FOSSEE Semester Long Internship Report

On

Osdag on Cloud

Submitted by

Raghav Sharma

Under the Guidance of

Prof. Siddhartha Ghosh

Department of Civil Engineering

Indian Institute of Technology Bombay

Mentors:

Parth Karia

Ajmal Babu M S

Ajinkya Dahale

June 16, 2025

Contents

Chapter 1

Introduction

This chapter provides a comprehensive overview of both “Osdag” and its cloud-based counterpart “Osdag on Cloud,” while also highlighting the distinctions between these two products. Additionally, it offers a succinct overview of the intended audience for these solutions.

1.1 What is Osdag?

“Osdag” is a trans-formative exemplar of Free/Libre and Open Source Software (FLOSS), meticulously crafted to redefine steel structure design. Rooted in Python, this software embodies collaborative excellence and technical finesse. The integration of PythonOCC, a 3D CAD modeling framework, elevates “Osdag” beyond the ordinary. By intertwining code with visual representation, the software offers an immersive platform for users to engage with and refine their structural concepts.

The “Share alike” policy, advocated by FOSSEE, adds a layer of significance. Beyond open source principles, it envisions a perpetually evolving software landscape, aligned with industry and academic needs. In essence, “Osdag” transcends software conventions, embodying a collaborative journey that empowers engineers, architects, and learners. In a rapidly changing world, it stands as a beacon of progress, fueled by the ethos of open source collaboration.

1.2 Challenges with the Osdag software

While undoubtedly a valuable tool, the “Osdag” software is not without its challenges. One prominent obstacle is the time-intensive nature of setting up the desktop application on local systems, a concern shared by both developers and users. The process can often be compounded by potential bugs stemming from version disparities or operating system variations. This complexity can lead to a range of issues during setup. Enterprises into this backdrop, “Osdag on Cloud” emerges as a promising solution.

1.3 What is Osdag on Cloud?

“Osdag on Cloud” is an innovative approach that circumvents the intricate setup process altogether, offering a streamlined alternative. By shifting the software’s operations to the cloud, users and developers can sidestep compatibility issues and operating system nuances. This forward-looking initiative not only enhances convenience but also empowers seamless accessibility, ushering in a new era of user experience.

“Osdag on Cloud” represents the cloud-based iteration of Osdag built with React.js and Django. With this version, users are relieved of the necessity to download the Osdag desktop application onto their computers. Instead, they can readily harness the complete spectrum of Osdag features by simply navigating to the website. This streamlined approach allows users to access the software’s functionalities without any local installations. Notably, intricate tasks such as computations and report generation, design computations, etc. are seamlessly handled within the cloud environment, further enhancing user convenience.

1.4 Who can use Osdag on Cloud?

“Osdag on Cloud” is created for both educational purposes and industry professionals. As FOSS is currently funded by MHRD, the Osdag team is developing the software in such a way that it can be used by students during their academics to provide them with a better insight into the subject. Osdag is designed to be usable by anyone, from novices to professionals. Its simple user interface makes it more flexible and attractive than other software options. Video tutorials are available to help users get started. You can access the video tutorials for Osdag here.

Chapter 2

Screening Task: Osdag Web Module Development

The screening task served as an initial assessment to demonstrate proficiency in full-stack development by creating a new module for the Osdag-on-Cloud application. The task was to build a "Beam-to-Beam Splice (Cover Plate Bolted)" design module from the ground up.

2.1 Project Overview

2.1.1 Objective

The primary objective was to develop a fully functional web module that mirrors the capabilities of the existing desktop version. This required adherence to several key standards:

- **Engineering Standards:** All design calculations and checks had to be based on the Indian Standard **IS 800:2007** for steel structures.
- **Software Standards:** The project was to be developed following Free/Libre and Open Source Software (FLOSS) practices, licensed under the **GNU LGPL v3**.
- **Technical Architecture:** The implementation must use Osdag's modern web stack, which consists of a **Django** backend and a **React** frontend.

2.1.2 Significance

This project is significant as it directly contributes to FOSSEE's mission of promoting FLOSS in engineering education. By developing this module, we help bridge the feature gap between the established desktop version of Osdag and its more accessible web counterpart, providing a valuable, free design tool for students and engineers across India.

2.2 Methodology

2.2.1 Tools & Technologies

A specific set of open-source tools was used to accomplish the task, as detailed in the table below.

Table 2.1: Tools and Technologies Used

Category	**Open-Source Tools Used**	**Purpose**
Frontend	React, Material-UI	UI development and creating a responsive user experience.
Backend	Django, Django REST Framework	API development and handling design calculations.
3D Visualization	Three.js, @react-three/fiber, @react-three/drei	Interactive 3D rendering of the steel connection.
Documentation	Markdown, LaTeX	Creating technical specifications and user guides.

2.2.2 Development Workflow

The development process was broken down into four logical steps:

1. **UI Development:** The first step was to replicate the input and output docks from the desktop version's user interface using React and Material-UI components.

2. **API Integration:** Next, RESTful API endpoints were built using the Django REST Framework to handle communication between the frontend and backend for design calculations.

3. **3D Visualization:** An interactive 3D model was integrated into the UI using Three.js and the '@react-three/fiber' library to provide visual feedback to the user.
4. **Validation:** The module's logic was tested against more than 15 compliance checks specified in IS 800:2007 to ensure design accuracy (though implemented with mock data for the screening task).

2.3 Implementation Details

The implementation is divided into three parts: the React frontend, the Three.js 3D visualizer, and the Django backend.

2.3.1 Frontend Development (React)

The user interface is handled by a React component named `BeamSpliceForm.js`. This component manages the application's state using the 'useState' hook to store user inputs for parameters like section size, bending moment, and shear force. On submission, it uses the 'axios' library to send a POST request containing the form data to the Django backend API. The response, containing the design results, is then stored in the state and displayed in the output dock.

Listing 2.1: BeamSpliceForm.js - React Component

```
import React, { useState } from "react";
import { TextField, MenuItem, Button, Grid, Paper, Typography }
   from "@mui/material";
import axios from "axios";
import BeamVisualizer from "./BeamVisualizer";

const BeamSpliceForm = () => {
  const [formData, setFormData] = useState({
    sectionSize: "ISMB 300",
    bendingMoment: "",
    shearForce: "",
    axialForce: "",
    flangeThickness: 12,
    webThickness: 10
  });

  const [results, setResults] = useState(null);

  const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      const response = await axios.post('http://localhost:8000/
         api/beam-splice/', {
        section_size: formData.sectionSize,
        bending_moment: parseFloat(formData.bendingMoment),
        shear_force: parseFloat(formData.shearForce),
        axial_force: parseFloat(formData.axialForce),
        // ... other data
      });
      setResults(response.data);
    } catch (error) {
      console.error('Error:', error.response?.data);
    }
  };

  return (
    // JSX for the form, inputs, and results display
    <Paper elevation={3} sx={{ p: 4 }}>
        {/* ... */}
        <Button onClick={handleSubmit}>Calculate Design</Button>
        {/* ... */}
    </Paper>
  );
};
export default BeamSpliceForm;
```

2.3.2 3D Visualization (React Three Fiber)

For the 3D visualization, a separate component named `BeamVisualizer.js` was created. It uses '@react-three/fiber', a React renderer for Three.js, to create a declarative 3D scene. The component sets up a `Canvas`, defines a `PerspectiveCamera` and `OrbitControls` for user interaction, and adds lighting. The beam itself is constructed from simple 'Box' geometries representing the flanges and web. The dimensions of these boxes are passed down as props from the parent `BeamSpliceForm` component, allowing the model to update in real-time based on calculation results.

Listing 2.2: BeamVisualizer.js - 3D Visualization Component

```
import React from 'react';
import { Canvas } from '@react-three/fiber';
import { OrbitControls, PerspectiveCamera, Box, Text } from '
   @react-three/drei';

const BeamVisualizer = ({ flangeWidth = 200, webHeight = 300 })
   => {
  return (
    <Canvas
      style={{ height: '500px', background: '#f8f9fa' }}
      camera={{ position: [500, 300, 500], fov: 45 }}
    >
      <PerspectiveCamera makeDefault position={[500, 400, 500]}
         />
      <OrbitControls />
      <ambientLight intensity={0.5} />
      <pointLight position={[10, 10, 10]} />

      {/* Main Beam (ISMB Section) */}
      <group>
        {/* Flanges */}
        <Box args={[flangeWidth, 20, 20]} position={[0, webHeight
            / 2, 0]}>
          <meshStandardMaterial color="#3498db" />
        </Box>

        {/* Web */}
        <Box args={[20, webHeight, 20]} position={[0, 0, 0]}>
          <meshStandardMaterial color="#e74c3c" />
        </Box>
      </group>

      {/* Labels */}
      <Text position={[0, webHeight / 2 + 50, 0]} fontSize={20}
         color="black">
        Web Height: {webHeight}mm
      </Text>
    </Canvas>
  );
};
export default BeamVisualizer;
```

2.3.3 Backend Development (Django REST Framework)

The backend follows Django's Model-View-Template (MVT) architecture.

- **Model (`models.py`):** This file defines the database schema using a Django 'Model'. The `BeamSpliceBolted` class contains fields for all input parameters (e.g., `section_size`, `bending_moment`) and output results (e.g., `member_capacity`, `design_status`).
- **Serializer (`serializers.py`):** A 'ModelSerializer' is used to convert the complex data from the Django model into a JSON format that can be easily transmitted over the API. It also handles validation for incoming data.
- **View (`views.py`):** The 'CreateAPIView' handles the incoming POST request from the frontend. It uses the serializer to validate the data, then calls a 'perform create' method. Inside this method, mock design calculations are performed, and the results are saved to the database along with the input data. The serializer then returns the complete object, including the results, in the HTTP response.

