FOSSEE Summer Fellowship Report

On

Development of the 3psLCCA (Life Cycle Cost Assessment) Web Platform and Static WebAssembly Runtime for Osdag

Submitted by

Swayam Prakash Panda

4th Year B.Tech Student, Computer Science and Engineering (AIML)

School of Computing Science & Artificial Intelligence (SCAI)

VIT Bhopal University

Under the Guidance of

Prof. Siddhartha Ghosh

Department of Civil Engineering

Indian Institute of Technology Bombay

Mentor:

Swastik Gupta

July 2026

Acknowledgments

- I would like to express my sincere gratitude to everyone who has supported and guided me throughout the course of this fellowship.
- I extend my heartfelt thanks to my mentor, Swastik Gupta, for his continuous guidance, detailed code reviews, and technical direction throughout the development of the 3psLCCA web platform and WebAssembly runtime.
- I am also grateful to the project staff of the Osdag team — Ajmal Babu M. S., Ajinkya Dahale, and Parth Karia — for their support during the fellowship.
- I am deeply grateful to Prof. Siddhartha Ghosh, Principal Investigator of Osdag and faculty member of the Department of Civil Engineering at IIT Bombay, for his remarkable leadership and for the opportunity to contribute to the 3psLCCA project.
- I would also like to thank Prof. Kannan M. Moudgalya, Project Investigator of FOSSEE and faculty member of the Department of Chemical Engineering at IIT Bombay, for providing this valuable opportunity through the FOSSEE initiative.
- Special thanks are due to Usha Viswanathan and Vineeta Parmar, Managers at FOSSEE, along with their dedicated team, for their coordinated support.
- I gratefully acknowledge the support received from the National Mission on Education through Information and Communication Technology (ICT), Ministry of Education (MoE), Government of India, whose facilitation was critical to the execution of this project.
- I would like to thank my fellow intern Priyansh Vaish, who worked alongside me on the 3psLCCA web application. His collaboration on the input parameter pages,

traffic data tabs, and `.3ps` import support truly enriched the project and my experience.

- Lastly, I am thankful to VIT Bhopal University, the School of Computing Science & Artificial Intelligence, and my faculty for their constant encouragement during my studies and this fellowship.

Contents

Chapter 1

Introduction

1.1 National Mission in Education through ICT

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

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

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

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

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

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

1.1.1 ICT Initiatives of MoE

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

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

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

1.2 FOSSEE Project

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

1.2.1 Projects and Activities

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

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

1.2.2 Fellowships

FOSSEE offers various internship and fellowship opportunities for students:

- Winter Internship
- Summer Fellowship
- Semester-Long Internship

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

For more details, visit the official FOSSEE website.

Figure 1.1: FOSSEE Projects and Activities

1.3 Osdag Software

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

1.3.1 Osdag GUI

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

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

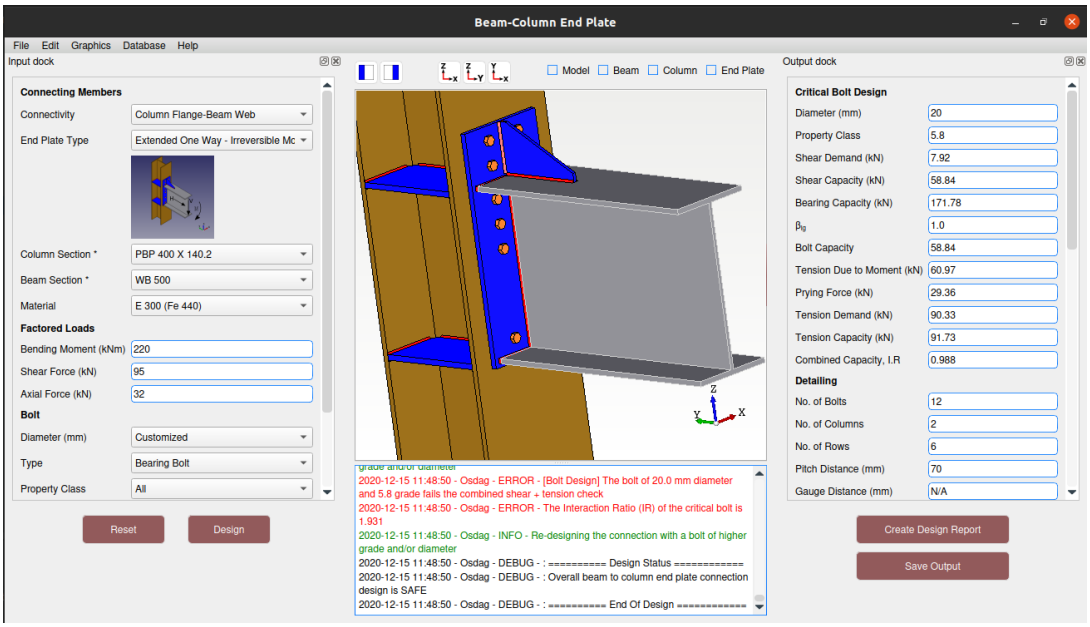

Figure 1.2: Osdag GUI

For more details, visit the official Osdag website.

1.4 The 3psLCCA Project

3psLCCA (3 Pillars of Sustainability Life Cycle Cost Assessment) is a FLOSS tool developed under the Osdag umbrella for the life cycle cost analysis (LCCA) of steel bridges. Rather than considering only the initial construction cost, 3psLCCA evaluates a bridge across its entire service life and across the three pillars of sustainability:

- **Economic**: initial construction, routine and major maintenance, inspection, repair and rehabilitation, bearing and expansion-joint replacement, reconstruction, and demolition costs, discounted to present value.
- **Social**: road user costs (vehicle operating cost, value of travel time, accident costs) incurred during construction and maintenance-related traffic disruption, computed from traffic composition and IRC-based models.
- **Environmental**: embodied and vehicular carbon emission costs, valued through the social cost of carbon.

1.4.1 Ecosystem Before the Fellowship

At the start of the fellowship, the 3psLCCA ecosystem consisted of three repositories:

- `3psLCCA-gui`: a PySide6 desktop application that provides the reference user experience, with sequential data input widgets (structure works, financial, carbon emission, bridge and traffic, maintenance, demolition), an SQLite persistence layer, results visualisation, and LaTeX report generation.
- `3psLCCA-core`: a pure-Python calculation engine exposing `run_full_lcc_analysis()`, which takes the project input dictionary, construction costs, and an optional Wholesale Price Index (WPI) dataset, and returns stage-wise results (`initial_stage`, `use_stage`, `reconstruction`, `end_of_life`) along with warnings and notes.
- `3psLCCA-web`: an early React web port of the desktop application, seeded by previous contributors, with the page skeletons in place but with broken home-screen flows, no user accounts, no reliable persistence, a schema that had drifted from the desktop application, and calculations that required a locally running FastAPI server.

1.4.2 Fellowship Objectives

The objective of this fellowship was to take the web application from that early state to production parity with the desktop application, specifically:

1. Stabilise the application shell and introduce authentication, cloud syncing, and offline-first project storage (Chapter 4).

2. Canonicalise the project schema page by page against the desktop application and harden the web-to-core adapter with strict validation and tests (Chapter 5).

3. Eliminate the calculation server by running `3psLCCA-core` directly in the browser as WebAssembly via Pyodide, with reference tests and CI on the core repository (Chapter 6).

4. Generate desktop-equivalent LaTeX design reports entirely in the browser using SwiftLaTeX (Chapter 7).

The screening task that preceded the fellowship, based on the OsdagBridge plate girder module, is described in Chapter 2.

Chapter 2

Screening Task

2.1 Problem Statement

The screening task for the fellowship was based on **OsdagBridge**, the bridge design module of the Osdag ecosystem. The plate girder module of OsdagBridge renders Shear Force Diagrams (SFD) and Bending Moment Diagrams (BMD) of the analysed bridge as interactive 3D Plotly figures embedded in the Qt desktop application. The task (*FOSSEE Plot Screening Task*) required refining this plotting subsystem:

- The SFD/BMD geometry was rendered as monolithic traces, so individual girders could not be isolated or inspected independently.
- Moment results had a colour-contoured view, but shear results did not, so the two force views were inconsistent.
- Plot controls (load case and force selection) lived inside the plot widget itself instead of the application's Output Dock, cluttering the main layout, and every control change forced a full backend re-render.
- The Max/Min value toggles were tied directly to the plot state and did not persist across redraws, and the plot occasionally rendered blank immediately after the design step completed.

The work was submitted as pull request #5 on the `Nidhikhare12/OsdagBridge` repository (branch `screening-task-plot`), adding 565 lines and removing 145 across exactly three files to avoid scope creep: `plots_widget.py`, `plots_UI.py`, and `output_dock.py`.

2.2 Tasks Done

2.2.1 Per-Girder Traces and Legend Groups

The SFD/BMD geometry construction in src/osdagbridge/core/bridge_types/plate_girder/plots_widget.py was decomposed so that each girder produces its own set of Plotly traces under a dedicated legendgroup. A custom legend click handler toggles grouped visibility, which means a girder can be isolated or hidden with a single click in the legend.

Listing 2.1: Per-girder vertical drop lines grouped by legendgroup

```
# vertical drop lines
for xi, zi, mzi, nid in zip(xs, zs, mz, node_ids):
    htext = f"Node {nid}<br>X={xi:.2f}<br>{force_key}={mzi:.2f}"
    fig.add_trace(go.Scatter3d(
        x=[xi, xi], y=[0, mzi * moment_scale], z=[zi, zi],
        mode="lines+markers",
        line=dict(width=4, color=[mzi, mzi], colorscale="Jet",
                  cmin=min(mzfull), cmax=max(mzfull)),
        marker=dict(size=12, opacity=0),
        text=[htext, htext], hoverinfo="text",
        legendgroup=girder_name, showlegend=False
    ))
```

2.2.2 Shear Contour View

A new build_figure_sfd_contour() function was implemented to bring the shear view to parity with the existing moment contour view. The global shear range is first collected across *all* girders so that the Jet colourscale is normalised consistently, and each drop line is then coloured by its shear magnitude.

Listing 2.2: Global shear range collection for consistent contour colouring

```
# collecting global shear range from all girders
vy_all = []
for elems in girders.values():
    xs, ys, zs, vy, _ = build_polyline(elems, comp_i, comp_j)
```

```
    vy_all.extend(vy)
vy_min, vy_max = min(vy_all), max(vy_all)

# vertical drop lines colored by shear magnitude
for xi, vyi in zip(xs, Vy):
    fig.add_trace(go.Scatter3d(
        x=[xi, xi], y=[0, vyi * shear_scale], z=[z_base, z_base],
        mode="lines+markers",
        line=dict(width=4, color=[vyi, vyi], colorscale="Jet",
                  cmin=vy_min, cmax=vy_max),
        marker=dict(size=12, opacity=0), hoverinfo="skip",
        legendgroup=girder_name, showlegend=False
    ))
```

2.2.3 Client-Side JavaScript Plot Controls

Native Plotly bindings were added through `QWebChannel` in `plots_UI.py` for a **grid toggle**, a **scale slider**, and a **girder isolation dropdown**. Because these controls are handled client-side inside the embedded web view, they update the figure instantly without triggering a backend re-render of the analysis.

2.2.4 Output Dock Integration

The legacy load case and force comboboxes were stripped from the plot UI and the analysis controls were migrated into the application's Output Dock. The force checkbox grid in `output_dock.py` is mapped to plot force keys through a `_FORCE_LABEL_MAP` dictionary and wired to setter methods on the plot widget:

Listing 2.3: Wiring Output Dock force checkboxes to the plot widget

```
# 1) Wiring the force checkbox grid
for cb in self.output_widget.findChildren(RichCheckBox):
    label_text = cb.text()
    for pattern, force_key in self._FORCE_LABEL_MAP.items():
        if pattern in label_text:
            cb.clicked.connect(
```

```
                lambda checked, fk=force_key: (
                    plot_widget.set_force(fk) if checked else
                        None
                )
            )
            break

# Default to first loadcase and Fy force
self._current_loadcase = loadcases[0] if loadcases else ""
self._current_force = "Fy"
```

2.2.5 Max/Min Toggle Persistence and Initial Render Fix

The Max/Min button states, previously coupled to the plot instance, were reconnected to the "Display Options" toggles in the dock so that they persist cleanly across redraws. A minor edge case, where the plot sat blank immediately after the design step finished, was resolved by explicitly invoking a generic `update_plot()` hook in the widget's `setup()` block.

