FOSSEE Semester Long Internship Report

On

Development of IFC Export Module for Osdag

Submitted by

Kavish Bishnoi

3rd Year B.Tech Student, School of Computing Science & Engineering (SCOPE)

VIT Bhopal University

Under the Guidance of

Prof. Siddhartha Ghosh

Department of Civil Engineering

Indian Institute of Technology Bombay

Mentors:

Parth Karia

Ajinkya Dahale

Nidhi Khare

Aditya Donde

22 May 2026

Abstract

This report documents the design, implementation, and internals of the **Osdag IFC Export Module** (`osdag_core.export_ifc`), a software pipeline developed to enable the Osdag structural steel design application to export its three-dimensional connection models into the **Industry Foundation Classes (IFC)** open standard format.

The module was built from scratch as a six-file Python package integrated into the Osdag codebase. It supports **28 distinct structural connection types** spanning shear connections, moment connections, base plates, truss members, and simple plated joints. The pipeline achieves **High Level of Detail (High LOD)** fidelity by accurately modelling parametric structural profiles, fastener hardware (bolts, nuts, washers, anchor bolts), weld geometry, and by physically cutting bolt holes into members and plates using IFC boolean operations in accordance with **IS 800:2007 Table 19** clearance rules.

Additionally, this report outlines the development of a specialised IFC export pipeline for **OsdagBridge**. Documented in the appendices, this module facilitates the parametric extraction of bridge superstructures—including deck slabs, crash barriers, and composite girders—ensuring compliance with structural requirements derived from **IRC:5-2015**.

This document is intended primarily as a **developer handover guide** for future interns and contributors who will maintain or extend this module.

Acknowledgments

I express my sincere gratitude to all who supported me during this internship at FOSSEE, IIT Bombay. I am deeply thankful to **Prof. Siddhartha Ghosh**, Principal Investigator of the Osdag project, Department of Civil Engineering, IIT Bombay, for providing this invaluable opportunity to work on cutting-edge structural engineering software development.

I am immensely grateful to my mentors **Parth Karia**, **Ajinkya Dahale**, **Nidhi Khare**, and **Aditya Donde** for their patient guidance, technical expertise, and constant support. Their insights into pythonOCC, 3D modeling, and software development were invaluable in completing this project successfully. I also thank my fellow interns at FOSSEE for their collaboration and shared learning experiences.

I acknowledge the **National Mission on Education through ICT, Ministry of Education, Government of India** for funding this project. Finally, I am grateful to **VIT Bhopal University** and the **School of Computing Science & Engineering (SCOPE)** for their encouragement in pursuing this internship opportunity.

Kavish Bishnoi
School of Computing Science & Engineering (SCOPE)
VIT Bhopal University

Contents

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:

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.

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

1.2.1 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.

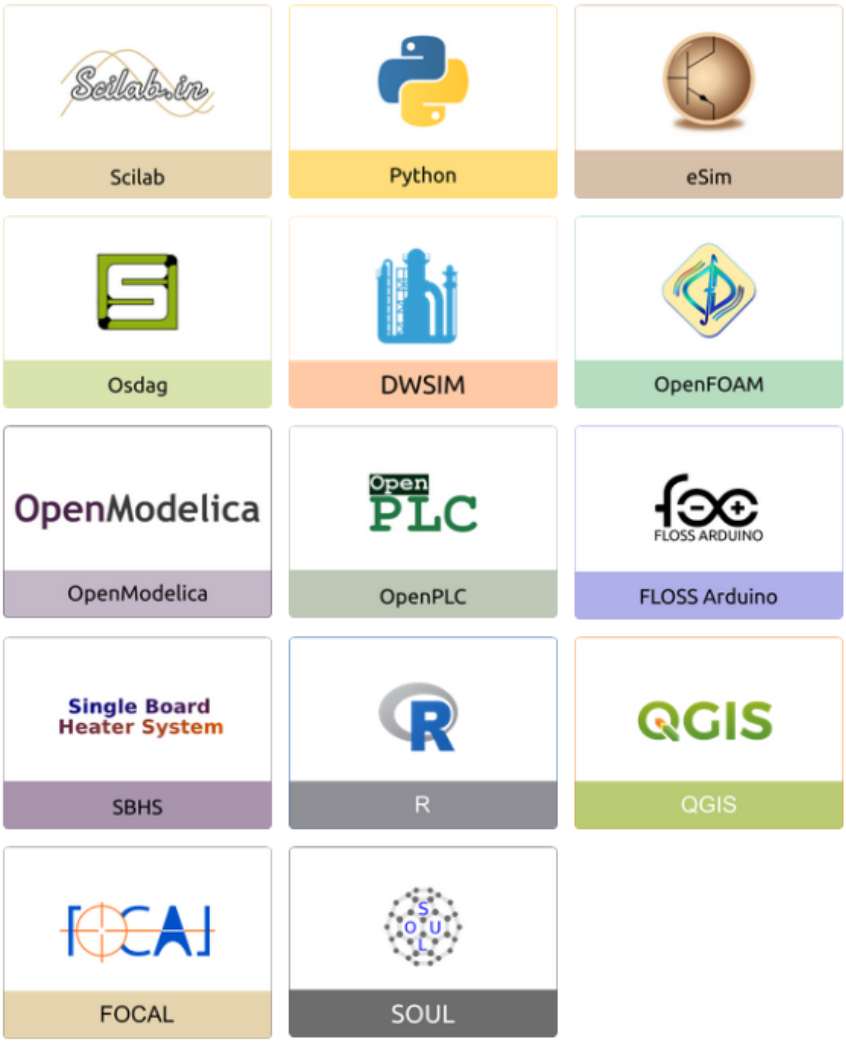

Figure 1.1: FOSSEE Projects and Activities

1.2.2 Fellowships

FOSSEE offers various internship and fellowship opportunities for students:

- Semester Long 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.

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. 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.

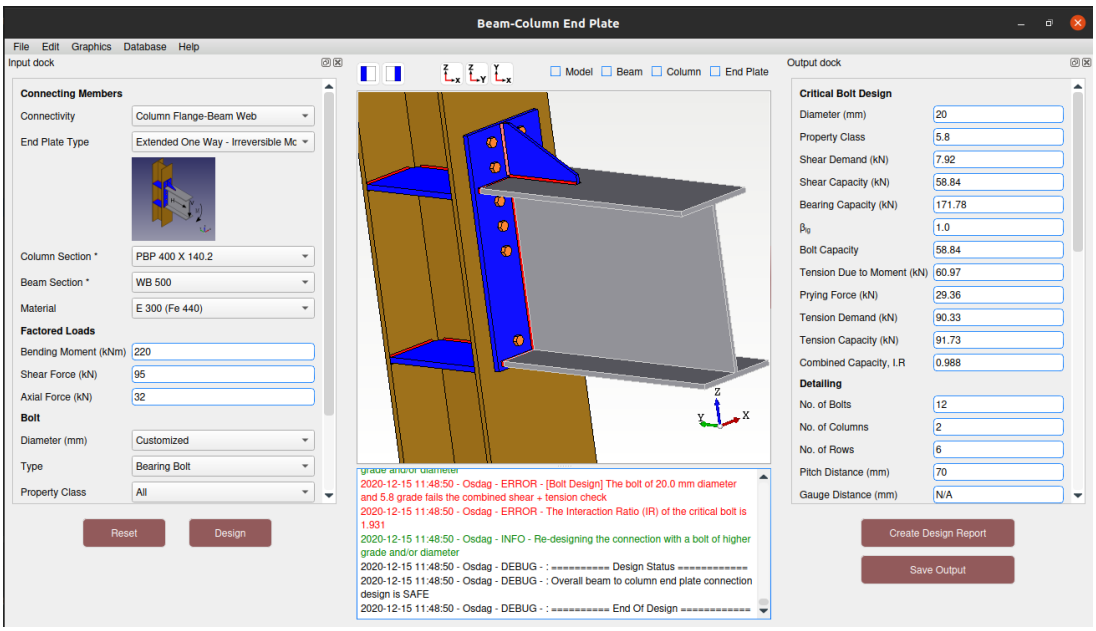

Figure 1.2: Osdag GUI

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.

2. Screening Task

2.1 Problem Statement

Osdag is a cross-platform, free/libre, and open-source software developed for the design and detailing of steel structures in strict accordance with the Indian Standard IS 800:2007. Through an interactive graphical user interface, Osdag empowers engineers to seamlessly design steel connections, members, and complete structural systems. Under the hood, Osdag relies on PythonOCC (OpenCASCADE) to generate robust 3D CAD representations of these designs.

While Osdag successfully outputs standard CAD formats, the modern construction industry heavily relies on Building Information Modelling (BIM) workflows. The primary objective of this screening task was to develop and enhance an initial **IFC (Industry Foundation Classes) wrapper** for Osdag's 3D CAD models. This wrapper acts as a critical bridge, translating Osdag's native geometry into the globally recognised IFC standard, ensuring structural interoperability with major BIM software.

2.2 Methodology and Tasks Done

To achieve the objectives outlined in the problem statement, a systematic approach was adopted. The following tasks were successfully executed:

2.2.1 Environment Setup and Deployment

The development process began by cloning the active Osdag repository from GitHub. The software environment was carefully configured using a dedicated Conda environment to manage the complex dependency tree required by the project. The initial, foundational version of the IFC wrapper present on the `dev` branch was analysed and successfully executed locally to establish a baseline for further enhancement.

2.2.2 Targeted IFC Wrapper Development

The core development effort focused on navigating to the CAD directory (`../Osdag/src/osdag/cad/`) and extending the wrapper logic to support highly specific structural

connection modules. The wrapper was rigorously coded and tested against the following Osdag modules by loading their respective `.osi` input files:

- **Beam-to-Column End Plate** (`/MomentConnections/BCEndplate`)
- **Column-to-Column Cover Plate Welded** (`/MomentConnections/CCSplice CoverPlateCAD`)
- **Beam-to-Beam Cover Plate Bolted** (`/BBCad`)
- **Tension Member - Bolted to End Gusset** (`/Tension`)

For each module, the generated IFC output was visually and geometrically verified against the native Osdag CAD output to ensure strict fidelity in dimensioning, placement, and element classification.

2.2.3 Deliverables and Reporting

Upon successful implementation, the final deliverables were compiled for review:

- **Source Code**: The enhanced wrapper code was packaged alongside a `README.txt` and pushed to a GitHub repository.
- **Demonstration**: A demonstration video was recorded, showcasing the successful loading of an `.osi` file and the subsequent generation of the accurate IFC model.
- **Documentation**: A comprehensive report was authored detailing the specific methodologies employed, the technical challenges encountered during the setup phase, and a bibliography of the references utilized for the development process.

3. Internship Task 1: IFC Export Module for Osdag

3.1 Task 1: Problem Statement

Osdag generates detailed, parametric 3D CAD models of structural steel connections using the **OpenCASCADE (PythonOCC)** geometry kernel. Prior to this internship, these models existed exclusively within the Osdag application window. There was no programmatic pathway to export a designed connection into a vendor-neutral, internationally standardised format recognised by the broader BIM industry.

As a direct consequence, structural engineers could not:

- Integrate Osdag-designed connections into enterprise BIM workflows (e.g., Autodesk Revit, Tekla Structures).
- Share connection models with contractors, fabricators, or certifying authorities in a non-proprietary format.
- Leverage the connection geometry for downstream processes such as clash detection, quantity estimation, or fabrication planning.
- Automatically extract a Bill of Quantities (BOQ) with material grades and section weights directly from the structural model.

The objective of this task was to design, implement, and integrate a complete **IFC (Industry Foundation Classes) export pipeline** into the Osdag codebase. The exporter was required to support all major connection categories, achieve geometric fidelity using standard IFC profile definitions, and attach structured metadata (property sets and quantity take-offs) to every exported element in compliance with buildingSMART International standards.

3.2 Task 1: Tasks Done

The following implementation tasks were completed as part of this internship task:

3.2.1 CAD Extraction Layer

A dispatcher-based extraction engine (`cad_extraction.py`) was developed to identify any of **28 supported CAD connection types** and decompose the live Osdag CAD object into five structured lists: structural members, connection plates, fasteners (bolts, nuts, washers, anchor bolts), welds, and other elements (concrete, grout). A `DISPATCH_MAP` was implemented to route each connection class to its dedicated specialist extractor function.

3.2.2 Geometry Mapping Layer

A comprehensive geometry translation engine (`geometry_mapper.py`) was implemented to convert Osdag's internal parametric objects into IFC-compliant geometric solids. This included:

- Parametric extrusion of **13 distinct structural profile types**: I-sections (with sloped flanges via `IfcArbitraryClosedProfileDef`), channels, angles, double angles, rectangular and circular hollow sections, plates, stiffener plates, gusset plates, concrete blocks, and quarter-cone bearings.
- Complete fastener hardware geometry: hexagonal bolt heads, cylindrical shanks, hex nuts with central voids, square washers, and multi-part anchor bolt assemblies.
- IFC boolean cut operations (`IfcBooleanClippingResult`) that physically remove bolt-hole cylinders from plate and member solids using clearance values derived from **IS 800:2007 Table 19**.

3.2.3 Metadata and Property Set Layer

A metadata engine (`metadata_mapper.py`) was implemented to attach rich, structured data to every IFC element:

- **Property Sets (Psets)**: Material grade, section designation, design standard, weld size, bolt grade.
- **Quantity Take-Off Sets (Qtos)**: Member length, plate area, volume, and mass per metre queried from the Osdag SQLite material database (`Intg_osdag.sqlite`).
- **Element Assembly**: All connection elements are grouped under a single `IfcElementAssembly` entity for clean BIM model organisation.

3.2.4 IFC File Orchestration and GUI Integration

The central class `OsdagIfcExporter` (`ifc_generator.py`) was implemented to initialise the IFC file, construct the full project spatial hierarchy (`IfcProject` → `IfcSite` → `IfcBuilding` → `IfcBuildingStorey`), and coordinate all mappers to write a standards-compliant `.ifc` file. A subprocess isolation layer (`subprocess_ifc_exporter.py`) was also developed to decouple the export from the PyQt GUI thread, preventing UI freezes. The **"Export to IFC"** button was integrated into the Osdag Output Dock GUI.

3.2.5 Pull Request and Merge

Upon successful testing of all 28 connection types, the complete implementation was submitted as a **Pull Request** to the official Osdag GitHub repository on **1st April 2026**. The PR was reviewed, feedback was addressed, and the module was successfully merged into the main codebase.

