FOSSEE Autumn Long Internship Report

On

Refactoring of Osdag GUI, Development of Plugin Management System, CLI Module and maintenance of Windows Installer for Osdag

Submitted by

Mehendi Hasan

4th Year B.Sc(Research) Physics Major, Computer Science Minor

Kirori Mal College

University of Delhi

Under the Guidance of

Prof. Siddhartha Ghosh

Department of Civil Engineering

Indian Institute of Technology Bombay

Mentors:

Ajmal Babu M S

Parth Karia

Ajinkya Dahale

January 23, 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 summarises 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, 3, 5]. Built on Python and several Python-based FLOSS tools (e.g., PyQt and PythonOCC), Osdag is licensed under the GNU Lesser General Public License (LGPL) Version 3.

1.3.1 Osdag GUI

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

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

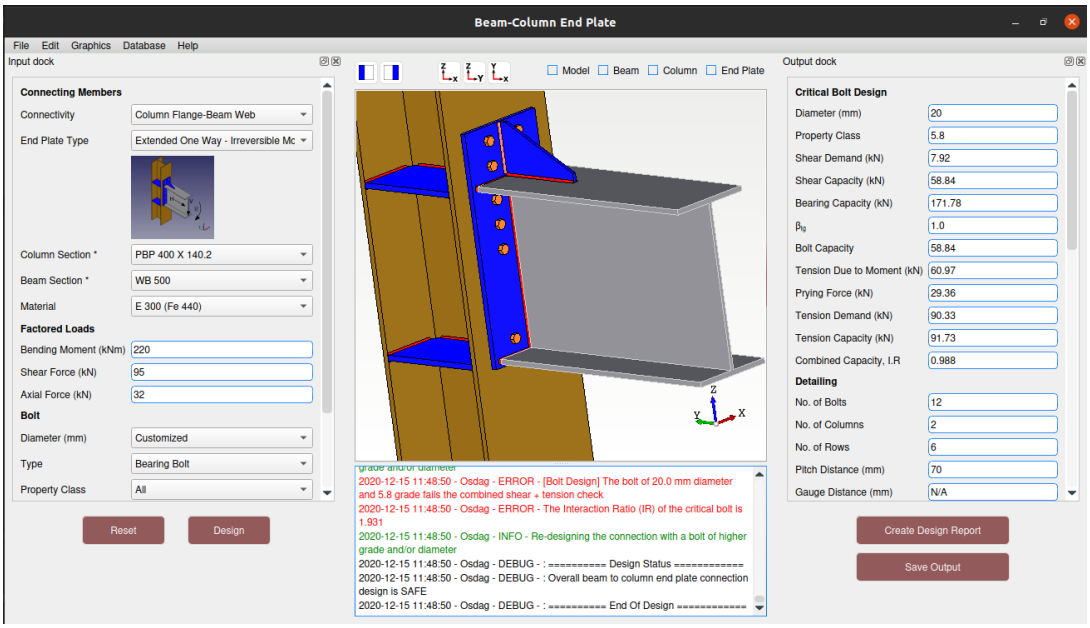

Figure 1.2: Osdag GUI

1.3.2 Features

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

For more details, visit the official Osdag website.

Chapter 2

Internship Task 1: Osdag Command Line Interface (CLI) Module Development

Osdag primarily provides a graphical user interface (GUI) for steel design workflows. However, for automation, testing, and batch processing use cases, a command-line interface (CLI) becomes essential. As part of this task, a CLI module was developed to allow Osdag designs to be executed directly from the terminal using existing input files (`.osi`).

The CLI enables users to:

- Execute Osdag design modules using existing `.osi` input files
- Extract design output parameters directly in the terminal
- Generate CSV reports programmatically
- Generate complete PDF design reports without launching the GUI

This functionality is particularly useful for automated testing, batch processing, and integration with external pipelines.

2.1 OSI File Workflow in Osdag

Osdag uses `OSI` files (e.g., `xyz.osi`) as structured input files for all design modules. These files follow a YAML-based format and contain:

- The module identifier to be executed

- All required input dock parameters for that module

Once an OSI file is imported into Osdag, its contents are converted into a `design_dictionary`. Each Osdag module class internally maintains such a dictionary, allowing the same workflow to be reused across GUI and CLI executions.

An example OSI file is provided in Appendix A.1.

2.2 CLI Module Architecture

The CLI backend is implemented as a dedicated module inside `osdag_core`. This design choice ensures that CLI functionality remains independent of the GUI layer and can be reused for automated testing or headless execution environments.

The CLI module performs the following responsibilities:

- Maps OSI module identifiers to their corresponding design classes
- Loads and validates OSI files
- Executes design calculations
- Extracts output data in a structured form
- Routes output to the terminal, CSV files, or PDF reports

2.2.1 Module Registry via `available_modules`

A central component of the CLI module is the `available_modules` dictionary, which maps module display keys to their corresponding design classes.

```
available_modules = {
    KEY_DISP_BASE_PLATE: BasePlateConnection,
    KEY_DISP_CLEATANGLE: CleatAngleConnection,
    KEY_DISP_TENSION_BOLTED: Tension_bolted,
    ...
}
```

Listing 2.1: Mapping OSI module keys to design classes

This explicit mapping ensures deterministic module resolution during CLI execution. While this approach requires manual updates when new modules are added, it provides clarity and safety during early-stage CLI development.

In future iterations, dynamic discovery mechanisms could be explored to automate this mapping.

2.3 Atomic Helper Methods in the CLI Module

The CLI module is composed of small, atomic helper functions designed to perform a single responsibility. These functions are reusable, testable, and composable.

2.3.1 OSI Parsing and Validation

`_get_design_dictionary` reads and parses the OSI file into a Python dictionary using safe YAML loading. This dictionary is passed directly to the module validation logic, ensuring parity with GUI-based workflows.

2.3.2 Output Extraction Logic

The method `_get_output_dictionary` extracts computed design results from the module instance. It selectively processes:

- Text-based output fields
- Computed outputs triggered by internal output buttons

This logic mirrors how output is generated in the GUI, ensuring consistent reporting across interfaces.

2.3.3 Report and CSV Generation

Two dedicated helper functions handle report generation:

- `_generate_csv`: Converts output dictionaries into CSV format
- `_generate_report`: Triggers the internal PDF report generation pipeline

By reusing the module's existing report generation APIs, the CLI avoids duplicating reporting logic.

2.4 Core Execution Flow: `run_module`

The `run_module` function serves as the orchestration layer for CLI execution. It performs the following steps:

1. Validates the provided OSI file path
2. Resolves the target module class
3. Initializes the module and validates inputs
4. Executes the design computation
5. Dispatches output based on the selected operation type

Supported operation modes include:

- `print_result` — display output in the terminal
- `save_csv` — generate a CSV report
- `generate_report` — generate a full PDF design report

The method returns a structured result dictionary, enabling future integration with automated test suites.

2.5 CLI Integration with Osdag Application

The CLI is integrated into the Osdag application via the `click` library. Both GUI and CLI entry points are unified under a single executable.

2.5.1 Unified Entry Point

The application entry point resides in `__main__.py`. If no CLI subcommand is provided, the GUI is launched by default.

```
@click.group(invoke_without_command=True)
def main(ctx):
    if ctx.invoked_subcommand is None:
        GUI()
```

Listing 2.2: Default GUI launch behavior

This design ensures backward compatibility while enabling CLI-based workflows.

2.5.2 CLI Namespace Structure

The CLI functionality is grouped under the `cli` namespace, providing commands such as:

- `osdag cli`
- `osdag cli run`

This hierarchical structure avoids namespace pollution and leaves room for future CLI extensions.

2.5.3 CLI Argument Handling

The `run` subcommand defines required and optional arguments, including:

- Input OSI file path (mandatory)
- Operation type (CSV, PDF, or terminal output)
- Optional output file path

These arguments are validated by `click` before execution, reducing runtime error handling complexity.

2.6 Change in Application Entry Points

Prior to refactoring, Osdag launched exclusively through a GUI-centric entry function. With the December 2025–January 2026 refactor, a unified entry point was introduced via `__main__.py:main()`, enabling both GUI and CLI execution paths from a single executable.

This architectural change simplifies deployment and ensures consistent startup behavior across usage modes.

2.7 Future Scope of the CLI Module

The CLI module opens several avenues for future development:

- Automatic discovery of available modules
- Headless execution for CI-based testing
- Batch processing of multiple OSI files
- Programmatic generation of CAD designs

By residing within `osdag_core`, the CLI remains a first-class interface to Osdag's design engine, independent of GUI constraints. This cli module should be refactored in such a way it resides only in `osdag_core` and not in `osdag_gui`, enabling seamless integration with automated workflows and external systems.

2.8 Source Code Reference

The complete implementation of the CLI module is provided in Appendix A.2 for reference and maintenance purposes.

Chapter 3

Internship Task 2: In-Application Update Mechanism for Osdag

The refactored Osdag application (Dec 2025-Jan 2026 release) introduces a built-in *Check for Updates* feature to streamline the upgrade process for end users. The primary objective of this feature is to eliminate the need for manual downloads and reinstallation by enabling version checks and updates directly from within the application.

This functionality is particularly important given Osdag's distribution through multiple package managers (such as Conda and Pixi), each of which requires different invocation strategies while maintaining a consistent user experience.

3.1 Purpose and Design Overview

The update system is designed to:

- Detect newer versions of Osdag available in the configured distribution channel
- Compare the installed version with the latest available version
- Allow users to initiate updates from within the GUI
- Perform update operations asynchronously without blocking the interface
- Handle platform-specific executable paths transparently

The implementation deliberately separates **version discovery**, **user interaction**, and **update execution** to ensure maintainability and extensibility.

3.2 Code Overview

3.2.1 File: `osdag_gui/ui/components/dialogs/check_for_updates.py`

This file implements a self-contained dialog responsible for checking and applying application updates. It interacts with external package managers using `QProcess`, allowing update commands to be executed asynchronously while preserving GUI responsiveness.

The dialog does not directly embed package-manager-specific logic throughout the code. Instead, such differences are isolated in clearly defined decision points based on the detected installation type.

3.3 UpdateDialog Class

3.3.1 Initialization and State Setup

The `UpdateDialog` class is responsible for initializing the update UI, capturing the current application state, and triggering the version check process.

Key initialization steps include:

- Capturing the currently installed version
- Detecting internet connectivity
- Configuring the dialog layout and visual theme
- Resolving executable paths for package managers

```
1 self.old_version = Version(VERSION)
2 self.internet_connectivity = QApplication.instance().
    internet_connectivity
```

Listing 3.1: Capturing application state during dialog initialization

The use of `packaging.version.Version` ensures reliable semantic version comparison rather than fragile string-based checks.

3.3.2 Executable Path Resolution

Osdag may be installed using different package managers and across multiple platforms. The dialog dynamically resolves the executable paths for Conda and Pixi at runtime.

```
if sys.platform.startswith("win"):
    self.conda_path = base_conda_path / "Scripts" / "conda.exe"
else:
    self.conda_path = base_conda_path / "bin/conda"
```

Listing 3.2: Resolving package manager executable paths

Fallback resolution using `shutil.which()` ensures that updates remain functional even when executables are not located in standard paths.

3.4 Asynchronous Update Check

3.4.1 Rationale for Using QProcess

All version checks and update operations are executed using `QProcess` rather than Python subprocess calls. This decision provides:

- Native Qt event-loop integration
- Non-blocking execution
- Real-time capture of standard output and error streams

3.4.2 Launching the Update Check

The update check is initiated immediately after dialog creation. Internally, the dialog selects the appropriate update strategy based on the installation type detected at runtime.

Osdag currently supports two distinct installation modes:

- **Conda-based installation**, where Osdag is installed into an existing Conda environment
- **Pixi-based installation**, used by the installer, where no system-wide Conda installation is required

The Pixi-based workflow is particularly important for users installing Osdag via the official installer, as it provides an isolated and self-contained environment. In such cases, the update mechanism must function without assuming the presence of Conda on the host system.

```
if INSTALLATION_TYPE == "conda":
    cmd = [
        self.conda_path,
        "search",
        "-c", "conda-forge",
        "osdag::osdag",
        "--info",
        "--json"
    ]
elif INSTALLATION_TYPE == "pixi":
    cmd = [
        self.pixi_path,
        "search",
        "osdag",
        "--channel",
        "osdag"
    ]
```

Listing 3.3: Selecting update check command based on installation type

In the **Conda-based installation**, version metadata is retrieved in structured JSON format, allowing reliable extraction and comparison of available versions.

For the **Pixi-based installation**, version information is obtained through command-line output parsing. This approach enables update checks even in environments where Conda is not available, which is essential for installer-based deployments.

At the time of implementation, the Conda-based update workflow has been tested more extensively, while the Pixi-based update path is functional but has not yet undergone exhaustive validation across platforms. The code structure explicitly anticipates further testing and refinement of the Pixi workflow as installer adoption increases.

By centralizing installation-specific logic at the command-selection stage, the remain-

der of the update pipeline—including asynchronous execution, output handling, and user interface updates—remains installation-agnostic. This design ensures that support for both installation modes can evolve independently without requiring major architectural changes.

3.4.3 Processing Update Check Results

Once the external process completes, the output is parsed to extract the latest available version. The installed version is then compared against the retrieved version.

```
lv = Version(latest_version)
if lv > self.old_version:
    # Update available
```

Listing 3.4: Comparing installed and latest versions

This comparison drives subsequent UI state changes, such as enabling update controls or displaying confirmation messages.

3.5 User Interaction and Update Execution

3.5.1 Conditional UI State Management

The dialog dynamically adjusts available actions based on the update state:

- No update available → Informational message only
- Update available → Show *Update Now* and *Update Later* buttons

This prevents users from initiating invalid operations and simplifies the interaction flow.

3.5.2 Executing the Update Process

When the user chooses to update, the dialog launches a new `QProcess` to execute the update command asynchronously.

```
self.process = QProcess(self)
self.process.start(cmd[0], cmd[1:])
```

Listing 3.5: Launching the update command asynchronously

During execution:

- Progress indicators are shown
- Output is streamed to the dialog
- User controls are temporarily disabled

3.5.3 Handling Completion and Errors

Upon completion, the dialog evaluates the process exit code to determine success or failure.

```
exit_code = self.process.exitCode()
if exit_code == 0:
    self.update_text("Update completed successfully!")
```

Listing 3.6: Handling update completion

This approach allows graceful recovery from failures and clear communication to the user.

3.6 Process Cleanup and Safety

To prevent orphaned processes or resource leaks, the dialog explicitly terminates any running update process when closed.

```
if self.process and self.process.state() != QProcess.NotRunning:
    self.process.kill()
```

Listing 3.7: Ensuring safe termination of update processes

This safeguard ensures predictable application behavior even if the user closes the dialog mid-operation.

3.7 Design Considerations and Extensibility

The update system is designed to be extensible:

- Additional package managers can be supported by extending command selection logic
- Progress reporting can be enhanced by parsing structured output
- Automatic update scheduling can be introduced without redesigning the UI

By relying on Qt-native asynchronous mechanisms and clear separation of concerns, the update feature integrates seamlessly into the Osdag application architecture.

3.8 Source Code Reference

The complete implementation of the update dialog is provided in Appendix B for reference and maintenance purposes.

Chapter 4

Internship Task 3: Development of Plugin Manager

The Osdag application is being extended with a plugin-based architecture to improve modularity, scalability, and long-term maintainability. The primary objective of this system is to allow independent functional modules to be developed, distributed, installed, activated, and managed without requiring modifications to the core Osdag codebase.

The plugin management system serves as the backbone of this architecture. It is responsible for discovering plugins from multiple sources, maintaining their lifecycle state, enabling or disabling them at runtime, and ensuring persistence of plugin activation states across application restarts. This design allows Osdag to evolve as an extensible platform rather than a monolithic application.

The source code for the plugin manager developed during this task is available on the Osdag GitHub repository (subject to change; please contact FOSSEE for access) [4, 2].

4.1 Overview of the Plugin Management System

The plugin management system is implemented as a layered design consisting of:

- A **core management layer** responsible for plugin discovery, lifecycle operations, and active state management.
- A **user interface layer** that provides dialogs for managing local plugins and interacting with the plugin store.

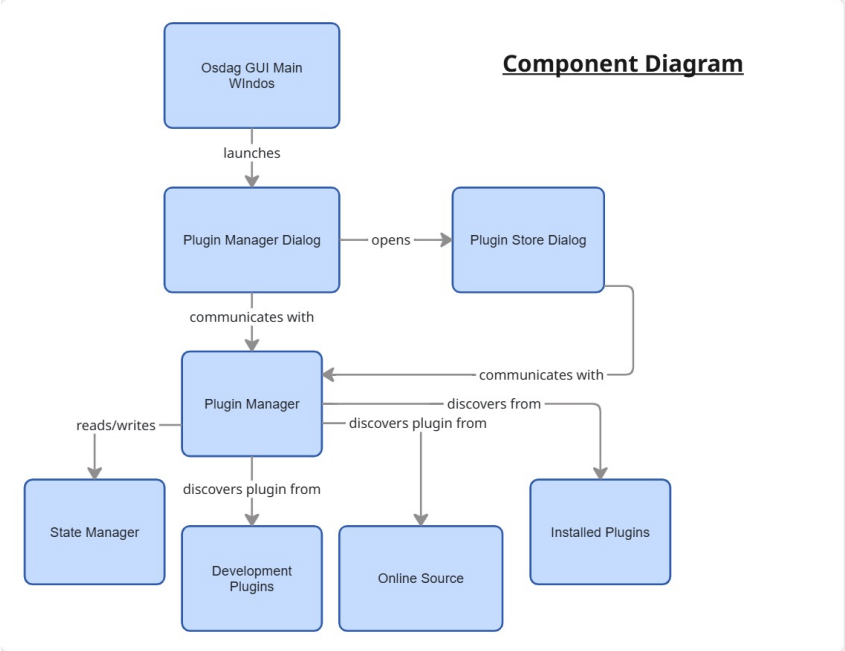

Figure 4.1: Architecture of the Plugin Management System

From an architectural standpoint, the core layer is independent of the graphical user interface. This separation ensures that plugin management logic remains reusable, testable, and unaffected by UI-specific changes.

