FOSSEE Semester Long Internship Report

On

Development of Structural Visualization and UI for OsdagBridge

Submitted by

Vaibhav Tiwari

3rd Year B.Tech Student, Department of CSE & AI

VIT Bhopal University

Sehore

Under the Guidance of

Prof. Siddhartha Ghosh

Department of Civil Engineering

Indian Institute of Technology Bombay

Mentors:

Nidhi Khare

Aditya Donde

February 15, 2026 – May 15, 2026

Abstract

This report documents the work carried out during the FOSSEE Semester Long Internship at the Indian Institute of Technology Bombay, under the Osdag project. The internship focused on the development of graphical visualization tools and design verification user interfaces for **OsdagBridge**, a plugin of the Osdag software tailored for composite steel–concrete bridge engineering.

The primary contributions span three interconnected tasks. First, a modular *Graph Engine* and an interactive *Girder 2D Plots Widget* were developed to generate and display four-panel structural diagrams—girder schematic, bending moment diagram (BMD), shear force diagram (SFD), and deflection curve—directly from grillage analysis results. The engine enforces a strict separation between data extraction and rendering, enabling independent girder selection, load case switching, and two interactive exploration modes.

Second, a comprehensive *Steel Design Dialog* was built as a three-tab PySide6 dialog presenting girder details, analysis results with embedded matplotlib visualizations, and IRC 22:2015 design compliance checks. The Design Check tab features custom utilization bar and status badge widgets, rendering demand–capacity ratios with mathematical equations for eight structural checks.

Third, a *Deck Design Dialog* was implemented to display reinforced concrete deck properties, reinforcement details, and utilization summaries with pass/fail indicators for twelve ULS and SLS design checks.

All modules were developed using Python, PySide6, matplotlib, and numpy, adhering to the established OsdagBridge design system and architectural patterns.

Acknowledgments

I would like to express my sincere gratitude to all those who made this internship a truly enriching experience.

I am deeply grateful to **Prof. Siddhartha Ghosh**, Department of Civil Engineering, IIT Bombay, for his guidance and vision as the Osdag Principal Investigator, whose direction shaped the goals and standards of this project.

I extend my heartfelt thanks to the Osdag project staff—**Ajmal Babu M. S.**, **Nidhi Khare**, **Aditya Donde**, **Ajinkya Dahale**, and **Parth Karia**—for their continuous mentorship, technical guidance, and patient code reviews throughout the internship. Their expertise in structural engineering and software architecture was invaluable in helping me understand the domain and build production-quality code.

I am thankful to **Prof. Kannan M. Moudgalya**, FOSSEE Project Investigator, Department of Chemical Engineering, IIT Bombay, for providing this opportunity through the FOSSEE initiative.

I would also like to thank **Usha Viswanathan**, **Vineeta Parmar**, and the entire FOSSEE management team for their administrative support and for ensuring a smooth internship experience.

I acknowledge the support from the **National Mission on Education through Information and Communication Technology (ICT)**, Ministry of Education (MoE), Government of India, for their role in facilitating this project.

I am grateful to my fellow interns—**Kavish Bishnoi**, **Sarthak Shrivastava**, and **Manav Sharma**—for the collaborative discussions and camaraderie that made this experience both productive and enjoyable.

Finally, I thank VIT Bhopal University, my department, and my family for their constant encouragement and support during my studies and this internship.

Contents

Chapter 1

Introduction

1.1 National Mission in Education through ICT

The National Mission on Education through ICT (NMEICT) is a scheme under the Department of Higher Education, Ministry of Education, Government of India. It aims to leverage the potential of ICT to enhance teaching and learning in Higher Education Institutions in an anytime-anywhere mode.

The mission aligns with the three cardinal principles of the Education Policy—**access, equity, and quality**—by:

- Providing connectivity and affordable access devices for learners and institutions.
- Generating high-quality e-content free of cost.

NMEICT seeks to bridge the digital divide by empowering learners and teachers in urban and rural areas, fostering inclusivity in the knowledge economy. Key focus areas include:

- Development of e-learning pedagogies and virtual laboratories.
- Online testing, certification, and mentorship through accessible platforms like EduSAT and DTH.
- Training and empowering teachers to adopt ICT-based teaching methods.

For further details, visit the official website: www.nmeict.ac.in.

1.1.1 ICT Initiatives of MoE

The Ministry of Education (MoE) has launched several ICT initiatives aimed at students, researchers, and institutions. The table below summarizes the key details:

No.	Resource	For Students/Researchers	For Institutions
Audio-Video e-content			
1	SWAYAM	Earn credit via online courses	Develop and host courses; accept credits
2	SWAYAMPRABHA	Access 24x7 TV programs	Enable SWAYAMPRABHA viewing facilities
Digital Content Access			
3	National Digital Library	Access e-content in multiple disciplines	List e-content; form NDL Clubs
4	e-PG Pathshala	Access free books and e-content	Host e-books
5	Shodhganga	Access Indian research theses	List institutional theses
6	e-ShodhSindhu	Access full-text e-resources	Access e-resources for institutions
Hands-on Learning			
7	e-Yantra	Hands-on embedded systems training	Create e-Yantra labs with IIT Bombay
8	FOSSEE	Volunteer for open-source software	Run labs with open-source software
9	Spoken Tutorial	Learn IT skills via tutorials	Provide self-learning IT content
10	Virtual Labs	Perform online experiments	Develop curriculum-based experiments
E-Governance			
11	SAMARTH ERP	Manage student lifecycle digitally	Enable institutional e-governance
Tracking and Research Tools			
12	VIDWAN	Register and access experts	Monitor faculty research outcomes
13	Shodh Shuddhi	Ensure plagiarism-free work	Improve research quality and reputation
14	Academic Bank of Credits	Store and transfer credits	Facilitate credit redemption

Table 1.1: Summary of ICT Initiatives by the Ministry of Education

1.2 FOSSEE Project

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

1.2.1 Projects and Activities

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

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

1.2.2 Fellowships

FOSSEE offers various internship and fellowship opportunities for students:

- Winter Internship
- Summer Fellowship
- Semester-Long Internship

Students from any degree and academic stage can apply for these internships. Selection is based on the completion of screening tasks involving programming, scientific computing, or data collection that benefit the FLOSS community. These tasks are designed to be completed within a week.

For more details, visit the official FOSSEE website.

Figure 1.1: FOSSEE Projects and Activities

1.3 Osdag Software

Osdag (Open steel design and graphics) is a cross-platform, free/libre and open-source software designed for the detailing and design of steel structures based on the Indian Standard IS 800:2007. It allows users to design steel connections, members, and systems through an interactive graphical user interface (GUI) and provides 3D visualizations of designed components. The software enables easy export of CAD models to drafting tools for construction/fabrication drawings, with optimized designs following industry best practices [1, 2, 3]. Built on Python and several Python-based FLOSS tools (e.g., PyQt and PythonOCC), Osdag is licensed under the GNU Lesser General Public License (LGPL) Version 3.

1.3.1 Osdag GUI

The Osdag GUI is designed to be user-friendly and interactive. It consists of

- **Input Dock**: Collects and validates user inputs.
- **Output Dock**: Displays design results after validation.
- **CAD Window**: Displays the 3D CAD model, where users can pan, zoom, and rotate the design.
- **Message Log**: Shows errors, warnings, and suggestions based on design checks.

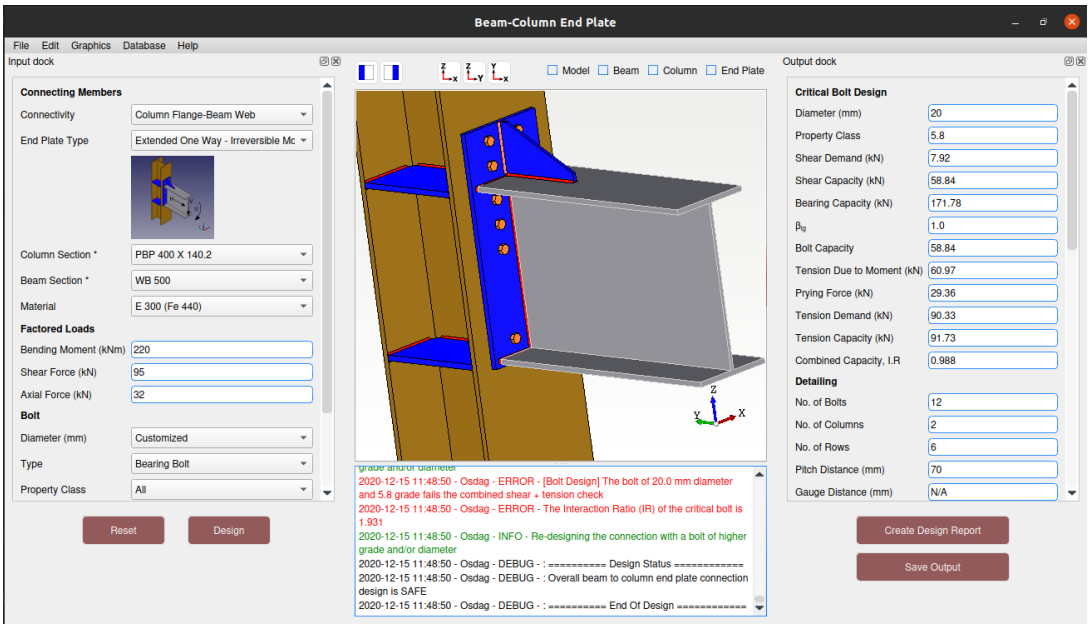

Figure 1.2: Osdag GUI

1.3.2 Features

- **CAD Model**: The 3D CAD model is color-coded and can be saved in multiple formats such as IGS, STL, and STEP.
- **Design Preferences**: Customizes the design process, with advanced users able to set preferences for bolts, welds, and detailing.
- **Design Report**: Creates a detailed report in PDF format, summarizing all checks, calculations, and design details, including any discrepancies.

