
This is the **published version** of the bachelor thesis:

García i Girona, Jan. *Design and implementation of a software interface for real-time control and analysis of multiplexed biosensors for clinical diagnostics.*

Treball de Final de Grau (Universitat Autònoma de Barcelona), 2026
(Enginyeria Informàtica)

This version is available at <https://ddd.uab.cat/record/334367>

under the terms of the  license.

Design and implementation of a software interface for real-time control and analysis of multiplexed biosensors for clinical diagnostics

Jan García i Girona

June 29, 2026

Resum– Aquest treball presenta el disseny i la implementació d'una interfície software basada en Python per controlar i analitzar una plataforma de biosensors fotònics orientada al diagnòstic clínic. El projecte desenvolupa una aplicació pròpia per al flux de treball del laboratori, amb una estructura modular clara, dependències actualitzades i funcionalitats addicionals per a l'adquisició, el processament i la visualització de dades. El treball segueix una metodologia d'enginyeria del programari basada en l'anàlisi arquitectònica, la refactorització progressiva i la integració de nous components per donar suport a l'operació en temps real i a futures ampliacions. El resultat és una aplicació completa i mantenible, compatible amb el maquinari de laboratori del grup NanoB2A de l'ICN2 i preparada per a continuar evolucionant.

Paraules clau– Biosensors fotònics, diagnòstic clínic, Python, enginyeria del programari, arquitectura modular, operació en temps real, visualització de dades.

Abstract– This project presents the design and implementation of a Python-based software interface for controlling and analysing a photonic biosensor platform for clinical diagnostics. The project develops a dedicated application for the laboratory workflow, with a clear modular structure, modern dependencies, and additional functionality for data acquisition, processing, and visualisation. The work follows a software engineering methodology based on architectural analysis, progressive refactoring, and the integration of new components to support both real-time operation and future extension. The resulting system is a complete and maintainable application that is compatible with the NanoB2A laboratory hardware at ICN2 and prepared for further development.

Keywords– Photonic biosensors, Clinical diagnostics, Python, Software engineering, Modular architecture, Real-time operation, Data visualisation.



1 INTRODUCTION – CONTEXT OF THE WORK

1.1 Clinical diagnostics and point-of-care motivation

C LINICAL diagnostics is still largely organised around centralised laboratory workflows, in which a sample is collected in one location, transported to a specialised facility, processed with complex instrumentation, and analyzed hours or days later as illustrated in Figure 1. This model has proven to be robust and accurate, but it also delays results, is infrastructure-dependent and

-
- Contact e-mail: 1574727@uab.cat
 - Mention: Software Engineering
 - Supervised by: Joan Protasio (Department of Computer Science)
 - Academic year 2025/26

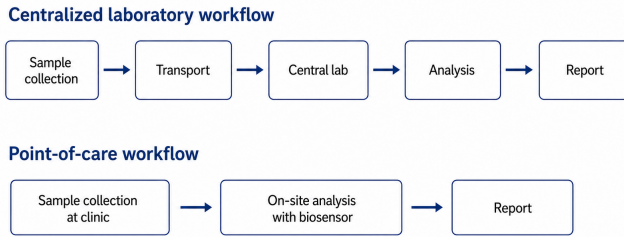


Fig. 1: Centralised laboratory workflow versus point-of-care diagnostic workflow.

difficult to adapt to scenarios where rapid decision making is essential, such as primary care, emergency medicine or resource-limited settings.[1][2]

The concept of *decentralised* or *point-of-care* (POC) diagnostics aims to shorten this delay by moving the analytical capability closer to the patient[1]. Instead of relying exclusively on large laboratory analysers, the goal is to deploy compact devices that can deliver clinically useful results in minutes using only a small biological sample (for example blood, serum, urine or saliva) and minimal operator intervention.[2] In this context, biosensor technologies have emerged as key enablers for the next generation of diagnostic systems.[1][3]

1.2 Biosensors and photonic platforms

In broad terms, a biosensor is a device that combines a biological recognition element with a physical measurement component. The main elements are:

- A *recognition element*, such as an antibody or a short DNA/RNA sequence, which is engineered to selectively capture a specific substance from the sample.
- The captured substance, known as the *analyte*, which in medical applications is often also a *biomarker*, i.e., a measurable biological signal associated with a disease or physiological state.
- A *transducer*, the part of the device that converts the biochemical binding event into a quantitative signal that can be read and processed (for example an electrical or optical change).

Depending on the nature of the transducer, biosensors can be classified into several families. Common examples include:

- *Electrochemical* biosensors, where binding events produce changes in current, potential or impedance at an electrode surface.
- *Mass-sensitive* biosensors, such as quartz crystal microbalances or surface acoustic wave devices, which detect changes in mass or viscoelastic properties via shifts in resonance.

- *Optical* biosensors, where the signal is encoded in changes of light intensity, phase, wavelength or polarisation; this family includes fluorescence-based assays, surface plasmon resonance and integrated photonic biosensors.

This work focuses on a particular class of optical biosensors known as *photonic* biosensors, in which light guided on a chip is used directly as the sensing mechanism.[2][3][4] In these devices, light is confined to a small structure called an *optical waveguide*, which can be thought of as the on-chip analogue of an optical fibre. The waveguide core is made of a dielectric material (for example silicon nitride) with a refractive index higher than its surroundings, and it is embedded in a lower-index environment formed by the transparent substrate (typically silicon dioxide) and the liquid sample above.[2][3]

Most of the light remains confined inside the high-index core and travels along a well-defined path, but a small fraction of the electromagnetic field extends a short distance into the lower-index surrounding medium as an evanescent field.[2][4] When analyte molecules bind to the recognition layer at the sensor surface in this region, they slightly modify the local refractive index, and this produces a measurable change in the light that exits the device, such as a variation in phase, intensity or wavelength.[2][4][5]

A key advantage of these photonic biosensors is that they can operate in a *label-free* manner: the device can directly monitor the binding of the analyte to the recognition layer by observing how the light changes, without needing to attach any extra fluorescent or colour-producing molecules to the sample beforehand.[5] Compared with many traditional laboratory techniques, this:

- Reduces the number of additional chemical substances (known as *reagents*, i.e., chemicals added to trigger or reveal a reaction) that must be prepared and handled during the assay.
- Simplifies the workflow and shortens the total analysis time.
- Facilitates continuous or real-time monitoring of molecular interactions.

