FOSSEE Semester Long Internship Report

On

Development of Modular Report Generation and PDF Export for the Osdag 3psLCCA Desktop App

Submitted by

Raghav Dadhich

3rd Year B.Tech Student, Department of SCAI

VIT Bhopal University

Sehore

Under the Guidance of

Prof. Siddhartha Ghosh

Department of Civil Engineering

Indian Institute of Technology Bombay

Mentors:

Ajmal Babu M S

Parth Karia

Swastik Gupta

May 29, 2026

Acknowledgments

- Start with a general statement of thanks. Express your overall gratitude to everyone who supported you during your project or research.
- Project staff at the Osdag team, Ajmal Babu M. S., Ajinkya Dahale, and Parth Karia,
- Osdag Principal Investigator (PI) Prof. Siddhartha Ghosh, Department of Civil Engineering at IIT Bombay
- FOSSEE PI Prof. Kannan M. Moudgalya, FOSSEE Project Investigator, Department of Chemical Engineering, IIT Bombay
- FOSSEE Managers Usha Viswanathan and Vineeta Parmar and their entire team
- Acknowledge the support from the National Mission on Education through Information and Communication Technology (ICT), Ministry of Education (MoE), Government of India, for their role in facilitating this project
- Acknowledge your colleagues who worked with you during your internship or project.
- If appropriate, thank your college, department, head, and principal for their support during your studies.

Contents

Chapter 1

Introduction

1.1 National Mission in Education through ICT

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

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

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

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

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

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

1.1.1 ICT Initiatives of MoE

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

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

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

1.2 FOSSEE Project

The FOSSEE (Free/Libre and Open Source Software for Education) project promotes the use of FLOSS tools in academia and research. It is part of the National Mission on

Education through Information and Communication Technology (NMEICT), Ministry of Education (MoE), Government of India.

1.2.1 Projects and Activities

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

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

1.2.2 Fellowships

FOSSEE offers various internship and fellowship opportunities for students:

- Winter Internship
- Summer Fellowship
- Semester-Long Internship

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

For more details, visit the official FOSSEE website.

Figure 1.1: FOSSEE Projects and Activities

1.3 Osdag Software

Osdag (Open steel design and graphics) is a cross-platform, free/libre and open-source software designed for the detailing and design of steel structures based on the Indian Standard IS 800:2007. It allows users to design steel connections, members, and systems through an interactive graphical user interface (GUI) and provides 3D visualizations of designed components. The software enables easy export of CAD models to drafting tools for construction/fabrication drawings, with optimized designs following industry best practices. 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 3psLCCA Desktop GUI

The 3psLCCA desktop GUI is designed for structured life-cycle cost analysis. It provides:

- **Input sidebar**: Structured navigation across general, bridge, construction, traffic, financial, and environmental inputs.
- **Outputs workspace**: Summary cards, charts, and report generation actions.
- **Report controls**: Direct access to PDF export and the modular report workflow.
- **Status indicators**: Clear feedback on save state and calculation progress.

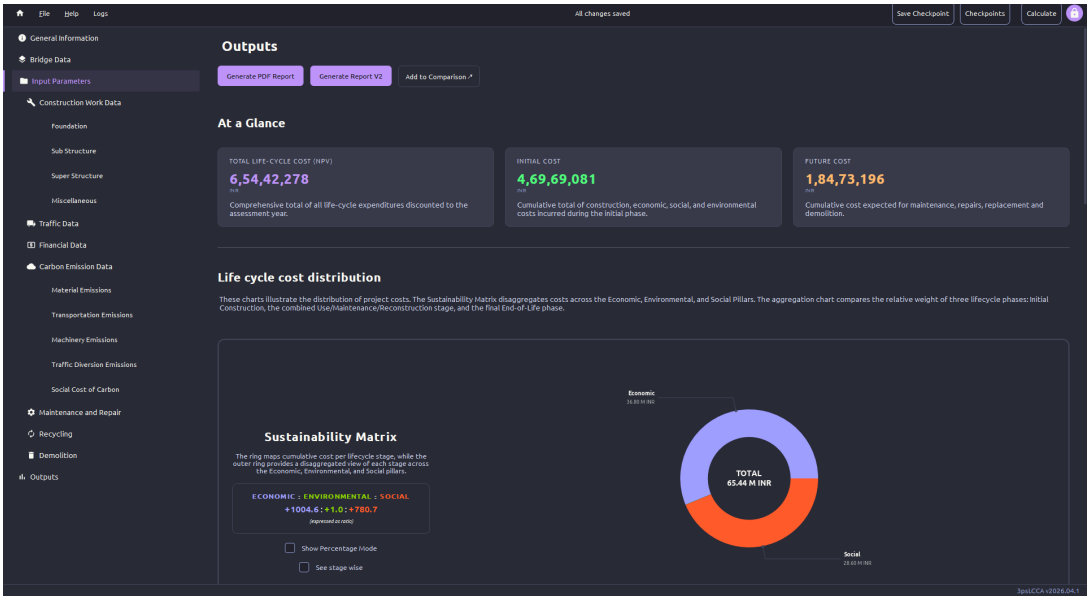

Figure 1.2: 3psLCCA Desktop Application (Outputs workspace)

1.3.2 Features

- **Structured LCCA workflow**: Guided input sections for construction, traffic, financial, and environmental data.
- **Computation and summaries**: Automatic calculation of NPV, stage costs, and sustainability breakdowns.
- **Visual outputs**: Charts for cost distribution and sustainability matrix views.
- **PDF export**: Standard and modular (V2) report generation with section-level customization.

Chapter 2

Screening Task

2.1 Problem Statement

The screening task was to generate the `beam_analysis_report.pdf` report using LaTeX. The expected output was not only a formatted PDF, but a reusable workflow that can convert beam data into a technical report with minimum manual editing.

The report had to include:

- A clean title page and table of contents.
- Beam description and source data explanation.
- Tabulated force data from Excel.
- Shear Force Diagram (SFD) and Bending Moment Diagram (BMD).
- A concise summary of results and theoretical appendix.

2.2 Tasks Done

Github Repo for Screening Task - https://github.com/raghav3615/fossee

I implemented the report generation workflow using Python and PyLaTeX in `resources/generate_report.py`. The script reads force data, generates all report sections, creates plots using TikZ/pgfplots, compiles the PDF, and performs post-build cleanup.

While implementing this task, I approached it as a sequence of practical problems rather than only formatting work. The major challenges and solutions are documented below.

2.2.1 Challenge 1: Making the pipeline fail-safe for input data

The first issue was input reliability. A report pipeline can silently produce wrong output if the source table has missing or renamed columns. To avoid that, I added schema validation before any rendering step.

Listing 2.1: Validating force table before report generation

```
def read_force_data(excel_path: str) -> pd.DataFrame:
    if not os.path.exists(excel_path):
        raise FileNotFoundError(f"Excel file not found: {excel_path}")

    df = pd.read_excel(excel_path)
    required_columns = ['x', 'Shear force', 'Bending Moment']

    for col in required_columns:
        if col not in df.columns:
            raise ValueError(f"Missing required column: {col}")

    return df
```

This helped in two ways:

- It made debugging immediate (clear error at source, instead of LaTeX failure later).
- It made the script reusable for future datasets with strict input expectations.

2.2.2 Challenge 2: Generating readable engineering plots from discrete data

The next challenge was producing clean SFD/BMD visuals from point-wise values. A direct line plot was not sufficient for readability, so I used dynamic axis scaling, annotations, and gradient-filled strips to show force variation more clearly.

Listing 2.2: Plot logic for adaptive SFD visualization

```python
def generate_sfd_tikz(df: pd.DataFrame) -> str:
    x_vals = df['x'].tolist()
    sf_vals = df['Shear force'].tolist()

    max_sf = max(sf_vals)
    min_sf = min(sf_vals)
    x_max = max(x_vals)

    num_ticks = 6
    x_step = x_max / (num_ticks - 1)
    all_xticks = [i * x_step for i in range(num_ticks)]

    y_range = max_sf - min_sf
    y_step = 20 if y_range <= 100 else 50 if y_range <= 200 else 100
    yticks = [i for i in range(int(min_sf // y_step * y_step), int(
        max_sf + y_step), y_step) if i != 0]

    # Coordinates and gradient strips are generated here and returned
        as TikZ code
    # so the final PDF has vector-quality diagrams with force
        annotations.
```

Key decisions here were:

- Axis ticks are computed from data range, not hard-coded.
- Max/min values are identified and labeled directly on the plot.
- The same approach is mirrored for BMD to keep visual consistency.

2.2.3 Challenge 3: Ensuring stable and repeatable PDF builds

After generating the `.tex` file, compilation had to be dependable on repeated runs. A single-pass compile occasionally gave incomplete references and contents entries, so I used two compilation passes and controlled cleanup.

Listing 2.3: Reliable compile and cleanup flow

```python
def compile_latex(output_name: str):
    for i in range(2):
        subprocess.run(
```

```
            ['pdflatex', '-interaction=nonstopmode', f'{output_name}.
                tex'],
            capture_output=True,
            text=True
        )

def cleanup_latex_files(output_name: str):
    extensions = ['.aux', '.log', '.out', '.toc', '.tex']
    for ext in extensions:
        filepath = f'{output_name}{ext}'
        if os.path.exists(filepath):
            os.remove(filepath)
```

This solved two recurring problems:

- Incomplete TOC/header references in first-pass builds.
- Clutter from intermediate files during repeated testing.

2.2.4 End-to-End Orchestration

Once the above issues were solved separately, I integrated everything into a single pipeline function.