For more details, visit the official Osdag website.

Chapter 2

Screening Task

2.1 Problem Statement

The screening task for the FOSSEE Semester Long Internship under the Osdag project required candidates to extract internal force data from an `Xarray` dataset and generate structural visualization plots using Python. Specifically, the task comprised two sub-tasks:

1. **Task 1:** Generate 2D Shear Force Diagram (SFD) and Bending Moment Diagram (BMD) for the central longitudinal girder using data from the provided `screening_task.nc` Xarray dataset.

2. **Task 2:** Generate 3D MIDAS-style SFD and BMD for all longitudinal girders, using node coordinates, element connectivity, and force/moment values from the Xarray dataset.

2.2 Dataset Description

The provided Xarray dataset (`screening_task.nc`) stores all internal forces and moments in a single variable: `forces(Element, Component)`. The relevant components used in this project are:

- `Mz_i`, `Mz_j` — Bending Moment at the i-node and j-node of each element
- `Vy_i`, `Vy_j` — Shear Force at the i-node and j-node of each element

Data extraction was performed using coordinate-based selection in Xarray. The girder geometry was reconstructed from node coordinates and element connectivity data stored in separate Python modules.

2.3 Tasks Done

2.3.1 Task 1: 2D SFD and BMD

For the 2D visualization, the force data for the central longitudinal girder was extracted by identifying the girder's element IDs from the connectivity table and selecting the corresponding force components from the dataset.

The continuity of the force array was ensured by:

- Using the `_i` values (start-node forces) for all elements along the girder path
- Appending the `_j` value (end-node force) of the final element

This produces a continuous array of length $n_{\text{elements}} + 1$, suitable for plotting against the nodal x-coordinates. No manual sign changes were applied; the dataset sign convention was preserved as provided. The resulting SFD and BMD were rendered using `matplotlib`.

2.3.2 Task 2: 3D MIDAS-Style Visualization

For the 3D visualization, the approach extended the 2D extraction to all longitudinal girders simultaneously. The key steps were:

1. **Geometry Construction:** Node coordinates were loaded from `data/node.py`, and element connectivity from `data/element.py`. Girder paths were identified by tracing connected element sequences along the bridge length.
2. **Force Extraction:** For each girder, the same i/j extraction and concatenation logic from Task 1 was applied.
3. **3D Extrusion:** The bridge geometry lies in the X–Z plane. Force and moment magnitudes were extruded along the global Y-axis, with one continuous polyline per girder. A color scale encodes the force/moment magnitude.

This approach closely matches MIDAS post-processing visualization style. The interactive 3D plots were rendered using `Plotly`.

2.4 Repository Structure

The screening task solution was organized as follows:

Listing 2.1: Screening Task Repository Structure

```
osdag-screening/
  |-- data/
  |      |-- screening_task.nc      # Xarray dataset
  |      |-- node.py                # Node coordinates
  |      |-- element.py             # Element connectivity
  |-- utils/
  |      |-- data_loader.py         # Xarray force extraction
  |      |-- geometry_loader.py     # Girder geometry construction
  |      |-- plotting.py            # 2D plotting (Matplotlib)
  |      |-- plotting_3d.py         # 3D plotting (Plotly)
  |-- task1_sfd_bmd.py              # Task-1: 2D SFD & BMD
  |-- task2_3d_sfd_bmd.py           # Task-2: 3D MIDAS-style
  |-- report.pdf
  |-- recording.mp4
```

The complete source code is available at: `https://github.com/VT69/osdag-screening`

2.5 Key Code

The core data extraction logic from `task1_sfd_bmd.py` demonstrates the Xarray-based force extraction approach:

Listing 2.2: task1_sfd_bmd.py — 2D SFD/BMD Generation (excerpt)

```
from utils.data_loader import ForceDataLoader
from utils.plotting import plot_bmd, plot_sfd


central_elements = [15, 24, 33, 42, 51, 60, 69, 78, 83]
```

```
loader = ForceDataLoader("data/screening_task.nc")
x, Vy, Mz = loader.extract_sfd_bmd(central_elements)

plot_bmd(
    x, Mz,
    title="Bending Moment Diagram - Central Longitudinal Girder",
    save_path="E:/osdag-screening/results/bmd_central_girder.png"
)

plot_sfd(
    x, Vy,
    title="Shear Force Diagram - Central Longitudinal Girder",
    save_path="E:/osdag-screening/results/sfd_central_girder.png"
)
```

2.6 Documentation

- **Video Demonstration:** https://youtu.be/E2rb8BRoz1M
- **Source Repository:** https://github.com/VT69/osdag-screening
- **Dependencies:** numpy, matplotlib, plotly, xarray

Chapter 3

Graph Engine and Girder 2D Plots Widget

3.1 Problem Statement

The Osdag Bridge (OsdagBridge) application performs grillage analysis of composite steel–concrete plate girder bridges using the `ospgrillage` library. Once analysis is complete, the backend produces a comprehensive results dataset containing element-wise forces, moments, reactions, and displacements for every load case across all girders. The challenge was to design and implement a modular visualization subsystem that:

1. Extracts structural data from the analysis results handler (`PlateGirderAnalysisResults`) without direct access to the underlying `xarray` dataset or `openseespy` model.

2. Renders four synchronized structural diagrams—girder schematic, bending moment diagram (BMD), shear force diagram (SFD), and deflection curve—in a four-panel matplotlib figure embedded within the PySide6 UI.

3. Supports interactive exploration through two distinct modes: *Maximum Values* display and *Scroll for Values* cursor-based interpolation.

4. Enforces a strict separation between data extraction, array processing, and rendering logic to facilitate future API migration.

3.2 Tasks Done

3.2.1 Architecture Design

The visualization subsystem was designed as a three-layer engine encapsulated within a single class, `GirderGraphEngine`, ensuring zero PySide6 imports and complete isolation from the UI framework. The architecture is illustrated in Figure 3.1.

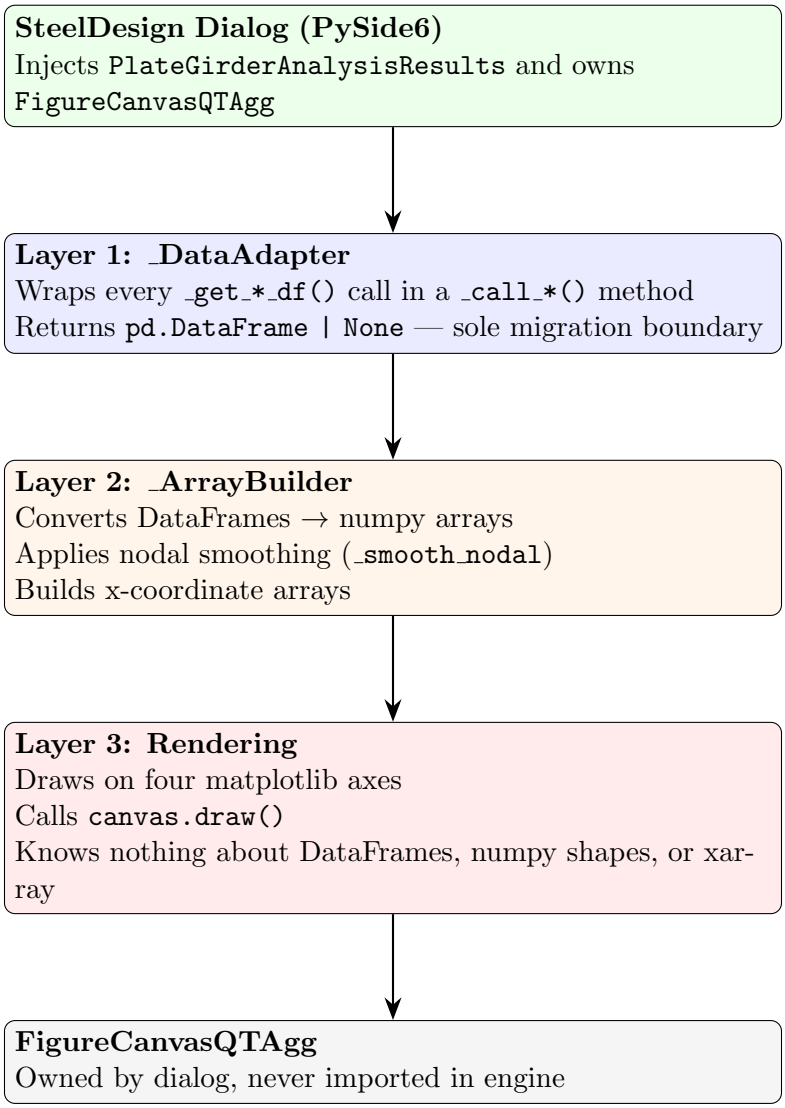

Figure 3.1: GirderGraphEngine — Three-Layer Architecture

Layer 1: _DataAdapter

The `_DataAdapter` layer acts as the sole migration boundary. Each method wraps exactly one `_get_*_df()` call on the injected `PlateGirderAnalysisResults` handler:

