Final Report

April 11, 2025

Lumberjack Balancing

Project Sponsor: Dr. Scot Raab

Project Mentor: Paul Deasy

Team Members: Riley Burke, Cristian Marrufo, Sergio Rabadan, Braden Wendt

Table of Contents

Introduction

In the evolving landscape of higher education, efficient administrative systems are vital to supporting academic excellence. One such critical task at Northern Arizona University (NAU) involves the calculation and management of faculty workloads, which is an essential responsibility carried out by associate deans like Dr. Scot Raab. Historically performed manually, this process is not only time-consuming but also susceptible to human error, especially given the volume of faculty and the complexity of workload policies. These inefficiencies can hinder transparency, burden administrative staff, and compromise equitable workload distribution.

To address these challenges, our team developed Lumberjack Balancing, which is a Python-based desktop application designed specifically to automate NAU's faculty workload calculation process. The system enables users to upload faculty assignment data and policy parameters through Excel files, which are then processed using a flexible, dynamically configurable algorithm. A clean, minimal user interface guides non-technical staff through data uploads, real-time validation, and report generation with ease. The result is a streamlined, accurate, and policy-compliant workload summary that helps associate deans reduce clerical effort and make informed staffing decisions.

This project empowers NAU to modernize its internal processes while preserving flexibility to adapt to changing institutional policies. By shifting from manual spreadsheet calculations to an automated, user-friendly platform, Lumberjack Balancing enhances operational efficiency, reduces administrative strain, and supports a more balanced academic environment across colleges. The remainder of this report details the requirements, implementation, testing, and evaluations that support this compelling solution.

Process Overview

The development of Lumberjack Balancing followed a structured and collaborative software engineering process designed to ensure consistent progress, maintain code quality, and align closely with the client's evolving needs. We adopted a modified waterfall model with iterative feedback loops to support early planning, phased development, and continuous testing. This approach provided enough structure for clear milestones while allowing for flexibility in implementation and feedback integration.

Development Lifecycle and Tools

We broke the project into distinct phases: requirements gathering, technology feasibility analysis, design, implementation, testing, and refinement. These phases were supported by various tools and engineering practices:

- **Version Control**: Git and GitHub were used for all source code and documentation versioning. Each team member created and reviewed pull requests, ensuring peer-reviewed, atomic commits to maintain a clean, readable history.
- **Task Management**: Weekly meetings and text messages were used to delegate tasks and discuss bugs.
- **Documentation and Collaboration**: Google Drive housed shared documentation, planning materials, and presentation slides.
- **Prototyping and Interface Design**: The user interface was developed using Python’s PyQt6 library, enabling quick iterations and testable prototypes that could be reviewed by our client and peers.
- **Testing and Validation**: As detailed in our Software Testing Plan, we ran unit, integration, and usability tests throughout development to verify correctness, functionality, and ease of use.

Team Roles and Responsibilities

Our team structure and role assignments were defined early in the project to streamline collaboration and support individual strengths:

- **Riley Burke** served as **Team Leader and Communications Lead**, overseeing project coordination, meeting facilitation, and final document reviews.
- **Cristian Marrufo** acted as **Architect and Release Manager**, guiding technical design decisions and maintaining consistent code structure and versioning practices.
- **Sergio Rabadan** was the **Recorder and Website Manager**, responsible for meeting documentation, project tracking, and hosting project artifacts.
- **Braden Wendt** held the role of **User Experience Lead**, ensuring the system interface was accessible and testing usability with feedback from stakeholders.

All team members contributed to core development tasks, with coding responsibilities distributed evenly across modules such as data processing, file handling, and workload computation.

Team Procedures and Collaboration Standards

We established detailed team standards early in the project to promote effective collaboration and accountability:

- Weekly meetings were held every Thursday, with Discord and group chat used for asynchronous updates and coordination.
- Decisions were made using majority rule, with emphasis on respectful debate and compromise in split scenarios.
- Peer reviews, document walkthroughs, and milestone retrospectives ensured that all voices were heard and that improvements could be made continuously throughout the semester.