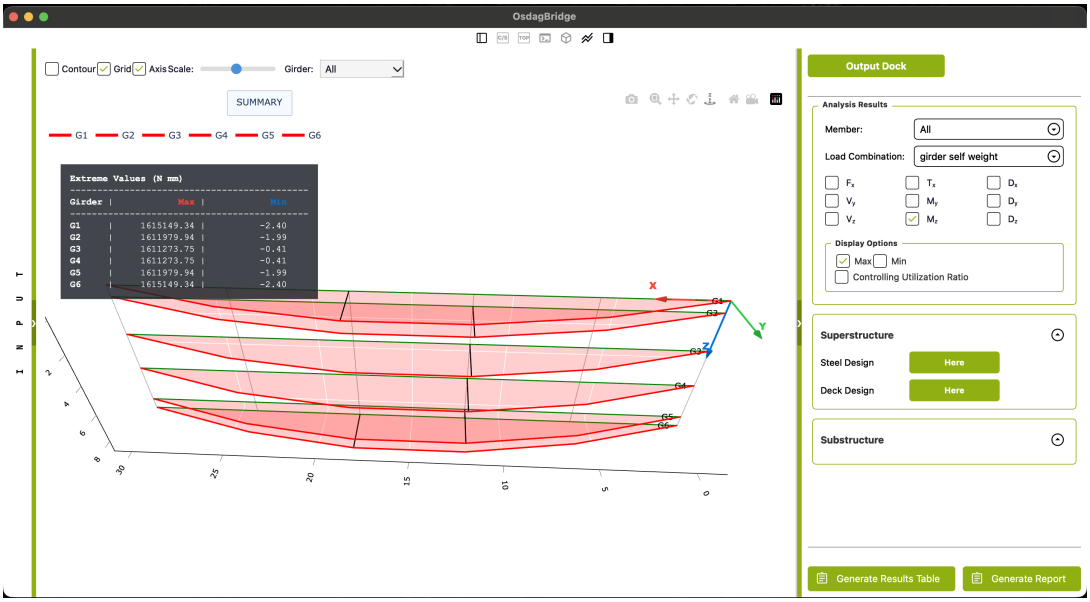

Figure 2.1: Plate girder plot after the screening task enhancements

2.3 Outcome

The pull request was reviewed with six commits and tested locally on macOS against a 30 m span configuration, verifying that no regressions were introduced outside the three modified files. The task demonstrated familiarity with the Osdag/OsdagBridge codebase and with PySide6–Plotly–`QWebChannel` interoperation, and led to selection for the FOSSEE Summer Fellowship 2026 on the 3psLCCA project.

Chapter 3

3psLCCA Web Application Architecture

3.1 Overview

The `3psLCCA-web` application is a single-page React application built with Vite. It reproduces the sequential data-entry workflow of the 3psLCCA desktop application (general information, construction / structure works data, financial data, carbon emission data, bridge and traffic data, maintenance and repair, demolition, recycling, and outputs) as browser pages, while the numerical analysis is delegated to the same `3psLCCA-core` Python engine used by the desktop application.

The technology stack is summarised below:

- **React 18 + Vite**: component model, routing via `react-router-dom`, and fast development builds.
- **Bootstrap / React-Bootstrap**: layout and form styling consistent across pages.
- **Appwrite**: authentication (email and Google OAuth) and the cloud project database (Chapter 4).
- **Pyodide**: CPython compiled to WebAssembly, used to run the `3psLCCA-core` wheel inside a Web Worker (Chapter 6).
- **SwiftLaTeX**: in-browser PdfTeX WebAssembly engine used for desktop-equivalent LaTeX report compilation (Chapter 7).

- **Recharts / D3**: results charts in the Outputs page.
- **ExcelJS, JSZip, pako**: construction data Excel import and `.3ps` project file import/export.
- **Playwright, ESLint**: end-to-end testing and linting.

A FastAPI backend remains in the repository, but after the WebAssembly migration it is only an *explicit development fallback*; production builds are fully static.

3.2 Complete Directory Structure

Listing 3.1: 3psLCCA-web repository layout (trimmed)

```
3psLCCA-web/
|
+-- index.html                  # SPA entry document
+-- vite.config.js              # Vite build configuration
+-- package.json                # Dependencies and npm scripts
+-- playwright.config.js        # End-to-end test configuration
+-- wasm-smoke.html             # WebAssembly parity smoke test
   page
+-- swiftlatex-report-smoke.html # In-browser LaTeX smoke test
   page
|
+-- scripts/
|   +-- prepare-wasm-assets.mjs      # Builds core wheel +
   Pyodide assets
|   +-- generate-native-reference.py # Native CPython reference
   outputs
|
+-- backend/                    # FastAPI dev fallback (optional)
|   +-- app/main.py
|   +-- app/adapters/web_to_core.py  # Re-exports the shared
   adapter
|   +-- tests/                  # Adapter and API tests
```

```
|
+-- public/
|   +-- data/                    # Static reference data
|   +-- vendor/swiftlatex/       # SwiftLaTeX engine + TeX Live
    footprint
|
+-- src/
    +-- main.jsx                 # React entry point
    +-- App.jsx                  # Router and top-level layout
    +-- contexts/
    |   +-- ProjectDataContext.jsx   # Shared project state
    +-- lib/
    |   +-- appwrite.js              # Appwrite client
        configuration
    |   +-- projectStorageService.js # Offline-first storage
        service
    |   +-- lccaApi.js               # Engine facade used by the
        UI
    |   +-- lccaEngine/
    |       +-- wasmEngine.js        # Main-thread WebAssembly
        engine API
    |       +-- lcca.worker.js       # Pyodide Web Worker
    |       +-- backendEngine.js     # FastAPI dev fallback
        client
    +-- wasm/python/
    |   +-- web_to_core.py           # Validation + mapping
        adapter
    |   +-- bridge.py                # JSON bridge exposed to
        Pyodide
    +-- utils/
    |   +-- projectCreation.js
    |   +-- constructionExcel.js     # Excel import/export
        helpers
    +-- data/
```

```
|   +-- wpi_db.json              # WPI database shipped to the engine
+-- gui/
    +-- Login/Loginpage.jsx
    +-- components/
        +-- Homepage.jsx         # Landing page
        +-- ProjectLayout.jsx    # Project workspace shell
        +-- Sidebar.jsx          # Left navigation tree
        +-- ProjectNavbar.jsx    # Top bar with save indicator
        +-- ProfileAvatar.jsx    # Account menu / logout
        +-- NewProjectModal.jsx / OpenProjectModal.jsx
        +-- ProjectInfoModal.jsx / RenameProjectModal.jsx
        +-- VersionHistoryModal.jsx / SettingsModal.jsx
        +-- global_info/         # General information page
        +-- constructionworkdata/
        |   +-- ConstructionWorkData.jsx
        |   +-- MaterialTable.jsx / MaterialAddModal.jsx
        |   +-- ImportPreviewModal.jsx   # Excel import preview
        |   +-- ConstructionTrash.jsx    # Global trash UI
        |   +-- foundation/ substructure/ superstructure/
        |       miscellaneous/
        +-- financialdata/FinancialData.jsx
        +-- carbon_emission/     # Material/machinery/traffic/
        |                        # transportation emissions + SCC
        +-- bridgedata/BridgeData.jsx
        +-- trafficdata/TrafficData.jsx
        +-- maintenance_and_repair/MaintenanceAndRepair.jsx
        +-- demolition/Demolition.jsx
        +-- recycling/Recycling.jsx
        +-- outputs/
```

```
72                  +-- Outputs.jsx           # Results, charts,
                        breakdowns
73                  +-- reportModel.js        # Report data model
74                  +-- latexReport.js        # Desktop-style .tex
                        generator
75                  +-- latexReportEngine.js  # SwiftLaTeX worker
                        bridge
```

3.3 Application Shell

3.3.1 Routing and Project Workspace

`App.jsx` defines the route tree: the login page, the homepage, and a project workspace route rendered by `ProjectLayout.jsx`. The workspace follows the desktop application's layout: a left `Sidebar` navigation tree of input parameter pages and outputs, a top `ProjectNavbar` carrying the project name and the dynamic save indicator, and a central content area in which the selected data page is mounted. Shared project state flows through `ProjectDataContext.jsx`, so every page reads and writes the same canonical project object (Chapter 5).

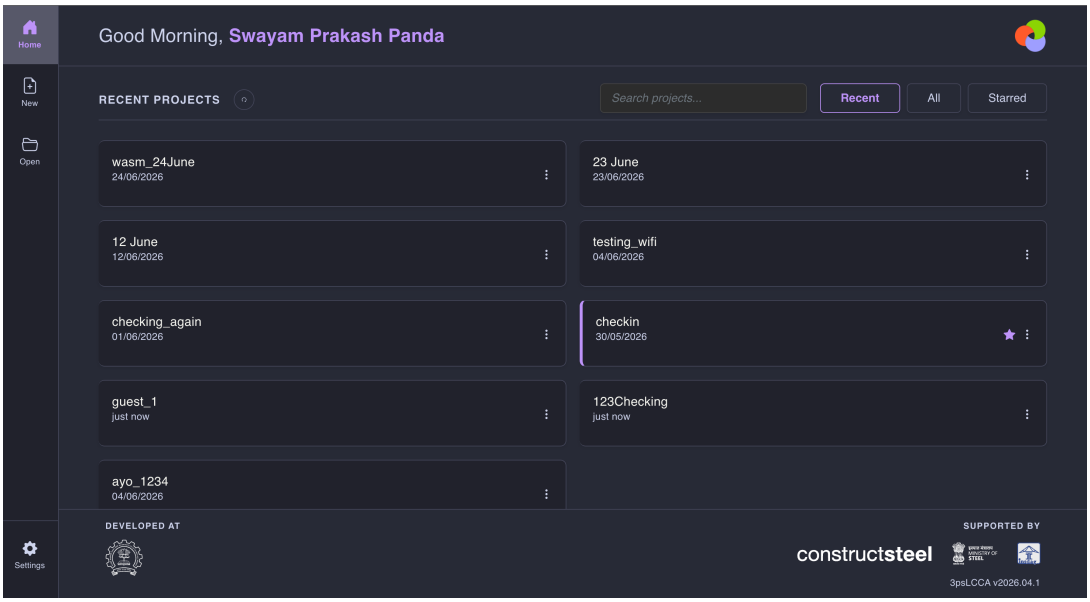

Figure 3.1: 3psLCCA web homepage

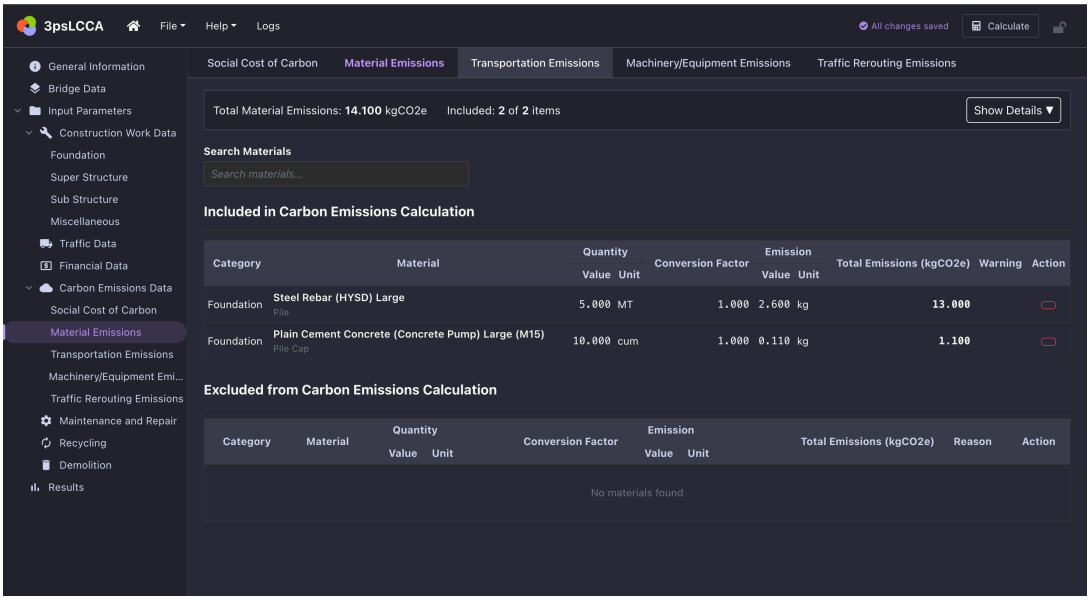

Figure 3.2: Project workspace layout (`ProjectLayout.jsx`)

3.3.2 Initial Stabilisation Work

The first phase of the fellowship (25 May – 1 June 2026) stabilised this shell, which was only partially functional in the inherited codebase:

- Fixed the homepage search bar, the Recent Projects list, and all primary action buttons, which previously did not respond or navigated incorrectly.
- Fixed the three-button project card actions and the profile functionality.
- Repaired sidebar dropdown behaviour (with Priyansh Vaish) and added focus guards so open menus close correctly.
- Fixed guest project pinning, currency auto-fill during new project creation, and the project details info modal.
- Added a scroll-lock fix for modals, a dynamic save indicator in the navbar, and resolved a refresh-overwrite bug that could clobber a newer save with stale in-memory state.