Listing 2.3: views.py - Django View with Mock Logic

```
from rest_framework import generics
from .models import BeamSpliceBolted
from .serializers import BeamSpliceSerializer

class BeamSpliceCreateView(generics.CreateAPIView):
    queryset = BeamSpliceBolted.objects.all()
    serializer_class = BeamSpliceSerializer

    def perform_create(self, serializer):
        data = serializer.validated_data

        # Mock design calculations for the screening task
        member_capacity = 1725
        flange_plate_thickness = max(data.get('flange_thickness',
            0), 10)
        web_plate_thickness = max(data.get('web_thickness', 0),
           8)
        design_status = "SAFE"

        # Save the input data along with the calculated results
        serializer.save(
            member_capacity=member_capacity,
            flange_plate_thickness=flange_plate_thickness,
            web_plate_thickness=web_plate_thickness,
            design_status=design_status
        )
```

Chapter 3

Environment Setup and Initial Debugging

The journey into the Osdag-web project began not with feature development, but with the fundamental and most critical prerequisite: the setup and configuration of a complete local development environment. This initial phase proved to be a significant and challenging task in its own right. The Osdag platform is a complex, multi-layered system composed of a Python-based Django backend, a PostgreSQL database, a Node.js-driven React frontend, and a highly specialized stack of scientific and Computer-Aided Design (CAD) libraries.

This chapter provides a comprehensive, narrative account of the methodical process undertaken to build a functional local instance of the application. It goes beyond a simple list of commands to discuss the rationale behind each step and, more importantly, provides a deep-dive into the series of technical hurdles encountered and the debugging strategies employed to resolve them. This detailed record is intended to serve as a practical guide for future developers, aiming to streamline their onboarding process and demystify the complexities of the initial project setup.

3.1 The Challenge: Replicating a Complex Production Stack

The primary objective was to create a local development environment that precisely mirrored the production server, ensuring that any code developed locally would behave predictably when deployed. The core challenge lay in successfully integrating several disparate technologies, each with its own dependencies and configuration requirements:

- **Isolated Python Environment:** The project requires a specific version of Python (3.8), and its numerous dependencies must be isolated from other projects on the system. This was managed using the **Conda** package and environment management system.
- **Relational Database:** The application relies on a **PostgreSQL** database. This required not just installing the database server but also correctly initializing it with a dedicated user, a new database instance, and the correct permissions.
- **Backend Framework and Dependencies:** The Django backend has a long list of Python packages specified in a `requirements.txt` file. These include not only web-related libraries but also scientific packages and CAD library wrappers that often have complex native dependencies.
- **Frontend Toolchain:** The client-side application is built in React and requires **Node.js** and its package manager, **npm**, to install dependencies and run the local development server.

3.2 The Setup Process: A Methodical Approach

The setup was approached systematically, layer by layer, to isolate and resolve issues effectively.

3.2.1 Phase 1: Environment and Database Preparation

The foundation was laid by preparing the core environment and the database. An isolated Conda environment named `osdag` was created to house the project's specific Python 3.8

interpreter and its packages. The official Osdag-web repository was forked on GitHub and then cloned to a local directory.

Following this, the PostgreSQL server was installed. A secure password was configured for the administrative `postgres` user. The crucial next step was to create the application-specific database and role. Using the `psql` command-line utility, the following SQL commands were executed to create a new role named `osdagdeveloper` and a database named `postgres_Intg_osdag`, assigning ownership to the new role. Finally, the Django project's settings files (`settings.py`, `populate_database.py`, `update_sequences.py`) were manually edited to replace the placeholder credentials with the ones just created.

3.2.2 Phase 2: Backend Dependency Installation and Debugging

This phase was the most challenging and required significant troubleshooting. The seemingly straightforward command, `pip install -r requirements.txt`, triggered a cascade of errors that needed to be diagnosed and resolved one by one.

Hurdle 1: The C++ Compilation Error

The first failure was an error message stating: `error: Microsoft Visual C++ 14.0 or greater is required.`

- **Problem Analysis:** This error indicated that a dependency (in this case, `pycosat`) was not available as a pre-compiled binary "wheel" for my system's architecture. Therefore, `pip` was attempting to compile it from its C++ source code, but the necessary compiler toolchain was not installed on the system.

- **Solution:** The resolution was to install the official **Microsoft C++ Build Tools** from the Visual Studio website. During the installation, care was taken to select the "C++ build tools" workload and the correct Windows SDK. After a system reboot to ensure the environment variables were correctly registered, the `pip install` command was re-run, and it successfully compiled and installed the package.

Hurdle 2: The Missing CAD Library

After installing the base requirements, the next step was to run the database migrations. This failed immediately with a new error: `ModuleNotFoundError: No module named 'OCC'`.

- **Problem Analysis:** OCC refers to OpenCASCADE Technology, a professional-grade 3D CAD kernel. The Python wrapper for it, `pythonocc-core`, is a complex package with many native, non-Python dependencies. It is notoriously difficult to install with `pip` alone, as `pip` does not manage non-Python dependencies.
- **Solution:** This is a classic use case where the Conda package manager excels. Conda is designed to manage complex binary dependencies alongside Python packages. The fix was to use Conda to install the library from the community-maintained `conda-forge` channel via the command: `conda install -c conda-forge pythonocc-core`. This successfully installed the library and all of its required native components.

Hurdle 3: The Python Version Incompatibility

With the CAD library in place, the migration was attempted again, only to be met with a subtle Python error: `ImportError: cannot import name '_Protocol' from 'typing'`.

- **Problem Analysis:** This error pointed to a version mismatch between the Python language itself and a library's expectations. The `_Protocol` class, a feature for static type checking, is not part of the standard `typing` module in Python 3.8. A dependency within the project was written against a newer Python version's standard library.
- **Solution:** This issue required a direct code modification. The solution involved two steps: first, ensuring the `typing_extensions` package, which provides back-ported typing features, was up-to-date (`pip install --upgrade typing-extensions`). Second, the problematic import statement in the file `osdag_api/module_finder.py` had to be manually changed, as detailed in the next section.

3.2.3 Phase 3: Finalizing Setup and Launching the Application

With all dependencies installed and code fixes applied, the environment was ready. The database was seeded using the project-specific scripts (`populate_database.py`, `update_sequences.py`), and the Django migrations (`python manage.py migrate`) were finally run to completion. The backend server was launched on port 8000.

Concurrently, the frontend was set up by navigating to the `osdagclient` directory, installing all Node.js dependencies with `npm install`, and launching the React development server on port 5173 with `npm run dev`. Accessing `http://localhost:5173/` in a web browser successfully loaded the Osdag-web application.

3.3 Code-Level Intervention: Resolving the _Protocol ImportError

The ImportError was a critical bug that completely halted the application's startup. Fixing it required modifying the project's source code to use a more backward-compatible approach.

3.3.1 Description of the Script Change

The file `osdag_api/module_finder.py` contained an import statement that was only valid in Python 3.9+. The fix was to avoid importing from the standard `typing` library and instead import the equivalent class from the `typing_extensions` library, aliasing it to match the name expected by the rest of the code.

3.3.2 Code Modification

The following surgical change was applied to the Python script to resolve the import error.

Listing 3.1: Fixing the _Protocol ImportError in module_finder.py

```
#-------------------- Original Failing Code -------------

# File: osdag_api/module_finder.py
# This line fails on Python 3.8 because _Protocol is not in the
    standard library.

from typing import Dict, Any, List, _Protocol

#-------------------- Corrected Code -------------------

# File: osdag_api/module_finder.py
# The import statement is separated for clarity and correctness.

from typing import Dict, Any, List

# The equivalent 'Protocol' class is imported from the '
    typing_extensions'
# library and aliased as '_Protocol' to ensure the rest of the file
    functions
# without needing further changes. This is the standard way to handle
    such
# backward-compatibility issues.

from typing_extensions import Protocol as _Protocol
#-------------------- end code ------------------------
```

3.4 Distilling Experience into Documentation

This challenging setup process provided a wealth of practical knowledge that was not fully captured in the existing documentation. To ensure future developers could avoid these same pitfalls, a new "Troubleshooting" guide was drafted for the project's internal developer manual, converting these hard-earned lessons into an actionable resource.

3.4.1 Developer Manual: Troubleshooting Common Installation Errors

This guide provides solutions to common errors encountered during the initial setup of the Osdag-web development environment.

Table 3.1: Common Installation Errors and Solutions

Error Message	**Solution and Rationale**
`error: Microsoft Visual C++ 14.0 or greater is required.`	**Reason:** A package needs to be compiled from C++ source. **Solution:** Install "Visual Studio Build Tools". Ensure the "Desktop development with C++" workload is selected. Reboot the system after installation to update system PATH.
`ModuleNotFoundError: No module named 'OCC'`	**Reason:** The OpenCASCADE CAD kernel wrapper has complex binary dependencies that `pip` cannot manage. **Solution:** Use the Conda package manager, which excels at this. Run: `conda install -c conda-forge pythonocc-core`
`ModuleNotFoundError: No module named 'pylatex'`	**Reason:** A dependency for PDF report generation is missing. **Solution:** This is a standard Python package and can be installed with pip: `pip install pylatex`
`ImportError: cannot import name '_Protocol' from 'typing'`	**Reason:** The code is using a typing feature from a newer Python version. **Solution:** Ensure backward compatibility. First, run `pip install --upgrade typing-extensions`. Then, modify the import statement in the specific file causing the error as detailed in Section 3.1.

Chapter 4

Internship Task 1: Debugging the End Plate CAD Module

4.1 Problem Statement

The first task undertaken was to diagnose and resolve a critical bug within the existing "End Plate Connection" module of the Osdag-on-Cloud platform. The module exhibited a perplexing and non-intuitive failure mode: while it successfully processed all user inputs and returned a complete and accurate set of numerical design calculations in the output dock, it consistently failed to generate the corresponding 2D and 3D CAD models.

The primary issue was that this failure occurred silently. The backend process would complete without raising any server-side errors, and the frontend would not display any warning or error message to the user. This resulted in an incomplete and confusing user experience, as a key feature of the software was simply missing without explanation, undermining the tool's reliability and usability. The objective was to trace the root cause of this silent failure and implement a permanent fix.

4.2 Tasks Done

The debugging process was methodical, based on the hypothesis that the input data, while valid for the numerical calculation engine, was being lost or malformed before reaching the separate CAD generation service. The methodology involved tracing the data's entire lifecycle from the moment it was submitted by the user to the final CAD rendering call.

The core of the work involved modifying three key backend files: `end_plate_input.py`, `endplate_outputView.py` (the Django API view), and `end_plate_connection.py` (the core logic module). The following steps were taken:

1. **Strategic Logging:** To make the silent failure visible, extensive logging was injected into the code. Specifically, `print()` statements were used to output the entire input data dictionary at critical hand-off points between functions and modules. Using the `json` library to pretty-print the dictionary made it easy to visually inspect the data for any inconsistencies.