These practices ensured that our team stayed organized, cohesive, and responsive to feedback, ultimately resulting in a well-documented, well-tested, and client-ready solution tailored to Northern Arizona University’s needs.

Requirements

The requirements for Lumberjack Balancing were established through a collaborative acquisition process between our development team and our client, Dr. Scot Raab, Associate Dean at Northern Arizona University (NAU). The team engaged in several meetings to understand the challenges of the existing manual workload calculation process and to identify the functional needs, user expectations, and policy constraints that the new system would need to accommodate. These discussions, along with direct analysis of example Excel data and workload policies, led to a set of well-defined functional and non-functional requirements.

Functional Requirements

The core functional requirements of the system are centered on automating the faculty workload calculation process and providing a streamlined, user-friendly experience. Key high-level capabilities include:

- **Data Import and Management**: Users must be able to upload Excel files containing faculty course assignments, teaching responsibilities, and policy data.
- **Policy Configuration**: The system must allow administrators to define and update workload policies using a customizable Excel-based conditions table.
- **Automated Workload Calculation**: Based on the uploaded data and policies, the system must compute individual faculty workloads and identify discrepancies such as overloads and underloads.
- **Report Generation and Exporting**: The system must produce summary workload reports that can be exported in Excel format for further distribution or record-keeping.

Non-Functional Requirements

In addition to the core functionality, several non-functional requirements were identified to ensure usability, performance, maintainability, and security:

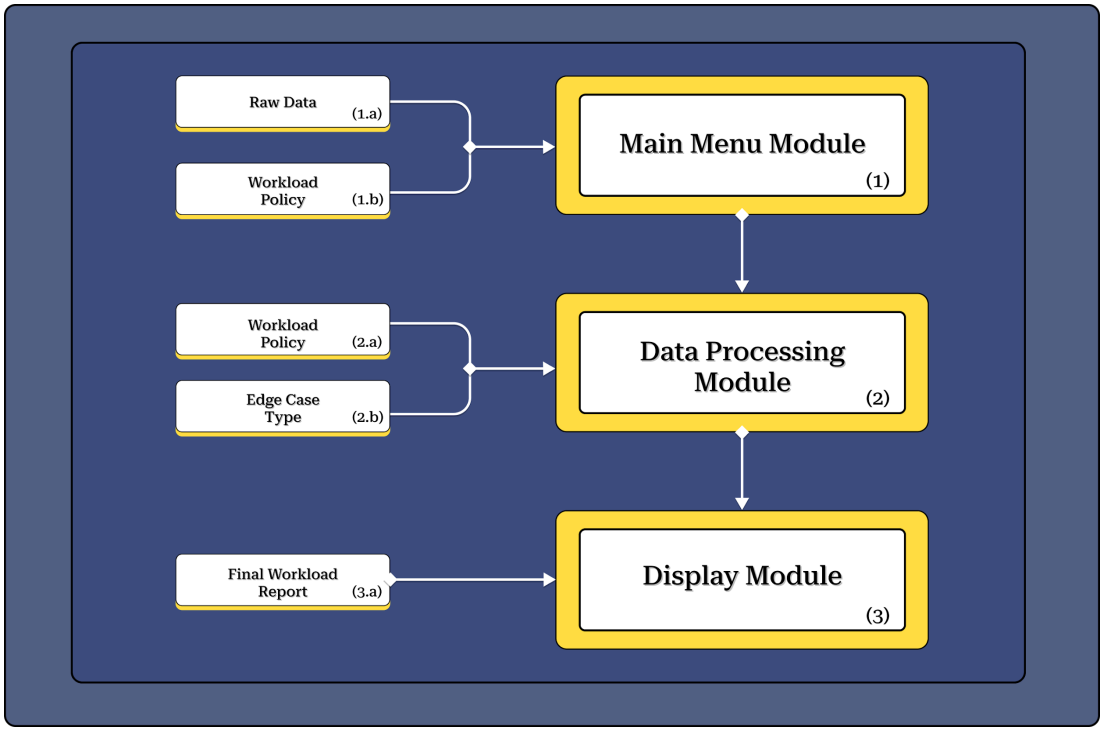
- **Usability**: The application must provide a clean, intuitive graphical user interface (GUI) suitable for non-technical administrative staff, requiring minimal training.
- **Accuracy**: The system must maintain a workload calculation error rate of less than 0.1% to comply with NAU's policy standards.
- **Performance**: It must process large Excel datasets (up to 2000 faculty entries) in under one minute to preserve efficiency during peak usage periods.
- **Flexibility and Maintainability**: The workload calculation algorithm must be driven by external configuration (Excel), allowing policy changes without modifying the source code.
- **Security and Data Privacy**: No input data should be stored locally or transmitted externally. All processing must occur on the client's machine to protect sensitive faculty information.

