Expense Tracker Application: A Comparative Language Analysis of C++ and Python Implementations

Parthasarathi Ponnapalli and Ashish Mahajan

Cross Language Application Development — Team 4

February 2026

Author Note

Parthasarathi Ponnapalli was responsible for the C++ implementation, documentation, and presentation. Ashish Mahajan was responsible for the Python implementation, packaging, documentation, and presentation. Both authors contributed equally to the design planning and comparison report.

Abstract

This report presents a comparative analysis of two implementations of a desktop expense tracking application — one developed in C++ using the Qt6 framework, and one developed in Python using PySide6. Both applications provide identical user-facing features: multi-user support, expense entry and editing, date and category filtering, a live summary panel with a custom pie chart, and JSON-based persistence. The implementations differ fundamentally in their data modelling approach, type system, memory management strategy, error detection timing, and development complexity. The C++ implementation leverages compile-time type safety, explicit memory ownership, and the RAII idiom to produce a deterministic and performant application. The Python implementation achieves the same functionality with substantially less code through dynamic typing, garbage-collected memory management, and expressive language features such as list comprehensions and the defaultdict container. This report discusses these differences in depth, includes representative code snippets, and reflects on the practical implications of each language's characteristics for desktop application development.

Keywords: C++, Python, Qt6, PySide6, cross-language development, type safety, memory management, MVC architecture

Expense Tracker Application: A Comparative Analysis of C++ and Python Implementations

Introduction

This report documents and compares two implementations of the same desktop application — an expense tracker — developed as part of the Cross Language Application Development group project. The application was built twice: once in C++ using the Qt6 framework, and once in Python using PySide6, the official Qt6 binding for Python. The project specification required both implementations to produce a fully functional desktop application with identical features, allowing a direct, controlled comparison of the two languages.

The choice of C++ and Python as the two languages under comparison is instructive precisely because they occupy opposite ends of several important axes: compiled versus interpreted, statically typed versus dynamically typed, manual memory management versus garbage-collected, and verbose-by-necessity versus expressive-by-default. These differences are not incidental — they shape every aspect of how a developer approaches a problem, from data modelling and storage to error handling and debugging strategy.

Both implementations share a common foundation: the same Qt Designer .ui files define the user interface once, the same .qss stylesheet provides visual styling, and the same JSON file format stores expense data, allowing the two applications to read each other's data. This deliberate sharing of infrastructure isolates the meaningful differences to the language-specific implementation logic, which is where this comparison focuses.

The report is organized as follows. First, it describes the application architecture and features. Second, it examines data representation and storage. Third, it compares memory management approaches. Fourth, it analyses type system and safety differences. Fifth, it discusses UI wiring and signal/slot patterns. Finally, it reflects on the overall language experience, including ease of use, performance characteristics, and the challenges encountered in each implementation.

Application Overview and Shared Architecture

The expense tracker application supports adding, editing, and deleting expense records; filtering by date range and category; switching between multiple named users; viewing a live summary panel that includes per-category totals and a visual breakdown; and exporting filtered data to a CSV file. Data persisted automatically to a local JSON file on every write operation.

Both implementations follow the Model-View-Controller (MVC) architecture, which maps naturally onto Qt's design philosophy. The model layer is responsible for all data storage and business logic; the view layer displays data without modifying it; and the controller layer responds to user input, coordinates between model and view, and triggers UI updates. Table 1 summarizes how each MVC layer is realized in the two languages.

Table 1

MVC Layer Realization in C++ and Python Implementations

Layer	C++ / Qt6	Python / PySide6
Model	ExpenseRepository holds std::vector<ExpenseRecord>. Private data, compiler-enforced encapsulation.	ExpenseManager holds list[dict[str, Any]]. Shape enforced by validate_expense() at entry time.
View	.ui files compiled at build time. ExpenseTableModel overrides QAbstractTableModel. PieChartWidget drawn with QPainter.	Same .ui files loaded at runtime via QUiLoader. ExpenseTableModel implemented in PySide6. Same QPainter-based chart.
Controller	MainWindow wires signals/slots using typed connect(). Single refresh() method is the only update point.	MainWindow uses @Slot decorators and .connect(). Equivalent refresh() logic applied after every mutation.
Persistence	QJsonDocument writes to QStandardPaths::AppLocalDataLocation. Saves on every write.	Python stdlib json.dump/json.load. Path resolved relative to __file__. Validated on load.

Note. Both implementations load the same .ui and .qss files from the shared/ directory. The JSON data format is identical and interoperable between the two applications.