3.4 Engine Facade

All calculation requests from the UI go through a single facade, `src/lib/lccaApi.js`, which selects between the WebAssembly engine (default, Chapter 6) and the FastAPI development fallback (`backendEngine.js`). Pages therefore never talk to a transport directly; they call `calculateLcca(payload)` and receive the same result envelope regardless of where the Python engine actually ran. This separation is what later allowed the FastAPI server to be removed from production without touching any page component.

3.5 Summary

The web application is a Vite/React SPA whose pages mirror the desktop data-entry workflow, with project state held in a shared context, one storage service, one engine facade, and static vendor assets for Pyodide and SwiftLaTeX. The following chapters describe each of these subsystems.

Chapter 4

Authentication, Cloud Sync, and Offline-First Storage

4.1 Overview

The inherited web application had no notion of a user: projects lived in a fragile "checkpoint" mechanism in browser storage, could not follow a user across devices, and were easily lost. This chapter documents the identity and persistence layer added during the fellowship: Appwrite-backed authentication (email/password and Google OAuth), a cloud project database, a persistent guest mode, and an offline-first storage service with last-write-wins conflict resolution.

4.2 Appwrite Integration

4.2.1 Client Configuration

File Location: `src/lib/appwrite.js`

Appwrite was chosen as the backend-as-a-service because it is open source, self-hostable, and provides authentication and a document database from a single SDK. The client is configured once from environment variables and exports the `Account` and `Databases` services used by the rest of the application:

Listing 4.1: Appwrite client configuration

```
import { Client, Account, Databases, ID, Query } from 'appwrite';
```

```
const client = new Client();

client
    .setEndpoint(import.meta.env.VITE_APPWRITE_ENDPOINT)
    .setProject(import.meta.env.VITE_APPWRITE_PROJECT_ID);

export const account = new Account(client);
export const databases = new Databases(client);

export const APPWRITE_CONFIG = {
    databaseId: import.meta.env.VITE_APPWRITE_DATABASE_ID,
    collectionId: import.meta.env.VITE_APPWRITE_COLLECTION_ID,
};

export { ID, Query };
```

Keeping every Appwrite identifier in `import.meta.env` means no credentials are committed to the repository and deployments can point at different Appwrite instances without code changes.

4.2.2 Login Flows

File Location: `src/gui/components/Login/Loginpage.jsx`

The login page offers three entry paths:

1. **Email/password**: creates an Appwrite email session and routes to the homepage.
2. **Google OAuth**: delegates to Appwrite's OAuth2 session flow with success/failure redirect URLs.
3. **Continue as Guest**: sets an `isGuest` flag in `sessionStorage`; no account is created and all data stays on the device.

The page also received visual polish during the fellowship (project logo, colour scheme alignment with the desktop application) and a logout path was wired through the `ProfileAvatar` account menu.

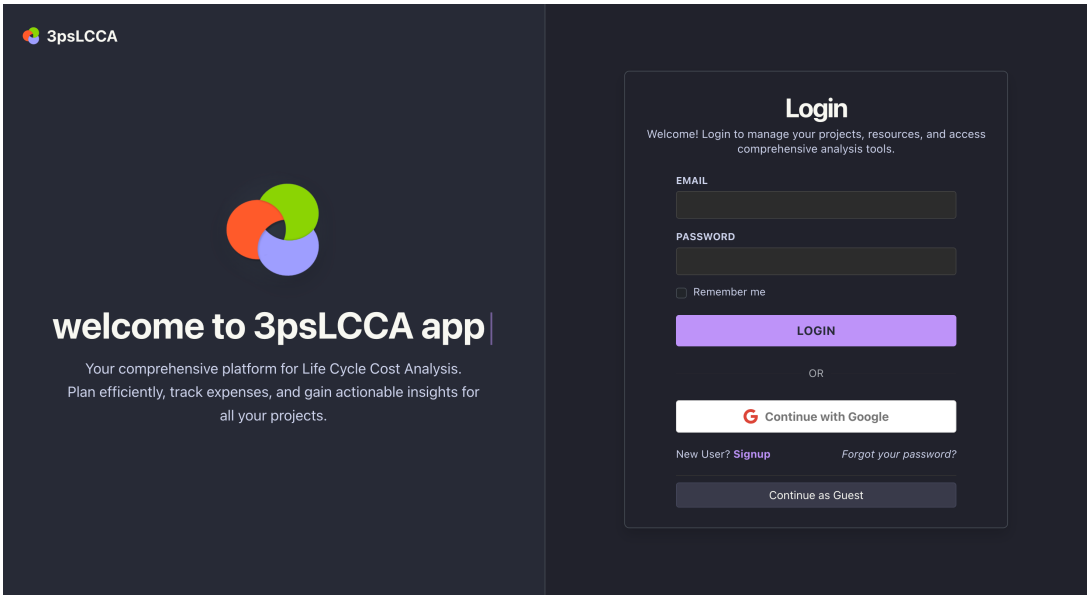

Figure 4.1: 3psLCCA web login page

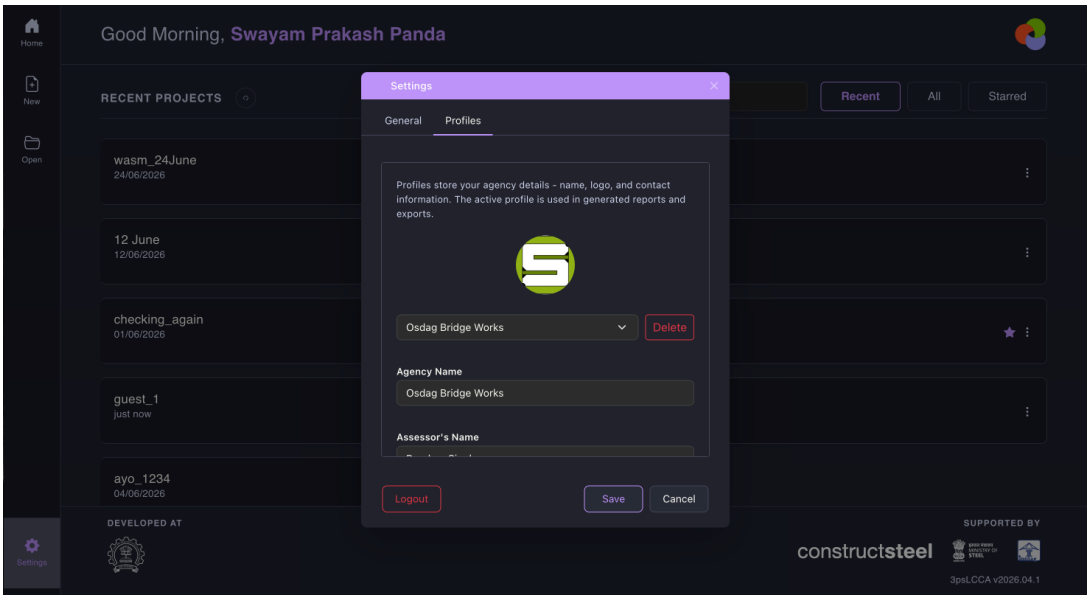

Figure 4.2: Profile settings with the active agency profile and logout

4.3 Guest Mode with Persistent Local Storage

The legacy checkpoint system was removed entirely and replaced with persistent `localStorage` -backed projects for guest users. Each project is stored under a stable key (`project_data_<id>`) together with a `recentProjects` index that drives the homepage cards. Guest projects therefore survive page refreshes and browser restarts, which the checkpoint system did not, and the same storage code path is reused as the local half of the offline-first

design below.

4.4 Offline-First Storage Service

File Location: src/lib/projectStorageService.js

4.4.1 Design

The storage service abstracts *where* a project lives behind three operations (saveProject, loadProject, listProjects). Its strategy is offline-first:

1. Every save is written to localStorage *first*, wrapped with a sync_status flag (synced or pending).
2. If the user is authenticated, the service then attempts to sync the document to the Appwrite database in the background, updating the flag on success.
3. Conflicts between local and cloud copies are resolved with a last-write-wins policy using a _lastModified timestamp embedded in the normalised project data.

4.4.2 Save Path

Listing 4.2: Offline-first save with background cloud sync (abridged)

```
export const projectStorageService = {
    async saveProject(projectId, projectData) {
        const isGuest = sessionStorage.getItem('isGuest') === '
           true';
        const now = Date.now();
        const normalizedProject = normalizeProjectData({
            ...projectData,
            _lastModified: now,
        });

        // 1. Always Save Locally First
        const storageKey = `project_data_${projectId}`;
```

```
        const localWrapper = {
            data: normalizedProject,
            sync_status: isGuest ? 'synced' : 'pending'
        };
        localStorage.setItem(storageKey, JSON.stringify(
           localWrapper));

        // 2. Try to sync to cloud if logged in
        if (!isGuest) {
            try {
                const userId = await getCurrentUserId();
                if (!userId) throw new Error("Not logged in");

                const cloudData = {
                    name: normalizedProject.name || 'Unnamed
                       Project',
                    data: JSON.stringify(normalizedProject)
                };

                try {
                    await databases.getDocument(
                        APPWRITE_CONFIG.databaseId,
                        APPWRITE_CONFIG.collectionId, projectId);
                    await databases.updateDocument(
                        APPWRITE_CONFIG.databaseId,
                        APPWRITE_CONFIG.collectionId,
                        projectId, cloudData);
                } catch (e) {
                    await databases.createDocument(
                        APPWRITE_CONFIG.databaseId,
                        APPWRITE_CONFIG.collectionId,
                        projectId, { ...cloudData, userId: userId
                            });
                }
```

```
                // Cloud save success: update local status
                localWrapper.sync_status = 'synced';
                localStorage.setItem(storageKey,
                    JSON.stringify(localWrapper));
            } catch (err) {
                console.error(
                  "Failed to save project to cloud. Preserved
                     locally.",
                  err);
                throw new Error("offline");
            }
        }
    },
    // loadProject(), listProjects(), ...
};
```

In this listing:

- Every payload passes through `normalizeProjectData()` (Chapter 5) before it is persisted, so both storage tiers always hold canonical-schema documents.
- The get-then-update/create pattern makes saves idempotent: the same project ID is used as the Appwrite document ID, so a document is updated if it exists and created otherwise.
- A cloud failure does not lose data: the local copy is already written and remains flagged `pending`; the thrown `"offline"` error is what flips the navbar save indicator into its offline state.

4.4.3 Load Path and Conflict Resolution

On load, guests read only the local copy. Authenticated users fetch the cloud document and compare `_lastModified` timestamps against any local copy: the newer document wins and, when the local copy is newer (an offline edit), it is re-synced upward. Backwards

compatibility is preserved for projects stored before the wrapper format existed: a raw project object without `sync_status` is detected and normalised on read.

4.4.4 Save Indicator and Outputs Unlock

The `sync_status` lifecycle drives a dynamic save indicator in `ProjectNavbar.jsx` (*Saving…* → *Saved* → *Offline – saved locally*), which keeps the user informed about persistence state. The same sync work also unlocked the Outputs page: results can now be produced and revisited reliably because the inputs they were computed from are always persisted first.

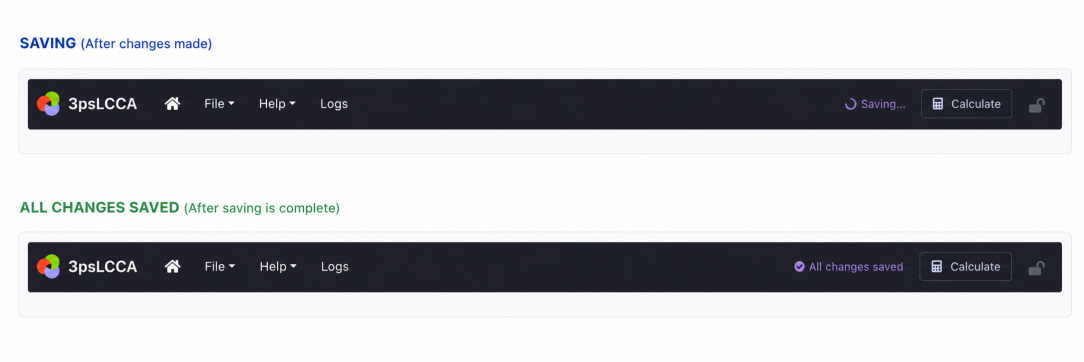

Figure 4.3: Dynamic save indicator in the project navbar

4.5 Project Import and Export

Together with Priyansh Vaish, native support for `.3ps` project files was implemented so projects can move between the web and desktop applications. Export serialises the canonical project object; import parses and normalises an uploaded `.3ps` file into a new local (and, if logged in, cloud) project. The open-project flow for signed-in users was fixed as part of this work.

4.6 Summary

The web application now has user accounts and durable persistence: Appwrite email and Google OAuth sessions, a cloud project database keyed by user, a persistent guest mode, and an offline-first storage service with last-write-wins sync. The features described in the following chapters read and write projects only through this storage layer.

