Expense Tracker Application: Deliverable 1: Initial Planning and Cross-Language Design Specification

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 24, 2025

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

1. Project Introduction and Goal Definition

This document serves as the formal design and planning specification for the Expense Tracker application. The goal is to design and implement an identical application in two contrasting programming languages - Python (high-level, dynamically-typed) and C++ (lower-level, statically-typed) - to facilitate a detailed comparative analysis.

The project emphasizes how different language paradigms influence design, implementation efficiency, and the utilization of language-specific features, such as memory management, data structures, and error handling. The application will be realized as a Command-Line Interface (CLI) tool in both languages. This choice is crucial, as it intentionally reduces complexity related to graphical user interface (GUI) frameworks, allowing the team to concentrate the comparison on the core functional and performance aspects of each language.

2. Application Design and Requirements

2.1. Core Application Functionality

The Expense Tracker application is designed to provide users with essential personal financial management tools, focusing on data entry, persistence, retrieval, and analysis. Both the Python and C++ implementations will meet the following functional requirements:

1. **Add Expense:** Secure input and validation for new expenses, requiring fields for: date (YYYY-MM-DD), amount (numeric), category (pre-defined list: Housing,

Transportation, Food & Dining, Utilities & Communication, Healthcare & Insurance, Personal & Debt, Other), and a description (text).

2. **View Expenses:** Display all stored records in a readable, chronological, and formatted list.
3. **Filter Expenses:** Implement advanced filtering capabilities by date range (start and end date) or a specific category.
4. **Summary Report:** Generate an aggregated report showing the total expenses overall and a breakdown of total expenses per category.
5. **Data Persistence:** Implement functionality to save all in-memory expense data to a file and load data upon application start to maintain continuity.

2.2. Modular Architectural Structure

A modular architecture will be employed in both implementations to promote code organization, maintainability, and clear separation of concerns. The system will be logically divided into four distinct components:

1. **Input Handler:** Manages all user interactions and ensures data integrity through necessary validation checks before data is passed to the storage module.
2. **Storage/Persistence Module:** Maintains the in-memory data structure (holding expense objects/structs) and handles the reading and writing of data to external files (JSON/CSV).
3. **Processing/Logic Module:** Contains the primary business logic, including the filtering algorithms (date range comparison, category matching) and the aggregation logic for summary reports.

4. **Output Formatter:** Renders the application's responses, lists, and computed summary reports in a clean, consistent, and user-friendly CLI format.

2.3. Language-Specific Design Commitments

The following design decisions commit the team to leveraging the unique and idiomatic features of the assigned languages, ensuring a meaningful basis for the final comparison report.

Feature	**Python Implementation (Idiomatic)**	**C++ Implementation (Idiomatic)**
Data Storage	A list of dictionaries is chosen to capitalize on Python's dynamic typing and flexible data structure creation.	A well-defined C++ class (Expense) will be used, with instances stored within the std::vector<Expense> Standard Template Library (STL) container.
Data Processing	Utilize list comprehensions and built-in higher-order functions like filter() for concise and readable data manipulation.	Implement filtering and searching using explicit iteration combined with powerful STL algorithms such as std::copy_if and iterators.
Summary Calculation	Employ the collections.defaultdict or standard dictionary operations for simple and efficient grouping and aggregation of totals by category.	Use the std::unordered_map<std::string, double> container for high-performance key-value mapping to calculate category totals.

Date Handling	Leverage the robust datetime standard library module for reliable parsing, validation, and comparison of date inputs.	Manage dates using the C++ standard library's <ctime> header and its relevant structures (e.g., std::chrono) for precise time-based operations.
Error Handling	Employ concise try-except blocks for structured error management during I/O operations and input validation (e.g., handling non-numeric amounts).	Implement robust error control using explicit try-catch blocks, focusing on the appropriate use of standard exceptions (e.g., std::invalid_argument).
Data Persistence	Use the built-in json library for structured data serialization and deserialization.	Utilize file streams (fstream, ofstream, ifstream) to manage persistent data storage, likely in a structured CSV format.

