INF1002 C Project Report

Class Management System

Team ID: P1-08

Name	Student ID	Email
Mohammed Aamir	2500933	2500933@sit.singaporetech.edu.sg
Muhammad Hasif Bin Mohd Faisal	2500619	2500619@sit.singaporetech.edu.sg
Timothy Chia Kai Lun	2501530	2501530@sit.singaporetech.edu.sg
Low Gin Lim	2501267	2501267@sit.singaporetech.edu.sg
Dalton Chng Cheng Hao	2504003	2504003@sit.singaporetech.edu.sg

Course: INF1002 – Programming Fundamentals

Institution: Singapore Institute of Technology

Date of Submission: 25th November, 2025

Table of Contents

Executive Summary

This report presents the design, implementation, and evaluation of a modular Class Management System (CMS) developed in the C programming language for the INF1002 Programming Fundamentals module. The CMS provides a file-based solution for managing student records through a structured command-line interface. Core functionalities include loading and saving database files, displaying and modifying records, validating user input, and enforcing a controlled workflow that preserves data integrity.

The system is architected around a clear separation of concerns. A presentation layer handles user interaction and menu flow; a controller and operation registry coordinate command execution; and a data layer manages file parsing, in-memory storage, and persistence. Student records are stored in a dynamically resizing array, chosen for its simplicity and predictable performance on small to medium datasets typical of academic usage. Additional modules support sorting, summary statistics, advanced query filtering, file-integrity checks using 32-bit cyclic redundancy checks, and per-session event logging to improve reliability and traceability.

Development followed a structured process beginning with requirements analysis and domain understanding, leading to a layered program design and iterative refinement during implementation. Defensive programming practices, typed status codes, and strict parsing rules were used extensively to prevent malformed input and to ensure consistent behaviour across all commands.

System correctness and robustness were validated through a comprehensive testing strategy incorporating both white-box unit tests and black-box scenario testing. These tests covered normal operations, edge cases, error handling, and all enhancement features. Performance measurements across datasets up to 1000 records confirmed that the CMS delivers instantaneous responses for its intended scale.

Overall, the CMS meets all required specifications and integrates several meaningful enhancements that strengthen usability, analysis capabilities, and data protection. The result is a reliable, extensible, and well-structured application demonstrating effective use of modular design, validation techniques, and core programming concepts in C.

Introduction

Background Information

A database is a structured collection of data that supports the storage, retrieval, and manipulation of information. In practice, this includes collections of related information like customer records, financial transactions, or inventory lists that organisations rely on daily.

A Database Management System (DBMS) is a software system that provides the tools and interface for managing databases, including organising, storing, and retrieving high volumes of data efficiently and reliably (Database Management System, 2003). DBMSs underpin most modern information systems and are an essential requirement for data-driven domains.

In many classes, lecturers and teaching staff struggle with student information scattered across spreadsheets or paper records, making it slow to find a specific student, correct mistakes, or update marks. This leads to inconsistent data, difficulty generating summaries (such as averages or top performers), and a higher risk of losing important records between sessions. The Class Management System (CMS) addresses this by providing a single, persistent system where student data can be quickly retrieved, safely updated, validated, and summarised.

Objectives and Scope

The objectives of this project are to design and implement a simple, single-user, file-based CMS for managing student records, and to provide a controlled, command-line interface for standard record operations while minimising the risk of file corruption. The CMS follows a clear, linear workflow so that all viewing and modification of records occurs through defined operations, preserving data integrity and ensuring accurate retrieval, updates, and deletions.

Table 1 summarises the in-scope features: opening and saving a data file, displaying, inserting, querying, updating, and deleting records, as well as enhancement features such as sorting, summary statistics, advanced querying, checksum data integrity checks, and tracking events within a CMS session. Out of scope are graphical interfaces, support for multiple data table schemas, and relational queries.

In addition to this report, the project deliverables include a complete C codebase (Chia et al., 2025), which is publicly available for review.

Feature	Module
OPEN	Open the database file and read in all the records.
SHOW ALL	Display all the current records in the read-in data.
INSERT	Insert a new data record.
QUERY	Search if there is any existing record with a given student ID.
UPDATE	Update the data for a record with a given student ID.
DELETE	Delete the record with a given student ID.
SAVE	Save all the current records from memory into the database file.
SORT	Sort records in place by ID/Mark in ascending/descending order.
ADV QUERY	Runs the advanced query pipeline.
STATISTICS	Display summary statistics for all students.
SHOW LOG	Display operation history for the current session.
CHECKSUM	Verify database integrity and display checksums.