Chapter 5

Desktop Schema Parity and Data Validation

5.1 Overview

The web application can only reproduce the desktop application's results if its project data means the same thing as the desktop application's data. At the start of the fellowship this was not the case: pages stored loosely structured state with ad-hoc field names, results were produced by a JavaScript re-implementation of parts of the engine, and several inputs silently diverged from the desktop application. The sections below describe the canonicalisation of the project schema, the shared normalisers, and the validation adapter that maps web projects onto `3psLCCA-core`.

5.2 Canonical Project Schema

5.2.1 Phase 1: Standardising Pages Against the Desktop Application

The General Information page was standardised first: its fields (company name, project title, description, valuer, job number, client, country, base year) were aligned with the desktop application and integrated with the user profile. The project schema was then standardised as a whole and the calculation path was switched from the JavaScript re-implementation to the real Python engine (initially through the FastAPI development

backend), after which `3psLCCA-core` was the only calculation engine in use.

5.2.2 Phase 2: Page-by-Page Canonicalisation and Shared Normalisers

Each page of the workflow (General Information, Input Parameters with the structure works components, Bridge Data, and Traffic Data) was then canonicalised page by page against the corresponding desktop widget. A shared `normalizeProjectData()` utility (`src/utils/projectSchema.js`) enforces the canonical shape at every boundary: on page save, in the storage service (Chapter 4), on `.3ps` import, and before every calculation. Normalisation fills defaults for missing blocks, coerces numeric strings, migrates legacy field names, and strips unknown keys, so that old projects load cleanly into the new schema.

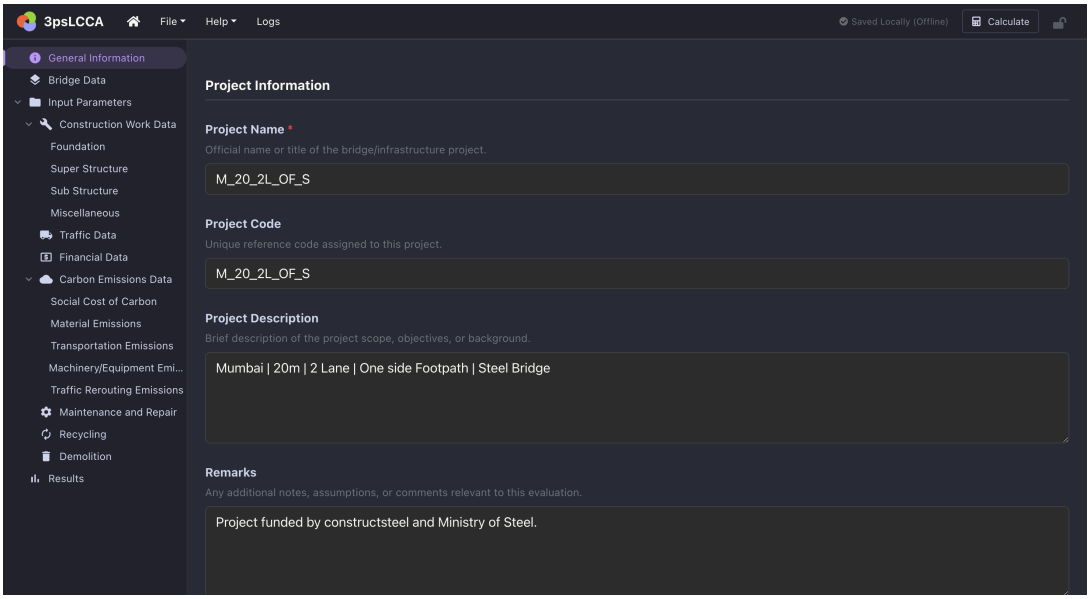

Figure 5.1: Canonicalised General Information page

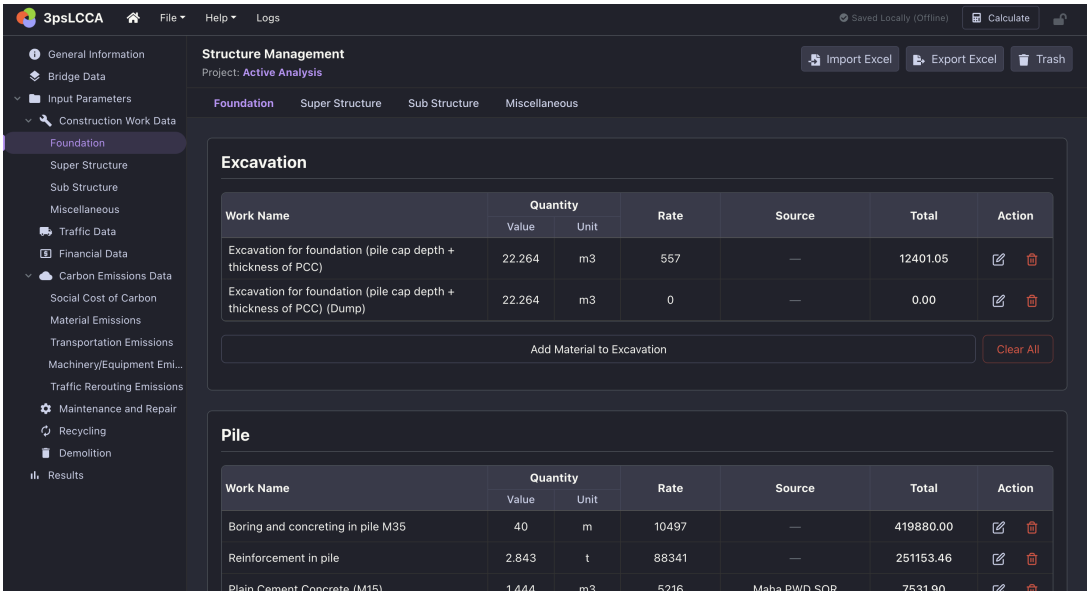

Figure 5.2: Structure works input with the canonical material schema

5.3 The Web-to-Core Adapter

5.3.1 Role and Location

File Location: `src/wasm/python/web_to_core.py` (shared; re-exported by `backend/app/adapters/web_to_core.py` for the FastAPI fallback)

The adapter is an 834-line Python module that defines the contract between the web schema and the core engine. It has three responsibilities:

1. **Validate** a web project dictionary and produce human-readable, field-addressed error and warning lists.

2. **Map** the validated web schema onto the exact input dictionary expected by `run_full_lcc_analys` `()` (general parameters, vehicle data for the eight vehicle classes, accident severity distribution, maintenance stage parameters, construction costs, and optional WPI).

3. **Execute** the core calculation and shape the response envelope (`results`, `computed`, `validation`) consumed by the UI.

Because the adapter is plain Python with no server dependencies, the same file runs under FastAPI in development and inside Pyodide in the browser (Chapter 6), so validation behaves identically in both.

5.3.2 Phase 5: Adapter Hardening

The adapter was hardened with strict validation constraints and a backend test suite, and the residual JavaScript calculation engine was removed. The validation helpers address every field by its schema path so errors surface next to the offending input:

Listing 5.1: Strict numeric validation with field-addressed errors (illustrative)

```
class AdapterValidationError(ValueError):
    """Raised when a web project cannot be mapped onto the core."
       ""
    def __init__(self, errors, warnings=None):
        super().__init__("; ".join(errors))
        self.errors = errors
        self.warnings = warnings or []

def _require_number(block, key, path, errors, *,
                    minimum=None, maximum=None):
    """Fetch block[key] as a float, recording a field-addressed
    error such as 'financial_data.discount_rate' on failure."""
    ...

# Examples of hardened constraints applied during validation:
design_life = _require_number(
    general, "design_life", "general.design_life", errors,
       minimum=1)
_require_number(financial, "discount_rate",
    "financial_data.discount_rate", errors, minimum=0)
_require_number(financial, "inflation_rate",
    "financial_data.inflation_rate", errors, minimum=0)
_require_number(financial, "interest_rate",
    "financial_data.interest_rate", errors, minimum=0)
```

Validation failures raise `AdapterValidationError` carrying the full error and warning lists, which the bridge (Chapter 6) serialises into the response envelope; the Outputs page then renders them as validation messages beside the results area (contributed by Priyansh

Vaish).

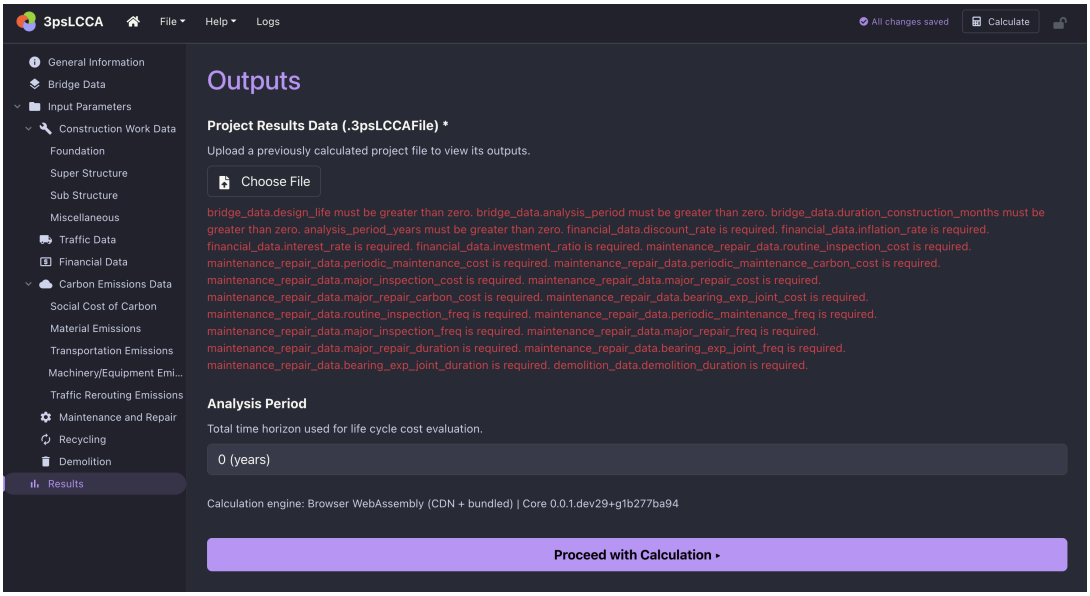

Figure 5.3: Field-addressed validation messages in the results section

5.4 Bridge and Traffic Data Parity

5.4.1 Bridge Data Schema Polish

The Bridge Data page schema and its adapter constraints were polished so that carriageway configuration (lane codes such as `SL`, `IL`, `2L`, ...), carriage width, road roughness, rise/fall, re-route distance and time, crash rate, work-zone multiplier, and peak-hour parameters map one-to-one onto the desktop application's fields.

5.4.2 Traffic Schema Strict Parity and WPI Parsing

The traffic schema was overhauled for strict parity: the eight desktop vehicle classes (`small_cars`, `big_cars`, `two_wheelers`, `o_buses`, `d_buses`, `lcv`, `mcv`, `hcv`) with vehicles per day, per-kilometre carbon emissions, accident percentages, and power-to-weight ratios where applicable, plus the minor/major/fatal accident severity distribution. Strict Wholesale Price Index (WPI) parsing was implemented against the vehicle-grouped WPI structure used by the core, backed by the `wpi_db.json` database shipped with the application, so Indian projects apply the same inflation adjustments as the desktop tool.

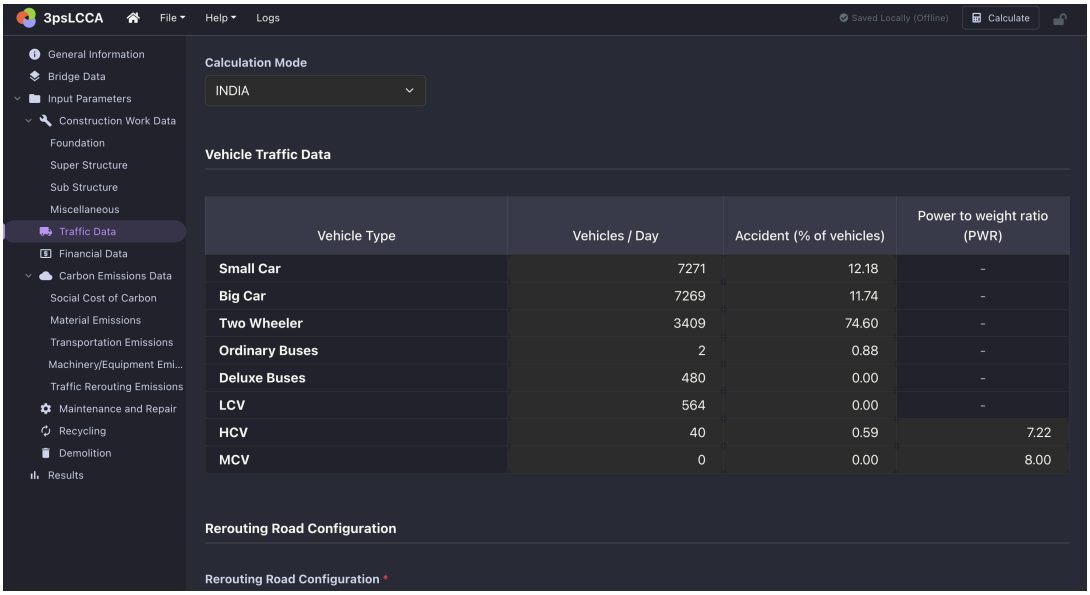

Figure 5.4: Traffic data page with strict desktop parity

5.5 Construction Data Workflow Enhancements

5.5.1 Soft Deletion and the Global Trash

Construction materials and whole sections are no longer destroyed on delete. A soft-deletion model marks records as trashed while keeping them in the project document, and a global `ConstructionTrash` view lists everything trashed across components with restore and permanent-delete actions. This mirrors safe data-handling expectations from the desktop workflow and protects users from accidental loss during long data-entry sessions.