2. **Targeted Exception Handling:** The existing code used very broad `try...except` blocks that suppressed the specific nature of the errors. These were replaced with narrow, granular `try...except` blocks wrapped tightly around individual function calls within the CAD generation pipeline. This was designed to isolate the exact line of code that was failing.

3. **Error Triangulation:** By comparing the logged data and the precise location of the caught exception, the problem was triangulated. The logs showed the data was correct when leaving the Django view, but the targeted exception handling caught a `KeyError` deep within the CAD setup functions.

4. **Root Cause Analysis:** The `KeyError` revealed that a downstream function expected a dictionary key with a different name than what was being supplied (`"Bolt.Connectivity"` vs. `"Connectivity"`). This subtle data contract mismatch was the root cause of the failure.

5. **Implementation of Fix:** The solution was to harmonize the dictionary keys, ensuring that the data structure remained consistent across all modules that consume it.

4.3 Python Code

This section presents the Python code modifications made to the `end_plate_connection.py` file. The changes enhance the module to add robust diagnostic capabilities. The code demonstrates how targeted exception handling can pinpoint silent failures in a complex application.

4.3.1 Description of the Script

The code snippet below is from the `create_cad_model` function. The original function had a single, large `try...except` block that made it impossible to know which part of the CAD generation process was failing. The modifications refactor this into multiple, specific blocks to isolate failures in three distinct stages:

- **Initialization**: Creating the main CAD logic object.
- **Data Setup**: Loading the user's input data into the CAD object.
- **Geometry Rendering**: The final call to generate the 2D shapes.

4.3.2 Python Code Snippet

Listing 4.1: Targeted Exception Handling for CAD Generation

```python
def create_cad_model(input_values: Dict[str, Any], section: str,
   session: str) -> str:
    """Generate the CAD model from input values as a BREP file.
       Return file path."""
    if section not in ("Model", "Beam", "Column", "Plate"):
        raise InvalidInputTypeError("section", "'Model', 'Beam',
           'Column' or 'Plate'")

    module = create_from_input(input_values)

    # --- MODIFICATION 1: Isolate failures in CAD object
       initialization ---
    try:
        cld = CommonDesignLogic(None, '', module.module, module.
           mainmodule)
        print("[DEBUG] CAD logic object created successfully.")
    except Exception as e:
        print(f'[FATAL_ERROR] Failed during CommonDesignLogic instantiation: {e}')
        return False

    # --- MODIFICATION 2: Isolate failures when setting up CAD data ---
    try:
        scc.setup_for_cad(cld, module)
        print("[DEBUG] CAD data setup successful.")
    except Exception as e:
        print(f'[FATAL_ERROR] Failed during scc.setup_for_cad: {e}')
        return False

    cld.component = section

    # --- MODIFICATION 3: Isolate failures in the final geometry rendering ---
    try:
        model = cld.create2Dcad()
        print("[DEBUG] 2D CAD model generated successfully.")
    except Exception as e:
        print(f'[FATAL_ERROR] Failed during cld.create2Dcad() execution: {e}')
        return False

    # (Code to save the file follows)
    return file_path
```

4.3.3 Explanation of the Code

- **Initialization**: Wraps the instantiation of the `CommonDesignLogic` object. If this fails, it indicates a problem with the core CAD engine.
- **Data Setup**: Wraps the `setup_for_cad` function where user data is loaded. Failure

here (like a `KeyError`) means data contract mismatch.

- **Geometry Rendering**: Wraps the final rendering call. Failure here means the data was valid but causes geometric or computational issues.

4.4 Documentation

A new section for the Osdag Developer Manual was drafted to guide future developers on how to efficiently diagnose and solve similar silent failures.

4.4.1 Developer Manual: Debugging Silent Failures

When a feature in an Osdag module fails without an obvious error, it is often due to a "data contract" mismatch between the Django web layer and the core Python logic module. The following procedure is recommended:

Step 1: Make the Failure Visible with Logging

At every boundary between major components, log the data being transferred.

```python
import json
print("[DEBUG] Data being passed to core module:")
print(json.dumps(request.data, indent=2))
try:
    output = core_module.some_function(request.data)
except Exception as e:
    print(f"[ERROR] Core module failed with exception: {e}")
```

Step 2: Use Targeted, Not General, Exception Handling

Avoid wrapping an entire function in a single `try...except` block.

```python
# BAD: General handling hides the real source
try:
    initialize_object()
    load_data_into_object()
    render_geometry()
```

```python
except:
    print("Something went wrong.")

# GOOD: Targeted handling pinpoints the error
try:
    initialize_object()
except Exception as e:
    print(f"Error during initialization: {e}")

try:
    load_data_into_object()
except Exception as e:
    print(f"Error loading data: {e}")

try:
    render_geometry()
except Exception as e:
    print(f"Error during rendering: {e}")
```

Following this two-step process helps quickly determine whether the problem lies in data being sent, received, or processed—turning a silent failure into a solvable bug.

Chapter 5

Internship Task 2: Fixing the Session Management API

5.1 Task 2: Problem Statement

The second major task involved resolving a persistent bug in the platform's session management system. The existing API was intended to create a unique session when a user opened a design module and to track it using a browser cookie. However, the logic for reliably ending a session was flawed. Specifically, the mechanism to delete the session from the database and clear the cookie from the user's browser often failed.

This inconsistency resulted in a poor user experience. After completing work in one module (e.g., "End Plate Connection"), if a user tried to open another (e.g., "Fin Plate Connection"), the stale cookie would persist in their browser. The session creation API would detect this leftover cookie, interpret it as an ongoing session, and refuse to create a new one. This locked the user out with a misleading "session does not exist" error, often requiring manual cookie deletion via developer tools. The goal of this task was to redesign the session API to ensure robust and predictable session handling for all modules.

5.2 Task 2: Tasks Completed

A comprehensive refactoring of both backend logic and frontend behavior was carried out to achieve a "fail-safe" session lifecycle.

1. **Centralized Configuration:** The original `CreateSession` view contained a lengthy

chain of `if/elif` statements to handle each possible module cookie. This was replaced with a single Python dictionary, `cookie_keys`, which maps each `module_id` to its specific cookie key. This centralization makes the code more maintainable and scalable—adding a new module now only requires adding an entry to this dictionary.

2. **Refactored Session Creation:** The cumbersome `if/elif` logic was replaced with a concise loop over the dictionary values. This checks for the presence of any active session cookie with just a few lines of code, regardless of the module.

3. **Robust Session Deletion Endpoint:** The `DeleteSession` API was redesigned to accept a `module_id` from the frontend. Using the same centralized dictionary, it dynamically identifies the appropriate cookie. The endpoint then:

   - Deletes the session record from the database by matching the cookie ID.
   - Sends an explicit `Set-Cookie` header to the browser with an expired date, instructing it to remove the cookie immediately.

   This explicit cookie removal was the missing piece in the original design.

4. **Frontend Integration:** The frontend JavaScript was updated to call the new deletion endpoint whenever a user closes or navigates away from a design module. This ensures that no stale cookies linger to block future sessions.

5.3 Task 2: Python Code

The final refactored `session_api.py` defines two API views: `CreateSession` and `DeleteSession`. These ensure a complete and reliable session lifecycle.

5.3.1 Description of the Script

- **CreateSession (APIView):** Handles requests to start a new session. It first checks for any active session cookies. If none exist, it generates a unique ID, stores it in the database, and sets the appropriate cookie.
- **DeleteSession (APIView):** Handles requests to terminate an active session. It deletes the corresponding database record and explicitly removes the session cookie from the user's browser.

5.3.2 Refactored Python Code

Listing 5.1: Refactored Session Management API (`session_api.py`)

```python
from django.http import JsonResponse
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from django.utils.crypto import get_random_string
from django.utils.decorators import method_decorator
from django.views.decorators.csrf import csrf_exempt
from osdag.models import Design
from osdag.serializers import Design_Serializer
from osdag_api import developed_modules

class CreateSession(APIView):
    def post(self, request):
        module_id = request.data.get("module_id")
        if not module_id:
            return JsonResponse({"error": "Module ID is required"},
                status=400)

        cookie_keys = {
            "Fin Plate Connection": "fin_plate_connection_session",
            "End Plate Connection": "end_plate_connection_session",
            "Cleat Angle Connection": "cleat_angle_connection_session",
            "Seated Angle Connection": "seated_angle_connection_session
                ",
            "Cover Plate Bolted Connection": "
                cover_plate_bolted_connection_session",
            "Beam Beam End Plate Connection": "
                beam_beam_end_plate_connection_session",
            "Cover Plate Welded Connection": "
                cover_plate_welded_connection_session",
            "Beam-to-Column End Plate Connection": "
                beam_to_column_end_plate_connection_session",
        }

        for session_key in cookie_keys.values():
            if request.COOKIES.get(session_key):
                return Response({"status": "set", "message": "An existing session is active."}, status=status.HTTP_200_OK)

        if module_id not in developed_modules:
            return JsonResponse({"error": "This module is not developed yet"}, status=501)

        cookie_id = get_random_string(length=32)
        serializer = Design_Serializer(data={"cookie_id": cookie_id, "module_id": module_id})

        if serializer.is_valid():
            serializer.save()
            response = JsonResponse({"status": "created"}, status=201)
            cookie_key = cookie_keys.get(module_id)
            response.set_cookie(key=cookie_key, value=cookie_id, samesite="None", secure=True)
            return response
        else:
            return JsonResponse(serializer.errors, status=500)

class DeleteSession(APIView):
    @method_decorator(csrf_exempt)
    def dispatch(self, *args, **kwargs):
        return super().dispatch(*args, **kwargs)

    def post(self, request):
        module_id = request.data.get("module_id")
        if not module_id:
            return JsonResponse({"error": "Module ID is required"}, status=400)

        cookie_keys = {
            "Fin Plate Connection": "fin_plate_connection_session",
            "End Plate Connection": "end_plate_connection_session",
            "Cleat Angle Connection": "cleat_angle_connection_session",
            "Seated Angle Connection": "seated_angle_connection_session",
            "Cover Plate Bolted Connection": "cover_plate_bolted_connection_session",
            "Beam Beam End Plate Connection": "beam_beam_end_plate_connection_session",
            "Cover Plate Welded Connection": "cover_plate_welded_connection_session",
            "Beam-to-Column End Plate Connection": "beam_to_column_end_plate_connection_session",
        }

        cookie_key = cookie_keys.get(module_id)
        if not cookie_key:
            return JsonResponse({"error": "Invalid module ID"}, status=400)

        cookie_id = request.COOKIES.get(cookie_key)
        if not cookie_id:
            return JsonResponse({"status": "not_found", "message": "No active session for this module."}, status=200)

        try:
            design_session = Design.objects.get(cookie_id=cookie_id)
            design_session.delete()
        except Design.DoesNotExist:
            pass  # Even if not found, we should still clear the cookie

        response = JsonResponse({"status": "deleted"}, status=200)
        response.delete_cookie(key=cookie_key, samesite="None")
        return response
```

5.4 Task 2: Developer Documentation

Detailed usage instructions were added to the developer manual to ensure the frontend team correctly integrates with the new session lifecycle.

