Expense Tracker Application: Deliverable 3: Final Features, Documentation, and Presentation

Sri Sai Palamoor, Oishani Ganguly

Department of Computer and Information Science

University of the Cumberlands

Executive Master’s in Computer Science

Professor Jay Thom

October 26, 2025

GitHub Repository: https://github.com/spalamoor39148/MSCS-632-M80-GroupProject

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

Abstract

This report presents a comprehensive comparative analysis of a cross-language command-line expense tracker application implemented in both Python and C++. The purpose of the project was to design, implement, and evaluate identical functionalities in two fundamentally different languages to highlight how language paradigms affect development style, performance, syntax, and system design. Each application includes identical features - adding, viewing, filtering, deleting, and summarizing expenses, as well as persistent data storage in CSV/JSON formats. However, the implementations differ greatly in structure and complexity because of each language's characteristics. Python's dynamic typing and expressive standard library enable concise, readable code, while C++ provides explicit control over types, memory, and performance through the Standard Template Library (STL). This report analyzes how these contrasting approaches influenced modular design, error handling, file I/O, and developer experience. It concludes that Python promotes rapid prototyping and ease of use, whereas C++ emphasizes robustness and fine-grained optimization - together illustrating how language paradigms shape software architecture and engineering strategy.

1. Introduction

The Expense Tracker project was designed to explore how two distinct programming paradigms - Python's dynamically-typed interpreted model and C++'s statically-typed compiled model - handle identical functional requirements within a controlled environment. By intentionally avoiding GUI complexity, the project maintains a narrow focus on language fundamentals rather than framework-specific concerns. Both implementations share seven core features - Add Expense, View Expenses, Delete Expense, Filter Expenses, Summary Report, Data Persistence, and File Import/Export - implemented through a modular architecture comprising input handling, data storage, business logic, and output formatting modules.

2. Overview of the Expense Tracker Application

Both versions provide users with tools to record daily expenditures, filter or summarize those entries, and persist the information across sessions. Each entry contains a date (YYYY-MM-DD), amount, category (e.g., Food, Housing, Transportation), and description. The modular design is divided into four components:

1. **Input Handler**: Performs user interaction and input validation.
2. **Storage/Persistence Module**: Maintains in-memory structures and manages file I/O.
3. **Processing/Logic Module**: Implements filtering and aggregation algorithms.
4. **Output Formatter**: Displays lists and summary reports in a consistent CLI layout.

Although both systems exhibit functional parity, their internal designs reflect the idiomatic conventions of their respective languages. In Python, the team used high-level built-ins such as lists, dictionaries, and the `json` library; in C++, explicit class design, STL containers, and manual file streams were employed.

3. Python Implementation Analysis

The Python implementation, found in `python_implementation/expense_tracker`, delivers a concise, high-level approach to the Expense Tracker system. Its structure emphasizes clarity, modularity, and ease of iteration through dynamic typing and built-in library support. Key characteristics include:

- **Architecture and Data Model**:
  - Uses a simple list of dictionaries to represent expenses.
  - Each expense includes fields for date, amount, category, and description.
  - Avoids class overhead, instead relying on direct manipulation of native types.
- **Core Functionality**:
  - Implements CRUD features through straightforward functions (`add_expense`, `view_expenses`, `delete_expense`, etc.).
  - Employs list comprehensions and built-in functions like `filter()` for readable and concise iteration.
  - Uses the `datetime` library for validation of date inputs.
- **Persistence and I/O**:
  - Relies on Python's `json` module for serialization and deserialization.
  - Automatically loads and saves data using context managers (`with open(...)`) to ensure safe file handling.

- **Error Handling and Validation**:
    - Employs `try/except` blocks to manage invalid numeric or date input dynamically.
    - Prints clear, user-friendly feedback messages during input errors.
- **Testing and Maintainability**:
    - Uses the `pytest` framework to automate testing of both CLI and logic components.
    - Modular separation ensures that core business logic remains testable without direct CLI interaction.

Overall, the Python version demonstrates Python's strengths: rapid development, minimal boilerplate, and intuitive syntax. It achieves all project requirements with limited code complexity, prioritizing developer productivity and user experience over low-level optimization.