Table 1. *CMS core user operations and enhancement features.*

Methodology

Analysis

Understanding the Context Problem

In practice, lecturers and administrators typically use Excel sheets or CSV files to create class lists, update marks, and check individual student details when queries arise. Common tasks include adding new students at the start of a term, updating marks after assessments, correcting mistakes, and scanning for students who are failing or missing results.

However, these tasks are often done directly on raw files, which introduces several pain points: different staff may keep separate copies of the same file, manual edits can break formatting or misalign data, and searching for a single student or re-sorting the class list is slow and error-prone. There is also little validation of input and no enforced workflow, so invalid IDs, empty fields, or out-of-range marks can easily slip in. The CMS is designed around these real workflows and channels them through predefined operations so that the same tasks can be performed in a controlled, consistent, and less error-prone way.

Domain Knowledge, Algorithms and Tools

The CMS is implemented in ISO C using the C standard library for console and file I/O, string manipulation, general utilities, and basic character handling (International Organization for Standardization, 2024). Core algorithms include linear search over the in-memory array of records, simple statistical aggregation for summary statistics, and a lightweight parsing routine to interpret user commands and structured arguments. Sorting is implemented using a stable, in-place bubble sort, chosen for its deterministic behaviour and suitability for the small datasets typical of this system (GeeksforGeeks, 2025a). To support data integrity, the CMS also applies a 32-bit Cyclic Redundancy Check (CRC32) algorithm to detect unsaved changes and verify consistency between the in-memory dataset and the persisted file (GeeksforGeeks, 2025b).

Development is supported by `make` for build automation, with `gcc` and `clang` used as C compilers, and Git/GitHub for version control and collaboration. No external application framework is used; the system is a standalone C command-line application built directly on the C standard library.

Use Case Descriptions

A user managing student records typically begins by loading an existing dataset from the storage file. After the data is loaded, the user may view all records to confirm the state of the dataset or inspect its overall structure. When new records need to be added, the user enters the relevant details, and the system validates that no duplicate identifier exists. If a user needs to check information for a specific student, the query operation retrieves the corresponding record. When corrections or updates are required, the update function modifies the selected fields. Records may also be removed when they are no longer needed, followed by an explicit confirmation step to prevent unintended deletions. Finally, the user saves all changes, ensuring that the dataset remains up to date for future sessions. These use cases reflect the typical tasks performed by individuals responsible for maintaining basic academic records and form the basis for the system's overall workflow and design.

Design

System Architecture

The CMS follows a modular, layered architecture separating the Presentation, Service, and Data layers as illustrated in Figure 1. The UI Loader captures user commands and forwards them to the CMS Controller, which dispatches operations through an Operation Registry that maps each command to its handler. This design isolates control flow from individual features and allows new operations to be added without modifying the main loop.