5.4.1 Session API Usage Guide

- **POST** `/api/sessions/create` Creates a new session for a specified module. If an active session exists, it must first be deleted.
- **POST** `/api/sessions/delete` Explicitly terminates a session and clears the cookie. This must be called when a user closes or navigates away from a design module.

Listing 5.2: Example Frontend Usage

```
const deleteSessionOnModuleClose = async (moduleId) => {
    try {
        await fetch('/api/sessions/delete', {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ module_id: moduleId }),
            credentials: 'include'
        });
        console.log(`Session for ${moduleId} terminated.`);
    } catch (err) {
        console.error('Failed to terminate session:', err);
    }
};
```

Chapter 6

Internship Task 3: Development of the Cover Plate Welded Module

6.1 Task 3: Problem Statement

The third major internship task was the ground-up development of a new, feature-complete design module for the Osdag platform: the "Beam-to-Beam Cover Plate Welded Connection." This project was initiated to add a new module in Osdag on cloud. The core requirement was to create a robust and intuitive module that performs all design calculations in strict accordance with the Indian Standard for steel structures, **IS 800:2007**.

The scope of this task was comprehensive, covering the full software development lifecycle. This included designing and implementing the user-facing frontend in React, developing the backend API and engineering logic in Django and Python, and ensuring seamless integration into the existing Osdag-on-Cloud infrastructure. The final deliverable had to be a reliable, accurate, and user-friendly module.

6.2 Task 3: Tasks Done

The development was systematically divided into frontend and backend workstreams, which were executed in parallel to ensure efficient progress.

6.2.1 Frontend Development (React)

The frontend was crafted to provide an interactive and intuitive user experience, using **React** for the component architecture and **Ant Design** for a consistent UI.

1. **UI/UX and Component Architecture:** The main user interface was built as a single React component, `CoverPlateWelded.js`. This component renders the classic three-panel Osdag layout: an **Input Dock** on the left for user-configurable parameters, a central **3D CAD Viewer**, and a **Output Dock** on the right to display results. All interactive elements, such as dropdowns (`<Select>`), text fields (`<Input>`), and modals (`<Modal>`), were implemented using the Ant Design library.

2. **State Management and API Integration:** All user-driven input values are managed within the component's local state using the `useState` hook. For communication with the backend, a centralized `ModuleContext` was used. This context provides shared functions (`createSession`, `createDesign`, etc.) that abstract the underlying `fetch` API calls, keeping the main component clean and focused on the UI logic.

3. **Interactive 3D Visualization:** To provide users with immediate visual feedback, an interactive 3D viewer was implemented using **@react-three/fiber** and **Three.js**. When a design is successfully completed, the backend sends the path to a CAD model file. The frontend then fetches this file and the `<Model>` component dynamically renders it within the scene. The viewer includes custom camera controls, allowing the user to switch between pre-set views (”Model”, ”Beam”, ”Connector”) for detailed inspection.

6.2.2 Backend Development (Django/Python)

The backend provides the data, performs the engineering calculations, and generates the CAD model.

1. **Input API View (`cover_plate_weld_input.py`):** This initial API endpoint is responsible for populating the frontend's dropdown menus. On page load, the React component calls this endpoint. The view then queries the Osdag database

using Django's ORM to fetch comprehensive lists of available beam sections (`Beams` model), material grades (`Material` model), and other necessary options, returning them as a JSON object.

2. **Core Engineering Logic (`cover_plate_welded_connection.py`):** This file is the intellectual core of the module. A dedicated Python class, `BeamCoverPlateWeld` , was authored to encapsulate every aspect of the IS 800:2007 design procedure for this specific connection. This class accepts the validated user input and contains numerous methods to perform checks, including:

    - Calculation of design forces (moment and shear) on the flange and web splices.
    - Determination of the required weld size and length based on force demand, including checks for minimum and maximum permissible weld sizes per IS 800.
    - Sizing of the flange and web cover plates to ensure they have sufficient capacity to resist yielding and rupture failure modes.
    - Verification of the main member's strength at the net section.
    - Generation of detailed log messages to inform the user about the design decisions made by the engine.

3. **Output API View (`cover_plate_weld_output.py`):** This is the main design endpoint. It receives the complete input JSON from the frontend, orchestrates the entire design process by calling the core logic module, saves the transaction (inputs, outputs, logs) to the database via the `Design_Serializer`, and returns the final, comprehensive results to the frontend.

6.3 Task 3: Python and React Code

This section presents annotated code snippets from each major component of the module's architecture, providing a clear view of the implementation details.

6.3.1 Frontend: React Component State and Submission Logic

The following snippet from `CoverPlateWelded.js` shows how user inputs are managed in the component's state and how the data is compiled and sent to the backend when the

"Design" button is clicked.

Description of the Script

This React code uses the `useState` hook to hold an object of all input parameters. The `handleSubmit` function gathers this state, maps it to the API's expected key names, and calls the `createDesign` function (from context) to trigger the backend calculation.

React Code

Listing 6.1: State Management and API Submission in React

```
//--------------------begin code-------------
function CoverPlateWelded() {
const [output, setOutput] = useState(null);
const [loading, setLoading] = useState(false);

// Accessing shared functions from ModuleContext
const { createDesign } = useContext(ModuleContext);

// State hook to store all user inputs from the form
const [inputs, setInputs] = useState({
flange_plate_preferences: "Outside",
flange_plate_thickness: [],
connector_material: "E 165 (Fe 290)",
web_plate_thickness: [],
load_axial: "100",
load_moment: "100",
load_shear: "100",
member_designation: "MB 600",
module: "Beam-to-Beam Cover Plate Welded Connection",
weld_fab: "Shop Weld",
weld_type: "Fillet Weld"
});

// This function is called when the user clicks the "Design" button
const handleSubmit = async () => {
// Basic validation to ensure required fields are filled
if (!inputs.member_designation || !inputs.load_shear) {
alert("Please input all the required fields");
return;
}

// Construct the JSON payload with keys that match the backend API
const api_params = {
  "Connector.Flange_Plate.Preferences": inputs.flange_plate_preferences,
  "Connector.Flange_Plate.Thickness_list": inputs.flange_plate_thickness,
  "Connector.Web_Plate.Thickness_List": inputs.web_plate_thickness,
  "Load.Axial": inputs.load_axial,
  "Load.Moment": inputs.load_moment,
  "Load.Shear": inputs.load_shear,
  "Member.Designation": inputs.member_designation,
  "Module": inputs.module,
  "Weld.Fab": inputs.weld_fab,
  "Weld.Type": inputs.weld_type,
  "Material": inputs.material,
  "Member.Material": inputs.member_material,
  "Design.Design_Method": "Limit State Design",
  "Detailing.Gap": "3",
48   "Connector.Material": inputs.connector_material,
49   "Weld.Material_Grade_OverWrite": "410",
50 };
51
52 setLoading(true); // Set loading state for UI feedback
53 // Call the abstracted API function from context
54 createDesign(api_params, "Cover-Plate-Welded-Connection");
55 };
56
57 return (
58 // JSX for rendering the Input Dock with an onClick handler for the button
59 <div className="InputDock">
60 {/* ... many <Select> and <Input> components ... */}
61 <Input type="button" value="Design" onClick={handleSubmit} />
62 </div>
63 );
64 }
65 //-------------------- end code ---------------
```

Explanation of the Code

- **Line 8-21**: The `useState` hook initializes the component's state with default values for every input field on the form.
- **Line 24**: The `handleSubmit` function is defined. It will be attached to the "Design" button's `onClick` event.
- **Line 31-48**: Inside `handleSubmit`, a new object `api_params` is created. This is crucial as it acts as a translation layer, mapping the component's state names to the specific, dot-separated key names required by the Python backend API.
- **Line 52**: The `createDesign` function is called. This function, provided by a shared context, contains the actual `fetch` call to the backend, sending the `api_params` object as a JSON payload.

6.3.2 Backend: Core Engineering Logic Snippet

The snippet below from `cover_plate_welded_connection.py` shows a representative method from the `BeamCoverPlateWeld` class, demonstrating how a specific design calculation (determining the required weld size) is performed.

Description of the Script

This Python method calculates the required size of the weld connecting the flange cover plate to the beam flange. It considers the design moment, beam geometry, and material properties to find the force per unit length on the weld, and from that, determines the required throat thickness and final weld size, checking it against the minimums specified in IS 800:2007.