3.3 Overview of the IFC Export Module

3.3.1 What is IFC?

Industry Foundation Classes (IFC) is an open, neutral, and internationally standardised file format for exchanging and sharing information in the construction and facility management industry. It is maintained by **buildingSMART International** and is the backbone of modern **Building Information Modelling (BIM)** workflows.

An IFC file stores not just three-dimensional geometry, but also rich metadata — material grades, structural loads, element classifications, property sets, and quantity take-offs — making it far more informative than traditional CAD formats such as DXF or STEP. Any certified BIM application, including Autodesk Revit, Tekla Structures, BIMcollab, and the open-source IfcOpenShell viewer, can open and interpret an IFC file without any proprietary plugin.

The two schemas relevant to this project are:

- **IFC2X3** — The most widely deployed schema, supported by virtually all BIM software. This is the **default schema** used by the Osdag exporter (as defined in `ifc_generator.py`, line 12: `schema="IFC2X3"`).
- **IFC4** — The newer schema with improved geometry representations and richer material definitions. The Osdag exporter has been written to be schema-aware and includes conditional branches to support IFC4 in the future.

An IFC file represents a full building project hierarchy: `IfcProject` → `IfcSite` → `IfcBuilding` → `IfcBuildingStorey` → individual structural elements. The Osdag exporter creates this complete hierarchy automatically for every exported connection.

3.3.2 Why Does Osdag Need an IFC Exporter?

Osdag is an open-source structural steel connection design software developed at **IIT Bombay**. It performs rigorous code-based design of steel connections per **IS 800:2007**, the Indian Standard for General Construction in Steel, and generates detailed 3D CAD models of connections using **OpenCASCADE (pythonOCC)** as its geometry kernel.

Prior to this module, the 3D model generated by Osdag existed *only within the application window*. There was no way for a structural engineer to:

- Export the designed connection into a BIM environment such as Revit or Tekla.
- Share the model with other engineers or contractors in a vendor-neutral format.
- Use the connection geometry for clash detection or fabrication workflows.
- Automatically populate a Bill of Quantities (BOQ) from the structural model.

The IFC Export Module resolves all of these limitations. Once a design is completed and approved in Osdag, the engineer can click a single **"Export to IFC"** button in the Output Dock, and the software generates a fully structured, standards-compliant `.ifc` file containing the complete connection model with all its structural metadata.

3.3.3 Scope of Work

The following items were designed, implemented, and tested as part of this internship:

1. **CAD Extraction Layer** (`cad_extraction.py`)
   A dispatcher-based extraction engine that identifies any of **28 supported CAD class types** and decomposes the live Osdag object into five structured lists: structural members, connection plates, fasteners (bolts/nuts/washers/anchor bolts), welds, and other elements (concrete, grout).
2. **Geometry Mapping Layer** (`geometry_mapper.py`)
   A translation engine that converts Osdag's internal parametric objects into IFC-compliant geometric solids. This includes:
   - Parametric extrusion of **13 distinct structural profile types** (I-sections with

tapered flanges, channels, angles, hollow sections, plates, stiffeners, etc.).

- Full hardware geometry: hexagonal bolt heads, cylindrical shanks, hex nuts with central voids, square washers with holes, and anchor bolt representations.
- IFC boolean cut operations that physically remove bolt-hole cylinders from plates and members using `IfcOpeningElement` and `IfcRelVoidsElement`.

3. **Metadata and Property Set Layer** (`metadata_mapper.py`)
   Attaches structured **buildingSMART Property Sets (Psets)** and **Quantity Take-Off sets (Qtos)** to every exported IFC element, enabling full BIM lifecycle tracking including material grade, section designation, design code (`IS 800:2007`), member weight, and fastener specifications.
4. **IFC File Orchestrator** (`ifc_generator.py`)
   The central class `OsdagIfcExporter` that initialises the IFC file, builds the project spatial hierarchy, coordinates the geometry and metadata mappers, groups all elements into an `IfcElementAssembly`, and writes the final `.ifc` file to disk.
5. **Subprocess Isolation Layer** (`subprocess_ifc_exporter.py`)
   A standalone CLI runner that executes the IFC generation in a **separate Python process**, completely decoupled from the main Osdag PyQt GUI thread, preventing application freezes during complex export operations.
6. **Database Connector** (`database_connector.py`)
   A lightweight SQLite interface that queries `Intg_osdag.sqlite` — the Osdag structural section database — to retrieve accurate profile cross-sectional area and mass per metre values for BOQ weight calculations.
7. **GUI Integration** (`output_dock.py` in `osdag_gui`)
   The "Export to IFC" button and its supporting logic were integrated into the existing Output Dock panel of the Osdag GUI, including user-facing validation, a save-file dialog, and a success/failure message box.

3.3.4 Target Audience for This Document

This report is written **for future interns and developers** who will continue, maintain, or extend this module. It assumes the reader has:

- Basic proficiency in **Python 3** (classes, inheritance, attribute access).
- A general understanding of **3D coordinate systems** (origin points, direction vectors).
- Familiarity with the Osdag application at a user level (i.e., they have run at least

one design and seen the 3D model).

No prior knowledge of IFC or `ifcopenshell` is assumed. The necessary concepts are introduced throughout this document where relevant.

Do **not** attempt to run IFC export logic directly on the main PyQt thread. The `subprocess_ifc_exporter.py` isolation layer exists for a critical reason — `ifcopenshell` processing can block the GUI for several seconds on complex connections. Always trigger exports through the subprocess mechanism described in Chapter 4.

3.4 Prerequisites and Setup

This chapter covers everything a new developer needs to get the IFC export pipeline running on their machine.

3.4.1 Required Dependencies

The IFC export module relies on a specific set of Python libraries. These are part of the broader Osdag project dependency tree declared in `requirements.txt` and `pyproject.toml` at the repository root.

The table below focuses specifically on the libraries that the `export_ifc` package directly imports and cannot function without.

Library	Declared In	Purpose in the IFC Module
ifcopenshell	`requirements.txt`	The core IFC engine. Used in `ifc_generator.py`, `geometry_mapper.py`, and `metadata_mapper.py` to create every IFC entity — elements, property sets, geometry solids, GUIDs, and the final `.ifc` file output.

Library	Declared In	Purpose in the IFC Module
numpy	`requirements.txt`	Used throughout `geometry_mapper.py` for all 3D vector arithmetic: normalising direction vectors, computing cross-products to build orthogonal placement axes, and handling coordinate arrays from Osdag CAD objects.
sqlite3	Python Standard Library	Used in `database_connector.py` to query `Intg_osdag.sqlite` for section profile data (cross-sectional area and mass per metre) needed for BOQ weight calculations. No installation needed — it ships with Python.
subprocess	Python Standard Library	Used in `output_dock.py` to launch `subprocess_ifc_exporter.py` as an isolated child process, keeping the PyQt GUI thread responsive during export.
json	Python Standard Library	Used to serialise live Osdag CAD objects into a temporary `.json` file that is passed to the subprocess. The subprocess reads it back and reconstructs objects using the `DictToObj` class.
re, sys, os, time, argparse, uuid	Python Standard Library	Various standard utilities used across the module for string parsing, GUID generation, file paths, timestamps, and CLI argument handling.

Table 3.1: Direct dependencies of the `export_ifc` module.

`ifcopenshell` is **not** available on PyPI via a simple `pip install ifcopenshell`. It must be installed from the correct channel. The recommended method for this project is via **Conda**:

```
conda install -c conda-forge ifcopenshell
```

Alternatively, pre-built wheels are available at

`https://github.com/IfcOpenShell/IfcOpenShell` for manual installation. Installing from the wrong source will result in import errors or missing `.ifc` schema files.

The full Osdag environment (required to run the GUI and trigger IFC export from the interface) is set up using a **Conda environment** with **Python 3.10–3.12**. The version constraint is explicit in `requirements.txt`:

```
python>=3.10,<3.13
```

The complete environment is installed with:

```
# From the repository root (OsdagCoreWork/Osdag/)
pip install -e .
```

3.4.2 How to Run and Test the Exporter Locally

The IFC export module can be tested in two ways: through the full Osdag GUI, or directly from the command line using the subprocess script as a standalone tool. The CLI method is **strongly recommended** during development because it does not require launching the entire application.

Method A: Via the Osdag GUI (End-to-End Test)

1. Launch Osdag from the project root:

   ```
   osdag
   # or equivalently:
   python -m osdag_gui
   ```

2. Open any connection module (e.g., Fin Plate, Base Plate, End Plate).
3. Enter design inputs and click **Design**.
4. Once the design passes (green status), navigate to the **Output Dock** on the right panel.
5. Click the **"Export to IFC"** button.
6. A save-file dialog will appear. Choose a location and confirm.
7. Open the resulting `.ifc` file in any IFC viewer (e.g., **IfcOpenShell Viewer**, **BIMcollab ZOOM**, or **usBIM.viewer+**) to verify the geometry.

Method B: Via Command Line (Developer Testing)

The `subprocess_ifc_exporter.py` script acts as a standalone CLI entry point. It accepts a pre-serialised JSON file representing a connection and produces an `.ifc` file directly — no GUI required.

Step 1: Prepare a JSON file representing the connection. The JSON must follow this schema (each entry is a list of serialised CAD objects):

```
{
   "metadata": {
      "Material": "E250",
      "Profile": "ISMB 400",
      "DesignStatus": "True",
      "ConnectionType": "Fin Plate"
   },
   "members": [ { "_class_name": "ISection", "B": 140, "D": 400, ... } ],
   "plates": [ { "_class_name": "Plate", "L": 250, "W": 200, "T": 12, ... } ],
   "bolts": [ { "_class_name": "Bolt", "r": 10, "R": 18, "H": 60, ... } ],
   "welds": [],
   "others": []
}
```

Listing 3.1: Expected JSON structure for the subprocess exporter

Step 2: Run the exporter script directly:

```
python subprocess_ifc_exporter.py \
   --json path/to/connection_data.json \
   --ifc path/to/output_model.ifc \
   --id  MyConnectionID
```

Step 3: Check the exit code and console output:

- Exit code **0** — Export succeeded. The `.ifc` file has been written to the path specified by `--ifc`.
- Exit code **1** — An exception occurred. The full Python traceback will be printed to `stderr`, which the GUI captures and displays in the error message box.

A quick sanity check without a BIM viewer: use the `ifcopenshell` Python API to inspect the generated file:

```python
import ifcopenshell
model = ifcopenshell.open("output_model.ifc")

# Count exported elements
print(model.by_type("IfcBeam")) # Supported beams
print(model.by_type("IfcColumn")) # Supporting columns
print(model.by_type("IfcPlate")) # Connection plates
print(model.by_type("IfcFastener")) # Bolts, nuts, welds
print(model.by_type("IfcOpeningElement")) # Bolt holes
```

Database Dependency Check

The `database_connector.py` module queries the Osdag SQLite database for profile mass and area. The database is located at:

```
osdag_core/data/ResourceFiles/Database/Intg_osdag.sqlite
```

If this file is missing, the BOQ weight calculations will silently fall back to a steel density estimate ($\rho = 7850$ kg/m^3) using computed volume. The export will still succeed — only the weight values in the `Qto_BeamBaseQuantities` property set will be approximate rather than database-accurate.

3.5 Module Architecture and File Structure

3.5.1 Where Is the Code Located?

The entire IFC export pipeline lives inside a single self-contained Python package located at:

```
Osdag/src/osdag_core/export_ifc/
```

The package consists of **six source files** plus an `__init__.py` package initialiser. The GUI integration point lives separately in the `osdag_gui` package but depends entirely on this module.

The full directory layout is as follows:

```
osdag/
  src/
    osdag_core/
      export_ifc/              <-- The IFC package (this module)
        __init__.py            # Package marker (2 lines)
        cad_extraction.py      # Step 1: Geometry extraction
        geometry_mapper.py     # Step 2: IFC geometry translation
        metadata_mapper.py     # Step 3: Property sets and BOQ
        ifc_generator.py       # Step 4: File orchestrator
        subprocess_ifc_exporter.py # Step 5: Isolated CLI runner
        database_connector.py # Helper: SQLite section lookup
      data/
        ResourceFiles/
          Database/
            Intg_osdag.sqlite  # <-- Required for BOQ weights
    osdag_gui/
      ui/components/docks/
        output_dock.py         # GUI trigger (export_to_ifc method)
```

Everything inside `export_ifc/` is **backend-only** and has no direct dependency on PyQt or the Osdag GUI. This clean separation is what makes the subprocess isolation strategy work — the exporter package can be imported and executed in a headless Python process without any UI framework being present.

3.5.2 Role of Each File

The following table gives a precise one-line description of every file, its key classes and functions, and which other module it calls.

File	Key Classes / Functions	Responsibility
cad_-extraction.py	extract_cad_items(), extract_shear_-connections(), extract_-base_plate(), extract_-moment_endplate(), extract_moment_-coverplate_bolted(), extract_moment_-coverplate_welded(), extract_truss_-connection(), extract_-simple_plated(), assign_ifc_name(), obj_to_dict(), extract_-metadata(), DISPATCH_-MAP	**Entry point for geometry extraction.** Inspects any live Osdag CAD object, identifies its class, and routes it to the correct specialist extractor. Returns five lists: members, plates, bolts, welds, others. Also serialises objects to JSON-safe dictionaries and extracts non-geometric design metadata.
geometry_-mapper.py	GeometryMapper, map_-extruded_solid(), map_-fastener(), map_weld(), _generate_i_section_-polyline(), _create_-bolt_representation(), _create_nut_-representation(), _create_washer_-representation(), _create_anchor_bolt-_representation(), create_opening_-element(), perform_-boolean_cut(), get_-is800_clearance()	**Translates Osdag parametric objects into IFC geometry.** Handles 13 structural profile types via parametric extrusion, hardware geometry for bolts/nuts/washers/anchor bolts, weld solids (fillet and groove), and boolean operations that physically cut bolt holes into members and plates using IS 800:2007 Table 19 clearances.

File	Key Classes / Functions	Responsibility
metadata_mapper.py	MetadataMapper, assign_standard_pset(), assign_standard_qto(), assign_osdag_design_data(), assign_member_boq(), assign_plate_boq(), assign_fastener_boq(), assign_weld_boq(), create_element_assembly()	**Attaches all non-geometric data to IFC elements.** Creates buildingSMART Property Sets (Psets) for material grade, design code, and element specifications. Creates Quantity Take-Off sets (Qtos) for length, area, volume, and weight. Groups all connection elements into a single IfcElementAssembly.
ifc_generator.py	OsdagIfcExporter, __init__(), setup_header(), setup_project_hierarchy(), export_connection(), generate_guid(), _create_placement(), _create_shape_representation(), save()	**Orchestrates the entire IFC export operation.** Initialises the IFC file with schema IFC2X3, builds the full project spatial hierarchy (Project → Site → Building → Storey), calls the geometry and metadata mappers for each element category, and writes the .ifc file to disk.
subprocess_ifc_exporter.py	DictToObj, run_export(), __main__ block with argparse	**Standalone CLI entry point for isolated execution.** Reads a serialised JSON file, reconstructs CAD-like Python objects using DictToObj, calls OsdagIfcExporter, and exits with code 0 on success or code 1 on failure. Launched as a child process by the GUI to prevent UI thread blocking.