[3][5]

From an implementation point of view, many photonic biosensors are built using *dielectric* materials, that is, insulating optical materials such as glass or silicon-based materials that guide light efficiently while not conducting electricity. Integrated waveguides fabricated in these dielectrics form the basis of a wide range of sensor architectures. Among them, platforms based on *interferometric* read-out, that is, read-out based on the interference between two optical signals or modes, have demonstrated particularly high sensitivity for detecting small refractive-index changes at the sensor surface.[3][4][5]

1.3 State of the art in laboratory instrumentation software interfaces

In the domain of scientific instrumentation and laboratory automation, software control layers have traditionally relied on commercial proprietary suites, most notably

National Instruments LabVIEW. While LabVIEW offers graphical rapid prototyping and deep ecosystem integration with hardware components such as data acquisition (DAQ) cards, its file-based development model can complicate version control and long-term maintenance, particularly in collaborative projects where changes must be tracked and merged over time.[10]

In addition, LabVIEW applications often depend on proprietary licensing and deployment mechanisms, which may increase distribution complexity for research groups and laboratories that need flexible cross-platform workflows.[11]

To establish more sustainable software architectures, current practice increasingly combines open-source ecosystems with high-level graphical frameworks. In this context, Python-based desktop applications represent a strong alternative for data collection and signal analysis, especially when paired with GUI toolkits designed for interactive and responsive user interfaces. Native toolkits such as `Tkinter` remain useful for lightweight applications, but they provide a more limited widget ecosystem than Qt-based frameworks. By contrast, web-based visualization stacks such as Plotly Dash or Streamlit are effective for dashboards and lower-frequency interaction patterns, although they are less directly suited to tightly coupled real-time acquisition loops.

Consequently, native client architectures built with `PyQt6` or `PySide6` (the Python bindings for Qt6) provide an appropriate design path for this project.[7] They support a signal-and-slot communication model and thread-safe background processing, which makes it possible to separate high-frequency DAQ routines from the main graphical execution thread while keeping the interface responsive.[7][8]

1.4 BiMW platform and software engineering objectives

Among integrated photonic biosensors, the *bimodal waveguide* (BiMW) interferometric platform developed at ICN2 is a particularly relevant example for this project.[4][5] In a BiMW sensor, a single waveguide is designed to support two optical modes with the same polarisation; small refractive-index changes near the sensor surface alter the relative phase between these modes, and the resulting interference pattern can be used for highly sensitive label-free detection.[4][5] This sensing principle has been demonstrated experimentally in earlier work and has also been extended with new read-out strategies for improved linearity and signal interpretation.[4][5]

The current ICN2 platform is based on this BiMW principle and is used in the laboratory for calibration and measurement of interferometric signals.[4][5] At a high level, the experimental setup comprises an integrated BiMW sensor chip with microfluidics for sample delivery, an optical subsystem with the laser source, coupling optics and photodetectors, and a control computer that drives the hardware and acquires the signals.[4][5][6] The software must therefore coordinate the complete experimental workflow: initialise the hardware, configure the acquisition parameters, perform calibration, record the measurement and process the resulting data.

The existing control application was developed incrementally over time, mixing hardware control, data acquisition, signal processing, visualisation and data persistence in a single code base.[4][5] This has made the system difficult to maintain, to extend and to transfer to new users. From a software engineering perspective, the main problem is not the scientific principle of the BiMW sensor, but the way in which the experimental workflow is implemented and supported by the control software.

This *Treball de Fi de Grau* (TFG, Final Degree Project) addresses that limitation by redesigning and reimplementing the software interface for the BiMW multiplexed biosensing platform.[4][5][6][7] The objective is to produce a cleaner and more modular application that preserves compatibility with the existing experimental setup while improving usability, maintainability and support for future extensions.

In the context of the 2025/26 academic year, the scope of this TFG is limited to the redesign and reimplementation of the control software for the existing BiMW platform, including the migration to a modular MVC-style architecture, the addition of the *Intensity check* and *Processing* tabs, and the packaging of the application as a standalone executable for the NanoB2A laboratory workstation.[6][7][8][9] More advanced extensions, such as support for additional hardware configurations, integration of auxiliary analysis routines, or adaptation of the interface to other photonic platforms, are considered future work beyond the scope of this TFG.

2 EXPERIMENTAL HARDWARE AND MATHEMATICAL CONTEXT

2.1 Overview of the experimental platform

The experiments in this project are performed on a BiMW photonic biosensing platform controlled from a laboratory computer running the custom software developed in this TFG. The computer communicates with the instrumentation through a National Instruments (NI) USB data acquisition (DAQ) card, which provides analogue input and output channels as well as digital I/O lines. In general terms, a DAQ system measures electrical signals (for example voltages or currents) that encode physical quantities such as light intensity, temperature or pressure, and makes them available to a PC where they can be processed by programmable software.

In the present setup, the NI DAQ card plays a central role in both directions of the measurement chain:

- It generates the analogue control waveform that drives the laser diode through its current driver, including the sinusoidal modulation required by the interferometric read-out algorithm.
- It simultaneously acquires the analogue voltages corresponding to the light intensities measured by the photodetector array, which are then processed in real time by the Python application.

During the development phase of the software, the optical alignment between the laser and the photodetector was deliberately simplified. Instead of coupling the light through

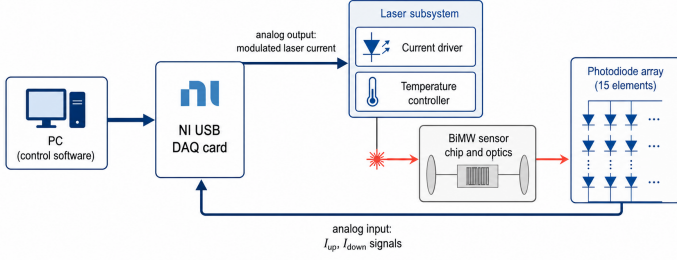


Fig. 2: Simplified hardware architecture of the BiMW experimental platform, showing the PC, NI data acquisition card, laser subsystem and photodetector array.

the full BiMW chip and imaging system, basic functional tests were carried out by covering the photodiode or illuminating it directly with a smartphone flashlight to verify that the acquisition, plotting and basic processing responded as expected. The complete hardware setup, including the final optical coupling, has been designed and validated within the NanoB2A group, and the software presented in this work is intended to replace the legacy control application previously used with this platform.

A simplified block diagram of the experimental platform is shown in Figure 2 (Section 5.3 presents the corresponding software architecture design that coordinates these components), highlighting the main elements and signal flows between the PC, the NI DAQ card, the laser subsystem and the photodetector.

2.2 Optical path and photodetector array

The photodetector used in this work consists of a linear array of fifteen small active areas (“dents”) arranged along the vertical direction. The optical system is configured so that the output line from the BiMW sensor is divided into fifteen corresponding light spots, one per detector element. One of these spots is reserved as a reference, while the remaining fourteen are organised into seven pairs, each pair forming an upper and a lower element for a given channel. This configuration enables multiplexed measurements: in a single experiment the system can monitor seven independent sensing positions on the chip.

The analogue voltages generated by each photodiode element are routed to separate analogue-input channels of the NI DAQ card. For a given sensing channel k , the software therefore receives two time series:

- $I_{\text{up},k}(t)$: the voltage proportional to the light intensity on the upper detector element of channel k .
- $I_{\text{down},k}(t)$: the voltage proportional to the light intensity on the lower detector element of channel k .

These signals constitute the raw data from which the biosensor response is computed.