Python Code

Listing 6.2: Weld Design Calculation in Core Logic Class

```
#--------------------begin code-------------

In class BeamCoverPlateWeld:

def flange_weld_calculation(self):
"""
Calculates the required weld size for the flange plate connection.
Based on IS 800:2007 design principles.
"""
# Retrieve pre-calculated values from object state
flange_plate_length = self.flange_plate_length_provided
beam_depth = self.D
beam_flange_thickness = self.T

# Design force on the flange splice (moment divided by lever arm)
flange_splice_force = (self.load_moment * 10**6) / (beam_depth -
    beam_flange_thickness) # in N

# Force per unit length on the weld (total force / total weld length)
# Total weld length is 2 times the plate length (for the two
    longitudinal welds)
force_per_mm = flange_splice_force / (2 * flange_plate_length) # N/mm

# Required throat thickness (tt) of the weld
# Strength of weld = (0.7 * tt * f_u) / (sqrt(3) * gamma_mw)
# We equate this to force_per_mm and solve for tt
gamma_mw = self.gamma_mw_weld # Partial safety factor for welds (e.g.,
    1.25 for shop weld)
f_u = self.fu # Ultimate stress of the plate material

```

```
28 required_throat_thickness = (force_per_mm * math.sqrt(3) * gamma_mw) / (0.7 * f_u)
29
30 # Required weld size (s) = throat thickness / 0.7
31 required_weld_size = required_throat_thickness / 0.7
32
33 # Check against minimum weld size as per IS 800:2007, Table 21
34 # This depends on the thickness of the thicker part being joined.
35 min_weld_size = self.get_min_weld_size(beam_flange_thickness, self.flange_plate_thickness_provided)
36
37 # The final provided weld size must be the larger of the required and the minimum
38 final_weld_size = max(required_weld_size, min_weld_size)
39
40 # Round up to the nearest integer
41 self.flange_weld_size_provided = math.ceil(final_weld_size)
42
43 # Log the decision for the user
44 self.logger.info(f"Required weld size based on force: {required_weld_size:.2f} mm")
45 self.logger.info(f"Minimum weld size as per IS 800 Table 21: {min_weld_size} mm")
46 self.logger.info(f"Provided flange weld size: {self.flange_weld_size_provided} mm (rounded up)")
47
48 #-------------------- end code ---------------
```

Explanation of the Code

- **Line 13**: Calculates the axial force experienced by the flange plates due to the bending moment.
- **Line 17**: Distributes this total force over the total length of the two welds that connect the plate to determine the force that each millimeter of weld must resist.
- **Line 24**: This is the core IS 800 formula, rearranged to solve for the required throat thickness of the weld based on the force demand and material properties.

- **Line 27**: Converts the required throat thickness to the required nominal weld size (which is what is specified on drawings).
- **Line 31**: Calls another method to get the code-mandated minimum weld size, which is based on the thickness of the steel parts.
- **Line 34**: The actual weld size to be used is determined by taking the maximum of what is required by force and what is required by the code minimums. This ensures all conditions are met.
- **Line 40-42**: Informative messages are added to a logger. These messages are sent back to the frontend to be displayed in the "Logs" panel, providing transparency to the user.

6.3.3 Full code

The complete frontend and backend source code for this module, including all calculation methods and API views, is available in the project's source code repository.

Chapter 7

Internship Task 4: Development of the Beam-to-Column End Plate Module

7.1 Task 4: Problem Statement

The fourth major task was the conception and end-to-end development of a highly complex and critical new module: the ”Beam-to-Column End Plate Connection.” This type of connection is a moment-resisting joint, fundamental to the design of rigid steel frames. The project's objective was to create a comprehensive design tool that allows engineers to design a bolted end plate connection for various structural scenarios, including beam-to-column flange and beam-to-column web configurations. The module had to perform a multitude of complex checks for bolts, welds, the end plate itself, and the connected members, all in strict compliance with the design procedures outlined in the Indian Standard **IS 800:2007**. The final deliverable was to be a robust, accurate, and fully integrated module within the Osdag-on-Cloud platform.

7.2 Task 4: Tasks Done

The development process was a substantial undertaking, segmented into distinct frontend and backend workstreams that were developed in tandem.

7.2.1 Frontend Development (React)

The frontend was engineered to manage the module's high degree of complexity while maintaining an intuitive user experience, using **React** and the **Ant Design** component library.

1. **Complex and Conditional UI:** The main component, `BeamToColumnEndPlate.js`, was designed to be highly dynamic. It features critical conditional inputs, such as "Connectivity," which alters the available member sections, and "End Plate Type," which changes the underlying design assumptions. This required sophisticated state management to ensure the UI always reflects a valid configuration.

2. **Comprehensive State Management:** Given the large number of inputs (beam and column sections, materials, factored loads, detailed bolt properties, plate thicknesses, and weld types), the `useState` hook was used to manage a single, comprehensive state object. This centralized approach simplifies data handling and submission. For customized list selections (e.g., bolt diameters), the Ant Design `<Transfer>` component was integrated into a modal window, providing a user-friendly way to select multiple values.

3. **Robust API Integration:** Communication with the backend was handled via the shared `ModuleContext`. This allowed for clean and reusable API calls. The component fetches initial data on load, submits the final design payload, and handles the rendering of results and logs returned from the server.

4. **Advanced 3D Visualization:** The **@react-three/fiber** viewer was implemented to render a detailed 3D model of the complex connection. This visualization is crucial for helping the user understand the geometry of the bolt layout, stiffeners, and the interaction between the beam, column, and end plate. The viewer includes camera controls to focus on specific components like the "EndPlate" or the "Column". Use code with caution.

7.2.2 Backend Development (Django/Python)

The backend was architected to be robust, extensible, and capable of handling the intricate engineering calculations required for a moment connection.

1. **Dynamic Input API View (`beam_column_end_plate_input.py`):** This API view was designed to be more dynamic than in simpler modules. It handles multiple query types based on user selection. For example, when the user selects a "Connectivity" type, the frontend sends this information, and the view returns the appropriate lists of beam and column sections from the database using Django's ORM.

2. **Advanced Core Engineering Logic (`beam_column_end_plate.py`):** This is the most complex part of the module. A new Python class was developed to encapsulate the entire IS 800:2007 design process for moment end plate connections. Its responsibilities are extensive:

    - **Bolt Group Analysis:** Calculating forces on each bolt due to combined shear, moment, and axial load. This includes modeling bolt tension and accounting for prying action, a critical secondary force in end plate connections.
    - **Strength Checks:** Performing numerous checks, including bolt shear capacity, bolt tension capacity, combined shear and tension interaction, and bolt bearing capacity.
    - **End Plate Design:** Determining the required end plate thickness to prevent failure in bending at the bolt lines and checking shear capacity.
    - **Weld Design:** Designing the flange and web welds connecting the beam to the end plate to safely transfer the moment and shear forces.
    - **Column Member Checks:** Verifying the connected column for local failures, such as panel zone shear and flange bending, and determining if stiffeners are required.

3. **Resilient Output API View (`beam_column_end_plate_output.py`):** Drawing lessons from previous modules, this API view was built with production-grade robustness. It features extensive logging using Python's `logging` module, structured error handling with specific `try...except` blocks for input validation, core logic execution, and database saving, ensuring that any failure is caught, logged, and reported back to the user with a clear error message. Use code with caution.

7.3 Task 4: Python and React Code

The following sections provide annotated code samples from the key architectural components, illustrating the implementation's complexity and robustness.

7.3.1 Frontend: Consolidating and Submitting Complex Input State

The snippet from `BeamToColumnEndPlate.js` below demonstrates the `handleSubmit` function. It showcases how the various pieces of state managed by the component are gathered, validated, and structured into the precise JSON format required by the backend API.

Description of the Script

This React function is the primary event handler for initiating a design. It collects data from multiple state variables, including the main `inputs` object, the `allSelected` state (which determines whether to use customized or default lists), the user's selected `connectivity`, and the chosen `endPlateType`. It then maps all of this into a single, comprehensive `param` object for API submission.

React Code

Listing 7.1: Handling Complex State Submission in React

```
//-------------------begin code-------------
function BeamToColumnEndPlate() {
// ... numerous useState hooks for inputs, selectedOption, endPlateType, etc.
const { createDesign, boltDiameterList, propertyClassList, thicknessList } = useContext(ModuleContext);
const handleSubmit = async () => {
// Perform validation on critical user inputs
if (!inputs.beam_section || !inputs.column_section || !inputs.load_shear || !inputs.load_moment) {
alert("Please input all the required fields");
return;
}
// Define module names for consistency
let moduleId = "Beam-to-Column End Plate Connection";
let apiModuleId = "Beam-to-Column-End-Plate-Connection";

// Assemble the parameter object by combining multiple state sources
let param = {
  "Bolt.Bolt_Hole_Type": inputs.bolt_hole_type,
  // Conditionally choose the full list or the user's customized list
  "Bolt.Diameter": allSelected.bolt_diameter ? boltDiameterList : inputs.bolt_diameter,
  "Bolt.Grade": allSelected.bolt_grade ? propertyClassList : inputs.bolt_grade,
  "Bolt.Slip_Factor": inputs.bolt_slip_factor,
22    "Bolt.TensionType": inputs.bolt_tension_type,
23    "Bolt.Type": inputs.bolt_type.replaceAll("_", " "),
24    "Connectivity": selectedOption, // From its own state variable
25    "EndPlateType": endPlateType,   // From its own state variable
26    "Connector.Material": inputs.connector_material,
27    "Design.Design_Method": inputs.design_method,
28    "Detailing.Corrosive_Influences": inputs.detailing_corr_status,
29    "Detailing.Edge_type": inputs.detailing_edge_type,
30    "Detailing.Gap": inputs.detailing_gap,
31    "Load.Axial": inputs.load_axial || "0",
32    "Load.Shear": inputs.load_shear,
33    "Load.Moment": inputs.load_moment,
34    "Material": inputs.connector_material,
35    "Member.Supported_Section.Designation": inputs.beam_section,
36    "Member.Supported_Section.Material": inputs.supported_material,
37    "Member.Supporting_Section.Designation": inputs.column_section,
38    "Member.Supporting_Section.Material": inputs.supporting_material,
39    "Module": moduleId,
40    "Weld.Fab": inputs.weld_fab,
41    "Weld.Material_Grade_OverWrite": inputs.weld_material_grade,
42    "Weld.Type": inputs.weld_type,
43    "Connector.Plate.Thickness_List": allSelected.plate_thickness ? thicknessList : inputs.plate_thickness,
44  };
45
46  try {
47    setLoading(true);
48    // Call the design function from context with the fully assembled parameters
49    createDesign(param, apiModuleId);
50    setOutputDisabled(false);
51    setModelKey((prev) => prev + 1);
52  } catch (error) {
53    console.error("Error submitting design:", error);
54    setLoading(false);
55  }
56  };
57  return (
58  <Input type="button" value="Design" onClick={handleSubmit} />
59  );
60  }
61  //-------------------- end code ---------------
```

Explanation of the Code

- **Line 14-20**: The `param` object is constructed. This is the "data contract" that translates the React component's state into the language the backend API understands.
- **Line 16**: This line is a prime example of conditional logic in the data assembly. It checks the `allSelected.bolt_diameter` boolean. If true, it sends the complete list of all possible diameters; otherwise, it sends only the specific list the user customized in the `<Transfer>` modal.
- **Line 21-22**: The state for `Connectivity` and `EndPlateType`, which are managed by their own separate `useState` hooks, are integrated into the final parameter

object.