4. C++ Implementation Analysis

The C++ implementation, located in `c++_impementation/expense_tracker_src`, illustrates a disciplined, performance-oriented design that leverages the Standard Template Library (STL) and object-oriented principles. Unlike Python's dynamically typed design, this version emphasizes strong typing, memory safety, and explicit data control.

- **Architecture and Data Model**:
    - Defines an `Expense` class encapsulating fields for date, amount, category, and description.

  - Stores objects in an `std::vector<Expense>` container for efficient dynamic array management.
  - Separates responsibilities into files (`Expense.cpp`, `FileManager.cpp`, `Utils.cpp`), enhancing modularity.
- **Core Functionality**:
  - Implements add, view, delete, and filter features using STL algorithms such as `std::copy_if` and `std::remove_if`.
  - Aggregates totals by category using `std::unordered_map<std::string, double>`.
  - Maintains a menu-driven CLI that guides users through numbered options.
- **Persistence and File I/O**:
  - Utilizes `std::ofstream` and `std::ifstream` for reading and writing expense data in CSV format.
  - Parses input manually using `std::stringstream` and converts strings to numeric values via `std::stod`.
  - Requires explicit file closure and validation checks for I/O integrity.
- **Error Handling and Validation**:
  - Employs try/catch blocks with standard exceptions such as `std::invalid_argument`.
  - Incorporates `std::cin.fail()` checks to detect invalid stream input during runtime.
- Testing and Compilation:
  - Includes a custom assert-based test harness to validate core operations.

- The provided Makefile automates compilation and linking, ensuring consistent builds and rapid testing.

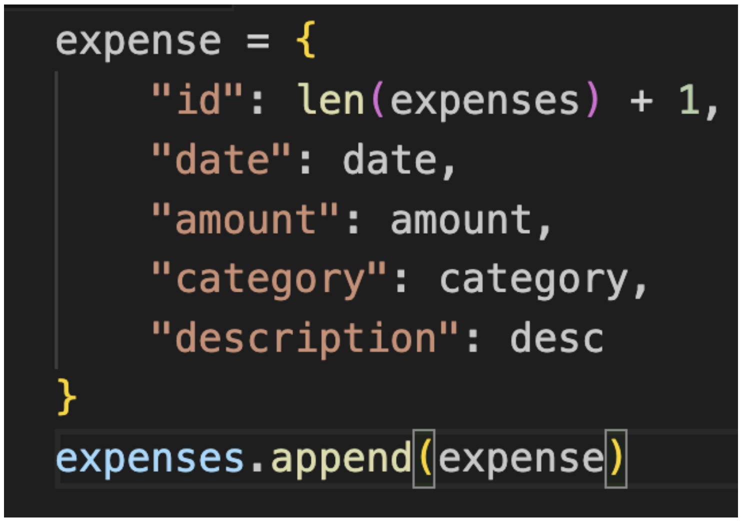
While more verbose than its Python counterpart, the C++ implementation provides deterministic memory control, compile-time error detection, and superior runtime efficiency. It exemplifies the rigor and predictability of a statically typed language and demonstrates how robust structure and explicit control yield long-term maintainability and performance benefits.

5. Comparative Analysis

5.1 Core Functionality and Data Structures

Both implementations deliver identical functionality: adding, viewing, deleting, filtering, and summarizing expenses, with persistent storage across sessions.

