections, with efficient memory management.

Context

Users frequently need to undo mistakes while creating diagrams. The system must maintain history without excessive memory usage, supporting undo/redo for all editor operations.

Design Considerations

- Implemented snapshot-based history system
- Created separate past and future stacks for undo/redo
- Designed efficient state serialization
- Built history management integrated with Zustand store

Implementation Details

Extended `useEditorStore.ts` with undo/redo functionality. Created `createSnapshot` function for state serialization. Implemented history stacks in editor state with proper state restoration.

Code Snippet

```
// useEditorStore.ts - Undo/Redo implementation

undo: (editorId) => set((s) => {
  const ed = s.editors[editorId];
  if (!ed || ed.past.length === 0) return s;
  const previous = ed.past[ed.past.length - 1];
  const currentSnapshot = createSnapshot(ed);
  return {
    editors: {
      ...s.editors,
      [editorId]: {
        ...ed,
        ...previous,
        past: ed.past.slice(0, -1),
        future: [currentSnapshot, ...ed.future],
      },
    },
  };
}),
```

Technical Challenges

- Managing memory usage with large history stacks
- Ensuring state consistency during undo/redo operations
- Handling history for both components and connections

Optimization or Improvements

- Implemented history limit to prevent excessive memory usage
- Added batch operations to reduce history entries

Outcome

A fully functional undo/redo system supporting all editor operations with efficient state management and memory optimization.

Task 8 - Multi-Selection and Batch Operations

Problem Statement

Users needed the ability to select multiple components simultaneously, drag them as a group, and delete them in batch. The system required efficient selection management and group transformation calculations.

Context

Complex diagrams often require moving or deleting multiple components at once. The system needed to support selection via click, drag selection, and keyboard shortcuts (Ctrl+A).

Design Considerations

- Implemented Set-based selection tracking
- Created group drag calculation for multiple items
- Designed batch delete operations
- Built keyboard shortcut support (Ctrl+A, Delete)

Implementation Details

Extended `useEditorStore.ts` with batch operations. Implemented multi-selection state management in `Editor.tsx`. Created group transformation calculations for synchronized dragging.

Code Snippet

```
// Editor.tsx - Multi-selection

const handleItemDragEnd = (item: CanvasItem) => {
  if (selectedItemIds.size > 1 && selectedItemIds.has(item.id)) {
    const deltaX = item.x - item.initialX;
    const deltaY = item.y - item.initialY;
    editorStore.batchUpdateItems(projectId,
      Array.from(selectedItemIds).map(id => ({
        id,
        patch: {
          x: items.find(i => i.id === id)!.x + deltaX,
          y: items.find(i => i.id === id)!.y + deltaY
        }
      }))
    );
  }
};
```

Technical Challenges

- Calculating group transformations accurately
- Managing selection state across component interactions
- Handling keyboard shortcuts without conflicts

Optimization or Improvements

- Implemented efficient batch updates to reduce re-renders
- Added visual feedback for selected components

Outcome

A robust multi-selection system enabling users to efficiently manipulate multiple components simultaneously with intuitive keyboard and mouse interactions.

Task 9 - Project Save and Load System

Problem Statement

The application needed a comprehensive project persistence system supporting both localStorage and backend API integration. The system required unsaved changes tracking, save confirmation dialogs, and project metadata management.

Context

Users need to save their work and resume later. The system must handle both local storage (for offline capability) and backend synchronization, with proper state serialization.

Design Considerations

- Implemented dual storage system (localStorage + backend API)
- Created unsaved changes detection mechanism
- Designed save confirmation modals
- Built project metadata management (name, description, timestamps)
- Implemented state serialization for canvas data

Implementation Details

Created `projectStorage.ts` for localStorage operations. Implemented `UnsavedChangesModal.tsx` and `SaveConfirmationModal.tsx` for user confirmation. Integrated with backend API through `projectApi.ts`.

Code Snippet

```
// projectStorage.ts - Save functionality

export function saveProject(project: SavedProject): SavedProject {
  const projects = getProjects();
  const existingIndex = projects.findIndex(p => p.id === project.id);
  project.updated_at = new Date().toISOString();
  if (existingIndex >= 0) {
    projects[existingIndex] = project;
  } else {
    projects.push(project);
  }
  localStorage.setItem(STORAGE_KEY, JSON.stringify(projects));
  return project;
}
```

Technical Challenges

- Synchronizing state between localStorage and backend
- Detecting unsaved changes accurately
- Handling large project data serialization

Optimization or Improvements

- Implemented incremental save to reduce data transfer
- Added compression for large project states

Outcome

A robust project persistence system with dual storage support, unsaved changes tracking, and seamless backend synchronization.