- **Line 46**: The fully constructed `param` object is passed to the `createDesign` function, initiating the backend process.

7.3.2 Backend: Robust Output API View with Logging

The following snippet from `beam_column_end_plate_output.py` showcases the production-quality `post` method. It demonstrates a resilient architecture with comprehensive logging and structured exception handling.

Description of the Script

This Django API view method is the entry point for a design request. It is heavily instrumented with Python's `logging` module to provide deep introspection into the execution flow. It systematically validates inputs, calls the core engineering module within a dedicated `try...except` block, saves the results, and returns a structured response, ensuring that any failure at any stage is caught and handled gracefully.

Python Code

Listing 7.2: Resilient Output View with Structured Logging

```python
#--------------------begin code-------------
... Logger configuration ...
@method_decorator(csrf_exempt, name='dispatch')
class BeamToColumnEndPlateOutputData(APIView):
def post(self, request):
logger.info("Request received for Beam-to-Column End Plate design.")
try:
        cookie_id = request.COOKIES.get('beam_to_column_end_plate_connection_session')
        if not cookie_id:
            logger.error("No session cookie found.")
            return JsonResponse({"success": False, "logs": [{"type": "error", "msg": "Session not found."}]})

        module_api = get_module_api('Beam-to-Column End Plate Connection')
        input_values = request.data
        logger.debug(f'Received input_values: {json.dumps(input_values,
            indent=2)}')

        # --- Stage 1: Input Validation ---
        try:
            validate_input(input_values)
            logger.info("Input validation successful.")
        except (MissingKeyError, InvalidInputTypeError) as e:
            logger.error(f"Input validation failed: {str(e)}")
            return JsonResponse({"success": False, "logs": [{"type": "
               error", "msg": str(e)}]}, status=400)

        # --- Stage 2: Core Logic Execution ---
        output, logs = {}, []
        try:
            logger.info("Calling core module to generate output...")
            output, logs = module_api.generate_output(input_values)
            logger.info(f"Core module execution successful. {len(logs)}
                logs generated.")
        except Exception as e:
            logger.error(f"Exception during core logic execution: {str(
               e)}")
            logger.error(traceback.format_exc())
            return JsonResponse({"success": False, "logs": [{"type": "
               error", "msg": f"Design Engine Error: {e}"}]}, status
               =400)

        # --- Stage 3: Database Persistence ---
        try:
            designObject = Design.objects.get(cookie_id=cookie_id)
            designObject.logs = self.combine_logs(logs)
            designObject.output_values = output
            designObject.design_status = self.check_non_zero_output(
               output)
            designObject.save()
            logger.info("Design results and logs saved to database
               successfully.")
        except Exception as e:
```

```
45            logger.error(f'Failed to save results to database: {str(e)}')
46            # Note: We still return success to the user even if DB save fails.
47
48        logger.info("Returning successful response to frontend.")
49        return JsonResponse({"data": output, "logs": logs, "success": True}, safe=False, status=201)
50
51    except Exception as e:
52        logger.critical(f"Unhandled exception in post method: {str(e)}")
53        logger.critical(traceback.format_exc())
54        return JsonResponse({"success": False, "logs": [{"type": "error", "msg": "An internal server error occurred."}]}, status=500)
55 Use code with caution.
56 #-------------------- end code ---------------
```

Explanation of the Code

- **Line 7**: The entry point of the method logs the initial request, providing an immediate trace.
- **Line 17-23**: The first stage of execution is a dedicated block for input validation. This ensures that corrupted or incomplete data is caught early, before being passed to the complex engineering logic. It returns a specific 400 error.
- **Line 26-34**: The second stage wraps the call to the core design engine (`module_api.generate_output`). Any exception from the complex engineering calculations is caught here, logged with a full traceback, and a specific error message is sent to the user.
- **Line 37-45**: The third stage handles saving the results to the database. Failures at this stage are logged but importantly, the process continues so that the user still receives their design results, making the system more resilient.
- **Line 49-52**: A final, all-encompassing `try...except` block acts as a safety net to

catch any other unforeseen errors, ensuring the server never returns an unhandled 500 error without logging it first.

7.3.3 Full code

The complete source code for all components of this module is available in the project's source code repository.

Chapter 8

Internship Task 5: Refactoring the modules with a Reusable Module Architecture

8.1 Task 5: Problem Statement

Following the successful development of several new modules, a critical architectural issue became apparent: significant code duplication and a lack of scalability in the frontend. Each module, such as "Cover Plate Welded" and "Beam-to-Column End Plate," was built as a large, monolithic React component. These components, while functional, contained hundreds of lines of code that were nearly identical—handling UI layout, session management, state initialization, API submission, modal pop-ups, and log display.

This approach presented several major problems:

- **High Maintenance Overhead:** A bug in the common logic (e.g., in the "Create Design Report" modal) had to be fixed manually in every single module file, a process that is both tedious and highly prone to error.
- **Poor Scalability:** Creating a new module required copying an existing monolithic component and painstakingly modifying hundreds of lines of code. This dramatically slowed down the development of new features for the platform.
- **Inconsistent Implementations:** With code being copied and pasted, subtle vari-

ations and inconsistencies inevitably crept into different modules, leading to a fragmented user experience and unpredictable behavior.

- **Reduced Readability:** The core engineering logic and UI definition for a specific module were deeply entangled with generic boilerplate code, making it difficult for new developers to understand and contribute.

The objective of this task was to fundamentally refactor the entire frontend architecture by creating a generic, reusable, and configuration-driven framework. The goal was to abstract all common logic into a single, powerful component, allowing future modules to be defined almost entirely through declarative configuration objects, drastically reducing code duplication and accelerating future development.

8.2 Task 5: Tasks Done

The refactoring process involved a complete paradigm shift from imperative, monolithic components to a declarative, component-based architecture. The solution was centered around creating a new shared component, `EngineeringModule`, and a set of configuration standards.

1. **Development of a Generic `EngineeringModule` Component:** A new, highly reusable React component was created to encapsulate all logic and UI structure common to every design module. This component is now the single source of truth for:
    - The overall three-panel page layout (Input Dock, 3D Viewer, Output Dock).
    - Session management, including automatically creating a session on load and deleting it on exit using React's `useEffect` hook.
    - All common state management (e.g., for inputs, outputs, logs, modal visibility).
    - The logic for handling user actions like the "Design" and "Reset" buttons.
    - The dynamic rendering of the top navigation menu and its associated actions.

2. **Configuration-Driven UI Generation:** The `EngineeringModule` is designed to be entirely driven by a "configuration object" passed to it as a prop. A new folder, `configs`, was created within each module's directory to house these objects.

    - **Input Dock Configuration (`moduleConfig.js`):** A large JavaScript object was created to define the entire structure and behavior of the Input Dock. An array within this object, `inputSections`, declaratively builds the form. Each object in the array represents a section ("Factored Loads", "Bolt", etc.), and each object within its `fields` array defines a single input widget (e.g., its type, label, options, and any special ‘onChange‘ handlers).
    - **Output Dock Configuration (`moduleOutputConfig.js`):** A similar configuration object was designed for the Output Dock. It defines the different output sections, the data keys to look for in the API response, the display labels, and the configuration for any special modal pop-ups (like detailing diagrams).

3. **Creation of a Generic `BaseOutputDock` Component:** To complement the configuration-driven approach, a reusable `BaseOutputDock` component was created. It accepts the API output data and an `outputConfig` object as props and dynamically renders the entire output display, including all sections, fields, and interactive modal triggers.

4. **Refactoring Existing Modules:** The "Cover Plate Welded" and "Beam-to-Column End Plate" modules were systematically refactored to use this new architecture. Their massive component files were deleted and replaced with a simple file that does little more than import the generic `EngineeringModule`, the module-specific configurations, and the module-specific Output Dock, and render them together.

The new directory structure reflects this clean separation of concerns:

Figure 8.1: Cover Plate Welded Structure

```
coverPlateWelded
├─ components
│   └─ CoverPlateWeldedOutputDock.jsx
├─ configs
│   ├─ coverPlateWeldedConfig.js
│   └─ coverPlateWeldedOutputConfig.js
└─ CoverPlateWelded.jsx
```

Figure 8.2: Beam-to-Column End Plate Structure

```
beamToColumnEndPlate
├─ components
│   └─ BeamToColumnEndPlateOutputDock.jsx
├─ configs
│   ├─ beamToColumnEndPlateConfig.js
│   └─ beamToColumnEndPlateOutputConfig.js
└─ BeamToColumnEndPlate.jsx
```

8.3 Task 5: React Code

This section presents the code that defines the new architecture. It shows how a once-complex module is reduced to a simple composition of a generic component and its specific configurations.

8.3.1 Description of the Scripts

- `BeamToColumnEndPlate.jsx`: This is the "after" picture of the refactoring. The file is now incredibly lightweight. Its only responsibility is to import the generic `EngineeringModule`, the module-specific configuration and Output Dock, and pass them as props.

- `beamToColumnEndPlateConfig.js`: This is the "brain" of the Input Dock. This declarative JavaScript object defines everything the `EngineeringModule` needs to know to build the UI, validate inputs, and construct the API submission payload for this specific module.

- `beamToColumnEndPlateOutputConfig.js`: This object defines the entire structure and content of the Output Dock for the module.

- `BeamToColumnEndPlateOutputDock.jsx`: This simple component acts as a wrapper, passing the raw output data and the specific output configuration to the generic `BaseOutputDock` for rendering.