4.2 Relevant Source Files

The plugin management system is implemented across the following Python files:

- `osdag_core/utils/plugin_manager.py`
- `osdag_gui/ui/components/dialogs/plugin_manager_dialog.py`
- `osdag_gui/ui/components/dialogs/plugin_store_dialog.py`

The correct sequence for understanding the system begins with the core plugin manager, followed by the local plugin management dialog, and finally the plugin store dialog. Accordingly, this chapter first describes the implementation of `plugin_manager.py`, which defines the fundamental plugin abstraction and lifecycle logic used throughout the application.

4.3 Core Plugin Management Implementation

4.3.1 Plugin Manager Module

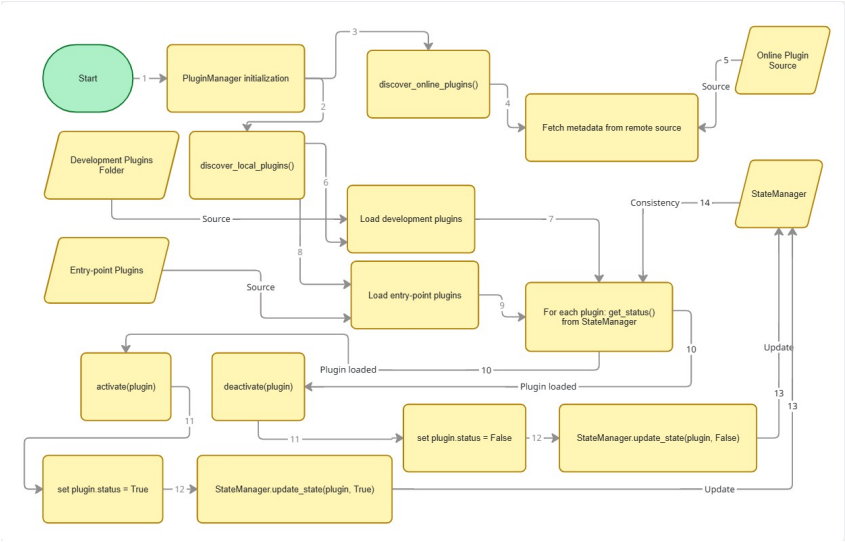

Figure 4.2: Plugin Manager Module Code Structure

This file implements the core plugin infrastructure and is independent of any graphical user interface. It defines the plugin data structure, plugin discovery mechanisms, lifecycle operations, and persistent active state handling.

PluginMetaData Class

The `PluginMetaData` class is a dataclass that represents the canonical description of a plugin within the Osdag ecosystem. All plugins, regardless of their source (installed, development, or online), are normalized into this structure.

Each instance contains:

- A unique identifier (`id`) used for persistence and lookup.
- Metadata such as name, description, authors, and version.
- Runtime fields such as activation status, imported module reference, and entry class.
- A development flag (`is_dev`) to distinguish development plugins from packaged plugins.

This unified representation allows the rest of the system to treat all plugins uniformly. Each plugin is self-descriptive and contains all necessary information for management and execution. In future extensions, additional metadata fields can be added without impacting existing functionality.

PluginManager Class

The `PluginManager` class acts as the central authority for all plugin-related operations. It is responsible for discovery, activation, deactivation, installation, removal, and update of plugins. While the `PluginManagerDialog` class in the GUI layer provides user interaction, the `PluginManager` class encapsulates all core logic.

Initialization During initialization, the plugin manager:

- Instantiates a `StateManager` to handle persistent plugin states.
- Initializes an internal list to store discovered plugins.
- Resolves the filesystem path for development plugins.

This setup ensures that the manager is ready to perform discovery operations immediately after initialization.

Local Plugin Discovery The method `discover_local_plugins()` does the discovery of all locally available plugins(installed plugins). It performs the following steps:

1. Loads installed plugins registered via Python entry points(osdag.plugin inside site-packages folder along with osdag module itself).
2. Loads under development plugins from the local filesystem.
3. Restores activate states using the state manager.

So each time the plugin manager is asked to discover local plugins, it refreshes its internal list to reflect the current state of the system.

Entry Point Plugin Loading Installed plugins are discovered using Python's entry point mechanism under the group `osdag.plugins`. Each candidate plugin must explicitly declare itself as an Osdag plugin and provide required metadata and an entry point class, which will discuss in subsequent chapters.

Invalid or incomplete plugins are skipped gracefully, ensuring that a single faulty plugin cannot disrupt the discovery process.

Development Plugin Loading Under development plugins are discovered by scanning a dedicated plugins directory within the Osdag source tree. This mechanism is intended to support rapid development and testing without requiring packaging or installation.

To prevent ambiguity, plugins already loaded via entry points are not reloaded from the development directory.

In future enhancements, additional discovery sources (e.g., user-defined directories) will be supported to further increase flexibility.

Entry Class Resolution For both installed and development plugins, the plugin manager resolves the plugin entry class dynamically using the path specified in the plugin metadata. This resolution step ensures that only plugins with valid and accessible entry points are registered.

Online Plugin Discovery

The method `discover_online_plugins()` retrieves plugin metadata from a Conda channel by fetching and parsing its repository metadata. The discovered plugins are converted into `PluginMetaData` objects but are not executable until installed locally.

This functionality enables users to browse and install new plugins directly from within the Osdag application. The plugin store dialog utilizes this method to present available plugins to the user. The current implementation fetches metadata from a specified channel and constructs plugin representations based on package information. Which is not very robust and can be improved in future iterations. Like adding support for multiple channels, pagination, and search/filtering capabilities. The current implementation is mere a proof of concept.

Plugin Lifecycle Operations

The plugin manager provides explicit methods for:

- Activating plugins by updating their runtime status and persistent state.
- Deactivating plugins in a reversible manner.
- Installing plugins using Conda package management.
- Removing installed plugins and cleaning up associated state.
- Updating plugins to newer available versions.

All package management operations are executed via subprocess calls to Conda, ensuring compatibility with Osdag's existing environment management strategy(conda environments). Each operation provides feedback on success or failure, allowing the GUI layer to inform the user accordingly.

StateManager Class

The `StateManager` class is responsible solely for persisting plugin activation states. It maintains a mapping between plugin identifiers and their activation status and stores this information in a JSON file. Not the effective way of storing states but it is simple and easy to implement. Can be improved in future iterations by using databases or other structured storage mechanisms.

State updates are protected using a thread lock, and file writes are performed atomically to prevent corruption. This guarantees reliable recovery of plugin states even in the event of unexpected application termination.

4.3.2 Plugin Manager Dialog - GUI interface

This file implements the graphical user interface responsible for managing locally available plugins. While the core plugin manager encapsulates all discovery and lifecycle logic, this dialog acts as a controller that connects user interactions to the core plugin manager API.

The dialog is intentionally designed to be lightweight: it does not perform plugin discovery logic itself, nor does it manipulate the filesystem or package manager directly.

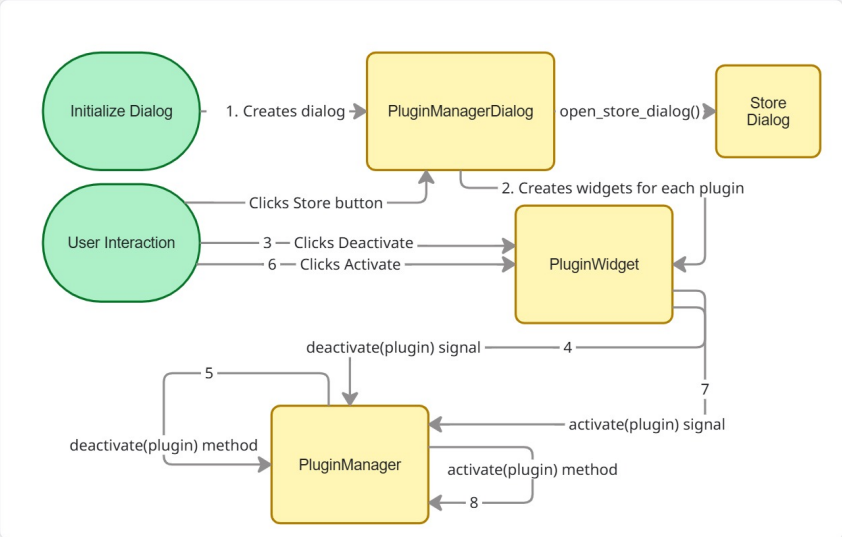

Figure 4.3: Plugin Manager Dialog Code Structure

Instead, it delegates all such responsibilities to the `PluginManager` and focuses on visualization, user interaction, and state synchronization.

High-Level Responsibilities

The Plugin Manager Dialog is responsible for:

- Displaying all locally discovered plugins and their metadata
- Allowing users to activate or deactivate plugins at runtime
- Providing controls to delete installed plugins(should be done be plugin store dialog ideally)
- Serving as an entry point to the Plugin Store dialog
- Persisting plugin activation state on application exit

StatusIndicator Widget

The `StatusIndicator` widget provides a compact visual representation of a plugin's activate state. It is implemented as a standalone widget to avoid duplication of status-handling logic across the interface.

```
1 class StatusIndicator(QWidget):
2     def update_status(self, status: bool) -> None:
```

```
        if status:
            self.circle.setStyleSheet(
                "border-radius: 6px; background-color: green;")
            self.text.setText("Active")
        else:
            self.circle.setStyleSheet(
                "border-radius: 6px; background-color: red;")
            self.text.setText("Inactive")
```

Listing 4.1: Visual status indicator for plugin activation state

Separating this logic into a dedicated component allows future extensions such as intermediate states (e.g., loading, error) without modifying the main plugin widget.

PluginWidget: Representation of a Single Plugin

Each locally available plugin is represented by a `PluginWidget`. This widget encapsulates the presentation of plugin metadata along with user actions, while remaining agnostic of the underlying plugin management logic.

Signal-Based Interaction Rather than directly invoking core methods, the widget emits intent-based signals:

```
class PluginWidget(QWidget):
    activate = Signal(PluginMetaData)
    deactivate = Signal(PluginMetaData)
```

Listing 4.2: Signals emitted by PluginWidget to express user intent

This design decouples the UI from the plugin manager implementation and allows the dialog to act as an intermediary controller.

Activation and Deactivation Logic The widget maintains a local visual state and emits the appropriate signal when the user interacts with activation controls.

```
def _emit_activate(self):
    if not self.plugin.status:
        self.plugin.status = True
```

```
4           self.status_indicator.update_status(True)
5           self.activate.emit(self.plugin)
6
7   def _emit_deactivate(self):
8       if self.plugin.status:
9           self.plugin.status = not self.plugin.status
10          self.status_indicator.update_status(self.plugin.status)
11          self.deactivate.emit(self.plugin)
```

Listing 4.3: Activation logic within PluginWidget

PluginManagerDialog: Controller Logic

The `PluginManagerDialog` class serves as the controller that coordinates between UI widgets and the core plugin manager.

Initialization During initialization, the dialog:

- Retrieves the shared `PluginManager` instance from the application
- Discovers all locally available plugins
- Restores activation states
- Dynamically creates UI widgets for each plugin
- Adds them to the dialog's layout
- Connects each widget's signals to core(`PluginManager`) lifecycle methods
- Connects signals to dialog-level handlers for additional state synchronization

```
1 self.plugin_manager = QApplication.instance().plugin_manager
2 self.plugins = self.plugin_manager.discover_local_plugins()
```

Listing 4.4: Initialization and plugin discovery in the dialog

Signal Connection and Lifecycle Control Signals emitted by individual `PluginWidget` instances are connected to both core lifecycle methods and dialog-level handlers:

```
pw.activate.connect(self.plugin_manager.activate)
pw.activate.connect(self.activate_plugin)
pw.deactivate.connect(self.plugin_manager.deactivate)
pw.deactivate.connect(self.deactivate_plugin)
pw.delete.connect(self.plugin_manager.delete_plugin)
```

Listing 4.5: Connecting plugin widget signals to core logic and GUI handlers

This dual-connection strategy ensures that:

- The core plugin state is updated
- The dialog's internal representation of active plugins remains synchronized
- The user interface reflects changes immediately

Integration with Application Modules

Activated plugins are registered in the application's module registry, enabling them to appear in relevant menus or toolbars. The module registry is a list of tuples containing the plugin name and an associated icon path. This integration allows activated plugins to be seamlessly incorporated into the application's workflow. As shown below, when a plugin is activated, its name and icon are appended to the `active_plugins` list in the application data. This way the plugin becomes accessible through the main menu of the osdag as `Add-Ons`.

```
self.active_plugins: list[tuple[str, str]] = Data.MODULES["Add-Ons
    "]

def activate_plugin(self, plugin: PluginMetaData):
    plugin = self.plugin_manager.get_plugin(plugin.id)
    if plugin and (plugin.name, ":/vectors/IITB_logo.svg") not in
        self.active_plugins:
         self.active_plugins.append(
             (plugin.name, ":/vectors/IITB_logo.svg"))
```

Listing 4.6: Registering active plugins with the application

```
def deactivate_plugin(self, plugin: PluginMetaData):
    self.active_plugins[:] = [
        p for p in self.active_plugins if p[0] != plugin.name
    ]
```

Listing 4.7: Deregistering deactivated plugins from the application

Plugin Store Integration

The dialog provides access to the Plugin Store through a dedicated button. The store dialog is launched modally to ensure that plugin installation or removal operations complete before further interaction with local plugins.

```
def open_store_dialog(self):
    store_dialog = PluginStoreDialog()
    store_dialog.exec()
```

Listing 4.8: Launching the Plugin Store dialog

Upon closing the store dialog, the local plugin list can be refreshed to reflect newly installed or removed plugins.

State Persistence

Before the dialog is closed, the plugin states in the plugin manager are flushed(saved) to disk to ensure that activation changes persist across application restarts.

```
def closeEvent(self, event):
    self.plugin_manager.state_manager.flush()
    super().closeEvent(event)
```

Listing 4.9: Persisting plugin state on dialog closure

This explicit persistence step guarantees deterministic restoration of plugin states on subsequent launches.

4.4 Plugin Store Dialog - Online Plugin Distribution and Management

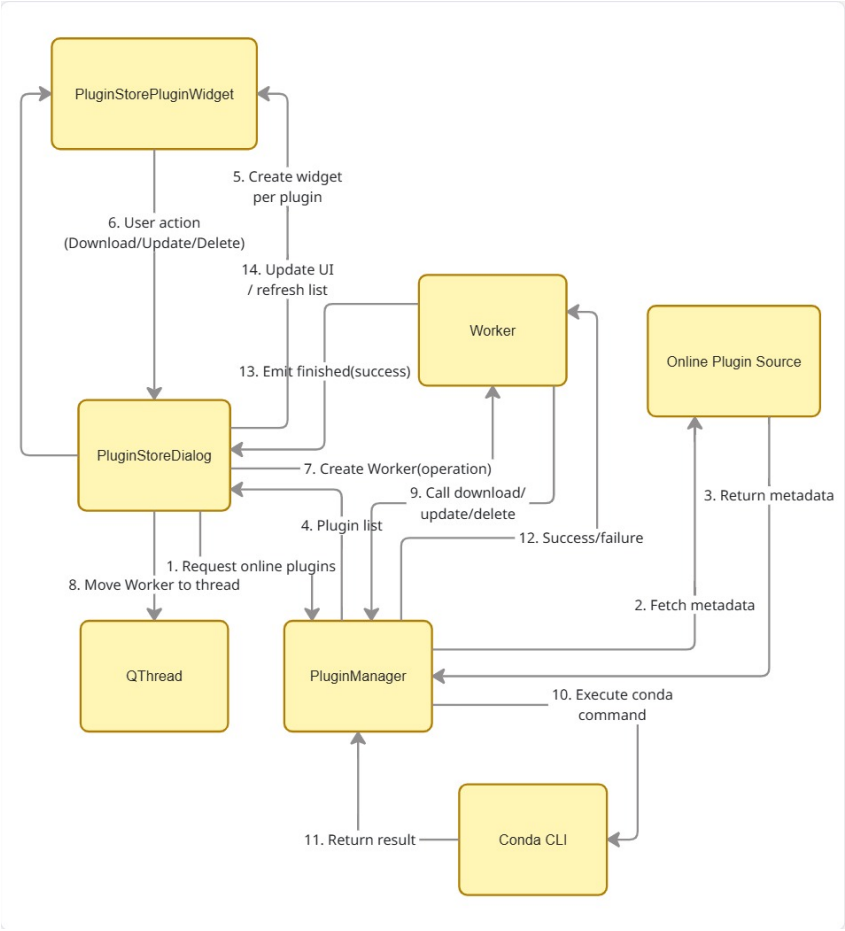

Figure 4.4: Plugin Store Dialog Code Structure

While the Plugin Manager dialog is responsible for managing locally available plugins, the Plugin Store dialog extends this functionality by enabling discovery, installation, deletion, and updates of plugins distributed through an external package repository. This dialog acts as the bridge between the Osdag application and the external plugin distribution channel.

The Plugin Store is intentionally implemented as a separate dialog to isolate long-running and potentially failure-prone operations (such as network access and package installation) from the core plugin management interface.

Design Goals

The Plugin Store dialog was designed with the following goals:

- Support discovery of plugins available in an online repository
- Allow safe installation, deletion, and updating of plugins
- Prevent blocking of the main GUI thread during package operations
- Provide clear feedback to users regarding operation status
- Show relevant actions based on plugin state (installed, not installed, update available)
- Search for osdag plugins from online repository, conda channel

Current implementation uses the `zehen-249` conda channel as the online repository for plugins. In future, this could be made configurable. Add multiple channels support.

Worker Abstraction and Threading Model

Package installation and removal are potentially slow operations involving disk access, network communication, and subprocess execution. To avoid blocking the Qt event loop, all such operations are executed in background threads.