2.3 Sensitivity parameter and interferometric signal

Following the BiMW literature, the basic observable used in this platform is a normalised interferometric signal known as the *sensitivity parameter* SR . [4][5] For each channel k and at each time instant t , it is defined as

$$SR_k(t) = \frac{I_{\text{up},k}(t) - I_{\text{down},k}(t)}{I_{\text{up},k}(t) + I_{\text{down},k}(t)}.$$

In the absence of any external modulation, and assuming stable coupling conditions, this signal can be expressed as

$$SR_k(t) = A_k + B_k \cos \varphi_k(t),$$

where A_k is a constant offset, B_k is the contrast or modulation amplitude, and $\varphi_k(t)$ is the intermodal phase difference between the two guided modes at the output of the BiMW waveguide. [4][5] Changes in the refractive index at the sensing window (for example due to biomolecular binding) translate into small changes in $\varphi_k(t)$, which in turn modify the value of $SR_k(t)$. [4][5]

To improve robustness and avoid ambiguities associated with the periodic nature of the cosine function, the BiMW platform operates under an all-optical sinusoidal modulation regime. [5] In practice, the laser current is modulated at a frequency of 10 Hz over a range between 130 mA and 170 mA, which induces a controlled wavelength change and hence a phase modulation. Under these conditions, the sensitivity parameter becomes

$$SR_k(t) = A_k + B_k \cos(\Delta\varphi_k + \varphi_{0,k} + \mu \sin(\omega t)),$$

where $\Delta\varphi_k$ is the phase change related to the analyte, $\varphi_{0,k}$ is the baseline phase in running buffer, μ is the modulation depth and $\omega = 2\pi \cdot 10 \text{ Hz}$ is the modulation angular frequency. [5]

Before applying the phase-retrieval algorithm, the software removes the offset A_k by dynamically centring the signal around zero. This is done by computing a centred version $SR'_k(t)$ as

$$SR'_k(t) = SR_k(t) - \frac{\max(SR_k) + \min(SR_k)}{2},$$

$$SR'_k(t) = B_k \cos(\Delta\varphi_k + \varphi_{0,k} + \mu \sin(\omega t)),$$

which enforces the mathematical conditions required for the subsequent processing. [5]

2.3.1 Trigonometric Phase-Retrieval Implementation

The detailed trigonometric phase-retrieval algorithm, based on evaluating $SR'_k(t)$ at two specific time instants within each modulation cycle and computing the phase shift from their ratio, yields a linear estimate of $\Delta\varphi_k(t)$ for each channel, which is the quantity later converted into refractive-index changes or biomarker concentrations through calibration. [5]

2.4 Signals visualised in the software

To aid both instrument control and data analysis, the software developed in this TFG provides real-time plots of several key signals for each sensing channel:

- The sinusoidal laser current waveform sent to the driver via the NI DAQ analogue output, showing that the modulation amplitude and frequency are within the required operating range.
- The raw photodetector traces $I_{\text{up},k}(t)$ and $I_{\text{down},k}(t)$, which allow the user to verify correct optical alignment and detect saturation or coupling problems.
- The normalised sensitivity parameter $SR_k(t)$ and its centred version $SR'_k(t)$, in which the interferometric response and its periodic modulation become clearly visible.
- The unwrapped phase $\Delta\varphi_k(t)$ obtained from the trigonometric phase-retrieval algorithm, which is the main quantity used to build calibration curves and biosensing sensorgrams.

Representative screenshots of these plots, captured from the final version of the control software, are included in the Appendix (see Figures 7, 8, 9, and 10) to provide a visual example of the real-time behaviour of the system.

3 OBJECTIVES

3.1 General objective

The main objective of this Final Degree Project is to redesign and extend the control software of a BiMW photonic biosensing platform so that it becomes a maintainable, modular and evolvable tool. The new application must follow good software engineering practices, remain compatible with the existing National Instruments data acquisition hardware used in the laboratory environment, simplify day-to-day operation for researchers, and correct the functional issues and bugs accumulated over years of incremental development on an obsolete Python code base.

3.2 Specific objectives

The general objective can be decomposed into the following specific objectives.

O1. Analysis and restructuring of the legacy code. The first objective is to analyse the current application and derive a clean architecture for the new project, based on a Model–View–Controller (MVC) pattern. In the legacy software, almost all the logic is concentrated in a single file, `Controller.py`, with several thousand lines of code that initialise global variables and call all back-end methods. This monolithic design hinders understanding, testing and extension.

The new project adopts an MVC-style architecture in which the responsibilities are clearly separated: controllers orchestrate the acquisition and processing workflows, models encapsulate the data and domain logic, and views implement the graphical user interface. Hardware-specific functions related to the NI DAQ board, its drivers and vendor

libraries are isolated in dedicated modules. The final directory structure and module decomposition are documented in the Appendix (see Appendix A.1), so that future developers can quickly understand where each concern is implemented.

O2. Development of a new, up-to-date project. The second objective is to rebuild the application on a modern and supported technology stack. The original software targets Python 3.9, which no longer receives security updates, and relies on the PyQt5 graphical library, officially supported only up to Python 3.10. This combination results in a project that is both obsolete and difficult to maintain.

The new application is implemented in Python 3.12, a currently supported version that remains compatible with the National Instruments drivers required by the USB DAQ card. On the user-interface side, the project migrates from PyQt5 to PyQt6 [7, 8], enabling a more responsive and future-proof front end. From the outset, the project is prepared for distribution via `PyInstaller`, so that the researcher can run a self-contained executable instead of having to open the project in an IDE and manage a virtual environment each time.

An additional practical constraint encountered during the development was the network policy at ICN2, which does not allow direct installation of all required packages via standard terminal commands. As a workaround, the necessary wheels and archives were downloaded one by one from the official Python Package Index (PyPI) [9] repository and installed locally from a folder bundled with the project.

O3. Optimisation and bug fixing. The third objective is to improve the reliability and performance of the software by refactoring inefficient code paths and eliminating bugs observed in the legacy application. Several parts of the original code showed strong coupling between modules, duplicated logic, and methods that continuously increased memory usage until data acquisition froze. Other functions performed long sequences of almost identical calls that could be replaced by simple loops, and some GUI routines unnecessarily re-created entire widgets instead of updating their content.

In the redesigned application these issues are addressed systematically. Long-running acquisition tasks are decoupled from the user interface, memory usage is stabilised, and repetitive code is replaced by parameterised loops. GUI updates are performed through dedicated update methods that reuse existing widgets. This includes the refactoring of the signal-processing pipeline that implements the phase-retrieval algorithm used to extract the phase shift from the interferometric sensor output.

O4. Addition of new core functionalities as extra tabs.

The fourth objective is to extend the functionality of the software through new tabs in the graphical interface. In the legacy application, the main window contains only two views: *Calibration* and *Measurement*, which together implement the core workflow to prepare the sensor and record real-time data from the photodiode.

The new interface incorporates an intensity verification workflow as an additional tab (see Figure 7), with a PyQt6-based layout that includes *Start*, *Stop* and *Clear* controls

TABLE 1: SUMMARY OF PROJECT OBJECTIVES

ID	Description
O1	Analyse the legacy application and refactor it into an MVC-style architecture with clear separation of responsibilities.
O2	Rebuild the project using a modern and supported technology stack, ensuring compatibility with the NI hardware and improving future maintainability.
O3	Optimise the code base by removing inefficient routines, reducing memory issues, and fixing functional bugs.
O4	Add new interface tabs for hardware checking, data processing, and improved real-time monitoring.
O5	Provide clear documentation for installation, execution, and future maintenance of the software.

and a real-time plot for visual inspection of the initial channel behaviour. A second new tab is dedicated to data processing. The *Processing* tab (see Figure 10) allows the user to select a stored .DAT file, plot the corresponding static traces, apply a configurable cut-off frequency to smooth the curves, and optionally save the processed data once a satisfactory representation has been obtained. The redesign of the front end with PyQt6 has also been used to introduce additional usability improvements. For instance, the Measurement tab (see Figure 9) includes a timer that shows the elapsed acquisition time, and a drop-down selector that lets the user choose which hardware channel to display in each of the three plots.