- `_call_girder_paths()` — fetches girder path table (Girder, Start, End, Length, Nodes)
- `_call_node_coords(girder)` — fetches per-node X/Y/Z coordinates
- `_call_forces(load_case, girder, component)` — fetches element-wise force values
- `_call_reactions(load_case, girder)` — fetches support reactions (R_A, R_B)
- `_call_loads(load_case)` — parses load descriptors for scheme overlay rendering
- `_call_envelopes()` — fetches max/min envelope data for moving load cases
- `_call_displacements()` — fetches per-node displacement values

The contract is: return `pd.DataFrame | None`. `None` means data is unavailable—never an empty DataFrame, never a partial result.

Layer 2: _ArrayBuilder

The `_ArrayBuilder` converts DataFrames into numpy arrays suitable for matplotlib. The key algorithm is **nodal smoothing** (`_smooth_nodal`), which assembles a continuous nodal force array from per-element i-node and j-node values:

- **First node:** $v_i[0]$ (no smoothing at boundary)
- **Interior nodes:** Average of the incoming right-face value ($-v_j[k-1]$) and the outgoing left-face value ($v_i[k]$) — the sign flip converts from element-local to global frame
- **Last node:** $-v_j[-1]$ (flipped to global sense)

Layer 3: Rendering

The rendering layer operates exclusively on numpy arrays and matplotlib axes. It draws:

- **Girder Schematic (top):** Support symbols (A, B), reaction annotations, beam line, and load overlays (UDL fill regions, point load arrows)
- **BMD (second panel):** Filled red curve with zero-line reference

- **SFD (third panel):** Filled blue curve with zero-line reference
- **Deflection (bottom):** Black curve with inverted y-axis

All visual parameters are read from a centralized `_STYLE` dictionary, ensuring no magic numbers appear in rendering methods.

3.2.2 Interactive Girder 2D Plots Widget

The `Girder2DPlotsWidget` (defined in `girder_2d_plots.py`) provides the UI container for the four-panel figure and manages two interaction modes:

1. **Maximum Values Mode:** On selection, the engine queries the data for peak force/moment values across the girder span and annotates them with markers and value boxes directly on the diagrams.
2. **Scroll for Values Mode:** A vertical cursor line tracks the mouse position across the girder span. As the user hovers, the engine interpolates the BMD, SFD, and deflection values at the cursor x-position and displays them in real-time in side panel fields and annotation boxes.

3.2.3 Steel Design Analysis Tab Integration

The `SteelDesignAnalysisTab` (defined in `steel_design_analysis.py`) integrates the graph engine into the Steel Design dialog. The tab manages:

- A **Component** combo box with HTML-rendered labels (using the custom `RichTextComboBox` widget) for Major (M_z, V_y, D_y), Minor (M_y, V_z, D_z), and Axial (T_x, F_x, D_x) component groups
- A **Display Location** radio group toggling between Maximum Values and Scroll for Values
- A left panel displaying **Support Reactions** (R_A, R_B) and **Maximum Values** for all nine force/displacement components
- A right-hand value column with labels vertically aligned to their corresponding subplot panels via fixed pixel spacers

When the user changes the girder, load combination, or component selection, the tab signals the engine to re-extract data and redraw all four panels.

3.3 Screenshots

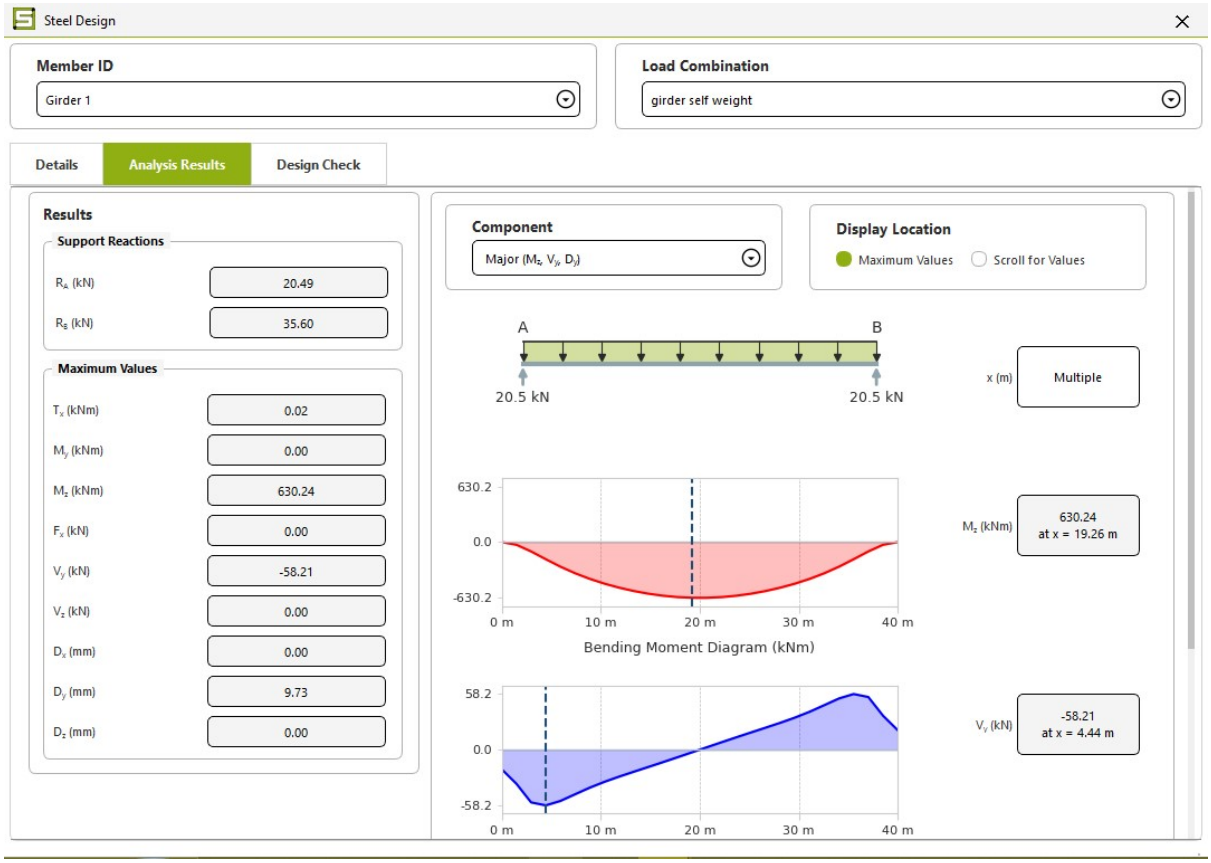

Figure 3.2: Analysis Results Tab — Four-panel girder visualization showing BMD, SFD, support schematic with load arrows, component selection, and interactive value readout

3.4 Python Code

The complete source files for this task are included in the appendix:

- **graph_engine.py** (1858 lines) — `GirderGraphEngine` class with the three-layer architecture
- **girder_2d_plots.py** (525 lines) — `Girder2DPlotsWidget` with interactive cursor and value panel

- **steel_design_analysis.py** (671 lines) — `SteelDesignAnalysisTab` with component combo and signal wiring

Key code excerpts:

Listing 3.1: graph_engine.py — Nodal Smoothing Algorithm (_smooth_nodal)

```
    @staticmethod
    def _smooth_nodal(
        v_i: list[float],
        v_j: list[float],
    ) -> np.ndarray:
        """
        Assemble a nodal-smoothed force array from per-element i-
           node and j-node values.

        Parameters
        ----------
        v_i : list of float
            i-node force values, one per element, in the unit
               already converted
            (kN or kNm).
        v_j : list of float
            j-node force values, one per element, in the same
               unit as ``v_i``.

        Returns
        -------
        np.ndarray
            Nodal array of length ``n_elements + 1``.

        Notes
        -----
        **Beam sign convention**     In openseespy / ospgrillage
           the j-node force
        is returned in the *element's local frame*, where
```

```
    equilibrium requires
``Vy_j = -Vy_i`` for a simply supported beam with no
    intermediate
loads.  To obtain a *global* diagram value at the j-end
    we negate the
stored j value: ``v_global_j = -v_j``.

**Boundary conditions**     the first node value equals
    ``v_i[0]``
exactly (no smoothing at the support); the last node
    value equals
``-v_j[-1]`` exactly.  Interior nodes are the arithmetic
    mean of the
incoming right face value (``-v_j[k-1]``) and the
    outgoing left face
value (``v_i[k]``).

**Single-element edge case**     when there is only one
    element the
array contains exactly two values: ``[v_i[0], -v_j[0]]``.
     No
averaging is performed because there is no interior node.
"""
if not v_i:
    return np.zeros(1, dtype=float)

if len(v_i) == 1:
    return np.array([v_i[0], -v_j[0]], dtype=float)

nodal: list[float] = []

# Start boundary: first element i-node, no smoothing
nodal.append(v_i[0])
```

```
        # Interior nodes: average of right face of previous
            element and left
        # face of next element (both expressed in the global
            sense)
        for k in range(1, len(v_i)):
            v_left  = -v_j[k - 1]   # flip j-node to global sense
            v_right =  v_i[k]
            nodal.append((v_left + v_right) / 2.0)

        # End boundary: last element j-node, flipped to global
            sense
        nodal.append(-v_j[-1])

        return np.array(nodal, dtype=float)
```

3.5 Documentation

3.5.1 Design Decisions

- **Zero PySide6 imports in the engine:** The engine imports only `matplotlib`, `numpy`, and the data handler. This ensures testability without a running Qt event loop and prevents UI framework lock-in.
- **Centralized style dictionary:** All hex colors, line widths, font sizes, and alpha values are defined in a single `_STYLE` dict at module level. No rendering method contains magic numbers.
- **PENDING API pattern:** Features requiring upstream changes (e.g., displacement data, node coordinates) are documented as numbered `PENDING-*` items at the top of the file, with stub methods that return `None` and display user-friendly messages in the UI.
- **Migration boundary:** The `_DataAdapter` layer is designed so that future migration from `_get_*_df()` calls to a `result_handler.query()` API requires changes only to `_call_*()` method bodies—no other part of the file changes.

Chapter 4

Steel Design Dialog Box UI

4.1 Problem Statement

After grillage analysis completes, the bridge engineer needs to verify the steel girder design against IRC 22:2015 code provisions. The challenge was to create a comprehensive, three-tab dialog box within the OsdagBridge PySide6 application that presents:

1. **Details Tab:** Read-only display of girder geometry, material properties, shear connector specifications, section properties, and stiffener details.

2. **Analysis Results Tab:** Interactive four-panel structural diagrams (BMD, SFD, deflection) with girder/load case selection and two exploration modes (as described in Chapter 3).

3. **Design Check Tab:** Eight IRC 22:2015 compliance check cards showing governing equations, computed demand/capacity values, utilization ratios, and pass/fail status.

The dialog must adhere to the existing OsdagBridge design system (rounded card frames, Osdag green accent, consistent typography) and must not modify any backend structural analysis logic.

4.2 Tasks Done

4.2.1 Details Tab (`steel_design_details.py`)

The Details tab implements a scrollable layout with five information sections, all displayed as read-only cards:

- **Dimensional Details:** Grade of material, section type, section designation, section class, total depth, web thickness, flange widths and thicknesses, torsional/warping restraint, web type, and effective slab width. The card layout follows a two-column form pattern with left-aligned labels (230 px minimum width) and right-expanding `QLineEdit` fields with the shared field style.
- **Shear Connector Details:** Material, diameter, height, transverse spacing, number of studs per section, and average longitudinal spacing.
- **Section Properties:** Mass, sectional area, second moments of area (I_z, I_y), radii of gyration (r_z, r_y), elastic moduli (Z_z, Z_y), plastic moduli (Z_{pz}, Z_{py}), torsion constant (I_t), and warping constant (I_w). Labels use HTML rich text for proper subscript/superscript rendering.
- **Stiffener Details:** A `NoScrollTable` (custom `QTableWidget` subclass that passes wheel events to the parent scroll area) displaying intermediate, longitudinal, and bearing stiffener properties.
- **CAD Placeholders:** Two placeholder areas for cross-section and elevation views, to be populated at runtime when the CAD model is mounted.

Data is loaded via the `load_data(cad_state)` method, which populates all fields from a dictionary snapshot.

4.2.2 Analysis Results Tab (`steel_design_analysis.py`)

The Analysis Results tab (described in detail in Chapter 3) integrates the `GirderGraphEngine` into a split-panel layout. The left panel (fixed 350 px width) contains the results card with Support Reactions and Maximum Values sub-groups. The right panel contains the

diagram card with Component and Display Location controls above the four-panel figure canvas.

Key UI components developed for this tab include:

- **RichTextComboBox:** A custom `QComboBox` subclass that supports HTML rendering for its items. It overrides `paintEvent()` to render HTML via `QTextDocument`, and uses a custom `_HTMLDelegate` to render formatted text (e.g., M_z, V_y, D_y) in the dropdown list.

- **Signal wiring pipeline:** When the user changes the Component combo, the dialog's `_update_analysis_plots()` method is triggered, which calls the engine's rendering layer with the appropriate force components for the selected group (Major, Minor, or Axial). The `update_rhs_labels()` method simultaneously updates the right-hand side labels to reflect the correct subscripts.

4.2.3 Design Check Tab (`steel_design_check.py`)

The Design Check tab is the most complex component. It renders eight IRC 22:2015 compliance check cards in a two-column grid layout:

Table 4.1: Design Check Cards — IRC 22:2015 Checks

Left Column	**Right Column**
Strength Limit State (Flexure)	Resistance to Longitudinal and Transverse Shear
Strength Limit State (Shear)	Resistance to Fatigue
Interaction	Stress Limitation
Lateral Torsional Buckling	Deflection and Crack Control

Each card displays five layers:

1. **Bold title** — check name

2. **Styled equation box** — governing equation rendered via matplotlib mathtext (with HTML fallback). Each equation set is rendered into a `QPixmap` and cached per application session to avoid repeated matplotlib Figure creation.

3. **Computed values** — demand and capacity with units, displayed as rich text

4. **Utilization ratio** — colored DCR label (`green` for ≤ 1.0, `red` for > 1.0)

5. **Visual indicators** — `UtilizationBar` (horizontal fill-bar) and `StatusBadge` (PASS/-FAIL label)

Custom Widgets: UtilizationBar and StatusBadge

Two reusable widgets were created for visual status indication:

- **UtilizationBar:** A `QWidget` subclass with a fixed height of 10 px. It contains an inner `QWidget` whose width is proportional to the demand/capacity ratio (capped at 1.0). The fill color changes dynamically: `#28a745` (green) when DCR $\leq$ 1.0 and `#dc3545` (red) when DCR $>$ 1.0. The bar automatically resizes on parent `resizeEvent`.
- **StatusBadge:** A `QLabel` subclass (60×22 px) with three states: PASS (green text on green background), FAIL (red text on red background), and neutral (grey).

The `populate_from_results()` Pipeline

The `populate_from_results(demand, capacity, engine)` method is the public API called by `steel_design.py` after the IRC 22:2015 design check pipeline completes. It receives three objects from the backend:

1. `demand` — `DemandEnvelope` containing factored force demands
2. `capacity` — `CapacityResults` from `IRC22CapacityCalculator.compute_all()`
3. `engine` — `DCREngine` (with `run_all_checks()` already called)

The method maps engine check IDs (1–8) to card keys, computes composite results for multi-check cards (e.g., deflection takes the worst of live and total checks), and populates each card independently so that a failure in one card never prevents others from rendering. Finally, the summary bar is refreshed with aggregate pass/fail counts.

4.2.4 Summary Bar

A summary bar at the top of the Design Check tab shows the total number of checks, passed count, failed count, and an overall StatusBadge. The bar uses the Osdag green accent (`#90AF13`) as its border color and a light green background (`#f0f4e8`).