3. Task Assignment and Collaboration Plan

The group has adopted a specialized yet collaborative approach, assigning primary language development responsibilities based on individual strengths while maintaining a shared commitment to quality and documentation. All code will be managed on a collaborative GitHub repository.

Team Member	Primary Language Focus	Key Responsibilities
Oishani Ganguly	Python	Leads the full Python implementation. Responsible for core logic, documentation adherence (APA), and the initial draft of the comparison report.
Sri Sai Palamoor	C++	Leads the full C++ implementation. Responsible for memory management design, compilation, system testing, and performance profiling.
Shared Responsibilities	Both	Daily code commits, peer review, consistency checks between implementations, preparing the final presentation slides, and finalizing the comparison report.

4. Project Timeline and Milestones

This three-day project necessitates a rigorous timeline, with specific milestones established to ensure timely completion of all deliverables.

Day	Date (2025)	Key Objectives	Deliverable
Day 1	Friday, Oct 24	Finalize architecture. Assign roles. Initialize GitHub repository. Complete and submit this Planning and Design Document.	**Deliverable 1**
Day 2	Saturday, Oct 25	Implement core functionality in both languages (Add, View, Filter, Summary). Integrate language-specific features. Conduct basic unit testing and commit operational code.	**Deliverable 2**
Day 3	Sunday, Oct 26	Complete all remaining features, specifically data persistence (save/load). Conduct final, comprehensive system testing. Complete the comparison report and finalize presentation slides.	**Deliverable 3**

5. Anticipated Cross-Language Challenges

Successful completion of this project hinges on effectively managing the fundamental differences between Python and C++. The primary challenges are categorized below, providing a roadmap for focused debugging and comparison efforts during the implementation phase.

5.1. System and Data Typing Disparities

- **Dynamic vs. Static Typing**: The most significant difference is the contrast between Python's dynamic typing and C++'s static typing.

    - In Python, the challenge lies in ensuring runtime data integrity. Since type checking occurs during execution, strict input validation and defensive coding are necessary to prevent crashes (e.g., trying to perform arithmetic on a string).
    - In C++, the initial challenge is verbosity and compilation errors. Every data type must be explicitly defined (e.g., std::string, double, int), which requires more development time up front but mitigates most type-related errors at the compilation stage.

5.2. Memory and Resource Management

- **Explicit vs. Automatic Memory**: C++ necessitates a deliberate approach to memory management. While we will primarily use STL containers (e.g., std::vector) to leverage RAII (Resource Acquisition Is Initialization), careful attention must be paid to scope and object lifetimes to prevent memory leaks or dangling pointers.
- **Python's Garbage Collection**: While Python's automatic garbage collector simplifies development, it can obscure memory usage inefficiencies. The challenge here is ensuring the high-level Python code is not unnecessarily resource-intensive, even if the memory is managed for us.

5.3. I/O and External Library Integration

- **Date Handling**: Implementing reliable date range filtering will present a challenge due to the distinct approaches. Python's datetime module is intuitive, but C++ requires navigating the more complex <ctime> or <chrono> libraries for reliable comparison and parsing from string inputs.

- **Data Persistence**: File serialization will require different strategies. Python's use of the high-level json library for structured data is straightforward. In contrast, the C++ implementation will require manual reading and parsing of text lines using file streams (fstream), demanding more explicit error handling and format validation.

6. Conclusion

This initial planning and design document outlines a rigorous strategy for the cross-language development of the Expense Tracker application. By committing to a Command-Line Interface (CLI) structure, the team ensures that the core focus remains on the utilization and comparison of fundamental language features, rather than complex external frameworks.

The document details the application design, task assignment, and a structured timeline. The explicit definition of anticipated cross-language challenges - particularly regarding the inherent differences in memory management, data typing, and I/O handling between Python and C++ provides a proactive framework for managing development risks and ensures a focused analysis for the final comparison report. The team is well-prepared to transition into the implementation phase with clear objectives and a comprehensive plan to successfully deliver both operational applications and the analysis and presentation.

7. 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/