O5. Documentation and future maintainability. The final objective is to provide basic but clear documentation so that future researchers and developers can install, configure and extend the application with minimal friction. In the new project, the README.md file explains how to prepare a fresh setup on a new computer, lists the software and hardware prerequisites, summarises the project structure, and describes the steps required to generate a new executable using PyInstaller.

3.3 Summary of objectives

For quick reference, Table 1 summarises the main objectives of the project.

4 SYSTEM REQUIREMENTS

This section refines the project objectives into concrete functional and non-functional requirements that the new software must satisfy.

4.1 Functional requirements

- **RF1. Session control.** The system shall allow the user to start, stop and restart a measurement session from

the GUI without closing the application.

- **RF2. Hardware availability check.** Before starting a session, the software shall verify the presence and basic configuration of the NI DAQ device and report any errors.
- **RF3. Data acquisition.** The system shall acquire data from the configured NI DAQ channels and feed them into the processing pipeline.
- **RF4. Calibration workflow.** The system shall guide the user through the calibration procedure and store the resulting calibration parameters for later measurements.
- **RF5. Real-time visualisation.** The GUI shall display in real time the most relevant signals (laser current, I_{up} , I_{down} , SR , phase) for a selected channel.
- **RF6. Intensity pre-check.** The application shall provide a dedicated view to monitor the instantaneous intensity (voltage) of each hardware channel before calibration.
- **RF7. Offline processing.** The system shall be able to load a .DAT file generated by a measurement, plot the stored traces, apply a configurable cut-off frequency, and export the processed data.
- **RF8. Data persistence.** The system shall save measurement data and associated configuration to disk for later inspection.

4.2 Non-functional requirements

- **RNF1. Maintainable architecture.** The code base shall be organised in modules following the MVC-style structure described in Section 3, avoiding a single monolithic controller file.
- **RNF2. Compatibility.** The software shall run on the laboratory Windows PC using Python 3.12, PyQt6 and the NI DAQ drivers approved for the BiMW setup.
- **RNF3. Responsiveness.** During normal operation the GUI shall remain responsive while data acquisition is running; long-running tasks must not block the main thread.
- **RNF4. Robustness.** The system shall handle configuration and hardware errors gracefully, reporting clear messages instead of crashing or freezing.
- **RNF5. Usability.** The GUI shall present the measurement state and signals in a way that can be understood by lab users who are not developers of the code.
- **RNF6. Deployability.** It shall be possible to generate a standalone executable using PyInstaller so that the software can be run without an IDE.

5 METHODOLOGY

5.1 Development approach and lifecycle management

The project followed an iterative and incremental software engineering process inspired by agile development practices. Because this Final Degree Project was carried out by a single developer, heavy-weight team frameworks (such as full Scrum ceremonies) were not applied. Instead, the work was organised into two-week iterations (sprints) from week 4 to week 21 of the semester, with each sprint delivering a small but functional code increment that could be tested on the laboratory setup.

Crucially, the external interdisciplinary collaborators comprising the primary hardware researcher and the experimental PhD students from the NanoB2A group at ICN2 fulfilled the project role of Stakeholders and Product Owners. At the conclusion of each two-week sprint, a review demonstration was performed on the laboratory PC setup. This structured evaluation permitted direct inspection of the user interface evolution and numerical output verification against legacy recordings. The continuous stream of experimental feedback triggered localized backlog grooming and scope changes, which were dynamically managed across the development timeline.

5.2 Sprint chronology and iterative evolution

The practical progression of the implementation was marked by several environmental constraints and hardware-dependent milestones that shaped the software's final layout:

During early iterations (Sprints 1 and 2), focus was placed on isolating requirements from the 2 500-line legacy monolithic codebase and mapping the algebraic dependencies of the bimodal waveguide phase equations. Engineering benchmarks were derived by cross-referencing visual design standards found in modern PyQt6 documentation guides [7].

In Sprint 3, a practical constraint appeared due to the network security policy at ICN2: laboratory computers are not allowed to install packages directly from public repositories (for example via `pip install`). To satisfy the deployability requirements (**RNF2**, **RNF6**), all required wheels for Python 3.12 were instead downloaded manually from the official PyPI repository and stored in a local directory, allowing fully offline installation on the target machine.

Sprints 4 and 5 centered on implementation of the high-frequency asynchronous acquisition loops (**RF3**) relying heavily on the official NI-DAQmx Python low-level binding specifications [6]. During live validations with the laboratory instrumentation, intermittent voltage coupling behavior was observed from the laser diode driver. Specifically, when stopping an ongoing calibration routine, the automated command signals did not consistently return the physical laser operating point to a baseline state of 150 mA. Collaborative investigations with the experimental team confirmed this anomaly was dominated by physical thermal drifts within the analog hardware driver rather than numeric overflows in the software logic. This discovery guided a major scope adjustment on May 4th: the re-

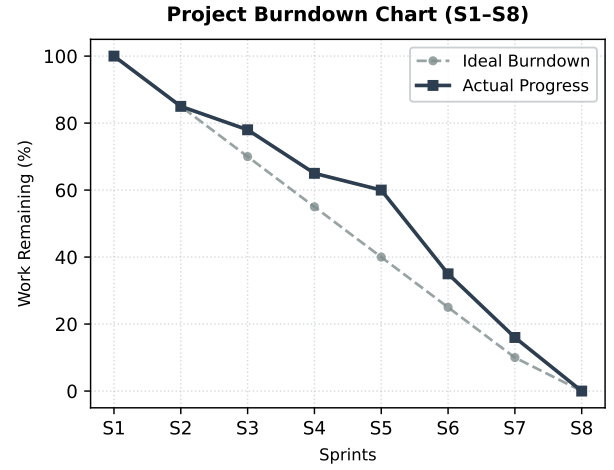


Fig. 3: Project burndown chart showing ideal versus actual task mitigation across the eight development sprints.

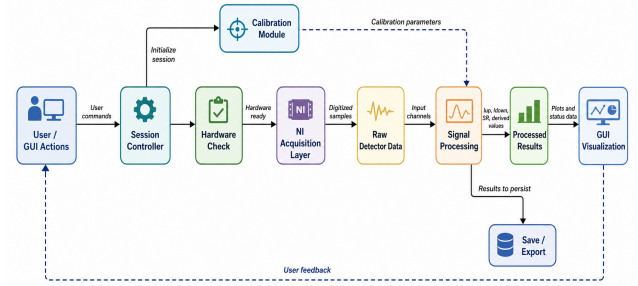


Fig. 4: High-level data flow and module responsibilities in the BiMW-v4 control software, from user actions to processed results.

searcher requested a permanent pre-calibration diagnostic panel to prevent active runs under unsafe laser conditions. This feature was integrated as a primary *Intensity Check* tab.