5.5.2 Excel Import

File Location: `src/utils/constructionExcel.js`

Construction material data can be imported from Excel workbooks using ExcelJS. The importer parses the workbook, maps sheets and columns onto the canonical component/material schema, and presents an `ImportPreviewModal` so the user can review exactly which rows will be created before committing; invalid rows are listed instead of silently dropped.

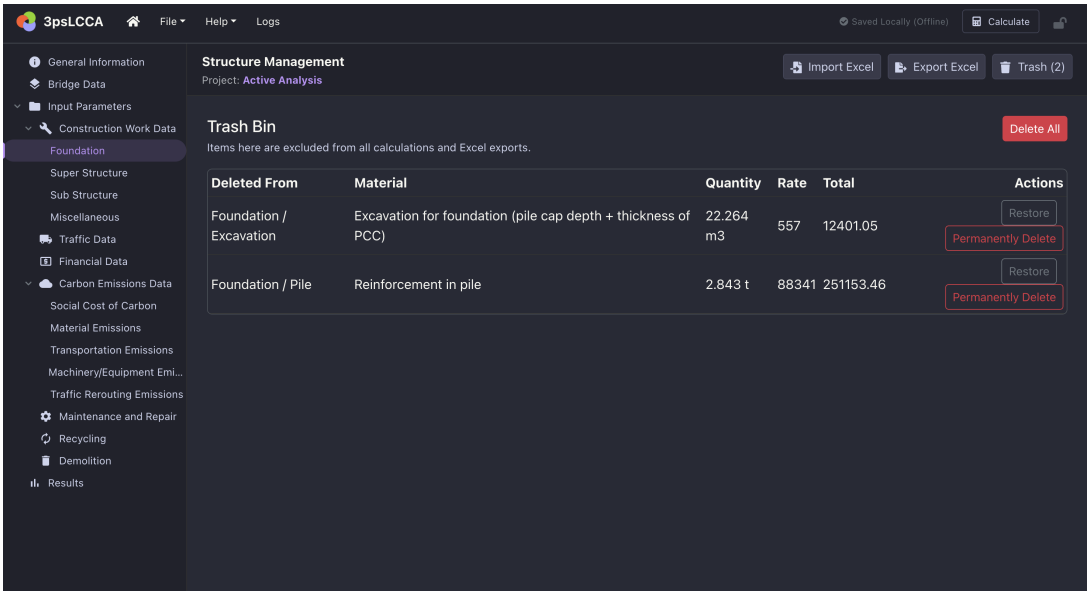

Figure 5.5: Global trash for construction data

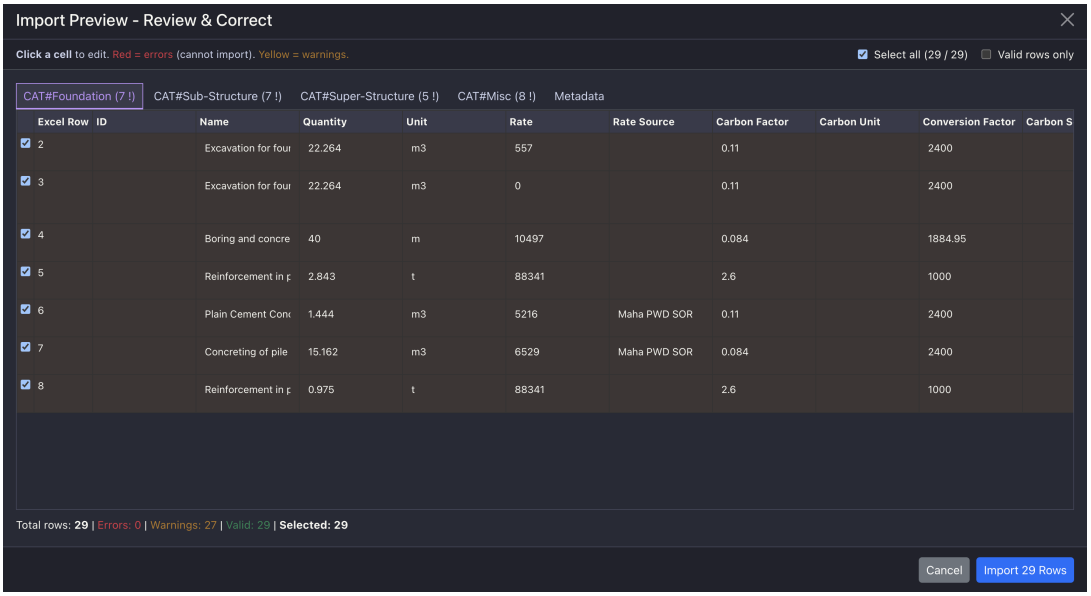

Figure 5.6: Excel import preview for construction materials

5.6 Backend Test Suite

File Location: `backend/tests/`

The adapter is covered by pytest suites (`test_adapter.py`, `test_api.py`) exercising: complete valid projects mapping to core inputs, every strict constraint rejecting bad values with the correct field path, alias/legacy field migration, and the FastAPI endpoints returning the same envelopes as the in-browser bridge. These tests, together with the

core-side reference suite of Chapter 6, pin the web-to-core contract.

5.7 Summary

After this phase the web application stores one canonical schema enforced by shared normalisers, validates and maps it through a single Python adapter with field-addressed errors, and covers that contract with tests. Construction data handling also became safer through soft deletion, the trash view, and the previewed Excel import. The remaining dependency was the server that ran the engine, which the next chapter removes.

Chapter 6

Static WebAssembly Calculation Runtime

6.1 Overview

Until this phase, every calculation still required a FastAPI server running the Python engine, even after the schema work of Chapter 5. The web application therefore could not be deployed on its own; a Python backend had to be hosted and paid for alongside it. This chapter covers the main technical contribution of the fellowship: running `3psLCCA-core` in the browser as WebAssembly via Pyodide, inside a Web Worker, from fully static files. It also covers the supporting work on the core repository itself: a native reference test suite, a continuous integration pipeline that builds and verifies the wheel, and a standalone JavaScript wrapper library with a demo page for third-party integrators.

6.2 Architecture

The runtime has four layers, from the UI downward:

1. `wasmEngine.js`: the main-thread API (`initializeLccaEngine`, `calculateLcca`, `validateLcca`) exposing promise-based calls and engine status.

2. `lcca.worker.js`: a module Web Worker that boots Pyodide, installs the core wheel, and services requests, so the UI thread stays free while Python executes.

3. bridge.py: a thin JSON-in/JSON-out Python module injected into Pyodide, delegating to the shared adapter of Chapter 5.

4. 3psLCCA-core: the unmodified pure-Python engine, installed from its wheel into Pyodide's virtual filesystem.

Everything the runtime needs (the Pyodide distribution, the core wheel, and a manifest) is produced at build time and served as static files, so no external CDN or server is contacted at runtime.

6.3 Build-Time Asset Pipeline

File Location: scripts/prepare-wasm-assets.mjs

The asset pipeline locates a checkout of 3psLCCA-core (via LCCA_CORE_PATH or known sibling paths), builds it into a pure-Python wheel, copies the self-hosted Pyodide distribution, computes checksums, and writes a manifest.json recording the wheel filename, the core version, and the Pyodide version:

Listing 6.1: Locating the core checkout in the asset pipeline (excerpt)

```
const possiblePaths = [
  process.env.LCCA_CORE_PATH,
  '../3psLCCA-gui-python-venv/3psLCCA-core',
  '../3psLCCA-core',
].filter(Boolean)

let coreRoot = null
for (const p of possiblePaths) {
  const resolvedPath = resolve(projectRoot, p)
  if (existsSync(resolvedPath)) { coreRoot = resolvedPath; break
     }
}
if (!coreRoot) {
  throw new Error(
    `ERROR: Could not find 3psLCCA-core in: ${possiblePaths.join
       (', ')}`)
```

```
}
```

`npm run build` therefore emits a `dist/` folder containing the React app, the worker, Pyodide, the wheel, and reference data, which can be deployed to any static host with no server cost.

6.4 The Web Worker Runtime

File Location: `src/lib/lccaEngine/lcca.worker.js`

6.4.1 Initialisation

The worker imports the adapter and bridge *source code* as raw strings through Vite's `?raw` imports, boots Pyodide from the self-hosted distribution, installs the wheel named by the manifest, and then materialises the two Python modules inside the interpreter:

Listing 6.2: Worker initialisation: Pyodide boot, wheel install, module injection

```
import adapterSource from '../../wasm/python/web_to_core.py?raw'
import bridgeSource from '../../wasm/python/bridge.py?raw'
import wpiDatabase from '../../data/wpi_db.json'

const initialize = async () => {
  if (bridgeReady) return bridgeReady
  bridgeReady = (async () => {
    postStatus('loading-runtime')
    const { loadPyodide } =
      await import(assetUrl('pyodide/pyodide.mjs'))
    pyodide = await loadPyodide({ indexURL: assetUrl('pyodide/')
       })

    postStatus('loading-core')
    const manifest =
      await (await fetch(assetUrl('lcca-wasm/manifest.json'))).
         json()
```

```
    await pyodide.loadPackage(assetUrl(`lcca-wasm/${manifest.
       wheel}`))

    pyodide.globals.set('_lcca_adapter_source', adapterSource)
    pyodide.globals.set('_lcca_bridge_source', bridgeSource)
    pyodide.globals.set('_lcca_wpi_database_json',
                        JSON.stringify(wpiDatabase))
    await pyodide.runPythonAsync(`
import sys, types
lcca_web_adapter = types.ModuleType("lcca_web_adapter")
sys.modules["lcca_web_adapter"] = lcca_web_adapter
exec(compile(_lcca_adapter_source, "lcca_web_adapter.py", "exec")
   ,
     lcca_web_adapter.__dict__)

lcca_web_bridge = types.ModuleType("lcca_web_bridge")
sys.modules["lcca_web_bridge"] = lcca_web_bridge
exec(compile(_lcca_bridge_source, "lcca_web_bridge.py", "exec"),
     lcca_web_bridge.__dict__)
lcca_web_bridge.initialize(_lcca_wpi_database_json)
`)
    postStatus('ready')
    return { status: 'ready',
             coreVersion: manifest.coreVersion,
             pyodideVersion: manifest.pyodideVersion }
  })()
  return await bridgeReady
}
```

Injecting the adapter as source (rather than baking it into the wheel) keeps the web-schema contract versioned with the web repository, while the wheel stays a plain build of the core repository. The core and Pyodide versions returned here are surfaced through the engine status and stamped into generated reports as provenance (Chapter 7).

6.4.2 Request Handling

Calculation and validation requests serialise the project to JSON, invoke the bridge, and parse the JSON response; no PyProxy objects cross the worker boundary, which avoids Pyodide object-lifetime problems:

Listing 6.3: JSON-only bridge invocation inside the worker

```
const callBridge = async (method, payload) => {
  await initialize()
  pyodide.globals.set('_lcca_request_json', JSON.stringify({
    project: payload.project,
    analysis_period_years: payload.analysisPeriodYears,
  }))
  try {
    const response = await pyodide.runPythonAsync(
      `lcca_web_bridge.${method}(_lcca_request_json)`)
    return JSON.parse(response)
  } finally {
    pyodide.globals.delete('_lcca_request_json')
  }
}
```

6.4.3 The Python Bridge

File Location: src/wasm/python/bridge.py

The bridge exposes three functions (initialize, validate, and calculate), each taking and returning JSON strings, and translates adapter failures into structured error envelopes rather than exceptions:

Listing 6.4: calculate() in bridge.py: success, validation-error, and runtime-error envelopes

```
def calculate(request_json: str) -> str:
    request: dict[str, Any] = json.loads(request_json)
    try:
        calculation = calculate_project(
```

```
            request.get("project", {}),
            int(request.get("analysis_period_years", 50)),
            debug=False,
        )
        return json.dumps({"status": "success", **calculation},
                          allow_nan=False)
    except AdapterValidationError as exc:
        return json.dumps({
            "status": "error", "results": {}, "computed": {},
            "validation": {"errors": exc.errors,
                           "warnings": exc.warnings},
        }, allow_nan=False)
    except Exception as exc:
        return json.dumps({
            "status": "error", "results": {}, "computed": {},
            "validation": {"errors": [str(exc)], "warnings": []},
            "runtime_error": {
                "type": type(exc).__name__,
                "traceback": traceback.format_exc(),
            },
        }, allow_nan=False)
```

Setting `allow_nan=False` makes serialisation fail if a NaN appears anywhere in the results, instead of letting the charts render `NaN` without warning.