These requirements shaped the design and implementation of ***Lumberjack Balancing***, ensuring that the delivered product directly addressed the inefficiencies of NAU's current manual process while offering a scalable and user-friendly solution for future semesters.

Architecture and Implementation

1. Architectural Overview

   Lumberjack Balancing employs a modular, three-layered software architecture with clearly delineated responsibilities, depicted in the following high-level architectural diagram:



Component Responsibilities & Interactions:

1. **Main Menu User Interface (PyQt GUI - main.py):** Provides intuitive graphical interface for selecting input files and initiating workload calculations.
2. **Processing & Logic Layer (excel_processor.py, algorithmPolicy.py):** Core computation logic: data validation, course grouping (team-taught/co-convened), workload calculations, and policy compliance.

3. **Data Access Layer (Excel files via Pandas):** Reads, cleans, validates, and writes Excel files containing raw course data, workload policies, instructor tracks, and final workload outputs.

Information Flow:

- **GUI to Processing Layer:** File paths chosen by users are passed down to `excel_processor.py`.
- **Processing Layer to Data Access Layer:** The processor uses pandas to load Excel files. Data is cleaned, filtered, and structured into internal Python objects.
- **Data Access Layer to Processing Layer:** Processed and structured data (Course & FacultyMember objects) returned to the processing layer, where business logic is executed (grouping, calculation, etc.).
- **Processing Layer to GUI & Output Excel:** Computed results are saved back into Excel output files. GUI displays status and provides user feedback.

Example Use-Case Flow:

- User launches Lumberjack Balancing GUI (`main.exe`).
- User selects input Excel files (raw data, policy, special courses, instructor track).
- Clicks “Run Workload Calculation”:
    1. Excel Processor loads data, eliminates invalid or duplicate rows.
    2. Course objects created, grouped by shared attributes for team-taught or co-convened detection.
    3. Policy rules applied: WLUs calculated, caps enforced, workload adjusted.
    4. Results compiled and saved as Excel summary files.
- User reviews generated Excel reports for faculty workload details.

Architectural Style & Influences:

- **Modularity & Layering:** Clear separation of concerns, enabling independent modifications or testing of components.
- **Event-driven GUI:** PyQt GUI operates independently, passing events (file selection, run analysis) to backend logic modules.
- **Data-centric Design:** Heavy use of structured data manipulation with Pandas, facilitating scalability and clarity.

2. Detailed Component Implementation

a.1. User Interface Layer Role (**PyQt - main.py**):

- Offers intuitive GUI enabling file selection and starting workload analysis.
- Abstracts complexity, providing a clear interface to less technical end-users.

a.2. User Interface Layer Key Details (**PyQt - main.py**):

- Built using PyQt6 library.
- Minimalist design to reduce user errors.
- Directly calls methods in `excel_processor.py`.

b.1. Processing & Logic Layer Role (**excel_processor.py**):

- Coordinates overall processing.
- Loads and preprocesses Excel data.
- Orchestrates calls to policy calculation logic.
- Generates final Excel outputs.

b.2. Processing & Logic Layer Core functions (**excel_processor.py**):

- File loading (`pandas.read_excel`).
- Data validation and deduplication.
- Course and FacultyMember object instantiation.
- Calling workload calculation methods and adjustments.

b.3. Processing & Logic Layer Role (**algorithmPolicy.py**):

- Encapsulates workload calculation logic and policy rules.
- Defines Course and FacultyMember classes.
- Implements co-convened and team-taught logic, WLU calculations, and capping rules.