A generic `Worker` class is introduced to encapsulate this behavior.

```
class Worker(QObject):
    finished = Signal(bool)

    def __init__(self, func, *args, **kwargs):
        self.func = func
        self.args = args
        self.kwargs = kwargs

    @Slot()
    def run(self):
        try:
            result = self.func(*self.args, **self.kwargs)
```

```
            self.finished.emit(bool(result))
        except Exception:
            self.finished.emit(False)
```

Listing 4.10: Generic worker abstraction for background tasks

This abstraction allows any plugin lifecycle operation (download, delete, update) to be executed asynchronously without duplicating threading logic. The dialog only needs to provide the target function and arguments.

PluginWidget for Online Plugins

The Plugin Store uses its own `PluginWidget` implementation, distinct from the local plugin widget, to reflect the different set of available actions. Each widget represents a plugin available in the online repository.

Unlike the local plugin widget, this version exposes signals for:

- Download
- Delete
- Update

```
class PluginWidget(QWidget):
    download = Signal(PluginMetaData)
    delete = Signal(PluginMetaData)
    update = Signal(PluginMetaData)
```

Listing 4.11: Signals exposed by Plugin Store plugin widget

The widget itself does not decide which buttons should be visible or enabled. That decision is delegated to the dialog based on the comparison between local and online plugin states.

Discovery of Online and Local Plugins

Upon initialization, the dialog retrieves:

- All plugins available in the configured online channel

- All locally installed plugins
- A derived list of plugins for which updates are available

```
self.plugins = self.plugin_manager.discover_online_plugins(channel
    ="zehen-249")
self.local_plugins = self.plugin_manager.discover_local_plugins()
self.updates_available = self._check_for_updates()
```

Listing 4.12: Initial discovery of plugins in the store dialog

This separation ensures that version comparison logic remains explicit and traceable.

Version Resolution Strategy

Online repositories may contain multiple versions of the same plugin. The dialog resolves this by selecting only the latest version of each plugin based on semantic version comparison.

```
latest = {}
for plugin in self.plugins:
    if plugin.name not in latest:
        latest[plugin.name] = plugin
    else:
        if version.parse(plugin.version) > version.parse(latest[
            plugin.name].version):
             latest[plugin.name] = plugin
```

Listing 4.13: Resolving latest plugin versions

This approach avoids cluttering the UI while still enabling update detection.

Dynamic UI State Resolution

For each plugin, the dialog determines its effective state by comparing online metadata with locally installed plugins:

- Not installed → Download enabled
- Installed, same version → Delete enabled

- Installed, older version → Update and Delete enabled

This logic is centralized in the plugin population phase, ensuring consistent UI behavior.

Asynchronous Download, Delete, and Update Operations

Each operation follows the same pattern:

1. Disable relevant UI controls
2. Update the widget label to reflect the ongoing operation
3. Start a background thread
4. Execute the operation through the Plugin Manager
5. Update UI based on success or failure

```
thread = QThread()
worker = Worker(self.plugin_manager.download_plugin, plugin)
worker.moveToThread(thread)

thread.started.connect(lambda: self.start_worker(worker))
worker.finished.connect(lambda success: self._on_download_finished
    (success, widget, plugin))
thread.start()
```

Listing 4.14: Starting a background download operation

The use of `QMetaObject.invokeMethod` ensures that the worker executes in the correct thread context.

Correct Thread Context and Event Loop Integration In Qt, merely moving a worker object to a separate `QThread` using `moveToThread()` does not automatically guarantee that its methods will execute in that thread. A method will only run in the target thread if it is invoked through Qt's event system.

To ensure this, the Plugin Store dialog uses `QMetaObject.invokeMethod` with a `Qt::QueuedConnection`:

```
QMetaObject.invokeMethod(worker, "run", Qt::QueuedConnection)
```

This mechanism schedules the `run()` method as an event in the target thread's event loop, rather than executing it immediately in the caller's thread. As a result, the method executes safely within the worker thread context.

This distinction is critical because directly calling `worker.run()` after moving the object to a thread would still execute the method in the main GUI thread, defeating the purpose of offloading long-running tasks and potentially freezing the user interface.

Why Queued Connections Are Necessary Qt distinguishes between direct and queued method invocations:

- **Direct invocation** executes the method immediately in the caller's thread.
- **Queued invocation** posts the call to the receiver's event loop, ensuring execution in the receiver's thread.

By explicitly requesting a queued connection, the plugin manager guarantees that:

- Network requests and subprocess calls do not block the GUI thread
- Thread affinity rules are respected
- Signal-slot delivery remains deterministic and safe

Design Rationale This approach provides a reusable and robust threading pattern:

- The worker remains agnostic of threading details
- The dialog controls execution context explicitly
- Additional background tasks can be added without duplicating thread logic

This design choice also simplifies future extensions, such as progress reporting, cancellation, or task prioritization, since all execution is routed through Qt's event-driven threading model.

Synchronization with the Plugin Manager Dialog

When the Plugin Store dialog is closed, it explicitly refreshes the Plugin Manager dialog to ensure that newly installed, updated, or removed plugins are immediately reflected.

```
def _refreshManager(self):
    self.plugin_manager_dialog = QApplication.instance().
        plugin_manager_dialog
    self.plugin_manager_dialog.refresh_plugins()
```

Listing 4.15: Refreshing the plugin manager after store operations

This loose coupling avoids direct dependencies while maintaining consistency across dialogs.

State Persistence and Cleanup

As with the Plugin Manager dialog, the store dialog flushes plugin state on closure to ensure consistency across application restarts.

```
def closeEvent(self, event):
    self.plugin_manager.state_manager.flush()
    self._refreshManager()
    super().closeEvent(event)
```

Listing 4.16: State persistence on dialog closure

This ensures that plugin lifecycle operations remain atomic and recoverable.

4.5 Source Code Reference

The complete implementation of the plugin management system is provided in Appendix C for reference and maintenance purposes.

Chapter 5

Internship Task 4: Development of Plugins for Osdag

The integration of a plugin system in Osdag enables the application to evolve beyond a monolithic design into a modular and extensible engineering platform. Instead of embedding all functionality directly into the core system, independent feature modules can now be developed, distributed, and maintained as plugins.

This chapter focuses on the design principles, structural requirements, and development workflow for creating Osdag-compatible plugins. It is intended as both a development strategy and a practical guide for contributors who wish to extend Osdag without modifying its core source code.

Each plugin is treated as a self-contained software component that integrates seamlessly with the existing `osdag_core` and `osdag_gui` packages through the Plugin Manager and Plugin Store. The plugin system supports:

- Dynamic discovery and loading of new functionality at runtime,
- Versioned distribution and updates using the Plugin Store,
- Isolation of experimental or domain-specific features from the core application.
- Clear separation of concerns, allowing developers to focus on specific engineering tasks without needing to understand the entire Osdag codebase.
- Facilitation of community contributions by lowering the barrier to entry for new developers.

To ensure consistency and reliability across independently developed plugins, a standard plugin skeleton is defined in this documentation. This skeleton specifies the mandatory metadata, entry-point structure, and integration contracts required by the Osdag Plugin Manager.

After this minimal structural compliance is satisfied, developers are free to design internal logic, user interfaces, and computational workflows according to their specific engineering or research requirements.

The remainder of this chapter provides a detailed, developer-oriented description of the plugin architecture, required interfaces, packaging strategy, and recommended extension patterns.

5.1 Concept of a Plugin in Osdag

In the context of Osdag, a *plugin* is not defined by its internal engineering logic, computational methods, or user interface complexity. Instead, a plugin is defined primarily by the constraints required for its integration into the Osdag application through the Plugin Manager System.

The responsibilities of the plugin framework are limited to:

- Discovering the plugin package,
- Validating its metadata and entry point,
- Managing its activation and deactivation state,
- Providing a mechanism to launch the plugin from the Osdag GUI.

Once a plugin is successfully discovered, installed through the Plugin Store, and correctly registered with the Plugin Manager, the internal implementation of the plugin is intentionally unconstrained. From this point onward, the plugin behaves as an independent extension module that may utilize `osdag_core`, `osdag_gui`, or any external libraries as required by the developer.

During this internship, the existing Osdag *Purlin* module was refactored and implemented as a plugin to validate this concept. By introducing only minimal structural modifications, the module was made compatible with the plugin system, demonstrating that:

- Activation and deactivation are fully managed by the Plugin Manager,
- The plugin can be launched directly from the Add-ons interface,

This experiment motivated the development of a more general plugin structure that is not tightly coupled to the current Osdag application design, but instead provides a stable and reusable contract for future extensions.

The following sections describe this generalized plugin structure and the development guidelines required to implement new Osdag plugins.

5.2 Plugin Structure and Development Guidelines - A Developers' Guide

To create a new plugin for Osdag, developers must adhere to a defined structure and set of guidelines. This section outlines the necessary components and steps to develop a compliant plugin.

5.2.1 Plugin Package Structure

Each Osdag plugin is distributed as an installable Python package. In the current implementation, plugins are packaged and distributed using the Conda ecosystem. A typical plugin follows the structure shown below:

```
demo_plugin/
|
|-- conda-recipe/
|   |-- meta.yaml
|
|-- demo_plugin/
|   |-- __init__.py
|   |-- demo_plugin.py
|
`-- pyproject.toml
```

Listing 5.1: Directory Structure of an Osdag Plugin

Directory Description

- `demo_plugin/` (root folder): Project root used for packaging and distribution.
- `conda-recipe/meta.yaml`: Defines the Conda build metadata, dependencies, versioning, and entry-point registration required by the Plugin Store.
- `demo_plugin/` (Python package): Contains the implementation of the plugin logic.
- `demo_plugin.py`: Implements the main plugin entry class and runtime logic.
- `pyproject.toml`: Specifies build configuration and Python packaging metadata.

5.2.2 Mandatory Plugin Interface and Registration

For a Python package to be recognized as a valid Osdag plugin, it must explicitly declare itself as such. This is achieved through a small set of mandatory variables and an entry-point specification that together form the integration contract with the Plugin Manager.

These definitions are intentionally minimal and do not restrict the internal architecture of the plugin.

Required Module-Level Declarations The root module of the plugin package (e.g., `__init__.py`) must define the following symbols:

```
# Indicates that this module is a valid Osdag plugin
IS_OSDAG_PLUGIN = True

# Metadata consumed by the Plugin Manager
META = {
    "id": "demo_plugin",
    "name": "Demo Plugin",
    "description": "A demonstration plugin for Osdag.",
    "authors": ["FOSSEE Team"],
    "version": "1.0.0"
}

# Entry class path used to launch the plugin
ENTRY_POINT = "demo_plugin.DemoPlugin"
```

Listing 5.2: Mandatory Plugin Declarations

Purpose of Each Declaration

- `IS_OSDAG_PLUGIN`: Acts as a validation flag during discovery. Modules not defining this symbol are ignored.
- `META`: Supplies human-readable and versioning information used by the Plugin Manager and Plugin Store.
- `ENTRY_POINT`: Specifies the class to be instantiated when the plugin is launched from the Osdag interface.

Entry Class Requirement The class referenced by `ENTRY_POINT` must be importable from the plugin module and implement the runtime behavior of the plugin. A minimal example is shown in Appendix D.

At runtime, the Plugin Manager dynamically resolves this class using the entry-point definition and transfers control to the plugin implementation.

This mechanism allows Osdag to remain agnostic to the internal structure of the plugin while guaranteeing a consistent activation and launch workflow.

Entry Point Resolution

Each plugin must expose a single entry class that is loaded dynamically by the Osdag Plugin Manager. The class is specified using the `ENTRY_POINT` variable inside the plugin module.

```
1 ENTRY_POINT = "demo_plugin.DemoPlugin"
```

Listing 5.3: Defining the plugin entry point

The value of `ENTRY_POINT` is not arbitrary. It represents the *import path of the class relative to the plugin package.*

Internally, the Plugin Manager resolves this path using Python's dynamic import mechanism:

- The string is split at dots.

- All segments except the last are treated as the module path.
- The last segment is treated as the class name.

For example:

Class Location	ENTRY_POINT Value
`demo_plugin/__init__.py`	`"DemoPlugin"`
`demo_plugin/demo_plugin.py`	`"demo_plugin.DemoPlugin"`
`demo_plugin/ui/main.py`	`"ui.main.DemoPlugin"`

If only the class name is provided (e.g., `"DemoPlugin"`), the Plugin Manager assumes that the class is defined directly inside `__init__.py`. In most practical plugins, the implementation resides in a separate module, therefore the fully qualified path must be used.

This design enforces an explicit and unambiguous contract between the plugin package structure and the Plugin Manager, ensuring reliable dynamic loading and future extensibility.

5.2.3 Package Configuration and Entry-Point Resolution

All Osdag plugins are distributed as standard Python packages. The primary contract between a plugin and the Osdag application is defined through the plugin's `pyproject.toml` file, which specifies how the package is built, identified, installed, and discovered at runtime.

A minimal, valid configuration is shown in Appendix D.

This file serves three independent but tightly coupled purposes:

- It defines the external distribution identity of the plugin.
- It specifies the importable Python package.
- It registers the plugin with the Osdag Plugin Manager via an entry-point contract.

Understanding how these layers interact is essential for reliable plugin discovery.

Naming Layers in an Osdag Plugin

An Osdag plugin is identified through three related naming levels.

1. Distribution Name (Project Name) Defined under:

```
[project]
name = "demo_plugin"
```

This is the name used by the package manager (Conda/Pip). It is the identifier under which the plugin is published, installed, and displayed in the Plugin Store.

```
conda install demo_plugin
```

2. Python Package Name Defined by the source layout and build configuration:

```
[tool.setuptools]
packages = ["demo_plugin"]
```

And corresponding directory:

```
demo_plugin/
|-- __init__.py
|-- demo_plugin.py
```

This name controls how the plugin is imported at runtime:

```
import demo_plugin
```

3. Osdag Plugin Entry Point Defined under:

```
[project.entry-points."osdag.plugins"]
demo_plugin = "demo_plugin"
```

The left-hand identifier is the logical plugin name inside Osdag. The right-hand value is the module path that Osdag dynamically imports during discovery.

Internally, the Plugin Manager executes:

```
import demo_plugin
```

Recommended Convention

For deterministic behavior and maintainability, all three names should remain identical:

```toml
[project]
name = "demo_plugin"

[tool.setuptools]
packages = ["demo_plugin"]

[project.entry-points."osdag.plugins"]
demo_plugin = "demo_plugin"
```

Implications of Incorrect Naming

- A mismatch between the distribution name and the Python package name may result in successful installation but failed imports.
- An incorrect entry-point import path prevents the Plugin Manager from discovering the plugin.
- Inconsistent naming complicates debugging, maintenance, and Plugin Store management.

In summary, the package manager installs the distribution, Python imports the package, and Osdag discovers the plugin through the entry-point registry. Correct alignment of these layers is mandatory for successful integration.

5.2.4 Conda Packaging and Distribution Metadata

Osdag plugins are distributed through a Conda channel and installed dynamically by the Plugin Store. For this reason, each plugin must provide a Conda build recipe that describes how the package is built, what it depends on, and how it integrates with Osdag.

A minimal `meta.yaml` recipe is shown appendix D.

Key Sections and Their Role **Package Definition**

The `name` and `version` define the identity of the plugin in the Conda ecosystem. This name is the same identifier used by the Plugin Store during installation and updates.

Source

The `path: ..` directive instructs Conda to build the package from the local project root, enabling reproducible packaging directly from the plugin source tree. This path is relative to the location of `meta.yaml`. and must point to the root of the plugin project (e.g., `demo_plugin/`).

Build Section

- `noarch: python` indicates that the plugin is architecture-independent.
- The build script uses standard PEP 517 packaging: `pip install .` ensures that the Python metadata and entry-points defined in `pyproject.toml` are correctly registered.

Requirements

Two dependency layers are defined:

- `host`: tools required to build the plugin.
- `run`: runtime dependencies required after installation.

The dependency on `osdag::osdag` ensures that the plugin is always installed against a compatible Osdag version and can safely access its public APIs.

About Section

This section provides descriptive metadata used by the Plugin Store UI, documentation, and auditing tools.

Extra Metadata

`recipe-maintainers` records ownership and long-term maintenance responsibility.

Integration with the Plugin Store When a user installs a plugin from the Osdag Plugin Store, the Plugin Manager internally executes a Conda installation using this recipe. Successful installation automatically registers the plugin's entry point and makes it discoverable by the Osdag runtime without requiring any manual configuration.

5.2.5 Implementing the Plugin Entry Class

Each Osdag plugin must expose a single entry class that is discoverable by the Plugin Manager. This class must inherit from one of the base plugin interfaces provided by Osdag:

- `WidgetPlugin` - used for plugins that integrate as dockable tabs within the Osdag GUI.
- `WindowPlugin` - used for plugins that launch as independent Qt windows from the Osdag interface.

See Appendix D.

5.2.6 Plugin Base Classes and Runtime Contract

Osdag defines a minimal inheritance hierarchy that every plugin must follow. This hierarchy establishes the lifecycle, embedding strategy, and UI integration model for all external extensions.

PluginBase `PluginBase` is the root abstraction for all plugins:

```
class PluginBase:
    """Base class for all Osdag plugins."""
    def __init__(self, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)

    def setupUI(self) -> None:
        """Setup UI for the Plugin"""
        pass
```

Listing 5.4: PluginBase Definition

This class defines the minimal interface expected by the Plugin Manager. All concrete plugins must implement the `setupUI()` method, which acts as the primary initialization hook.

WidgetPlugin `WidgetPlugin` is intended for plugins that integrate as dockable components inside the Osdag main window:

```
class WidgetPlugin(PluginBase, QWidget):
    """Base class for widget-based plugins."""

```

```
4     def __init__(self, parent: QWidget = None, title: str = "
          Default Title") -> None:
5         super().__init__(parent)
6         self.title = title
7
8     def setupUI(self) -> None:
9         self.layout = QVBoxLayout(self)
10        label = QLabel(f"Default UI for {self.title}")
11        self.layout.addWidget(label)
```

Listing 5.5: WidgetPlugin Definition

Inheritance from `QWidget` allows the plugin to be embedded directly into the Osdag workspace. The framework manages the container lifecycle, while the developer controls only the internal layout.