Final iterations (Sprints 6 to 8) were dedicated to quality-of-life additions, layout realignments to preserve scientific visual patterns familiar to lab users (**RNF5**), and the design of automated data persistence protocols.

The historical trajectory of task mitigation against the project timeline is summarized in the burndown visualization in Figure 3. Table 2 in Appendix A.2 summarises the timeline and main activities of these development iterations.

5.3 Refactoring and design architecture

The key strategy adopted for the structural refactoring of the legacy application followed an architectural decomposition into a Model–View–Controller framework, diagrammed systematically in Figure 4.

The responsibilities are isolated across explicit packages:

- *Models* (`models/`) encapsulate configuration and state as Python `@dataclass` objects (for example `MeasurementConfig`, `CalibrationResult`, `AcquisitionState`) with strict static type-hinting to prevent runtime typing failures.
- *Core* modules (`core/daq` and `core/signal`) implement DAQ abstraction and signal processing as

hardware-agnostic pure functions operating exclusively on NumPy arrays.

- *Controllers* (`controllers/`) orchestrate the functional pipelines and map system inputs without maintaining direct references to UI widgets.
- *Views* (`views/`) contain PyQt6 interface definitions restricted to visual layout declarations.

Hardware abstraction layer. To ensure code execution on development machines lacking physical access to National Instruments hardware controllers, the low-level library bindings were wrapped inside an isolated `AcquisitionTask` worker. The module traps missing module exceptions for `nidaqmx` and degrades functionality gracefully by feeding synthetic waveforms or zeroed arrays back into the control loop, separating developer work from strict laboratory access schedules.

Automated testing verification. A suite of regression assertions was developed in `tests/test_core.py`. These automated unit tests evaluate edge conditions inside the numerical engine, ensuring explicit handling of algebraic anomalies such as division-by-zero during sensitivity parameter calculation ($SR_k(t)$ when $I_{up} + I_{down} = 0$).

5.4 Use of tools and artefacts

The main development environment consisted of Python 3.12, PyQt6 and the NI-DAQmx drivers installed on a Windows laboratory PC. PyCharm was used as the IDE, and `PyInstaller` was adopted from the beginning to ensure that the final application could be distributed as a standalone executable. A `README.md` file in the project repository documents the installation procedure, the required local packages and the steps to build the executable.

Screenshots of the old script and the four main tabs (Intensity Check, Calibration, Measurement and Processing) were captured during the final sprints and are included in Appendix A.4 (Figures 6–10), where they serve as visual support for the description of the user workflow.

5.5 Risks and limitations

During the development of the new control software several practical risks and limitations were identified. A first risk was the strict network security policy at ICN2, which prevents direct installation of Python packages from public repositories on the laboratory computers. This constraint could have blocked the migration of the project to Python 3.12 and PyQt6. To mitigate it, all required wheels were downloaded manually from the official PyPI repository and installed locally from a project folder, at the cost of extra setup effort and of having to maintain this local package archive over time.

A second limitation was the restricted access to the full BiMW experimental setup during some phases of the project. In particular, early functional tests of the acquisition and plotting pipelines were performed using simplified optical conditions (for example by covering or directly

illuminating the photodiode) instead of the complete chip-level alignment. As a consequence, the present validation focuses on the software behaviour and numerical consistency, while more extensive characterisation of the system under all experimental protocols is left to future work.

Finally, some residual risks are inherent to the analogue nature of the hardware. Thermal drifts in the laser driver and other components can still lead to slow changes in the operating point that are outside the scope of the software. The new application includes safeguards such as the *Intensity check* tab and clearer error reporting to detect unsafe conditions, but it cannot fully eliminate hardware-induced instabilities. These limitations should be considered when interpreting results obtained with the current platform configuration.

6 RESULTS AND DISCUSSION

6.1 Functional results

The main outcome of the project is a stable application capable of managing the complete experimental workflow of the BiMW biosensor platform, from pre-calibration checks to data acquisition and offline processing. All core functionalities defined in Section 4 have been implemented: initialisation of the experimental session, hardware availability checks, calibration, real-time measurement, status reporting and data persistence. In addition, two new tabs—Intensity Check and Processing—have been integrated into the workflow, providing a more robust and usable interface for the researchers who operate the system.

A high-level comparison between the legacy application and the redesigned software is summarised in Table 3 (Appendix A.3).

6.2 Experimental validation and User Acceptance Testing

Validation of the BiMW-v4 software framework was divided into automated numeric assertions and formal laboratory User Acceptance Testing (*proves d'acceptació*) conducted with active research staff at the ICN2 facilities.

The user acceptance evaluations focused on three main criteria: system stability during long runs, responsiveness under sustained acquisition, and efficiency of post-processing. The refactored application completed continuous tests of up to 60 minutes without UI freezing, visible lag or memory build-up, in contrast with the legacy platform, which had shown instability during long measurement sessions. This improvement comes from separating the high-frequency acquisition workers from the main Qt interface thread.

A final one-hour laboratory experiment was also performed using repeated doses of the same chemical compound on the full seven-channel BiMW setup. During this session, the software completed acquisition, phase reconstruction, filtering and saving without freezing, data loss or visual artefacts. Figure 5 shows the resulting loaded phase data for the `test_final.DAT` experiment, where the stepwise response is consistently reproduced across channels and provides practical evidence that the application behaves correctly under realistic operating conditions.

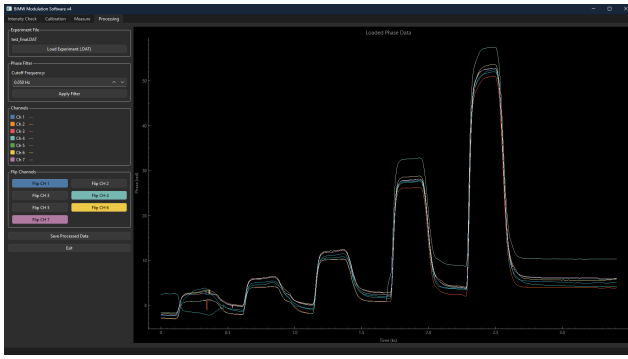


Fig. 5: Loaded phase data for the final one-hour validation experiment with increasing doses of the same chemical compound.

From a qualitative workflow perspective, validation feedback gathered from lab personnel indicated a significant reduction in post-processing friction. The introduction of the integrated offline filtration tool in the *Processing* tab (visualized in Figure 10) eliminated the traditional requirement for researchers to manually port experimental output .DAT data text into external processing software or localized standalone scripts. Researchers successfully completed the end-to-end operational cycle—comprising initial intensity baseline validation, automated sensor calibration tracking, real-time signal extraction, and noise filtering—independently upon their first interactive session.

6.3 Relation to objectives and feasibility

Overall, the results obtained align with the objectives defined at the beginning of the project. The redesign has transformed a fragile, tightly coupled codebase into a modular application that can be understood, extended and maintained by future engineers in the group. The new features requested by the researcher—especially the Intensity Check tab, status indicators, timer control and improved plotting options—have been implemented and incorporated into the normal workflow, improving the day-to-day usability of the platform without altering the underlying physics of the BiMW sensor.