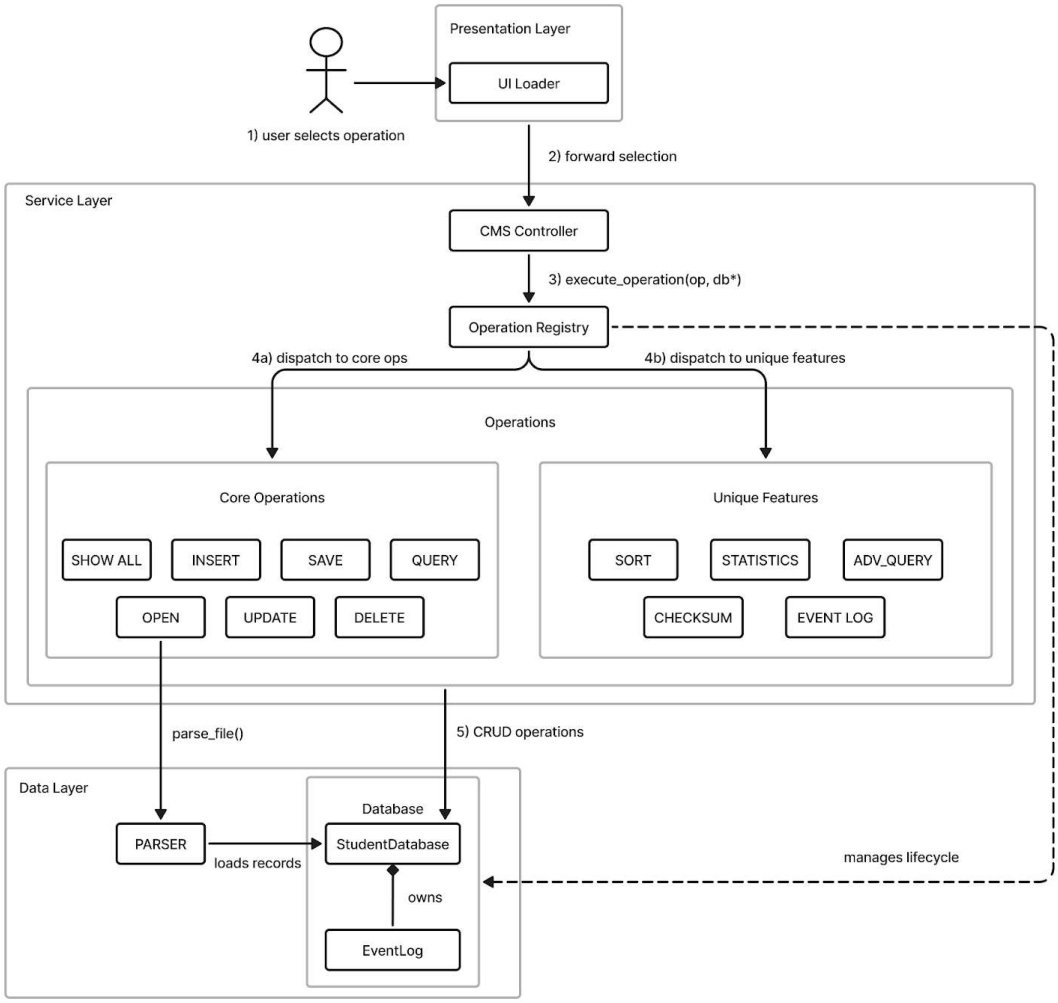

Figure 1. *High-level layered architecture of the CMS, showing how user operations flow through the orchestrator from the UI to the Controller and Operation Registry to core and unique operations over the StudentDatabase and EventLog.*

In the Presentation Layer, the UI Loader displays the text-based menu, captures user selections, and forwards them to the Service Layer. The CMS Controller invokes `execute_operation(op, db*)` on a shared `StudentDatabase` instance. Command execution is handled via an Operation Registry that maps each command listed in Table 1 to a dedicated handler, decoupling the main control loop from the concrete operations and allows for expansion without modifying the CMS Controller.

The Data Layer provides storage, parsing, and validation. The Parser loads the text file, validates each line, and populates the `StudentDatabase`. The database module maintains dynamic tables of records, while the `EventLog` records actions during a session. The filesystem is the only external dependency, used to persist the CMS text file.

Data Structure Design

Student data in the CMS is organised using a set of structured C types, with `StudentRecord` serving as the basic unit containing each student's identifier, name, programme, and mark. These records are stored within a `StudentTable`, which maintains a heap-allocated, dynamically resizing array along with size and capacity metadata. This design supports small to medium class sizes efficiently: memory begins with a modest initial allocation and expands by doubling only when necessary, avoiding the overhead of more complex data structures while still allowing the system to scale smoothly as records are added. At a higher level, a `StudentDatabase` struct aggregates one or more tables and provides the interface through which user operations are performed. All structures central to this data model are defined within the database.h module.

Software Solution Process

The CMS follows a menu-driven flow from initialisation to termination, as shown in Figure 2. On startup, the CMS displays a declaration, initialises its data structures, and enters the main session loop. Each cycle displays the command menu, captures the user's selection, and validates the input. Invalid input triggers an error message and a return to the menu.

For valid selections, the CMS first checks whether the EXIT command was chosen. If so, and unsaved changes exist, the system prompts the user to save, discard, or cancel the exit attempt. Saving writes the current dataset to file, while discarding terminates without persisting changes; cancelling resumes the session. For all other operations, the system verifies that a database has been loaded before executing the requested action.

After an operation completes, the CMS displays either the result or an error and returns to the session loop. This cycle continues until the user confirms exit and all required save or discard actions are resolved, at which point the session ends cleanly.