b.4. Processing & Logic Layer Core functions (**algorithmPolicy.py**):

- `Course`: Encapsulates attributes like enrollments, units, instructors, and methods for calculating load.
- `FacultyMember`: Tracks courses taught by a faculty member, computes total WLU.
- Methods such as:
  - `adjust_co_convened()`: collapses co-convened courses.
  - `getGroupKeyForGrouping()`: identifies team-taught courses.
  - `calculateLoad()`: computes WLUs per policy.

c.1. Data Access Layer Role (Excel via Pandas/OpenPyXL):

- Interacts directly with Excel files.
- Ensures robust data handling and consistency.
- Supports structured and efficient reading and writing of data.

c.2. Data Access Layer Key Operations

- Reads Excel files (`pandas`).
- Writes summary reports (`openpyxl` or `xlsxwriter`).
- Enforces consistency in data structure and format.

3. Differences: As-Planned vs. As-Built

During implementation, several practical deviations occurred from the initial design, the following table illustrates the most significant changes:

Difference	Explanation & Reason
Expanded Instructor Inclusion	Originally only primary instructors (PIs) were considered; now all faculty roles are processed to meet broader institutional requirements.
Switching GUI Toolkit (Tkinter to PyQt6)	Originally planned with Tkinter for simplicity, but switched to PyQt6 for richer functionality and improved end-user experience.
Team-taught/Co-convened Logic Refinement	Original grouping logic was overly simplistic. Added additional course attributes (e.g., instructor IDs, class numbers) to keys to accurately separate team-taught from co-convened scenarios.
Simplified User Workflow	Streamlined the GUI interaction flow based on user feedback, reducing required steps for daily operation from original plans.

Features Not Implemented:

- **User Accounts & Admin Profiles:** Originally planned but removed due to limited user base and the introduction of unnecessary complexity. The application remains single-user-oriented without login requirements.
- **Database Backend:** Initially planned to use a database backend, but settled on Excel files to simplify deployment and user familiarity.

Overall, the Lumberjack Balancing architecture achieves clear modularity, maintainability, and usability goals, despite some minor but beneficial deviations from the initial design. The implemented solution provides a robust foundation for future expansion and straightforward maintenance. Given the architecture documentation, a new software engineer should now have sufficient knowledge to efficiently contribute to ongoing system development and improvement.

Testing

A robust testing strategy was critical to ensuring that Lumberjack Balancing delivered accurate, reliable, and user-friendly functionality to support the needs of Northern Arizona University's associate deans. Our approach to testing was multi-layered, covering unit, integration, and usability testing to evaluate both the correctness of the codebase and the quality of the user experience. We focused not only on verifying the internal logic of the application but also on ensuring that the system works cohesively across modules and is accessible and intuitive to our intended users.

Testing Strategy & Activities

We divided our testing process into three main categories:

Unit Testing

Unit testing focused on the core components of the application, especially the data processing modules that parse faculty information, apply workload policies, and calculate final teaching loads. We used the unittest and pytest libraries in Python to implement tests across a wide range of equivalence partitions and boundary conditions. This included testing for edge cases such as zero enrollment, missing data fields, and unusual course types like research or co-convened classes. We also verified individual class methods like calculateLoad(), adjustLoadDivision(), and calculatePercentage() to ensure reliable, rule-compliant output.

Integration Testing

We conducted integration testing to verify that the modules within the system—file loading, policy configuration, workload calculation, and report generation—functioned correctly when combined. These tests ensured that data passed between modules accurately and that the application could handle complete workflows from input to report export without error. Key integration scenarios included loading malformed Excel files, verifying error detection and

messaging, and ensuring calculated results matched expected output from known datasets.

Usability Testing

Given that our target users are associate deans and administrative staff—many of whom do not have technical backgrounds—usability was a central concern. We conducted guided walkthroughs and acceptance testing with our client, Dr. Scot Raab, to validate that users could intuitively perform all required actions, such as uploading data, initiating calculations, and reviewing reports. Feedback from these sessions led to refinements in UI layout, clearer error messages, and tooltips to reduce confusion and support independent use.