Listing 2.4: End-to-end report generation pipeline

```
def generate_report(excel_path: str, beam_image_path: str, output_name:
    str):
    df = read_force_data(excel_path)
    beam_length = max(df['x'].tolist())

    doc = create_document()
    add_title_page(doc, beam_length)
    add_table_of_contents(doc)

    abs_beam_path = os.path.abspath(beam_image_path)
    add_introduction(doc, abs_beam_path, beam_length, len(df))
    add_input_data_section(doc, df)

    doc.append(NewPage())
    add_analysis_section(doc, df)
```

```
15
16      doc.append(NewPage())
17      add_appendix(doc)
18
19      doc.generate_tex(output_name)
20      compile_latex(output_name)
21
22      if os.path.exists(f'{output_name}.pdf'):
23          cleanup_latex_files(output_name)
```

2.2.5 Results Obtained

The final report was generated successfully with an 8-page structure and all required sections. For the input dataset used in the screening task, the key results were:

- Maximum positive shear force: $45\,\text{kN}$ at $x = 0\,\text{m}$.
- Maximum negative shear force: $-45\,\text{kN}$ at $x = 15\,\text{m}$.
- Maximum bending moment: $168.75\,\text{kN} \cdot \text{m}$ at midspan $x = 7.5\,\text{m}$.

These values were correctly reflected in the generated table, diagram annotations, and summary section.

2.2.6 Outcome

The screening task outcome was a complete and reproducible reporting workflow. Instead of creating a one-time LaTeX document manually, I built a pipeline that can be reused for future beam datasets with only input-file changes.

The final output `beam_analysis_report.pdf` includes:

- Proper report structure and navigation.
- Engineering-quality tables and diagrams.
- Correctly computed and highlighted shear/moment extremes.
- Stable generation through automated compile and cleanup steps.

Chapter 3

Internship Task 1

3.1 Task 1: Problem Statement

The assigned task was to generate an LCCA report for Osdag using Python and LaTeX, with output format aligned to the reference report provided by the team.

The required output (`task1/LCCA_Report.pdf`) had to include:

- **Chapter 1:** Introduction to LCCA with the 3PS framework figure.
- **Chapter 2:** Input data chapter with bridge, financial, construction, traffic, and environmental inputs.
- **Structured tables:** Sequentially numbered tables (2-1 to 2-17) with formatting consistency.
- **Reusable flow:** Same script should work for new datasets without changing report layout code.

In practice, this was a report-engineering task: convert a complex static format into a reliable and reusable generation pipeline.

3.2 Task 1: Tasks Done

I implemented the LCCA reporting module under `task1/`. The core work is in `lcca_report_generator.py` and `lcca_report_data.py`. The implementation for this task is CLI based and focused on reproducible report generation.

3.2.1 Task 1.1: Baseline report generation and format matching

I first built a baseline PDF generator to reproduce the exact chapter and table layout expected in the reference output. The first milestone was to ensure that the generated report had:

- Correct section hierarchy (Introduction to LCCA, Input data).
- Proper table captions and subsection flow.
- Stable page breaks and readable table placement.

This step established the skeleton that later dynamic inputs could plug into.

3.2.2 Task 1.2: Dynamic data integration and decoupling

After validating static formatting, I removed hardcoded values and moved all report inputs to a separate data layer (`lcca_report_data.py`).

This solved two practical issues:

- Report logic and project data were tightly coupled in the initial implementation.
- Updating values for new bridges required code edits in multiple places.

By moving bridge, financial, traffic, and emission inputs into dictionaries, the same report template became reusable for new project scenarios.

3.2.3 Task 1.3: Configurable section toggling with JSON

The next requirement was report customization. Not every user needs all 17 input tables in every run. To support this, I implemented configuration flags (`show_...`) and JSON-based overrides.

This allowed selective report generation without modifying Python source code.

3.2.4 Task 1.4: CLI workflow and execution paths

Since this task is CLI based, I finalized three execution paths for report generation:

- Default run: generate full report using internal data dictionaries.

- Custom output run: generate report with user-specified output file name.
- JSON override run: inject section toggles or custom values through an external JSON file.

This made the module script-driven and automation-friendly for repeated report generation.

3.2.5 Task 1.5: Final output validation

The final generated `LCCA_Report.pdf` was checked against the reference for:

- Section ordering and table numbering.
- Header-level consistency for wide tables.
- Input data completeness from bridge description to on-site emissions.

The output successfully included Introduction and Input Data chapters with all expected tables.

3.3 Task 1: Architecture and Workflow

The implementation follows a two-layer structure:

- **Data layer** - `lcca_report_data.py`: stores all project parameters as dictionaries.
- **Generator layer** - `lcca_report_generator.py`: builds LaTeX sections/tables dynamically and exposes CLI options.

This structure made debugging and extension easier, especially while handling large tables, optional sections, and command-line overrides.

3.4 Task 1: Python Code

3.4.1 Description of the Script

The script is designed around report reliability and customization:

- Reusable helper functions generate key-value and multi-column tables.
- A configuration map controls section visibility.
- Data values are loaded from a dedicated data module and can be overridden using JSON.
- CLI arguments provide output naming and optional JSON-based customization.

3.4.2 Python Code Snippet 1: Reusable table builders

One recurring issue in this task was avoiding duplicate table code while preserving consistent spacing and caption style. The following helper functions were used across all sections.

Listing 3.1: Reusable helpers for key-value and multi-column tables

```
1  def add_kv_table(doc, caption, data, col1="10cm", col2="4.5cm"):
2      doc.append(NoEscape(r"\vspace{4pt}"))
3      doc.append(NoEscape(r"\needspace{12\baselineskip}"))
4      doc.append(NoEscape(r"\noindent\captionof{table}{" + escape_latex(
           caption) + r"}"))
5      doc.append(NoEscape(r"\vspace{4pt}"))
6      with doc.create(Tabular(f"|p{{{col1}}}|p{{{col2}}}|")) as table:
7          table.add_hline()
8          for key, val in data.items():
9              table.add_row([escape_latex(key), escape_latex(str(val))])
10             table.add_hline()
11
12
13 def add_multi_table(doc, caption, headers, data, col_spec):
14     doc.append(NoEscape(r"\vspace{4pt}"))
15     doc.append(NoEscape(r"\needspace{12\baselineskip}"))
16     doc.append(NoEscape(r"\noindent\captionof{table}{" + escape_latex(
           caption) + r"}"))
17     with doc.create(Tabular(col_spec)) as table:
18         table.add_hline()
19         table.add_row([bold(h) for h in headers])
20         table.add_hline()
21         for key, vals in data.items():
```

```
            row = [escape_latex(key)] + [escape_latex(str(v)) for v in
                vals]
            table.add_row(row)
            table.add_hline()
```

3.4.3 Python Code Snippet 2: JSON-driven configuration override

The core problem solved here was making the report configurable without editing source code each time. The snippet below merges user-provided JSON settings with defaults.

Listing 3.2: Config and data override mechanism

```
config = {
    "show_bridge_description": True,
    "show_financial_data": True,
    "show_construction_materials": True,
    "show_lcc_assumptions": True,
    "show_use_stage_details": True,
    "show_avg_daily_traffic": True,
    "show_road_traffic_data": True,
    "show_human_injury_cost": True,
    "show_vehicle_damage_cost": True,
    "show_tyre_cost_data": True,
    "show_fuel_oil_grease": True,
    "show_new_vehicle_cost": True,
    "show_social_cost_carbon": True,
    "show_material_emission_factors": True,
    "show_use_stage_emission_assumptions": True,
    "show_vehicle_emission_factors": True,
    "show_onsite_emissions": True,
}

if input_json and os.path.exists(input_json):
    with open(input_json, "r") as f:
        user_data = json.load(f)
        if "config" in user_data:
            config.update(user_data["config"])
        for key in data.keys():
```

```
            if key in user_data:
                data[key] = user_data[key]
```

3.4.4 Python Code Snippet 3: CLI entry and argument parsing

The CLI parser keeps report generation simple while allowing customization through command-line flags.

Listing 3.3: CLI interface for report generation

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Generate LCCA Report")
    parser.add_argument("--json", type=str, help="Path to JSON data and config file")
    parser.add_argument(
        "--out",
        type=str,
        default="LCCA_Report",
        help="Output PDF filename (without extension)",
    )
    args = parser.parse_args()

    generate_report(args.out, input_json=args.json)
```

3.4.5 Python Code Snippet 4: Compile fallback for debugging

A practical issue during development was compile-time dependency errors on some systems. The generator therefore keeps a fallback mode that saves `.tex` even if PDF compilation fails.

Listing 3.4: Compile with fallback to TeX source

```
try:
    doc.generate_pdf(
        output_filename,
        clean_tex=False,
        compiler="pdflatex",
        compiler_args=["-interaction=nonstopmode"],
```