WindowPlugin `WindowPlugin` is intended for standalone tools launched from Osdag:

```
1 class WindowPlugin(PluginBase, QMainWindow):
2     """Plugin that provides a Window."""
3
4     def setupUI(self) -> None:
5         super().setupUI()
6         self.window = QMainWindow()
```

Listing 5.6: WindowPlugin Definition

Inheritance from `QMainWindow` enables the plugin to behave as an independent application window while remaining discoverable and launchable through Osdag.

Design Implications The inheritance relationship defines how the plugin is embedded into the application and how its lifecycle is managed.

In this example, `DemoPlugin` extends `WidgetPlugin` (see Appendix D), meaning:

- The plugin is loaded as a GUI component inside the main Osdag workspace.
- The framework automatically creates and manages the plugin container widget.
- The developer is only responsible for implementing the internal UI and logic.

The `setupUI()` method serves as the construction contract between the plugin and the Osdag runtime. Once this method returns, the plugin is considered fully initialized and ready for user interaction.

Optional initialization logic (for example, configuration validation or informational dialogs) may be implemented in the constructor, as demonstrated by `show_default_ui_message()`.

5.3 Minimal Working Plugin Template

This section presents the smallest functional Osdag plugin that satisfies all structural and runtime requirements of the Plugin Manager. Developers can use this template as a starting point for new plugin development.

5.3.1 Project Structure

```
sample_plugin/
|
|-- conda-recipe/
|   |-- meta.yaml
|
|-- sample_plugin/
|   |-- __init__.py
|   |-- sample_plugin.py
|
`-- pyproject.toml
```

Listing 5.7: Minimal Plugin Layout

5.3.2 Plugin Entry Implementation

```python
# sample_plugin/sample_plugin.py

from osdag_gui.plugins.widget_plugin import WidgetPlugin
from PySide6.QtWidgets import QLabel

class SamplePlugin(WidgetPlugin):
```

```
7
8       def __init__(self, parent=None, title="Sample Plugin"):
9           super().__init__(parent, title)
10
11      def setupUI(self):
12          super().setupUI()
13          self.layout.addWidget(QLabel("Hello from Sample Plugin!"))
```

Listing 5.8: sample_plugin/sample_plugin.py

```
1  # sample_plugin/__init__.py
2
3  IS_OSDAG_PLUGIN = True
4
5  META = {
6      "id": "sample_plugin",
7      "name": "sample_plugin",
8      "version": "1.0.0",
9      "authors": ["FOSSEE Team"],
10     "description": "Minimal Osdag Plugin Example"
11 }
12
13 ENTRY_POINT = "sample_plugin.SamplePlugin"
```

Listing 5.9: sample_plugin/__init__.py

5.3.3 Packaging Configuration

```
1  # pyproject.toml
2
3  [build-system]
4  requires = ["setuptools>=61.0"]
5  build-backend = "setuptools.build_meta"
6
7  [project]
8  name = "sample_plugin"
```

```
version = "1.0.0"
description = "Minimal Osdag Plugin Example"
requires-python = ">=3.10"

[tool.setuptools]
packages = ["sample_plugin"]

[project.entry-points."osdag.plugins"]
sample_plugin = "sample_plugin"
```

Listing 5.10: pyproject.toml

5.3.4 Behavior at Runtime

Once installed and activated:

- The Plugin Manager discovers the package via the `osdag.plugins` entry point.
- The `SamplePlugin` class is instantiated.
- `setupUI()` is executed.
- The widget is embedded into the Osdag workspace under the Add-ons section.

This template guarantees compatibility with the Osdag plugin framework while leaving complete freedom for internal implementation.

Chapter A

CLI Module - Code Files

A.1 Example OSI File

```
Conn_Location: Long Leg
Connector.Material: E 165 (Fe 290)
Connector.Plate.Thickness_List:
- '8'
- '10'
- '12'
- '14'
- '16'
- '18'
- '20'
- '22'
- '25'
- '28'
- '32'
- '36'
- '40'
- '45'
- '50'
- '56'
- '63'
- '75'
```

```
- '80'
- '90'
- '100'
- '110'
- '120'
Design.Design_Method: Limit State Design
Load.Axial: '7'
Material: E 165 (Fe 290)
Member.Designation:
- 20 x 20 x 3
- 20 x 20 x 4
- 25 x 25 x 3
- 25 x 25 x 4
- 25 x 25 x 5
- 30 x 30 x 3
- 30 x 30 x 4
- 30 x 30 x 5
- 35 x 35 x 3
- 35 x 35 x 4
- 35 x 35 x 5
- 35 x 35 x 6
- 40 x 40 x 3
- 40 x 40 x 4
- 40 x 40 x 5
- 40 x 40 x 6
- 45 x 45 x 3
- 45 x 45 x 4
- 45 x 45 x 5
- 45 x 45 x 6
- 50 x 50 x 3
- 50 x 50 x 4
- 50 x 50 x 5
- 50 x 50 x 6
- 55 x 55 x 4
```

```
- 55 x 55 x 5
- 55 x 55 x 6
- 55 x 55 x 8
- 60 x 60 x 4
- 60 x 60 x 5
- 60 x 60 x 6
- 60 x 60 x 8
- 65 x 65 x 4
- 65 x 65 x 5
- 65 x 65 x 6
- 65 x 65 x 8
- 70 x 70 x 5
- 70 x 70 x 6
- 70 x 70 x 8
- 70 x 70 x 10
- 75 x 75 x 5
- 75 x 75 x 6
- 75 x 75 x 8
- 75 x 75 x 10
- 80 x 80 x 6
- 80 x 80 x 8
- 80 x 80 x 10
- 80 x 80 x 12
- 90 x 90 x 6
- 90 x 90 x 8
- 90 x 90 x 10
- 90 x 90 x 12
- 100 x 100 x 6
- 100 x 100 x 8
- 100 x 100 x 10
- 100 x 100 x 12
- 110 x 110 x 8
- 110 x 110 x 10
- 110 x 110 x 12
```

```
- 110 x 110 x 16
- 130 x 130 x 8
- 130 x130 x 10
- 130 x130 x 12
- 130 x130 x 16
- 150 x 150 x 10
- 150 x 150 x 12
- 150 x 150 x 16
- 150 x 150 x 20
- 200 x 200 x 12
- 200 x 200 x 16
- 200 x 200 x 20
- 200 x 200 x 25
- 50 x 50 x 7
- 50 x 50 x 8
- 55 x 55 x 10
- 60 x 60 x 10
- 65 x 65 x 10
- 70 x 70 x 7
- 100 x 100 x 7
- 100 x 100 x 15
- 120 x 120 x 8
- 120 x 120 x 10
- 120 x 120 x 12
- 120 x 120 x 15
- 130 x 130 x 9
- 150 x 150 x 15
- 150 x 150 x 18
- 180 x 180 x 15
- 180 x 180 x 18
- 180 x 180 x 20
- 200 x 200 x 24
- 30 x 20 x 3
- 30 x 20 x 4
```

```
- 30 x 20 x 5
- 40 x 25 x 3
- 40 x 25 x 4
- 40 x 25 x 5
- 40 x 25 x 6
- 45 x 30 x 3
- 45 x 30 x 4
- 45 x 30 x 5
- 45 x 30 x 6
- 50 x 30 x 3
- 50 x 30 x 4
- 50 x 30 x 5
- 50 x 30 x 6
- 60 x 40 x 5
- 60 x 40 x 6
- 60 x 40 x 8
- 65 x 45 x 5
- 65 x 45 x 6
- 65 x 45 x 8
- 70 x 45 x 5
- 70 x 45 x 6
- 70 x 45 x 8
- 70 x 45 x 10
- 75 x 50 x 5
- 75 x 50 x 6
- 75 x 50 x 8
- 75 x 50 x 10
- 80 x 50 x 5
- 80 x 50 x 6
- 80 x 50 x 8
- 80 x 50 x 10
- 90 x 60 x 6
- 90 x 60 x 8
- 90 x 60 x 10
```

```
- 90 x 60 x 12
- 100 x 65 x 6
- 100 x 65 x 8
- 100 x 65 x 10
- 100 x 75 x 6
- 100 x 75 x 8
- 100 x 75 x 10
- 100 x 75 x 12
- 125 x 75 x 6
- 125 x 75 x 8
- 125 x 75 x 10
- 125 x 95 x 6
- 125 x 95 x 8
- 125 x 95 x 10
- 125 x 95 x 12
- 150 x 115 x 8
- 150 x 115 x 10
- 150 x 115 x 12
- 150 x 115 x 16
- 200 x 100 x 10
- 200 x 100 x 12
- 200 x 100 x 16
- 200 x 150 x 10
- 200 x 150 x 12
- 200 x 150 x 16
- 200 x 150 x 20
- 40 x 20 x 3
- 40 x 20 x 4
- 40 x 20 x 5
- 60 x 30 x 5
- 60 x 30 x 6
- 60 x 40 x 7
- 65 x 50 x 5
- 65 x 50 x 6
```

```
- 65 x 50 x 7
- 65 x 50 x 8
- 70 x 50 x 5
- 70 x 50 x 6
- 70 x 50 x 7
- 70 x 50 x 8
- 75 x 50 x 7
- 80 x 40 x 5
- 80 x 40 x 6
- 80 x 40 x 7
- 80 x 40 x 8
- 80 x 60 x 6
- 80 x 60 x 7
- 80 x 60 x 8
- 90 x 65 x 6
- 90 x 65 x 7
- 90 x 65 x 8
- 90 x 65 x 10
- 100 x 50 x 6
- 100 x 50 x 7
- 100 x 50 x 8
- 100 x 50 x 10
- 100 x 65 x 7
- 120 x 80 x 8
- 120 x 80 x 10
- 120 x 80 x 12
- 125 x 75 x 12
- 135 x 65 x 8
- 135 x 65 x 10
- 135 x 65 x 12
- 150 x 75 x 9
- 150 x 75 x 15
- 150 x 90 x 10
- 150 x 90 x 12
```

```
- 150 x 90 x 15
- 200 x 100 x 15
- 200 x 150 x 15
- 200 x 150 x 18
Member.Length: '1250'
Member.Material: E 165 (Fe 290)
Member.Profile: Angles
Module: Tension Member Design - Welded to End Gusset
Weld.Fab: Shop Weld
Weld.Material_Grade_OverWrite: '290'
out_titles_status:
- 1
- 1
- 1
- 1
- 0
- 0
- 0
- 0
```

Listing A.1: Example OSI file for Tension Member Design - Welded to End Gusset

A.2 CLI Module

```
1
2 from osdag_core.design_type.connection.fin_plate_connection import
      FinPlateConnection
3 from osdag_core.design_type.connection.cleat_angle_connection
     import CleatAngleConnection
4 from osdag_core.design_type.connection.seated_angle_connection
     import SeatedAngleConnection
5 from osdag_core.design_type.connection.end_plate_connection import
      EndPlateConnection
```

```
from osdag_core.design_type.connection.base_plate_connection
    import BasePlateConnection
from osdag_core.design_type.connection.beam_cover_plate import
    BeamCoverPlate
from osdag_core.design_type.connection.beam_cover_plate_weld
    import BeamCoverPlateWeld
from osdag_core.design_type.connection.column_cover_plate_weld
    import ColumnCoverPlateWeld
from osdag_core.design_type.tension_member.tension_bolted import
    Tension_bolted
from osdag_core.design_type.tension_member.tension_welded import
    Tension_welded
from osdag_core.design_type.connection.beam_beam_end_plate_splice
    import BeamBeamEndPlateSplice
from osdag_core.design_type.connection.beam_column_end_plate
    import BeamColumnEndPlate
from osdag_core.design_type.connection.column_cover_plate import
    ColumnCoverPlate
from osdag_core.design_type.connection.column_end_plate import
    ColumnEndPlate
from osdag_core.design_type.compression_member.compression import
    Compression
from osdag_core.design_type.main import Main
from osdag_core.Common import TYPE_TEXTBOX, TYPE_OUT_BUTTON
from osdag_core.Common import (
    # Shear Connection
    KEY_DISP_FINPLATE,
    KEY_DISP_ENDPLATE,
    KEY_DISP_CLEATANGLE,
    KEY_DISP_SEATED_ANGLE,

    # Base Plate Connection
    KEY_DISP_BASE_PLATE,

```

```
    # Moment Connection
    KEY_DISP_BEAMCOVERPLATE,
    KEY_DISP_COLUMNCOVERPLATE,
    KEY_DISP_BEAMCOVERPLATEWELD,
    KEY_DISP_COLUMNCOVERPLATEWELD,
    KEY_DISP_BB_EP_SPLICE,
    KEY_DISP_COLUMNENDPLATE,
    KEY_DISP_BCENDPLATE,

    # Tension Member
    KEY_DISP_TENSION_BOLTED,
    KEY_DISP_TENSION_WELDED,

    # Compression Member
    KEY_DISP_COMPRESSION

)


available_modules = {
    KEY_DISP_BASE_PLATE:BasePlateConnection,
    KEY_DISP_BEAMCOVERPLATE:BeamCoverPlate,
    KEY_DISP_CLEATANGLE:CleatAngleConnection,
    KEY_DISP_COLUMNCOVERPLATE:ColumnCoverPlate,
    KEY_DISP_COLUMNENDPLATE:ColumnEndPlate,
    KEY_DISP_ENDPLATE:EndPlateConnection,
    KEY_DISP_FINPLATE:FinPlateConnection,
    KEY_DISP_SEATED_ANGLE:SeatedAngleConnection,
    KEY_DISP_TENSION_BOLTED:Tension_bolted,
    KEY_DISP_TENSION_WELDED:Tension_welded,
    KEY_DISP_COMPRESSION:Compression,
    KEY_DISP_BEAMCOVERPLATEWELD:BeamCoverPlateWeld,
    KEY_DISP_COLUMNCOVERPLATEWELD:ColumnCoverPlateWeld,
    KEY_DISP_BB_EP_SPLICE:BeamBeamEndPlateSplice,
```

```
    KEY_DISP_BCENDPLATE:BeamColumnEndPlate,
}

from pathlib import Path
import yaml, click
import pandas as pd

def _print_result(out_dict:dict):
    print("="*100)
    for key, value in out_dict.items():
        print(f"|| {key}: {value}")
    print("="*100)

def _get_design_dictionary(osi_path:Path) -> dict:
    """return the design dictionary from an OSI file."""
    with open(osi_path, 'r') as file:
        return yaml.safe_load(file)

def _get_output_dictionary(module_class:Main) -> dict:
    """return the output dictionary for the design"""
    status = module_class.design_status
    out_list = module_class.output_values(module_class, status)
    out_dict = {"Parameter": "Value"}
    for option in out_list:
        if option[0] is not None and option[2] == TYPE_TEXTBOX:
            out_dict[option[0]] = option[3]
        if option[2] == TYPE_OUT_BUTTON:
            tup = option[3]
            fn = tup[1]
            for item in fn(module_class, status):
                lable = item[0]
                value = item[3]
                if lable!=None and value!=None:
                    out_dict[lable] = value
```

```
97      return out_dict
98
99
100 def _save_to_csv(output_dictionary:dict, output_file:str):
101     """save the output dictionary to a csv file"""
102     df = pd.DataFrame(output_dictionary.items())
103     df.to_csv(output_file, index=False, header=None)
104
105 def _save_to_pdf(module_class:Main, output_file:Path):
106     """save the output dictionary to a pdf file"""
107     popup_summary = {
108             'ProfileSummary': {
109             'CompanyName': 'LoremIpsum',
110             'CompanyLogo': '',
111             'Group/TeamName': 'LoremIpsum',
112             'Designer': 'LoremIpsum'
113         },
114         'ProjectTitle': 'Fossee',
115         'Subtitle': '',
116         'JobNumber': '123',
117         'AdditionalComments': 'No comments',
118         'Client': 'LoremIpsum',
119         'filename': f'{output_file}',
120         'does_design_exist': True,
121         'logger_messages': ''
122         }
123     module_class.save_design(module_class, popup_summary)
124
125
126
127 def run_module(*args, **kargs) -> dict:
128     """Run the module specified in the OSI file located at
        osi_path."""
129     osi_path = kargs["input_path"] if len(kargs) > 0 else None
```

```
130     op_type = kargs["op_type"] if len(kargs) > 1 else "
            print_result"
131     output_path = kargs["output_path"] if len(kargs) > 2 else None
132
133     result = {
134         "success": False,
135         "operation": op_type,
136         "input": str(osi_path) if osi_path else None,
137         "output": None,
138         "data": None,
139         "errors": [],
140     }
141
142     if osi_path is None:
143         result["errors"].append("No input file provided.")
144         print(result)
145         return result
146
147     osi_path = Path(osi_path) if osi_path else None
148     output_path = Path(output_path) if output_path else None
149     if not osi_path.exists():
150         result["errors"].append(f"File not found: {osi_path}")
151         print(result)
152         return result
153
154     design_dict = _get_design_dictionary(osi_path)
155     module_name = design_dict.get("Module")
156     if not module_name:
157         result["errors"].append("Module not specified.")
158         print(result)
159         return result
160
161     module_class = available_modules.get(module_name)
162     if not module_class:
```

```
163         result["errors"].append(f"Not a valid module class: {
                module_name}")
164         print(result)
165         return result
166 
167     input_filename = osi_path.stem
168     output_filename = output_path.stem if output_path else None
169     if not output_path:
170         output_folder_path = osi_path.parent / "Outputs" / f"{
                module_class.__name__}"
171     else:
172         output_folder_path = output_path.parent / f"{module_class.
                __name__}"
173     output_folder_path.mkdir(parents=True, exist_ok=True)
174     output_file = output_folder_path / f"{output_filename if
            output_filename else input_filename}"
175 
176     module_class.set_osdaglogger(None)
177     val_errors = module_class.func_for_validation(module_class,
            design_dict)
178 
179     if val_errors:
180         result["errors"].extend(val_errors)
181         print(result)
182         return result
183 
184     out_dict = _get_output_dictionary(module_class)
185     result["data"] = out_dict
186     if op_type == "save_csv":
187         try:
188             _save_to_csv(out_dict, str(output_file) + ".csv")
189             result["success"] = True
190             result["output"] = str(output_file)
191         except Exception as e:
```

```
            result["success"] = False
            result["errors"].append(f"Failed to save CSV: {e}")

    elif op_type == "save_pdf":
        try:
            _save_to_pdf(module_class, output_file)
            result["success"] = True
            result["output"] = str(output_file)
        except Exception as e:
            result["success"] = False
            result["errors"].append(f"Failed to save PDF: {e}")

    elif op_type == "print_result":
        try:
            result["success"] = True
            click.echo(_print_result(out_dict=out_dict))
        except Exception as e:
            result["success"] = False
            result["errors"].append(f"Failed to get result: {e}")

    else:
        result["errors"].append(f"Unsupported op_type: {op_type}")

    if len(result["errors"]) > 0:
        print(result)

    return result
```