Results & Design Revisions

Testing surfaced several critical insights that directly informed improvements to our system:

- Edge Case Adjustments: Testing exposed calculation inconsistencies for research courses missing certain scheduling data. This led us to refine our row validation logic and customize rules for these course types.
- UI Enhancements: During usability testing, users requested more descriptive labels and status feedback. In response, we added status indicators for upload and calculation phases and improved the labeling of buttons and fields to better guide the user through the workflow.
- Algorithm Fixes: Acceptance testing with Dr. Raab highlighted discrepancies between our calculator totals and his. We discovered that we had our math set up much more complicated than was necessary. We removed the use of linear interpolation and instead utilized the maximums provided in the policy document.

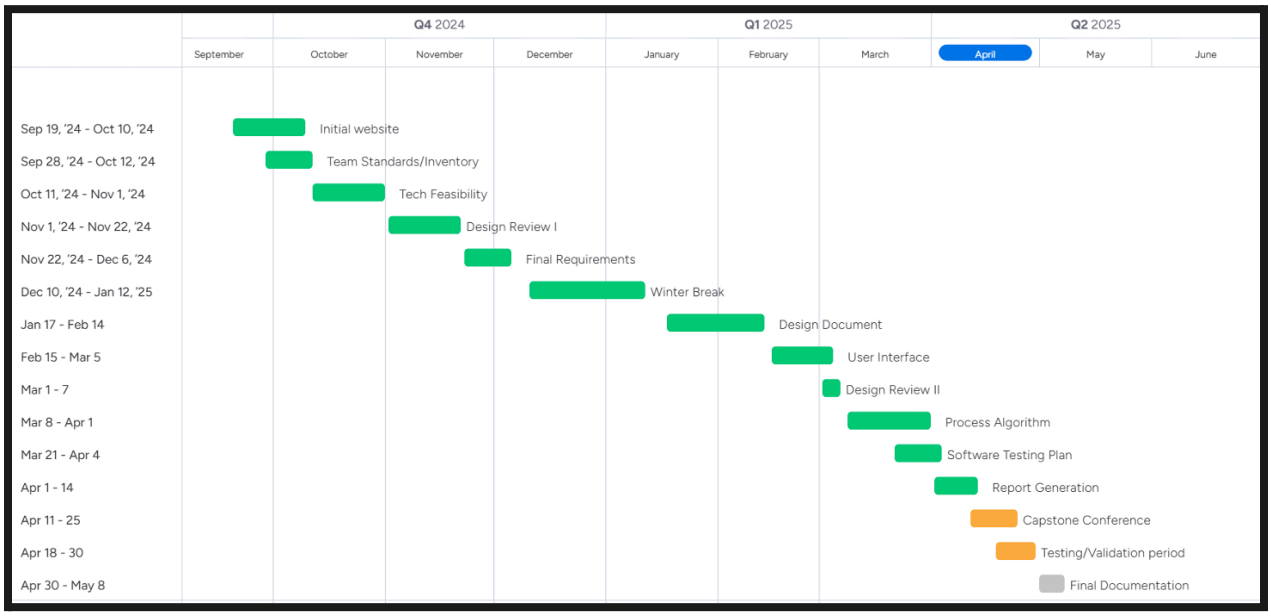
Final Thoughts

Through this rigorous and multi-dimensional testing process, we verified that Lumberjack Balancing is a reliable, accurate, and user-friendly tool for automating faculty workload calculations. Each testing phase contributed to a more stable and polished final product. The iterative nature of our testing approach, particularly through

continuous client feedback and usability observations, allowed us to deliver a solution that not only meets its functional requirements but is also tailored to the real-world needs of its users.

Project Timeline

Now that we know what your team produced, give a quick review of the overall project timeline. You could do this as a Gantt chart...or as a sequential list of milestones. In either case, briefly walk through the schedule you present, describing the key phases as well as any interesting factoids (how work was divided up in the team, who led on what, etc.) related to individual phases. Be sure to say where you are at this moment in the schedule you have outlined.



At the beginning of the semester, we started by building the initial website and setting up our team standards and tools. After that, we worked on tech feasibility to make sure everything we planned was actually doable. Then, we had our first design review, which helped us organize our ideas better.