```
    )
    print(f"Done -> {output_filename}.pdf")
except Exception as e:
    print(f"PDF compile issue: {e}")
    doc.generate_tex(output_filename)
    print(f"TeX saved -> {output_filename}.tex")
```

3.4.6 Outcome of Task 1

The final module generates `LCCA_Report.pdf` with Introduction and Input Data chapters, including all required tables (2-1 to 2-17), and supports:

- CLI-based report generation,
- JSON-driven customization,
- repeatable batch-friendly workflow through command-line execution.

This task established a reusable reporting foundation for future LCCA workflows in Osdag.

Chapter 4

Internship Task 2

4.1 Task 2: Problem Statement

The second internship task was to build an interactive desktop workflow for LCCA report generation, using the Task 1 report engine as the backend. In Task 1, the flow was CLI driven and suitable for scripted runs. In Task 2, the same report had to be generated through a GUI so that users could customize report content without modifying Python files.

The target output (`task2/LCCA_Report.pdf`) had to preserve the existing technical format while adding selection control at runtime. The key requirements were:

- **Interactive section selection:** Users should be able to choose which tables to include.
- **Hierarchical behavior:** Section, subsection, and table checkboxes should stay synchronized.
- **Direct integration:** GUI output should map directly to `KEY_SHOW_*` flags used by the generator.
- **Formatting stability:** Chapter structure and table numbering must remain consistent with Task 1 output.
- **Desktop usability:** Button state, status text, success/error dialogs, and visibility in different OS themes should be handled cleanly.

So, this task was not only about designing a window. It required a reliable bridge between GUI state management and deterministic LaTeX report rendering.

4.2 Task 2: Tasks Done

I implemented the GUI workflow in `task2/lcca_gui.py` and connected it to the class-based generator in `task2/lcca_report_generator.py`. The data and visibility flags were sourced from `task2/lcca_report_data.py`.

4.2.1 Task 2.1: Translating report structure into GUI metadata

The first step was to represent report hierarchy in a machine-readable format. I used:

- `SECTION_MAP` for section to subsection mapping.
- `SUBSECTION_TABLE_MAP` for subsection to table-label and config-key mapping.

This separated report content logic from widget construction. As a result, the tree can be rebuilt automatically if report sections are extended later.

4.2.2 Task 2.2: Building the section tree dynamically

Instead of hardcoding every node in UI code, I generated tree items by iterating the metadata maps. The generated tree has three levels:

- Level 1: report sections,
- Level 2: subsections,
- Level 3: table entries linked to `KEY_SHOW_*` identifiers.

Each table node stores its config key using `Qt.UserRole`, which later enables direct extraction into a backend config dictionary.

4.2.3 Task 2.3: Implementing parent-child state propagation

The most important interaction challenge was checkbox consistency. If a parent node is unchecked, all descendants must be unchecked. If only some descendants are checked, the parent must appear partially checked.

I implemented recursive child-state updates and upward parent-state recomputation in `SectionTreeWidget`. Signal blocking is used while updating nodes to avoid recursive event loops.

4.2.4 Task 2.4: Converting selected nodes into generator config

After selection logic was stable, I implemented recursive tree traversal to produce a config dictionary in the expected format: `KEY_SHOW_* -> bool`.

This step made GUI selection directly usable by the report generator, without any custom translation layer outside the tree widget.

4.2.5 Task 2.5: Wiring GUI action to report generation

The Generate button now performs a direct backend call:

- collect config from tree,
- call `generate_report(..., config_override=config)`,
- show success or error dialog,
- restore button state in `finally`.

I avoided subprocess-based invocation to keep flow simple and to provide immediate feedback from Python exceptions.

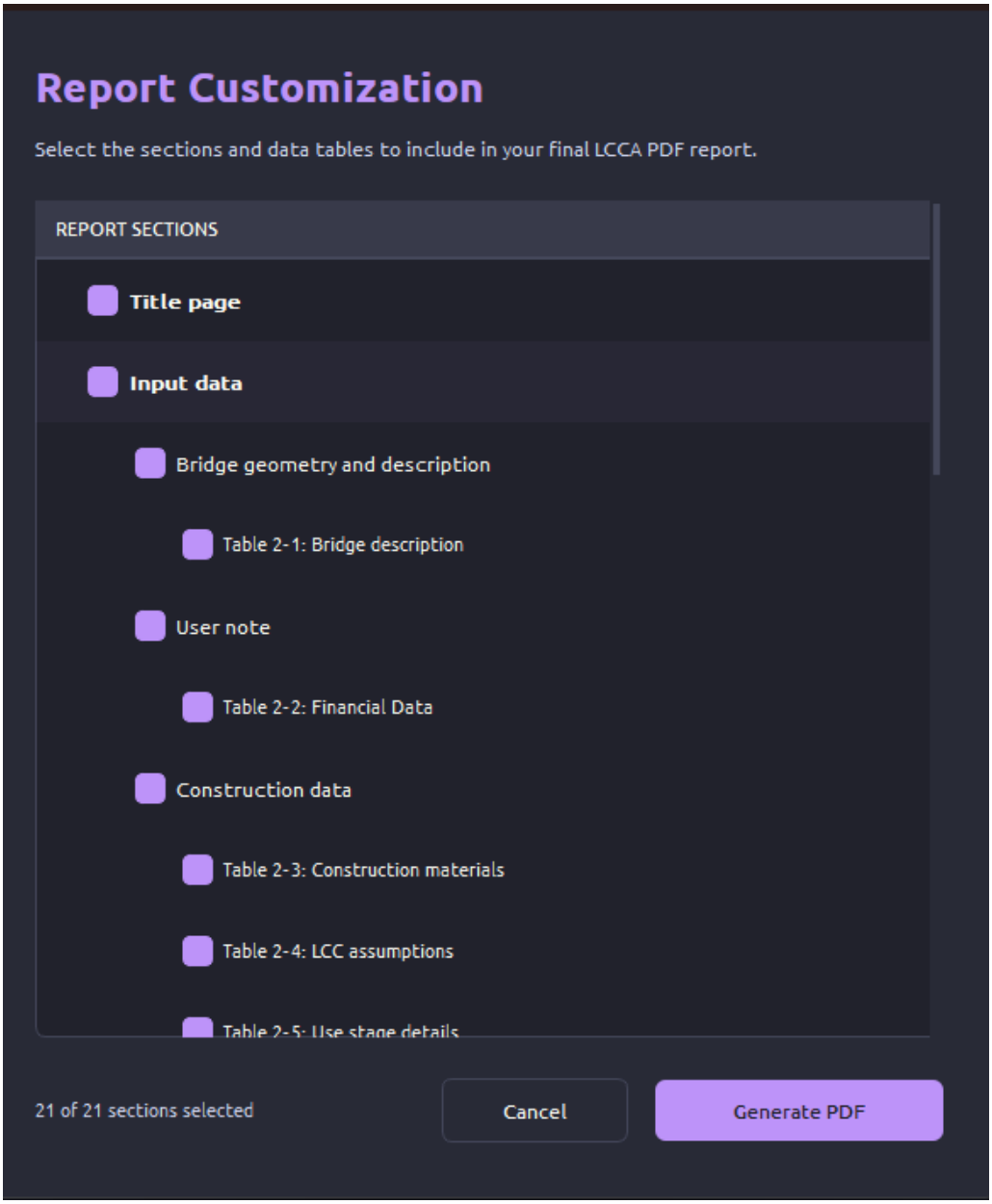

Figure 4.1: Report customization window for selecting LCCA sections before PDF generation

4.2.6 Task 2.6: Reliability and UX safeguards

To make repeated report generation practical, I added:

- temporary button disable and progress text (`Generating...`),
- live status label showing selected table count,
- automatic PDF opening when generation succeeds,
- exception dialogs for user-visible error reporting,

- compile fallback in generator to save `.tex` when PDF build fails.

These safeguards improved user trust and reduced failure ambiguity.

4.2.7 Task 2.7: Preserving CLI and JSON workflows

While adding GUI support, I retained existing CLI flexibility. The report wrapper now supports both:

- `--json` based configuration/data override (terminal workflow),
- programmatic `config_override` from GUI checkbox states.

This maintained backward compatibility and ensured no regression for automated usage.

4.3 Task 2: Architecture and Workflow

The final design has three coordinated layers:

- **Data layer** - `lcca_report_data.py`: visibility keys, section map, and default data dictionaries.
- **GUI orchestration layer** - `lcca_gui.py`: section tree creation, selection propagation, config extraction, and trigger button logic.
- **LaTeX rendering layer** - `lcca_report_generator.py`: conditional table rendering using config flags and PDF compilation.

Execution flow implemented in Task 2:

1. Build tree from `SECTION_MAP` and `SUBSECTION_TABLE_MAP`.
2. User selects/deselects report tables.
3. `get_config()` extracts `KEY_SHOW_* -> bool` values.
4. Generator merges defaults, optional JSON values, and GUI override.
5. `LCCAReportLatex.save_latex()` creates report and compiles PDF.

4.4 Task 2: Python Code

4.4.1 Description of the Script

The script design keeps report formatting decisions in the generator and customization decisions in the GUI tree. This separation made the implementation easier to maintain and test, especially for selective table rendering.

4.4.2 Python Code Snippet 1: Mapping model

This model maps visible tree labels to backend config keys.

Listing 4.1: Metadata maps used by the section selector

```
SECTION_MAP = {
    "Introduction to LCCA": [],
    "Input data": [
        "Bridge geometry and description",
        "User note",
        "Construction data",
        "Traffic data",
        "Environmental input data",
    ],
}

SUBSECTION_TABLE_MAP = {
    "Bridge geometry and description": [
        ("Table 2-1: Bridge description", KEY_SHOW_BRIDGE_DESC),
    ],
    "User note": [
        ("Table 2-2: Financial Data", KEY_SHOW_FINANCIAL),
    ],
    "Construction data": [
        ("Table 2-3: Construction materials", KEY_SHOW_CONSTRUCTION),
        ("Table 2-4: LCC assumptions", KEY_SHOW_LCC_ASSUMPTIONS),
        ("Table 2-5: Use stage details", KEY_SHOW_USE_STAGE),
    ],
    "Traffic data": [
        ("Table 2-6: Average daily traffic", KEY_SHOW_AVG_TRAFFIC),
```

```
        ("Table 2-7: Road and traffic data", KEY_SHOW_ROAD_TRAFFIC),
        ("Table 2-8: Human injury cost", KEY_SHOW_HUMAN_INJURY),
        ("Table 2-9: Vehicle damage cost", KEY_SHOW_VEHICLE_DAMAGE),
        ("Table 2-10: Tyre cost data", KEY_SHOW_TYRE_COST),
        ("Table 2-11: Fuel, oil and grease", KEY_SHOW_FUEL_OIL),
        ("Table 2-12: Cost of new vehicle", KEY_SHOW_NEW_VEHICLE),
    ],
    "Environmental input data": [
        ("Table 2-13: Social cost of carbon", KEY_SHOW_SOCIAL_CARBON),
        ("Table 2-14: Material emission factors",
           KEY_SHOW_MATERIAL_EMISSION),
        ("Table 2-15: Use stage emissions", KEY_SHOW_USE_EMISSION),
        ("Table 2-16: Vehicle emission factors",
           KEY_SHOW_VEHICLE_EMISSION),
        ("Table 2-17: On-site emissions", KEY_SHOW_ONSITE_EMISSION),
    ],
}
```

4.4.3 Python Code Snippet 2: Dynamic tree construction

The tree is built programmatically and each table node stores its corresponding key.

Listing 4.2: Tree generation from metadata maps