6.5 Main-Thread Engine API

File Location: `src/lib/lccaEngine/wasmEngine.js`

The main-thread module manages the worker lifecycle: lazy creation, an idempotent `initializeLccaEngine()` promise, correlation of requests and responses by ID, status reporting for the UI (`idle` / `loading-runtime` / `loading-core` / `ready` / `calculating` / `failed`), and full teardown-and-reject semantics if the worker crashes:

Listing 6.5: Promise-correlated requests to the worker

```
const request = (type, payload = {}) =>
```

```
  new Promise((resolve, reject) => {
    const id = nextRequestId++
    pending.set(id, { resolve, reject })
    ensureWorker().postMessage({ id, type, payload })
  })

export const initializeLccaEngine = () => {
  if (!initializePromise) {
    initializePromise = request('initialize').then((result) => {
      status = { state: 'ready', mode: 'wasm', ...result }
      return result
    }).catch((error) => { resetWorker(error); throw error })
  }
  return initializePromise
}

export const calculateLcca = (payload) => run('calculate',
   payload)
export const validateLcca  = (payload) => run('validate', payload
   )
```

6.5.1 Parity Smoke Test

A dedicated smoke page (wasm-smoke.html) loads the runtime and compares browser WebAssembly results against native CPython reference outputs generated by scripts/generate-native-reference.py. Any numerical divergence between browser and native execution of the same core version fails the smoke test.

6.6 Core Repository: Tests, CI, and Distribution

The browser runtime depends on the wheel it installs, so the fellowship also contributed directly to 3psLCCA-core (branch feat/wasm-static-runtime).

6.6.1 Reference Test Suite

File Location: tests/ (in 3psLCCA-core)

A pytest suite pins the engine's behaviour to known-good reference values for both the global and the India (WPI-adjusted) calculation paths, plus edge cases:

Listing 6.6: Reference calculations pinned by pytest (excerpt)

```
def test_india_reference_calculation(
    india_input: dict, india_wpi: dict, construction_costs: dict,
) -> None:
    result = run_full_lcc_analysis(
        india_input, construction_costs.copy(),
        wpi=india_wpi, debug=False,
    )
    assert set(result) == EXPECTED_STAGES
    assert result["initial_stage"]["social"][
        "initial_road_user_cost"] == pytest.approx(132_466.08)
    assert result["initial_stage"]["environmental"][
        "initial_vehicular_emission_cost"] == pytest.approx(4_444
           .43)
    assert result["end_of_life"]["social"][
        "ruc_demolition"] == pytest.approx(917_989.9)

def test_zero_traffic_short_circuits_road_user_cost(
    india_input, india_wpi, construction_costs,
) -> None:
    for vehicle in india_input[
            "traffic_and_road_data"]["vehicle_data"].values():
        vehicle["vehicles_per_day"] = 0
        vehicle["accident_percentage"] = 0
    result = run_full_lcc_analysis(
        india_input, construction_costs.copy(),
        wpi=india_wpi, debug=False)
    assert result["initial_stage"]["social"][
        "initial_road_user_cost"] == 0
```

```
def test_invalid_input_returns_a_clear_error(construction_costs):
    with pytest.raises(ValueError,
                       match="Missing 'general_parameters' block"
                          ):
        run_full_lcc_analysis({}, construction_costs.copy(),
                              debug=False)
```

Shared fixtures in `conftest.py` deep-copy the example input dictionaries so tests cannot contaminate each other, and the packaging was extended (`pyproject.toml`) with a `[test]` extra, pytest configuration, and a `setuptools_scm` fallback version so the wheel builds outside a git checkout.

6.6.2 Continuous Integration

File Location: `.github/workflows/ci.yml`

A GitHub Actions workflow now runs on every push and pull request:

Listing 6.7: Core CI: test, build, verify, checksum (abridged)

```
jobs:
  test-and-build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with: { fetch-depth: 0 }
      - uses: actions/setup-python@v5
        with: { python-version: "3.12" }
      - run: python -m pip install --upgrade pip build pytest
      - run: python -m pip install -e .[test]
      - name: Run native reference tests
        run: python -m pytest -q
      - name: Build wheel
        run: python -m build --wheel
      - name: Verify clean wheel installation
        run: |
```

```
          python -m venv wheel-venv
          wheel-venv/bin/python -m pip install dist/*.whl
          wheel-venv/bin/python -c "from three_ps_lcca_core.core.
             main \
            import run_full_lcc_analysis; \
            assert callable(run_full_lcc_analysis)"
      - name: Generate checksum
        run: sha256sum dist/*.whl > dist/SHA256SUMS
      - uses: actions/upload-artifact@v4
        with:
          name: three-ps-lcca-core-wheel
          path: |
            dist/*.whl
            dist/SHA256SUMS
```

The clean-install verification step checks that the wheel is self-contained, i.e. importable in a fresh virtual environment without a repository checkout, which is the condition Pyodide's micropip/loadPackage requires.

6.6.3 Standalone JavaScript Wrapper and Demo

File Location: wasm-demo/ (in 3psLCCA-core)

For integrators outside the React application, a dependency-free wrapper library, 3pslcca-wasm.js, packages the whole runtime behind a two-method API, accompanied by a demo page (index.html, demo.js), a sample payload, and a WASM_INTEGRATION_GUIDE.md:

Listing 6.8: Public API of the ThreePSLCCA wrapper library

```
const engine = new ThreePSLCCA();
await engine.init({
  onStdout: (msg) => console.log("[Python]", msg),
  onStderr: (msg) => console.error("[Python]", msg),
});
const results = await engine.calculate(payload, /* debug */ true)
   ;
```

```
const initialCost = results.initial_stage.total_initial_cost;
```

Internally, `init()` dynamically loads the Pyodide script, boots the interpreter with the caller's stdout/stderr hooks, installs the wheel via `micropip`, and sets a ready flag; `calculate()` injects the payload into Python globals, runs `run_full_lcc_analysis()`, converts the result to plain JavaScript objects with `toJs` using `Object.fromEntries` as the dictionary converter, and destroys the PyProxy to prevent memory leaks.

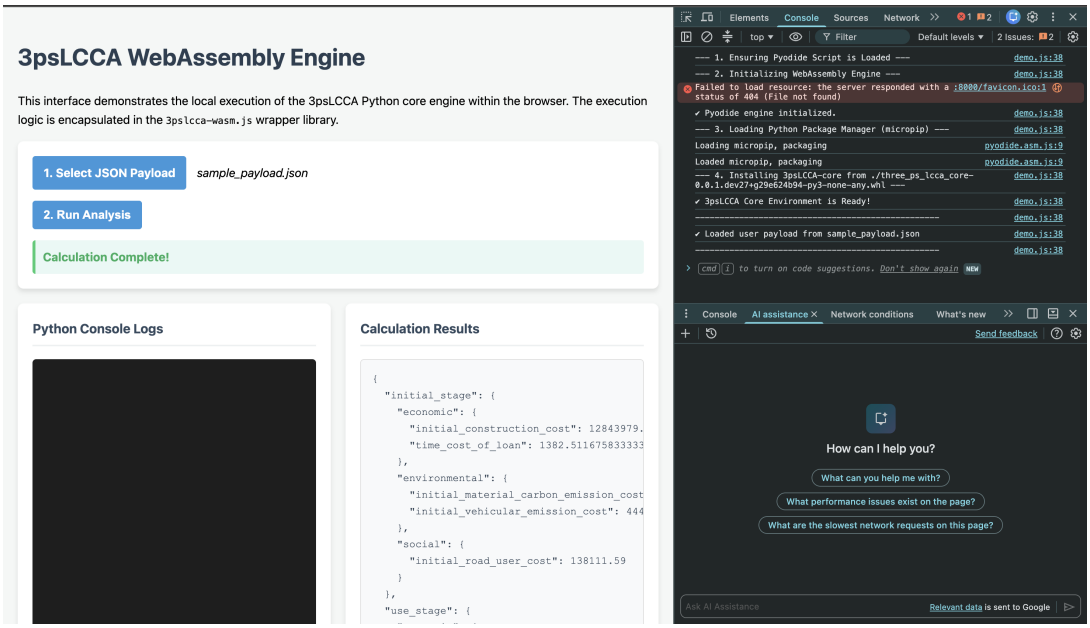

Figure 6.1: Browser console during wrapper initialisation: Pyodide boot, micropip install of the core wheel, and ready state

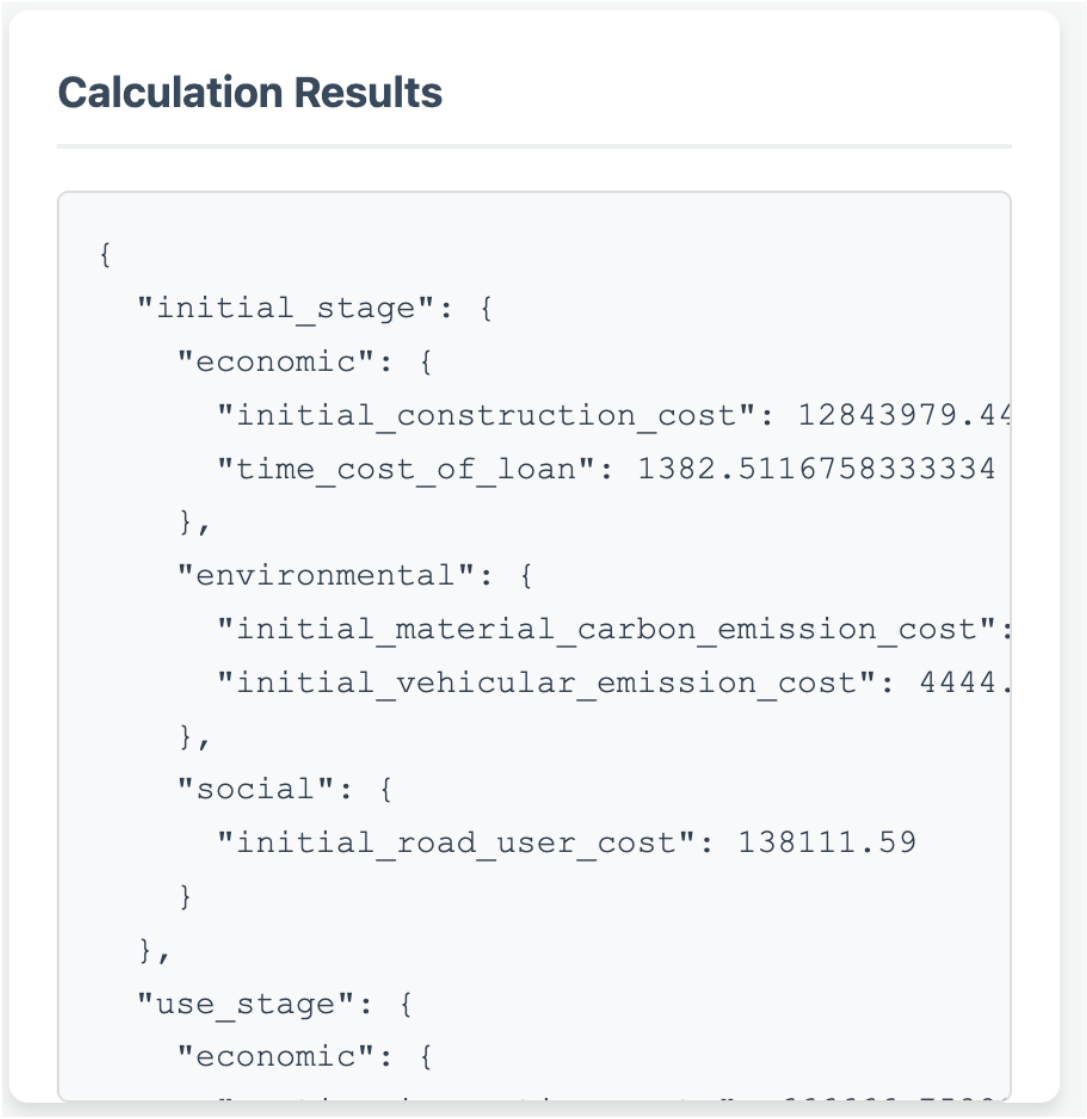

Figure 6.2: Demo page after a calculation: stage-wise LCCA results computed entirely in the browser