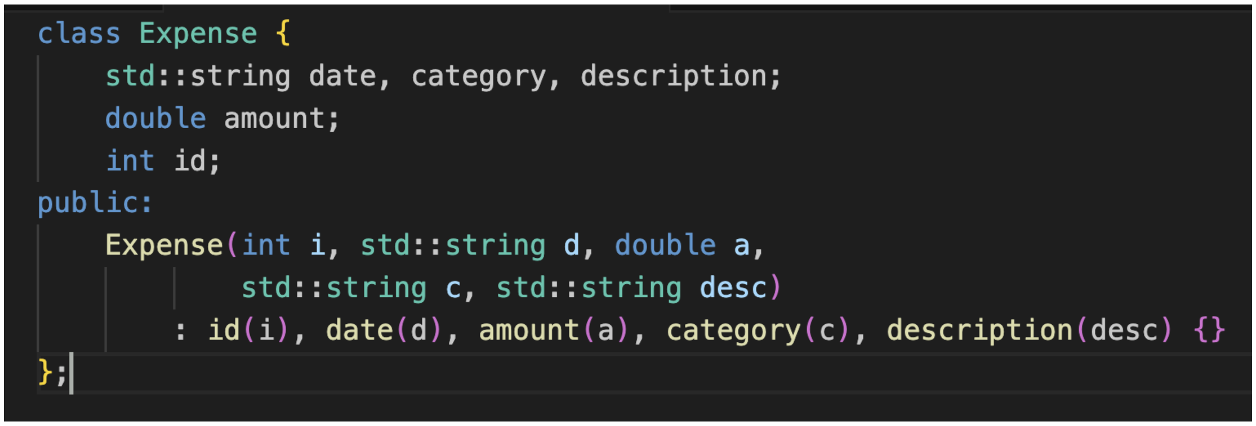
In the Python version, an expense is a simple dictionary appended to a list:



```python
expense = {
    "id": len(expenses) + 1,
    "date": date,
    "amount": amount,
    "category": category,
    "description": desc
}
expenses.append(expense)
```

This flexible structure uses dynamic typing and eliminates class boilerplate. The in-memory list directly maps to JSON persistence.

In C++, a strongly typed class model was implemented:

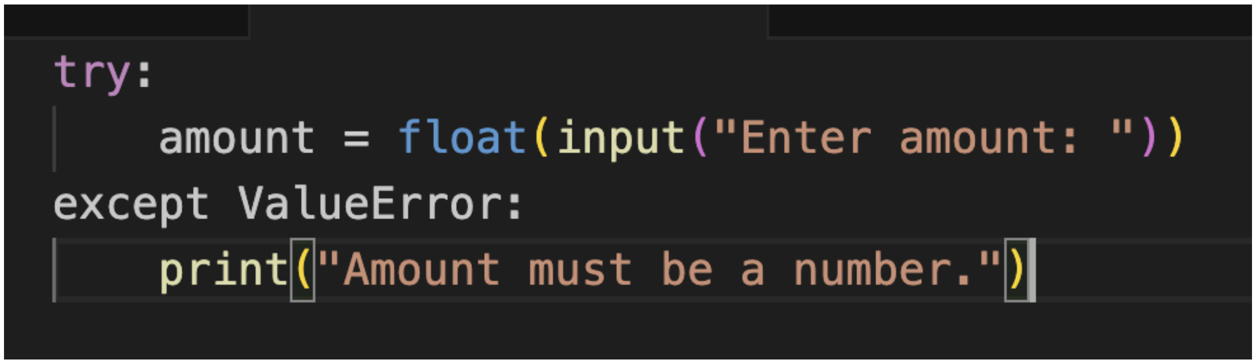


```cpp
class Expense {
    std::string date, category, description;
    double amount;
    int id;
public:
    Expense(int i, std::string d, double a,
            std::string c, std::string desc)
        : id(i), date(d), amount(a), category(c), description(desc) {}
};
```

Here, `std::vector<Expense>` replaces Python's list, reflecting static typing and explicit memory control. While Python emphasizes convenience, C++ ensures type safety and deterministic memory behavior.