```
def build_from_sections(self, section_map, table_map):
    self.clear()

    for section_name, subsections in section_map.items():
        section_item = QTreeWidgetItem(self, [section_name])
        section_item.setFlags(section_item.flags() | Qt.
           ItemIsUserCheckable)
        section_item.setCheckState(0, Qt.Checked)

        for subsection in subsections:
            sub_item = QTreeWidgetItem(section_item, [str(subsection)])
            sub_item.setFlags(sub_item.flags() | Qt.ItemIsUserCheckable
               )
            sub_item.setCheckState(0, Qt.Checked)

            if subsection in table_map:
```

```
                for label, config_key in table_map[subsection]:
                    table_item = QTreeWidgetItem(sub_item, [label])
                    table_item.setFlags(table_item.flags() | Qt.
                        ItemIsUserCheckable)
                    table_item.setCheckState(0, Qt.Checked)
                    table_item.setData(0, Qt.UserRole, config_key)

    self.expandAll()
```

4.4.4 Python Code Snippet 3: Parent-child checkbox synchronization

This prevents inconsistent selection state during user interaction.

Listing 4.3: Signal-safe hierarchical checkbox propagation

```
def on_item_changed(self, item, column):
    if column == 0:
        self.blockSignals(True)

        state = item.checkState(0)
        self._set_children_state(item, state)
        self.update_parent_state(item)

        self.blockSignals(False)
        self.selectionChanged.emit()

def update_parent_state(self, item):
    parent = item.parent()
    if parent is None:
        return

    total_children = parent.childCount()
    checked_children = sum(
        1 for i in range(total_children)
        if parent.child(i).checkState(0) == Qt.Checked
    )

    if checked_children == total_children:
```

```
24            parent.setCheckState(0, Qt.Checked)
25        elif checked_children == 0:
26            parent.setCheckState(0, Qt.Unchecked)
27        else:
28            parent.setCheckState(0, Qt.PartiallyChecked)
29
30        self.update_parent_state(parent)
```

4.4.5 Python Code Snippet 4: Config extraction from selected nodes

The tree is traversed recursively to produce generator-ready config.

Listing 4.4: Collecting KEY_SHOW flags from tree state

```
1  def get_config(self):
2      config = {}
3      self._collect_config(self.invisibleRootItem(), config)
4      return config
5
6  def _collect_config(self, item, config):
7      for i in range(item.childCount()):
8          child = item.child(i)
9          config_key = child.data(0, Qt.UserRole)
10         if config_key:
11             config[config_key] = child.checkState(0) == Qt.Checked
12         self._collect_config(child, config)
```

4.4.6 Python Code Snippet 5: GUI trigger and feedback loop

This block handles button state, report generation, and user feedback.

Listing 4.5: Generate action with status and error handling

```
1  def generate_report(self):
2      config = self.tree_sections.get_config()
3
4      try:
5          self.btn_generate.setText("Generating...")
```

```
6           self.btn_generate.setEnabled(False)
7           QApplication.processEvents()
8
9           generate_report("LCCA_Report", config_override=config)
10
11          QMessageBox.information(
12              self,
13              "Success",
14              "Report generated successfully!\n\nLCCA_Report.pdf",
15          )
16
17          if os.path.exists("LCCA_Report.pdf"):
18              os.startfile("LCCA_Report.pdf")
19
20      except Exception as e:
21          QMessageBox.critical(self, "Error", f"An error occurred: {str(e
                )}")
22      finally:
23          self.btn_generate.setText("Generate Report")
24          self.btn_generate.setEnabled(True)
```

4.4.7 Python Code Snippet 6: Unified generator wrapper (CLI + GUI)

The wrapper merges defaults with JSON and GUI overrides, then invokes the class-based renderer.

Listing 4.6: Backward-compatible report wrapper with overrides

```
1  def generate_report(output_filename="LCCA_Report", input_json=None,
       config_override=None):
2      config = {
3          KEY_SHOW_BRIDGE_DESC: True,
4          KEY_SHOW_FINANCIAL: True,
5          KEY_SHOW_CONSTRUCTION: True,
6          KEY_SHOW_LCC_ASSUMPTIONS: True,
7          KEY_SHOW_USE_STAGE: True,
8          KEY_SHOW_AVG_TRAFFIC: True,
9          KEY_SHOW_ROAD_TRAFFIC: True,
```

```
        KEY_SHOW_HUMAN_INJURY: True,
        KEY_SHOW_VEHICLE_DAMAGE: True,
        KEY_SHOW_TYRE_COST: True,
        KEY_SHOW_FUEL_OIL: True,
        KEY_SHOW_NEW_VEHICLE: True,
        KEY_SHOW_SOCIAL_CARBON: True,
        KEY_SHOW_MATERIAL_EMISSION: True,
        KEY_SHOW_USE_EMISSION: True,
        KEY_SHOW_VEHICLE_EMISSION: True,
        KEY_SHOW_ONSITE_EMISSION: True,
    }

    if input_json and os.path.exists(input_json):
        with open(input_json, "r") as f:
            user_data = json.load(f)
            if "config" in user_data:
                config.update(user_data["config"])

    if config_override is not None:
        config.update(config_override)

    doc = LCCAReportLatex()
    doc.save_latex(config, data, output_filename)
```

4.5 Outcome of Task 2

Task 2 delivered a working GUI customization layer for the LCCA report pipeline. Users can now generate a filtered report through checkbox selection while keeping the same technical structure and output quality.

The final implementation achieved:

- 17 table-level visibility controls through a 3-level tree interface.
- Stable parent-child checkbox logic, including partial-check state.
- Shared backend path for GUI and CLI workflows.
- Consistent PDF output structure aligned with expected chapter and table formatting.

Chapter 5

Internship Task 3

5.1 Task 3: Problem Statement

The third internship task was to build a TeX-to-HTML converter for LCCA reports so that report content could be viewed in a browser without a LaTeX toolchain.

Direct conversion from LaTeX to HTML is not format-preserving. LCCA reports are table-heavy and rely on captions, section-based numbering, and math blocks. A naive conversion produced readable HTML but with structural defects in tables and captions.

The expected output was a small, reliable utility under `task3/` with:

- file and folder conversion,
- correct table captions and headings,
- math rendering support,
- predictable output beside the source `.tex` file,
- documented conversion limitations.

5.2 Task 3: Tasks Done

I implemented the converter in `task3/convert.py`, with usage notes in `task3/README.md` and limitations recorded in `task3/limitation.md`.

5.2.1 Task 3.1: Baseline conversion audit

I first ran a plain Pandoc conversion and recorded the main defects:

- `captionof table` blocks did not always become `<caption>` tags,
- key-value tables occasionally placed a data row in `<thead>`,
- section-style table numbering (Table 2-1, 2-2, ...) was missing.

5.2.2 Task 3.2: Pre-processing and post-processing

I added a pre-processing pass to normalize LaTeX patterns before Pandoc and a post-processing pass to fix table structure in HTML. This included wrapping `captionof` + `tabular` blocks into explicit table floats and repairing captions and headers after conversion.

5.2.3 Task 3.3: Numbering and CLI workflow

I implemented section-based table numbering by scanning `<h1>` and `<table>` elements in document order. The CLI supports single-file conversion, recursive folder conversion, optional TOC generation, and choice of math renderer.

5.2.4 Task 3.4: Validation and limitations

I validated the utility on `task3/texfile/LCCA_Report.tex` and documented known conversion limits in `task3/limitation.md` for layouts and packages that do not translate cleanly to HTML.

5.3 Task 3: Architecture and Workflow

The converter follows a compact pipeline:

- detect input files (single or recursive),
- pre-process LaTeX to stabilize tables and captions,
- run Pandoc with math rendering enabled,

- post-process HTML for captions, headers, and numbering.

5.4 Task 3: Python Code

5.4.1 Description of the Script

The script is split into a pre-processing stage, a Pandoc conversion call, and a post-processing stage that repairs HTML structure before saving output.

5.4.2 Python Code Snippet 1: TeX pre-processing

The pre-processor removes fragile wrappers and converts `captionof` blocks into explicit table floats.

Listing 5.1: Pre-processing TeX before Pandoc

```
def preprocess_tex(content: str) -> str:
    content = re.sub(
        r'\{\\footnotesize%?\s*'
        r'(\\begin\{tabular\}.*?\\end\{tabular\}%?)'
        r'\s*\}',
        lambda m: m.group(1),
        content,
        flags=re.DOTALL,
    )

    content = re.sub(
        r'\\noindent\\captionof\{table\}\{([^}]+)\}%?\s*'
        r'(?:\\vspace\{[^}]*\}%?\s*)?'
        r'(\\begin\{tabular\}.*?\\end\{tabular\}%?)',
        lambda m: (
            f'\\begin{{table}}[H]%\n\\caption{{{m.group(1)}}}%\n'
            f'{m.group(2)}\n\\end{{table}}%'
        ),
        content,
        flags=re.DOTALL,
    )
    return content
```

5.4.3 Python Code Snippet 2: End-to-end conversion

This function calls Pandoc and applies HTML post-processing before writing output.

Listing 5.2: End-to-end conversion flow