6.7 Full Code

6.7.1 lcca.worker.js

Listing 6.9: Complete LCCA Web Worker

```
import adapterSource from '../../wasm/python/web_to_core.py?raw'
import bridgeSource from '../../wasm/python/bridge.py?raw'
import wpiDatabase from '../../data/wpi_db.json'

let pyodide
```

```
let bridgeReady

const appBaseUrl = new URL(import.meta.env.BASE_URL, self.
   location.href)
const assetUrl = (path) => new URL(path, appBaseUrl).href

const postStatus = (state) => self.postMessage({ type: 'status',
   state })

const initialize = async () => {
  if (bridgeReady) return bridgeReady

  bridgeReady = (async () => {
    postStatus('loading-runtime')
    const { loadPyodide } = await import(/* @vite-ignore */
       assetUrl('pyodide/pyodide.mjs'))
    pyodide = await loadPyodide({ indexURL: assetUrl('pyodide/')
       })

    postStatus('loading-core')
    const manifestResponse = await fetch(assetUrl('lcca-wasm/
       manifest.json'))
    if (!manifestResponse.ok) {
      throw new Error(`Unable to load the LCCA Wasm manifest (${
         manifestResponse.status}).`)
    }
    const manifest = await manifestResponse.json()
    await pyodide.loadPackage(assetUrl(`lcca-wasm/${manifest.
       wheel}`))

    pyodide.globals.set('_lcca_adapter_source', adapterSource)
    pyodide.globals.set('_lcca_bridge_source', bridgeSource)
    pyodide.globals.set('_lcca_wpi_database_json', JSON.stringify
       (wpiDatabase))
```

```
    await pyodide.runPythonAsync(`
import sys
import types

lcca_web_adapter = types.ModuleType("lcca_web_adapter")
lcca_web_adapter.__file__ = "lcca_web_adapter.py"
sys.modules["lcca_web_adapter"] = lcca_web_adapter
exec(compile(_lcca_adapter_source, "lcca_web_adapter.py", "exec")
    , lcca_web_adapter.__dict__)

lcca_web_bridge = types.ModuleType("lcca_web_bridge")
lcca_web_bridge.__file__ = "lcca_web_bridge.py"
sys.modules["lcca_web_bridge"] = lcca_web_bridge
exec(compile(_lcca_bridge_source, "lcca_web_bridge.py", "exec"),
   lcca_web_bridge.__dict__)
lcca_web_bridge.initialize(_lcca_wpi_database_json)

del _lcca_adapter_source
del _lcca_bridge_source
del _lcca_wpi_database_json
`)
    postStatus('ready')
    return {
      status: 'ready',
      coreVersion: manifest.coreVersion,
      pyodideVersion: manifest.pyodideVersion,
    }
  })()

  try {
    return await bridgeReady
  } catch (error) {
    bridgeReady = undefined
    pyodide = undefined
```

```
    postStatus('failed')
    throw error
  }
}

const callBridge = async (method, payload) => {
  await initialize()
  pyodide.globals.set('_lcca_request_json', JSON.stringify({
    project: payload.project,
    analysis_period_years: payload.analysisPeriodYears,
  }))
  try {
    const response = await pyodide.runPythonAsync(
      `lcca_web_bridge.${method}(_lcca_request_json)`,
    )
    return JSON.parse(response)
  } finally {
    pyodide.globals.delete('_lcca_request_json')
  }
}

self.addEventListener('message', async (event) => {
  const { id, type, payload } = event.data
  try {
    let result
    if (type === 'initialize') result = await initialize()
    else if (type === 'calculate') result = await callBridge('
       calculate', payload)
    else if (type === 'validate') result = await callBridge('
       validate', payload)
    else throw new Error(`Unsupported LCCA worker request: ${type
       }`)
    self.postMessage({ id, type: 'result', result })
  } catch (error) {
```

```
    self.postMessage({
      id,
      type: 'error',
      error: {
        name: error?.name || 'Error',
        message: error?.message || String(error),
        stack: error?.stack || '',
      },
    })
  }
})
```

6.7.2 bridge.py

Listing 6.10: Complete Python JSON bridge

```
from __future__ import annotations

import json
import traceback
from typing import Any

from lcca_web_adapter import (
    AdapterValidationError,
    calculate_project,
    configure_wpi_database,
    validate_project,
)


def initialize(wpi_database_json: str) -> str:
    configure_wpi_database(json.loads(wpi_database_json))
    return json.dumps({"status": "ready"})
```

```
def validate(request_json: str) -> str:
    request = json.loads(request_json)
    validation = validate_project(
        request.get("project", {}),
        int(request.get("analysis_period_years", 50)),
    )
    return json.dumps(
        {
            "status": "success" if not validation["errors"] else
               "error",
            "results": {},
            "computed": {},
            "validation": validation,
        },
        allow_nan=False,
    )


def calculate(request_json: str) -> str:
    request: dict[str, Any] = json.loads(request_json)
    try:
        calculation = calculate_project(
            request.get("project", {}),
            int(request.get("analysis_period_years", 50)),
            debug=False,
        )
        return json.dumps(
            {"status": "success", **calculation},
            allow_nan=False,
        )
    except AdapterValidationError as exc:
        return json.dumps(
            {
                "status": "error",
```

```
                "results": {},
                "computed": {},
                "validation": {
                    "errors": exc.errors,
                    "warnings": exc.warnings,
                },
            },
            allow_nan=False,
        )
    except Exception as exc:
        return json.dumps(
            {
                "status": "error",
                "results": {},
                "computed": {},
                "validation": {
                    "errors": [str(exc)],
                    "warnings": [],
                },
                "runtime_error": {
                    "type": type(exc).__name__,
                    "traceback": traceback.format_exc(),
                },
            },
            allow_nan=False,
        )
```

6.8 Summary

After this phase no server remains in the production deployment. The core engine ships as a CI-verified pure-Python wheel; the web application builds it into static assets, boots it in a Web Worker via self-hosted Pyodide, and talks to it through a JSON-only bridge whose validation behaviour matches the development backend. The parity smoke test

and the reference suite check the numerical results, and the standalone wrapper library makes the same runtime available to other JavaScript frontends.

Chapter 7

In-Browser LaTeX Report Generation

7.1 Overview

The desktop application generates a typeset LaTeX design report as its main output. The web application previously depended on server-side PDF generation (or fell back to simplified jsPDF output that did not match the desktop report). The final phase of the fellowship replaced this with SwiftLaTeX: the PdfTeX engine compiled to WebAssembly, running in a Web Worker against a statically bundled TeX Live footprint, so desktop-equivalent reports now compile entirely on the client.

7.2 Architecture

File Locations: src/gui/components/outputs/ (reportModel.js, latexReport.js, latexReportEngine.js); public/vendor/swiftlatex/ (engine and TeX Live assets)

The pipeline is a pure data transformation followed by a worker compilation:

Listing 7.1: Report pipeline

```
projectInputs --> buildReportModel()            (reportModel.js)
              --> buildDesktopStyleLatexReport() (latexReport.js)
              --> .tex source string
              --> SwiftLaTeX Web Worker          (PdfTeX
                  WebAssembly)
              --> PDF ArrayBuffer --> download
```

- `reportModel.js` assembles a report data model from the canonical project inputs and the calculation metadata, including the provenance recorded by the engine (calculation source, core version, Pyodide version, timestamp; Chapter 6).
- `latexReport.js` renders that model into a *desktop-style* `.tex` document: title block, general information, stage-wise cost tables, breakdowns, and appendices mirroring the desktop report's structure (`desktopAppendices.js`, `reportSections.js`).
- `latexReportEngine.js` hands the `.tex` string to the SwiftLaTeX worker and resolves with the compiled PDF.

7.3 The SwiftLaTeX Worker Bridge

The engine bridge resolves all asset URLs against the Vite base path so the same code works at any deployment prefix, and compiles with a hard timeout so a pathological document cannot hang the page:

Listing 7.2: Building the payload and configuring the SwiftLaTeX worker

```
export const buildLatexReportPayload = ({
  projectInputs, results = {}, computedData = {},
  selections = {}, calculationMetadata = {}, fileName,
} = {}) => {
  const report = buildReportModel(projectInputs || {},
                                  calculationMetadata);
  const tex = buildDesktopStyleLatexReport({
    report, results, computedData, selections,
  });
  return { tex, fileName: fileName || DEFAULT_REPORT_FILENAME,
           report, engine: 'swiftlatex' };
};

export const getSwiftLatexReportEngineConfig =
  (basePath = getBasePath()) => ({
    workerUrl: joinAssetPath(basePath, DEFAULT_WORKER_NAME),
```

```
    swiftlatexBase:
      joinAssetPath(basePath, DEFAULT_SWIFTLATEX_ASSET_ROOT),
    texliveMode: 'local',
  });

export const compileLatexReportPdf = ({ tex, fileName, assets =
   {},
  timeoutMs = DEFAULT_TIMEOUT_MS, onStatus,
  config = getSwiftLatexReportEngineConfig(),
} = {}) => new Promise((resolve, reject) => {
  const worker = new Worker(config.workerUrl);
  const timeout = setTimeout(() => {
    worker.terminate();
    reject(new Error(
      `SwiftLaTeX compilation timed out after ${timeoutMs} ms.`))
        ;
  }, timeoutMs);
  // ... message protocol: load engine, write main.tex + assets,
  //     compile, return PDF ArrayBuffer ...
});
```

The main deployment decision is `texliveMode: 'local'`: SwiftLaTeX normally fetches TeX Live packages on demand from a remote service, but here a static, pruned TeX Live footprint is bundled under `public/vendor/swiftlatex/texlive/`, containing exactly the format and font files the desktop-style report needs. Report generation therefore works offline and does not depend on a third-party service, in line with the static deployment of Chapter 6.

7.4 User Flow

On the Outputs page, the user selects report sections (`ReportSectionModal`), and the report is compiled in the worker with progress states surfaced through `onStatus`; the resulting PDF is offered as a download. Because the model embeds calculation provenance, every generated report records which engine (browser WebAssembly or development back-

end), which core version, and which Pyodide version produced its numbers, so the results in a report can be traced back to the engine that produced them.

A standalone smoke page (`swiftlatex-report-smoke.html`) compiles a reference document outside the React app, which was used to bisect TeX Live asset issues during development.