Listing A.2: File: `osdag_core/cli.py`

A.3 CLI Runner Script

```

```

```
% def GUI():
%     app = QApplication(sys.argv)
%     fid = QFontDatabase.addApplicationFont(":/fonts/UbuntuSans-Regular.ttf")
%     font = QFontDatabase.applicationFontFamilies(fid)[0]
%     # app.setFont(QFont(font))

%     # ====== Plugin Manager ======
%     print("\n[INFO] Initializing Plugin Management System...")
%     from osdag_core.utils.plugin_manager import PluginManager
%     from osdag_gui.ui.components.dialogs.plugin_manager_dialog import PluginManagerDialog
%     if not hasattr(app, "plugin_manager"):
%         app.plugin_manager = PluginManager()
%     if not hasattr(app, "plugin_manager_dialog"):
%         app.plugin_manager_dialog = PluginManagerDialog()
%     print("[INFO] Plugin Management System initialized.\n")

%     # ============================

%     file = QFile(":/themes/lightstyle.qss")
%     if file.open(QFile.ReadOnly | QFile.Text):
%         stream = QTextStream(file)
%         stylesheet = stream.readAll()
%         file.close()
%         app.setStyleSheet(stylesheet)

%     def show_main_window():
%         from osdag_gui.main_window import MainWindow
%         app.main_window = MainWindow()
%         app.main_window.show()

%     splash = LaunchScreenPopup(on_finish=show_main_window)
%     splash.show()
```

```
%      sys.exit(app.exec())
%
%
% # --- Main CLI group ---
% help_msg = """\n\b
% ==================================================
% Osdag Steel Design and Graphics Application
%
% Usage:\n
%   osdag                          # Launch GUI (default)\n
%   osdag cli run                  # Use CLI tools (see below)
%
% By default, running 'osdag' launches the GUI.
% You can also run in CLI mode using 'osdag cli run'.
%
% Examples:\n
%   osdag\n
%   osdag cli run -i TensionBolted.osi\n
%   osdag cli run -i TensionBolted.osi -op save_csv -o result.csv\
   n
%   osdag cli run -i TensionBolted.osi -op save_pdf -o result.pdf\
   n
%   osdag cli run -i TensionBolted.osi -op print_result\n
% ==================================================\n
% """
%
% @click.group(invoke_without_command=True,
%              help="\nOsdag Application. Run osdag to launch GUI,
   or use 'osdag cli run' for command-line tools.\n",
%              epilog=help_msg,
%              context_settings=dict(help_option_names=['-h', '--
   help']),
%              )
%
```

```
% @click.pass_context
% def main(ctx):
%     if ctx.invoked_subcommand is None:
%         GUI()
%
%
% # --- CLI group ---
% @main.group(help="\nRun in CLI mode (use subcommands like 'run').\n",
%             epilog=help_msg,
%             context_settings=dict(help_option_names=['-h', '--help']),
%             )
% def cli():
%     pass
%
%
% # --- Subcommand: run ---
% @cli.command(help="\nOsdag Application. Run osdag to launch GUI, or use 'osdag cli run' for command-line tools.\n",
%             epilog=help_msg,
%             context_settings=dict(help_option_names=['-h', '--help']),
%             )
% @click.option("-i", "--input", "input_path",
%               type=click.Path(exists=True, dir_okay=False),
%               required=True,
%               help="Path to input file (.osi)")
% @click.option("-op", "--op_type",
%               type=click.Choice(["save_csv", "save_pdf", "print_result"]),
%               default="print_result",
%               show_default=True,
%               help="Type of operation")
```

```
% @click.option("-o", "--output", "output_path",
%               type=click.Path(dir_okay=False, writable=True),
%               help="Path for output file")
% def run(input_path, op_type, output_path):
%     result = run_module(input_path=input_path,
%                         op_type=op_type,
%                         output_path=output_path)
%
%     if not result["success"]:
%         click.echo("Errors encountered:")
%         for err in result["errors"]:
%             click.echo(f"   - {err}")
%     else:
%         click.echo("Operation completed successfully")
%         if result.get("output"):
%             click.echo(f"Output saved at: {result['output']}")
%
%
% if __name__ == "__main__":
%     main()
```

Listing A.3: File: osdag_gui/__main__.py

Chapter B

In-app Update feature - Code File

```
from pathlib import Path
import sys, shutil, json
from packaging.version import Version

from PySide6.QtWidgets import (
    QApplication,
    QDialog,
    QVBoxLayout,
    QPushButton,
    QHBoxLayout,
    QWidget,
    QTextBrowser,
    QSizePolicy,
    QProgressBar
)
from PySide6.QtCore import QProcess, Qt
from PySide6.QtSvgWidgets import QSvgWidget
from PySide6.QtGui import QIcon

from osdag_gui.__config__ import INSTALLATION_TYPE, VERSION
from osdag_gui.ui.components.dialogs.custom_titlebar import
   CustomTitleBar

```

```
23 class UpdateDialog(QDialog):
24     def __init__(self):
25         super().__init__()
26         self.old_version = Version(VERSION)
27         self.setWindowFlags(Qt.FramelessWindowHint)
28         self.setAttribute(Qt.WA_StyledBackground, True)
29         self.setObjectName("UpdateDialog")
30         self.setWindowIcon(QIcon(":/images/osdag_logo.png"))
31         self.setFixedSize(580, 450)
32 
33         # Layout and style
34         self.setStyleSheet("""
35             QDialog#UpdateDialog { background-color: #ffffff;
                   border: 1px solid #90AF13; }
36             QWidget#ContentWidget { background-color: #ffffff; }
37             QTextBrowser { background-color: #ffffff; border: 1px
                   solid #e0e0e0; border-radius: 4px;
38                            font-family: 'Arial'; font-size: 8pt;
                               padding: 8px; }
39             QProgressBar { border: 1px solid #ccc; border-radius:
                   5px; text-align: center; height: 20px; }
40             QProgressBar::chunk { background-color: #90AF13; }
41             QPushButton { background-color: #90AF13; color: white;
                    border: none; border-radius: 5px;
42                           padding: 5px 20px; font-size: 12px; font
                               -weight: bold; }
43             QPushButton:hover { background-color: #7A9611; }
44             QPushButton:pressed { background-color: #6B850F; }
45         """)
46 
47         mainLayout = QVBoxLayout(self)
48         mainLayout.setContentsMargins(1, 1, 1, 1)
49         mainLayout.setSpacing(0)
50 
```

```
51          # Title bar
52          self.titleBar = CustomTitleBar()
53          self.titleBar.setTitle("Check for Updates")
54          mainLayout.addWidget(self.titleBar)
55
56          # Content
57          contentWidget = QWidget(self)
58          contentWidget.setObjectName("ContentWidget")
59          contentLayout = QVBoxLayout(contentWidget)
60          contentLayout.setContentsMargins(10, 10, 10, 10)
61          contentLayout.setSpacing(10)
62
63          self.logoLabel = QSvgWidget(":/vectors/Osdag.svg", self)
64          self.logoLabel.setFixedHeight(106)
65          self.logoLabel.setSizePolicy(QSizePolicy.Expanding,
               QSizePolicy.Fixed)
66          contentLayout.addWidget(self.logoLabel, 0, Qt.AlignCenter)
67
68          self.textBrowser = QTextBrowser(self)
69          self.textBrowser.setOpenExternalLinks(True)
70          self.textBrowser.setSizePolicy(QSizePolicy.Expanding,
               QSizePolicy.Expanding)
71          self.textBrowser.setStyleSheet(
72              "font-size: 12pt;color:green"
73              )
74          contentLayout.addWidget(self.textBrowser)
75
76          self.progressBar = QProgressBar(self)
77          self.progressBar.setRange(0, 0)
78          contentLayout.addWidget(self.progressBar)
79
80          buttonLayout = QHBoxLayout()
81          buttonLayout.addStretch()
82          self.okButton = QPushButton("OK", self)
```

```
83          self.okButton.setFixedHeight(30)
84          self.okButton.setStyleSheet("""
85              QPushButton { background-color: #90AF13; color: white;
                    border: none; border-radius: 5px;
86                              padding: 5px 20px; font-size: 12px;
                                font-weight: bold; }
87              QPushButton:hover { background-color: #7A9611; }
88              QPushButton:pressed { background-color: #6B850F; }
89          """)
90          self.okButton.clicked.connect(self.accept)
91          buttonLayout.addWidget(self.okButton)
92
93          # Update buttons (hidden initially)
94          self.updateNowButton = QPushButton("Update Now", self)
95          self.updateLaterButton = QPushButton("Update Later", self)
96          self.updateNowButton.setFixedHeight(30)
97          self.updateLaterButton.setFixedHeight(30)
98          self.updateNowButton.clicked.connect(self.update_to_latest
                )
99          self.updateLaterButton.clicked.connect(self.close)
100         self.updateNowButton.hide()
101         self.updateLaterButton.hide()
102         buttonLayout.addWidget(self.updateNowButton)
103         buttonLayout.addWidget(self.updateLaterButton)
104
105         contentLayout.addLayout(buttonLayout)
106         mainLayout.addWidget(contentWidget)
107
108
109         self.textBrowser.setHtml("<p>Checking for updates...</p>")
110         self._set_exec_paths()
111         self.check_for_updates()
112
113     def _set_exec_paths(self):
```

```
114         env_path = Path(sys.prefix)
115         base_conda_path = env_path.parents[1]
116         if sys.platform.startswith("win"):
117             self.conda_path = base_conda_path / "Scripts" / "conda
                .exe"
118             self.pixi_path = env_path / "Scripts" / "pixi.exe"
119         else:
120             self.conda_path = base_conda_path / "bin/conda"
121             self.pixi_path = env_path / "bin/pixi"
122 
123         self.conda_path = str(self.conda_path if self.conda_path.
            exists() else shutil.which("conda"))
124         self.pixi_path = str(self.pixi_path if self.pixi_path.
            exists() else shutil.which("pixi"))
125 
126     def check_for_updates(self):
127         """Run QProcess to fetch version asynchronously."""
128         if not (self.conda_path or self.pixi_path):
129             self.update_text("<p style='color:red;'>Executable Not
                Found.</p>")
130             return
131 
132         try:
133             if INSTALLATION_TYPE == "conda":
134                 cmd = [self.conda_path, "search", "-c", "conda-
                    forge", "osdag::osdag", "--info", "--json"]
135             elif INSTALLATION_TYPE == "pixi":
136                 cmd = [self.pixi_path, "search", "osdag", "--
                    channel", "osdag"]
137             else:
138                 self.update_text("<p style='color:red;'>Unknown
                    installation type.</p>")
139                 return
140         except Exception as e:
```

```
141             self.update_text(f"<p style='color:red;'>Error: {e}</p
                    >")
142             return
143 
144         self.process = QProcess(self)
145         self.process.readyReadStandardOutput.connect(self.
                handle_stdout_update_check)
146         self.process.readyReadStandardError.connect(self.
                handle_stderr_update_check)
147         self.process.finished.connect(self.
                handle_update_check_finished)
148         self.process.start(cmd[0], cmd[1:])
149 
150     def handle_stdout_update_check(self):
151         self.output_data = self.process.readAllStandardOutput().
                data().decode()
152 
153     def handle_stderr_update_check(self):
154         self.error_data = self.process.readAllStandardError().data
                ().decode()
155 
156     def handle_update_check_finished(self):
157         self.progressBar.hide()
158         latest_version = None
159 
160         try:
161             if INSTALLATION_TYPE == "conda":
162                 data = json.loads(self.output_data)
163                 versions = data.get("osdag", [])
164                 if versions:
165                     latest_version = sorted(versions, key=lambda x
                            : x["version"])[-1]["version"]
166             elif INSTALLATION_TYPE == "pixi":
167                 for line in self.output_data.splitlines():
```

```
168                     if line.strip().startswith("Version"):
169                         latest_version = line.split()[-1].strip()
170
171             if latest_version:
172                 lv = Version(latest_version)
173                 if lv > self.old_version:
174                     self.update_text(
175                         f"<p style='color:green;'>A new version of
                                Osdag is available: "
176                         f"<b>{latest_version}</b> (You have {
                            VERSION}).<br>"
177                         f"Visit <a href='https://osdag.fossee.in/
                            resources/downloads'>downloads</a>.</p>
                            "
178                     )
179                     self.okButton.hide()
180                     self.updateNowButton.show()
181                     self.updateLaterButton.show()
182                 else:
183                     self.update_text(
184                         f"<p style='color:#90AF13;'>You are using
                            the latest version of Osdag "
185                         f"(<b>{VERSION}</b>).</p>"
186                     )
187             else:
188                 self.update_text("<p style='color:orange;'>Could
                    not fetch version information.</p>")
189
190         except Exception as e:
191             self.update_text(f"<p style='color:red;'>Error
                checking for updates: {e}</p>")
192
193
194     def update_to_latest(self):
```

```
195         """Run QProcess to update Osdag asynchronously."""
196         try:
197             if INSTALLATION_TYPE == "conda":
198                 cmd = [self.conda_path, "update", "-y", "osdag", "
                        --channel", "osdag"]
199             elif INSTALLATION_TYPE == "pixi":
200                 cmd = [self.pixi_path, "update", "-y", "osdag", "
                        --channel", "osdag"]
201             else:
202                 self.update_text("<p style='color:red;'>Unknown
                        installation type.</p>")
203                 return
204 
205             self.progressBar.show()
206             self.progressBar.setRange(0, 0)
207             self.updateNowButton.hide()
208             self.updateLaterButton.hide()
209 
210             # Configure process
211             self.process = QProcess(self)
212             self.process.readyReadStandardOutput.connect(self.
                    handle_stdout_update)
213             self.process.readyReadStandardError.connect(self.
                    handle_stderr_update)
214             self.process.finished.connect(self.
                    handle_update_finished)
215 
216             # Start update
217             self.update_text("<p style='color:blue;'>Updating
                    Osdag to the latest version...</p>")
218             self.process.start(cmd[0], cmd[1:])
219 
220         except Exception as e:
```

```
221             self.update_text(f"<p style='color:red;'>Update failed
                    : {e}</p>")
222
223
224     def update_text(self, html: str):
225         self.textBrowser.setHtml(f"<p'>{html}</p>")
226
227
228     def handle_stdout_update(self):
229         output = self.process.readAllStandardOutput().data().
                decode()
230         self.update_text(f"<pre style='color:#90AF13a'>{output}</
                pre>")
231
232     def handle_stderr_update(self):
233         error = self.process.readAllStandardError().data().decode
                ()
234         self.update_text(f"<pre style='color:red;'>{error}</pre>")
235
236     def handle_update_finished(self):
237         self.progressBar.hide()
238         self.okButton.show()
239         exit_code = self.process.exitCode()
240         if exit_code == 0:
241             self.update_text("<p style='color:#90AF13;'>Update
                    completed successfully!</p>")
242         else:
243             self.update_text(f"<p style='color:red;'>Update failed
                     with exit code {exit_code}.</p>")
```