File	Key Classes / Functions	Responsibility
database_connector.py	get_db_path(), fetch_profile_data(), fetch_bolt_data()	**SQLite interface to the Osdag section database.** Queries Intg_osdag.sqlite across nine profile tables (Beams, Columns, Channels, Angles, EqualAngle, UnequalAngle, RHS, SHS, CHS) to retrieve mass per metre and cross-sectional area for BOQ calculations. Also looks up bolt yield and ultimate strength by grade.

Table 3.2: Quick reference: all files in export_ifc/ and their responsibilities.

3.5.3 Module Dependency Graph

The files in the export_ifc package have a strictly layered dependency structure. No circular imports exist. The flow of control and data moves in one direction only — from the GUI trigger down through extraction, geometry, metadata, and finally to file output.

```
output_dock.py (osdag_gui)
   |
   |-- imports --> cad_extraction.py
   |              |-- returns: members, plates, bolts, welds, others
   |              |-- returns: serialised JSON dict
   |
   |-- spawns subprocess -->
         subprocess_ifc_exporter.py
            |-- imports --> ifc_generator.py (OsdagIfcExporter)
                          |-- imports --> geometry_mapper.py (
                             ↪ GeometryMapper)
                          |              |-- imports -->
                             ↪ database_connector.py
                          |-- imports --> metadata_mapper.py (
                             ↪ MetadataMapper)
                                         |-- imports -->
                                            ↪ database_connector.py
```

A few key observations about this dependency graph that future developers should keep in mind:

- `cad_extraction.py` is the **only file imported directly by the GUI**. All other files in the package are only ever executed inside the subprocess.
- `database_connector.py` is a **pure utility** with no knowledge of IFC — it is imported by both `geometry_mapper.py` and `metadata_mapper.py` independently. It can fail silently without breaking the export.
- `ifc_generator.py` is the **sole owner** of the `ifcopenshell.file` object. Both mapper classes receive a reference to the exporter instance (`self.exporter`) in their constructors and call `self.ifc_file` through it. This ensures all IFC entities are written to a single, consistent file object.

3.5.4 The IFC File Spatial Hierarchy

Every `.ifc` file written by this module contains the following mandatory spatial hierarchy, which is created by `OsdagIfcExporter.setup_project_hierarchy()` in `ifc_generator.py`:

```
IfcProject ("Osdag Connection Design")
  |
  +-- IfcSite ("Site")
       |
       +-- IfcBuilding ("Building")
            |
            +-- IfcBuildingStorey ("Level 1", Elevation = 0.0)
                 |
                 +-- IfcElementAssembly ("Connection_<id>")
                      |
                      +-- IfcColumn  (Supporting column)
                      +-- IfcBeam    (Supported beam)
                      +-- IfcPlate   (Fin plate / End plate / etc.)
                      +-- IfcFastener (Bolt / Nut / Washer / Weld)
                      +-- IfcBuildingElementProxy (Concrete / Grout)
                      +-- [IfcOpeningElement] (Bolt holes, via
                          ↪ IfcRelVoidsElement)
```

The **unit system** embedded in every exported file is:

Quantity	IFC Unit	Value
Length	`IfcSIUnit`	`MILLI METRE` (millimetres)
Area	`IfcSIUnit`	`SQUARE_METRE`
Volume	`IfcSIUnit`	`CUBIC_METRE`
Mass	`IfcSIUnit`	`KILO GRAM`
Angle	`IfcSIUnit`	`RADIAN`

All geometry coordinates passed to `geometry_mapper.py` must be in **millimetres**. Osdag internally works in millimetres, so this is consistent. However, if you ever add support for a new section type and source geometry from an external library that uses metres, you **must** multiply all coordinates by 1000 before passing them to the IFC entity creation functions. Failing to do so will produce a correct-looking but incorrectly-scaled model (1000× too small).

3.6 The Export Pipeline (Data Flow)

This chapter walks through the entire lifecycle of an IFC export operation, from the moment the user clicks the button in the GUI to the final `.ifc` file being saved to disk. Understanding this seven-step data flow is critical for debugging and extending the module.

3.6.1 Step 1: User Triggers Export (`output_dock.py`)

The process begins in the Osdag GUI. When the user clicks the **"Export to IFC"** button in the Output Dock panel, the `export_to_ifc(self, main, ifc_path=None)` method is invoked.

Validation

Before attempting to export, the UI performs strict validation:

- `main.design_button_status`: Ensures the user has actually clicked the "Design" button and generated a 3D model.
- `main.design_status`: Ensures the connection passed the design checks (is structurally safe). Exporting a failed design to IFC is blocked.

CAD Object Retrieval

Once validated, the script must locate the live 3D CAD object. Because Osdag has evolved organically, different connection modules store their primary CAD object under different attribute names on the `commLogicObj`. The exporter uses a robust fallback loop to find it:

```
possible_attribs = [
    'connectivityObj', 'CPObj', 'CEPObj', 'BPObj', 'TObj',
    'ColObj', 'FObj', 'PGObj', 'design_obj'
]
cad_obj = None
for attr in possible_attribs:
    cad_obj = getattr(self.parent.commLogicObj, attr, None)
    if cad_obj is not None:
        break
```

Listing 3.2: Attribute search loop in `output_dock.py`

If `cad_obj` is successfully found, it represents the entire connection assembly in Osdag's internal Python object structure.

3.6.2 Step 2: CAD Object Classification (`cad_extraction.py`)

Osdag's CAD objects are complex, deeply nested structures. An IFC file, however, needs flat lists of specific entity types (Members, Plates, Fasteners). The `cad_extraction.py` module serves as the translation boundary.

The Dispatcher Pattern

The single entry point to this layer is the `extract_cad_items(cad_obj)` function. It uses a **Dispatcher Pattern** to route the object to a specialist extractor function based on its class name.

```
def extract_cad_items(cad_obj):
    cad_class = type(cad_obj).__name__

    if cad_class in DISPATCH_MAP:
        # Route to the correct specialist extractor
        members, plates, bolts, welds, others = DISPATCH_MAP[cad_class](
            ↪ cad_obj)

        # Tag every item with its parent connection type
        for items_list in [members, plates, bolts, welds, others]:
            for item in items_list:
```

```
            item.parent_connection_type = cad_class
            # ... (Header Plate isolation logic) ...

    return members, plates, bolts, welds, others
```

Listing 3.3: The unified dispatcher in `cad_extraction.py`

The `DISPATCH_MAP` dictionary maps **28 different Osdag CAD class names** (e.g., `FinPlateCAD`, `BasePlateCad`, `StrutAngleWeldCAD`) to one of six specialist extractor functions:

- `extract_shear_connections()`
- `extract_base_plate()`
- `extract_moment_endplate()`
- `extract_moment_coverplate_bolted()`
- `extract_moment_coverplate_welded()`
- `extract_truss_connection()`
- `extract_simple_plated()`

Extraction Output

Every specialist extractor navigates the specific attribute tree of its connection type and returns exactly five lists:

1. **Members**: Structural profiles like Beams and Columns (`ISection`, `Channel`, etc.).
2. **Plates**: Connection plates like Fin Plates, Base Plates, Stiffeners.
3. **Bolts**: Hardware items like Bolts, Nuts, Washers, and Anchor Bolts.
4. **Welds**: Weld geometries.
5. **Others**: Non-steel components like Concrete foundations or Grout.

As seen in the code snippet above, the dispatcher artificially injects a `parent_connection_type` attribute into every extracted item. This is a crucial design decision: it allows downstream geometry mappers to apply connection-specific coordinate shifts (e.g., the High LOD Header Plate offset or the Hollow Base Plate vertical shift) without needing the entire assembly object.

3.6.3 Step 3: JSON Serialisation and Subprocess Launch

Generating an IFC file for a complex connection (e.g., one with dozens of bolts, where each bolt hole requires a boolean subtraction) can take several seconds. If this were executed on the main PyQt thread, the Osdag UI would freeze, and Windows might mark the application as "Not Responding".

To prevent this, the export operation is physically isolated in a separate Python process.

Serialisation (`obj_to_dict`)

The five lists of objects extracted in Step 2 are complex Python instances that cannot be passed directly between processes. They are converted to pure data dictionaries using `obj_to_dict()`:

```
def obj_to_dict(obj):
    d = {'_class_name': type(obj).__name__}
    for k in dir(obj):
        if not k.startswith('_') and k not in ('create_model', 'place', '
            ↪ compute_params', 'getPoint'):
            val = getattr(obj, k)
            if isinstance(val, (int, float, str, bool)):
                d[k] = val
            elif type(val).__name__ == 'ndarray':
                d[k] = val.tolist()
    return d
```

Listing 3.4: JSON Serialiser in `cad_extraction.py`

This safely captures all dimensions (e.g., `D`, `B`, `T`), origins, and NumPy direction vectors (`uDir`, `wDir`), while dropping un-serialisable OpenCASCADE `TopoDS_Shape` solids (which we don't need, since the IFC mapper redraws everything from scratch).

Subprocess Execution

The dictionaries are dumped to a temporary `.json` file. `output_dock.py` then launches the CLI exporter script with a 60-second timeout:

```
result = subprocess.run(
    [sys.executable, exporter_script,
     '--json', tmp.name, '--ifc', ifc_path, '--id', connection_id],
    capture_output=True, text=True, timeout=60
)
```

Listing 3.5: Launching the subprocess in `output_dock.py`

3.6.4 Step 4: Object Reconstruction (`DictToObj`)

Inside the new, isolated Python process, `subprocess_ifc_exporter.py` reads the JSON file. The downstream `geometry_mapper.py` expects to interact with these items using standard dot-notation (e.g., `obj.uDir`, `obj.T`) and checks their type using `type(obj).__name__`.

Because dictionaries don't support dot-notation, and native Python `SimpleNamespace` objects return `"SimpleNamespace"` as their class name (which breaks the geometry dispatcher), a custom wrapper class is used:

```
class DictToObj(SimpleNamespace):
    def __init__(self, dictionary, **kwargs):
        super().__init__(**kwargs)
        for key, value in dictionary.items():
            if isinstance(value, dict):
                setattr(self, key, DictToObj(value))
            elif isinstance(value, list) and len(value) > 0 and isinstance(
                ↪ value[0], dict):
                setattr(self, key, [DictToObj(v) for v in value])
            else:
                setattr(self, key, value)

    @property
    def __class__(self):
        # We need this to fake the class name for geometry_mapper
        class FakeClass:
            __name__ = getattr(self, '_class_name', 'Unknown')
        return FakeClass
```

Listing 3.6: The Object Reconstructor in `subprocess_ifc_exporter.py`

By overriding the `__class__` property to return a `FakeClass`, the reconstructed object successfully fools the `geometry_mapper.py`. When the mapper asks `type(obj).__name__`, it receives `"FinPlateCAD"` or `"ISection"` instead of `"DictToObj"`.

With the objects reconstructed, the script creates an instance of `OsdagIfcExporter` and passes the objects into `export_connection()`, beginning the actual IFC file generation.

3.6.5 Step 5: Geometry Mapping (`geometry_mapper.py`)

With objects successfully reconstructed inside the isolated process, the entry point `OsdagIfcExporter.export_connection()` loops through all five element lists and delegates to `GeometryMapper` for every item.

Structural Members and Plates: `map_extruded_solid()`

All structural members (beams, columns) and connection plates are handled by `map_extruded_solid()`. The method follows a strict three-stage logic:

Stage 1 — Placement. The object's 3D coordinate frame is established first:

```
origin = np.array(getattr(osdag_obj, 'sec_origin', None)
             or getattr(osdag_obj, 'origin', [0,0,0]))
uDir = getattr(osdag_obj, 'uDir', [1,0,0])
wDir = getattr(osdag_obj, 'wDir', [0,0,1])
vDir = getattr(osdag_obj, 'vDir', None)
if vDir is None:
   vDir = np.cross(wDir, uDir)
placement = self._create_placement(origin, wDir, uDir)
```

Listing 3.7: Placement computation in `map_extruded_solid()`

`_create_placement()` forces orthogonal normalisation (`np.dot(dir_z, dir_x) > 0.01`) before writing the `IfcAxis2Placement3D`, preventing degenerate placements.

Stage 2 — Profile Creation. The method branches on `_class_name`. The table below maps every supported class to its IFC profile entity and the dimension attributes it reads:

Class	IFC Profile Entity	Dimensions Read
`ISection`	`IfcArbitraryClosedProfileDef`	`B, D, t, T, R1, R2, alpha` via `_generate_i_section_polyline()`
`Channel`	`IfcArbitraryClosedProfileDef`	`D, B, t, T`; 8-point manual polygon
`Angle / DoubleAngle`	`IfcLShapeProfileDef`	`A, B, T, R1, R2`
`RectHollow`	`IfcRectangleHollowProfileDef`	`H` (XDim), `W` (YDim), `T` (WallThickness)
`CircularHollow`	`IfcCircleHollowProfileDef`	`r` (Radius), `T` (WallThickness)

Class	IFC Profile Entity	Dimensions Read
Plate / stiffener / Continuity Plate	IfcRectangleProfileDef	T (XDim), L (YDim); extruded by W
StiffenerPlate	IfcArbitraryClosed-ProfileDef	8 corner points from L, W, L11, L12, R11, R12, R21, R22
GassetPlate	IfcArbitraryClosed-ProfileDef	H, L, degree; trapezoidal polygon
Stiffener_CAD	IfcArbitraryClosed-ProfileDef	Hst, Lst, Tst; 4-point clipped corner
Concrete / Grout / foundation	IfcRectangleProfileDef	T, L, W
QuarterCone	*RevolveSolid*	b, h, coneAngle via IfcRevolvedAreaSolid

Table 3.3: Profile mapping table in `map_extruded_solid()`.

`_generate_i_section_polyline()` computes a precise multi-point polygon for ISMB/ISLB sections whose inner flanges are sloped. It uses the flange slope angle `alpha` (degrees) to calculate tangent points for the root fillet (radius `R1`) and toe fillet (radius `R2`). If `alpha` falls outside the realistic range 80°–100°, or if both fillet radii are zero, it falls back to a simplified 12-point straight polygon to avoid trigonometric errors.