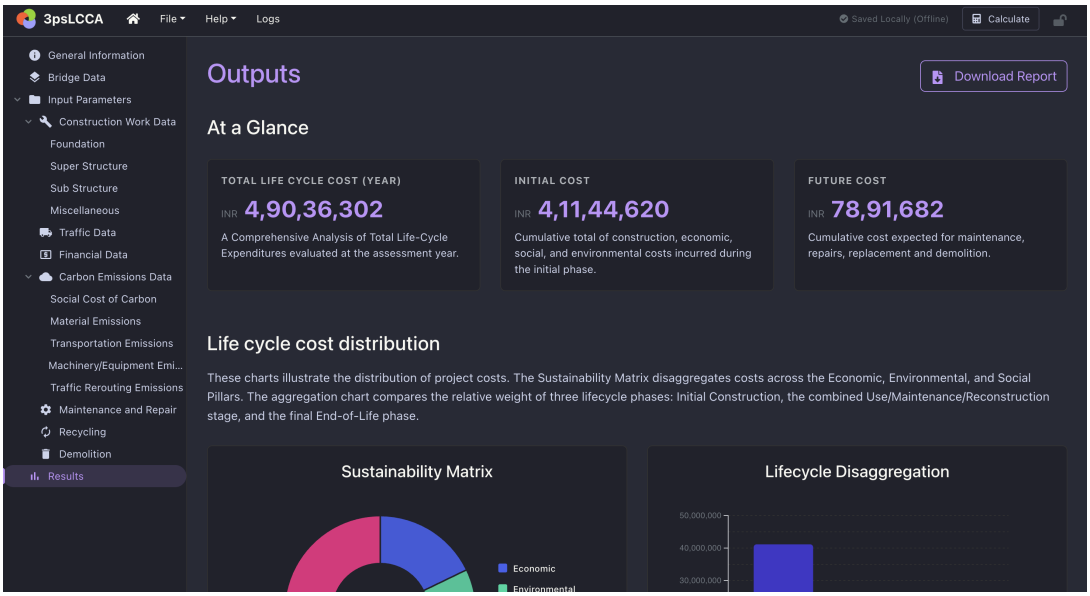

Figure 7.1: Report download action on the Outputs page

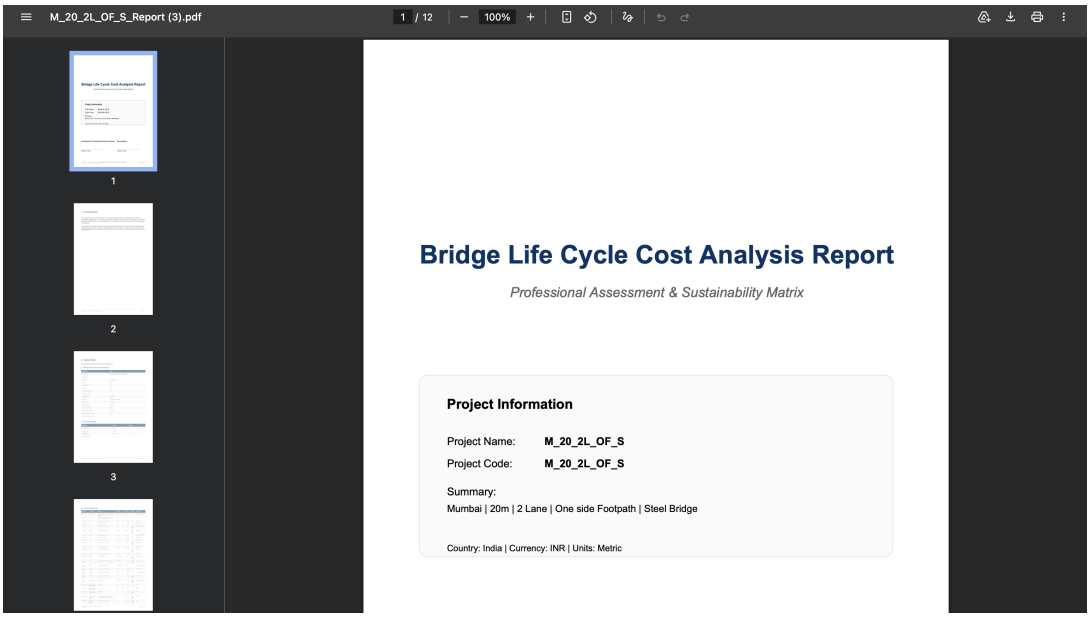

Figure 7.2: Desktop-equivalent LaTeX report compiled in the browser

7.5 Summary

Report generation was the last feature that required a server. With the report model, the desktop-style LaTeX generator, and the SwiftLaTeX worker running against bundled TeX Live assets, the web application produces the same PDF reports as the desktop tool, entirely on the client.

Chapter 8

Conclusions

This fellowship developed the 3psLCCA web platform from an early prototype into a production-ready, fully static application with strict parity to the 3psLCCA desktop tool, and made the shared Python calculation engine runnable in the browser as WebAssembly.

8.1 Tasks Accomplished

- Stabilised the inherited React application shell: homepage search, recent projects, action buttons, profile, sidebar navigation, modal behaviour, and save-state feedback.
- Implemented Appwrite authentication (email/password and Google OAuth), a per-user cloud project database, logout, and a persistent guest mode replacing the fragile checkpoint system.
- Built an offline-first project storage service with last-write-wins conflict resolution, a dynamic save indicator, and `.3ps` import/export interoperability with the desktop application (with Priyansh Vaish).
- Canonicalised the project schema page by page against the desktop application with shared normalisers, and hardened the web-to-core adapter with strict, field-addressed validation backed by pytest suites; removed the divergent JavaScript calculation engine.
- Achieved strict traffic-schema parity including the eight-vehicle classification and

Wholesale Price Index parsing; polished the bridge data schema and adapter constraints.

- Implemented soft deletion with a global trash for construction materials and sections, and Excel import with a validation preview.
- Implemented the static WebAssembly calculation runtime: a build-time asset pipeline packaging `3psLCCA-core` as a wheel with self-hosted Pyodide, a Web Worker runtime with a JSON-only Python bridge, a main-thread engine API, and a native-CPython parity smoke test, which together removed the FastAPI server from production.
- Contributed a reference pytest suite, packaging fixes, and a GitHub Actions CI pipeline (test, wheel build, clean-install verification, checksums, artifacts) to the core repository, plus a standalone `ThreePSLCCA` JavaScript wrapper library with a demo page and integration guide.
- Implemented desktop-equivalent LaTeX report generation entirely in the browser via SwiftLaTeX with a statically bundled TeX Live footprint and calculation provenance embedded in every report.

8.2 Skills Developed

- **Frontend engineering**: React 18 component and context architecture, react-router, Vite build pipelines, Web Workers, and UI state management.
- **WebAssembly and Python packaging**: Pyodide internals (package loading, globals, PyProxy lifetimes), building pure-Python wheels with `setuptools_scm`, micropip, and designing JSON-only language bridges.
- **Backend-as-a-service and data engineering**: Appwrite authentication and databases, offline-first synchronisation patterns, schema design, normalisation, and migration of legacy data.
- **Quality engineering**: pytest fixture design, reference (golden-value) testing, cross-runtime parity testing, Playwright end-to-end tests, and GitHub Actions CI design.

- **Document engineering**: programmatic LaTeX generation and in-browser TeX compilation with SwiftLaTeX.
- **Professional practice**: scoped pull requests, code review with a fellow intern, phased delivery against a desktop reference implementation, and technical writing (the WebAssembly integration guide and this report).

8.3 Future Work

- Publish the core wheel to PyPI and version the web manifest against released wheels rather than local checkouts.
- Extend the parity smoke test into a full golden-file suite run in CI against every core release.
- Add collaborative features on top of the Appwrite layer (shared projects, roles) and project version history restore.
- Progressive Web App packaging so the fully static build installs and runs offline end to end, including report generation.
- Upstream the comparison (multi-project) view from the desktop application into the web platform.

The desktop application, the web application, and the standalone wrapper now share one calculation core, covered by the same tests and distributed as a single wheel, so all three produce the same results.

Chapter A

Appendix

A.1 Technical Reference

The work described in this report is available at the following repositories and branches:

- **Web application (all contributions consolidated):**
  swas02/3psLCCA-web — Pull Request #10 (`Swayam200:main` → `swas02:main`; +618,524 / –3,498 across 744 files, 36 commits, with Priyansh Vaish)
- **WebAssembly-enabled web branch:**
  github.com/Swayam200/3psLCCA-web (branch `feat/wasm-static-web`, merged to `main`)
- **Core engine contributions (tests, CI, wrapper, demo):**
  github.com/Swayam200/3psLCCA-core (branch `feat/wasm-static-runtime`)
- **Screening task:**
  Nidhikhare12/OsdagBridge — Pull Request #5 (branch `screening-task-plot`)

A.2 Work Reports

Internship Work Report			
Name:	Swayam Prakash Panda		
Project:	Osdag (3psLCCA)		
Internship:	FOSSEE Summer Fellowship 2026 (15 May – 6 July 2026)		
DATE	**DAY**	**TASK**	**Hours**
15-May-2026	Friday	Fellowship kickoff; mentor meeting with Swastik Gupta; understood 3psLCCA project scope and repository layout	4
16-May-2026	Saturday	Set up and explored the 3psLCCA desktop application (PySide6) to learn the reference data-entry workflow	4.5
17-May-2026	Sunday		
18-May-2026	Monday	Studied the 3psLCCA-core engine: input schema, `run_full_lcc_analysis()` flow, and stage-wise outputs	5
19-May-2026	Tuesday	Set up the 3psLCCA-web development environment (Node, Vite); explored the inherited React codebase	4.5
20-May-2026	Wednesday	Compared web pages against desktop widgets field by field; documented parity gaps	5
21-May-2026	Thursday	Triaged broken flows in the homepage (search bar, recent projects, action buttons) and profile	4
22-May-2026	Friday	Planned the UI stabilisation roadmap with mentor; prioritised fixes and phase plan	4.5
23-May-2026	Saturday	Began implementing homepage fixes locally; tested search and recent-projects behaviour	5
24-May-2026	Sunday		
25-May-2026	Monday	Fixed search bar, recent projects, and all buttons in the home page	5
26-May-2026	Tuesday	Fixed profile functionality; fixed the three-button project card actions in the homepage	5.5

DATE	DAY	TASK	Hours
27-May-2026	Wednesday	Reviewed and tested the sidebar dropdown and general-information fix branch (with Priyansh); merge support	4.5
28-May-2026	Thursday	Investigated checkpoint-storage limitations; designed the persistent guest-storage approach	5
29-May-2026	Friday	Removed checkpoints and added persistent local storage for Guest Mode	6
30-May-2026	Saturday	Added Appwrite email & Google auth, project database syncing, and user logout	6.5
31-May-2026	Sunday		
1-Jun-2026	Monday	Added lock-scroll fix, dynamic save indicator, refresh-overwrite bug fix, login page logo/colour update; fixed guest project pinning, currency auto-fill, and project info modal	6
2-Jun-2026	Tuesday	Tested authentication flows end to end; handled session edge cases and error states	4.5
3-Jun-2026	Wednesday	Designed the offline-first sync strategy (localStorage-first, last-write-wins via timestamps)	5
4-Jun-2026	Thursday	Implemented robust offline-first storage sync and unlocked the Outputs page	6
5-Jun-2026	Friday	Standardised the General Information page with desktop & profile integration; standardised the project schema and integrated the Python backend for calculation	6.5
6-Jun-2026	Saturday	Verified the standardised schema against desktop reference projects; fixed mapping mismatches	5
7-Jun-2026	Sunday		
8-Jun-2026	Monday	Implemented phase 2 page-by-page canonicalisation and shared normalisers	6

DATE	DAY	TASK	Hours
9-Jun-2026	Tuesday	Phase 5 adapter hardening: strict validation, backend tests, removed the JS calculation engine	6
10-Jun-2026	Wednesday	Extended adapter pytest coverage; fixed validation failures surfaced by the new tests	5
11-Jun-2026	Thursday	Cross-checked web calculation results against desktop reference outputs	5
12-Jun-2026	Friday	Polished bridge data schema and adapter constraints; implemented soft deletion for construction materials/sections; construction Excel import and global trash UI	6.5
13-Jun-2026	Saturday	Tested Excel import against sample construction workbooks; refined the import preview validation	4.5
14-Jun-2026	Sunday		
15-Jun-2026	Monday	Studied desktop WPI (Wholesale Price Index) handling and vehicle-grouped WPI structure	5
16-Jun-2026	Tuesday	Reviewed result-section validation messages (Priyansh); planned traffic tab parity changes	4.5
17-Jun-2026	Wednesday	Implemented groundwork for strict WPI parsing against `wpi_db.json`	5
18-Jun-2026	Thursday	Traffic schema strict parity and WPI parsing; lint cleanup and traffic logic updates	6
19-Jun-2026	Friday	Reviewed and tested `.3ps` import support and signed-in open-project fix (Priyansh)	4.5
20-Jun-2026	Saturday	Researched Pyodide architecture, wheel packaging, and Web Worker execution for the WASM runtime	5
21-Jun-2026	Sunday		
22-Jun-2026	Monday	Prototyped loading the core wheel in Pyodide; validated a first in-browser calculation	5.5
23-Jun-2026	Tuesday	Built the `prepare-wasm-assets` pipeline (wheel build, self-hosted Pyodide, manifest, checksums)	6

DATE	DAY	TASK	Hours
24-Jun-2026	Wednesday	Implemented the Python JSON bridge and adapter source injection into the worker	6
25-Jun-2026	Thursday	Implemented `lcca.worker.js` and `wasmEngine.js` request/response protocol with status reporting	6
26-Jun-2026	Friday	Built the WASM smoke test comparing browser results against native CPython reference outputs	5.5
27-Jun-2026	Saturday	Implemented the static WebAssembly LCCA runtime in the web app; added the WebAssembly runtime test foundation to 3psLCCA-core	7
28-Jun-2026	Sunday		
29-Jun-2026	Monday	Explored SwiftLaTeX; assembled the static bundled TeX Live footprint under `public/vendor`	5.5
30-Jun-2026	Tuesday	Built the report model and the desktop-style LaTeX report generator	6
1-Jul-2026	Wednesday	Implemented the SwiftLaTeX Web Worker compilation pipeline with timeout and status handling	6
2-Jul-2026	Thursday	Wired report provenance (engine source, core version, Pyodide version, timestamp) into generated reports	5
3-Jul-2026	Friday	Wrote the core reference pytest suite (global, India+WPI, zero-traffic, invalid-input cases) and set up the GitHub Actions CI pipeline (wheel build, clean-install verification, checksums)	6
4-Jul-2026	Saturday	Built the standalone `ThreePSLCCA` JS wrapper library, CDN demo page, and WASM integration guide	6.5
5-Jul-2026	Sunday		
6-Jul-2026	Monday	Completed desktop-equivalent LaTeX report via SwiftLaTeX in-browser compilation; merged main and resolved schema conflicts; opened consolidated PR #10 to upstream	7.5

Bibliography

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

[2] FOSSEE Project. FOSSEE News - January 2018, vol 1 issue 3. Accessed: 2026-07-06.

[3] FOSSEE Project. Osdag website. Accessed: 2026-07-06.