```
def convert_file(tex_path: Path, pandoc: str, toc: bool, math_renderer:
    str) -> Path:
    html_path = tex_path.with_suffix(".html")
    processed_tex = preprocess_tex(tex_path.read_text(encoding="utf-8")
        )

    tmp_tex = tex_path.parent / f"._{tex_path.stem}_pre.tex"
    tmp_tex.write_text(processed_tex, encoding="utf-8")

    cmd = [pandoc, str(tmp_tex), "-f", "latex", "-t", "html5", "--
        standalone", "-o", str(html_path)]
    cmd.append("--katex" if math_renderer == "katex" else "--mathjax")
    if toc:
        cmd.extend(["--toc", "--toc-depth=3"])

    subprocess.run(cmd, capture_output=True, text=True, timeout=120)
    tmp_tex.unlink(missing_ok=True)

    html = html_path.read_text(encoding="utf-8")
    html_path.write_text(postprocess_html(html), encoding="utf-8")
    return html_path
```

5.5 Outcome of Task 3

Task 3 delivered a compact TeX-to-HTML converter that preserves the essential structure of LCCA reports.

Key outcomes:

- usable CLI tool for single-file and batch conversion,
- corrected captions and headers in generated tables,
- section-based table numbering in HTML output,

Chapter 6

Internship Task 4

6.1 Task 4: Problem Statement

The fourth internship task addressed a critical distribution bottleneck for the `3psLCCA` desktop application: generating PDF reports without forcing end-users to install a LaTeX distribution.

The application, built with Python and PySide6, features a comprehensive Life Cycle Cost Analysis workflow. It captures bridge data, construction quantities, traffic projections, financial parameters, and carbon emissions, culminating in an interactive results dashboard. However, exporting these results to a professional PDF report required a TeX engine. Requiring civil engineers to install and configure a 1.3 GB distribution like MiKTeX or TeX Live was unacceptable from a user experience perspective.

The expected outcome was a self-contained LaTeX environment wrapper, `OsdagLatexEnv`, under the `task4/core/report-env/` module. The requirements were:

- **Zero-configuration deployment:** The application must discover bundled binaries automatically.
- **Multi-tier discovery:** Support environments ranging from local developer checkouts (repo-local assets) to production Conda deployments.
- **Dynamic environment injection:** Set `TEXMFHOME`, `TEXINPUTS`, and `PATH` dynamically at runtime so custom `.sty` files resolve correctly without global registry edits.

- **Crash-safe compilation:** Run `pdflatex` in a non-interactive mode that never hangs the PySide6 UI thread.
- **Developer ergonomics:** Provide a CLI tool to stage the heavy TeX binaries locally without polluting the git repository.

This task was a deep dive into environment integration, bridging a high-level Python GUI with a low-level, legacy binary toolchain.

6.2 Task 4: Tasks Done

I engineered the environment module from the ground up, separating the concerns into Python wrapper logic (`latex_env.py`), configuration constants, and a developer onboarding script (`setup_assets.py`).

6.2.1 Task 4.1: Designing a resilient asset discovery strategy

The primary challenge was that the TeX binaries live in different locations depending on how the application is executed. A developer running from source needs assets relative to the repository root, while a user running a built Conda package needs assets relative to `sys.prefix`.

To solve this, I designed a three-tier fallback mechanism in the `OsdagLatexEnv` class:

- **Tier 1 – Repo-local assets:** Checks `core/report-env/assets/<platform>/`. This is the primary target for developers who run the setup script.
- **Tier 2 – Conda prefix:** Scans the Conda share directory for `osdag_latex_env`, where production builds store the binaries.
- **Tier 3 – System PATH:** Uses `shutil.which("pdflatex")` as a last resort, allowing power users to opt into using their system-wide LaTeX installation.

This architecture guarantees that the application works flawlessly across development, testing, and production phases without any hardcoded absolute paths or manual configuration switches.

6.2.2 Task 4.2: Dynamic environment variable injection

Finding `pdflatex` is only half the battle; the executable must also know where to find packages, fonts, and temporary caching directories. I implemented a `configure_environment()` method to dynamically inject state into the OS environment just before compilation.

Crucially, this method maps:

- `TEXMFHOME` to the bundled `texmf-dist` directory.
- `TEXMFVAR` and `TEXMFCONFIG` to local paths to prevent permission errors when writing cached fonts.
- `TEXINPUTS` is prepended with the local `tex/latex` directory to ensure proprietary report styles take precedence over generic ones.
- `PATH` is modified at runtime to include the specific binary directory for the current operating system architecture (e.g., `x86_64-windows`).

This sandboxing ensures that our report compilation is completely decoupled from any existing, potentially conflicting system configurations.

6.2.3 Task 4.3: Automating developer asset staging

Because the raw LaTeX binaries sum to roughly 1.3 GB, committing them to the `3psLCCA` git repository was out of the question. I authored `setup_assets.py`, a dedicated Python CLI tool using `argparse` to solve developer onboarding.

The script safely copies the platform-specific binaries from an external git submodule (`osdag-latex-env`) into the `core/report-env/assets/` directory, which is heavily protected by `.gitignore`. I added auto-detection for Windows, Linux, and macOS platforms, ensuring that a developer only pulls the binaries relevant to their hardware. A `--status` flag was also implemented to quickly audit the integrity of the staged environment.

6.2.4 Task 4.4: Orchestrating the compilation pipeline

To generate a reliable PDF, LaTeX often requires multiple passes to resolve cross-references and table of contents numbering. I designed the `compile_tex` method to handle this orchestration robustly.

It constructs a `subprocess` call using `-interaction=nonstopmode`. This flag is vital: it prevents `pdflatex` from pausing and waiting for console input on errors, which would permanently freeze the application's GUI thread. The method iterates through the specified number of compilation runs, piping output quietly unless a critical failure occurs, and finally verifies the existence of the generated `.pdf` before yielding control back to the caller.

6.3 Task 4: Architecture and Workflow

The integration layer operates strictly across three decoupled phases, ensuring that UI code remains clean and unaware of LaTeX internals.

- **Discovery Phase:** Instantiating `OsdagLatexEnv` immediately triggers `_detect_tex_root` and `_detect_bin_dir`. The class caches the validated executable paths.
- **Configuration Phase:** Before any subprocess is spawned, `configure_environment` builds a strict sandbox via `os.environ`, scoping `TEXMF` and `PATH` specifically to the discovered assets.
- **Execution Phase:** `compile_tex` locks the execution context, fires two sequential non-blocking passes of `pdflatex`, and handles artifact verification.

For a new developer, the workflow is straightforward: clone the repository, run the setup script, and start generating PDFs. For the end-user, the workflow is invisible: they click "Generate Report" in the PySide6 UI, and the embedded Conda package handles the heavy lifting in the background.

6.4 Task 4: Python Code

6.4.1 Description of the Module

The implementation relies heavily on Python's `pathlib` for cross-platform path resolution and `subprocess` for safe execution. The codebase is heavily logged to provide audit trails for the discovery tier mechanism.

6.4.2 Python Code Snippet 1: OsdagLatexEnv class initialization and discovery

This excerpt demonstrates how the environment discovers its roots at instantiation, immediately securing references to the core executables.

Listing 6.1: OsdagLatexEnv discovery and initialization

```
class OsdagLatexEnv:
    def __init__(self) -> None:
        self.__system = platform.system().lower()
        self.__machine = platform.machine().lower()

        self.tex_root: Path | None = self._detect_tex_root()
        self.bin_dir: Path | None = self._detect_bin_dir()
        self.pdflatex: Path | None = self._get_executable("
           pdflatex")
        self.bibtex: Path | None = self._get_executable("bibtex")
        self.makeindex: Path | None = self._get_executable("
           makeindex")

        if self.tex_root:
            logger.info("TeX root   : %s", self.tex_root)
        if self.bin_dir:
            logger.info("TeX bindir: %s", self.bin_dir)
        if self.pdflatex:
            logger.info("pdflatex   : %s", self.pdflatex)
```

6.4.3 Python Code Snippet 2: TeX root detection across installation contexts

This specific logic enforces the three-tier hierarchy, branching gracefully between developer (repo-local) and production (Conda) environments.

Listing 6.2: Three-tier TeX root detection

```
def _detect_tex_root(self) -> Path | None:
```

```
    # 1. Repo-local
    plat = self._platform_asset_dir()
    if plat and (plat / "texmf-dist").exists():
        return plat

    # 2. Conda prefix
    prefix = Path(sys.prefix)
    if self.__system == "windows":
        conda_dir = prefix / "Library" / "share" / "
           osdag_latex_env"
    else:
        conda_dir = prefix / "share" / "osdag_latex_env"
    if conda_dir.exists() and (conda_dir / "texmf-dist").exists()
       :
        return conda_dir

    return None
```

6.4.4 Python Code Snippet 3: Environment variable configuration

Here, the operating system's environment is dynamically overridden to sandbox the LaTeX compilation process entirely within the bundled assets.

Listing 6.3: Configuring TEXMFHOME, TEXINPUTS, and PATH

```
def configure_environment(self) -> None:
    if not self.tex_root or not self.bin_dir:
        logger.warning(
            "Cannot configure environment -- TeX root or bin_dir
               missing."
        )
        return

    texmf_dist = str(self.tex_root / "texmf-dist")
```

```
9
10      os.environ["TEXMFHOME"] = texmf_dist
11      os.environ["TEXMFVAR"] = str(self.tex_root / "texmf-var")
12      os.environ["TEXMFCONFIG"] = str(self.tex_root / "texmf-config
            ")
13
14      current_path = os.environ.get("PATH", "")
15      bin_str = str(self.bin_dir)
16      if bin_str not in current_path:
17          os.environ["PATH"] = bin_str + os.pathsep + current_path
18
19      sty_base = (self.tex_root / "texmf-dist" / "tex" / "latex").
            as_posix()
20      sep = ";" if self.__system == "windows" else ":"
21      pkg_dirs = [f"{sty_base}//"]
22      existing = os.environ.get("TEXINPUTS", "")
23      os.environ["TEXINPUTS"] = sep.join(pkg_dirs) + sep + existing
24
25      logger.info("LaTeX environment configured.")
```

6.4.5 Python Code Snippet 4: Two-pass TeX compilation with error handling

This method represents the single integration point for the UI, ensuring compilation is non-blocking and artifacts are thoroughly validated.

Listing 6.4: compile method with nonstopmode and post-run validation