Stage 3 — Extrusion. All profiles (except `GassetPlate` and `QuarterCone`, which have custom exits) reach the same final statement:

```
return self.ifc_file.createIfcExtrudedAreaSolid(
    SweptArea=profile,
    Position=placement,
    ExtrudedDirection=self.ifc_file.createIfcDirection((0., 0., 1.)),
    Depth=extrusion_depth
)
```

Listing 3.8: Uniform extrusion return statement

Hardware: `map_fastener()` via Instancing

Bolts, nuts, washers, and anchor bolts are handled by `map_fastener()`. The function dispatches on `_class_name`:

- **Bolt** → `_create_bolt_representation()`: A hexagonal prism (bolt head, extruded in $-Z$) and a circular cylinder (shank, extruded in $+Z$) are combined. Head uses `R, T`; shank uses `r, H`.
- **Nut** → `_create_nut_representation()`: Uses `IfcArbitraryProfileDefWithVoids` — a hexagonal outer curve with a circular inner void (`r1`) representing the threaded bore.
- **Washer** → `_create_washer_representation()`: A square outer profile (`a`) with a circular punch-out (`d/2`).
- **AnchorBolt_A / B / Endplate** → `_create_anchor_bolt_representation()`: Since the J/L-shaped anchor geometry cannot be cleanly represented by a simple sweep, a conservative straight cylinder is used with total length `l + ex`.

Each geometry is defined **once** inside an `IfcRepresentationMap`. Every instance across the connection is then placed using a `IfcCartesianTransformationOperator3D` that encodes the bolt's specific `origin`, `uDir`, and `wDir`. This instancing strategy keeps the IFC file compact regardless of the bolt count.

Welds: `map_weld()`

Welds are handled by `map_weld()` and come in three types:

- **Weld** and **Weld_sec**: Rectangular profile (`W, L`) extruded by `T`.
- **FilletWeld**: Right-triangular cross-section (`b, h`) extruded by `L`.

Both apply the `HollowBasePlateCad` vertical correction (subtracting `depth/2` from the Z-origin) where applicable.

3.6.6 Step 6: High LOD Boolean Cuts

This is the most computationally intensive part of the pipeline. To achieve High Level of Detail (High LOD), bolt holes must be physically cut out of every structural member and plate they pass through — they cannot be approximated by textures or decals.

Clearance Standard

Hole size is calculated by `get_is800_clearance(bolt_dia)` per **IS 800:2007 Table 19**:

Nominal Bolt Diameter	Clearance	Hole Radius used
$d \leq 14\,\text{mm}$	$+1$ mm	$(d+1)/2$
$14 < d \leq 24\,\text{mm}$	$+2$ mm	$(d+2)/2$
$d > 24\,\text{mm}$	$+3$ mm	$(d+3)/2$

Opening Cylinder Construction

`create_opening_element()` builds the cutting cylinder:

```
bolt_dia = 2.0 * float(bolt_r)
clearance = self.get_is800_clearance(bolt_dia)
hole_r  = (bolt_dia + clearance) / 2.0

# Start 1x bolt-length before the bolt origin; end 3x beyond it
start_point = np.array(bolt_origin) - 1.0 * float(bolt_len) * wDir

solid = self.ifc_file.createIfcExtrudedAreaSolid(
    SweptArea=IfcCircleProfileDef(Radius=hole_r),
    Position=self._create_placement(start_point, wDir, uDir),
    ExtrudedDirection=(0., 0., 1.),
    Depth=float(bolt_len) * 3.0 # <-- Oversized for a guaranteed clean
        ↪ cut
)
```

Listing 3.9: `create_opening_element()` in `geometry_mapper.py`

The cylinder is intentionally **3× the bolt length** — it starts one shank-length before the nominal origin and ends two shank-lengths after — to guarantee a clean boolean cut even when member thickness varies.

Applying the Cut to the Parent Element

`perform_boolean_cut()` wraps the solid in an `IfcOpeningElement` and links it to the target structural element:

```
opening_element = ifc_file.createIfcOpeningElement(
    GlobalId=..., Name="Bolt Hole",
    ObjectPlacement=local_placement,
    Representation=ProductDefinitionShape([opening_solid])
)
```

```
ifc_file.createIfcRelVoidsElement(
    RelatingBuildingElement=base_element, # e.g., IfcBeam or IfcPlate
    RelatedOpeningElement=opening_element
)
```

Listing 3.10: `perform_boolean_cut()` IFC relationship

The `IfcRelVoidsElement` relationship instructs every compliant BIM viewer (Revit, Tekla, BIMcollab) to subtract the cylinder from the parent solid during rendering. Critically, an `ObjectPlacement` is always provided, as several IFC parsers will silently discard `IfcOpeningElements` without one.

Execution Scope

Boolean cuts are applied **twice** inside `export_connection()`: once to every **member** (beam/column), and once to every **plate** (fin plate, end plate, base plate). Only objects whose `_class_name == 'Bolt'` generate cuts — nuts and washers do not. A `try/except` guard ensures that a single failed cut does not abort the entire export.

3.6.7 Step 7: Metadata, Property Sets, and File Save

After geometry is mapped and all boolean cuts are applied, the `MetadataMapper` attaches non-geometric BIM data to every element.

Property Sets (Psets)

`assign_standard_pset()` is the generic writer. It accepts any Python dictionary and creates correctly typed `IfcPropertySingleValue` entries: `IfcReal` for `float`, `IfcInteger` for `int`, `IfcBoolean` for `bool`, and `IfcLabel` for everything else.

The following named Psets are written:

Pset Name	Applied To	Properties Written
Pset_OsdagDesignData	All elements & assembly	MaterialGrade, SectionProfile, DesignCode (IS 800:2007), ExportTime
Pset_BeamCommon	Members	Material, Reference (section designation)
Pset_PlateCommon	Plates	Material
Pset_FastenerCommon	Bolts/Nuts	Type, Grade, YieldStrength (fy), UltimateStrength (fu)
Pset_WeldCommon	Welds	WeldType (Fillet / Groove), Grade

Quantity Take-Offs (QTOs)

assign_standard_qto() creates IfcElementQuantity sets. Each value is typed as IfcQuantityLength, IfcQuantityArea, IfcQuantityVolume, IfcQuantityWeight, or IfcQuantityCount.

- **Members** (Qto_BeamBaseQuantities): Length, Depth, Width, CrossSectionArea, NetVolume, NetWeight. The weight is computed as mass_per_m $\times$ $L/1000$. If the section is found in Intg_osdag.sqlite, database mass is used; otherwise, $\rho = 7850$ kg/m^3 is applied to the computed volume.
- **Plates** (Qto_PlateBaseQuantities): Length, Width, Depth (thickness), GrossArea, NetVolume, NetWeight. Special area formulas are applied for GassetPlate (trapezoid) and Stiffener_CAD (rectangle minus clipped corners).
- **Fasteners** (Qto_FastenerBaseQuantities): Diameter, Length, Count (always 1 per instance), NetWeight (cylindrical volume $\times 7850 \times 1.15$ to account for head, nut, and washer mass).
- **Welds** (Qto_WeldBaseQuantities): Length, Depth (weld size).

IfcElementAssembly and Spatial Containment

Once all elements have geometry and metadata, create_element_assembly() groups them:

```
assembly = ifc_file.createIfcElementAssembly(
    Name=f"Connection_{connection_id}",
    ObjectType="CONNECTION",
```

```
    PredefinedType="USERDEFINED"
)
ifc_file.createIfcRelAggregates(
    RelatingObject=assembly,
    RelatedObjects=ifc_elements # All members, plates, fasteners, welds
)
```

Listing 3.11: Assembly creation in `metadata_mapper.py`

The assembly is then anchored into the spatial hierarchy via `IfcRelContainedInSpatialStructure`, pointing to `IfcBuildingStorey` (Level 1, Elevation = 0.0). The connection-level metadata dictionary (material grade, design status, design forces) is attached as `Pset_OsdagDesignData` directly to this root assembly entity.

File Save

The very last call in the pipeline is:

```
self.ifc_file.write(self.filename)
```

This writes the complete in-memory `ifcopenshell.file` object to disk as a plain-text STEP Physical File (`.ifc`). The subprocess then exits with code `0`, which `output_dock.py` detects via `result.returncode` to display the success message box to the user.

3.7 Connection Types Supported

The exporter natively supports **28 distinct Osdag CAD classes**. These are grouped into seven functional categories inside the `DISPATCH_MAP` dictionary in `cad_extraction.py`. If a new connection module is built in Osdag, its CAD class **must** be registered here before it can be exported.

Connection Category	Supported CAD Classes
Simple Plated	`BoltedLapJointCAD`, `WeldedLapJointCAD`, `BoltedButtJointCAD`, `WeldedButtJointCAD`
Shear Connections	`FinPlateCAD`, `CleatAngleCAD`, `HeaderPlateCAD`, `SeatedAngleCAD`, `ColFlangeBeamWeb`, `ColWebBeamWeb`, `BeamWebBeamWeb`
Base Plates	`BasePlateCad`, `HollowBasePlateCad`

Connection Category	Supported CAD Classes
Moment End Plate	`CADGroove`, `CADFillet`, `CADcolwebGroove`, `CADColWebFillet`, `CCEndPlateCAD`
Moment Cover Plate (Bolted)	`CCSpliceCoverPlateBoltedCAD`, `BBCoverPlateBoltedCAD`
Moment Cover Plate (Welded)	`CCSpliceCoverPlateWeldedCAD`, `BBSpliceCoverPlateWeldedCAD`
Truss / Tension / Compression	`TensionAngleBoltCAD`, `TensionChannelBoltCAD`, `TensionAngleWeldCAD`, `TensionChannelWeldCAD`, `StrutAngleBoltCAD`, `StrutChannelBoltCAD`, `StrutAngleWeldCAD`, `StrutChannelWeldCAD`, `CompressionMemberCAD`

Table 3.4: All 28 supported Osdag connection classes and their extraction families.

3.8 How-To Guides for Future Developers

This chapter provides precise, step-by-step instructions for the most common extension tasks future interns will face. Each recipe lists the exact files to edit and the exact code patterns to follow.

3.8.1 How to Add Support for a New Connection Type

If the Osdag core team adds a new connection module (e.g., a "Tubular Truss Joint"), follow these steps:

1. Identify the name of the new CAD class (e.g., `TubularTrussCAD`). To confirm it, open the connection's Python file and look for `class TubularTrussCAD:`.
2. Open `cad_extraction.py` and check whether the new connection's attribute structure fits an existing extractor function:
   - If it stores bolts in `cad_obj.bolt_values.bolt_list_main`, it fits `extract_shear_connections()`.
   - If it has an anchor bolt pattern, it fits `extract_base_plate()`.
   - Otherwise, write a new extractor.
3. If a new extractor is needed:

```
def extract_tubular_truss(cad_obj):
    members, plates, bolts, welds, others = [], [], [], [], []
    members.append(cad_obj.chord_member)
    members.append(cad_obj.brace_member)
    welds.append(cad_obj.weld_joint)
    return members, plates, bolts, welds, others
```

4. Register the new class in `DISPATCH_MAP`:

```
DISPATCH_MAP = {
    ...
    'TubularTrussCAD': extract_tubular_truss,
}
```

5. Run a quick test via the CLI method (Method B from Section 2.2) with a pre-prepared JSON file to verify the 5 lists are populated correctly.

3.8.2 How to Add Support for a New Geometry Type

If Osdag introduces a new profile shape (e.g., `TSection` or `StarAngle`):

1. Open `geometry_mapper.py` and find `map_extruded_solid()`.
2. Locate the `elif obj_class == ...` chain and add a new branch.
3. Read the dimensions from `osdag_obj` and define the 2D cross-section as a closed polyline of (x, y) points:

```
elif obj_class == "TSection":
    B = float(osdag_obj.B) # Flange width
    D = float(osdag_obj.D) # Total depth
    tf = float(osdag_obj.tf) # Flange thickness
    tw = float(osdag_obj.tw) # Web thickness
    points_2d = [
        (-B/2, D/2), (B/2, D/2), (B/2, D/2-tf),
        (tw/2, D/2-tf),(tw/2, -D/2),(-tw/2,-D/2),
        (-tw/2, D/2-tf),(-B/2, D/2-tf)
    ]
    profile = self.ifc_file.createIfcArbitraryClosedProfileDef(
        ProfileType="AREA",
        OuterCurve=self.ifc_file.createIfcPolyline([...])
    )
    extrusion_depth = float(osdag_obj.L)
```

4. The existing `IfcExtrudedAreaSolid` return at the bottom of the function will handle it automatically.

3.8.3 How to Add a New Metadata Property

To attach a new field to the exported IFC model (e.g., the Fabrication Status or a connection design force):

1. Open `metadata_mapper.py` and find `assign_osdag_design_data()`.
2. Add the new value to the `props` dictionary:

```
props['FabricationStatus'] = 'Pending'
props['ShearCapacity_kN'] = 185.4
```

3. `assign_standard_pset()` automatically selects `IfcLabel` for strings and `IfcReal` for floats. No other changes are needed.

3.8.4 How to Switch to the IFC4 Schema

The exporter defaults to `IFC2X3` for maximum compatibility with older BIM software. To upgrade:

1. In `ifc_generator.py`, change the default argument:

```
def __init__(self, filename="Osdag_Model.ifc", schema="IFC4"):
```

2. The `IfcPerson` entity uses `Id` in IFC2X3 but `Identification` in IFC4. This is already handled in `setup_header()` by the `if self.schema == "IFC4":` branch.
3. The `assign_standard_qto()` method also contains a similar IFC4/IFC2X3 branch for `IfcQuantityLength` arguments.
4. (Optional) In IFC4, the `IfcFastener` welds can use the native `IfcFastenerTypeEnum.WELD` instead of the IFC2X3 workaround `ObjectType="WELD"`.

3.9 Known Issues and Gotchas

The following sections document cases where the exporter code uses hardcoded offsets, fallback logic, or workarounds. These are **not bugs** — they are deliberate engineering decisions to compensate for inconsistencies in how the Osdag CAD core positions objects in 3D space.

3.9.1 The `ColFlangeBeamWeb` (Header Plate) Coordinate Offset

In `geometry_mapper.py`, inside both `map_extruded_solid()` and `map_fastener()`, you will see:

```
if parent_conn == 'ColFlangeBeamWeb':
    if not getattr(osdag_obj, 'is_header_plate', False):
        origin[2] -= 35.0
        origin[0] += 4.0
```

Why: The Osdag CAD module for Column-Flange-Beam-Web connections places bolts and plates in the 3D viewer using an internal visual transformation matrix that is *not* reflected in the raw `sec_origin` values stored on the objects. Without this correction, all bolts and the fin plate appear displaced by approximately 35 mm in Z and 4 mm in X relative to the column face.