For clarity, the main results can be mapped directly to the specific objectives defined in Section 3:

- **O1 – Legacy analysis and MVC refactoring.** Achieved through the modular package structure shown in Figure 4 and detailed in Appendix A.1, which separates controllers, models, views and core DAQ/signal modules.
- **O2 – Modern, maintainable technology stack.** Implemented by migrating the application to Python 3.12 and PyQt6, while preserving compatibility with the existing NI DAQ drivers and preparing the project for distribution via PyInstaller.
- **O3 – Optimisation and bug fixing.** Validated by the absence of UI freezes or memory build-up in 60-minute test runs, and by centralising long-running acquisition work in dedicated worker threads instead of the main GUI thread.

- **O4 – New interface tabs and workflow support.** Satisfied by the addition of the Intensity Check and Processing tabs and the improved Measurement and Calibration views, which together support the complete experimental workflow from pre-checks, to offline filtering.
- **O5 – Documentation and onboarding.** Addressed through the project README.md, the directory map in Appendix A.1 and the GUI mapping tables in Appendix A.4, which document how to install, run and extend the software.

From the point of view of an engineering degree, the scope and depth of the results are realistic and achievable with the available resources. The project uses widely adopted technologies (Python, PyQt, NI-DAQmx), follows a clear iterative methodology with two-week sprints, and focuses on architectural quality, reliability and user-centred improvements rather than on building an entirely new system from scratch. This makes the work representative of the type of software engineering contribution that can be expected from a computer engineer collaborating in an interdisciplinary research environment such as ICN2.

7 CONCLUSIONS

The project has achieved its main objective: to redesign and extend the control software of the BiMW photonic biosensor platform into a modular, maintainable and usable application that remains compatible with the existing NI hardware. The legacy monolithic `Controller.py` has been replaced by an MVC-style architecture with clearly separated models, controllers, views and core modules for DAQ and signal processing, supported by structured dataclasses, shared services and a small but meaningful unit-test suite for the most critical numerical routines. From a practical standpoint, the new Intensity Check and Processing tabs, together with the improvements in the Calibration and Measurement tabs (status indicators, timer, clear plots, axis labels, automatic navigation and safer exit handling), provide a smoother and more robust workflow for the researchers using the platform in their daily experiments.

Not all aspects of the initial vision could be completed to the same level of maturity. Some user-facing documentation remains limited to the project README.md and the explanations in this report, and additional in-application guidance would further lower the entry barrier for new users. The control over the laser current after stopping calibration or measurement could not be fully stabilised around 150 mA, as the residual drift is dominated by hardware behaviour rather than by the software waveform generation. Similarly, the use of a local installation of all Python dependencies—initially introduced as a workaround for network restrictions at ICN2—has advantages for reproducibility but also implies that future updates must be carefully managed.

Looking ahead, several lines of continuation are natural extensions of this work:

- Centralising and structuring the storage of acquired data to facilitate long-term organisation and analysis.

- Investigating, together with hardware specialists, improvements to the laser control loop to keep the intensity closer to 150 mA when stopping calibration or measurement.
- Migrating the project to a shared version-control platform (for example GitHub or an internal repository) to support collaborative development, issue tracking and long-term maintenance.
- Adding an integrated help or tutorial view within the application, describing the purpose of each tab, the meaning of each plot and the recommended interpretation of the signals, so that new users can learn the workflow directly from the software.

USE OF GENERATIVE AI

In accordance with the institutional guidelines on the responsible use of generative AI in Final Degree Projects, large language models have been used exclusively as an auxiliary tool during the development of this work. Concretely, generative AI has supported: (i) the refinement of the English writing style in some sections of the report, including the abstract and introductory paragraphs; (ii) the restructuring and clarification of explanations aimed at software engineers who are not familiar with photonic biosensors; and (iii) the generation of alternative formulations for figure captions, table descriptions and internal comments in the code.

In all cases, the technical content, architectural decisions and implementation of the software have been defined and validated by the author, based on the project requirements and the feedback received from the tutor and the research group. When generative AI suggested code fragments or data structures, they were used only as inspiration and were subsequently rewritten, verified and integrated manually into the project. No external training datasets or proprietary experimental data from ICN2 have been uploaded to AI services; all interactions have been limited to high-level descriptions that do not reveal confidential information.

Concretely, tools such as Gemini and Perplexity AI were used only for these language-level tasks.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my tutor, Joan Protasio, for his continuous guidance, detailed feedback and constructive recommendations throughout the entire project. His support has been essential both for framing the work from a software engineering perspective and for keeping the scope of the TFG realistic.

I am also very grateful to Jhonattan Córdoba at ICN2 for his help and tutoring during the development, especially regarding the experimental context of the BiMW platform and the practical constraints of working with the existing laboratory infrastructure. His explanations and comments have been key to understanding how the software fits within the broader research pipeline.

Finally, I would like to thank Andrés Alonso for his availability and support in configuring and debugging the hardware setup, as well as for his continuous feedback on how

the software behaved during real measurements. His suggestions on usability and workflow have directly influenced the design of the new interface and have ensured that the final application is truly useful for the researchers who will use it.

REFERENCES

- [1] L. M. Lechuga, “Tecnología de biosensores para el diagnóstico descentralizado,” *Boletín SEBBM*, no. 34, pp. 1–4, 2021.
- [2] H. Altug, S.-H. Oh, S. A. Maier, and J. Homola, “Advances and applications of nanophotonic biosensors,” *Nature Nanotechnology*, vol. 17, pp. 5–16, 2022.
- [3] M. Soler and L. M. Lechuga, “Label-free photonic biosensors: key technologies for precision diagnostics,” *ChemistryEurope*, vol. 3, e202400106, 2025.
- [4] K. E. Zinoviev, A. B. González-Guerrero, C. Domínguez, and L. M. Lechuga, “Integrated bimodal waveguide interferometric biosensor for label-free analysis,” *Journal of Lightwave Technology*, vol. 29, no. 13, pp. 1926–1934, 2011.
- [5] B. Bassols-Cornudella *et al.*, “Novel sensing algorithm for linear read-out of bimodal waveguide interferometric biosensors,” *Journal of Lightwave Technology*, vol. 40, no. 1, pp. 237–244, 2022.
- [6] National Instruments, “NI-DAQmx Python Documentation,” *Read the Docs*, 2026. [Online]. Available: <https://nidaqmx-python.readthedocs.io/>
- [7] The Qt Company, “Qt for Python Documentation (Qt6 Layouts),” *Qt Documentation Center*, 2026. [Online]. Available: <https://doc.qt.io/qtforpython-6/>
- [8] M. Fitzpatrick, “PyQt6 Tutorial - Create Desktop Applications with Python,” *PythonGUIs UI Guides*, 2026. [Online]. Available: <https://www.pythonguis.com/pyqt6-tutorial/>
- [9] Python Software Foundation, “PyPI — The Python Package Index Repository,” 2026. [Online]. Available: <https://pypi.org/>
- [10] National Instruments, “Using Source Control in LabVIEW,” *NI Documentation*, 2025. [Online]. Available: <https://www.ni.com/docs/en-US/bundle/labview/page/using-source-control-in-labview.html>
- [11] National Instruments, “Licensing My LabVIEW Application or LabVIEW Library,” *NI Support*, 2018. [Online]. Available: NI Knowledge Article

