ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THPD078

PREPARING THE MIGRATION OF PYTHON GUIs FROM Qt5 TO Qt6 AT CERN

I. Sinkarenko*, S. Deghaye†, C. Roderick‡, P. Mantion§, CERN, Geneva, Switzerland

Abstract

The use of Python in CERN's particle accelerator domain has grown steadily over the last decade. For non-software developers, it integrates well into the reality of a physics laboratory, where people who develop hardware or design operational workflows often author associated software, including Graphical User Interfaces (GUIs). While the general strategy is to standardise GUIs using no-/low-code solutions, complex use cases require more customised GUIs. For this, Python and Qt are combined through PyQt bindings.

Qt5 has been used for several years but with its end-of-life already reached, migration to Qt6 is a priority. At the same time, GUI applications in the CERN Controls ecosystem require long-term stability that exceeds the software industry average. Therefore, the migration requires a lot of care, taking into account both maintenance costs and long-term risks associated with reliance on a 3rd-party technology. To plan the upgrades, alternatives to the current PyQt approach have been reviewed. In the past, PyQt5 was an obvious choice, but investment into the PySide library by the Qt Company now presents more options.

We outline the criteria considered in our comparison of PyQt, PySide, and QML, and present the resulting decisions and rationale behind them. Finally, we discuss long-term risks associated with Qt bindings for Python and describe how we will manage the migration to future versions of the Qt framework, such as Qt7.

INTRODUCTION

Since Python became an officially supported language of the accelerator controls group at CERN in 2019, it has catalysed the development of a substantial body of new software [1]. This includes Graphical User Interfaces (GUIs) of varying complexity, powered by PyQt, the Python bindings for the Qt framework. PyQt5, in particular, has been a mainstay for several years, facilitating user applications in control rooms, hardware development in laboratories, and personal research initiatives. Despite the fact that Qt5 (the underlying C++ library upon which PyQt is based) has reached its official end-of-life, it remains the predominant version of the toolkit within CERN's Accelerators and Technology Sector (ATS). The prerequisites for migrating to the newer Qt version have only been fulfilled recently, with the move away from CentOS7 [2] which was too old for Qt6.

At CERN, PyQt initially emerged as an alternative to the long-established GUI technology, Java Swing, which served as the primary GUI toolkit for over two decades. However, Oracle's lack of investment in Swing, combined with the steep decline in its wide-spread usage, raised significant concerns and prompted the search for viable alternatives. Simultaneously, the growing volume of in-house control room software and the associated challenges of maintaining an expanding portfolio encouraged a reassessment of the GUI strategy at a higher level. After several iterations, the new strategy took shape, advocating for no-code or low-code solutions for standardised or simpler scenarios, typically represented in a dashboard-like format. This approach left more complex use cases to be addressed through traditional imperative programming, either in Python (PyQt) or Java (Swing). A new web-based solution, WRAP [3], emerged to address the no-code requirements. The CERN community of equipment experts and operations teams soon began to favour Python over Java for new GUI projects. This was driven by the increasing global popularity of Python, which was particularly apparent with new recruits, and also coincides with the rise of AI.

In light of these developments, it was anticipated that Qt6 would become the sole recommended framework for complex desktop GUI applications. For most CERN teams, this implies migrating their existing Swing-based software. The planning for such substantial changes is typically aligned with CERN's accelerator schedule, which allocates 2-3 year periods for Long Shutdowns (LS) dedicated to upgrades of the Large Hadron Collider (LHC) and its injectors, followed by several years of physics operation, known as a *Run*. The upcoming third Long Shutdown, LS3 (starting mid-2026), represents the deadline by which the migration strategy, framework, and supporting tools must be fully developed. However, certain critical applications will undergo migration prior to this, subject to their operational criticality, driven by the fact that Qt5 has already reached its end-of-life. Concurrently, an evaluation of the long-term stability and associated risks of the chosen framework is crucial, as the software must remain stable and operational throughout the duration of the next Run, with the likelihood of continued use well beyond that time frame.