5.2 Input Handling, Validation, and Error Control

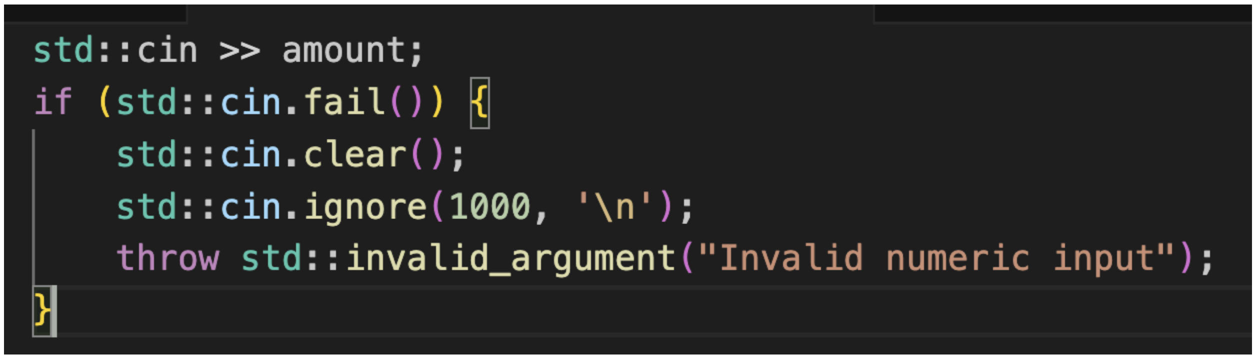
User input in both languages follows a structured CLI prompt cycle but differs in how validation is enforced. The Python approach leverages runtime exception handling:



```python
try:
    amount = float(input("Enter amount: "))
except ValueError:
    print("Amount must be a number.")
```

Python's `try/except` dynamically traps type errors during conversion, allowing smooth user recovery.

In C++, input validation is coupled with stream error detection and typed exceptions:



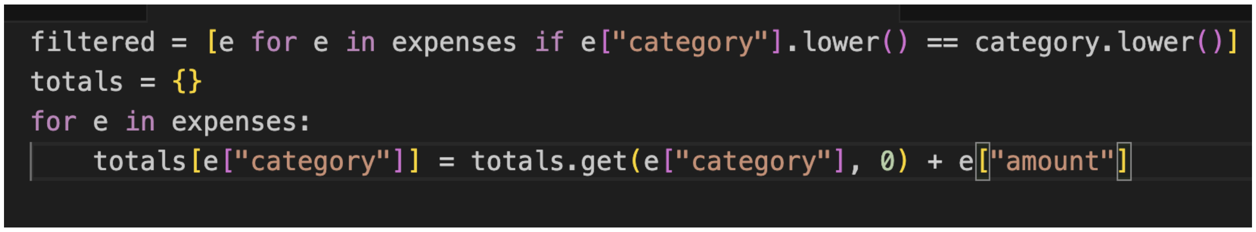
```cpp
std::cin >> amount;
if (std::cin.fail()) {
    std::cin.clear();
    std::cin.ignore(1000, '\n');
    throw std::invalid_argument("Invalid numeric input");
}
```

C++ requires explicit stream checks and exception propagation. This distinction reflects the deeper philosophical divide: Python relies on runtime safety nets; C++ demands compile-time and runtime discipline.

Both implementations also validate date strings. Python's version calls `datetime.strptime` to confirm the YYYY-MM-DD format, while C++ uses `<ctime>` parsing, a more verbose but precise solution.