APPENDIX

A.1 Project structure and module decomposition

The final directory structure of the BiMW-v4 project is summarised below.

```

BiMW-v4/
| main.py
| controllers/
|   | main_controller.py
|   | calibration_controller.py
|   | measurement_controller.py
|   | processing_controller.py
|   | intensity_controller.py
|   | measurement_setup_controller.py
|   | data_controller.py
| core/
|   | daq/
|   |   | acquisition.py
|   |   | daq_setup.py
|   |   | device_discovery.py
|   | signal/
|   |   | signal_generation.py
|   |   | signal_filter.py
|   |   | sr_processor.py
| models/
|   | measurement_config.py
|   | calibration_result.py
|   | acquisition_state.py
| services/
|   | logger_service.py
|   | timer_service.py
| views/
|   | main_window.py
|   | tab_calibration.py
|   | tab_measure.py
|   | tab_processing.py
|   | tab_intensity.py
| Lib/
|   | Colors.py
|   | Strings.py
| scripts/
|   | build.py
| tests/
|   | test_core.py
| README.md

```

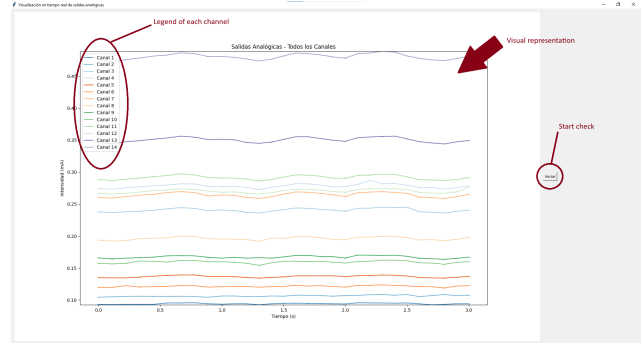


Fig. 6: The legacy standalone hardware checking script used prior to architectural refactoring, showing limited user controls and monolithic visualization boundaries.

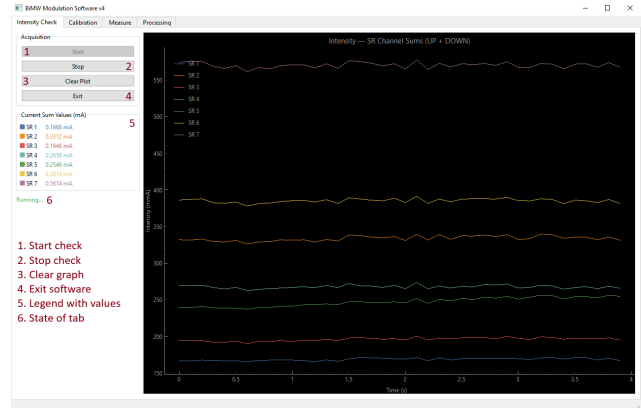


Fig. 7: Annotated view of the new Intensity Check Tab, designed to handle primary pre-calibration hardware telemetry assertions safely.

A.2 Historical development sprint summary

Table 2 details the sprint chronological roadmap and the functional goals targeted during the engineering development process.

A.3 Implementation comparison metrics

Table 3 presents a systematic structural comparison between the legacy monolithic architecture and the refactored engineering solution.

A.4 Graphical User Interface Manual and Functional Mapping

This section serves as both a visual guide to the developed BiMW-v4 system and a functional map connecting the interactive view elements to their underlying software execution logic. For each view, an annotated screenshot with numbered callouts is paired with a comprehensive tracking table mapping elements to their software controllers.



Fig. 8: Annotated view of the refactored Calibration Tab. Table 4 below explains each part.

TABLE 2: SUMMARY OF HISTORICAL SPRINTS AND MAIN OBJECTIVES

Sprint	Weeks (2026)	Main activity summary
S1	4–5	Initial domain exploration, paper reading, and exploratory analysis of the legacy codebase requirements.
S2	6–7	Structural codebase evaluation, dependencies auditing, and formulation of target MVC patterns.
S3	8–9	Baseline PyQt6 view designs and configuration of localized package archives for network compliance.
S4	10–11	Implementation of high-frequency telemetry routines and integration of quality-of-life UI features.
S5	12–13	Mid-project scope modification to engineer the localized hardware diagnostic panel (Intensity Check tab).
S6	14–15	Development of flow automation algorithms, status reporting mechanics, and automated file serialization.
S7	16–17	Full hardware loop validation tests inside the final laboratory framework and deployment documentations.
S8	18–19	Execution of prolonged stability stress runs and formulation of future extension guidelines.

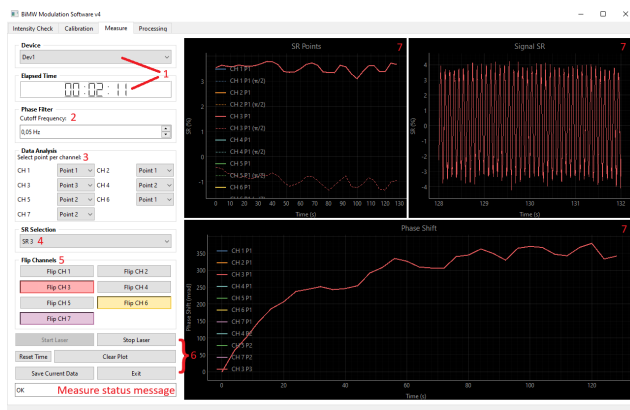


Fig. 9: Annotated view of the core Real-Time Measurement Tab window. Table 5 below explains each part

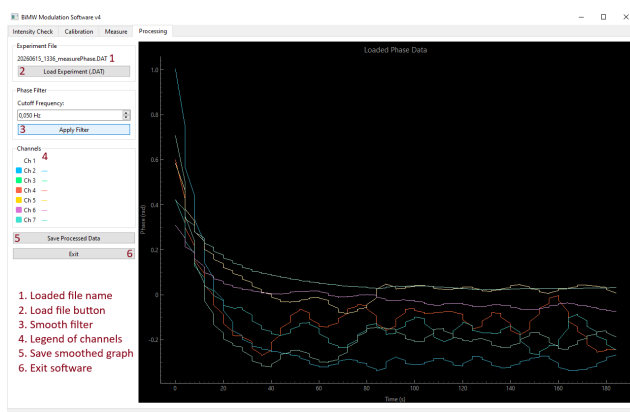


Fig. 10: Annotated view of the analytical Offline Processing Tab interface, highlighting filtering customization panels.