Once we got through that, we gathered our final requirements before winter break and came back in January to start working on the design document.

In February, Braden worked on the user interface, which came out looking clean and easy to use. We had another design review in early March to check in on our progress.

While that was going on, Cristian focused on building the process algorithm, and Sergio helped a little bit there. Then, Sergio took the lead on report generation, which pulls everything together into a final Excel report. Around this time, we also made our software testing plan to figure out how we would check for bugs.

Riley helped out with everything throughout the project.

Right now (end of April), we’re finishing up our testing and validation. After that, we’ll wrap up with our final documentation.

Future Work

As development of Lumberjack Balancing progressed, several promising opportunities emerged for expanding the system's capabilities in future iterations. While our current version successfully automates the core faculty workload calculation process, feedback from our client and our own reflections during implementation revealed valuable directions for continued development.

- One potential enhancement is the addition of **predictive analytics and data visualization tools**. These features could help associate deans anticipate future staffing needs, identify workload trends across semesters, and make data-driven decisions about faculty allocation. Visual dashboards showing workload distributions by department or role would further support strategic planning.

- Another recommended feature is **multi-user support with role-based access control**. Currently, the system is designed for single-user, local operation. Enabling secure access for multiple users—such as department chairs or administrative assistants—could improve collaboration and workflow distribution while preserving data integrity.

- We also recognize the value of developing **policy editing tools directly within the application**. While our current approach uses Excel for workload condition configuration, a built-in policy editor with form validation would improve usability and reduce errors, especially for users less comfortable with spreadsheet tools.

- Lastly, a few lower-priority features outlined early in the project were deprioritized to focus on core functionality. These include export options in formats beyond Excel (such as PDF) and automated email distribution of reports. These features remain viable candidates for future updates as the system matures.

Altogether, these ideas point toward a more powerful and collaborative platform in version 2.0, one that continues to align with NAU's evolving administrative needs and strengthens the long-term value of the system.

Conclusion

The Lumberjack Balancing project represents a significant step forward in modernizing Northern Arizona University’s faculty workload management process. By automating a task that was traditionally manual, time-intensive, and prone to human error, our system provides a more accurate, efficient, and user-friendly solution tailored specifically to the needs of NAU's associate deans and administrative staff. Through a careful combination of structured development, rigorous testing, client feedback, and continuous refinement, we delivered a system that meets both functional and usability requirements while maintaining flexibility for future policy changes. Our modular architecture, comprehensive testing strategy, and focus on ease of use ensure that Lumberjack Balancing will not only address the university’s immediate workload calculation challenges but will also serve as a sustainable foundation for future enhancements. Additionally, the lessons learned throughout development—such as adapting algorithms to user expectations, refining UI design based on user preference, and emphasizing modularity—position the system for continued growth. Looking ahead, there are exciting opportunities to expand Lumberjack Balancing’s capabilities through data visualization, multi-user collaboration features, and integrated policy editing. With these future enhancements, the system could evolve into a comprehensive administrative tool that supports strategic decision-making across departments. Ultimately, Lumberjack Balancing demonstrates how thoughtful software design, close client collaboration, and adaptive engineering practices can transform institutional processes for the better. We are confident that the solution will deliver lasting value to NAU and provide a scalable model for similar administrative challenges in higher education.

Glossary

Domain Specific Terms

Term	Definition
Workload Unit (WLU)	A numeric measure representing faculty teaching load, calculated from course attributes (enrollment, units, type) and university policy rules.
Course Object	A Python data structure encapsulating individual course details (e.g., units, enrollment, instructor, schedule), used in WLU calculations.
Faculty Member Object	Python structure aggregating all courses taught by a single instructor, calculating their total workload units (WLUs).
Team-Taught Courses	Courses taught simultaneously by multiple instructors; workload is evenly divided among participating instructors.
Co-Convened Courses	Distinct course sections (e.g., undergrad & grad) taught simultaneously by the same instructor. These are identified and collapsed into one course for workload calculations.
Base Rate	Multiplicative factor from workload policies applied per course type and instructor category (Tenure Track, Career Contract, etc.) for WLU calculation.
Cap	A maximum workload limit assigned by policy per course category (e.g., lecture, independent study, thesis).
Instructor Track	Instructor category influencing workload calculations (e.g., TT for Tenure Track, CC for Career Contract).
Policy File	An external Excel file containing institutional rules for calculating workload, including base rates and caps.
Special Assignments File	An Excel file specifying additional workload assignments or overrides for special cases.
Group Key	Composite identifier (instructor ID, schedule, class number, etc.) used to detect team-taught or co-convened courses.