The `is_header_plate` flag isolates the Header Plate sub-variant: its elements are positioned differently and must **not** receive this shift.

3.9.2 The `HollowBasePlateCad` Vertical Position Fix

In `map_extruded_solid()`:

```
if parent_conn == 'HollowBasePlateCad':
    if obj_class in ['RectHollow', 'CircularHollow']:
        h_val = float(getattr(osdag_obj, 'H', 0.0))
        origin[2] -= h_val / 2.0
```

And in `map_weld()`:

```
if parent_conn == 'HollowBasePlateCad':
    origin[2] -= depth / 2.0
```

Why: In Osdag's native CAD for hollow section base plates, the column and its perimeter welds store their `sec_origin` at the *centroid* of the element (i.e., at height $H/2$ above the base plate surface). However, IFC extrusion strictly sweeps upward from the base of the profile. If the origin were used as-is, the column would float half its height above the base plate. The $H/2$ subtraction corrects this so the column sits flush on the plate.

3.9.3 The I-Section Sloped Flange Fallback

In `_generate_i_section_polyline()`:

```
if alpha < 80 or alpha > 100 or (R1 == 0 and R2 == 0):
    return [
        (t/2, D/2-T), (B/2, D/2-T), (B/2, D/2), (-B/2, D/2),
        (-B/2, D/2-T), (-t/2, D/2-T),
        (-t/2, -D/2+T), ...
    ]
```

Why: The full sloped-flange algorithm uses `math.tan()` and `math.sqrt()` to compute tangent points for root and toe fillets. Some Osdag CAD modules store `alpha=1` (a near-vertical slope, which is geometrically impossible for a real ISMB section) or pass zero fillet radii when generating simplified preview shapes. These inputs cause the trigonometric calculations to produce coordinates outside the section boundary, resulting in a degenerate `IfcPolyline`. The fallback returns a clean 12-point polygon that correctly represents the outer section envelope without fillets.

3.9.4 Ghost Fastener Filtering in Base Plates

In `extract_base_plate()` in `cad_extraction.py`:

```
for anchor in cad_obj.anchor_bolt_list:
    if np.linalg.norm(np.array(anchor.origin)) < 1e-6:
        continue # Skip template objects at world origin
    bolts.append(anchor)
```

Why: Osdag's `BasePlateCad` holds `AnchorBolt` objects in an array where the first element is often a "master template" with its origin at `[0, 0, 0]`. Including it would create a phantom anchor bolt at the absolute world origin inside the IFC file. The norm-check filters it out silently without breaking the export.

3.9.5 The Bolt Attribute Dual-Name Problem

Throughout `ifc_generator.py` you will see patterns like:

```
origin  = getattr(fastener, 'origin', None) or getattr(fastener, '
    ↪ sec_origin', None)
shaft_dir = getattr(fastener, 'shaftDir', None) or getattr(fastener, '
    ↪ uDir', None)
r       = getattr(fastener, 'r', None) # Shaft radius
h       = getattr(fastener, 'H', None) # Shank length (UPPERCASE)
```

Why: Different Osdag connection modules historically named bolt attributes differently. Some use `origin`, others use `sec_origin`. Some use `shaftDir`, others use `uDir`. The dual-`getattr` chain with `or` is a robust fallback that handles both naming conventions without requiring changes to the CAD core.

Note the case sensitivity: `r` (lowercase) is the **shaft radius** and `H` (uppercase) is the **total shank length**. Using `R` (uppercase) would retrieve the bolt **head radius**, producing an enormous hole. This is an easy mistake to make when extending the code.

3.9.6 Class Naming Conventions (Gusset vs. Gasset)

Throughout the exporter codebase and this report, the term `GassetPlate` is used to refer to gusset plates (e.g., in truss connections). While the standard structural engineering term is "Gusset Plate," the native Osdag CAD core internally defines this geometry class as `GassetPlate` (located in `Gasset_plate.py`).

The IFC exporter deliberately mirrors this spelling to maintain strict parity with the upstream CAD extraction objects, ensuring the dispatcher successfully matches the `_class_name` attribute at runtime. This is a purposeful architectural decision, not a typographical error within the export module.

3.10 Future Roadmap

The IFC pipeline is robust and production-ready for single-connection exports. The following enhancements are recommended for the next development phase, ordered by impact:

1. **Native Material Assignment (`IfcMaterial`)**
   Currently, material grade (e.g., E250, Fe410) is stored only as a string inside `Pset_OsdagDesignData`. Implementing `IfcMaterial` with `IfcRelAssociatesMaterial` will allow BIM viewers to apply material-specific rendering (e.g., grey steel, brown rust patina) and enable downstream structural analysis tools to extract material properties directly.

2. **Complex Weld Path Geometry**
   Welds are currently represented as straight rectangular or right-triangular extrusions. Many real welds, especially around hollow section base plates and moment connections, follow a continuous closed path. Replacing the current `IfcExtrudedAreaSolid` approach with `IfcSweptDiskSolid` along an `IfcPolyline` path would allow accurate wrap-around weld representation.

3. **Multi-Connection Frame Export**
   Osdag currently designs one connection at a time. If a future "Frame Module" provides access to multiple designed connections simultaneously, the exporter should be upgraded to accept an existing `.ifc` file and *append* new `IfcElementAssembly` groups to the existing `IfcBuildingStorey`, rather than creating a new `IfcProject` on every call.

4. **GUI Progress Feedback**
   The subprocess runs asynchronously, but the GUI provides no visual feedback during the wait period. Implementing a `QProgressDialog` or an animated spinner in

`output_dock.py` (driven by a `QTimer` polling `subprocess.poll()`) would greatly improve user experience on complex connections that take several seconds to export.

5. **Validation Report**
   After a successful export, the subprocess could write a brief JSON summary (element counts, total weight, detected geometry warnings) that `output_dock.py` reads and displays in a results panel. This would give engineers a quick sanity check without needing to open the IFC file in a viewer.

4. Internship Task 2: OsdagBridge IFC Export Module

4.1 Task 2: Problem Statement

OsdagBridge is a companion software to Osdag, designed specifically for the parametric modelling and structural design of plate girder bridge superstructures. While it successfully generates internal 3D CAD representations of complex bridge geometries—including skew alignments and various IRC 5:2015 safety components—it lacked the ability to export these models into a standard BIM format.

Without an IFC export capability, bridge engineers were unable to seamlessly transfer OsdagBridge models into civil engineering BIM environments, hindering collaborative infrastructure design, clash detection, and automated quantity take-offs for massive concrete and steel components.

The objective of this task was to develop an independent, metre-scale IFC export pipeline specifically tailored for OsdagBridge. This module needed to accurately handle large-scale geometric challenges, such as skew angle lateral shear transformations, and faithfully model complex concrete geometries like RCC crash barriers, medians, and deck slabs in accordance with Indian Roads Congress (IRC) specifications.

4.2 Task 2: Tasks Done

The following tasks were successfully executed to build the OsdagBridge IFC export module:

4.2.1 Bridge Geometry Extraction

Implemented a specialised extraction engine (`bridge_cad_extraction.py`) to process the `PlateGirder` design object. This involved dynamic layout solving to extract exact 3D coordinates for girder webs, flanges, intermediate stiffeners, end diaphragms, and cross-bracings (X-type and K-type).

4.2.2 Skew Geometry and Deck Modelling

Developed mathematical transformations to handle bridge skew angles accurately. Implemented lateral shear transformations (using the tangent of the skew angle) to correctly offset cross-bracings and diaphragm members. The deck slab was modelled as a skew-aware `IfcSlab` with exact width synchronisation based on IRC 5:2015 specifications.

4.2.3 IRC 5:2015 Safety Components

Programmed parametric cross-section profiles using `IfcArbitraryClosedProfileDef` and `IfcArbitraryProfileDefWithVoids` to accurately represent complex concrete safety elements:

- **Crash Barriers**: RCC polygonal profiles and metallic W-beam rails with C-channel posts.
- **Medians**: Raised kerbs, RCC crash barriers, and metallic medians.
- **Railings**: Hollow RCC railings and steel post-and-rail assemblies.

4.2.4 Scale and Material Rendering

Configured the IFC4 spatial hierarchy (e.g., `IfcFacility`) with a global metre-scale factor (`IfcSIUnit`), a critical departure from the millimetre scale used in standard Osdag. Explicit RGB surface styles were implemented to visually differentiate steel components (grey) from reinforced concrete elements (light grey/white).

4.2.5 Code Submission and Integration

The completed OsdagBridge module was subjected to integration testing across multiple skew configurations and median types. A Pull Request was raised, reviewed by the mentorship team, and successfully merged into the primary repository.

4.3 OsdagBridge IFC Module Overview

This section documents the IFC export pipeline developed for the **OsdagBridge** application, a companion tool to Osdag that designs plate girder bridge superstructures. The module is located at:

`osdagbridge/core/ifc_export_bridge/`

While the bridge exporter was inspired by the Osdag connection exporter documented in Task 1, it is an **independent implementation** built from scratch to handle the

unique requirements of bridge modelling — larger spans, skew geometry, IRC 5:2015 safety components, and metre-scale units.

4.3.1 Scope and Purpose

The Osdag (connection) exporter maps a single bolted/welded joint into IFC. The bridge exporter maps an **entire plate girder bridge superstructure**, including:

- **Plate Girders** — Web, top flange, and bottom flange as separate `IfcPlate` entities per girder.
- **Stiffeners** — Intermediate (vertical), end (bearing), and longitudinal stiffeners.
- **Cross Bracings** — X-type and K-type bracing frames with diagonal members, top chords, and bottom chords, using six cross-section profiles (Angle, Channel, Double Angle, Double Channel, I-Section, Rectangle).
- **Deck Slab** — Skew-aware RCC slab with variable width.
- **Safety Components** — Crash barriers (RCC and Metallic), medians (Raised Kerb, RCC Crash Barrier, Metallic W-Beam), and railings (RCC with passage voids, Steel post-and-rail) per **IRC 5:2015**.
- **Bearing Supports** — Triangular and cylindrical bearing approximations.

The governing standard for cross-section profiles of crash barriers, medians, and railings is **IRC 5:2015 Clause 109.6.3**. Profile dimensions are sourced from the `irc5_geometry.py` module (`CrashBarrierGeometry`, `MedianGeometry`, `RailingGeometry` classes).

4.3.2 Key Architectural Differences from Osdag

The bridge exporter differs from the connection exporter in several fundamental design decisions. These differences are summarised in Table 4.1.

Aspect	Osdag (Connections)	OsdagBridge
IFC Schema	IFC2X3 (default)	**IFC4** (hardcoded in `bridge_ifc_generator.py`, line 27)

Aspect	Osdag (Connections)	OsdagBridge
Working Units	Millimetres throughout	**Metres** — a global scale factor `s = 0.001` converts all mm values at the point of IFC entity creation (`bridge_ifc_generator.py`, line 70)
Process Model	Subprocess isolation (`subprocess.run()`) with JSON serialisation to avoid `ifcopenshell` import conflicts	**In-process threading** (`threading.Thread`) — `export_ifc_handler.py` line 40
Object Reconstruction	`DictToObj` class with `__class__` property override to fake native class names	**`ExtractedObject`** class — a lightweight mock that stores `_class_name` and arbitrary keyword attributes directly (`bridge_cad_extraction.py`, lines 32–37)
Geometry Strategy	Reads pre-computed `sec_origin`, `uDir`, `wDir` from Osdag CAD objects	**Parametric reconstruction** — all coordinates are calculated from scratch using span, spacing, and section dimensions. No OpenCASCADE B-Rep data is used.
Boolean Cuts	High LOD bolt holes via `IfcOpeningElement` + `IfcRelVoidsElement`	**Not applicable** — bridge models have no fastener-level bolt holes
Colour Support	None — BIM viewer defaults	**Explicit colours** via `IfcSurfaceStyleRendering`: Steel = RGB(0.6, 0.65, 0.7), RCC = RGB(0.85, 0.85, 0.82) (`bridge_ifc_generator.py`, lines 339–340)
Spatial Hierarchy	`IfcBuildingStorey` named "Level 1"	`IfcBuildingStorey` named "**Superstructure**" to reflect bridge engineering terminology

Aspect	Osdag (Connections)	OsdagBridge
Skew Geometry	Not supported	Full support via `_calculate_skew_offset()` which applies $\Delta x = y \cdot \tan(\theta)$ to every lateral position

Table 4.1: Architectural comparison between the Osdag (connections) and OsdagBridge IFC exporters.

4.3.3 Module File Structure

The module consists of six Python files. Each file has a single, well-defined responsibility:

File	Lines	Responsibility
`__init__.py`	6	Public API. Exports the `PlateGirderIfcExportHandler` class as the sole entry point for external consumers.
`export_ifc_handler.py`	43	Entry point. Instantiates the extractor and generator, wires the `extract()` → `generate_from_extracted_data()` pipeline, and provides `export_async()` for non-blocking GUI integration.
`bridge_cad_extraction.py`	480	CAD Extraction. The `PlateGirderIFCExtractor` class reads the `cad_generator` object (which holds all UI design parameters) and produces a dictionary of six component lists. This is the only file that imports from Osdag's native geometry libraries (`IRC5_2015`, `CrashBarrierGeometry`, etc.).

File	Lines	Responsibility
`bridge_geometry_mapper.py`	226	Geometry Translation. The `BridgeGeometryMapper` class provides profile creation methods (Rectangle, L-Shape, C-Shape, I-Shape, Double Angle, Double Channel, Polygon, Polygon with Voids) and the `IfcExtrudedAreaSolid` / `IfcFacetedBrep` generators.
`bridge_ifc_generator.py`	620	IFC Orchestration. The `BridgeIfcGenerator` class initialises the IFC4 file, creates the spatial hierarchy, and iterates through all extracted components with type-specific processor functions (`_process_plate`, `_process_brace`, `_process_deck`, `_process_barrier`, `_process_railing`).
`metadata_mapper.py`	280	BIM Metadata. The `BridgeMetadataMapper` class attaches `Pset_OsdagBridgeProperties` to every IFC element. Includes database-backed material property lookups from `Intg_osdag.sqlite`.

Table 4.2: File structure of the OsdagBridge IFC export module.