TABLE 3: COMPARISON BETWEEN THE LEGACY BiMW CONTROL SOFTWARE AND THE REDESIGNED BiMW APPLICATION

Aspect	Legacy software	New software
Architecture	Single monolithic file <code>Controller.py</code> of ~2 500 lines mixing GUI events, DAQ access, signal processing, calibration logic and file I/O.	MVC-style package structure with separate <code>models/</code> , <code>controllers/</code> , <code>views/</code> , <code>core/daq</code> and <code>core/signal</code> modules. Responsibilities are explicit and each layer can be understood and modified independently.
Configuration and state	Dozens of instance attributes initialised ad hoc in the controller, without typing or clear grouping.	Structured <code>@dataclass</code> models (<code>MeasurementConfig</code> , <code>CalibrationResult</code> , <code>AcquisitionState</code>) encapsulate parameters and runtime state, with helper methods for reset and derived values.
Signal-processing logic	SR computation, filtering and phase retrieval embedded directly in GUI/DAQ methods, with limited protection against numerical edge cases.	Pure functions in <code>core/signal/</code> (<code>compute_sr_signal()</code> , <code>center_sr_signal()</code> , <code>calculate_phase_unwrapped()</code> , etc.) operate on NumPy arrays and include explicit handling of corner cases such as division by zero.
Hardware interaction	Direct calls to <code>nidaqmx</code> scattered through the controller; the program cannot start on machines without NI drivers or hardware.	<code>AcquisitionTask</code> wraps all NI-DAQ calls and exposes a small public API. When <code>nidaqmx</code> is not available, methods degrade gracefully, allowing development and basic testing without hardware.
Tabs and workflow	Two main tabs (Calibration and Measurement) with manual navigation and no dedicated pre-calibration intensity check.	Four tabs (Intensity Check, Calibration, Measurement, Processing), with automatic handoff from calibration to measurement and a dedicated pre-check workflow for photodiode intensities and offline processing of <code>.DAT</code> files.
Logging and diagnostics	Repeated blocks of logging code in dozens of <code>try/except</code> blocks, with manual file management and inconsistent messages.	Centralised <code>LoggerService</code> with <code>log()</code> and <code>log_exception()</code> methods, providing uniform formatting and ensuring that all messages are flushed and timestamped consistently.
Code reuse (DRY)	Multiple almost identical methods and code blocks per channel (e.g., seven variants of the same operation on different indices).	Loops and parameterised helpers replace duplicated code; channel-specific behaviour is handled by passing indices or configuration objects instead of copy-paste.
User interface and plots	PyQt5 interface with unlabeled axes, limited status feedback and no way to clear plots without restarting measurements.	PyQt6 interface with labelled axes (including physical units), real-time status text in each tab, <i>Clear plot</i> buttons, timer for measurement duration and controls to select and flip individual channels.
Exit and resource cleanup	Application exit closes windows abruptly, with risk of leaving DAQ tasks or files open.	<code>MainController</code> implements explicit shutdown: stopping DAQ tasks, closing files, logging shutdown and asking for confirmation before exiting to avoid accidental termination during experiments.
Testing and verification	No automated tests; behavioural verification relies entirely on manual runs.	Unit tests in <code>tests/test_core.py</code> cover waveform generation, laser voltage limits, filter behaviour and critical SR and phase computations, providing a baseline of regression protection.

TABLE 4: FUNCTIONAL COMPONENT MAPPING FOR THE CALIBRATION TAB (FIGURE 8)

Id	UI element / block	Software logic / event handler
1	Device selector (QComboBox)	Allows the user to select the NI-DAQ device identifier. The choice is passed to the <code>AcquisitionTask</code> , which uses it when initialising the hardware session and configuring the input/output channels before calibration starts.
2	Signal parameters (modulation, frequency, laser current set, current range)	These widgets edit the fields of a <code>CalibrationConfig</code> dataclass. When the user presses <i>Start Laser</i> , the current configuration is read and used to build the analogue output waveform that drives the laser during the calibration interval.
3	Calibration controls (calibration time, <i>Start Laser</i> , <i>Start Calibration</i> , <i>Stop</i>)	Trigger the state machine implemented in <code>CalibrationController</code> . <i>Start Laser</i> enables the analogue output, <i>Start Calibration</i> launches the timed acquisition loop and computation of SR and coupling, and <i>Stop</i> aborts the process, stopping both DAQ tasks and GUI timers in a controlled way.
4	Filter / SR information and SR selection	The filter/SR panel allows the operator to select the reference SR channel and, optionally, to centre the SR signal of one channel. These settings are forwarded to the signal-processing functions that compute centred SR traces and to the plotting layer so that only the selected SRs are rendered or highlighted in the graphs.
5	Coupling summary (I.UP / I.DOWN values per channel)	Shows, for each SR channel, the mean currents measured in the UP and DOWN elements of the photodiode during calibration. The values are extracted from the acquired traces by helper functions in <code>core/signal</code> and stored inside the <code>CalibrationResult</code> object for later use in measurement.
6	Plot region (Laser Current, SR Signal, Readout Photodiode)	Contains three <code>pyqtgraph</code> widgets subscribed to streamed samples from <code>AcquisitionTask</code> . The top-left plot displays the laser drive current, the top-right plot shows all SR signals, and the bottom plot shows the raw photodiode currents, providing visual feedback that the calibration waveform and the sensor response are behaving as expected.

TABLE 5: FUNCTIONAL COMPONENT MAPPING FOR THE REAL-TIME MEASUREMENT TAB (FIGURE 9)

Id	UI element / block	Software logic / event handler
1	Device selector and elapsed-time display	The device combo reuses the same configuration used in calibration. The elapsed-time widget is driven by a <code>TimerService</code> associated with the measurement session; it is started when <i>Start Laser</i> is pressed in this tab and stopped when the user ends measurement, providing a reference for long experiments.
2	Phase filter configuration (cut-off frequency)	Lets the user choose the cut-off frequency of a low-pass filter applied to the unwrapped phase. When the value changes, the <code>MeasurementController</code> recomputes the filtered phase traces using the signal-processing functions and refreshes the <i>Phase Shift</i> plot to attenuate high-frequency noise.
3	Data analysis controls (select point per channel)	For each of the seven SR channels, the user selects which SR point (P1, P2, P3, ...) should be monitored. The selected indices are stored in the measurement state and used to extract a single SR value per channel at every update, forming the time series shown in the <i>SR Points</i> graph.
4	SR selection (QComboBox)	Chooses which SR channel is currently active for detailed inspection. The selected channel is highlighted in the plots and determines which phase trace is displayed in the bottom <i>Phase Shift</i> graph, allowing the user to focus on the channel corresponding to a particular biomarker or sensing region.
5	Flip channel buttons (<i>Flip CH 1</i> – <i>Flip CH 7</i>)	Each button toggles a sign flag for the corresponding channel. When the flag is active, the associated phase array is multiplied by -1 before plotting and before saving data, compensating for wiring or sign conventions and ensuring that all channels share a consistent orientation of the phase response.
6	Measurement actions (<i>Start Laser</i> , <i>Stop Laser</i> , <i>Reset Time</i> , <i>Clear Plot</i> , <i>Save Current Data</i> , <i>Exit</i>)	Control the life cycle of a measurement run. They start and stop the underlying DAQ tasks, reset the session timer (typically when the sample is injected), clear the plot buffers, persist the current SR and phase traces to timestamped files, and close the application after performing a clean shutdown of hardware and resources.
7	Plot region (SR Points, Signal SR, Phase Shift)	The three <code>pyqtgraph</code> widgets present the main read-out of the BiMW sensor during measurement: the evolution of the selected SR point per channel, the full SR waveform for the active SR channel, and the unwrapped phase shift over time. Together they give the operator immediate feedback on the stability of the signal and the progress of the molecular binding event being monitored.