5.3 Filtering, Summarization, and Business Logic

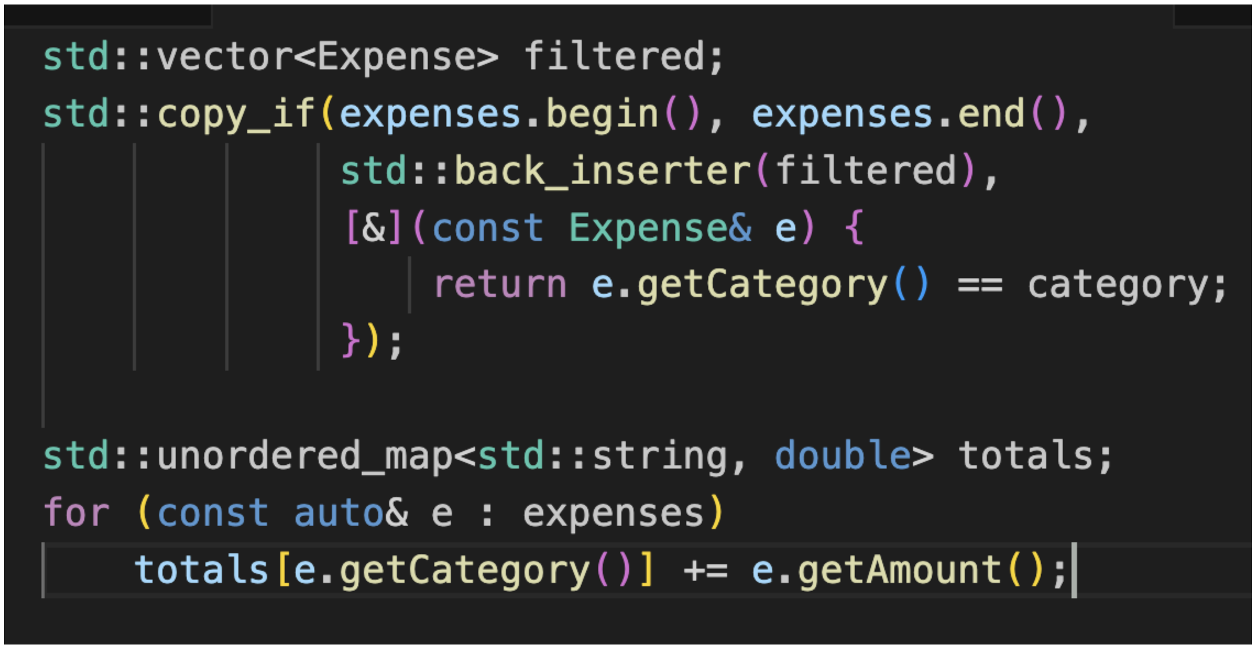
Filtering and summary reports illustrate how each language handles in-memory computation. In Python, list comprehensions and dictionaries achieve concise filtering and aggregation:



```python
filtered = [e for e in expenses if e["category"].lower() == category.lower()]
totals = {}
for e in expenses:
    totals[e["category"]] = totals.get(e["category"], 0) + e["amount"]
```

This compact syntax hides iteration details, producing readable yet powerful expressions.

In C++, equivalent logic employs STL algorithms and containers:



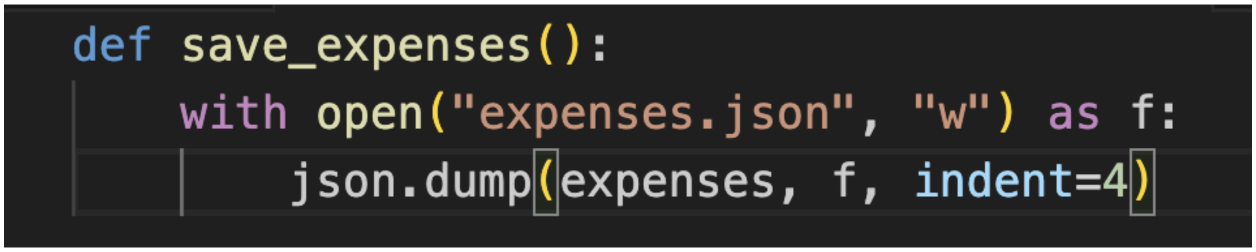
```cpp
std::vector<Expense> filtered;
std::copy_if(expenses.begin(), expenses.end(),
             std::back_inserter(filtered),
             [&](const Expense& e) {
                 return e.getCategory() == category;
             });

std::unordered_map<std::string, double> totals;
for (const auto& e : expenses)
    totals[e.getCategory()] += e.getAmount();
```

The C++ code is longer but faster at runtime. Both implementations produce identical console summaries, yet Python's simplicity contrasts with C++'s explicit iterator management. The difference underscores Python's abstraction power versus C++'s control and predictability.