A critical design decision in both implementations is that a single method — MainWindow::refresh() in C++ and an equivalent refresh() in Python — serves as the sole point at which the UI is updated. Every user action, whether adding an expense, deleting one, changing a filter, or switching users, ends by calling this method. This guarantees that the table and the summary panel are always consistent with the underlying data, and it simplifies debugging because there is only one code path responsible for UI synchronization.

Data Representation and Storage

C++: Typed Structs and STL Containers

In the C++ implementation, each expense is represented as a typed struct with explicitly declared fields. The struct includes a default constructor that initialises every field to a safe value, and an isValid() method that provides a single reusable correctness check. The following code illustrates this design.

```cpp
// expense_record.h — Snippet 1
struct ExpenseRecord {
    int     id;
    int     userId;
    QDate   date;
    double  amount;
    QString category;
    QString description;

    ExpenseRecord() : id(-1), userId(-1), amount(0.0) {}

    bool isValid() const {
        return id >= 0 && userId >= 0 && date.isValid()
               && amount >= 0.0 && !category.isEmpty();
    }
};
```

Snippet 1. C++ ExpenseRecord struct with default constructor and validation method.

All expenses are stored in a std::vector<ExpenseRecord> held as a private member of the ExpenseRepository class. The use of std::vector provides typed, contiguous storage — the container physically cannot hold any type other than ExpenseRecord. Category totals are accumulated into a std::map<QString, double>, which stores entries in sorted order by key, so category names always appear alphabetically in the pie chart legend and summary panel without any additional sorting step.

Python: Dicts and List Comprehensions

In the Python implementation, each expense is a plain dictionary with string keys. Python's dynamic typing means no field declarations are required, but it also means that a misspelled key or a missing field produces a KeyError at runtime rather than a compile error. To mitigate this, the ExpenseManager class provides a validate_expense() method that normalises and validates every expense dictionary as it enters the system.

```
# expense_manager.py — Snippet 2
def validate_expense(self, *, user, expense_date,
                     category, description, amount):
    return {
        'user':        self.validate_non_empty_string(user, 'user'),
        'date':        self.normalize_date(expense_date, 'date'),
        'category':    self.validate_non_empty_string(category, 'category'),
        'description': self.validate_non_empty_string(description,
'description'),
        'amount':      self.normalize_amount(amount),
    }
```

Snippet 2. Python validate_expense() ensures all fields are present and correctly typed before storage.

Filtering in Python is expressed as a list comprehension, which is more concise than the equivalent C++ for-loop. Category totals use a defaultdict(float), which initialises missing keys to zero automatically, eliminating the need for explicit key existence checks that the C++ std::map approach also avoids through its subscript operator. The two approaches are functionally equivalent but syntactically very different.

The fundamental difference is in when type errors surface. In C++, passing a value of the wrong type to a function expecting an ExpenseRecord causes a compile error before the program runs. In Python, a similar mistake is only discovered at runtime when the incorrect value is actually accessed, potentially much later in execution.

Memory Management

C++: Three Distinct Ownership Strategies

Memory management is one of the most significant areas of difference between the two implementations. The C++ implementation employs three distinct ownership strategies simultaneously, each appropriate to a different class of object.

The first strategy is stack allocation, used for the top-level objects in main(). When main() returns, the C++ runtime automatically calls the destructor of every stack object in reverse order — a pattern known as Resource Acquisition Is Initialization (RAII). The MainWindow destructor saves the data file, frees memory, and closes any open handles before the process exits. This cleanup is guaranteed and happens at a precise, predictable moment.

The second strategy is Qt's parent-child ownership system, used for all widget objects. When a widget is created with a parent, Qt's object tree takes ownership and deletes the child automatically when the parent is destroyed. This eliminates the need for manual deletion of most UI objects.

The third strategy is explicit new and delete, used for the three non-widget owned objects: m_ui, m_repo, and m_model. These are created with new in the constructor and deleted explicitly in the destructor. The following code illustrates all three strategies.

```
// main_window.h — Snippet 3
// Strategy A: manually owned — deleted in ~MainWindow()
Ui::MainWindow*    m_ui;
ExpenseRepository* m_repo;
```

```
ExpenseTableModel* m_model;

// Strategy B: Qt-parented — deleted automatically
QSortFilterProxyModel* m_proxy;   // parent = this
SummaryWidget*         m_summary;

// main.cpp — Strategy C: stack allocation (RAII)
int main(int argc, char* argv[]) {
    QApplication app(argc, argv);
    MainWindow window;  // no new, no delete
    window.show();
    return app.exec();
}  // destructors called here automatically
```
Snippet 3. Three memory ownership strategies used simultaneously in the C++ implementation.