4.3 Screenshots

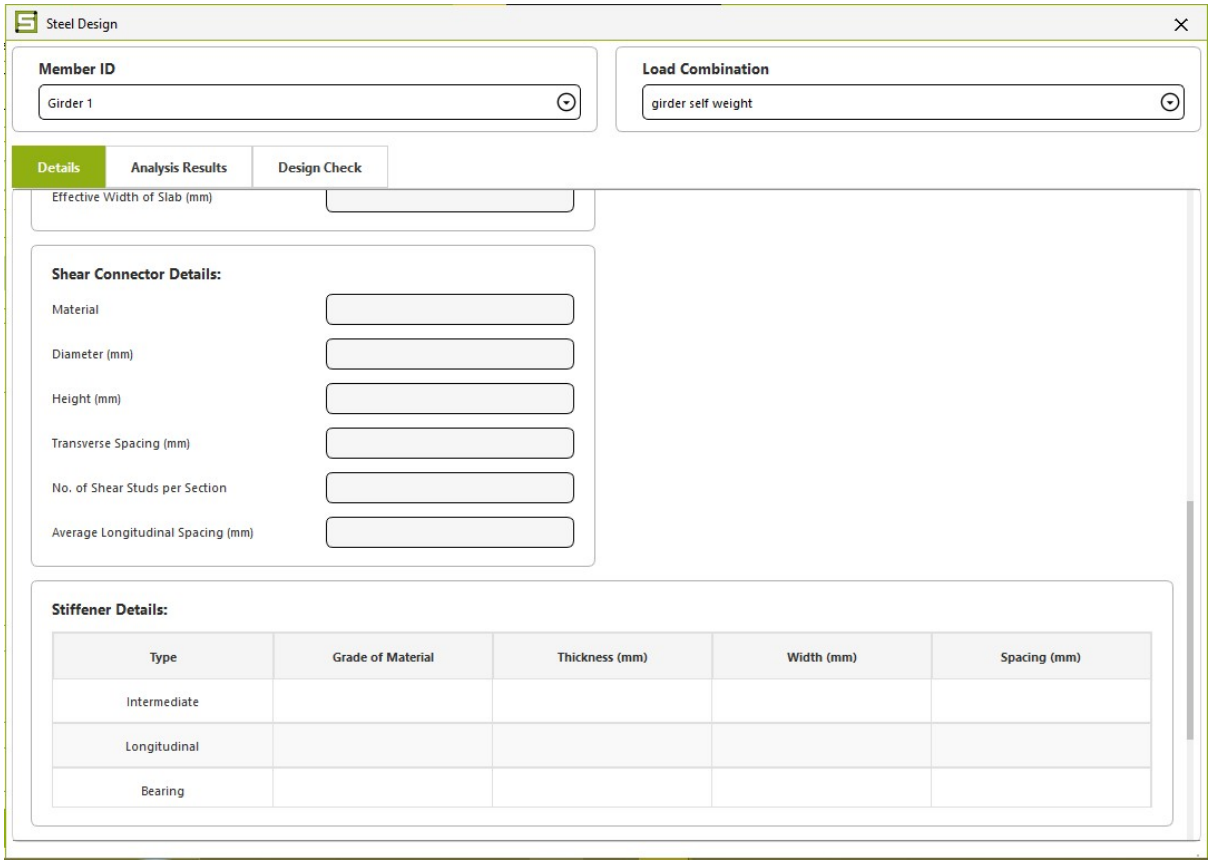

Figure 4.1: Steel Design Dialog — Details Tab showing dimensional details, shear connector details, and stiffener table

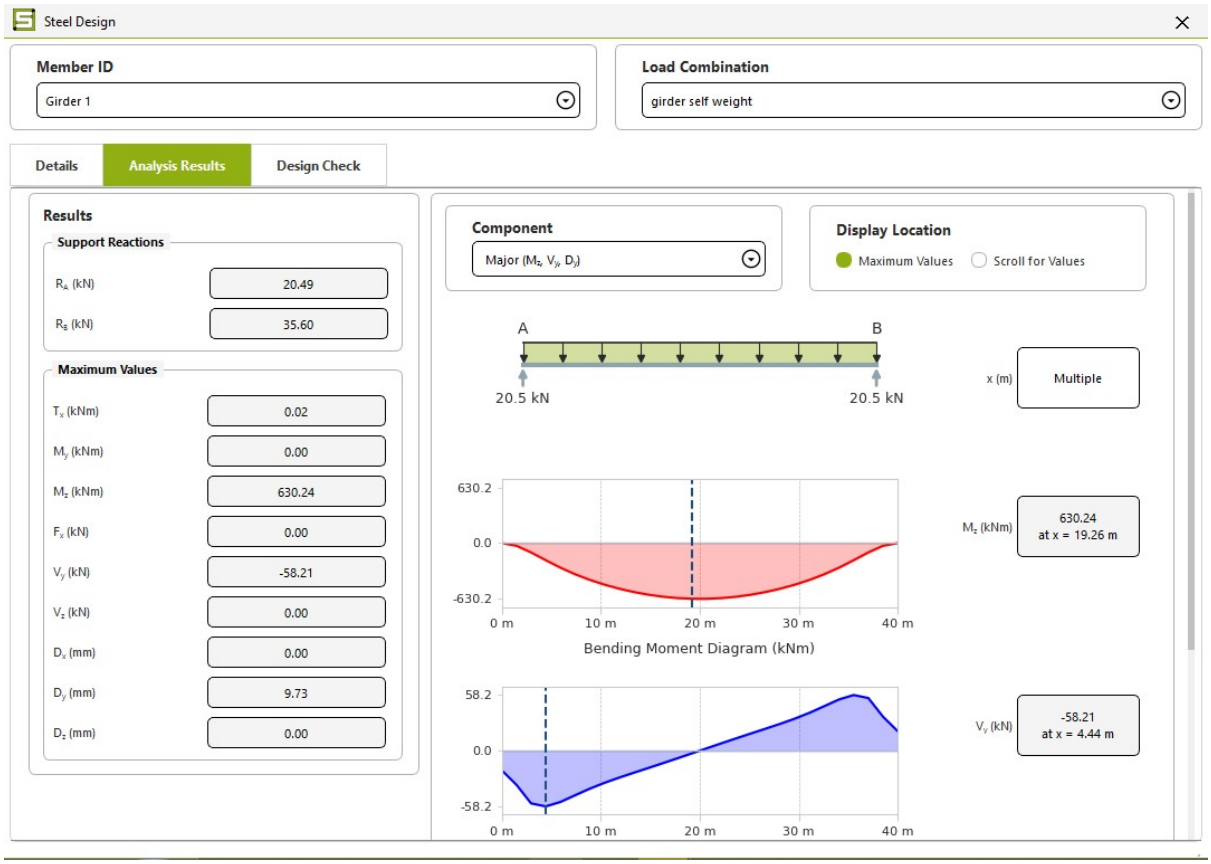

Figure 4.2: Steel Design Dialog — Analysis Results Tab showing four-panel girder visualization with interactive controls