8.3.2 React Code Snippets

Listing 8.1: The Refactored Main Module Component

```
// File: BeamToColumnEndPlate.jsx
//--------------------begin code-------------
import React from 'react';
import { EngineeringModule } from '../shared/components/
   EngineeringModule';
import { beamToColumnEndPlateConfig } from './configs/
   beamToColumnEndPlateConfig';
import BeamToColumnEndPlateOutputDock from './components/
   BeamToColumnEndPlateOutputDock';
import { menuItems } from '../shared/utils/moduleUtils';

function BeamToColumnEndPlate() {
// The entire module is now just a call to the generic component
// with its specific configuration and OutputDock passed as props.
return (
<EngineeringModule
moduleConfig={beamToColumnEndPlateConfig}
OutputDockComponent={BeamToColumnEndPlateOutputDock}
menuItems={menuItems}
title="Beam-to-Column End Plate Connection"
/>
);
}

export default BeamToColumnEndPlate;
//-------------------- end code ---------------
```

Listing 8.2: Excerpt from the Input Configuration Object

```
// File: configs/beamToColumnEndPlateConfig.js
//--------------------begin code-------------
export const beamToColumnEndPlateConfig = {
sessionName: "Beam-to-Column End Plate Connection",
routePath: "/design/connections/beam-to-column/end_plate",
designType: "Beam-to-Column-End-Plate-Connection",

// Defines the initial state of all inputs
defaultInputs: {
bolt_type: "Bearing Bolt",
connectivity: "Column-Flange-Beam-Web",
load_axial: "0",
load_moment: "2",
load_shear: "2",
// ... more default values
},

// This array declaratively builds the entire Input Dock form
inputSections: [
{
title: "Factored Loads",
fields: [
// Each object here creates one input field in the UI
{ key: "load_shear", label: "Shear Force(kN)", type: "number" },
{ key: "load_moment", label: "Bending Moment (kNm)", type: "number" },
{ key: "load_axial", label: "Axial Force(kN)", type: "number" }
]
},
{
title: "Bolt",
fields: [
{
key: "bolt_diameter",
label: "Diameter(mm)",
type: "customizable", // A special type for our custom modal selector
selectionKey: "boltDiameterSelect",
modalKey: "boltDiameter",
dataSource: "boltDiameterList"
},
// ... more field definitions
]
}
],

// Pure function to build the API payload from the component state
buildSubmissionParams: (inputs, allSelected, lists, extraData) => {
```

```
47 return {
48 "Bolt.Diameter": allSelected.bolt_diameter ? lists.boltDiameterList :
      inputs.bolt_diameter,
49 "Connectivity": inputs.connectivity,
50 "EndPlateType": extraData?.selectedOption,
51 // ... maps all other inputs to the API key format
52 };
53 },
54 };
55 //-------------------- end code ---------------
```

8.3.3 Explanation of the Code

- `BeamToColumnEndPlate.jsx` **(Listing 8.1)**: This file perfectly illustrates the benefit of the new architecture. It is now merely 20 lines long, down from over 1000. Its sole purpose is to compose the generic `EngineeringModule` with the specific configurations (`beamToColumnEndPlateConfig`) and components (`BeamToColumnEndPlateOutputDock`) required for this particular design type.

- `beamToColumnEndPlateConfig.js` **(Listing 8.2)**: This object now holds all the logic that was previously hardcoded inside the monolithic component.
  - **Line 4-7**: Defines metadata for session management and routing.
  - **Line 9-16**: The `defaultInputs` object provides the initial state for the form, completely decoupling default values from the component logic.
  - **Line 19-45**: The `inputSections` array is the core of the dynamic UI. The generic `EngineeringModule` maps over this array to render each section and its fields without having any hardcoded knowledge of them. A field with `type : "number"` will render an `<Input type="number"/>`, while one with `type: "customizable"` will render the `<Select>` component tied to a modal.
  - **Line 48-55**: The `buildSubmissionParams` function isolates the complex logic of translating the React state into the precise JSON format required by the backend API. This separation of concerns makes the code far easier to test and maintain.

8.3.4 Full code

The full source code for the generic components and the refactored modules is available in the project's source code repository.

8.4 Task 5: Documentation

To ensure all current and future developers adhere to this new, superior architecture, comprehensive documentation was created for the internal developer manual. This guide outlines the process for creating new frontend modules using the reusable component framework.

8.4.1 Developer Manual: Creating a New Frontend Module (Refactored Architecture)

All new frontend modules must be built using the shared `EngineeringModule` component to ensure consistency, maintainability, and rapid development. To create a new module, follow these steps:

Step 1: Create the Directory Structure

Create a new folder for your module (e.g., `newConnection`) inside the `src/modules` directory. Inside it, create the standard subdirectories: `components` and `configs`.

Step 2: Define the Input Configuration

Create a `newConnectionConfig.js` file inside the `configs` folder. Export a configuration object that defines, at a minimum:

- `sessionName`, `routePath`, `designType`
- `defaultInputs`: An object containing all input keys and their initial values.
- `inputSections`: An array of objects that declaratively defines the entire input form.
- `buildSubmissionParams`: A function that takes the state and returns the formatted API payload.

Step 3: Define the Output Configuration and Component

1. Create a newConnectionOutputConfig.js file in configs. This object will define the sections and fields to be displayed in the output dock.

2. Create a NewConnectionOutputDock.jsx file in components. This component will simply wrap the shared BaseOutputDock, passing it the output data and the new output configuration object.

Step 4: Assemble the Final Module

Create the main NewConnection.jsx file. This file should be very simple, containing only the code required to render the generic EngineeringModule with the configurations and components created in the previous steps passed as props.

Listing 8.3: Template for a New Module Component

```
import React from 'react';
import { EngineeringModule } from '../shared/components/
   EngineeringModule';
import { newConnectionConfig } from './configs/
   newConnectionConfig';
import NewConnectionOutputDock from './components/
   NewConnectionOutputDock';

function NewConnection() {
return (
<EngineeringModule
moduleConfig={newConnectionConfig}
OutputDockComponent={NewConnectionOutputDock}
// ... other props
/>
);
}

export default NewConnection;
```

Chapter 9

Conclusions

This internship at the Free/Libre and Open Source Software for Education (FOSSEE) project has been a period of immense learning and practical application of software engineering principles to solve real-world challenges in the domain of structural engineering. The work performed involved a combination of debugging critical issues, refactoring existing systems for scalability, and developing complex new features from the ground up. This concluding chapter summarizes the key tasks accomplished and the skills cultivated during this fellowship.

9.1 Tasks Accomplished

The internship was structured around a series of progressively complex tasks that provided a comprehensive full-stack development experience. The primary accomplishments are summarized below:

- **Module Debugging and Diagnostics:** The initial tasks involved deep-diving into the existing codebase to resolve critical bugs. This included diagnosing and fixing a silent CAD generation failure in the End Plate module by implementing strategic logging and targeted exception handling, and resolving a persistent session management flaw by re-engineering the session API for robust and reliable state control.

- **Full-Stack Module Development:** A significant portion of the internship was dedicated to the end-to-end development of two new, feature-complete design modules.

  - **Cover Plate Welded Connection:** A new module for designing welded beam splices was built from scratch. This involved creating the React frontend, the Django backend API, and the core Python engineering logic based on IS 800:2007.
  - **Beam-to-Column End Plate Connection:** A more complex moment-resisting connection module was developed, requiring the implementation of advanced engineering calculations for bolt group analysis, prying action, and column stiffener checks.

- **Major Frontend Architectural Refactoring:** The most impactful task was a fundamental re-architecture of the frontend. Monolithic, repetitive React components were replaced with a single, generic, and reusable `EngineeringModule`. This new framework is driven entirely by declarative configuration objects, which has drastically reduced code duplication by over 90%, improved maintainability, and established a scalable pattern that will accelerate all future module development.

9.2 Skills Developed

Through these hands-on tasks, a diverse set of technical and professional skills were developed and honed.

9.2.1 Technical Skills

- **Full-Stack Development:** Gained comprehensive, practical experience in the entire web development stack, utilizing **React.js** for building dynamic and interactive user interfaces, and **Django** with the **Django REST Framework** for creating robust, data-driven backend APIs.
- **Advanced React Proficiency:** Developed a deep understanding of modern React concepts, including advanced state management with `useState` and `useContext` hooks, component lifecycle management with `useEffect` for tasks like session control, and building a scalable frontend with a component-based, configuration-driven architecture.

- **Backend Engineering:** Acquired strong skills in Python-based backend development, including designing RESTful API endpoints, using the Django ORM for database interaction (**PostgreSQL**), and serializing complex data structures.

- **Software Architecture and Design Patterns:** Demonstrated the ability to identify architectural weaknesses (e.g., code duplication) and implement superior, scalable design patterns. The frontend refactoring task was a practical application of the Don't Repeat Yourself (DRY) principle and the move towards a more maintainable, component-based system.

- **Systematic Debugging:** Mastered systematic debugging techniques, including strategic logging, browser network analysis, and targeted exception handling to diagnose and resolve complex issues across both frontend and backend systems.

- **Application of Domain Knowledge:** Successfully translated complex engineering standards (**IS 800:2007**) into accurate and reliable Python algorithms, bridging the gap between structural engineering theory and practical software implementation.

9.2.2 Professional Skills

- **Problem Solving:** Consistently demonstrated the ability to analyze complex technical problems, formulate a clear hypothesis, and execute a methodical plan to reach a robust solution.

- **Code Quality and Best Practices:** Developed a strong commitment to writing code that is not only functional but also clean, readable, maintainable, and scalable, adhering to modern software engineering best practices.

- **Project and Task Management:** Successfully managed and executed long-term, multi-faceted development tasks from initial problem definition through to implementation, testing, and documentation.

- **Technical Documentation:** Gained experience in creating clear and concise technical documentation for developers, contributing to the project's knowledge base and ensuring the maintainability of new features.