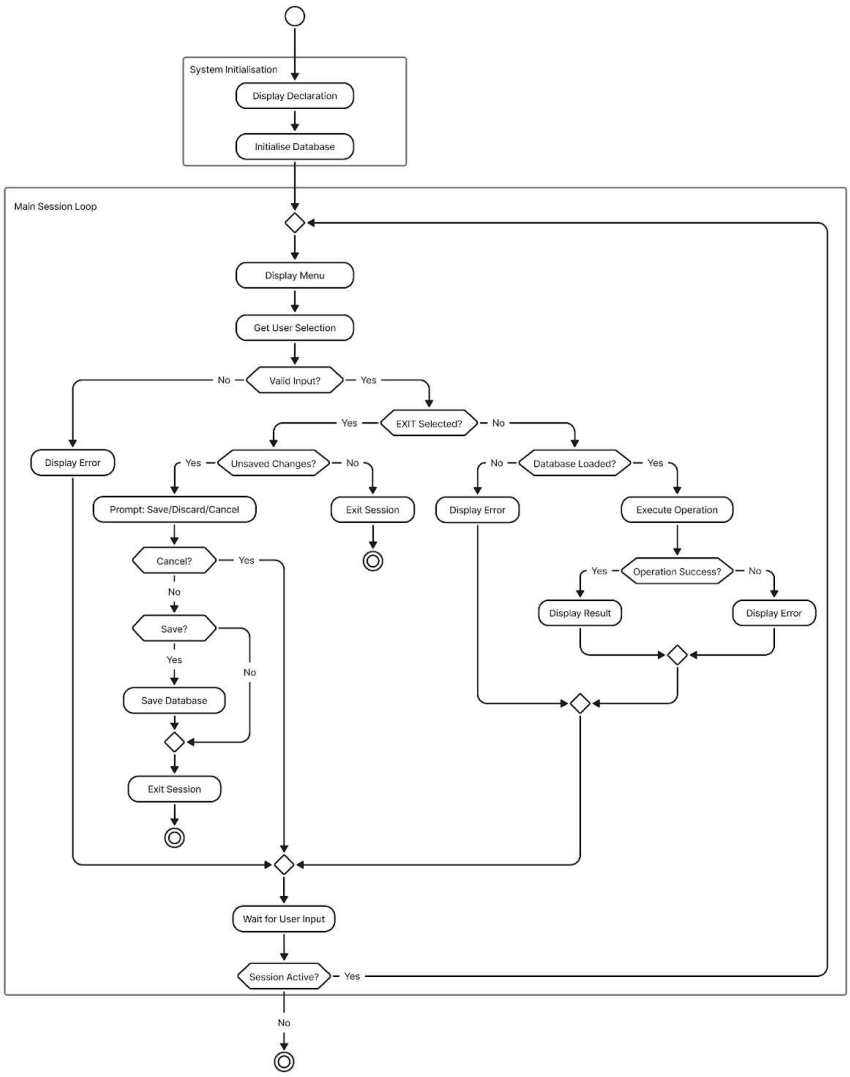

Figure 2. *High-level software solution process for the CMS, showing system initialisation, the main session loop for menu-driven user interaction, validation and execution of operations (including database preconditions and error handling), and the exit flow with save/discard options for unsaved changes.*

Implementation Overview

Development Environment

The CMS programme was implemented in ISO C using the GNU Compiler Collection (GCC), configured with strict warning flags (`-Wall -Wextra`) and debug symbols (`-g`) to surface potential defects early. The codebase depends only on standard C library headers (e.g. `stdio.h`, `stdlib.h`, `string.h`, `ctype.h`, `errno.h`, `time.h`, `sys/stat.h`), keeping the system lightweight and portable.

Builds are managed with GNU Make via a single Makefile, with targets for compiling the main programme, running it, executing the test suite, and cleaning build artefacts. Source files are organised under `src/`, headers under `include/`, tests under `tests/`, and binaries in `build/`.

Development occurred on Windows, and Unix-like systems with Visual Studio Code as the primary editor with gcc and clang integration. The code relies only on portable C and POSIX-compliant file operations, avoiding platform-specific APIs so that the programme can be compiled and run on any Unix-like system with minimal effort.

Most debugging was done with terminal print statements and by stepping through variable values when functions behaved unexpectedly. Typed enum status codes (for example, a CMS-wide status enum distinguishing success and various failures) improved readability, helped pinpoint failure points, and allowed the system to fail fast when operations did not succeed. We did not use external libraries or frameworks; features such as sorting, checksum computation, the advanced query pipeline, and event logging were implemented manually with standard C functions.

Code Organisation and Naming Conventions

The CMS codebase is organised into clearly separated folders as shown in Figure 3. Source files reside in `src/`, with one module per file, where a module is defined as a group of related functionalities implemented in a single source file. These modules cover distinct aspects of the system. Header files are placed in `include/`, `tests/` contains the test harness, `data/` stores the text database files, `assets/` holds presentation-layer content, and `build/` contains generated artefacts. Table 2 summarises the main modules and their responsibilities.