Listing B.1: File: osdag_gui/ui/components/dialogs/check_for_updates.py

Chapter C

Plugin Management System - Code Files

C.1 Plugin Manager

```
import json
import os, subprocess, sys, requests, platform
import tempfile
from dataclasses import dataclass, field
from threading import Lock
from importlib import metadata
from pathlib import Path
import pkgutil
import importlib, uuid
from types import ModuleType


@dataclass
class PluginMetaData:
    id: str
    name: str
    description: str = field(
        default_factory=lambda: "No description available.")
    authors: list[str] = field(default_factory=lambda: ["Unknown"
       ])
    version: str = field(default_factory=lambda: "1.0.0")
```

```
21      status: bool = field(default_factory=lambda: False)
22      module: object = field(default_factory=lambda: None)
23      entry_class: object = field(default_factory=lambda: None)
24      is_dev: bool = field(default_factory=lambda: False)
25
26
27  class PluginManager:
28      def __init__(self):
29          self.state_manager = StateManager()
30          self.plugins: list[PluginMetaData] = []
31          self.dev_plugins_path = Path(
32              __file__).resolve().parent.parent.parent / "plugins"
33          self.plugins_entry_point = None
34          print(f"[INFO] PluginManager initialized.")
35
36      # ----------------------------
37      # Plugin Discovery (Local Under Development & Installed
            Plugins)
38      # ----------------------------
39      def discover_local_plugins(self) -> list[PluginMetaData]:
40          self.plugins.clear()
41
42          self._load_entry_point_plugins()
43          self._load_dev_plugins()
44
45          for plugin in self.plugins:
46              plugin.status = self.state_manager.get_status(plugin)
47
48          return self.plugins
49
50      def _load_entry_point_plugins(self) -> None:
51          try:
52              self.plugins_entry_point = metadata.entry_points().
                    select(group="osdag.plugins")
```

```
53              for ep in self.plugins_entry_point:
54                  module = ep.load()
55                  if not getattr(module, "IS_OSDAG_PLUGIN", False):
56                      print(
57                          f"[WARN] Entry point '{ep.name}' is not a
                              valid OSDAG plugin.")
58                      continue
59                  meta = getattr(module, "META", {})
60                  name = meta.get("name")
61                  if name is None:
62                      print(f"[WARN] Entry point '{ep.name}' is
                             missing 'name'.")
63                      continue
64                  if "osdag_plugin_" in name:
65                      name = name[len("osdag_plugin_"):]
66                      meta["name"] = name
67
68                  entry_point = getattr(module, "ENTRY_POINT", None)
69                  entry_class = self._resolve_entry_class(
70                      module, name, entry_point)
71                  if entry_class:
72                      self.plugins.append(PluginMetaData(
73                          **meta, status=False, module=module,
                              entry_class=entry_class))
74                  else:
75                      print(
76                          f"[WARN] Entry point '{ep.name}' could not
                              resolve entry class.")
77                      continue
78          except Exception as e:
79              print(f"[WARN] Entry point loading failed: {e}")
80
81      def _load_dev_plugins(self) -> None:
82          if not self.dev_plugins_path.exists():
```

```
            return
        for _, name, ispkg in pkgutil.iter_modules([str(self.
            dev_plugins_path)]):
            if not ispkg:
                continue
            if name in [p.name for p in self.plugins]:
                print(
                    f"[WARN] Dev plugin '{name}' is already loaded
                        from osdag.plugins entry point.")
                continue
            try:
                module = importlib.import_module(f"plugins.{name}"
                    )
                if not getattr(module, "IS_OSDAG_PLUGIN", False):
                    print(
                        f"[WARN] Dev plugin '{name}' is not a
                            valid OSDAG plugin.")
                    continue
                meta = getattr(module, "META", {})
                name = meta.get("name")
                if name is None:
                    print(f"[WARN] Dev plugin '{name}' is missing
                        'name'.")
                    continue
                entry_class = self._resolve_entry_class(
                    module, name, getattr(module, "ENTRY_POINT",
                        None))
                if entry_class:
                    self.plugins.append(PluginMetaData(
                        **meta, module=module, entry_class=
                            entry_class, is_dev=True))
                else:
                    print(
```

```
109                         f"[WARN] Dev plugin '{name}' could not
                                resolve entry class.")
110             except Exception as e:
111                 print(f"[WARN] Could not load dev plugins: {e}")
112
113     def _resolve_entry_class(self, module: ModuleType, name: str,
        entry_path: str) -> object | None:
114         if not entry_path:
115             print(f"[WARN] Plugin '{name}' missing 'ENTRY_POINT'."
                    )
116             return None
117         try:
118             parts = entry_path.split(".")
119             submodule_name = ".".join(parts[:-1])
120             class_name = parts[-1]
121             entry_module = importlib.import_module(
122                 f"{module.__name__}" + (f".{submodule_name}" if
                        submodule_name else ""))
123             return getattr(entry_module, class_name)
124         except Exception as e:
125             print(
126                 f"[WARN] Could not resolve entry class '{
                        entry_path}' for plugin '{name}': {e}")
127             return None
128
129
130     def discover_online_plugins(self, channel: str) -> list[
        PluginMetaData]:
131         # Simulate discovering online plugins[sys.executable, "-m
                ", "conda", "search", "--json", "-c", channel]
132         url = f"https://conda.anaconda.org/{channel}/label/main/
            noarch/repodata.json"
133         pkgs = self._fetch_channel_pkgs(url=url)
134         online_plugins = []
```

```
135         for pkg_filename, info in pkgs.items():
136             meta = {
137                 "id": str(uuid.uuid5(uuid.NAMESPACE_DNS, f"{info.
                       get('name', '')}:{info.get('version', '')}")),
138                 "name": info.get("name", ""),
139                 "description": info.get("summary", "No description
                        available."),
140                 "authors": [info.get("author", "Unknown")],
141                 "version": info.get("version", "1.0.0"),
142             }
143             online_plugins.append(PluginMetaData(**meta))
144         return online_plugins
145 
146     def _fetch_channel_pkgs(self, url:str):
147         """Fetch and parse repodata.json from a conda channel and
               return packages."""
148         resp = requests.get(url, timeout=20)
149         resp.raise_for_status()
150         repo_data = resp.json()
151         pkgs = {}
152         pkgs.update(repo_data.get("packages", {}))
153         pkgs.update(repo_data.get("packages.conda", {}))
154         return pkgs
155 
156     # ----------------------------
157     # State Management
158     # ----------------------------
159     def activate(self, plugin: PluginMetaData):
160         print(f"[INFO] Activating plugin: {plugin.name}")
161         if plugin:
162             plugin.status = True
163             self.state_manager.update_state(plugin, plugin.status)
164 
165     def deactivate(self, plugin: PluginMetaData):
```

```
166         print(f"[INFO] Deactivating plugin: {plugin.name}")
167         if plugin:
168             plugin.status = False
169             self.state_manager.update_state(plugin, plugin.status)
170 
171     def download_plugin(self, plugin: PluginMetaData) -> bool:
172         print(f"[INFO] Downloading plugin: {plugin.name}")
173 
174         if not hasattr(self, 'conda_exe'):
175             self.conda_exe = os.environ["CONDA_EXE"]
176 
177         cmd = [
178             self.conda_exe,
179             "install",
180             "--prefix", os.environ["CONDA_PREFIX"],
181             f"zehen-249::{plugin.name}",
182             "-y"
183         ]
184         result = subprocess.run(cmd,capture_output=True, text=True
               )
185         # print(result.stdout)
186         # print(result.stderr)
187         if result.returncode == 0:
188             self.plugins.append(plugin)
189             return True
190         return False
191 
192     def delete_plugin(self, plugin: PluginMetaData) -> bool:
193         print(f"[INFO] Deleting plugin: {plugin.name}")
194 
195         if not hasattr(self, 'conda_exe'):
196             self.conda_exe = os.environ["CONDA_EXE"]
197 
198         cmd = [
```

```
199             self.conda_exe,
200             "remove",
201             "--prefix", os.environ["CONDA_PREFIX"],
202             f"{plugin.name}",
203             "-y"
204         ]
205         result = subprocess.run(cmd, capture_output=True, text=
                True)
206         # print(result.stdout)
207         # print(result.stderr)
208         if result.returncode == 0:
209             self.plugins = [p for p in self.plugins if p.id !=
                    plugin.id]
210             self.state_manager.delete_state(plugin)
211             return True
212         return False
213
214     def update_plugin(self, plugin: PluginMetaData) -> bool:
215         print(f"[INFO] Updating plugin: {plugin.name}")
216
217         if not hasattr(self, 'conda_exe'):
218             self.conda_exe = os.environ["CONDA_EXE"]
219
220         cmd = [
221             self.conda_exe,
222             "update",
223             "--prefix", os.environ["CONDA_PREFIX"],
224             f"{plugin.name}",
225             "-y"
226         ]
227         result = subprocess.run(cmd, capture_output=True, text=
                True)
228         # print(result.stdout)
229         # print(result.stderr)
```

```
230         if result.returncode == 0:
231             return True
232         return False
233 
234 
235     def get_plugin(self, plugin_id: str) -> PluginMetaData | None:
236         for p in self.plugins:
237             if p.id == plugin_id:
238                 return p
239         return None
240 
241     def get_plugin_by_name(self, plugin_name: str) ->
        PluginMetaData | None:
242         for p in self.plugins:
243             if p.name == plugin_name:
244                 return p
245         return None
246 
247 
248 class StateManager:
249     def __init__(self):
250         self.state_file = Path(__file__).resolve(
251         ).parent.parent / "data" / "ResourceFiles" / "plugins" / "
            plugin_state.json"
252         self.state_file.parent.mkdir(parents=True, exist_ok=True)
253         self._lock = Lock()
254         self.states: dict[str, bool] = self._load_states()
255         self._dirty = False
256         print(f"[INFO] StateManager initialized")
257 
258     def _load_states(self):
259         if not os.path.exists(self.state_file):
260             return {}
261         with self._lock:
```

```
262             try:
263                 with open(self.state_file, 'r', encoding='utf-8')
                        as f:
264                     return json.load(f)
265             except (json.JSONDecodeError, OSError) as e:
266                 print(
267                     f"[WARN] Could not load state file {self.
                            state_file}: {e}")
268                 return {}
269
270     def get_status(self, plugin: PluginMetaData) -> bool:
271         with self._lock:
272             return self.states.get(plugin.id, False)
273
274     def update_state(self, plugin: PluginMetaData, status: bool)
            -> None:
275         with self._lock:
276             prev = self.states.get(plugin.id)
277             if prev != status:
278                 self.states[plugin.id] = status
279                 self._dirty = True
280
281     def delete_state(self, plugin: PluginMetaData) -> None:
282         with self._lock:
283             if plugin.id in self.states:
284                 del self.states[plugin.id]
285                 self._dirty = True
286
287     def flush(self):
288         with self._lock:
289             if not self._dirty:
290                 return
291             self._atomic_save(self.states)
292             self._dirty = False
```

```
293
294     def _atomic_save(self, data: dict[str, bool]):
295         dir_name = os.path.dirname(self.state_file)
296         if not dir_name:
297             dir_name = "."
298         try:
299             with tempfile.NamedTemporaryFile("w", dir=dir_name,
                    delete=False, encoding="utf-8") as tmp:
300                 json.dump(data, tmp, indent=4)
301                 tmp.flush()
302                 os.fsync(tmp.fileno())
303                 temp_name = tmp.name
304             os.replace(temp_name, self.state_file)
305         except Exception as e:
306             print(
307                 f"[ERROR] Failed to save state file {self.
                        state_file}: {e}")
```

Listing C.1: File: osdag_core/utils/plugin_manager.py

C.2 Plugin Manager Dialog

```
1
2 from PySide6.QtWidgets import (
3     QApplication,
4     QDialog,
5     QVBoxLayout,
6     QPushButton,
7     QHBoxLayout,
8     QWidget,
9     QSizePolicy,
10    QLabel,
11    QTextEdit,
12    QScrollArea
```

```
)
from PySide6.QtCore import Qt , Signal, Slot
from PySide6.QtSvgWidgets import QSvgWidget
from PySide6.QtGui import QIcon
from osdag_gui.ui.components.dialogs.custom_titlebar import
    CustomTitleBar
from osdag_gui.data.ui_data import Data
from osdag_core.utils.plugin_manager import PluginManager,
    PluginMetaData
from osdag_gui.ui.components.dialogs.plugin_store_dialog import
    PluginStoreDialog

class StatusIndicator(QWidget):
    def __init__(self, plugin: PluginMetaData = None, parent=None)
        :
        super().__init__(parent)
        layout = QHBoxLayout(self)
        layout.setContentsMargins(0, 0, 0, 0)
        layout.setAlignment(Qt.AlignCenter | Qt.AlignVCenter)
        layout.setSpacing(5)

        self.circle: QLabel = QLabel()
        self.circle.setFixedSize(12, 12)
        self.circle.setStyleSheet("border-radius: 6px; background-
            color: red;")

        self.text: QLabel = QLabel("Inactive")
        self.text.setStyleSheet(f"font-size: 9pt; font-weight:
            bold; color: red;")

        layout.addWidget(self.circle)
        layout.addWidget(self.text)

        self.update_status(plugin.status if plugin else False)
```

```
41
42      def update_status(self, status: bool) -> None:
43          if status:
44              self.circle.setStyleSheet("border-radius: 6px;
                    background-color: green;")
45              self.text.setText("Active")
46              self.text.setStyleSheet("color: green;")
47          else:
48              self.circle.setStyleSheet("border-radius: 6px;
                    background-color: red;")
49              self.text.setText("Inactive")
50              self.text.setStyleSheet("color: red;")
51
52  class PluginWidget(QWidget):
53      activate = Signal(PluginMetaData)
54      deactivate = Signal(PluginMetaData)
55      delete = Signal(PluginMetaData)
56
57      def __init__(self, plugin: PluginMetaData, parent=None):
58          super().__init__(parent)
59          self.plugin = plugin
60          layout = QVBoxLayout(self)
61          layout.setContentsMargins(8, 8, 8, 8)
62          layout.setSpacing(12)
63
64          self.setStyleSheet("""
65              PluginWidget {
66                  border: 1px solid #d0d0d0;
67                  border-radius: 6px;
68                  background-color: #ffffff;
69              }
70          """)
71
72          # --- HEADER ROW ---
```

```
73          header = QWidget()
74          header_layout = QHBoxLayout(header)
75          header_layout.setContentsMargins(2, 2, 2, 2)
76          header_layout.setSpacing(5)
77
78          self.name_label = QLabel(f"{'<b>'+plugin.name+'</b> (Dev
                Plugin)' if plugin.is_dev else '<b>'+plugin.name+'</b
                >'}")
79          self.name_label.setStyleSheet("font-weight: bold; font-
                size: 12pt; color: #90AF13;")
80          self.status_indicator = StatusIndicator(plugin)
81
82          header_layout.addWidget(self.name_label, alignment=Qt.
                AlignLeft)
83          header_layout.addStretch()
84          header_layout.addWidget(self.status_indicator, alignment=
                Qt.AlignRight)
85
86          layout.addWidget(header)
87
88          # --- CONTENT ROW ---
89          content = QWidget()
90          content_layout = QHBoxLayout(content)
91          content_layout.setContentsMargins(10, 10, 10, 10)
92          content_layout.setSpacing(15)
93
94          # LEFT SIDE (description)
95          self.content = QTextEdit()
96          self.content.setReadOnly(True)
97          content_text = f"{'<b>Author</b>' if len(self.plugin.
                authors) == 1 else '<b>Authors</b>'}: ({', '.join(
                author for author in self.plugin.authors)})"
98          content_text += f"<br><b>Description</b>: {self.plugin.
                description}"
```

```
99          self.content.setText(content_text)
100         self.content.setMinimumHeight(100)
101         self.content.setMaximumHeight(200)
102         self.content.setSizePolicy(QSizePolicy.Expanding,
                QSizePolicy.Expanding)
103         self.content.setStyleSheet("""
104             QTextEdit {
105                 border: 1px solid #e0e0e0;
106                 border-radius: 4px;
107                 font-size: 9.5pt;
108                 color: #444;
109                 background: #fafafa;
110             }
111         """)
112         content_layout.addWidget(self.content, stretch=3)
113
114
115
116         # RIGHT SIDE (buttons)
117         btn_layout = QVBoxLayout()
118         btn_layout.setContentsMargins(4, 4, 4, 4)
119         btn_layout.setSpacing(10)
120
121         self.btnActivate = QPushButton("Activate")
122         self.btnDeactivate = QPushButton("Deactivate")
123         self.btnDelete = QPushButton("Delete")
124
125         for btn in (self.btnActivate, self.btnDeactivate, self.
                btnDelete):
126             btn.setMinimumHeight(30)
127             btn.setMinimumWidth(100)
128             btn.setSizePolicy(QSizePolicy.Fixed, QSizePolicy.Fixed
                    )
129             btn.setStyleSheet("""
```

```
                QPushButton {
                    background-color: #90AF13;
                    color: white;
                    border: none;
                    border-radius: 5px;
                    font-size: 9pt;
                    font-weight: bold;
                    margin: 4px;
                }
                QPushButton:hover { background-color: #7A9611; }
                QPushButton:pressed { background-color: #6B850F; }
            """)
            btn_layout.addWidget(btn, 0, Qt.AlignRight)

        content_layout.addLayout(btn_layout, stretch=1)

        layout.addWidget(content)

        # --- Button Callbacks ---
        self.btnActivate.clicked.connect(self._emit_activate)
        self.btnDeactivate.clicked.connect(self._emit_deactivate)
        self.btnDelete.clicked.connect(self._emit_delete)

        content_min_width = self.content.sizeHint().width() + 150
        self.setMinimumWidth(content_min_width)

    def _emit_activate(self):
        if not self.plugin.status:
            self.plugin.status = not self.plugin.status
            self.status_indicator.update_status(self.plugin.status
                )
            self.activate.emit(self.plugin)

    def _emit_deactivate(self):
```

```
163             if self.plugin.status:
164                 self.plugin.status = not self.plugin.status
165                 self.status_indicator.update_status(self.plugin.status
                        )
166                 self.deactivate.emit(self.plugin)
167 
168     def _emit_delete(self):
169         self.delete.emit(self.plugin)
170 
171 
172 class PluginManagerDialog(QDialog):
173     def __init__(self, parent=None):
174         super().__init__(parent)
175         self.plugin_manager: PluginManager = QApplication.instance
                ().plugin_manager
176         self.plugins: list[PluginMetaData] = self.plugin_manager.
                discover_local_plugins()
177         self.active_plugins: list[tuple[str, str]] = Data.MODULES[
                "Add-Ons"]
178         self.setWindowFlags(Qt.FramelessWindowHint)
179         self.setAttribute(Qt.WA_StyledBackground, True)
180         self.setObjectName("PluginManagerDialog")
181         self.setWindowIcon(QIcon(":/images/osdag_logo.png"))
182         self.setFixedSize(720, 600)
183 
184         # Layout and style
185         self.setStyleSheet("""
186             QDialog#PluginManagerDialog { background-color: #
                    ffffff; border: 1px solid #90AF13; }
187             QWidget#ContentWidget { background-color: #ffffff; }
188             QPushButton { background-color: #90AF13; color: white;
                     border: none; border-radius: 5px;
189                           padding: 5px 20px; font-size: 12px; font
                              -weight: bold; }
```

```
            QPushButton:hover { background-color: #7A9611; }
            QPushButton:pressed { background-color: #6B850F; }
        """)

        mainLayout = QVBoxLayout(self)
        mainLayout.setContentsMargins(1, 1, 1, 1)
        mainLayout.setSpacing(0)

        # Title bar
        self.titleBar = CustomTitleBar()
        self.titleBar.setTitle("Plugin Manager")
        mainLayout.addWidget(self.titleBar)

        # Content
        contentWidget = QWidget(self)
        contentWidget.setObjectName("ContentWidget")
        contentLayout = QVBoxLayout(contentWidget)
        contentLayout.setContentsMargins(10, 10, 10, 10)
        contentLayout.setSpacing(10)

        # Logo
        self.logoLabel = QSvgWidget(":/vectors/Osdag.svg", self)
        self.logoLabel.setFixedSize(325, 85)
        self.logoLabel.setSizePolicy(QSizePolicy.Expanding,
            QSizePolicy.Fixed)
        contentLayout.addWidget(self.logoLabel, 0, Qt.AlignLeft)

        # Scroll area for plugins
        scroll = QScrollArea()
        scroll.setWidgetResizable(True)
        scroll.setStyleSheet("QScrollArea { border: 0.5px solid
            black; background-color: #F0F0F0; }")
        contentLayout.addWidget(scroll)