PYTHON QT LANDSCAPE

PyQt and PySide

In the context of Qt for Python, two primary options are available: PyQt, a well-established third-party library, and PySide, which is currently developed by The Qt Company, the organisation behind the Qt framework itself. At first glance, one might expect PySide, being developed by the same entity responsible for Qt, to offer superior integration. However, PySide was effectively abandoned for a number

* ivan.sinkarenko@cern.ch
† Stephane.Deghaye@cern.ch
‡ Chris.Roderick@cern.ch
§ pierre.mantion@cern.ch

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2025). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THPD078

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2025). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

of years, allowing PyQt to gain a significant user base and establish itself as the dominant choice. Even today, PySide is still catching up with PyQt in certain areas, as evidenced by the fact that their API coverage page lists several features available in PyQt that have not yet been implemented in PySide [4]. When looking at the historical context, it is helpful to consider their versioning strategies. PyQt adheres to the versioning scheme of the underlying C++/Qt framework, with PyQt4, PyQt5, and PyQt6 corresponding to bindings for Qt4, Qt5, and Qt6, respectively. In contrast, PySide began with PySide1 for Qt4 and PySide2 for Qt5, but it has more recently aligned itself with PySide6 for Qt6. The Qt Company has marketed PySide under the name "Qt for Python", although the codebase still predominantly uses the "PySide" label. Both libraries are built using distinct binding generators: PyQt uses SIP, while PySide is based on Shiboken. Consequently, their respective runtime support libraries, *PyQt6-sip* and *shiboken6*, will appear as dependencies during installation. Understanding of the differences in bindings generators is largely not required from Python GUI developers, as it is only important when dealing with the boundary crossing between Python and C++, which so far has been not needed for CERN's Python community.

Qt Widgets and QtQuick

The Qt framework provides two primary approaches for GUI development: Qt Widgets and QtQuick. Qt Widgets represents the original, classical method of building GUIs, relying on imperative object-oriented programming techniques. In contrast, QtQuick offers a more modern and rapid approach, ideal for prototyping. This method leverages QML, a declarative markup language, to define the composition of user interface components, while JavaScript is used to specify their behaviour. QtQuick components can be bridged to both C++ and Python code. Originally, this new approach was targeting mobile platforms and IoT devices, but recent releases have been made more usable for Desktop development. Both PyQt and PySide provide APIs for both Qt Widgets and QtQuick.

Additional Qt Tools

The Qt framework also provides several built-in tools to facilitate GUI development. These include Qt Designer, which allows users to design Qt Widgets interfaces visually through a drag-and-drop interface; *uic*, a command-line tool for generating code from Qt Designer's definition files; *rcc*, a tool for packaging non-executable resources into the application; and *linguist*, which is used for managing application localisations. Additionally, Qt Designer supports the integration of custom in-house widgets via its plugin system. These custom widgets must be specifically developed to work with Qt Designer by utilising Qt Designer-specific APIs. Both PyQt and PySide offer access to *uic*, *rcc*, and *linguist*. Notably, both libraries also include a Qt Designer plugin, which enables the loading of custom Python-based widgets into Qt Designer. This plugin links with libpython.so, the embedded Python interpreter, to execute Python logic when invoked by Qt Designer. While PySide ships Qt Designer and its plugin as part of the add-on package, the Linux version of PyQt does not include this tool by default, though it can be built from source.

SELECTING THE FUTURE TECHNOLOGY

Prior to undertaking the development of tools and guidelines for the migration of CERN GUIs, a preliminary assessment was conducted to evaluate the suitability of the aforementioned options and to determine the most appropriate combination. Given the constraints of limited resources and the prevailing inclination to maintain legacy systems, a trade-off was required to balance the projected long-term stability of the selected technology stack against the immediate effort required to execute the migration from Swing and PyQt5.