Chapter A

Appendix

A.1 Work Reports

Internship Work Report			
Name:	Raghav Sharma		
Project:	Osdag on Cloud		
Internship:	FOSSEE Summer Long Internship 2025		
DATE	**DAY**	**TASK**	**Hours Worked**
11-Feb-2025	Tuesday	Download and Install Osdag and getting familiar	4
12-Feb-2025	Wednesday	Go through reports	4
13-Feb-2025	Thursday	Medical break	0
14-Feb-2025	Friday	Medical break	0
15-Feb-2025	Saturday	Task 0 - Module Documentation: Base plate connection	4
16-Feb-2025	Sunday	Task 0 - Module Documentation: Base plate connection	4
17-Feb-2025	Monday	Task 0 - Module Documentation: Base plate connection	4
18-Feb-2025	Tuesday	Documentation Team Meeting	0
19-Feb-2025	Wednesday	Understanding osdag web with eeshu	4
20-Feb-2025	Thursday	Understanding CAD	3
21-Feb-2025	Friday	CAD Team Meeting | Understanding OCC in Osdag	4
22-Feb-2025	Saturday	Setting up and learning Docker for Osdag Web	4
23-Feb-2025	Sunday	Learning ThreeJS for CAD in Osdag Web	3
24-Feb-2025	Monday	Going through Osdag Web Documentation	4
25-Feb-2025	Tuesday	Installing Osdag Web Environment and running it	4
26-Feb-2025	Wednesday	Understanding report - osdag_internship_2022_23_aaranyak	3
27-Feb-2025	Thursday	Exploring methods to do CAD using ThreeJS	4
28-Feb-2025	Friday	Installing and setting up Postgresql	3
1-Mar-2025	Saturday	Debugging Osdag on Cloud Errors	4
2-Mar-2025	Sunday	Debugging Osdag on Cloud Errors	4
3-Mar-2025	Monday	Fixing Errors with Samarpita	3
4-Mar-2025	Tuesday	Task - CAD model	3
5-Mar-2025	Wednesday	Exams	
6-Mar-2025	Thursday	Exams	
7-Mar-2025	Friday	Exams	
8-Mar-2025	Saturday	Exams	
9-Mar-2025	Sunday	Exams	
10-Mar-2025	Monday	Exams	
11-Mar-2025	Tuesday	Exams	
12-Mar-2025	Wednesday	Exams	
13-Mar-2025	Thursday	Learning ThreeJS for CAD in Osdag Web	3
14-Mar-2025	Friday	Holi	0
15-Mar-2025	Saturday	Implementing End plate 3D model	4
16-Mar-2025	Sunday	Implementing End plate 3D model and debugging	5
17-Mar-2025	Monday	Debugging Errors during Model generation	5
18-Mar-2025	Tuesday	Debugging Errors during Model generation (Input dock)	5
19-Mar-2025	Wednesday	Debugging Errors during Model generation (Module)	4
20-Mar-2025	Thursday	Debugging Errors during Model generation (Output View)	3
21-Mar-2025	Friday	Inspecting session api for possible bugs (End plate debugging) + fixes	4
22-Mar-2025	Saturday	Inspecting end plate input data and data processing (End plate debugging)	4
23-Mar-2025	Sunday	Inspecting end plate frontend (End plate debugging)	4
24-Mar-2025	Monday	Removing redundant logs to check errors (End plate debugging)	4
25-Mar-2025	Tuesday	End plate module final fixes	4
26-Mar-2025	Wednesday	Fixing Session Bug	4
27-Mar-2025	Thursday	Session Bug fix on the backend	4
28-Mar-2025	Friday	Session bug fix on the frontend	3
29-Mar-2025	Saturday	Making session flexible for all modules	4
30-Mar-2025	Sunday	Assisting in Code refactoring	4
31-Mar-2025	Monday	Applying the refactored code and testing it	4
1-Apr-2025	Tuesday	Cover plate welded module frontend	4
2-Apr-2025	Wednesday	Building the input dock UI	3

3-Apr-2025	Thursday	Building the output dock UI	4
4-Apr-2025	Friday	Organizing components	4
5-Apr-2025	Saturday	Creating new route creation for cover plate welded	4
6-Apr-2025	Sunday	Creating model render window for cover plate welded module	4
7-Apr-2025	Monday	Fix the cover plate UI rendering errors	4
8-Apr-2025	Tuesday	Combining Cover plate welded UI	3
9-Apr-2025	Wednesday	Inspecting session connection	4
10-Apr-2025	Thursday	Fixing session bug for cover plate welded	4
11-Apr-2025	Friday	Creating session connection feature for Cover plate welded	3
12-Apr-2025	Saturday	Writing frontend api call to create session	4
13-Apr-2025	Sunday	Medical Issue	0
14-Apr-2025	Monday	Creating the populate API	4
15-Apr-2025	Tuesday	Connecting the session and data population to frontend and bug fixes	4
16-Apr-2025	Wednesday	Creating the Input Data view for Cover Plate Welded	4
17-Apr-2025	Thursday	Creating the cover plate welded module files	4
18-Apr-2025	Friday	EXAMS	
19-Apr-2025	Saturday	EXAMS	
20-Apr-2025	Sunday	EXAMS	
21-Apr-2025	Monday	EXAMS	
22-Apr-2025	Tuesday	EXAMS	
23-Apr-2025	Wednesday	EXAMS	
24-Apr-2025	Thursday	EXAMS	
25-Apr-2025	Friday	EXAMS	
26-Apr-2025	Saturday	EXAMS	
27-Apr-2025	Sunday	EXAMS	
28-Apr-2025	Monday	EXAMS	
29-Apr-2025	Tuesday	debugging postgres container issues	4
30-Apr-2025	Wednesday	create design api debugging	4
1-May-2025	Thursday	debugging the input using osdag desktop app	4
2-May-2025	Friday	debugging output and Rendering values on Output Dock	4
3-May-2025	Saturday	Build CAD for Cover Plate Welded	4
4-May-2025	Sunday	Debugging CAD Issues	4
5-May-2025	Monday	Building modals for the output	4
6-May-2025	Tuesday	Finishing the cover plate welded module (minor fixes + input issues + design report)	5
7-May-2025	Wednesday	Beam to column end plate module input dock	4
8-May-2025	Thursday	Building session api for for beam to column end plate	4
9-May-2025	Friday	Initializing route and folder structure + Building the UI	4
10-May-2025	Saturday	Input dock frontend	4
11-May-2025	Sunday	Input dock populate api (backend)	4
12-May-2025	Monday	Building validation for input and api route for accepting input	4
13-May-2025	Tuesday	Building output dock UI	4
14-May-2025	Wednesday	Building output dock modals	4
15-May-2025	Thursday	Output view to calculate output	4
16-May-2025	Friday	Debugging the core module to work with the api	4
17-May-2025	Saturday	Testing output + fixing missing values bugs	4
18-May-2025	Sunday	Creating the api to link the outputview with the output dock + testing	4
19-May-2025	Monday	Debugging issues related to output rendering	4
20-May-2025	Tuesday	Fixing modal values along with missing values	4
21-May-2025	Wednesday	Building CAD model api	4
22-May-2025	Thursday	Debugging cad issues for beam column end plate	4
23-May-2025	Friday	Linking the cad model api with the frontend and rendering it	4
24-May-2025	Saturday	Sick Leave	0
25-May-2025	Sunday	Sick Leave	0
26-May-2025	Monday	Sick Leave	0
27-May-2025	Tuesday	Testing the entire module + fixing minor bugs	4
28-May-2025	Wednesday	Building navbar functions	4

29-May-2025	Thursday	Final debugging and solving backend related issues	4
30-May-2025	Friday	Beam to Column End Plate (Final debugging, minor bug fixes)	3
31-May-2025	Saturday	Refactoring (Making config)	4
1-Jun-2025	Sunday	Refactoring (Making Output Config)	4
2-Jun-2025	Monday	Refactoring (Input value api)	4
3-Jun-2025	Tuesday	Refactoring (Coupling into Engineering Module)	4
4-Jun-2025	Wednesday	Testing and Debugging refactored module	4
5-Jun-2025	Thursday	Refactoring (Bug Fixes)	4
6-Jun-2025	Friday	Refactoring (Beam column end plate)	4
7-Jun-2025	Saturday	Refactoring (Beam column end plate Input Config)	4
8-Jun-2025	Sunday	Refactoring (Beam Column End plate Output Config)	4
9-Jun-2025	Monday	Refactoring (Engineering Module coupling)	4
10-Jun-2025	Tuesday	Refactoring	4
11-Jun-2025	Wednesday	Testing and Debugging refactored module (minor bug fixes)	3

Bibliography

[1] Django Software Foundation. Django documentation. `https://www.djangoproject.com/`. Accessed: 2025-06-11.

[2] Free Software Foundation. Gnu lesser general public license v3.0. `https://www.gnu.org/licenses/lgpl-3.0.en.html`, 2007. Accessed: 2025-06-11.

[3] Python Software Foundation. Python language reference. `https://www.python.org/`. Accessed: 2025-06-11.

[4] Siddhartha Ghosh, Danish Ansari, Ajmal Babu Mahasrankintakam, Dharma Teja Nuli, Reshma Konjari, M. Swathi, and Subhrajit Dutta. Osdag: A software for structural steel design using is 800:2007. In Sondipon Adhikari, Anjan Dutta, and Satyabrata Choudhury, editors, *Advances in Structural Technologies*, volume 81 of *Lecture Notes in Civil Engineering*, pages 219–231, Singapore, 2021. Springer Singapore.

[5] Ant Group. Ant design documentation. `https://ant.design/`. Accessed: 2025-06-11.

[6] Encode OSS Ltd. Django rest framework documentation. `https://www.django-rest-framework.org/`. Accessed: 2025-06-11.

[7] Meta and community contributors. React documentation. `https://react.dev/`. Accessed: 2025-06-11.

[8] Mr.doob and Three.js authors. Three.js documentation. `https://threejs.org/`. Accessed: 2025-06-11.

[9] MUI. Mui: Material ui documentation. `https://mui.com/`. Accessed: 2025-06-11.

[10] Bureau of Indian Standards. Is 800:2007 - general construction in steel - code of practice (third revision), 2007. Accessed: 2025-06-11.

[11] Poimandres. Drei for react three fiber. `https://github.com/pmndrs/drei`. Accessed: 2025-06-11.

[12] Poimandres. React three fiber documentation. `https://docs.pmnd.rs/react-three-fiber/getting-started/introduction`. Accessed: 2025-06-11.

[13] FOSSEE Project. Fossee news - january 2018, vol 1 issue 3. `https://static.fossee.in/fossee/newsletters/Newsletter-Jan2018.pdf`. Accessed: 2025-06-11.

[14] FOSSEE Project. Fossee website. `https://fossee.in/`. Accessed: 2025-06-11.

[15] FOSSEE Project. Osdag website. `https://osdag.fossee.in/`. Accessed: 2025-06-11.

[16] The Axios Project. Axios http client documentation. `https://axios-http.com/docs/intro`. Accessed: 2025-06-11.