```

```
222         # Container for plugin widgets
223         self.pluginContainer = QWidget()
224         self.pluginLayout = QVBoxLayout(self.pluginContainer)
225         self.pluginLayout.setAlignment(Qt.AlignTop)
226         scroll.setWidget(self.pluginContainer)
227 
228         if len(self.plugins) > 0:
229             print("\n===========Loaded Plugins===========\n")
230             for plugin in self.plugins:
231                 print(f"    {plugin.name} by {', '.join(author for
                        author in plugin.authors)}")
232                 if plugin.status:
233                     print("     [INFO] Status: Active")
234                     self.activate_plugin(plugin)
235                 else:
236                     print("     [INFO] Status: Inactive")
237                 pw = PluginWidget(plugin=plugin, parent=self)
238                 pw.activate.connect(self.plugin_manager.activate)
239                 pw.activate.connect(self.activate_plugin)
240                 pw.deactivate.connect(self.plugin_manager.
                        deactivate)
241                 pw.deactivate.connect(self.deactivate_plugin)
242                 pw.delete.connect(self.plugin_manager.
                        delete_plugin)
243                 pw.setFixedHeight(120)
244                 pw.setSizePolicy(QSizePolicy.Expanding,
                        QSizePolicy.Fixed)
245                 self.pluginLayout.addWidget(pw)
246             print("\n====================================\n")
247 
248         # Plugin Store button
249         self.storeButton = QPushButton("Store", self)
250         self.storeButton.setFixedHeight(30)
251         self.storeButton.setStyleSheet("""
```

```
252             QPushButton {
253                 background-color: #0078D7; color: white; border:
                        none; border-radius: 5px;
254                 padding: 5px 20px; font-size: 12px; font-weight:
                        bold;
255             }
256             QPushButton:hover { background-color: #0063B1; }
257             QPushButton:pressed { background-color: #004E8C; }
258         """)
259         self.storeButton.clicked.connect(self.open_store_dialog)
260
261
262         # OK button
263         buttonLayout = QHBoxLayout()
264         buttonLayout.addWidget(self.storeButton)
265         buttonLayout.addStretch()
266         self.okButton = QPushButton("OK", self)
267         self.okButton.setFixedHeight(30)
268         self.okButton.setStyleSheet("""
269             QPushButton { background-color: #90AF13; color: white;
                    border: none; border-radius: 5px;
270                             padding: 5px 20px; font-size: 12px;
                                font-weight: bold; }
271             QPushButton:hover { background-color: #7A9611; }
272             QPushButton:pressed { background-color: #6B850F; }
273         """)
274         self.okButton.clicked.connect(self.okClicked)
275         buttonLayout.addWidget(self.okButton)
276
277         contentLayout.addLayout(buttonLayout)
278         mainLayout.addWidget(contentWidget)
279
280
281
```

```
282     def activate_plugin(self, plugin: PluginMetaData):
283         plugin = self.plugin_manager.get_plugin(plugin.id)
284         if plugin and (plugin.name, ":/vectors/IITB_logo.svg") not
                in self.active_plugins:
285             self.active_plugins.append((plugin.name, ":/vectors/
                    IITB_logo.svg"))
286     def deactivate_plugin(self, plugin: PluginMetaData):
287         self.active_plugins[:] = [
288             p for p in self.active_plugins if p[0] != plugin.name
289         ]
290 
291     def delete_plugin(self, plugin: PluginMetaData):
292         self.deactivate_plugin(plugin)
293         self.plugin_manager.delete_plugin(plugin)
294         self.refresh_plugin()
295 
296     def open_store_dialog(self):
297         # print("[PLUGIN STORE] Opening Plugin Store Dialog...")
298         store_dialog = PluginStoreDialog()
299         store_dialog.exec()
300 
301 
302     def refresh_plugins(self):
303         self.plugins = self.plugin_manager.discover_local_plugins
                ()
304         # Clear old plugin widgets
305         for i in reversed(range(self.pluginLayout.count())):
306             widget = self.pluginLayout.itemAt(i).widget()
307             if widget:
308                 widget.setParent(None)
309 
310         # Re-add plugin widgets
311         if len(self.plugins) > 0:
312             for plugin in self.plugins:
```

```
313                 pw = PluginWidget(plugin=plugin, parent=self)
314                 pw.activate.connect(self.plugin_manager.activate)
315                 pw.activate.connect(self.activate_plugin)
316                 pw.deactivate.connect(self.plugin_manager.
                        deactivate)
317                 pw.deactivate.connect(self.deactivate_plugin)
318                 pw.delete.connect(self.plugin_manager.
                        delete_plugin)
319                 pw.setFixedHeight(120)
320                 pw.setSizePolicy(QSizePolicy.Expanding,
                        QSizePolicy.Fixed)
321                 self.pluginLayout.addWidget(pw)
322
323     def closeEvent(self, event):
324         self.plugin_manager.state_manager.flush()
325         super().closeEvent(event)
326
327     def okClicked(self):
328         self.plugin_manager.state_manager.flush()
329         self.accept()
```

Listing C.2: File: osdag_gui/ui/components/dialogs/plugin_manager_dialog.py

C.3 Plugin Store Dialog

```
1
2 from packaging import version
3 from PySide6.QtWidgets import (
4     QApplication,
5     QDialog,
6     QVBoxLayout,
7     QPushButton,
8     QHBoxLayout,
9     QWidget,
```

```
    QSizePolicy,
    QLabel,
    QTextEdit,
    QScrollArea
)
from PySide6.QtCore import Qt , Signal, Slot, QObject, QThread,
   QTimer, QMetaObject
from PySide6.QtSvgWidgets import QSvgWidget
from PySide6.QtGui import QIcon
from osdag_gui.ui.components.dialogs.custom_titlebar import
   CustomTitleBar
from osdag_core.utils.plugin_manager import PluginMetaData

class Worker(QObject):
    finished = Signal(bool)

    def __init__(self, func, *args, **kwargs):
        super().__init__()
        self.func = func
        self.args = args
        self.kwargs = kwargs

    @Slot()
    def run(self):
        import threading
        try:
            result = self.func(*self.args, **self.kwargs)
            self.finished.emit(bool(result))
        except Exception as e:
            print(f"[WORKER] Exception: {e}")
            self.finished.emit(False)

class PluginWidget(QWidget):
    download = Signal(PluginMetaData)
```

```
42      delete = Signal(PluginMetaData)
43      update = Signal(PluginMetaData)
44      def __init__(self, plugin: PluginMetaData, parent=None):
45          super().__init__(parent)
46          self.plugin = plugin
47          layout = QVBoxLayout(self)
48          layout.setContentsMargins(8, 8, 8, 8)
49          layout.setSpacing(12)
50
51          self.setStyleSheet("""
52              PluginWidget {
53                  border: 1px solid #d0d0d0;
54                  border-radius: 6px;
55                  background-color: #ffffff;
56              }
57          """)
58
59          # --- HEADER ROW ---
60          header = QWidget()
61          header_layout = QHBoxLayout(header)
62          header_layout.setContentsMargins(2, 2, 2, 2)
63          header_layout.setSpacing(5)
64
65          self.name_label = QLabel(f"{'<b>'+plugin.name+'</b>' }")
66          self.name_label.setStyleSheet("font-weight: bold; font-
                size: 12pt; color: #90AF13;")
67          # self.status_indicator = StatusIndicator(plugin)
68
69          header_layout.addWidget(self.name_label, alignment=Qt.
                AlignLeft)
70          # header_layout.addStretch()
71          # header_layout.addWidget(self.status_indicator, alignment
                =Qt.AlignRight)
72
```

```
73          layout.addWidget(header)
74
75          # --- CONTENT ROW ---
76          content = QWidget()
77          content_layout = QHBoxLayout(content)
78          content_layout.setContentsMargins(10, 10, 10, 10)
79          content_layout.setSpacing(15)
80
81          # LEFT SIDE (description)
82          self.content = QTextEdit()
83          self.content.setReadOnly(True)
84          content_text = f"{'<b>Author</b>' if len(self.plugin.
               authors) == 1 else '<b>Authors</b>'}: ({', '.join(
               author for author in self.plugin.authors)})"
85          content_text += f"<br><b>Description</b>: {self.plugin.
               description}"
86          self.content.setText(content_text)
87          self.content.setMinimumHeight(100)
88          self.content.setMaximumHeight(200)
89          self.content.setSizePolicy(QSizePolicy.Expanding,
               QSizePolicy.Expanding)
90          self.content.setStyleSheet("""
91              QTextEdit {
92                  border: 1px solid #e0e0e0;
93                  border-radius: 4px;
94                  font-size: 9.5pt;
95                  color: #444;
96                  background: #fafafa;
97              }
98          """)
99          content_layout.addWidget(self.content, stretch=3)
100
101
102
```

```
103         # RIGHT SIDE (buttons)
104         btn_layout = QVBoxLayout()
105         btn_layout.setContentsMargins(4, 4, 4, 4)
106         btn_layout.setSpacing(10)
107 
108         self.btnDownload = QPushButton("Download")
109         self.btnDelete = QPushButton("Delete")
110         self.btnUpdate = QPushButton("Update")
111 
112         for btn in (self.btnDownload, self.btnDelete, self.
                btnUpdate):
113             btn.setMinimumHeight(30)
114             btn.setMinimumWidth(100)
115             btn.setSizePolicy(QSizePolicy.Fixed, QSizePolicy.Fixed
                    )
116             btn.setStyleSheet("""
117                 QPushButton {
118                     background-color: #90AF13;
119                     color: white;
120                     border: none;
121                     border-radius: 5px;
122                     font-size: 9pt;
123                     font-weight: bold;
124                     margin: 4px;
125                 }
126                 QPushButton:hover { background-color: #7A9611; }
127                 QPushButton:pressed { background-color: #6B850F; }
128             """)
129             btn_layout.addWidget(btn, 0, Qt.AlignRight)
130 
131         content_layout.addLayout(btn_layout, stretch=1)
132 
133         layout.addWidget(content)
134 
```

```
135             # --- Button Callbacks ---
136             self.btnDelete.clicked.connect(self._emit_delete)
137             self.btnDownload.clicked.connect(self._emit_download)
138             self.btnUpdate.clicked.connect(self._emit_update)
139 
140             content_min_width = self.content.sizeHint().width() + 150
141             self.setMinimumWidth(content_min_width)
142 
143         def _emit_download(self):
144             self.download.emit(self.plugin)
145 
146         def _emit_delete(self):
147             self.delete.emit(self.plugin)
148 
149         def _emit_update(self):
150             self.update.emit(self.plugin)
151 
152 
153 
154 
155 class PluginStoreDialog(QDialog):
156     def __init__(self, parent=None):
157         super().__init__(parent)
158         self.plugin_manager = QApplication.instance().
              plugin_manager
159         self.plugins: list[PluginMetaData] = self.plugin_manager.
              discover_online_plugins(channel="zehen-249")
160         self.local_plugins: list[PluginMetaData] = self.
              plugin_manager.discover_local_plugins()
161         self.updates_available = self._check_for_updates()
162         self.setWindowFlags(Qt.FramelessWindowHint)
163         self.setAttribute(Qt.WA_StyledBackground, True)
164         self.setObjectName("PluginManagerDialog")
165         self.setWindowIcon(QIcon(":/images/osdag_logo.png"))
```

```
166
167
168         mainLayout = QVBoxLayout(self)
169         mainLayout.setContentsMargins(1, 1, 1, 1)
170         mainLayout.setSpacing(0)
171
172         # Title bar
173         self.titleBar = CustomTitleBar()
174         self.titleBar.setTitle("Plugin Store")
175         mainLayout.addWidget(self.titleBar)
176
177         # Content
178         contentWidget = QWidget(self)
179         contentWidget.setObjectName("ContentWidget")
180         contentLayout = QVBoxLayout(contentWidget)
181         contentLayout.setContentsMargins(10, 10, 10, 10)
182         contentLayout.setSpacing(10)
183
184         # Logo
185         self.logoLabel = QSvgWidget(":/vectors/Osdag.svg", self)
186         self.logoLabel.setFixedSize(325, 85)
187         self.logoLabel.setSizePolicy(QSizePolicy.Expanding,
                QSizePolicy.Fixed)
188         contentLayout.addWidget(self.logoLabel, 0, Qt.AlignLeft)
189
190         # Scroll area for plugins
191         scroll = QScrollArea()
192         scroll.setWidgetResizable(True)
193         scroll.setStyleSheet("QScrollArea { border: 0.5px solid
                black; background-color: #F0F0F0; }")
194         contentLayout.addWidget(scroll)
195
196
197         # Container for plugin widgets
```

```
198         self.pluginContainer = QWidget()
199         self.pluginLayout = QVBoxLayout(self.pluginContainer)
200         self.pluginLayout.setAlignment(Qt.AlignTop)
201         scroll.setWidget(self.pluginContainer)
202 
203         # Populate plugins in the scroll area
204         self._populate_plugins()
205 
206          # OK button
207         buttonLayout = QHBoxLayout()
208         buttonLayout.addStretch()
209         self.okButton = QPushButton("OK", self)
210         self.okButton.setFixedHeight(30)
211         self.okButton.setStyleSheet("""
212             QPushButton { background-color: #90AF13; color: white;
                    border: none; border-radius: 5px;
213                             padding: 5px 20px; font-size: 12px;
                                font-weight: bold; }
214             QPushButton:hover { background-color: #7A9611; }
215             QPushButton:pressed { background-color: #6B850F; }
216         """)
217         self.okButton.clicked.connect(self.okClicked)
218         buttonLayout.addWidget(self.okButton)
219 
220         contentLayout.addLayout(buttonLayout)
221         mainLayout.addWidget(contentWidget)
222 
223     def delete_plugin(self, plugin: PluginMetaData, widget:
        PluginWidget):
224         widget.btnDelete.setEnabled(False)
225         widget.name_label.setText(f"<b>{plugin.name} (Deleting...)
            </b>")
226         QApplication.processEvents()  # ensures repaint before
            thread starts
```

```
227
228         thread = QThread()
229         worker = Worker(self.plugin_manager.delete_plugin, plugin)
230         worker.moveToThread(thread)
231
232         # Connect signals
233         thread.started.connect(lambda : self.start_worker(worker))
234         worker.finished.connect(lambda success: self.
                _on_delete_finished(success, widget, plugin))
235         worker.finished.connect(thread.quit)
236         worker.finished.connect(worker.deleteLater)
237         thread.finished.connect(thread.deleteLater)
238
239         # Keep a reference so thread isn't Garbagecollected
240         if not hasattr(self, "plugin_store_threads"):
241             self.plugin_store_threads = []
242         self.plugin_store_threads.append(thread)
243         thread.finished.connect(lambda: self.plugin_store_threads.
                remove(thread))
244         thread.start()
245
246     def _on_delete_finished(self, success, widget, plugin):
247         if success:
248             print(f"[INFO] Successfully deleted plugin: {plugin.
                    name}")
249             widget.btnDelete.setVisible(False)
250             widget.btnDownload.setVisible(True)
251             widget.btnDownload.setEnabled(True)
252             widget.name_label.setText(f"<b>{plugin.name}</b>")
253         else:
254             print(f"[INFO] Failed to delete plugin: {plugin.name}"
                    )
255             widget.name_label.setText(f"<b>{plugin.name} (
                    Downloaded)</b>")
```

```
256             widget.btnDelete.setEnabled(True)
257
258     def download_plugin(self, plugin: PluginMetaData, widget:
            PluginWidget):
259         widget.btnDownload.setEnabled(False)
260         widget.name_label.setText(f"<b>{plugin.name} (Downloading
                ...)</b>")
261         QApplication.processEvents()  # ensures repaint before
                thread starts
262
263         thread = QThread()
264         worker = Worker(self.plugin_manager.download_plugin,
                plugin)
265         worker.moveToThread(thread)
266
267         # Connect signals
268         thread.started.connect(lambda : self.start_worker(worker))
269         worker.finished.connect(lambda success: self.
                _on_download_finished(success, widget, plugin))
270         worker.finished.connect(thread.quit)
271         worker.finished.connect(worker.deleteLater)
272         thread.finished.connect(thread.deleteLater)
273
274
275         # Keep a reference so thread isn't Garbagecollected
276         if not hasattr(self, "plugin_store_threads"):
277             self.plugin_store_threads = []
278         self.plugin_store_threads.append(thread)
279         thread.finished.connect(lambda: self.plugin_store_threads.
                remove(thread))
280         thread.start()
281
282     def _on_download_finished(self, success, widget, plugin):
283         if success:
```

```
284             print(f"[INFO] Successfully downloaded plugin: {plugin
                    .name}")
285             widget.btnDownload.setVisible(False)
286             widget.btnDelete.setVisible(True)
287             widget.btnDelete.setEnabled(True)
288             widget.name_label.setText(f"<b>{plugin.name} (
                    Downloaded)</b>")
289         else:
290             print(f"[INFO] Failed to download plugin: {plugin.name
                    }")
291             widget.name_label.setText(f"<b>{plugin.name}</b>")
292             widget.btnDownload.setEnabled(True)
293 
294     def update_plugin(self, plugin: PluginMetaData, widget:
            PluginWidget):
295         widget.btnUpdate.setEnabled(False)
296         widget.name_label.setText(f"<b>{plugin.name} (Updating...)
                </b>")
297         QApplication.processEvents()  # ensures repaint before
                thread starts
298 
299         thread = QThread()
300         worker = Worker(self.plugin_manager.update_plugin, plugin)
301         worker.moveToThread(thread)
302 
303         # Connect signals
304         thread.started.connect(lambda : self.start_worker(worker))
305         worker.finished.connect(lambda success: self.
                _on_update_finished(success, widget, plugin))
306         worker.finished.connect(thread.quit)
307         worker.finished.connect(worker.deleteLater)
308         thread.finished.connect(thread.deleteLater)
309 
310 
```

```
311         # Keep a reference so thread isn't Garbagecollected
312         if not hasattr(self, "plugin_store_threads"):
313             self.plugin_store_threads = []
314         self.plugin_store_threads.append(thread)
315         thread.finished.connect(lambda: self.plugin_store_threads.
                remove(thread))
316         thread.start()
317 
318     def _on_update_finished(self, success, widget, plugin):
319         if success:
320             print(f"[INFO] Successfully updated plugin: {plugin.
                    name}")
321             widget.btnUpdate.setVisible(False)
322             widget.btnDownload.setVisible(False)
323             widget.btnDelete.setVisible(True)
324             widget.btnDelete.setEnabled(True)
325             widget.name_label.setText(f"<b>{plugin.name} (
                    Downloaded)</b>")
326         else:
327             print(f"[INFO] Failed to update plugin: {plugin.name}"
                    )
328             widget.name_label.setText(f"<b>{plugin.name} (Update
                    Available)</b>")
329             widget.btnUpdate.setEnabled(True)
330 
331     def start_worker(self, worker):
332             QMetaObject.invokeMethod(worker, "run", Qt.
                    QueuedConnection)
333 
334     def refresh_plugins(self):
335         # Re-discover both local and online plugins
336         self.plugins = self.plugin_manager.discover_online_plugins
                ()
```

```
337         self.local_plugins = self.plugin_manager.
               discover_local_plugins()
338 
339         # Clear all plugin widgets from the layout
340         for i in reversed(range(self.pluginLayout.count())):
341             widget = self.pluginLayout.itemAt(i).widget()
342             if widget:
343                 widget.setParent(None)
344 
345         # Rebuild plugin list widgets
346         self._populate_plugins()
347 
348     def _populate_plugins(self):
349 
350         latest = {}
351         for plugin in self.plugins:
352             if plugin.name not in latest:
353                 latest[plugin.name] = plugin
354             else:
355                 if version.parse(plugin.version) > version.parse(
                       latest[plugin.name].version):
356                     latest[plugin.name] = plugin
357 
358         self.plugins = list(latest.values())
359 
360         for plugin in self.plugins:
361             pw = PluginWidget(plugin=plugin, parent=self)
362 
363             pw.download.connect(lambda plugin, w=pw: self.
                   download_plugin(plugin, w))
364             pw.delete.connect(lambda plugin, w=pw: self.
                   delete_plugin(plugin, w))
365             pw.update.connect(lambda plugin, w=pw: self.
                   update_plugin(plugin, w))
```

```
366
367             pw.setFixedHeight(120)
368             pw.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.
                    Fixed)
369
370             is_downloaded = any(lp.name == plugin.name for lp in
                    self.local_plugins)
371
372             if is_downloaded:
373                 is_update_available = any(up.name == plugin.name
                        for up in self.updates_available)
374
375                 if is_update_available:
376                     pw.name_label.setText(f"<b>{plugin.name} (
                            Update Available)</b>")
377                     pw.btnDownload.setVisible(False)
378                     pw.btnUpdate.setVisible(True)
379                     pw.btnUpdate.setEnabled(True)
380                     pw.btnDelete.setVisible(True)
381                     pw.btnDelete.setEnabled(True)
382                 else:
383                     pw.name_label.setText(f"<b>{plugin.name} (
                            Downloaded)</b>")
384                     pw.btnDownload.setVisible(False)
385                     pw.btnUpdate.setVisible(False)
386                     pw.btnDelete.setVisible(True)
387                     pw.btnDelete.setEnabled(True)
388
389             else:
390                 pw.name_label.setText(f"<b>{plugin.name}</b>")
391                 pw.btnDownload.setVisible(True)
392                 pw.btnDownload.setEnabled(True)
393                 pw.btnUpdate.setVisible(False)
394                 pw.btnDelete.setVisible(False)
```

```
395
396             self.pluginLayout.addWidget(pw)
397
398
399     def _check_for_updates(self):
400         installed_plugins = self.plugin_manager.
               discover_local_plugins()
401         online_plugins = self.plugin_manager.
               discover_online_plugins(channel="zehen-249")
402         updates_available = []
403         for installed in installed_plugins:
404             for online in online_plugins:
405                 if online.name != installed.name:
406                     continue
407
408                 if version.parse(online.version) > version.parse(
                       installed.version):
409                     updates_available.append(online)
410
411
412         return updates_available
413
414     def closeEvent(self, event):
415         self.plugin_manager.state_manager.flush()
416         self._refreshManager()
417         super().closeEvent(event)
418
419     def okClicked(self):
420         self.plugin_manager.state_manager.flush()
421         self._refreshManager()
422         self.accept()
423
424     def _refreshManager(self):
```

```
425         self.plugin_manager_dialog = QApplication.instance().
                plugin_manager_dialog
426         self.plugin_manager_dialog.refresh_plugins()
```