QtQuick Vs Qt Widgets

An initial step involved re-evaluating the suitability of QtQuick and QML to determine whether they present a viable alternative to Qt Widgets. A similar assessment was previously conducted [5], during which QtQuick-based applications were deemed immature and ill-suited to traditional desktop environments. However, in recent years, The Qt Company has devoted substantial engineering effort towards enhancing QtQuick, including migrating its rendering pipeline to the GPU. In Python, interaction with QtQuick is achieved via the `QmlEngine`, which manages the execution of QtQuick logic. This architecture enables interesting use-cases, such as dynamically downloading and injecting portions of the UI and associated behaviours at runtime without restarting the application. Despite these advantages, the use of `QmlEngine` introduces an additional abstraction layer that can hinder maintainability, particularly in complex projects. Critically, integration with widely used third-party Python libraries, such as *matplotlib* or *pyqtgraph*, is not possible. Moreover, unit testing of UI components becomes more challenging, and tools such as *pytest-qt* are not supported. Static analysis workflows are reduced to using a proprietary tool, *qmllint*, diverging from the standard Python toolchain commonly employed at CERN (e.g. *mypy*, *flake8*, *pylint*). Furthermore, adopting QtQuick would impose an additional learning curve on maintainers, requiring proficiency not only in Python but also in QML and JavaScript, an unrealistic expectation for many CERN collaborators who are not professional software developers. Consequently, Qt Widgets continues to be the preferred solution for fulfilling the role of Python-based GUI development within CERN's GUI strategy.

PyQt6 Vs PySide6

The next step was to choose the bindings library. Fortunately, both PyQt and PySide offer APIs that closely mirror that of Qt/C++, and many third-party libraries are compatible with both. From a licensing perspective, both are suitable for in-house development and internal deployment at

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THPD078

CERN. From a tooling standpoint, PySide6 offers an advantage through the inclusion of the *pyside6-designer*. In contrast, PyQt requires the maintenance of a custom-built Qt Designer. Nevertheless, the *pyside6-designer* tool could not be used directly in our environment, as it searches for `libpython.so` in a number of filesystem locations which are not used at CERN. Consequently, a custom wrapper is necessary for both PyQt and PySide, though the maintenance overhead is lower with PySide6. An additional promising feature of PySide6 is its planned integration with *asyncio* [6]. As an increasing number of CERN control system libraries expose `asyncio`-based interfaces, this feature has a potential to remove hard to maintain layers of integration between the GUI and `asyncio` that require managing multiple event loops. However, at the time of evaluation, this functionality remained experimental and unavailable in stable releases. In terms of governance, PySide6 presents a more open and community-friendly model. Nevertheless, PyQt6 offers a more practical migration path, primarily because existing CERN software is already based on PyQt5. More critically, PyQt6 provides a more mature and stable API. For instance, PySide6 lacked support for custom enumerations in user-defined Qt Designer plugins — an essential feature relied upon by several existing CERN widgets. This functionality, while supported in PyQt5, was initially broken in early PyQt6 releases due to changes in Qt's meta-object system, but was subsequently restored in Qt 6.8.1 and PyQt 6.8.0 following our feedback. It is anticipated that PySide6 may implement similar capabilities in future versions. Another technical limitation observed in PySide6 was the mishandling of certain low-level APIs, such as those involving image processing. Specifically, an exposed API that accepts `void*` pointers in C++ was not correctly bridged in PySide6, leading to segmentation faults when invoked from Python due to the absence of appropriate glue code for Python object conversion. In light of these immediate technical considerations, it was concluded that PyQt6 currently best satisfies CERN's requirements. Nonetheless, PySide6 remains a compelling alternative, particularly in the context of long-term sustainability. These future prospects will be examined in more detail later in this paper.