```
1 def compile_tex(
2     self,
3     tex_path: str | Path,
4     output_dir: str | Path | None = None,
5     *,
6     runs: int = 2,
7     quiet: bool = True,
```

```
) -> Path:
    if not self.pdflatex:
        raise FileNotFoundError(
            "pdflatex not found. Ensure the LaTeX assets are present in "
            "core/report-env/assets/ or install a TeX distribution."
        )

    self.configure_environment()

    tex_path = Path(tex_path).resolve()
    if output_dir is None:
        output_dir = tex_path.parent
    else:
        output_dir = Path(output_dir).resolve()
        output_dir.mkdir(parents=True, exist_ok=True)

    cmd = [
        str(self.pdflatex),
        "-interaction=nonstopmode",
        f"-output-directory={output_dir}",
    ]
    if quiet:
        cmd.append("-quiet")
    cmd.append(str(tex_path))

    for i in range(runs):
        logger.info("pdflatex pass %d/%d ...", i + 1, runs)
        subprocess.run(cmd, check=True, cwd=str(tex_path.parent))

    pdf_name = tex_path.with_suffix(".pdf").name
    pdf_path = output_dir / pdf_name
    if not pdf_path.exists():
```

```
40          raise FileNotFoundError(f"Expected PDF not found: {
                pdf_path}")
41
42      logger.info("PDF generated: %s", pdf_path)
43      return pdf_path
```

6.4.6 Python Code Snippet 5: Setup assets for developer asset staging

This CLI utility simplifies onboarding by parsing arguments and safely mirroring the heavy binary assets out of the external submodule.

Listing 6.5: Developer asset setup script

```
1  def copy_platform(source: Path, platform_key: str) -> None:
2      src = source / platform_key
3      if not src.exists():
4          print(f"  Warning: Source not found -- skipping: {src}")
5          return
6
7      dst = TARGET_DIR / platform_key
8      if dst.exists():
9          print(f"  Already present: {dst}")
10         return
11
12     print(f"  Copying {platform_key} assets ...")
13     print(f"      {src}  ->  {dst}")
14     shutil.copytree(src, dst)
15     print(f"  Done ({platform_key})")
16
17
18 def main() -> None:
19     parser = argparse.ArgumentParser(
20         description="Set up LaTeX assets for Osdag's self-
               contained TeX environment.",
```

```
    )
    parser.add_argument(
        "--platform",
        choices=["win", "linux", "mac", "all", "auto"],
        default="auto",
        help="Which platform assets to install (default: auto-
            detect).",
    )
    parser.add_argument(
        "--source",
        type=Path,
        default=DEFAULT_SOURCE,
        help=f"Source directory of platform assets (default: {
            DEFAULT_SOURCE}).",
    )
    parser.add_argument(
        "--status",
        action="store_true",
        help="Only print current asset status, do not copy
            anything.",
    )
    args = parser.parse_args()

    TARGET_DIR.mkdir(parents=True, exist_ok=True)

    if args.status:
        check_status()
        return

    if not args.source.exists():
        print(f"Source asset directory not found: {args.source}")
        sys.exit(1)

    if args.platform == "all":
```

```
52          targets = ["win", "linux", "mac"]
53      elif args.platform == "auto":
54          key = current_platform_key()
55          targets = [key] if key else []
56      else:
57          targets = [args.platform]
58
59      for key in targets:
60          copy_platform(args.source, key)
61
62      check_status()
63      print("Setup complete!")
```

6.5 Outcome of Task 4

Task 4 successfully eliminated the most significant friction point in the `3psLCCA` reporting pipeline by delivering a robust, invisible, self-contained LaTeX environment.

The immediate technical outcomes included:

- **A seamless GUI integration:** The PySide6 application can now issue a simple `env.compile_tex()` call and reliably receive a PDF, without ever blocking the UI thread or confronting the user with a console window.
- **Intelligent Discovery:** The three-tier (local, Conda, PATH) asset discovery protocol eliminated environment-specific hardcoding, ensuring stability across Windows, macOS, and Linux.
- **Dynamic Sandboxing:** On-the-fly manipulation of `TEXMFHOME`, `TEXINPUTS`, and `PATH` ensured zero cross-contamination with any pre-existing LaTeX software on the user's machine.
- **Smooth Developer Experience:** The `setup_assets.py` tool abstracted away the management of the 1.3 GB binary dependencies, keeping the core git repository lightweight and fast.

Chapter 7

Internship Task 5

7.1 Task 5: Problem Statement

The fifth internship task focused on the V2 Modular and Provenance report pipeline for 3psLCCA, along with a polished GUI for report customization. The existing report flow was monolithic and difficult to adapt. The requirement was to modularize report generation into isolated builders, allow selective section output, and fix LaTeX compilation issues caused by special characters and missing packages.

This task therefore required both backend and UI changes:

- **Modular report generation:** Replace the V1 monolithic flow with V2 modular builders per section.
- **Missing tables and environmental module:** Complete input, traffic, and environmental tables to match V1 coverage.
- **Provenance-friendly configuration:** Drive sections from config keys tied to GUI selection.
- **LaTeX stability fixes:** Handle Unicode symbols and missing package dependencies.
- **GUI customization:** Provide a professional tree-based dialog for section selection and export control.
- **Layout improvements:** Reduce wasted whitespace caused by aggressive `needspace` usage.

This task was not just a UI enhancement. It re-architected the report system to be modular, extensible, and robust under real user inputs.

7.2 Task 5: Tasks Done

I implemented the V2 modular report system under `task5/src/three_ps_lcca_gui/report/` and updated the GUI export workflow in `task5/src/three_ps_lcca_gui/gui/components/outputs/`.

7.2.1 Task 5.1: Modular export template and config map

The V2 system starts by extracting data into a flat, serializable dictionary. I implemented this in `mod_lcca_template.py` using the `LCCATemplate` class. It consolidates input, computed, and result data and exposes a `get_config()` method that returns all section toggles with defaults set to `True`.

This allowed the GUI to override only the sections a user disables while keeping a complete report when no customization is required.

7.2.2 Task 5.2: Input data module completion

The input module was extended to match all V1 tables. Two key additions were:

- **LCC Assumptions (Table 2-4)** using a 3-column layout for parameter, value, and reference basis.
- **Use Stage Details (Table 2-5)** using a 2-column key-value layout via `add_kv_table`.

These were implemented in `mod_input_data.py` and tied to the `KEY_SHOW_*` flags so they can be toggled independently in the UI.

7.2.3 Task 5.3: Traffic data module completion

The traffic section was reorganized under a unified `Subsection("Traffic data")` in `mod_traffic_data.py`. I added the missing traffic cost tables to match the reference report:

- **Tyre Cost (Table 2-11)** with a 4-column structure.

- **Fuel, Oil and Grease (Table 2-12)** as a key-value table.
- **New Vehicle Cost (Table 2-13)** with a 3-column structure.

This ensured the V2 modular report preserved the traffic cost model from V1.

7.2.4 Task 5.4: Environmental data module and sequencing

The environmental group did not exist in V2, so I created `mod_environmental_data.py` and added six conditional tables (Tables 2-14 through 2-19). The module includes Social Cost of Carbon, Material Emission Factors, Use Stage Emissions, Vehicle Emission Factors, Transport Emissions, and On-site Emissions.

I also aligned the table order with the V1 monolithic sequence by ensuring Transport Emissions is Table 2-18 and On-site Emissions is Table 2-19. This required updating both the constants map and the environmental module order.

7.2.5 Task 5.5: LaTeX compilation fixes (Unicode and packages)

Two LaTeX-level issues were fixed in `mod_base_report.py`:

- Unicode symbols were mapped using `\DeclareUnicodeCharacter` for U+20B9 (Rupee) and U+2082 (CO2 subscript).
- The `enumitem` package was added to prevent list-formatting errors in Appendix B.

These fixes prevented compilation failures for real-world data that includes currency symbols and subscripted chemical formulas.

7.2.6 Task 5.6: Report customization dialog and UI integration

The GUI export flow was upgraded to use a hierarchical tree dialog instead of a flat checkbox list. In `report_section_dialog.py`, I implemented dynamic titles and subtitles based on mode, created a themed `SectionTreeWidget`, and connected selection state to export configuration.

The Outputs page now launches the dialog using `ReportSectionDialog(..., mode="provenance")` so that the same UI supports both standard and provenance report generation.

7.2.7 Task 5.7: Spacing and layout optimization

The V2 report used aggressive `\needspace{12\baselineskip}` before most tables, which produced large blank areas. I parameterized `need_lines` in `add_kv_table` and `add_multi_table` inside `mod_base_report.py`, reducing defaults to `6`. Individual environmental tables were further tightened to `5` or `6` lines.

This yielded a much denser, print-ready layout without sacrificing readability.

7.3 Task 5: Architecture and Workflow

The V2 report pipeline follows a clear modular flow:

1. Export a project dictionary from the main controller.
2. `LCCATemplate` flattens data and builds a default config map.
3. The GUI dialog modifies the config keys based on user selections.
4. `mod_provenance_generate.py` constructs `LCCAReportLatex`, adds each module sequentially, and compiles the PDF.
5. The final PDF is saved through the worker thread without blocking the UI.

This architecture decouples data extraction, LaTeX rendering, and UI selection while maintaining consistent output fidelity.

7.4 Task 5: Python Code

7.4.1 Description of the Module

The V2 system is split into focused modules: a template extractor, section builders for input and traffic data, a new environmental module, and an orchestrator that generates the final PDF. UI components read the same constants map that drives the builders, preventing drift between report output and customization controls.

7.4.2 Python Code Snippet 1: Template configuration defaults

The template returns a consistent config map, enabling full report generation by default.

Listing 7.1: Default config map for modular V2 report

