Expense Tracker Application - Team 4

Cross-Language Application Development – Design Planning Document

Team Member	Responsibility
Parthasarathi Ponnapalli	C++ Backend + Qt UI Design + Doc +Presentation
Ashish Mahajan	Python Backend + PySide6 Integration + Doc + Presentation

1. Project Overview

This document presents the Initial Planning and Design for the Expense Tracker Application, developed as part of the Cross-Language Application Development group project. The application will be built in two programming languages, C++ and Python, both producing a fully functional desktop application with identical features. The C++ version uses Qt6 for the UI, while the Python version uses PySide6, the official Qt6 binding for Python.

A shared UI approach is used: the Qt Designer .ui file defines all screens and layouts once, and both implementations load from the duplicate files. Optionally, a shared. The qss stylesheet ensures visual consistency between both versions. This decision reduces duplication and allows the team to focus effort on the language-specific backend logic, which is where the meaningful differences lie.

2. Application Features

Both implementations expose the same set of user-facing features as required by the project specification.

1.1 Expense Entry

- Add a new expense with a date, amount, category, and description.
- Edit an existing expense through a modal dialog.
- Delete a selected expense with a confirmation prompt.
- Input validation to ensure amounts are non negative and required fields are filled.

1.2 Expense Viewing and Filtering

- View all recorded expenses in a scrollable, sortable table.
- Filter expenses by date range using from and to date pickers.
- Filter expenses by category using a dropdown.
- Combine both filters simultaneously for more specific results.

1.3 Summary and Reporting

- A summary panel showing total spend per category.
- An overall total that updates whenever data changes.
- A simple bar chart displaying category totals visually.

1.4 Data Persistence

- Both applications save and load expense data using a shared JSON file format, so data created by one implementation can be opened by the other.
- C++: data is managed in memory using STL containers and written to JSON using a lightweight header only library.
- Python: data is managed as a list of dictionaries and written to JSON using Python's built-in json module.

3. Language-Specific Design

3.1 C++ Implementation - Parthasarathi Ponnapalli

The C++ implementation focuses on the language features specified in the project brief: structured data types, STL containers, memory management, and Qt's Model-View architecture.

Data Representation

Each expense is represented as a typed struct with explicitly declared fields for date, amount, category, and description. This enforces type safety at compile time and avoids the ambiguity of loosely typed data.

Storage and Logic

All expenses are stored in an STL vector. Filtering operations use STL algorithms such as find_if and ranges::filter, keeping the logic concise and idiomatic. Category totals are accumulated using a std::map. The design avoids raw pointers throughout - dynamically created objects use smart pointers, and Qt widgets follow Qt's parent-child ownership model.

UI Integration

A QAbstractTableModel subclass bridges the expense data to Qt's table view widget. The main window and dialogs are loaded from the shared .ui files at compile time via Qt's uic tool, which is invoked automatically by CMake. Signals and slots are connected using Qt's connect() function and the Q_OBJECT macro.

Build System

The C++ application is built using CMake. CMake handles Qt's code generation steps (moc, uic, rcc) automatically and manages the external JSON dependency through its FetchContent mechanism.

3.2 Python Implementation - Ashish Mahajan

The Python implementation focuses on the language features specified in the project brief: Python dictionaries, list comprehensions, dynamic typing, and the datetime module.

Data Representation

Each expense is stored as a Python dictionary with string keys for date, amount, category, and description. Python's dynamic typing means no explicit type declarations are needed, and the datetime module is used for date parsing and comparison.

Storage and Logic

All expenses are held in a Python list. Filtering is done with list comprehensions, which are concise and idiomatic in Python. Category totals are accumulated using a defaultdict from the collections module, which automatically initialises missing keys to zero without extra boilerplate.

UI Integration

A QAbstractTableModel subclass is also used on the Python side, implemented using PySide6. The shared .ui files are loaded at runtime using PySide6's uic.loadUiType() function. Signals and slots are connected using PySide6's @Slot decorator and the .connect() method.

Executable Packaging

The Python application is packaged into a self-contained executable using PyInstaller. The packaged binary bundles the Python interpreter, all PySide6 Qt libraries, the shared .ui and .qss files, and the application code, so no Python installation is needed on the end user's machine. A small resource_path() helper ensures that bundled files are found correctly both during development and inside the packaged executable.