5.4 Data Persistence and File Management

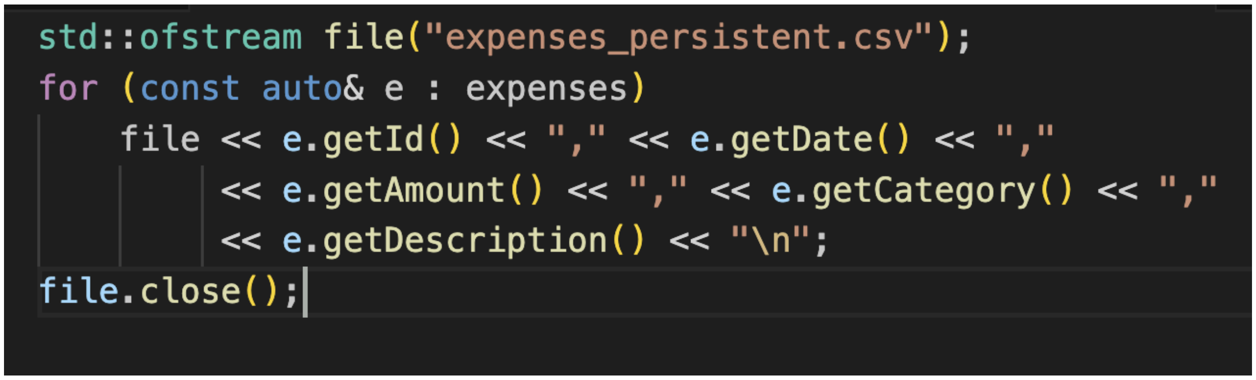
Persistent storage revealed the sharpest contrast in implementation strategy. Python uses JSON serialization through the standard library:



```python
def save_expenses():
    with open("expenses.json", "w") as f:
        json.dump(expenses, f, indent=4)
```

This single function call automatically converts nested data structures to a persistent file. Reloading is equally straightforward using `json.load`.

The C++ persistence layer, implemented in `FileManager.cpp`, requires explicit file-stream management:



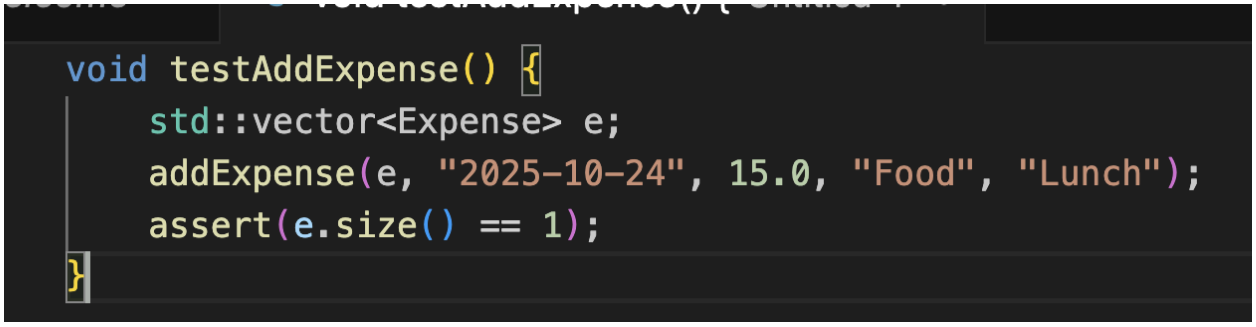
```cpp
std::ofstream file("expenses_persistent.csv");
for (const auto& e : expenses)
    file << e.getId() << "," << e.getDate() << ","
         << e.getAmount() << "," << e.getCategory() << ","
         << e.getDescription() << "\n";
file.close();
```

Loading re-parses each line with `std::stringstream` and converts strings to doubles via `std::stod`. The C++ approach gives finer control over formatting and I/O errors but requires significantly more code. Python's JSON system demonstrates high-level abstraction, while C++ emphasizes manual precision and portability.

5.5 Testing, Execution Flow, and Observed Behavior

Testing and runtime execution differed as much as syntax. The Python version integrates with `pytest`, executing automated test files (`test_expense_manager.py`) using a single command: `python3 -m pytest tests/.` This framework automates discovery and reporting, validating both logic and CLI prompts.

The C++ implementation uses a custom assertion-based harness compiled into a standalone binary:


```
void testAddExpense() {
    std::vector<Expense> e;
    addExpense(e, "2025-10-24", 15.0, "Food", "Lunch");
    assert(e.size() == 1);
}
```

The Makefile automates compilation: `make && ./expense_tracker`

Observed runtime differences were clear: Python loaded JSON instantly and displayed formatted summaries with minimal delay, while C++ parsed CSV line by line but executed numeric operations faster once loaded. Python handled invalid input more gracefully through printed messages; C++ provided stronger guarantees via typed exceptions and deterministic error exits.