```
def get_config(self) -> Dict[str, bool]:
    return {
        KEY_SHOW_TITLE_PAGE:         True,
        KEY_SHOW_BRIDGE_DESC:        True,
        KEY_SHOW_FINANCIAL:          True,
        KEY_SHOW_CONSTRUCTION:       True,
        KEY_SHOW_LCC_ASSUMPTIONS:    True,
        KEY_SHOW_USE_STAGE:          True,
        KEY_SHOW_AVG_TRAFFIC:        True,
        KEY_SHOW_ROAD_TRAFFIC:       True,
        KEY_SHOW_PEAK_HOUR:          True,
        KEY_SHOW_HUMAN_INJURY:       True,
        KEY_SHOW_VEHICLE_DAMAGE:     True,
        KEY_SHOW_TYRE_COST:          True,
        KEY_SHOW_FUEL_OIL:           True,
        KEY_SHOW_NEW_VEHICLE:        True,
        KEY_SHOW_SOCIAL_CARBON:      True,
        KEY_SHOW_MATERIAL_EMISSION:  True,
        KEY_SHOW_USE_EMISSION:       True,
        KEY_SHOW_VEHICLE_EMISSION:   True,
        KEY_SHOW_ONSITE_EMISSION:    True,
        KEY_SHOW_TRANSPORT_EMISSION: True,
        KEY_SHOW_LCCA_RESULTS:       True,
    }
```

7.4.3 Python Code Snippet 2: Modular report orchestration

The generator builds the document in ordered modules and compiles the PDF.

Listing 7.2: V2 report generation pipeline

```
def generate_report(output_filename="LCCA_Report", export_dict=None,
    config_override=None, output_dir=None):
    template = LCCATemplate(export_dict)
    config = template.get_config()
```

```
    if config_override:
        config.update(config_override)
    data = template.get_report_data()

    doc = LCCAReportLatex()
    if config.get(KEY_SHOW_TITLE_PAGE, True):
        data["report_title"] = "LCCA Modular Report"
        add_title_page(doc, config, data)

    add_input_data(doc, config, data)
    add_traffic_data(doc, config, data)
    add_environmental_data(doc, config, data)

    if config.get(KEY_SHOW_LCCA_RESULTS, True):
        add_lcca_results(doc, config, data)

    add_full_appendix(doc)
    doc.save_latex(filename=output_filename, output_dir=output_dir)
    return doc.generate_pdf_output(filename=output_filename, output_dir
        =output_dir)
```

7.4.4 Python Code Snippet 3: Environmental table ordering

The transport emissions table is rendered before the on-site emissions table.

Listing 7.3: Environmental emissions table sequence

```
# Table 2-18: Transport Emissions
if config.get(KEY_SHOW_TRANSPORT_EMISSION, True):
    doc.append(NoEscape(r"\noindent\captionof{table}{Data for emissions
        related to transportation of material}"))
    ...

# Table 2-19: On-site Emissions
if config.get(KEY_SHOW_ONSITE_EMISSION, True):
    doc.append(NoEscape(r"\noindent\captionof{table}{Emissions from on-
        site activities during construction}"))
    ...
```

7.4.5 Python Code Snippet 4: Unicode and package fixes

Unicode fallbacks and `enumitem` were added at the base report level.

Listing 7.4: Unicode fixes and enumitem package

```
self.packages.append(Package("enumitem"))

self.preamble.append(NoEscape(r"""
\DeclareUnicodeCharacter{20B9}{Rs.}
\DeclareUnicodeCharacter{2082}{\textsubscript{2}}
"""))
```

7.4.6 Python Code Snippet 5: Report dialog title and subtitle

The dialog changes its header based on mode to support provenance export.

Listing 7.5: Dynamic dialog title and subtitle

```
title_text = "Report Customization" if self._mode == "standard" else "
    Data Provenance Report V2"
subtitle_text = (
    "Select the sections and data tables to include in your final LCCA
        PDF report."
    if self._mode == "standard"
    else "Select the sections and data tables to include in your final
        modular LCCA PDF report."
)
```

7.4.7 Python Code Snippet 6: Outputs page integration

The Outputs page launches the new dialog in provenance mode.

Listing 7.6: Launching the provenance report dialog