Appendix A: Development Environment and Toolchain

This appendix guides new developers in setting up their development environment for the Faculty Workload Calculation Tool (Lumberjack Balancing), ensuring a smooth onboarding and effective development experience.

Hardware Environment

Development Platform:

- Primarily developed and tested on Windows 10/11.
- Compatible with Linux and MacOS (with minor adjustments).

Development Machine Specs Used:

- Processor: AMD Ryzen 9 or Intel Core i9.
- Memory: 16GB RAM recommended (8GB minimum).
- Storage: SSD recommended (for faster I/O with Excel data).

Minimum Requirements:

- Any modern PC/Mac with 8GB+RAM, Python 3.10+, and SSD preferred.

Software Toolchain

Tool	Description	Importance to Project
Python (3.10+)	Core programming language.	Essential language used for writing logic, calculations, and file handling.
Pandas	Data manipulation and analysis library.	Vital for reading, processing, and cleaning Excel data files efficiently.
OpenPyXL/XlsxWriter	Libraries for Excel file generation.	Required to output structured Excel summary files from Python.
PyQt6	GUI framework.	Provides a modern graphical interface for selecting input files and triggering workload computations.
Visual Studios Code	Code editor/IDE.	Provided efficient coding, debugging, and syntax highlighting during development.

PyInstaller	Build tool.	Used to package Python scripts into standalone executable files for easy distribution and deployment.
Git	Version control system.	Enabled organized collaboration, code tracking, backups, and rollbacks during project lifecycle.

Step-by-Step Setup Instructions

Follow these exact steps to configure a new machine for immediate development:

Step 1: Install Python

- Go to python.org and download Python (version 3.10+ recommended).
- During installation, select “Add Python to PATH”.

Step 2: Install Project Dependencies

- Open Command Prompt (Windows) or Terminal (Mac/Linux), and run:
  ```
  > pip install pandas openpyxl xlsxwriter pyqt5 pyinstaller
  ```

Step 3: Clone or Copy Project Files

- Place all files into a clearly named project folder (Lumberjack Balancing):
  ```
  LumberJack Balancing/
  ├── algorithmPolicy.py
  ├── excel_processor.py
  ├── main.py
  ├── workload_policy.xlsx
  ├── raw_data.xlsx
  ├── special_courses.xlsx
  ├── instructors_track.xlsx
  ```

Step 4: (Recommended) Install VS Code

- Download and install VS Code.
- Open the project folder within VS Code.
- Install the official Python extension through VS Code’s extension marketplace for enhanced coding support.

Production Cycle (Edit - Compile - Deploy)

1. **Make a Code Edit**
   For example, changing the GUI text:
   - Open main.py using VS Code.
   - Change a GUI element’s text (e.g., window title).

Original Code:
> `self.setWindowTitle("Faculty Workload Calculator")`
Modified Code:
> `self.setWindowTitle("Lumberjack Balancing - Workload Tool")`

2. **Test the Changes**
   - Run the application directly from the terminal for rapid iteration:
     > `python main.py`
   - Verify the change visually in the launched PyQt GUI window.

3. **Compile to Executable for Distribution**
   Once satisfied with the edit:
   - Compile the updated Python scripts into a new executable:
     > `pyinstaller --onefile --windowed main.py`
   - This generates a standalone executable at:
     `/dist/main.exe`

4. **Distribute the Executable**
   - Distribute main.exe to end-users who do not need Python installed.
   - Users simply run the executable, select their input files, and get the workload summaries directly.

5. **Use Version Control (Git)**
   - After successfully testing and compiling, commit your changes to github repository if available:
     > `git add .`
     > `git commit -m "Updated GUI title"`
     > `git push`