4.3.4 The `ExtractedObject` Pattern

Unlike the Osdag connection exporter (which reads attributes from live `OCC` objects), the bridge exporter **parametrically reconstructs** every geometric entity from the UI input values. The vehicle for this is the `ExtractedObject` class:

```
class ExtractedObject:
    """A generic mock object to hold geometric parameters
    for the GeometryMapper."""
    def __init__(self, obj_class, **kwargs):
        self._class_name = obj_class
        for k, v in kwargs.items():
            setattr(self, k, v)
```

Listing 4.1: `ExtractedObject` in `bridge_cad_extraction.py` (lines 32–37)

Every extraction method creates instances of this class with the exact attributes needed by the downstream geometry mapper. For example, a girder web plate is created as:

```
ExtractedObject(
    "Plate", T=tw, L=d, W=L,
    origin=[x_offset, y_offset, 0],
    uDir=[1,0,0], wDir=[0,1,0],
    ifc_name=f"Girder Web {i+1}"
)
```

Listing 4.2: Girder web extraction (`bridge_cad_extraction.py`, lines 155–158)

This approach eliminates the need for the `DictToObj` hack used by the Osdag exporter, because the bridge module never crosses a process boundary — all objects are created and consumed within the same Python process.

4.3.5 Dependencies

Package / Module	Source	Used For
`ifcopenshell`	conda-forge	IFC4 file creation, entity construction, GUID compression, and file serialisation
`numpy`	pip/conda	Vector arithmetic for skew offset computation and cross-product orthogonalisation
`math (stdlib)`	Python	`tan()`, `radians()`, `sqrt()`, `exp()` for skew offsets, bracing vectors, and W-beam Gaussian curves
`sqlite3 (stdlib)`	Python	Querying `Intg_osdag.sqlite` for steel yield strength, UTS, modulus of elasticity, Poisson's ratio, density; and concrete `fck`, `fctm`, `Ecm`
`threading (stdlib)`	Python	`export_async()` wraps the export pipeline in a daemon thread to prevent GUI freeze

Package / Module	Source	Used For
`irc5_geometry.py`	OsdagBridge	`CrashBarrierGeometry`, `MedianGeometry`, `RailingGeometry` — static lookup classes that return IRC 5:2015 profile dimension dictionaries
`IRC5_2015`	OsdagBridge	`cl_109_6_3_shapes()` — computes the exact barrier base width from the design code for deck width calculation

Table 4.3: Dependencies of the OsdagBridge IFC export module.

4.3.6 IFC Entity Mapping Summary

Each extracted component type is mapped to a specific IFC entity class in `bridge_ifc_generator.py`. This mapping is summarised below:

Extracted Class	IFC Entity	Description
`Plate`	`IfcPlate`	Girder webs, top/bottom flanges, stiffener plates
`StructuralMember`	`IfcMember`	Cross bracing diagonals, top/bottom chords
`SlabPolygon`	`IfcSlab`	Deck slab (skew-aware polygon extrusion)
`BarrierSweep (RCC)`	`IfcWall`	RCC crash barriers and RCC medians
`BarrierSweep (Metal)`	`IfcBuildingElementProxy`	Metallic crash barriers (multi-solid assembly)
`RailingSweep (RCC)`	`IfcRailing`	RCC railings with hollow passage voids
`RailingSweep (Steel)`	`IfcBuildingElementProxy`	Steel post-and-rail assemblies
`StiffenerPlate`	`IfcPlate`	Triangular bearing supports
`CircularSolid`	`IfcBuildingElementProxy`	Cylindrical roller bearing supports

4.4 Bridge Export Pipeline

This appendix provides a detailed step-by-step breakdown of the bridge IFC export pipeline, detailing how geometric and metadata extraction flows into the final IFC4 file generation.

4.4.1 Stage 1: Entry Point and Orchestration

The export process begins in `export_ifc_handler.py`, where the `PlateGirderIfcExportHandler` connects the OsdagBridge GUI export button to the backend engine.

To ensure the Qt UI remains responsive during complex geometry calculations, the export is triggered asynchronously:

```
def export_async(self):
    """Triggers export in a background thread to keep the Qt UI
        ↪ responsive."""
    thread = threading.Thread(target=self.export)
    thread.daemon = True
    thread.start()
```

The synchronous `export()` method orchestrates the two primary stages:

1. Calls `PlateGirderIFCExtractor.extract()` to generate the intermediate mock objects.
2. Calls `BridgeIfcGenerator.generate_from_extracted_data()` to consume those mock objects and build the IFC file.

Any exceptions encountered during the process are written to `ifc_export_error.txt` to assist in debugging.

4.4.2 Stage 2: CAD Extraction

Located in `bridge_cad_extraction.py`, the extraction stage reads the `cad_generator` (which contains all structural UI inputs) and produces a structured dictionary of bridge components.

The output dictionary contains six categories:

- `girders`: Main plate girders (Webs, Top Flanges, Bottom Flanges).
- `stiffeners`: Intermediate, End, and Longitudinal stiffener plates.
- `cross_bracings`: X-type and K-type bracing with diagonal and chord members.

- `deck_slab`: The RCC deck slab spanning the entire bridge.
- `crash_barriers`: RCC or Metallic safety boundaries at the edges and center (medians).
- `supports`: Triangular bearing and cylindrical roller supports.

Dynamic Girder Layout Solver

Rather than relying on fixed CAD origins, the bridge extractor dynamically solves the structural layout. The `_solve_girder_layout()` method calculates the exact number of girders (`n`), the internal `spacing`, and the edge `overhang` required to support the total bridge deck width.

```
def _solve_girder_layout(self, total_width):
    target_s = getattr(self.cad, 'girder_spacing', 2000)
    target_o = getattr(self.cad, 'deck_overhang', target_s / 2.0)

    # Calculate required number of girders
    n = max(2, int(round(total_width / target_s)))

    if n > 1:
        actual_s = (total_width - 2 * target_o) / (n - 1)
        actual_o = target_o
    else:
        actual_s, actual_o = 0, total_width / 2.0

    return n, actual_s, actual_o
```

Handling Skew Geometry

One of the most complex aspects of bridge modelling is dealing with skewed abutments and piers. The extractor handles this using a lateral shear transformation via `_calculate_skew_offset()`:

```
def _calculate_skew_offset(self, lateral_position, reference_position=0):
    ↪
    if self.cad.skew_angle == 0:
        return 0.0
    skew_rad = math.radians(self.cad.skew_angle)
    return (lateral_position - reference_position) * math.tan(skew_rad)
```

This offset is added to the `x_offset` (longitudinal position) based on the element's `y_offset` (lateral position), cleanly sweeping the entire structure without needing complex

3D rotations for every individual profile. This ensures stiffeners and cross-bracings remain perfectly aligned with the skewed abutment lines.

4.4.3 Stage 3: Geometry Mapping

Located in `bridge_geometry_mapper.py`, this module handles the creation of `ifcopenshell` geometries from the parametrically defined `ExtractedObject` instances.

Profile Creation

The mapper supports a variety of basic and composite cross-sections to handle standard Indian structural steel profiles:

- **Basic Profiles**: `IfcRectangleProfileDef` (plates), `IfcLShapeProfileDef` (angles), `IfcCShapeProfileDef` (channels), `IfcIShapeProfileDef` (I-sections).
- **Composite Profiles**: Custom algorithms build `IfcCompositeProfileDef` entities for `DoubleAngle` and `DoubleChannel` cross-bracing members. This involves creating two base profiles, shifting them symmetrically apart by a specified gusset plate thickness, and merging them into a single logical profile.
- **Arbitrary Profiles**: RCC elements like crash barriers and custom skewed decks use `IfcArbitraryClosedProfileDef` defined by a series of 2D Cartesian points mapped directly from IRC 5:2015 tables. RCC railings use `IfcArbitraryProfileDefWithVoids` to model the hollow rectangular passages through the concrete.

Faceted B-Rep Construction

To accurately represent a skewed deck slab (a parallelogram in plan view) without relying on directional sweeps which sometimes render incorrectly or twist in certain BIM viewers, the mapper includes a custom boundary representation (B-Rep) builder:

```
def create_faceted_brep_from_3d_deck(self, top_points_3d, thickness):
    """Constructs an IfcFacetedBrep directly from 3D shear-mapped
        ↪ coordinates."""
    bot_points_3d = [(p[0], p[1], p[2] - thickness) for p in
        ↪ top_points_3d]

    # Generate IfcPolyLoop for Top and Bottom faces
    top_loop = self.file.createIfcPolyLoop([self._create_cartesian_point(
        ↪ p) for p in top_points_3d])
    bot_loop = self.file.createIfcPolyLoop([self._create_cartesian_point(
        ↪ p) for p in reversed(bot_points_3d)])
```

```
8
9       # ... Create IfcFace for Top, Bottom, and 4 Sides ...
10
11      shell = self.file.createIfcClosedShell(ifc_faces)
12      return self.file.createIfcFacetedBrep(shell)
```

Material Styling

Unlike the connections exporter, the bridge exporter applies explicit RGB rendering styles using `IfcSurfaceStyleRendering`. This is crucial for large models where visual differentiation is required:

- **Steel Components**: `(0.6, 0.65, 0.7)` – A semi-matte cool grey.
- **RCC (Concrete) Components**: `(0.85, 0.85, 0.82)` – A warm, light beige/grey.

4.4.4 Stage 4: IFC Generation

Located in `bridge_ifc_generator.py`, this file manages the spatial hierarchy and delegates component creation to specific builder functions.

Spatial Hierarchy Initialization

The bridge model is instantiated with an `IFC4` schema. The baseline spatial hierarchy mirrors typical infrastructure projects: `IfcProject` → `IfcSite` → `IfcBuilding` → `IfcBuildingStorey` (named "Superstructure").

All length dimensions are converted from Osdag's native millimetres to standard IFC **metres** using a global scale factor: `s = 0.001`.

Component Assembly

The generator iterates through the extracted components, invoking specific processor methods:

- `_process_plate()`: Maps girder webs, flanges, and stiffeners to `IfcPlate`. It reads the `uDir` and `wDir` from the extracted object to properly orient the 3D `IfcAxis2Placement3D`.
- `_process_brace()`: Calculates accurate 3D extrusion vectors (`z_dir`) and orthogonal local axes (`x_dir`) via cross products to orient `IfcMember` braces correctly in space.

- `_process_deck()`: Transforms the 3D skew-mapped vertices into a pure 2D footprint profile (`IfcSlab`) placed at the exact Z-elevation of the top flanges.

Multi-Component Assembly: Metallic Medians

Metallic safety barriers require a complex multi-solid assembly comprising RCC kerbs, steel posts, spacers, and continuous W-beam rails. The generator builds this systematically in `_process_metallic_median`:

1. **Kerb Base**: A polygonal extrusion modeling the raised base.
2. **Steel Posts & Spacers**: C-channel profiles instantiated in a loop across the span length (`num_posts = int(span_l / post_spacing) + 1`).
3. **Continuous W-Beam Rails**: The complex "W" shape of the rail is generated parametrically using a Gaussian curve algorithm to ensure a smooth, realistic profile rather than a jagged approximation:

```
def _get_w_profile_pts_m(mult, s):
    # Generates a realistic double-hump W-Beam profile using Gaussian
        ↪ equations
    pts = []
    for i in range(40):
        y = -156.0 + i * (312.0 / 39.0)
        amp = D / (1.0 + math.exp(-((mu1-mu2)**2)/(2*sigma**2)))
        # Map normal distribution into cross-section profile
        ...
        pts.append((x * s, y * s))
    return pts
```

The assembled components (posts, spacers, rails) are then aggregated into a single `IfcBuildingElementProxy` using an `IfcShapeRepresentation` containing multiple `IfcExtrudedAreaSolid` items.

4.4.5 Stage 5: Metadata Mapping

Located in `metadata_mapper.py`, this module handles assigning rich engineering properties to the structural elements.

Database Integration

Instead of relying solely on the UI inputs, the mapper connects directly to the internal Osdag SQLite database (`Intg_osdag.sqlite`) to extract deep material properties based on the selected material grade.

For **Steel** (e.g., E250A):

- `YieldStrength_MPa`, `UltimateTensileStrength_MPa`
- `ModulusOfElasticity_GPa`, `PoissonsRatio`, `Density_N_m3`

For **Concrete** (e.g., M30):

- f_{ck} (`fck_MPa` - Characteristic Compressive Strength)
- f_{ctm} (`fctm_MPa` - Mean Tensile Strength)
- E_{cm} (`Ecm_GPa` - Secant Modulus of Elasticity)

Component-Specific Property Mapping

The `BridgeMetadataMapper` contains specialized methods for different elements. For example, `map_girder` differentiates between the structural role of plates making up the main girder:

```
def map_girder(self, element, cad, ifc_name):
    props = {
        "Material": cad.steel_grade,
        "SpanLength": self._to_meters(cad.span_length_L),
        "GirderSpacing": self._to_meters(cad.girder_spacing)
    }

    if "Web" in ifc_name:
        props["ComponentRole"] = "Girder Web"
        props["Depth"] = self._to_meters(cad.girder_section_d)
        props["Thickness"] = self._to_meters(cad.girder_section_tw)
    elif "TopFlange" in ifc_name:
        props["ComponentRole"] = "Girder Top Flange"
        # Adds Width and Thickness
...
```

Similarly, `map_brace` decodes the structural system (e.g., "End Diaphragm" vs "Intermediate Bracing") and the role (Top Chord, Bottom Chord, Diagonal). It also dynamically retrieves properties from the cross-section dictionary (like `leg_h`, `flange_width`, `web_thickness`) based on whether the brace is an Angle, Channel, or I-Section.

All properties are packaged into an `IfcPropertySet` named `Pset_OsdagBridgeProperties` and linked to the target element using `IfcRelDefinesByProperties`, fulfilling Level of Development (LOD) requirements for BIM deliverables.

4.5 Bridge Component Extraction and Assembly Details

This section provides detailed documentation of the individual extraction and assembly algorithms used by the OsdagBridge IFC export module. These cover the structural geometry calculations, IRC standard lookups, and component positioning logic that underpin the pipeline described above.