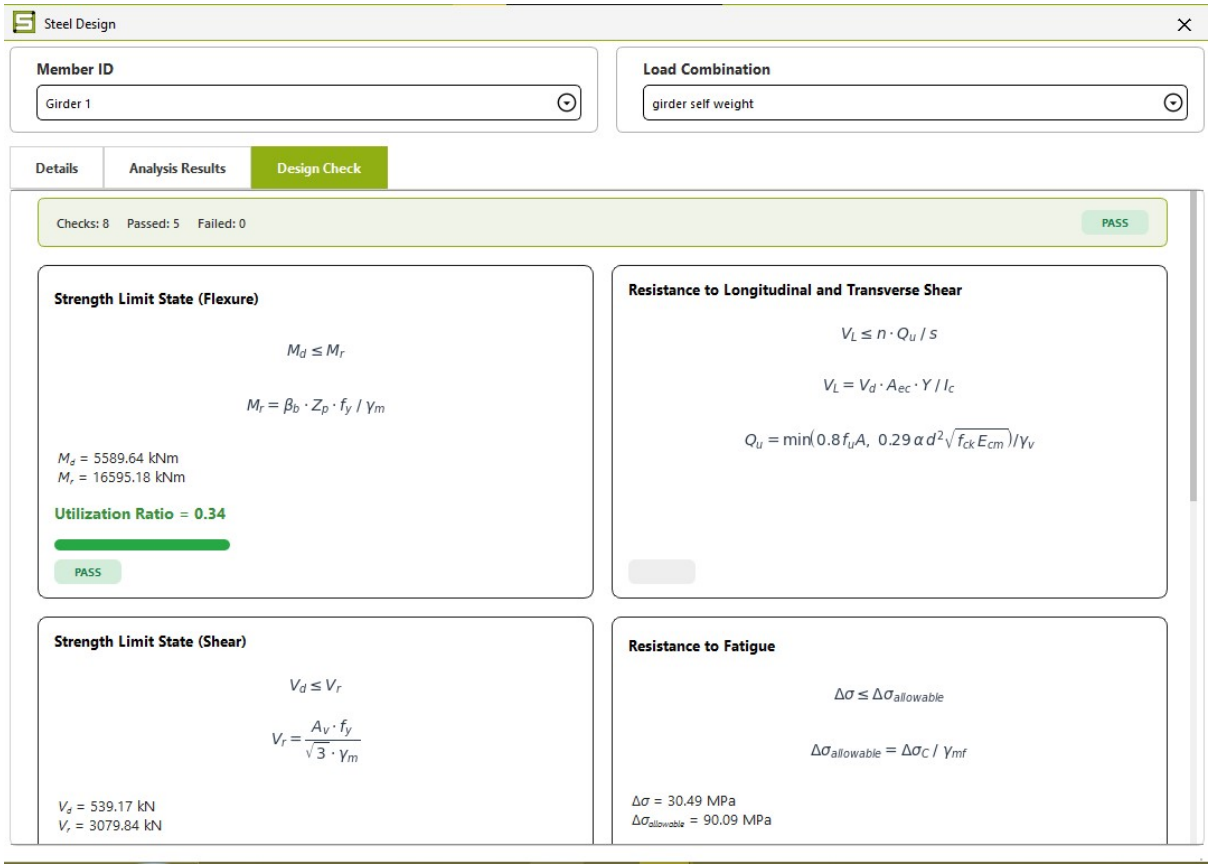

Figure 4.3: Steel Design Dialog — Design Check Tab showing IRC 22:2015 compliance check cards with equations, utilization bars, and pass/fail badges

4.4 Python Code

The complete source files for this task are:

- **steel_design_details.py** (459 lines) — Details tab with dimensional, shear, section property, and stiffener cards
- **steel_design_analysis.py** (671 lines) — Analysis Results tab with RichTextComboBox and signal wiring
- **steel_design_check.py** (858 lines) — Design Check tab with UtilizationBar, StatusBadge, mathtext rendering, and `populate_from_results`

Key code excerpt — the `UtilizationBar` and `StatusBadge` widgets:

Listing 4.1: steel_design_check.py — UtilizationBar and StatusBadge Custom Widgets

```
class UtilizationBar(QWidget):
```

```
    """Horizontal fill-bar showing demand/capacity ratio (0-1+)."""

    def __init__(self, parent=None):
        super().__init__(parent)
        self.setFixedHeight(10)
        self.setStyleSheet("background: #e0e0e0; border-radius: 5px;")
        self._fill = QWidget(self)
        self._fill.setFixedHeight(10)
        self._ratio = 0.0
        self._fill.resize(0, 10)
        self._update()

    def set_ratio(self, ratio: float):
        self._ratio = ratio
        self._update()

    def resizeEvent(self, event):
        super().resizeEvent(event)
        self._update()

    def _update(self):
        fill_w = int(min(self._ratio, 1.0) * self.width())
        color = "#28a745" if self._ratio <= 1.0 else "#dc3545"
        self._fill.setFixedWidth(max(0, fill_w))
        self._fill.setStyleSheet(f"background: {color}; border-radius: 5px;")


class StatusBadge(QLabel):
    """Small PASS / FAIL label badge with coloured background."""

    def __init__(self, parent=None):
```

```
        super().__init__(parent)
        self.setFixedSize(60, 22)
        self.setAlignment(Qt.AlignCenter)
        self.set_neutral()

    def _apply(self, state):
        text, fg, bg = _BADGE_STYLES[state]
        self.setText(text)
        self.setStyleSheet(
            f"color: {fg}; background: {bg};"
            "font-weight: bold; font-size: 10px; border-radius: 6
               px;"
        )

    def set_pass(self):
        self._apply("pass")

    def set_fail(self):
        self._apply("fail")

    def set_neutral(self):
        self._apply("neutral")
```

4.5 Documentation

4.5.1 Design Decisions

- **Mathtext over HTML for equations:** Equations are rendered using matplotlib's built-in mathtext engine (not system LaTeX) into cached `QPixmap` objects. This provides professional typesetting (fractions, square roots, Greek letters) without requiring a LaTeX installation. An HTML fallback ensures graceful degradation if mathtext fails.

- **Card-per-check architecture:** Each design check is an independent card with

its own equation label, value label, DCR label, utilization bar, and status badge. This ensures that a rendering failure in one check card never affects the others.

- **Consistent design system:** All cards, fields, labels, and combo boxes reuse the same style constants (`_FIELD_STYLE`, `_CARD_STYLE`, `_GROUP_STYLE`) ensuring visual consistency with the rest of the OsdagBridge application.
- **No backend modification:** The UI strictly consumes data from the backend pipeline. No structural analysis logic was modified or introduced in the UI layer.

Chapter 5

Deck Design Dialog Box UI

5.1 Problem Statement

In a composite steel–concrete bridge, the reinforced concrete deck slab is a critical structural element that must satisfy both Ultimate Limit State (ULS) and Serviceability Limit State (SLS) design checks. The OsdagBridge backend provides a `design_deck_slab()` function that computes deck properties, reinforcement requirements, and utilization ratios for twelve design criteria. The challenge was to create a user-facing dialog that presents these results in a clear, organized layout consistent with the established OsdagBridge design system.

5.2 Tasks Done

5.2.1 Dialog Layout

The `DeckDesign` dialog (defined in `deck_design.py`, 539 lines) implements a scrollable layout with four main sections:

1. Deck Properties Card

A read-only card displaying the concrete deck slab configuration:

- **Grade of Material:** Concrete grade (e.g., M15)
- **Thickness (mm):** Slab thickness

- **Deck Overhang (mm):** Cantilever overhang distance

Each field uses the shared OsdagBridge field style (`apply_field_style`) with a `QGridLayout` for consistent label–field alignment.

2. Reinforcement Details Table

A `NoScrollTable` (custom `QTableWidget` subclass) displaying reinforcement specifications across three positions:

Table 5.1: Reinforcement Details Table Columns

Column	**Description**
Position	Top Layer / Bottom Layer / Overhang
Material Yield Strength (MPa)	Rebar grade (e.g., 500 for Fe500)
Diameter (mm)	Bar diameter
Spacing (mm)	Centre-to-centre spacing
Clear Cover (mm)	Concrete cover to reinforcement
Area (mm^2)	Computed reinforcement area per unit length

The table mirrors the styling of the Lane Details table in the Typical Section Details dialog—same font, padding, alternating row colors, and header treatment—ensuring visual consistency across the application.

3. Utilization Summary

The utilization summary section displays twelve design checks with horizontal fill bars and pass/fail status badges. The checks span both ULS and SLS categories:

Table 5.2: Deck Design Utilization Checks

ULS Checks	**SLS Checks**
ULS – Bottom (Sagging)	SLS – Bottom Concrete Stress
ULS – Top (Hogging)	SLS – Bottom Steel Stress
ULS – Overhang	SLS – Top Concrete Stress
ULS – Transverse Shear	SLS – Top Steel Stress
	SLS – Deflection
	SLS – Crack Width (Bottom)
	SLS – Crack Width (Top)
	SLS – Crack Width (Overhang)

Each check row contains:

- A text label with the check name

- A `UtilizationBar` showing the demand/capacity ratio visually
- A numeric UR (Utilization Ratio) label
- A `StatusBadge` indicating PASS or FAIL

4. Design Check Summary

A text area at the bottom provides a formatted summary of the deck design check outcome, rendered via HTML-formatted `QLabel` content.

5.2.2 Data Flow

The dialog receives a `deck_results` dictionary from the backend `design_deck_slab()` function. The `load_data()` method maps dictionary keys to the appropriate UI elements:

1. Deck properties are extracted and displayed in the properties card
2. Reinforcement details populate the three table rows
3. Utilization ratios drive the bar widths, badge states, and numeric labels

The dialog does not perform any structural calculations—it is a pure visualization layer that consumes pre-computed backend results.