```
dlg = ReportSectionDialog(
    export_dict=self._build_export_dict(),
    mode="provenance",
    parent=self,
)
dlg.exec()
```

7.5 Outcome of Task 5

Task 5 delivered a complete V2 modular report system with GUI-based customization. Key outcomes:

- A modular report pipeline that mirrors V1 output while enabling selective section rendering.
- A new environmental module with six emission tables and correct sequencing.
- Robust LaTeX compilation with Unicode fallbacks and required package imports.
- A polished, tree-based report selection dialog integrated into the Outputs page.
- Reduced whitespace and improved layout stability through tuned `needspace` usage.

This task elevated the reporting system into a maintainable, extensible architecture aligned with real-world data and user-driven customization.

Chapter 8

Conclusions

8.1 Tasks Accomplished

1. **Screening Task – Automated Beam Analysis Report**

   - Built a Python-based reporting pipeline using `PyLaTeX` that validates engineering input data from Excel.
   - Generated dynamic LaTeX reports (`beam_analysis_report.pdf`) complete with calculated results and automated TikZ/pgfplots for Shear Force and Bending Moment Diagrams.
   - Established a fail-safe compilation process with structured cleanup of temporary auxiliary files.

2. **Internship Task 1 – CLI-based LCCA Report Generator**

   - Developed a robust, headless Python script (`lcca_report_generator.py`) to produce Life Cycle Cost Analysis reports matching Osdag's reference styling.
   - Decoupled raw bridge/financial data from LaTeX formatting using a structured JSON-like configuration system.
   - Implemented complex, multi-page tabular data formatting with consistent section-based table numbering (e.g., Table 2-1 to 2-17).

3. **Internship Task 2 – Interactive GUI Workflow for LCCA**

- Created a PySide6 desktop interface (`lcca_gui.py`) to allow interactive selection of report sections without editing code.
- Designed a dynamic, three-level `QTreeWidget` that translates report hierarchy maps into a visual selection tree.
- Built recursive parent-child checkbox state propagation to ensure logical consistency in user selections.
- Bridged the UI state directly into the backend generator via `KEY_SHOW_*` configuration flags, successfully generating customized PDFs on demand.

4. **Internship Task 3 – TeX-to-HTML Converter**

   - Built a command-line conversion utility (`convert.py`) utilizing Pandoc and BeautifulSoup4 to transform tabular LaTeX reports into web-ready HTML.
   - Engineered a robust pre-processing pipeline to normalize complex LaTeX patterns (e.g., `raggedright`, `captionof`) before Pandoc translation.
   - Implemented semantic HTML DOM post-processing to rescue orphan captions, fix false table headers, and restore section-based table numbering.
   - Documented system limitations and conversion boundaries for TeX layout capabilities.

5. **Internship Task 4 – Self-Contained LaTeX Environment Integration**

   - Integrated the `OsdagLatexEnv` module into the 3psLCCA desktop application to enable PDF generation without requiring a system-level TeX installation.
   - Engineered a resilient three-tier discovery system to locate 1.3 GB of TeX binaries across repo-local, Conda prefix, and system PATH environments.
   - Implemented dynamic environment sandboxing by injecting `TEXMFHOME`, `TEXINPUTS`, and `PATH` at runtime.
   - Developed `setup_assets.py`, a Python CLI tool for automated developer staging of the heavy TeX dependencies.

6. **Internship Task 5 – V2 Modular and Provenance Report System**

- Built a modular V2 report pipeline with section-level toggles and a template-driven export dictionary for provenance reporting.
- Completed the missing input, traffic, and environmental tables to match V1 report coverage (Tables 2-4 through 2-19).
- Implemented Unicode fallbacks and package fixes to prevent LaTeX compile failures on real-world data.
- Replaced the flat export dialog with a polished tree-based report selection UI integrated into the Outputs page.
- Optimized `needspace` usage to reduce empty page gaps and improve report density.

8.2 Skills Developed

LaTeX and TeX System Architecture

- Gained deep understanding of LaTeX compilation mechanics, cross-referencing, and auxiliary file management.
- Learned how to decouple LaTeX templating from raw data using Python.
- Mastered dynamic environment manipulation (`TEXMFHOME`, `TEXINPUTS`) to control package and font discovery without system-wide installations.

Python and PySide6 UI Development

- Built interactive desktop applications bridging complex backend engines with user-friendly interfaces.
- Implemented complex tree structures (`QTreeWidget`) with recursive state management and signal handling.
- Mastered `subprocess` execution for safely calling legacy CLI tools (like `pdflatex` and Pandoc) without hanging GUI threads.

Document Parsing and Transformation

- Gained expertise in manipulating document semantics using Regular Expressions for TeX pre-processing.
- Mastered DOM manipulation using `BeautifulSoup4` to enforce structural correctness in generated HTML.
- Learned to navigate the boundaries and limitations of format conversion tools like Pandoc.

Software Engineering and Automation

- Developed modular, class-based architectures to separate UI, data, and report generation logic.
- Built developer tooling (e.g., `setup_assets.py`) utilizing `argparse` and `shutil` for automated environment bootstrapping.
- Improved technical writing skills by documenting software architectures, workflow pipelines, and format limitations.

Modular Architecture and UI Integration

- Designed provenance-driven report modules with clean separation between template extraction and LaTeX rendering.
- Integrated tree-based section selection into the GUI to align user choices with backend report builders.
- Debugged layout spacing and LaTeX compilation stability across modular pipelines.

Chapter A

Appendix

A.1 Work Reports

Date	Day	Description of Work	Hours
16/02/25	Sunday	FOSSEE Fellowship Orientation Session. Introduction to workflows and repository setup.	4
17/02/25	Monday	Repository cloning, environment setup, and study of existing Osdag architecture.	4
18/02/25	Tuesday	Reviewed LCCA report requirements; explored PyLaTeX package capabilities.	4
19/02/25	Wednesday	Drafted initial LCCA report template and set up data validation scaffolding.	4
20/02/25	Thursday	Prototyped table layouts and formatting for LCCA report sections.	4
21/02/25	Friday	Refined LCCA report generation flow and tested error handling.	4
22/02/25	Saturday	Code cleanup and documentation of the initial LCCA report generator.	5
23/02/25	Sunday	WEEKLY OFF	0

Continued on next page

Table A.1 – continued from previous page

Date	Day	Description of Work	Hours
24/02/25	Monday	Reviewed feedback and began research on LCCA report format for Task 1.	4
25/02/25	Tuesday	Set up static LaTeX template replicating the LCCA reference document.	4
26/02/25	Wednesday	Developed lcca_report_generator.py to decouple data from LaTeX format.	4
27/02/25	Thursday	Implemented dynamic section and subsection formatting.	4
28/02/25	Friday	Debugged table column widths and table spanning across pages.	4
01/03/25	Saturday	Refined LCCA report layout; generated baseline PDF for review.	3
02/03/25	Sunday	WEEKLY OFF	0
03/03/25	Monday	Integrated JSON configuration system for dynamic report tables.	4
04/03/25	Tuesday	Implemented sequential table numbering logic (e.g. Table 2-1 to 2-17).	4
05/03/25	Wednesday	Finalized Task 1 CLI generator; validated against multiple datasets.	4
06/03/25	Thursday	Task 1 code review and started architecture planning for Task 2 GUI.	4
07/03/25	Friday	Built initial PySide6 QTreeWidget for report section selection.	4
08/03/25	Saturday	Mapped SECTION_MAP and SUBSECTION_TABLE_MAP into visual tree nodes.	4
09/03/25	Sunday	WEEKLY OFF	0
10/03/25	Monday	Implemented recursive parent-child checkbox state propagation in PySide6.	4

Continued on next page

Table A.1 – continued from previous page

Date	Day	Description of Work	Hours
11/03/25	Tuesday	Resolved signal recursion issues during tree updates.	4
12/03/25	Wednesday	Bridged GUI tree selection state to KEY_SHOW_* flags for generator backend.	5
13/03/25	Thursday	Added threading to prevent PySide6 UI freeze during LaTeX compilation.	4
14/03/25	Friday	Tested GUI-based report generation and fixed edge cases in partial selections.	5
15/03/25	Saturday	Task 2 finalization and review meeting with mentor.	4
16/03/25	Sunday	WEEKLY OFF	0
17/03/25	Monday	Started Task 3: Researched TeX to HTML conversion tools; evaluated Pandoc.	4
18/03/25	Tuesday	Set up convert.py and ran baseline Pandoc conversions on LCCA reports.	4
19/03/25	Wednesday	Identified HTML table structural defects; developed RegExp pre-processor.	4
20/03/25	Thursday	Removed unsupported LaTeX packages and minipage wrappers programmatically.	4
21/03/25	Friday	Integrated BeautifulSoup4 for post-processing Pandoc HTML output.	4
22/03/25	Saturday	Repaired orphan caption paragraphs and false table headers.	4
23/03/25	Sunday	WEEKLY OFF	0
24/03/25	Monday	Implemented automated section-based HTML table numbering (Table 2-1...).	4
25/03/25	Tuesday	Added batch conversion support and MathJax/KaTeX integration.	4

Continued on next page

Table A.1 – continued from previous page

Date	Day	Description of Work	Hours
26/03/25	Wednesday	Documented TeX-to-HTML conversion boundaries in limitation.md.	4
27/03/25	Thursday	Finalized Task 3 pipeline and presented generated HTML.	4
28/03/25	Friday	Started Task 4: Investigated standalone LaTeX distribution for 3psLCCA.	4
29/03/25	Saturday	Evaluated TinyTeX and TeX Live portable installation approaches.	3
30/03/25	Sunday	WEEKLY OFF	0
31/03/25	Monday	Designed OsdagLatexEnv module architecture and discovery strategy.	4
01/04/25	Tuesday	Implemented Tier 1 (repo-local) and Tier 2 (Conda prefix) TeX root detection.	4
02/04/25	Wednesday	Built platform-aware binary path resolution (Windows/Linux/Mac).	5
03/04/25	Thursday	Wrote configure_environment() method to inject TEXMFHOME and TEXINPUTS.	5
04/04/25	Friday	Successfully sandboxed pdflatex compilation within custom texmf tree.	4
05/04/25	Saturday	Resolved package discovery issues during compile_tex() execution.	4
06/04/25	Sunday	WEEKLY OFF	0
07/04/25	Monday	Implemented non-stop compilation mode to prevent subprocess hangs.	4
08/04/25	Tuesday	Developed setup_assets.py CLI script using argparse.	4
09/04/25	Wednesday	Added platform auto-detection and directory mirroring for TeX binaries.	4

Continued on next page

Table A.1 – continued from previous page

Date	Day	Description of Work	Hours
10/04/25	Thursday	Tested setup script on Windows and validated .gitignore protection.	4
11/04/25	Friday	Integrated OsdagLatexEnv directly into the 3psLCCA PySide6 frontend.	4
12/04/25	Saturday	Task 4 code review and pipeline stability testing.	4
13/04/25	Sunday	EXAM BREAK	0
14/04/25	Monday	EXAM BREAK	0
15/04/25	Tuesday	EXAM BREAK	0
16/04/25	Wednesday	EXAM BREAK	0
17/04/25	Thursday	EXAM BREAK	0
18/04/25	Friday	EXAM BREAK	0
19/04/25	Saturday	EXAM BREAK	0
20/04/25	Sunday	EXAM BREAK	0
21/04/25	Monday	EXAM BREAK	0
22/04/25	Tuesday	EXAM BREAK	0
23/04/25	Wednesday	EXAM BREAK	0
24/04/25	Thursday	EXAM BREAK	0
25/04/25	Friday	EXAM BREAK	0
26/04/25	Saturday	EXAM BREAK	0
27/04/25	Sunday	EXAM BREAK	0
28/04/25	Monday	Reviewed V2 modular report architecture and export dictionary flow.	4
29/04/25	Tuesday	Implemented LCC assumptions and use stage tables in mod_input_data.py.	4
30/04/25	Wednesday	Completed traffic cost tables (tyre, fuel, new vehicle) in mod_traffic_data.py.	4
01/05/25	Thursday	Created mod_environmental_data.py and added emission tables for V2.	4

Continued on next page

Table A.1 – continued from previous page

Date	Day	Description of Work	Hours
02/05/25	Friday	Swapped transport and on-site emission table order in constants and module.	4
03/05/25	Saturday	Added Unicode fixes (Rupee, CO2) and enumitem to base report.	3
04/05/25	Sunday	WEEKLY OFF	0
05/05/25	Monday	Implemented ReportSectionDialog with tree-based section selection.	4
06/05/25	Tuesday	Wired Outputs page to launch provenance report dialog.	4
07/05/25	Wednesday	Removed obsolete provenance_report_dialog and cleaned exports.	4
08/05/25	Thursday	Parameterized needspace defaults in mod_base_report.py.	4
09/05/25	Friday	Tuned needspace values in environmental tables to reduce whitespace.	4
10/05/25	Saturday	Validated V2 provenance export end-to-end with sample data.	3
11/05/25	Sunday	WEEKLY OFF	0
12/05/25	Monday	Verified LaTeX stability with Unicode and longtable content.	4
13/05/25	Tuesday	Documented V2 walkthrough and prepared task notes.	4
14/05/25	Wednesday	Integrated modular V2 workflow into report narrative.	4
15/05/25	Thursday	Refined Task 5 chapter content and code snippets.	4
16/05/25	Friday	Reviewed full report structure and prepared final edits.	4

Continued on next page

Table A.1 – continued from previous page

Date	Day	Description of Work	Hours
17/05/25	Saturday	Final report formatting and LaTeX styling adjustments.	3
18/05/25	Sunday	WEEKLY OFF	0
19/05/25	Monday	Compiled full report and resolved outstanding warnings.	4
20/05/25	Tuesday	Final proofreading and corrections before submission.	4
21/05/25	Wednesday	Integrated final feedback and verified report flow.	4
22/05/25	Thursday	Final formatting pass and consistency checks.	4
23/05/25	Friday	Prepared final submission bundle and reviewed output PDFs.	4
24/05/25	Saturday	Final compilation and submission of the FOSSEE Fellowship Report.	5

Chapter B

Bibliography

1. Siddhartha Ghosh, Danish Ansari, Ajmal Babu Mahasrankintakam, Dharma Teja Nuli, Reshma Konjari, M. Swathi, and Subhrajit Dutta. Osdag: A Software for Structural Steel Design Using IS 800:2007. In Sondipon Adhikari, Anjan Dutta, and Satyabrata Choudhury, editors, *Advances in Structural Technologies*, volume 81 of Lecture Notes in Civil Engineering, pages 219–231, Singapore, 2021. Springer Singapore.

2. FOSSEE Project. Osdag website. Available at: `https://osdag.fossee.in/`.

3. 3psLCCA Project. 3psLCCA GUI Repository. Available at: `https://github.com/3psLCCA/3psLCCA-gui`.

4. 3psLCCA Project. 3psLCCA Core Engine Repository. Available at: `https://github.com/3psLCCA/3psLCCA-core`.

5. Pandoc. A universal document converter. Available at: `https://pandoc.org/`.

6. Beautiful Soup. Python library for parsing HTML and XML. Available at: `https://www.crummy.com/software/BeautifulSoup/`.

7. PyLaTeX. Python library for creating LaTeX files. Available at: `https://jeltef.github.io/PyLaTeX/`.

8. PySide6 (Qt for Python). Available at: `https://doc.qt.io/qtforpython/`.