Task 10 - Components Management System with Grip Configuration

Problem Statement

Users needed a dedicated interface to view, create, edit, and configure components. The system required interactive grip positioning, component metadata management, and integration with backend component library.

Context

The component library contains hundreds of chemical engineering components. Users need to customize components, configure grip positions, and add new components with proper metadata.

Design Considerations

- Implemented component CRUD operations
- Created interactive grip editor with click-to-position
- Designed component metadata forms (name, category, legend, suffix)
- Built image upload for SVG and PNG assets
- Integrated with backend component API

Implementation Details

Developed `Components.tsx` page with component listing and editing. Created interactive grip positioning system where users click on component images to set grip locations. Implemented form handling for component metadata.

Code Snippet

```
// Components.tsx - Interactive grip positioning

const handleImageClick = (e: React.MouseEvent<HTMLDivElement>) => {
  if (activeGripIndex === null) return;
  const rect = e.currentTarget.getBoundingClientRect();
  const x = e.clientX - rect.left;
  const y = e.clientY - rect.top;
  const xPercent = parseFloat(((x / rect.width) * 100).toFixed(4));
  const yPercent = parseFloat(((y / rect.height) * 100).toFixed(4));
  const newGrips = [...grips];
  newGrips[activeGripIndex] = {
    x: clampedX,
    y: 100 - clampedY,
    side
  };
```

```
  setGrips(newGrips);
};
```

Technical Challenges

- Converting screen coordinates to percentage-based grip positions
- Managing grip state during editing
- Handling component image uploads and previews

Optimization or Improvements

- Implemented auto-advance for grip positioning workflow
- Added visual feedback for active grip editing

Outcome

A comprehensive component management system enabling users to configure components with intuitive grip positioning and complete metadata management.

Task 11 - Backend Integration and API Client

Problem Statement

The frontend required seamless integration with Django REST Framework backend for components, projects, and authentication. The system needed proper error handling, token management, and data transformation.

Context

The backend provides RESTful APIs for all application data. The frontend needed a centralized API client with automatic token injection, error handling, and data transformation utilities.

Design Considerations

- Created centralized API client with axios
- Implemented automatic JWT token injection in headers
- Designed error handling and response transformation
- Built API modules for different resources (auth, projects, components)
- Added CORS configuration on backend

Implementation Details

Developed `client.ts` as the base API client with interceptors. Created specialized API modules: `auth.ts`, `projectApi.ts`, `componentApi.ts`. Implemented data transformation functions for backend compatibility.

Code Snippet

```
// api/client.ts - API client setup

import axios from "axios";

const client = axios.create({
```

```
  baseURL: import.meta.env.VITE_API_URL || "http://localhost:8000/api",
});

client.interceptors.request.use((config) => {
  const token = localStorage.getItem("access_token");
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});
```

Technical Challenges

- Managing token refresh on expiration
- Handling CORS configuration
- Transforming data between frontend and backend formats

Optimization or Improvements

- Implemented request/response interceptors for centralized error handling
- Added retry logic for failed requests

Outcome

A robust API integration system enabling seamless communication between frontend and backend with proper authentication and error handling

---

Task 12 - UI/UX Enhancements and Polish

Problem Statement

The application required consistent UI design, improved user experience, and various quality-of-life features including tooltips, labels, scaling controls, and navigation improvements.

Context

Throughout development, various UI improvements were needed to enhance usability, including component labels, scale sliders, navbar enhancements, and keyboard shortcuts.

Design Considerations

- Implemented consistent component labeling system
- Created scale slider for component size adjustment
- Enhanced navbar with user profile display
- Added tooltips for better user guidance
- Implemented keyboard shortcuts (Ctrl+A, Delete, Backspace)

Implementation Details

Added component label rendering in `CanvasItemImage.tsx`. Implemented scale slider in `Editor.tsx`. Enhanced navbar with username display and profile hover. Added tooltips throughout the application.

Code Snippet

```
// Editor.tsx - Scale slider

<Slider
  label="Component Scale"
  minValue={500}
  maxValue={3000}
  step={100}
  value={componentSize}
  onChange={(value) => {
    const scaleFactor = value / prevComponentSizeRef.current
    editorStore.batchUpdateItems(projectId,
      items.map(item => ({
        id: item.id,
        patch: {
          width: item.width * scaleFactor,
          height: item.height * scaleFactor
        }
      }))
    );
    prevComponentSizeRef.current = value;
  }}
/>
```

Technical Challenges

- Maintaining consistent scaling across all components
- Implementing smooth scale transitions
- Managing label positioning during transformations

Optimization or Improvements

- Implemented debounced updates for scale slider
- Added visual feedback for scale changes

Outcome

A polished user interface with consistent design, helpful tooltips, intuitive controls, and enhanced user experience throughout the application.