4. Architecture Overview

Both implementations follow a Model-View-Controller structure, which maps naturally onto Qt's design. The table below shows how the MVC layers are realized in each language.

MVC Layer	C++ Component	Python Component
Model	ExpenseRepository + std::vector of structs. ExpenseTableModel bridges data to the view.	ExpenseRepository + list of dicts. ExpenseTableModel bridges data to the view.
View	Shared .ui files loaded at compile time via uic. Shared .qss stylesheet.	Duplicate .ui files loaded at runtime via uic.loadUiType(). Same .qss stylesheet.
Controller	MainWindow connects signals and slots using Q_OBJECT and connect().	MainWindow connects signals and slots using @Slot decorators and the .connect() method.
Persistence	Reads and writes shared/data/expenses.json using a header-only JSON library.	Reads and writes the same expenses.json using Python's stdlib json module.

5. Task Assignment

Task	Owner	Priority
Qt Designer UI - design all three .ui screens and the Optional shared .qss stylesheet	Both	High
Set up GitHub repository, branches, and .gitignore	Ashish	High
C++ expense data struct and repository class	Parthasarath i	High
C++ table model and filter proxy	Parthasarath i	High
C++ main window, dialogs, and signal/slot wiring	Parthasarath i	High

Task	Owner	Priority
C++ summary panel and bar chart	Parthasarath i	Medium
Python expense data class and repository module	Ashish	High
Python table model and filter proxy	Ashish	High
Python main window - load shared .ui files, wire signals and slots	Ashish	High
Python summary panel and bar chart	Ashish	Medium
PyInstaller packaging and resource path helper	Ashish	Medium
Comparison report and presentation slides (joint)	Both	High

6. Project Timeline

Day	C++ Milestones	Python Milestones	Joint
Friday	UI screens designed in Qt Designer. The repository is set up with branches. Data structure defined.	Environment set up. Repository cloned. Data module skeleton drafted.	Planning document finalized. UML is committed to docs/. JSON data format agreed.
Saturda y	Expense repository done. Table model and filtering are working. Main window and dialogs connected.	Expense repository done. Table model and filtering are working. The main window loads shared .ui files and all connected slots.	Both apps were tested against the same expenses.json. Pull requests reviewed.
Sunday	Summary panel and chart complete. Final polish and code comments.	Summary panel and chart complete. PyInstaller build tested.	Comparison report written (APA 7). Presentation prepared and rehearsed. Branches merged to main.

7. Anticipated Language Specific Challenges

C++ Challenges

- Qt's parent-child ownership system and smart pointers must be used carefully together. When a widget is given a Qt parent, Qt manages its lifetime - wrapping it in a smart pointer would cause a double-free error. The rule we will follow is: widgets get a Qt parent, non-widget objects get a smart pointer.
- Implementing QAbstractTableModel requires overriding several methods (rowCount, columnCount, data, headerData, flags) and is more boilerplate-heavy than the Python equivalent.
- Date handling requires converting between Qt's QDate type and the JSON file's string format.

Python Challenges

- Because Python is dynamically typed, mistakes such as a misspelled dictionary key will not be caught until the code runs. We will use type hints throughout to make these issues easier to spot during development.
- The .ui and .qss files must be found at runtime by the PyInstaller executable. The resource_path() helper resolves file paths correctly both during development and in the frozen binary.
- PySide6's QAbstractTableModel requires that the data() method return None for unhandled display roles rather than raising an exception, which is easy to overlook.

8. GitHub Repository Structure

The project uses a single GitHub repository with two primary development branches — one for the C++ implementation and one for Python. Both branches will be merged into the main branch once the implementations are complete and verified.

Branch / Path	Owner	Contents
feature/C++	Parthasarathi	All C++ source files, CMakeLists.txt, and build configuration.
feature/python	Ashish	All Python source files, requirements.txt, and PyInstaller spec.
shared/ui/	Both	Both implementations use Qt Designer .ui files.
shared/resources /	Both	Shared .qss stylesheet and application icon. (optional)
shared/data/	Both	expenses.json — the default data file shared between both apps.
docs/	Both	Planning document, UML diagram, and comparison report.
README.md	Both	Project overview and setup instructions for both implementations.
main	Both	Merged, final state of both implementations after Day 3.

References :
AI_ASSISTED formatting.