An important constraint drove the decision to use raw pointers with manual delete rather than smart pointers. Qt-parented widgets must not be wrapped in std::unique_ptr or std::shared_ptr, because Qt and the smart pointer would both attempt to delete the same object, resulting in a double-free memory corruption. The correct Qt idiom is to give widgets a parent and allow non-widget objects to be owned manually.

Python: Garbage Collection

The Python implementation requires no explicit memory management whatsoever. CPython, the reference implementation, uses reference counting to track object lifetimes and frees memory automatically when the reference count of an object reaches zero. From the developer's perspective, objects are created when needed and reclaimed without any manual intervention.

The practical trade-off is determinism. In C++, the exact moment of cleanup is known: it occurs at scope exit, at the delete statement, or when the parent widget is destroyed. In Python, the garbage collector decides when to reclaim memory, and this may not happen immediately after the last reference is dropped. For a desktop application with modest memory usage, this is rarely a concern in practice, but it can matter for applications managing large data sets or scarce system resources.

Type System and Safety Features

The type system differences between C++ and Python extend beyond data representation to encompass encapsulation, function semantics, and inheritance safety. This section examines four C++ features that have no direct Python equivalent, and one Python feature that partially compensates.

const Correctness

In C++, the const keyword can be applied to both methods and parameters. A const method guarantees — enforced by the compiler — that it does not modify any member variables of the class. A const reference parameter passes an object without copying it but prevents the function from modifying it. The getExpenses() method in ExpenseRepository illustrates both uses.

```
// expense_repository.h — Snippet 4
// const on the method = cannot modify the repository
// const& on parameters = no copy, cannot be modified
std::vector<ExpenseRecord> getExpenses(
    int userId,
    const QDate& from,
    const QDate& to) const;
```

Snippet 4. const on the method and const& on parameters provide compile-time read-only guarantees.

Python has no equivalent mechanism. By convention, methods that do not modify state are distinguished only by documentation or naming, and there is no compiler enforcement. A Python function declared to read data can freely modify it without any warning.

The override Keyword

When a C++ class inherits from a base class and overrides a virtual function, the override keyword instructs the compiler to verify that the function signature exactly matches a virtual function in the base class. If the name is misspelled or the signature differs, the compiler produces an error immediately. Without override, the programmer would silently create a new, unrelated function instead of overriding the intended one — a subtle bug that may only manifest as wrong runtime behaviour.

```cpp
// expense_table_model.h — Snippet 5
class ExpenseTableModel : public QAbstractTableModel {
    Q_OBJECT
public:
    // 'override' = compiler verifies these match the base class
    int      rowCount   (const QModelIndex&) const override;
    int      columnCount(const QModelIndex&) const override;
    QVariant data       (const QModelIndex&, int) const override;
    QVariant headerData (int, Qt::Orientation, int) const override;
};
```
Snippet 5. The override keyword catches method signature mismatches at compile time.

Python has no equivalent. In PySide6, if the developer misspells data() as Data() when subclassing QAbstractTableModel, no error is raised. The table view will simply produce no output, and the developer must diagnose the problem at runtime.

Private Encapsulation

C++ enforces access control through the private keyword. Code outside the class cannot read or write private members; the compiler rejects such access unconditionally. This means that if the m_expenses vector in ExpenseRepository is corrupted, the fault is guaranteed to be inside the ExpenseRepository class — nowhere else. Python's convention of prefixing member names with an underscore signals intent but does not prevent external access.

Typed Containers

std::vector<ExpenseRecord> is a compile-time-typed container. Attempting to push a value of any type other than ExpenseRecord produces a compiler error. Python's list accepts values of any type, so inserting an incompatible value produces no immediate error; the mistake becomes visible only when the incorrect value is later accessed.

Python Type Hints

Python 3.5 introduced optional type annotations, and PySide6 makes use of them throughout. The Python implementation uses annotations such as list[dict[str, Any]] and str | None. These hints help IDEs and static analysis tools identify likely type errors before runtime, partially closing the gap with C++'s compile-time checking. However, Python does not enforce type hints at runtime — they are advisory only.

Table 2 summarizes the type safety comparison across both implementations.