5.3 Screenshots

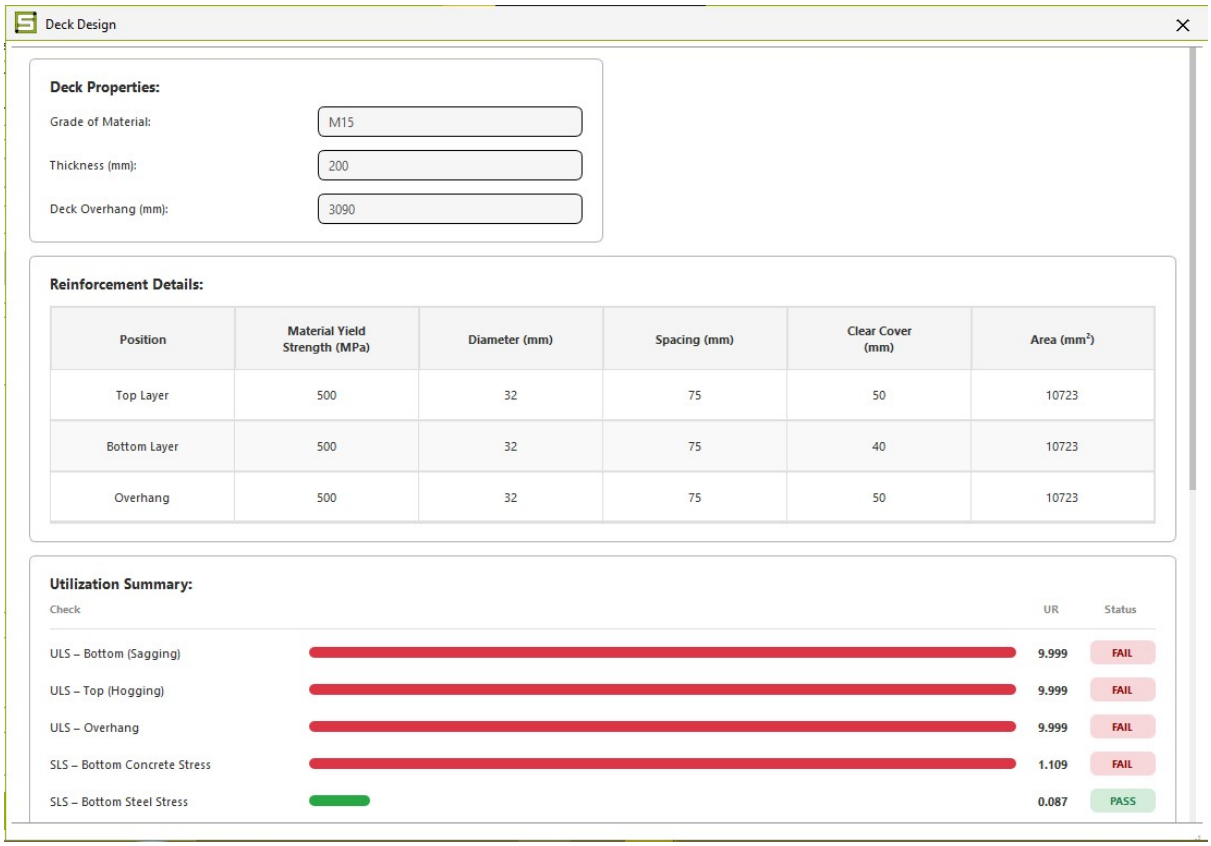

Figure 5.1: Deck Design Dialog — showing deck properties, reinforcement details table, and utilization summary with ULS/SLS checks

5.4 Python Code

The complete source file for this task is:

- **deck_design.py** (539 lines) — `DeckDesign` dialog with property cards, reinforcement table, and utilization summary

5.5 Documentation

5.5.1 Design Decisions

- **UI-only implementation:** The dialog strictly consumes pre-computed results from the backend `design_deck_slab()` function. No structural calculations or reinforcement sizing logic was implemented in the UI layer.

- **Consistent table styling:** The reinforcement table mirrors the Lane Details Inputs table styling (same stylesheet, padding, alternating-row colors, header treatment) to maintain visual consistency across the application.
- **Dynamic utilization bars:** Each check's `UtilizationBar` dynamically adjusts its fill width and color based on the utilization ratio, providing an immediate visual indicator of design adequacy.
- **Graceful handling of missing data:** The `load_data()` method silently ignores missing or invalid keys in the input dictionary, ensuring the dialog renders partial results without crashing.

Chapter 6

Conclusions

6.1 Tasks Accomplished

During the FOSSEE Semester Long Internship (February 15 – May 15, 2026), the following contributions were made to the OsdagBridge project:

1. **Graph Engine and Girder 2D Plots Widget:** Designed and implemented a three-layer visualization engine (`GirderGraphEngine`) comprising a data adapter, array builder, and rendering layer. The engine generates four-panel structural diagrams (girder schematic, BMD, SFD, deflection) from grillage analysis results, with two interactive exploration modes. The engine enforces zero PySide6 imports, centralized styling, and a clearly documented API migration boundary. Integrated the engine into the Steel Design dialog via the `SteelDesignAnalysisTab`.

2. **Steel Design Dialog Box:** Built a comprehensive three-tab PySide6 dialog (`SteelDesign`) encompassing:

    - A *Details Tab* displaying girder geometry, shear connector details, section properties, and stiffener tables
    - An *Analysis Results Tab* with embedded matplotlib figures, RichTextComboBox for component selection, and interactive cursor-based value exploration
    - A *Design Check Tab* with eight IRC 22:2015 compliance check cards featuring mathtext-rendered equations, utilization bars, and status badges, populated through the `populate_from_results()` pipeline

3. **Deck Design Dialog Box:** Implemented a UI dialog for reinforced concrete deck slab design verification, displaying deck properties, a reinforcement details table, and twelve ULS/SLS utilization checks with dynamic progress bars and pass/fail indicators.

4. **Report Generation Pipeline:** Contributed to the development of the Osdag-Bridge report generation pipeline, including fixing refactoring the report data processing architecture, and debugging the execution flow.

6.2 Skills Developed

6.2.1 Technical Skills

- **Python GUI Development:** Gained proficiency in PySide6/Qt for building professional desktop applications, including custom widget development, signal–slot wiring, stylesheet-based theming, and scroll area management.

- **Data Visualization:** Developed expertise in matplotlib for generating publication-quality structural engineering plots, including gridspec layouts, canvas embedding in Qt, mathtext equation rendering, and interactive cursor overlays.

- **Scientific Computing:** Worked extensively with `xarray` datasets and `numpy` arrays for structural analysis data extraction, nodal smoothing algorithms, and force array assembly.

- **Software Architecture:** Practiced layered architecture design with strict boundary enforcement (data adapter → array builder → renderer), migration-ready API patterns, and centralized configuration management.

- **Structural Engineering Domain:** Gained working knowledge of IRC 22:2015 composite bridge design provisions, grillage analysis methodology, demand–capacity ratio computations, and structural visualization conventions (BMD, SFD, deflection diagrams).

6.2.2 Professional Skills

- **Collaborative Development:** Worked within a structured mentorship framework with regular code reviews, following established coding conventions and architectural patterns.
- **Documentation:** Practiced comprehensive code documentation including module-level architecture docstrings, method-level parameter/return documentation, and PENDING API tracking.
- **Version Control:** Used Git for version control with meaningful commit messages and branch-based development workflows.

6.3 Future Work

The following items are proposed as future extensions and ongoing improvements for the OsdagBridge project:

- **Interactive 3D CAD/BIM Visualization Canvas:** Integrating an interactive 3D rendering engine (using PySide6 with PyQtGraph or PyOpenGL) directly into the main OsdagBridge workspace. This would provide real-time 3D CAD representation of the bridge geometry, including girders, cross-bracings, shear studs, and deck reinforcement layouts.
- **Splice and Bearing Connection Design Modules:** Extending the steel design checks to include design calculators and interactive UI tabs for bolted/welded girder splices (flange and web plates) and elastomeric or pot bearings in accordance with IRC:83 and IRC:24 standards.
- **Indian Standard Steel Database Integration:** Integrating a standardized SQL-based section database with Osdag's core library, allowing users to select standard Indian Standard (IS) rolled sections or save optimized built-up plate girder profiles directly into a reusable section library.
- **Automated Load Combination Generator:** Expanding the analysis module to automatically generate load combinations (Dead Load, Live Load, Wind, Seismic,

and Temperature effects) in compliance with IRC:6 and IRC:SP:114, removing the requirement for users to prepare load envelopes manually.

- **Support for Continuous and Multi-Span Bridges:** Upgrading the grillage data adapter and visualization engine to handle multi-span layouts, continuous girders, and variable-depth (haunched) profiles along the span length.

- **Design Report PDF Generation:** Enhancing the automated design report pipeline to generate fully formatted PDF reports directly from the design check tabs, integrating user input tables, equation typesetting, and generated force plots.