MIGRATION

Establishing a Baseline

Prior to commencing the migration process, two key decisions were taken to minimise long-term maintenance overhead: to discontinue PyQt5 support immediately, and to cease support for Python versions earlier than 3.11. Although internal policy mandates compatibility with Python 3.7 until the end of the current *Run* (mid-2026), such compatibility is not feasible with Qt6, as both PyQt6 and PySide6 [7] require a minimum of Python 3.9 stemming from the fact that Python 3.7 has reached its end-of-life [8]. Furthermore, support for Python 3.9 is expected to be phased out in the near future, as it too will reach end-of-life later this year. From an operating system perspective, Qt6 targets Red Hat Enterprise Linux 9, which has been adopted as the current standard platform within CERN's control system infrastructure [2].

In-House Widgets: Compatibility & Insights

A prerequisite for migrating operational applications was ensuring the compatibility of the in-house widget library that is utilised by the majority of Python-based GUI projects. Having a greater complexity relative to typical operational applications, particularly in handling edge cases, the library also served as a robust testbed for the migration process. As a result, porting this library to PyQt6 represented the most challenging step of the transition. The library is subject to rigorous testing using various automation tools and relies heavily on the availability and correctness of type stubs to enable static analysis with *mypy*. Therefore, it became necessary to develop a robust in-house type stubs package for PyQt6 to support this automation.

The custom `PyQt6-stubs` package was developed to override the built-in PyQt6 type annotations, which frequently do not accurately reflect the intended Qt API. It was found that type stubs are not a primary focus for the PyQt maintainer; besides, they are auto-generated from C++ signatures, which often fail to convey important information such as nullability or function overloads. To address these limitations, manual inspection of the relevant C++ source code was conducted to analyse the code paths of the most commonly used APIs. In PyQt5, supplying custom stubs was straightforward, as the PyQt5 package did not include a `py.typed` marker file. By contrast, PyQt6 introduced this marker, causing *mypy* to prioritise the built-in type definitions. To circumvent this, a *mypy* plugin was developed to suppress the marker file and direct mypy to utilise the improved definitions from the `PyQt6-stubs` package. While this workaround is effective, it introduces additional complexity. As such, the possibility of upstreaming the enhanced type annotations to the official PyQt6 package is under active consideration. However, doing so would likely necessitate modifications to the SIP bindings generator, a bespoke C++ tool used to generate the PyQt bindings, which would require additional investment.

In a manner analogous to the approach previously adopted for PyQt5, a PyQt6-compatible Qt Designer package was developed. The aforementioned in-house widget library leverages Qt Designer integration by exposing the widgets to be manipulated and rendered in Qt Designer. In addition, the library includes a collection of interactive examples, some of which rely on a functional installation of Qt Designer for demonstration purposes.

The initial migration phase proved instrumental in identifying key differences between Qt versions and in anticipating the most problematic changes. As a result, internal migration guidelines were published, documenting common issues along with recommended respective solutions. These guidelines also promote the use of static analysis, facilitated by the `PyQt6-stubs` package, as a means of detecting breaking changes in the API.

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2025). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THPD078

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2025). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

Challenges

One of the most significant technical challenges was the transition to scoped enumerations, primarily due to their widespread use. In PyQt6, all enum options are now represented as Python `enum.Enum` objects, for which access to their underlying values requires calling the `.value` attribute. However, data originating from Qt/C++ internals is still propagated as raw integer values. This discrepancy becomes particularly apparent when working with Qt signals or when overriding instance methods in model classes. Certain enums, such as `QCheckBox.CheckState`, are not `int` subclasses, while Qt/C++ continues to transmit their values as `int`. As a result, comparisons between `int` values and these enums always evaluate to false, introducing subtle bugs that static analysis cannot detect. Consequently, manual testing remains necessary in such cases. For widgets requiring integration with Qt Designer, replacing the `Q_ENUM()` macro with the `@pyqtEnum` decorator involved a degree of trial and error, particularly in scenarios where enums were defined in a superclass, but only the subclasses were exposed to Qt Designer. This led to inconsistencies in the XML serialisation performed by Qt Designer, producing output that could not be subsequently deserialised.