Both codebases produced equivalent outputs and passed all core scenarios - filtering by category, summarizing totals, persisting after exit, and merging data from external CSV/JSON files as confirmed in Deliverable 2 screenshots.

6. Challenges and Lessons Learned

During development, the team encountered several challenges directly tied to language paradigms.

- **Typing Disparities**: Dynamic typing in Python led to occasional runtime conversion errors when users entered invalid data, mitigated through defensive coding. In C++, compilation errors surfaced early, but syntax rigidity slowed initial progress.

- **Date Handling**: Python's datetime module simplified date parsing and comparisons. In C++, implementing date filters using `<ctime>` or `<chrono>` involved substantial boilerplate.
- **File Persistence**: Python's `json.dump()` completed serialization in one statement, whereas C++ demanded explicit stream management and data formatting.
- **Testing and Integration**: Python's automated tests executed seamlessly with `pytest`, while the C++ binary required manual rebuilding for each test case.

Despite these challenges, both implementations achieved full functional parity, demonstrating the adaptability of software design across paradigms. The project reinforced that productivity, maintainability, and performance are not mutually exclusive but are instead influenced by a language's ecosystem, syntax, and design philosophy.

7. Reflection

Developing the Expense Tracker in Python underscored the language's appeal for rapid application prototyping and data-centric problem solving. The concise syntax, dynamic typing, and powerful built-in libraries allowed for fast feature delivery without sacrificing readability. Implementing validation, filtering, and persistence required minimal code - each achievable through a single expressive statement or a short function. The experience highlighted Python's accessibility for collaborative development, where clear structure and modularity can be achieved without verbose class hierarchies. However, runtime type errors and dependency on implicit conversions demanded disciplined input validation. Although Python abstracted away concerns like memory and file buffer management, this abstraction also meant reduced control over execution performance. Overall, Python provided a productive and enjoyable development environment, making it particularly suitable for proof-of-concept systems and educational projects where flexibility and simplicity are valued over raw performance.

Implementing the same system in C++ revealed a contrasting development philosophy centered on precision, explicitness, and performance. Every design choice - from data typing to memory management - required deliberate planning. The need to declare data types, manage input streams, and handle exceptions explicitly reinforced a strong understanding of how data flows through memory and the operating system. Compared to Python’s ease of iteration, development in C++ was slower and more demanding, but it provided a deeper appreciation for compile-time guarantees, deterministic memory behavior, and the efficiency of STL containers. Once implemented, the resulting binary executed faster and handled larger data volumes more predictably. This implementation experience emphasized how C++ empowers developers with control and efficiency at the cost of verbosity and steeper learning curves. The process ultimately strengthened skills in modular design, algorithmic efficiency, and disciplined resource handling - key competencies in system-level software engineering.

8. Conclusion

This comparative study revealed that while both Python and C++ can implement identical business logic, their distinct paradigms lead to divergent development experiences. Python’s concise syntax, dynamic typing, and high-level abstractions support rapid prototyping and ease of testing. C++, conversely, enforces rigorous type safety, explicit memory control, and performance optimization suitable for production-grade systems.

The Expense Tracker project successfully illuminated how language features shape architecture, data modeling, and error handling strategies. Python demonstrates accessibility and readability ideal for academic and small-scale projects, while C++ excels in performance-critical applications where predictability and efficiency are paramount. Together, they illustrate a broader lesson: language selection profoundly influences not only the final codebase but also the developer mindset and software engineering workflow.

9. References

ISO/IEC. (2024). *ISO/IEC 14882:2024: Programming languages — C++. Edition 7*. International Organization for Standardization. Retrieved from https://www.iso.org/standard/83626.html.

Microsoft. (2025). *C++ Standard Library reference (STL)*. Retrieved from https://learn.microsoft.com/en-us/cpp/standard-library.

Python Software Foundation. (2025). *Python 3.14.0 documentation*. Retrieved from https://docs.python.org/3/.

9.1 Acknowledgment

Portions of the analysis and comparative synthesis in this report were supported by insights generated through ChatGPT (GPT-5), developed by OpenAI. The tool was used to summarize code behavior, verify repository content, and refine academic language for clarity and coherence. All interpretations and conclusions are the authors' own.