Listing C.3: File: osdag_gui/ui/components/dialogs/plugin_store_dialog.py

Chapter D

Developmento of Plugin - A Demo Plugin

This appendix contains the reference implementation of the Osdag plugin framework and example plugins used during this internship.

D.1 Directory Structure of Demo Plugin

```
demo_plugin/
|
|-- conda-recipe/
|   |-- meta.yaml
|
|-- demo_plugin/
|   |-- __init__.py
|   |-- demo_plugin.py
|
`-- pyproject.toml
```

Listing D.1: Demo Plugin Source Tree

D.2 pyproject.toml

```toml
# pyproject.toml

[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"

[project]
name = "demo_plugin"
version = "1.0.0"
description = "A demonstration plugin for Osdag."
authors = [
    {name = "FOSSEE Team"},
]

license = {text = "LGPL"}
requires-python = ">=3.10"
dependencies = [

]

[tool.setuptools]
packages = ["demo_plugin"]

# Entry point group `osdag.plugins` tells Osdag where to find this
    plugin.
[project.entry-points."osdag.plugins"]
demo_plugin = "demo_plugin"
```

Listing D.2: pyproject.toml

D.3 meta.yaml

```yaml
# conda-recipe/meta.yaml

```

```yaml
{% set name = "demo_plugin" %}
{% set version = "1.0.0" %}

package:
  name: {{ name|lower }}
  version: {{ version }}

source:
  path: ..

build:
  noarch: python
  script: "{{ PYTHON }} -m pip install . --no-deps -vv"

requirements:
  host:
    - python >=3.10
    - pip
    - setuptools >=61.0
  run:
    - osdag::osdag >=2025.01.a.2

about:
  license: LGPL
  summary: "A demonstration plugin for Osdag."
  description: |
    A demo plugin that integrates with the Osdag platform through
       the `osdag.plugins` entry point.
  authors:
    - FOSSEE Team
    - ABC
    - XYZ

extra:
```

```
  recipe-maintainers:
    - fossee-contributors
```

Listing D.3: Conda Recipe: meta.yaml

D.4 __init__.py

```python
# demo_plugin/__init__.py

from importlib.metadata import metadata
import uuid

_meta = metadata("demo_plugin")
IS_OSDAG_PLUGIN = True
META = {
    "id": str(uuid.uuid5(uuid.NAMESPACE_DNS, f"{_meta['name']}:{
        _meta['version']}")),
    "name": _meta["name"],
    "description": _meta["Summary"],
    "authors": [a.strip() for a in _meta["Author"].split(",")], #
        List of <str>
    "version": _meta["version"],
}

# --- Entry point: the class Osdag will instantiate on activation
    ---
ENTRY_POINT = "demo_plugin.DemoPlugin"
```

Listing D.4: Plugin Package Initialization

D.5 demo_plugin.py

```python
# demo_plugin/demo_plugin.py

```

```
from osdag_gui.plugins.widget_plugin import WidgetPlugin
from PySide6.QtWidgets import QLabel, QPushButton, QLineEdit,
    QHBoxLayout, QDialog, QVBoxLayout

class DemoPlugin(WidgetPlugin):

    def __init__(self, parent=None, title="Demo Plugin"):
        super().__init__(parent, title)
        self.show_default_ui_message()

    def setupUI(self):
        super().setupUI()
        # Add simple form-like UI
        label = QLabel("Enter your name:")
        input_box = QLineEdit()
        input_box.setPlaceholderText("Type something...")
        button = QPushButton("Submit")

        row_layout = QHBoxLayout()
        row_layout.addWidget(label)
        row_layout.addWidget(input_box)
        row_layout.addWidget(button)

        self.layout.addLayout(row_layout)
        self.layout.addWidget(QLabel("This is a demo plugin UI!"))

    def show_default_ui_message(self):
        """Custom window/dialog to show default plugin UI info."""
        dialog = QDialog(self)
        dialog.setWindowTitle("Osdag Plugin Info")

        dialog_layout = QVBoxLayout(dialog)
        message = QLabel(
```

```
            "<b>This is the default UI for an Osdag Plugin.</b><br>"
            "You can customize it by modifying the DemoPlugin class."
        )
        message.setWordWrap(True)
        ok_button = QPushButton("OK")
        ok_button.clicked.connect(dialog.accept)

        dialog_layout.addWidget(message)
        dialog_layout.addWidget(ok_button)

        dialog.exec()
```

Listing D.5: Plugin Entry Implementation

D.6 Plugin Classes

D.6.1 PluginBase

`PluginBase` is the root abstraction for all plugins:

```
# osdag_gui/plugins/plugin_base.py

class PluginBase:
    """Base class for all Osdag plugins."""
    def __init__(self, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)

    def setupUI(self) -> None:
        """Setup UI for the Plugin"""
        pass
```

Listing D.6: PluginBase Definition

This class defines the minimal interface expected by the Plugin Manager. All concrete

plugins must implement the `setupUI()` method, which acts as the primary initialization hook.

D.6.2 WidgetPlugin

`WidgetPlugin` is intended for plugins that integrate as dockable components inside the Osdag main window:

```python
# osdag_gui/plugins/widget_plugin.py

from abc import abstractmethod
from PySide6.QtWidgets import (
    QVBoxLayout,
    QWidget,
    QLabel,
)
from osdag_gui.plugins.plugin_base import PluginBase

class WidgetPlugin(PluginBase, QWidget):
    """Base class for widget-based plugins."""

    def __init__(self, parent: QWidget = None, title: str = "
       Default Title") -> None:
        super().__init__(parent)
        self.title = title

    def setupUI(self) -> None:
        """Override in subclass to create your plugin's UI."""
        self.layout = QVBoxLayout(self)
        label = QLabel(f"Default UI for {self.title}")
        self.layout.addWidget(label)
```

Listing D.7: WidgetPlugin Definition

Inheritance from `QWidget` allows the plugin to be embedded directly into the Osdag workspace. The framework manages the container lifecycle, while the developer controls only the internal layout.

D.6.3 WindowPlugin

`WindowPlugin` is intended for standalone tools launched from Osdag:

```python
# osdag_gui/plugins/window_plugin.py

from PySide6.QtWidgets import (
    QApplication,
    QDialog,
    QVBoxLayout,
    QPushButton,
    QHBoxLayout,
    QWidget,
    QMainWindow,
    QSizePolicy,
    QLabel,
    QTextEdit,
    QScrollArea
)
from osdag_gui.plugins.plugin_base import PluginBase

class WindowPlugin(PluginBase, QMainWindow):
    """Plugin that provides a Window."""

    def setupUI(self) -> None:
        """Setup UI of the Window."""
        super().setupUI()
        self.window = QMainWindow()

        return
```

Listing D.8: WindowPlugin Definition

Inheritance from `QMainWindow` enables the plugin to behave as an independent application window while remaining discoverable and launchable through Osdag.

Chapter E

Work Report

Date	DAY	TASK	# Hours worked
1 Aug 2025	Friday	Read on .osi file	3
4 Aug 2025	Monday	Read on .osi file	4
5 Aug 2025	Tuesday	Read on .osi file	2
6 Aug 2025	Wednesday	Go through the Osdag web and desktop new folser struc	4
7 Aug 2025	Thursday	Go through the Osdag web and desktop new folser struc	4
8 Aug 2025	Friday	See how to automate .osi file loading	4
11 Aug 2025	Monday	Help Faran to move core funtionality from old repo to ne	4
12 Aug 2025	Tuesday	See how to automate .osi file loading	2
13 Aug 2025	Wednesday	Go through the documents on .osi file from prev interns	4
14 Aug 2025	Thursday	Go through the documents on .osi file from prev interns	2
15 Aug 2025	Friday	Go through the documents on .osi file from prev interns	4
20 Aug 2025	Wednesday	Update feature in osdag desktop	4
21 Aug 2025	Thursday	Update feature in osdag desktop	5
22 Aug 2025	Friday	Update feature in osdag desktop	3
25 Aug 2025	Monday	read ui_template.py ui methods flow of input data and m	4
26 Aug 2025	Tuesday	read ui_template.py ui methods flow of input data and m	4
27 Aug 2025	Wednesday	Fix circular imports between is800_2007.py and Commc	4
28 Aug 2025	Thursday	See how to automate .osi file loading	4
29 Aug 2025	Friday	Import Tension_bolted module and generate design outp	4
1 Sep 2025	Monday	Import Tension_bolted module and generate design outp	4
2 Sep 2025	Tuesday	Stuck at arugements issue. Discussed with suhail and p	4
3 Sep 2025	Wednesday	Resolved issue and implemented the cli tool	6
4 Sep 2025	Thursday	Resolved issue and implemented the cli tool	4
5 Sep 2025	Friday	Refer to aum's work and migrate cli tool from argparser t	4
8 Sep 2025	Monday	Make the CLI tool robust	4
9 Sep 2025	Tuesday	Make the CLI tool robust	3
10 Sep 2025	Wednesday	Finalize the update feature with suggested changes	5
11 Sep 2025	Thursday	Integrate check for update feature in refactored osdag	4
12 Sep 2025	Friday	Integrate check for update feature in refactored osdag	4
15 Sep 2025	Monday	Integrate check for update feature in refactored osdag	5
16 Sep 2025	Tuesday	Integrate cli tool feature in refactored osdag	4
17 Sep 2025	Wednesday	Integrate cli tool feature in refactored osdag	4
18 Sep 2025	Thursday	Integrate cli tool feature in refactored osdag	3
19 Sep 2025	Friday	Integrate import export feature in refactoed osdag	5
22 Sep 2025	Monday	Read the Modelling Techniques for Short Span Bridge Girder Analysis thesis report, ospgrillage monash univer	4
23 Sep 2025	Tuesday	Add osdag-latex-env in refactored osdag	4

24 Sep 2025	Wednesday	Make tutorial Videos for interns to learn git/github and o	4
25 Sep 2025	Thursday	Make tutorial Videos for interns to learn git/github and o	2
26 Sep 2025	Friday	Make tutorial Videos for interns to learn git/github and o	3.5
29 Sep 2025	Monday	Make tutorial Videos for interns to learn git/github and o	4
30 Sep 2025	Tuesday	Make tutorial Videos for interns to learn git/github and o	4
1 Oct 2025	Wednesday	Start reading on plugin management system	4
2 Oct 2025	Thursday	Start reading on plugin management system	4
3 Oct 2025	Friday	Start reading on plugin management system	4
6 Oct 2025	Monday	Make Purlin module as plugin	4
7 Oct 2025	Tuesday	Make Purlin module as plugin	5
8 Oct 2025	Wednesday	Make Purlin module as plugin	5
9 Oct 2025	Thursday	Make Purlin module as plugin	4
10 Oct 2025	Friday	Make Purlin module as plugin	6
13 Oct 2025	Monday	read osdag bridge and Icca plugins docuemnt	4
14 Oct 2025	Tuesday	read osdag bridge and Icca plugins docuemnt	4
15 Oct 2025	Wednesday	Create plugin managements policy	5
16 Oct 2025	Thursday	Make Flow chart of plugins management system	4
17 Oct 2025	Friday	How to create new plugins, make a development workflo	5
20 Oct 2025	Monday	Make dummy osdag plugins, make plugin manager to di	6
21 Oct 2025	Tuesday	Make dummy osdag plugins, make plugin manager to di	5
22 Oct 2025	Wednesday	Make dummy osdag plugins, make plugin manager to di	4
23 Oct 2025	Thursday	Make dummy osdag plugins, make plugin manager to di	4
24 Oct 2025	Friday	Make dummy osdag plugins, make plugin manager to di	6
27 Oct 2025	Monday	Create Plugin template classes and common logic to fac	6
28 Oct 2025	Tuesday	Create Plugin template classes and common logic to fac	5
29 Oct 2025	Wednesday	Create Plugin template classes and common logic to fac	5
30 Oct 2025	Thursday	Create Plugin template classes and common logic to fac	6
31 Oct 2025	Friday	Create Plugin template classes and common logic to fac	6
3 Nov 2025	Monday	Create new conda release for osdag with flexure membe	4
4 Nov 2025	Tuesday	Create new conda release for osdag with flexure membe	4
5 Nov 2025	Wednesday	Create new conda release for osdag with flexure membe	4
6 Nov 2025	Thursday	Create new conda release for osdag with flexure membe	4
7 Nov 2025	Friday	Add internet connection check feature in osdag	4
10 Nov 2025	Monday	Add internet connection check feature in osdag	5
11 Nov 2025	Tuesday	Add internet connection check feature in osdag	3
12 Nov 2025	Wednesday	Plugin System Logic Implement(Plugin base classes and	4
13 Nov 2025	Thursday	Plugin System Logic Implement (Plugin manager and st	3

14 Nov 2025	Friday	Plugin System Logic Implement (Plugin manager and st	3
17 Nov 2025	Monday	Plugin System Logic Implement (Plugin manager and st	4
18 Nov 2025	Tuesday	Plugin System Logic Implement (Refactored existing plu	6
19 Nov 2025	Wednesday	Plugin System Logic Implement (Enhanced user interfac	6
20 Nov 2025	Thursday	Plugin System Logic Implement(Prepared for future plug	5
21 Nov 2025	Friday	Add cross platform capability for plugin store	4
24 Nov 2025	Monday	Add plugin update feature	6
25 Nov 2025	Tuesday	Finalze the plugin system	4
26 Nov 2025	Wednesday	Make Input Dock lock state more visible	4
27 Nov 2025	Thursday	Make Input Dock lock state more visible ,Change under-c	4
28 Nov 2025	Friday	Resize HomePage icons	2
29 Nov 2025	Saturday	Comment out unneccessary print statements, Fix valida	4
30 Nov 2025	Sunday	Add input dock unlock clears output dock feature	4

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] Mehendi Hasan. Osdag plugin manager source code repository. Accessed: 2024-12-05.

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

[4] FOSSEE Project. Fossee website. Accessed: 2024-12-05.

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