Early Adoption Strategy

With the foundation established and the libraries internally released as a beta version, several real-world applications were selected and migrated following the procedures outlined in the published tutorial. Many of these projects lacked established static analysis or testing infrastructure, as they are maintained by individuals without formal software engineering backgrounds. Introducing static analysis into such legacy codebases can be challenging, as the volume of false positives may overwhelm maintainers, often hiding genuine errors. An effective strategy was found through establishing a baseline of all existing issues. By recording and excluding these known issues from the analyser's output, attention can be focused on newly introduced problems. For *mypy* users, the *mypy-baseline* tool can be employed for this purpose, while *basedpyright* serves a similar function for those using *pyright*. Migration of smaller projects was carried out by a non-maintainer, with each project requiring between 30 minutes and 3 hours to complete, with a high level of confidence. In addition, a larger codebase of 38k lines of code was successfully migrated by its maintainer over the course of approximately one and a half days. These initial migration efforts provided valuable insights for refining both the tooling and the supporting guidelines. They also served as a timely reminder of the importance of improving project maintainability. The lessons learned and the recommended migration path were subsequently disseminated to the wider CERN community during the summer of 2025.

Initial Migration Results

The migration by software maintainers is anticipated to proceed gradually, beginning with lower-impact projects. All new development is strongly encouraged to adopt PyQt6 from the outset, without attempting to maintain backward compatibility. However, some existing projects may be unable to complete the migration before the start of LS3, and as such, continued support for Qt5 is expected to remain necessary in the interim. Several maintainers have already begun evaluating the new framework, with the overall experience being positive. Only a small number of cases required additional guidance. The most common issue encountered was the necessity of migrating entire codebases in a coordinated manner, as PyQt5 and PyQt6 widgets cannot be used concurrently within the same application. Such complication arises from the fact that PyQt5 and PyQt6 are separate Python packages. This makes it impossible to express the requirement of exclusively PyQt6 without PyQt5 using standard Python dependency notations (e.g. version ranges), which would have been feasible if both versions were released under a single package name. As a result, guidance on maintaining clean and separate development environments had to be communicated through documentation rather than enforced programmatically, which makes it harder to impose.

Subsequent Migration Outlook

Should the need arise, it would be feasible to develop tools to assist with the migration process. Potential options include exploring *OpenRewrite*-based solutions [9] or leveraging agentic Large Language Model (LLM)-based tools, such as Claude Code [10]. Additionally, given the team's prior experience with writing lightweight code migration utilities in Python using *libCST* [11], a bespoke solution could be developed, potentially also leaning on the help of agentic AI. However, at this stage, the effort required to develop such tools was considered disproportionate to the current need, as the migration is a one-off task. Tool development would only be justified if the unsupported migration of a large number of codebases were to generate a substantial support burden. Conversely, promoting the adoption of practices such as static analysis offers lasting long-term benefits beyond the migration effort.

LONG-TERM PROSPECTS

GUI strategy has been a challenge at CERN in recent years and reinforces the need to provide a stable and sustainable experience for software maintainers. On this occasion, PyQt6 was selected as the preferred framework, primarily due to resource constraints and the necessity to transition away from the unsupported Qt5. However, looking ahead, PySide is becoming an increasingly attractive option and may be considered for future adoption.

Although PyQt and PySide offer largely similar APIs, and both benefit from active user communities, they differ significantly in terms of governance and development models. PySide is maintained by The Qt Company, a well-resourced organisation, whereas PyQt is developed by Riverbank Computing Ltd, a small company led by a single developer. PySide follows a transparent and structured devel-