Chapter A

Appendix

A.1 Work Reports

Internship Work Report			
Name:	Vaibhav Tiwari		
Project:	OsdagBridge (plugin of Osdag)		
Internship:	FOSSEE Semester Long Internship		
DATE	**DAY**	**TASK**	**Hours**
15 Feb 2026	Sunday	Sunday (Off-duty)	0
16 Feb 2026	Monday	Setting up OsdagBridge development environment and resolving dependency issues on local system.	4
17 Feb 2026	Tuesday	Exploring the OsdagBridge codebase and studying the architecture of PyQt/PySide dialogs.	4
18 Feb 2026	Wednesday	Learning the integration patterns of ospgrillage and understanding grillage analysis input/output formats.	6
19 Feb 2026	Thursday	Analyzing the screening task requirements and force extraction from the NetCDF Xarray dataset.	5
20 Feb 2026	Friday	Implementing custom widget components and setting up basic data wiring for testing UI responses.	5

Internship Work Report (Continued)			
DATE	**DAY**	**TASK**	**Hours**
21 Feb 2026	Saturday	Studying the matplotlib embedding techniques inside PySide6 applications.	5
22 Feb 2026	Sunday	Sunday (Off-duty)	0
23 Feb 2026	Monday	Developing coordinate-based selection in Xarray for the central longitudinal girder path.	4
24 Feb 2026	Tuesday	Implementing 2D Shear Force Diagram (SFD) plot rendering with Matplotlib.	8
25 Feb 2026	Wednesday	Implementing 2D Bending Moment Diagram (BMD) plot rendering with Matplotlib.	4
26 Feb 2026	Thursday	Testing 2D plots continuity using elements' i-node and j-node forces.	8
27 Feb 2026	Friday	Creating geometry loaders to reconstruct bridge girder paths from node coordinate modules.	7
28 Feb 2026	Saturday	Implementing 3D MIDAS-style structural diagram rendering using Plotly.	4
01 Mar 2026	Sunday	Sunday (Off-duty)	0
02 Mar 2026	Monday	Extruding force and bending moment values along the vertical axis in 3D coordinate space.	4
03 Mar 2026	Tuesday	Designing interactive Plotly visualization for all longitudinal girders.	4
04 Mar 2026	Wednesday	Creating video demonstration of the screening task solutions for evaluation.	5
05 Mar 2026	Thursday	Documenting the screening task installation steps and pipeline in a markdown README.	5
06 Mar 2026	Friday	Submitting the screening task solution codebase to the public GitHub repository.	8
07 Mar 2026	Saturday	Saturday (Off-duty)	0
08 Mar 2026	Sunday	Sunday (Off-duty)	0

Internship Work Report (Continued)			
DATE	**DAY**	**TASK**	**Hours**
09 Mar 2026	Monday	Installing local packages and studying grillage analysis results handling in OsdagBridge.	8
10 Mar 2026	Tuesday	Reviewing existing UI controls and discussing coding style guidelines with internship mentors.	4
11 Mar 2026	Wednesday	Designing the architecture of the new GirderGraphEngine class.	8
12 Mar 2026	Thursday	Creating the _DataAdapter layer to isolate data queries from the main results handler.	5
13 Mar 2026	Friday	Developing the _ArrayBuilder layer to convert raw pandas DataFrames into structured numpy arrays.	8
14 Mar 2026	Saturday	Implementing the nodal smoothing algorithm to resolve sign changes at element faces.	7
15 Mar 2026	Sunday	Sunday (Off-duty)	0
16 Mar 2026	Monday	Writing test cases for the _smooth_nodal function using mock grillage analysis results.	5
17 Mar 2026	Tuesday	Setting up matplotlib GridSpec layout for a four-panel subplot architecture.	7
18 Mar 2026	Wednesday	Implementing girder support schematic rendering including supports A and B.	8
19 Mar 2026	Thursday	Designing reaction annotations and text position spacing rules on the scheme plot.	6
20 Mar 2026	Friday	Adding UDL and point load overlay visual indicators to the girder schematic.	4
21 Mar 2026	Saturday	Implementing Bending Moment Diagram (BMD) rendering with filled color regions and zero lines.	5
22 Mar 2026	Sunday	Sunday (Off-duty)	0
23 Mar 2026	Monday	Implementing Shear Force Diagram (SFD) rendering with blue color fills.	7

Internship Work Report (Continued)			
DATE	**DAY**	**TASK**	**Hours**
24 Mar 2026	Tuesday	Drafting Deflection curve rendering with an inverted vertical axis.	6
25 Mar 2026	Wednesday	Developing the Girder2DPlotsWidget PySide6 container class.	6
26 Mar 2026	Thursday	Implementing maximum values identification and dynamic labels on the subplots.	5
27 Mar 2026	Friday	Implementing the mouse tracking event filter to support interactive hover coordinates.	5
28 Mar 2026	Saturday	Saturday (Off-duty)	0
29 Mar 2026	Sunday	Sunday (Off-duty)	0
30 Mar 2026	Monday	Adding the vertical cursor line that tracks the mouse pointer.	6
31 Mar 2026	Tuesday	Designing the interpolation logic to calculate force values at arbitrary points along the span.	4
01 Apr 2026	Wednesday	Designing the tabbed layout of the new Steel Design Dialog Box.	4
02 Apr 2026	Thursday	Creating the SteelDesignDetailsTab PySide6 subclass.	7
03 Apr 2026	Friday	Building the Dimensional Details card with two-column aligned line edits.	4
04 Apr 2026	Saturday	Implementing the Shear Connector Details display card layout.	6
05 Apr 2026	Sunday	Sunday (Off-duty)	0
06 Apr 2026	Monday	Adding Section Properties card with rich-text subscript labels for sectional properties.	6
07 Apr 2026	Tuesday	Implementing the custom NoScrollTable widget to prevent scroll wheel capture in nested tables.	8
08 Apr 2026	Wednesday	Building the intermediate, longitudinal, and bearing stiffener table structure.	6

Internship Work Report (Continued)			
DATE	**DAY**	**TASK**	**Hours**
09 Apr 2026	Thursday	Developing the CAD representation placeholders inside the Details tab.	4
10 Apr 2026	Friday	Creating the SteelDesignAnalysisTab to group component controls and support reactions.	7
11 Apr 2026	Saturday	Integrating the plotting widget and component combo box.	8
12 Apr 2026	Sunday	Sunday (Off-duty)	0
13 Apr 2026	Monday	Formatting support reaction display labels with custom unit strings.	4
14 Apr 2026	Tuesday	Academic Break (University Term End Examinations)	0
15 Apr 2026	Wednesday	Academic Break (University Term End Examinations)	0
16 Apr 2026	Thursday	Academic Break (University Term End Examinations)	0
17 Apr 2026	Friday	Academic Break (University Term End Examinations)	0
18 Apr 2026	Saturday	Saturday (Off-duty)	0
19 Apr 2026	Sunday	Sunday (Off-duty)	0
20 Apr 2026	Monday	Creating the SteelDesignCheckTab PySide6 class.	7
21 Apr 2026	Tuesday	Designing the layout for the eight IRC 22:2015 check cards.	4
22 Apr 2026	Wednesday	Writing matplotlib-based equations and typesetting rules for strength limit states.	8
23 Apr 2026	Thursday	Caching mathematical equation QPixmaps to avoid redundant CPU rendering cycles.	6
24 Apr 2026	Friday	Building the custom UtilizationBar widget with paintEvent fill control.	8

Internship Work Report (Continued)			
DATE	**DAY**	**TASK**	**Hours**
25 Apr 2026	Saturday	Implementing dynamic color transitions in the UtilizationBar based on DCR.	6
26 Apr 2026	Sunday	Sunday (Off-duty)	0
27 Apr 2026	Monday	Designing the Deck Design Dialog Box UI layout.	8
28 Apr 2026	Tuesday	Creating the DeckDesign class as a scrollable QDialog subclass.	5
29 Apr 2026	Wednesday	Implementing the Deck Properties card structure and stylesheet.	4
30 Apr 2026	Thursday	Building the Reinforcement Details table mirroring the typical sections inputs style.	4
01 May 2026	Friday	Writing data loading logic for concrete grades, bar diameters, spacings, and clear covers.	5
02 May 2026	Saturday	Designing the twelve ULS/SLS deck check rows inside the Utilization card.	6
03 May 2026	Sunday	Sunday (Off-duty)	0
04 May 2026	Monday	Integrating UtilizationBar and StatusBadge widgets into each deck check row.	4
05 May 2026	Tuesday	Implementing the load_data method to parse the deck design results dictionary.	5
06 May 2026	Wednesday	Writing a formatted HTML summary generator for the overall deck status report.	4
07 May 2026	Thursday	Testing deck dialog with mock backend outputs and checking border cases.	7
08 May 2026	Friday	Investigating path resolution errors in the PDF report generator.	6
09 May 2026	Saturday	Saturday (Off-duty)	0
10 May 2026	Sunday	Sunday (Off-duty)	0
11 May 2026	Monday	Fixing relative resource paths for vector assets and logos in report_generator.py.	7

Internship Work Report (Continued)			
DATE	**DAY**	**TASK**	**Hours**
12 May 2026	Tuesday	Refactoring the OsdagBridge report generation pipeline to eliminate input_dict mutations.	6
13 May 2026	Wednesday	Enhancing the report generation data flow to enforce strict input/results boundaries.	5
14 May 2026	Thursday	Resolving TypeError and AttributeError exceptions during plategirderbridge design check.	6
15 May 2026	Friday	Aligning the internal bridge configuration logic with the designer.py API.	6

Total Hours Worked: 395 Hours

Bibliography

[1] 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.

[2] FOSSEE Project. FOSSEE News - January 2018, vol 1 issue 3. Accessed: 2024-12-05.

[3] FOSSEE Project. Osdag website. Accessed: 2024-12-05.