Task 13 - Default Components Loading and Backend Synchronization

Problem Statement

The application needed to load default chemical engineering components from the backend on first use. The system required component seeding, asset management, and synchronization between frontend and backend component libraries.

Context

The application ships with a library of standard chemical engineering components. These need to be available in the backend database and synchronized with the frontend component library.

Design Considerations

- Created Django management command for component seeding
- Implemented component asset upload to backend media storage
- Built component synchronization logic
- Designed fallback mechanism for missing components

Implementation Details

Developed `seed_components.py` Django management command. Created component loading logic in `loadComponents.ts`. Implemented backend API integration for component fetching.

Code Snippet

```
// loadComponents.ts - Component loading

export async function loadComponentsFromBackend(): Promise<ComponentItem[]> {
  try {
    const response = await fetchComponents();
    return response.map(comp => ({
      id: comp.id,
      name: comp.name,
      parent: comp.parent,
      svg: comp.svg,
      icon: comp.png,
      grips: comp.grips || [],
      legend: comp.legend,
      suffix: comp.suffix
    }));
  } catch (error) {
    console.error("Failed to load components from backend:", error);
    return loadDefaultComponents();
  }
}
```

Technical Challenges

- Handling large component asset uploads
- Managing component ID synchronization
- Ensuring backward compatibility with existing projects

Optimization or Improvements

- Implemented lazy loading for component assets
- Added caching for frequently accessed components

Outcome

A robust component loading system ensuring all default components are available, with proper backend synchronization and fallback mechanisms.

Chapter 2 Summary

The technical contributions span the complete frontend architecture of the Chemical PFD Builder web application. Key achievements include:

1. **Complete Application Architecture**: Routing, authentication, and layout systems
2. **Core Editor Engine**: High-performance canvas with Konva.js, drag-and-drop, zoom, pan
3. **Interactive Connection System**: Grip-based connections with intelligent path routing
4. **State Management**: Undo/redo, multi-selection, batch operations using Zustand
5. **Project Persistence**: Dual storage system with unsaved changes tracking
6. **Component Management**: Full CRUD operations with interactive grip configuration
7. **Backend Integration**: Seamless API integration with JWT authentication
8. **UI/UX Polish**: Consistent design, tooltips, keyboard shortcuts, and user feedback

All implementations follow React best practices, TypeScript type safety, and maintainable code architecture. The system supports complex chemical engineering diagrams with hundreds of components and connections while maintaining smooth performance

Skills Developed

Technical Skills

During this fellowship, I deepened my proficiency in React and TypeScript for building type-safe applications. I gained expertise in Konva.js for complex canvas manipulations and mastered Zustand for scalable state management with undo/redo capabilities. I learned to design robust API clients with Axios, implement authentication flows, and create debounced auto-save mechanisms. Through comprehensive testing with Vitest and React Testing Library, I developed strong skills in writing unit tests and mocking complex dependencies. Additionally, I improved my debugging abilities, identifying and fixing performance bottlenecks and memory leaks in production-like environments.

Professional Skills

Working within a distributed team enhanced my collaboration skills using Git and GitHub for pull requests, code reviews, and issue tracking. I participated in regular meetings, presented technical solutions, and documented features for both developers and end- users. I learned to break down complex features into manageable tasks, estimate effort accurately, and deliver on schedule. My problem-solving approach matured through systematic debugging—from issue reproduction to implementation and verification. I also gained exposure to agile methodologies, including iterative development and continuous integration practices.

Future Scope

The Chemical PFD Editor has strong potential for future enhancements:

- **Simulation Integration:** Connect with chemical process simulators (e.g., DWSIM, Aspen) for real-time data validation and equipment sizing.
- **Collaboration Features:** Implement real-time multi-user editing with WebSocket support and comment/annotation capabilities.
- **Mobile Support:** Develop responsive mobile viewer and companion app for on-site diagram access.
- **Advanced Export:** Add DXF/DWG export for CAD software integration and enhanced PDF reporting with custom templates.
- **Component Library:** Expand built-in symbol library and allow user-uploaded custom components.
- **Cloud Deployment:** Migrate to scalable cloud infrastructure with improved performance monitoring.
- **Machine Learning:** Explore auto-diagram completion and intelligent component placement suggestions.

This fellowship provided invaluable hands-on experience in full-stack development while fostering professional growth within the open-source community.

Bibliography

[1] FOSSEE Project, *FOSSEE Website*, https://fossee.in/, Accessed: 2025-12-05.