Table 2 *Type and Safety Feature Comparison*

Feature	C++ / Qt6	Python / PySide6
Typed containers	Enforced at compile time	Not enforced; type hints advisory only
Private members	Compiler blocks all external access	_name convention; not enforced
const correctness	Compile-time guarantee	No equivalent
override safety	Compile-time signature check	No equivalent; silent wrong name
Error detection	Compile time — before execution	Runtime — when bad value accessed

Note. Type hints in Python (PySide6) improve IDE support but are not enforced at runtime.

User Interface Architecture and Signal/Slot Wiring

UI File Loading

Both applications use the same Qt Designer .ui files to define the window layout and dialog layout. The difference lies in when these files are processed. In C++, the uic tool (part of Qt's build toolchain) compiles the .ui file into a C++ header at build time. This generated header defines a Ui:: class with typed member pointers for every widget in the file. A missing widget name or a misspelling in the .ui file produces a linker or compile error.

In Python, QUiLoader loads the .ui file at runtime and produces a widget tree. Individual child widgets are accessed by name using findChild(). A misspelled widget name returns None at runtime rather than an error at compile time, so the mistake is only discovered when the None object is later used.

```python
# add_expense_dialog.py — Snippet 6
loader = QUiLoader()
dialog_widget = loader.load(ui_file)

# findChild returns None if name is wrong — no compile-time check
self.categoryLineEdit = dialog_widget.findChild(
    QLineEdit, 'categoryLineEdit')

if self.categoryLineEdit is None:
    raise RuntimeError('categoryLineEdit not found in .ui')
```

Snippet 6. Python loads .ui at runtime and uses findChild() with explicit None checks.

Signal/Slot Connections

Qt's signal/slot mechanism connects user interface events to handler functions. In C++, the type-safe connect() syntax introduced in Qt5 uses pointer-to-member-function syntax. The compiler verifies that the signal and slot signatures are compatible. An incompatible connection, or a misspelled method name, produces a compile error.

```cpp
// main_window.cpp — Snippet 7
// Type-safe: compiler verifies signatures match
connect(actAdd,
        &QAction::triggered,
        this,
        &MainWindow::onAdd);

connect(m_ui->tableView, &QTableView::doubleClicked,
        this, &MainWindow::onEdit);
```

Snippet 7. C++ type-safe signal/slot connections verified at compile time.

In Python, slots are marked with the @Slot() decorator and connections are made using .connect(), which accepts any callable. This is more flexible but less safe: a misspelled method name, or a method that does not match the expected signature, is only discovered when the signal fires at runtime.

Challenges, Language Experience, and Reflection

C++ Challenges

The most significant challenge in the C++ implementation was navigating Qt's memory ownership model. The original design plan specified smart pointers (std::unique_ptr) for all dynamically allocated objects, following modern C++ best practice. However, this approach caused double-free crashes when applied to Qt-parented widgets, because Qt's parent-child system and the smart pointer destructor both attempted to delete the same object. The resolution required understanding two distinct ownership systems — RAII smart pointers for non-Qt objects, and Qt parent-child ownership for widgets — and applying each correctly.

The data file path was a second challenge. An initial approach used a path relative to the build directory, which broke whenever the application was launched from a different working directory. The correct solution was Qt's QStandardPaths::AppLocalDataLocation, which resolves to a per-user, platform-appropriate directory regardless of working directory. This required understanding a Qt API that has no direct Python equivalent.

Implementing QAbstractTableModel required overriding five virtual methods and understanding Qt's role-based data model. The boilerplate is significantly heavier than the equivalent Python implementation. However, the override keyword caught at least one case during development where a method name was spelled incorrectly, preventing a runtime debugging session.

Python Challenges

The primary challenge in the Python implementation was ensuring that the dynamic nature of Python did not introduce subtle bugs that would be difficult to diagnose. Because Python dicts have no fixed shape, a missing key in an expense record would only manifest as a KeyError at runtime, potentially in a code path that is rarely exercised. The validate_expense() method was introduced to address this by enforcing the required shape at the point of data entry, converting the problem from a runtime error deep in the application to an explicit ValueError at the boundary.

A second challenge was ensuring that the packaged PyInstaller executable could locate the shared .ui and .qss files at runtime. During development, these files are found relative to the source directory. Inside a packaged binary, the layout is different, and paths resolved via __file__ no longer work as expected. A resource_path() helper function was introduced to resolve paths correctly in both contexts.

A subtlety in PySide6's QAbstractTableModel implementation is that the data() method must return None for unhandled display roles rather than raising an exception. This differs from what a Python developer might intuitively expect and is easy to overlook when first working with the framework.