4.5.1 Deck Width Calculation

A critical prerequisite for the entire export pipeline is determining the total deck width accurately. The `_calculate_total_deck_width()` method in `bridge_cad_extraction.py` replicates Osdag's native `CrossSectionLayout` logic. The formula accounts for all roadway components:

```
def _calculate_total_deck_width(self, actual_base_width,
    ↪ actual_railing_width):
    cw = self.cad.carriageway_width
    cb = actual_base_width # Crash barrier width from IRC 5
    fp = self.cad.footpath_width
    rl = actual_railing_width # Railing width (375mm standard)

    # Base road width depends on median presence
    if self.cad.enable_median:
        median_geo = MedianGeometry.get_geometry("IRC 5 - Raised Kerb")
        mw = median_geo.get("median_width", 1200)
        road_width = (2 * cw) + mw
    else:
        road_width = cw

    # Total = road + 2x crash barriers + footpaths/railings
    total = road_width + (2 * cb)

    if self.cad.footpath_config == "BOTH":
        total += 2 * (fp + rl)
    elif self.cad.footpath_config in ("LEFT", "RIGHT"):
        total += fp + rl

    return total
```

Listing 4.3: Deck width calculation (simplified)

This width directly drives the girder layout solver, deck slab polygon dimensions, and all lateral positioning calculations for safety components.

4.5.2 IRC 5:2015 Design Dictionary Builder

Before extraction begins, the module constructs an IRC 5:2015 design dictionary via the `_build_design_dict()` method. This method maps the user's GUI selections (barrier type, subtype, footpath configuration, railing type) to the correct IRC clause lookup:

- **Barrier Type Mapping**: The GUI's "Flexible", "Semi-Rigid", and "Rigid" labels are converted to indexed keys that the `IRC5_2015.cl_109_6_3_shapes()` function accepts.
- **Rigid Subtypes**: "IRC-5R" and "High Containment" are mapped to their respective profile indices.
- **Metallic Subtypes**: "Single W-beam" and "Double W-beam" are mapped separately, producing significantly different post spacing and rail counts.
- **Railing Logic**: If the barrier type is not "Rigid", the railing type is forced to "RCC" regardless of the user's selection — this matches a constraint from the Osdag desktop CAD builder.

The returned design dictionary provides the exact `crash_barrier_width` (or `kerb_bottom_width` for metallic types), which feeds directly into the deck width calculation.

4.5.3 Stiffener Extraction Details

The stiffener extraction logic in `_extract_stiffeners()` handles three distinct stiffener types, each with unique placement algorithms:

Intermediate Stiffeners

These vertical stiffeners are placed symmetrically on both sides of each girder web at regular intervals defined by `intermediate_stiffener_spacing`.

Crucially, the algorithm enforces an **exclusion zone** near the bridge ends to avoid overlap with the end stiffeners:

```
end_zone = end_stiffener_gap + (num_end_stiffener_pairs - 1) *
    ↪ END_STIFFENER_SPACING + END_STIFFENER_SPACING

for j in range(1, num_panels):
    x_dist = j * spacing_stiff
    if x_dist <= end_zone or x_dist >= (L - end_zone):
        continue # Skip positions inside the end stiffener zone
```

End Stiffeners

End stiffeners are placed in pairs at both abutment ends of each girder. Multiple pairs can be specified via `num_end_stiffener_pairs`, and each subsequent pair is offset by the constant `END_STIFFENER_SPACING` imported from the Osdag plate girder builder module.

Longitudinal Stiffeners

When enabled, longitudinal stiffeners run the full effective span length of the girder (minus the end stiffener thickness on each side). They are placed at specific depth fractions — `D/3` from the top for a single stiffener, and `D/3` and `2D/3` for double stiffeners — on both sides of each girder web. Unlike the vertical stiffeners, longitudinal stiffeners extrude along the **global X axis** (the span direction), requiring a different axis orientation (`uDir = [1,0,0]`).

4.5.4 Cross-Bracing Type Dispatch

The cross-bracing extraction system in `_extract_cross_bracings()` is one of the most complex parts of the extraction engine, handling multiple bracing configurations and structural subsystems.

X-Type vs K-Type Bracing

The internal `build_bracing()` function generates different member topologies based on the `bracing_type`:

- **X-Type**: Two diagonal members crossing the full depth between adjacent girders, optionally with top and/or bottom chord members controlled by the `x_bracket_option` parameter ("LOWER", "UPPER", or "BOTH").
- **K-Type**: Two diagonal members converging at the midpoint of the bottom chord. The bottom chord is always included, while the top chord is controlled by `k_top_bracket`.

Each diagonal member stores its start (`p1`) and end (`p2`) coordinates as 3D points rather than parametric profiles, allowing the IFC generator to compute accurate extrusion vectors via cross-product operations.

End Diaphragm Offset Calculation

End diaphragms are placed at the abutment ends but are offset inward to avoid clipping with the bridge support:

```
base_offset = self.cad.end_diaphragm_spacing
extra_offset = 300.0 * math.tan(math.radians(abs(self.cad.skew_angle)))
offset = base_offset + extra_offset
```

The `extra_offset` term is a correction factor that increases with the skew angle, ensuring end diaphragms clear the skewed abutment plane. End diaphragms can also be configured as "Rolled Beam" or "Welded Beam" types (instead of cross bracing), in which case they are placed as a single horizontal `IfcMember` at the correct depth.

Six Cross-Section Profile Types

Each bracing member can independently be specified as one of six cross-section types: Angle, Channel, I-Section, Double Angle, Double Channel, or Rectangle. The profile type is stored in `sec_type` and the dimensions in a `dims` dictionary, allowing the IFC generator's `_process_brace()` method to dynamically select the correct `IfcProfileDef` constructor.

4.5.5 Safety Component Positioning and Assembly

The safety component extraction in `_extract_safety_components()` involves precise lateral positioning of crash barriers, medians, and railings based on the total deck geometry and footpath configuration.

Crash Barrier Y-Position Logic

Barrier centres are calculated from the deck edges inward:

```
y_l = -total_deck_width / 2.0 + cb / 2.0
if self.cad.footpath_config in ("LEFT", "BOTH"):
    y_l += (self.cad.footpath_width + actual_railing_width)

y_r = total_deck_width / 2.0 - cb / 2.0
if self.cad.footpath_config in ("RIGHT", "BOTH"):
    y_r -= (self.cad.footpath_width + actual_railing_width)
```

This ensures barriers are placed at the correct edge of the carriageway, not at the absolute deck edge when footpaths are present.

Barrier Label Mapping for IRC Geometry Lookup

The module supports both the new full IRC label format (e.g., "IRC 5 - RCC Crash Barrier") used by the updated UI, and the legacy short format ("Rigid", "Semi-Rigid")

from older versions. The mapping logic in the extractor normalises these labels before querying the `CrashBarrierGeometry` and `MedianGeometry` lookup classes.

RCC Crash Barrier Profile Construction

For RCC barriers, the IFC generator constructs a seven-point polygonal profile matching the exact IRC 5 dimensions:

- A vertical base section (`base_vertical` = 100 mm).
- A sloped transition zone (250 mm high) where the profile narrows by 100 mm on the traffic-facing side.
- A vertical top section tapering to the final `top_width` (typically 175 mm).

For medians configured as "RCC Crash Barrier", the generator creates **two mirrored barriers** separated by the median width, with one profile mirrored via $(bottom\_w - x, y)$ coordinate transformation.

Steel Post-and-Rail Railing Assembly

When the railing type is "Steel", the `_process_metallic_railing()` function builds a multi-component assembly:

1. A continuous rectangular RCC base (100 mm × 375 mm) extruded along the span.
2. Square steel posts (150 mm × 150 mm) at regular intervals, with the spacing calculated to ensure posts land exactly at both bridge ends.
3. Two continuous horizontal rails (40 mm × 40 mm) at the top and mid-height positions.

RCC Railing with Hollow Passages

RCC railings use `IfcArbitraryProfileDefWithVoids` to model three rectangular passages (hollow openings) through the concrete wall. The void dimensions are computed as 60% of the railing width and 50% of the individual bay height, evenly spaced across the body section above the 100 mm solid base.

Bearing Support Geometry

At each abutment, the extractor generates two types of bearing approximations per girder:

- **Fixed/Hinged (Left End)**: A triangular wedge modelled as an `IfcPlate` using the stiffener plate profile. The triangle height is computed as `base_dim / 1.5` where `base_dim = min(0.10L, 0.75D)`.

- **Roller (Right End)**: A cylinder modelled as an `IfcBuildingElementProxy` using the `CircularSolid` extracted class. The radius equals half the support height.

Both support types are placed at the exact bottom-of-flange Z elevation (`z_contact = -(D/2 + T_fb)`), and are laterally offset by the same skew correction applied to all other components.

4.6 Bridge-Specific Known Issues & Future Work

While the OsdagBridge IFC export module reliably generates detailed High LOD superstructure models, there are several known geometric and metadata limitations in the current implementation. This appendix outlines those issues and presents the roadmap for future development.

4.6.1 Known Issues and Limitations

1. Severe Skew Angle Distortions ($> 30°$)

The lateral shear transformation method (`_calculate_skew_offset`) used to model skewed bridges works perfectly for standard skews between 0° and 30°. However, for extreme skews ($> 30°$), the linear shear causes intermediate cross-bracings to warp or overlap with adjacent stiffeners. In reality, highly skewed bridges use staggered orthogonal bracings rather than sheared parallel bracings.
Workaround: Users modelling extreme skews must currently accept minor geometric clipping at the bracing connection nodes.

2. Hardcoded Metallic Barrier Components

While the overall barrier selection is driven by the GUI, the internal dimensions of metallic barrier components (W-beam post sizes, channel web thickness, spacer block depth, and the 1000 mm longitudinal post spacing) are hardcoded in `bridge_ifc_generator.py` to match IRC 5:2015 median defaults. If a user requires a custom barrier standard (e.g., AASHTO or a non-standard local requirement), the IFC exporter will still output the IRC 5 defaults.
Workaround: Direct modification of `bridge_ifc_generator.py` is required to change spacer and post profiles.

3. Support Geometry Simplification

The bearing supports at the abutment locations are exported as visual approximations rather than exact manufacturer profiles:

- Fixed/Hinged bearings are modelled as simple `IfcPlate` triangular wedges.
- Roller bearings are modelled as solid `IfcBuildingElementProxy` cylinders.

They do not currently include elastomer pads, sliding plates, or anchor bolts.

4.6.2 Future Development Roadmap

1. Substructure Export Integration

Currently, the module only exports the bridge **superstructure** (girders, deck, and safety components). Future versions will integrate with the forthcoming OsdagBridge Substructure module to export:

- Pier caps and pier shafts (circular and rectangular).
- Abutment walls and wing walls.
- Foundation piles and pile caps.

2. True `IfcMaterial` Associations

Presently, material grades (e.g., "E250A", "M30") and their physical properties (Density, Yield Strength, Poisson's Ratio) are mapped as metadata text fields within `Pset_OsdagBridgeProperties`. The roadmap includes upgrading this to native IFC4 `IfcMaterial`, `IfcMaterialList`, and `IfcRelAssociatesMaterial` entities. This will allow advanced 4D/5D BIM software to automatically calculate cost estimates and structural weights directly from the material tree rather than parsing custom property sets.

3. Wearing Coat / Asphalt Overlay

The current exporter assumes the vehicles drive directly on the structural RCC deck slab. In reality, an asphalt wearing coat (typically 40–75 mm thick) is placed over the deck. Future iterations will export an additional `IfcCovering` element resting on top of the `IfcSlab` to represent the wearing surface, complete with its own material properties.

4. Load Path Visualization

To assist in design review, future updates may leverage `IfcStructuralAnalysis-Model` to visually represent live load paths (e.g., Class 70R vehicle wheel loads) and reaction forces directly within the BIM environment as geometric vectors.

5. Conclusions

5.1 Tasks Accomplished

The internship resulted in the successful design, development, and integration of two distinct IFC export pipelines as part of the Osdag and OsdagBridge open-source projects.

Osdag IFC Export Module (`osdag_core/export_ifc/`):

- Built a complete six-file Python package (`export_handler.py`, `geometry_mapper.py`, `ifc_generator.py`, `metadata_mapper.py`, `material_handler.py`, `property_set_builder.py`) from scratch and integrated it into the main Osdag codebase.
- Implemented IFC export support for **28 distinct structural connection types**, including shear connections (Fin Plate, Cleat Angle, Seated Angle, End Plate), moment connections (Extended End Plate, Cover Plate Splice), truss joints, base plates, and welded connections.
- Achieved **High Level of Detail (High LOD)** geometric fidelity by accurately modelling parametric structural profiles (I-sections, channels, angles, plates), fastener hardware (bolts, nuts, washers, anchor bolts), weld geometry, and IFC boolean subtraction operations to physically cut bolt clearance holes in accordance with **IS 800:2007 Table 19**.
- Mapped all structural elements to appropriate IFC4 entities (`IfcMember`, `IfcPlate`, `IfcMechanicalFastener`) with full `Pset_OsdagConnectionProperties` property sets backed by the `Intg_osdag.sqlite` database.
- Integrated an asynchronous export trigger using `threading.Thread` to keep the PyQt GUI responsive during export.

OsdagBridge IFC Export Module (`osdagbridge/core/ifc_export_bridge/`):

- Designed and implemented a five-stage bridge IFC export pipeline: Handler → CAD Extraction → Geometry Mapping → IFC Generation → Metadata Mapping.
- Built a parametric girder layout solver (`_solve_girder_layout`) that dynamically computes the number of girders, spacing, and overhangs from a given bridge

deck width, avoiding any hardcoded origins.

- Implemented a lateral shear transformation algorithm (`_calculate_skew_offset`) to correctly handle skewed bridge abutments for all structural components including stiffeners and cross-bracings.
- Reconstructed complex structural profiles from scratch using IRC 5:2015 specifications, including arbitrary polygonal RCC crash barrier profiles, metallic W-beam railings (Gaussian curve parametrization), composite double-angle and double-channel cross-bracing sections, and hollow RCC railing profiles using `IfcArbitraryProfileDefWithVoids`.
- Implemented a direct Faceted B-Rep builder (`create_faceted_brep_from_3d_deck`) for skewed deck slabs to avoid directional sweep artifacts in IFC viewers.
- Integrated material property lookups via SQLite to populate `Pset_OsdagBridgeProperties` with steel grade mechanical properties (Yield Strength, UTS, Modulus) and concrete grade properties (f_{ck}, f_{ctm}, E_{cm}).

5.2 Skills Developed

The internship provided substantial growth across both software engineering and structural engineering domains:

Technical Skills:

- **BIM & IFC Standards**: Gained deep working knowledge of the IFC2X3 and IFC4 schemas, spatial hierarchy (`IfcProject` → `IfcSite` → `IfcBuilding` → `IfcBuildingStorey`), geometric representation types (SweptSolid, B-Rep, CSG Boolean), property sets, and material associations as defined by buildingSMART International.
- **ifcopenshell**: Developed proficiency with the `ifcopenshell` Python library for programmatic IFC file creation, entity management, and schema-compliant geometric construction.
- **Structural Engineering Standards**: Applied IS 800:2007 (Steel Design) and IRC 5:2015 (Bridge Loadings) standards in a software context, translating design tables and detailing rules directly into parametric geometry code.
- **Python Software Architecture**: Designed modular, maintainable Python packages with clear separation of concerns (extraction, mapping, generation, metadata) that integrate cleanly with a large existing codebase (Osdag/OsdagBridge).