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THPD078

opment model, offering multiple channels for community engagement, published roadmaps, and publicly available minutes from internal decision-making meetings. This transparency facilitates clear insight into ongoing development efforts and short-term priorities. Additionally, the published Qt for Python compatibility matrix [7] would allow us to better plan the required internal Python versions support. In contrast, PyQt adheres to a more traditional and closed development approach, with communication limited to a single email-based channel. While external contributions are possible, there is little guidance or encouragement for community involvement, and no formalised contribution workflow is provided. Furthermore, engineers at The Qt Company have invested considerable effort in producing quality documentation, including Python-specific API references translated from the original Qt/C++ documentation. This is complemented by the inclusion of an Intersphinx object inventory, enabling seamless cross-referencing of Qt APIs from third-party documentation sources. In summary, PySide presents a lower long-term risk profile due to its institutional maintenance, structured governance, and broader development team, which also has potential for delivering new capabilities, such as QtAsyncio [6].

With the emergence of Qt7 expected in the coming years, it is prudent to begin planning for the next iteration of the GUI framework. One proactive step that can be taken at present is the introduction of abstraction layers for internally reusable components. This can be achieved by leveraging existing compatibility libraries such as *qtpy*, and by drawing on the experience of libraries such as *matplotlib*, *pyqtgraph* and *pytest-qt*, that have successfully maintained compatibility across multiple Qt bindings simultaneously.

SUMMARY

Decisions that appear straightforward on the surface often require substantial background research and technical evaluation. In this case, for transitioning to Qt6, PyQt6 was selected, a seemingly natural progression for the CERN community. However, this decision was informed by a comprehensive review of alternative solutions, a comparison between development paradigms in Qt Widgets and QtQuick, and extensive compatibility testing with widely used third-party libraries. Defining a robust GUI strategy has historically proven challenging within CERN's accelerators and technology sector, and any significant change warrants careful consideration. To facilitate a smooth migration to PyQt6, a suite of tools, libraries, and supporting guidelines has been developed. Nonetheless, it is becoming increasingly evident that achieving long-term stability may require a closer re-evaluation of PySide, along with a clear understanding of the effort required to adopt it across the existing codebase, including considering the contributing role that rapidly evolving AI capabilities can play.

REFERENCES

[1] P. J. Elson, C. Baldi, and I. Sinkarenko, "Introducing Python as a Supported Language for Accelerator Controls at CERN", in *Proc. ICALEPCS'21*, Shanghai, China, Oct. 2021, pp. 236-241. doi:10.18429/JACoW-ICALEPCS2021-MOPV040

[2] A. Radeva, F. Locci, J.M. Elyn, T. Oulevey, M. Vanden Eynden, "Selecting a Linux Operating System for CERN Accelerator Controls", in *Proc. ICALEPCS'23*, Cape Town, South Africa, Oct. 2023, pp. 1475-1479. doi:10.18429/JACoW-ICALEPCS2023-THPDP064

[3] E. Galatas, A. Asko , E. Matli and C. Roderick, "WRAP - A Web-Based Rapid Application Development", in *Proc. ICALEPCS'21*, Shanghai, China, Oct. 2021, pp. 894-898. doi:10.18429/JACoW-ICALEPCS2021-THPV013

[4] Qt for Python Missing Bindings, https://wiki.qt.io/Qt_for_Python_Missing_Bindings

[5] I. Sinkarenko, S. Zanzottera, and V. Baggiolini, "Our Journey from Java to PyQt and Web for CERN Accelerator Control GUIs", in *Proc. ICALEPCS'19*, New York, NY, USA, Oct. 2019, pp. 807-811. doi:10.18429/JACoW-ICALEPCS2019-TUCPR03

[6] PySide6.QtAsyncio, https://doc.qt.io/qtforpython-6/PySide6/QtAsyncio

[7] Qt for Python, https://wiki.qt.io/Qt_for_Python

[8] Status of Python versions, https://devguide.python.org/versions

[9] OpenRewrite by Moderne, https://docs.openrewrite.org/.

[10] Claude Code overview, https://docs.anthropic.com/en/docs/claude-code/overview

[11] LibCST, https://libcst.readthedocs.io/.

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2025). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.