Ease of Use

Python required substantially less code to implement the same functionality. A comparison of equivalent logic in the two languages consistently favoured Python for conciseness. List comprehensions replaced explicit for-loops; defaultdict replaced manual key-existence checks; json.dump replaced a multi-step QJsonDocument construction. The Python implementation was also faster to iterate on during development, because the lack of a compilation step meant that any change could be tested immediately.

C++ offered advantages in a different dimension: confidence. Every compile success was a meaningful guarantee that no type errors, no missing fields, and no incorrect method overrides were present. The compile-time feedback loop, though slower, caught real bugs before they reached testing.

Performance

For a desktop application of this scale, both implementations performed acceptably. No measurable performance difference was observed in UI responsiveness or file I/O. For production applications that process large volumes of data, C++ would offer advantages through its tighter memory layout (contiguous struct arrays rather than heap-allocated Python dicts), absence of garbage collector pauses, and zero-overhead abstractions. The Python implementation, conversely, would likely be more maintainable as feature complexity grows, due to its conciseness and the availability of a broader third-party ecosystem.

Language Characteristics and Approach

The two implementations reflect a fundamental difference in philosophy. C++ requires the developer to be explicit about types, ownership, and lifetime. This explicitness is a burden during initial development but becomes a form of documentation: a reader of the C++ code can determine from the header alone which methods modify state (those without const), which objects are owned manually (those without a Qt parent), and what types every container holds. The language's strictness enforces architectural discipline.

Python prioritises developer productivity and expressiveness. The same level of architectural discipline can be achieved, but it requires deliberate effort — type hints, convention, and tools like validate_expense() — rather than compiler enforcement. The result is code that is shorter, more readable, and faster to write, at the cost of moving some safety guarantees from compile time to runtime.

Conclusion

This report has examined the C++ and Python implementations of the Expense Tracker application across five dimensions: data representation, storage, memory management, type safety, and UI architecture. The two implementations produce identical user-facing behaviour using a shared UI definition, a shared data format, and a shared MVC architecture, while differing fundamentally in almost every aspect of implementation.

The C++ implementation shifts the majority of safety guarantees to compile time. Typed structs, const correctness, private access control, typed containers, and the override keyword collectively ensure that a large class of errors — wrong types, missing fields, incorrect method overrides, and unintended mutations — are detected before the application runs. Memory management requires conscious attention but yields deterministic, predictable resource cleanup through RAII and Qt's parent-child system.

The Python implementation achieves the same functionality with less code and less ceremony. Dynamic typing and garbage collection remove entire categories of concern from the developer's attention. List comprehensions, defaultdict, and Python's standard library simplify filtering and persistence logic. The trade-off is that safety guarantees are enforced at runtime rather than compile time, and the developer must compensate through discipline, validation, and testing.

The project demonstrated that both languages are capable of producing a complete, polished desktop application using the Qt framework. The choice between them for a real project would depend on context. For applications where correctness guarantees, performance, and deterministic resource management are paramount — embedded systems, financial software, real-time applications — C++ is the stronger choice. For applications where developer productivity, rapid iteration, and ease of maintenance are the primary concerns — internal tools, data analysis applications, rapid prototypes — Python offers a compelling advantage.

The cross-language comparison also illuminated how much of a language's design philosophy is visible in day-to-day code. C++ communicates its requirements through syntax: a function that takes a const reference and is marked const cannot harm the object it reads, by compiler decree. Python communicates the same information through convention and documentation, relying on the developer's discipline rather than the compiler's enforcement. Both approaches produce working software; they differ in where the responsibility for correctness lies.

References

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software.* Addison-Wesley.

International Organization for Standardization. (2020). *ISO/IEC 14882:2020 — Programming languages — C++*. ISO. https://www.iso.org/standard/79358.html

Python Software Foundation. (2024). *Python 3 documentation.* https://docs.python.org/3/

Qt Group. (2024). *Qt 6 documentation: QAbstractTableModel.* https://doc.qt.io/qt-6/qabstracttablemodel.html

Qt Group. (2024). *Qt 6 documentation: Memory management.* https://doc.qt.io/qt-6/objecttrees.html

Qt Group. (2024). *PySide6 documentation: QUiLoader.* https://doc.qt.io/qtforpython-6/PySide6/QtUiTools/QUiLoader.html

Stroustrup, B. (2013). *The C++ programming language* (4th ed.). Addison-Wesley.

van Rossum, G., Warsaw, B., & Coghlan, N. (2001). *PEP 8 — Style guide for Python code.* Python Software Foundation. https://peps.python.org/pep-0008/