- **Geometric Algorithms**: Implemented 3D coordinate transformations, cross-product-based axis alignment for bracing members, lateral shear transforms for skew geometry, polygonal profile construction, and Gaussian curve parametrization for W-beam profiles.
- **Database Integration**: Used `sqlite3` to query the Osdag material database (`Intg_osdag.sqlite`) and bind retrieved engineering properties to IFC elements programmatically.
- **Asynchronous Programming**: Applied Python `threading` to decouple long-running IFC export operations from the Qt GUI event loop to maintain UI responsiveness.

Professional Skills:

- **Open-Source Contribution**: Learned to navigate, extend, and contribute to a large, multi-contributor open-source codebase following established coding conventions and documentation practices.
- **Technical Documentation**: Authored a comprehensive developer handover report (this document) covering architecture, pipeline design, code references, known issues, and future roadmaps to ensure continuity for future contributors.
- **Debugging & Testing**: Diagnosed and resolved complex geometric artifacts (floating barriers, IFC viewer rendering bugs, skew distortions, unit scaling mismatches) using IFC inspection tools and systematic unit testing.
- **Research & Self-Learning**: Independently studied IFC schema documentation, IRC 5:2015 bridge standards, and IS 800:2007 design provisions to implement features without direct precedent in the codebase.

Appendices

A. Technical References

The complete source code developed during this internship, including the IFC export modules for both Osdag and OsdagBridge, is open-source and maintained on GitHub.

- **Osdag IFC Export Module (Task 1):**
  The core module is integrated into the official Osdag repository on the `dev` branch:
  https://github.com/osdag-admin/Osdag/tree/dev/src/osdag_core/export_ifc
  Development Fork: https://github.com/kavish1919/Osdag/tree/ifcwrapper/src/osdag_core/export_ifc
- **OsdagBridge IFC Export Module (Task 2):**
  The bridge-specific extraction module is maintained within the OsdagBridge repository on the `dev` branch:
  https://github.com/Aditya-Donde/OsdagBridge/tree/dev/src/osdagbridge/core/ifc_export_bridge
  Development Fork: https://github.com/kavish1919/IFCWrapperBridge/tree/ifcwrapper/src/osdagbridge/core/ifc_export_bridge

B. Internship Work Report

Internship Work Report			
Name:	Kavish Bishnoi		
Project:	Osdag & OsdagBridge		
Internship:	FOSSEE Semester Long Internship 2026		
DATE	**DAY**	**TASK**	**Hours**
15-Feb-2026	Sunday	Set up development environment, cloned Osdag repository, studied project and directory structure	5
16-Feb-2026	Monday	Read Osdag codebase, studied PyQt GUI architecture and the CAD object model	4
17-Feb-2026	Tuesday	Studied IFC standard documentation and explored the ifcopenshell Python library API	4
18-Feb-2026	Wednesday	Explored existing export_ifc folder skeleton; understood the ifc_generator.py entry point	5
19-Feb-2026	Thursday	Implemented IFC4 spatial hierarchy: IfcProject, IfcSite, IfcBuilding, IfcBuildingStorey	5
20-Feb-2026	Friday	Mapped first connection type (Fin Plate) to IfcMember and IfcPlate entities	4
21-Feb-2026	Saturday	Debugged coordinate system alignment for Fin Plate geometry in the IFC viewer	4
22-Feb-2026	Sunday	Implemented parametric I-section profile using IfcIShapeProfileDef	4
23-Feb-2026	Monday	Added angle section support using IfcLShapeProfileDef for cleat angle connections	4
24-Feb-2026	Tuesday	Added channel section support using IfcCShapeProfileDef	4

DATE	DAY	TASK	Hours
25-Feb-2026	Wednesday	Implemented bolt geometry as IfcMechanicalFastener with cylinder and sphere solids	4
26-Feb-2026	Thursday	Added nut and washer geometry to the fastener assembly	4
27-Feb-2026	Friday	Implemented IFC boolean subtraction for bolt hole cutting in plates using IfcBooleanClippingResult	6
28-Feb-2026	Saturday	Tested boolean operations with IS 800:2007 Table 19 clearance values; verified hole geometry	5
01-Mar-2026	Sunday	Added metadata mapping for steel grade properties queried from Intg_osdag.sqlite	4
02-Mar-2026	Monday	Implemented Pset_OsdagConnectionProperties property set builder	4
03-Mar-2026	Tuesday	Extended export to support Cleat Angle shear connection type	4
04-Mar-2026	Wednesday	Extended export to support Seated Angle and Seated Beam shear connections	4
05-Mar-2026	Thursday	Implemented End Plate shear connection geometry mapping	4
06-Mar-2026	Friday	Added weld geometry representation using IfcAnnotation with bounding box solids	4
07-Mar-2026	Saturday	Tested weld geometry and fixed coordinate alignment issues for fillet welds	4
08-Mar-2026	Sunday	Implemented Extended End Plate moment connection geometry	4
09-Mar-2026	Monday	Added Cover Plate Splice moment connection profile mapping	4
10-Mar-2026	Tuesday	Implemented Base Plate connection with anchor bolt geometry and grout pad	4
11-Mar-2026	Wednesday	Fixed HollowBasePlateCad vertical position offset bug; verified alignment in viewer	4
12-Mar-2026	Thursday	Added support for Hollow Section (SHS/RHS) Base Plate connections	4

DATE	DAY	TASK	Hours
13-Mar-2026	Friday	Implemented Beam-Column moment connection at column flange	4
14-Mar-2026	Saturday	Tested moment connection export, fixed 3D orientation issues for beam orientation	4
15-Mar-2026	Sunday	Added truss joint connection type support with gusset plate geometry	4
16-Mar-2026	Monday	Implemented welded truss connections and gusset plate profile mapping	4
17-Mar-2026	Tuesday	Added ColFlangeBeamWeb (Header Plate) connection type	4
18-Mar-2026	Wednesday	Fixed coordinate offset bug in ColFlangeBeamWeb; updated origin calculation logic	4
19-Mar-2026	Thursday	Implemented I-section sloped flange fallback using IfcArbitraryClosedProfileDef	4
20-Mar-2026	Friday	Added ghost fastener filtering logic for base plate bolt patterns with zero-dia bolts	4
21-Mar-2026	Saturday	Resolved dual bolt attribute name issue (bolt_dia vs bolt_diameter) across all connection types	6
22-Mar-2026	Sunday	Added anchor bolt geometry for base plate connections with thread and head detail	4
23-Mar-2026	Monday	Implemented asynchronous export using threading.Thread to keep PyQt UI responsive	4
24-Mar-2026	Tuesday	Added IFC file save with proper IFC2X3 schema header and GUID generation	4
25-Mar-2026	Wednesday	Extended metadata to include connection-specific properties (bolt grade, weld size, IS standard)	5
26-Mar-2026	Thursday	Added JSON serialisation bridge for CAD object subprocess launch	4
27-Mar-2026	Friday	Integration tested full pipeline with 10 connection types; verified valid IFC output	4
28-Mar-2026	Saturday	Fixed unit conversion inconsistencies (mm to metres) in geometry output across all mappers	5

DATE	DAY	TASK	Hours
29-Mar-2026	Sunday	Integration tested all shear connection types; verified bolt hole geometry consistency	4
30-Mar-2026	Monday	Fixed material colour assignment for steel (grey) and weld (orange) surface rendering	4
31-Mar-2026	Tuesday	Full pipeline integration test; verified all 28 supported connection types compile to valid IFC	5
01-Apr-2026	Wednesday	Raised Pull Request for Osdag IFC Export Module on GitHub; wrote PR description and inline documentation	6
02-Apr-2026	Thursday	Addressed PR review comments; cleaned up debug print statements, added module docstrings	5
03-Apr-2026	Friday	Finalized and merged Osdag PR; began studying OsdagBridge repository structure and architecture	5
04-Apr-2026	Saturday	Cloned OsdagBridge repository; explored directory layout and understanding ospgrillage integration	6
05-Apr-2026	Sunday	Read bridge_cad_extraction.py to understand the parametric plate girder geometry model	5
06-Apr-2026	Monday	Studied ospgrillage integration and the PlateGirder design object attributes	4
07-Apr-2026	Tuesday	Understood IRC 5:2015 geometry classes (CrashBarrier, Median, Railing) and profile dimensions	5
08-Apr-2026	Wednesday	Started OsdagBridge IFC module; created export_ifc_handler.py with async threading architecture	6
09-Apr-2026	Thursday	Designed the five-stage pipeline architecture; implemented Stage 1 handler orchestration	5
10-Apr-2026	Friday	Implemented PlateGirderIFCExtractor class with dynamic girder layout solver (_solve_girder_layout)	6
11-Apr-2026	Saturday	Implemented skew angle offset calculation using lateral shear transformation (tan of skew angle)	5

DATE	DAY	TASK	Hours
12-Apr-2026	Sunday	Built girder web and flange extraction from plate girder design object	4
13-Apr-2026	Monday	Implemented intermediate stiffener and end diaphragm extraction with correct orientation	5
14-Apr-2026	Tuesday	Added cross-bracing extraction for X-type and K-type configurations with chord and diagonal roles	5
15-Apr-2026	Wednesday	Implemented deck slab extraction with IRC 5:2015 deck width calculation	4
16-Apr-2026	Thursday	Fixed deck width synchronization with native OsdagBridge width computation	4
17-Apr-2026	Friday	Implemented RCC crash barrier profile extraction using IRC 5 geometry lookup classes	4
18-Apr-2026	Saturday	Added metallic crash barrier extraction with W-beam post and kerb layout	4
19-Apr-2026	Sunday	Implemented bridge_geometry_mapper.py; added IfcRectangleProfileDef for plate members	5
20-Apr-2026	Monday	Added composite double-angle and double-channel profile construction using IfcCompositeProfileDef	5
21-Apr-2026	Tuesday	Implemented IfcArbitraryClosedProfileDef for RCC crash barrier polygonal cross-sections	5
22-Apr-2026	Wednesday	Leave	0
23-Apr-2026	Thursday	Leave	0
24-Apr-2026	Friday	Leave	0
25-Apr-2026	Saturday	Built Faceted B-Rep builder (create_faceted_brep_from_3d_deck) for skewed deck slab geometry	5
26-Apr-2026	Sunday	Added IfcArbitraryProfileDefWithVoids for hollow RCC railing profiles with internal void	5
27-Apr-2026	Monday	Implemented explicit RGB surface style rendering for Steel and RCC material differentiation	5

DATE	DAY	TASK	Hours
28-Apr-2026	Tuesday	Built bridge_ifc_generator.py with IFC4 spatial hierarchy and global mm-to-metre scale factor	5
29-Apr-2026	Wednesday	Implemented _process_plate() for girder webs, flanges, and stiffeners with axis orientation	5
30-Apr-2026	Thursday	Implemented _process_brace() with 3D extrusion vector via cross-product axis alignment	5
01-May-2026	Friday	Implemented _process_deck() for skewed slab IfcSlab geometry placement at top flange elevation	5
02-May-2026	Saturday	Built metallic median multi-solid assembly (kerb extrusion, C-channel posts, spacer blocks, rails)	6
03-May-2026	Sunday	Implemented Gaussian curve W-beam rail profile parametrization (40-point double-hump)	4
04-May-2026	Monday	Built triangular wedge bearing and cylindrical roller support geometry	4
05-May-2026	Tuesday	Implemented metadata_mapper.py with sqlite3 material property queries from Intg_osdag.sqlite	5
06-May-2026	Wednesday	Added component-specific property mapping: girder web depth/thickness, flange width	4
07-May-2026	Thursday	Implemented brace metadata decoder for structural system (End Diaphragm vs Intermediate) and member roles	6
08-May-2026	Friday	Linked all properties to Pset_OsdagBridgeProperties via IfcRelDefinesByProperties	4
09-May-2026	Saturday	Full pipeline integration test; verified girder, deck, and barrier IFC output in viewer	5
10-May-2026	Sunday	Tested skew bridge export (30° skew); verified all components align with skew geometry	5
11-May-2026	Monday	Raised Pull Request for OsdagBridge IFC Export Module; wrote PR description and code documentation	6
12-May-2026	Tuesday	Addressed mentor feedback on OsdagBridge PR; updated docstrings and inline comments	4

DATE	**DAY**	**TASK**	**Hours**
13-May-2026	Wednesday	Began drafting internship report; structured chapters, appendices, and TOC	4
14-May-2026	Thursday	Wrote Chapter 2: Screening Task and Chapter 3: Internship Task 1 Introduction & Setup	4
15-May-2026	Friday	Documented Task 1 Module Architecture, File Structure, and Export Pipeline	4
16-May-2026	Saturday	Wrote Task 1 Geometry Generation, Types Supported, and How-To Guides	4
17-May-2026	Sunday	Authored Chapter 4: Internship Task 2 Introduction, Architecture, and Setup	4
18-May-2026	Monday	Documented Task 2 Bridge Export Pipeline, Deck Width, and Stiffener Extraction	4
19-May-2026	Tuesday	Wrote Task 2 Cross-Bracing Dispatch and Safety Component Positioning	4
20-May-2026	Wednesday	Documented Task 2 Metadata Application and Known Issues	4
21-May-2026	Thursday	Formatted Appendices (Technical References, Work Report) and Conclusions chapter	4
22-May-2026	Friday	Final compilation, cross-reference check, and submission of internship report	4

Bibliography

[1] buildingSMART International. *Industry Foundation Classes Release 2x3 Technical Corrigendum 1.* buildingSMART International, 2007.

[2] buildingSMART International. *Industry Foundation Classes Release 4 Addendum 2.* buildingSMART International, 2016.

[3] Bureau of Indian Standards. *IS 800:2007 General Construction in Steel - Code of Practice (Third Revision).* Bureau of Indian Standards, New Delhi, India, 2007.

[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] Indian Roads Congress. *IRC:5-2015 Standard Specifications and Code of Practice for Road Bridges, Section I - General Features of Design (Eighth Revision).* Indian Roads Congress, New Delhi, India, 2015.

[6] Thomas Krijnen et al. Ifcopenshell: An open source ifc library and geometry engine, 2026.

[7] FOSSEE Project. FOSSEE News - January 2018, vol 1 issue 3.

[8] FOSSEE Project. Fossee website.

[9] FOSSEE Project. Osdag website.