Naming conventions are applied consistently to support readability and maintainability. Each module (a C file named as a noun) focuses on a single aspect of the system, while functions (named as verbs in lower snake case) implement well-scoped actions, and variables also use lower snake case. Custom types such as structs and enums use camel case, and constants and enum values use upper snake case for quick visual distinction. Public functions are documented with concise docstrings following the Google C++ style guide, clarifying on purpose, parameters, and return values (Google, n.d.).

```
INF1002-P1-08-CMS/
├── src/
│   ├── core modules (cms.c, database.c, parser.c, ui.c, ...)
│   └── commands/
│       └── *_command.c
├── include/          # header files
├── tests/            # test_*.c and fixtures
├── data/             # P1_8-CMS.txt
├── assets/           # declaration.txt, menu.txt
├── build/            # compiled artefacts
└── Makefile
```

Figure 3. *Directory layout of the CMS project, illustrating the separation of source modules, headers, tests, data files, assets, and build artefacts to support clear organisation and maintainability.*

Key Functions

Each operation is implemented as a dedicated command handler (`*_command.c`) and takes a `StudentDatabase` pointer as input, then returns an `OpStatus` code indicating success, validation failure, input error, or database-level error. Only the essential behaviour is outlined here; detailed logic is encapsulated within each module. Table 3 summarises all operations and their key responsibilities and behaviour. All key command handlers return an `OpStatus` value: `OP_SUCCESS` on success, and one of the `OP_ERROR_*` codes on failure.

The major operations: OPEN, INSERT, UPDATE, and DELETE have richer control flow than the others. OPEN handles checksum-based unsaved-change detection, confirms reloads, resets in-memory data, and performs file parsing and validation. INSERT validates all fields before appending, UPDATE locates a record and applies a single validated field change, and DELETE verifies the ID, confirms the action, and safely removes the entry while treating cancellations and missing IDs as normal outcomes.

Module	Location	Responsibility
`main.c`	`src/`	Programme entry point.
`cms.c`	`src/`	Overall control flow and user input handling.
`database.c`	`src/`	n-memory student record storage and access.
`parser.c`	`src/`	Loading and parsing the CMS text file.
`ui.c`	`src/`	Rendering menus and messages to the terminal.
`sorting.c`	`src/`	Sorting records by selected fields.
`statistics.c`	`src/`	Computing summary statistics over records.
`adv_query.c`	`src/`	Advanced query filtering and pipeline logic.
`checksum.c`	`src/`	Computing checksums for file integrity checks.
`event_log.c`	`src/`	Recording events and status messages.
`*_command.c`	`src/commands/`	Per-command handlers.
`operation_registry.c`	`src/commands/`	Registering and dispatching command handlers.

Table 2. *Overview of core CMS modules and their primary responsibilities.*

Operation	Function	Purpose
OPEN	`execute_open()`	Load and parse a database file into memory, validate each record, and update internal checksums and metadata.
INSERT	`execute_insert()`	Collect a new student's details, validate all fields, and append a new record to the in-memory database.
QUERY	`execute_query()`	Prompt for a student ID, search the database, display the matching record or report missing entries.
UPDATE	`execute_update()`	Locate a record by ID and update one selected field after validation.
DELETE	`execute_delete()`	Confirm and remove a student record while keeping remaining records compact and aligned.
SAVE	`execute_save()`	Write all in-memory records to the database file and refresh the stored checksum.
SHOW ALL	`execute_show_all()`	Display all records in a formatted table with dynamically sized columns.

Table 3. *Mapping of core CMS commands to their implementation functions and intended behaviour.*

Enhancement and Unique Features

Beyond the core operations, the CMS implements several enhancements and unique features that improve usability, analysis, and robustness. The Sorting module allows users to order records by Student ID or Mark in ascending or descending order. Sorting is implemented as a stable in-place bubble sort, chosen for its deterministic behaviour and suitability for the small datasets typical of this assignment. The Statistics module computes record counts, averages, and extremal marks in a single linear pass, applying a small tolerance to floating-point comparisons to avoid rounding artefacts.

Input validation is also strengthened beyond the baseline brief. Command handlers and the parser enforce range limits, character-set rules, length constraints, and safe numeric conversions, with failures signalled through typed status codes so that operations fail fast and predictably. These enhancement modules are summarised in Table 4.

Feature	Module	Description
Sorting	`sorting.c`	Orders records by Student ID or Mark in ascending/descending order using a stable in-place bubble sort, chosen for deterministic behaviour and suitability for small datasets.
Statistics	`statistics.c`	Computes total record count, average mark, and highest/lowest marks in a single linear scan, with a small tolerance applied to floating-point comparisons.
Advanced Query	`adv_query.c`	Supports multi-stage filtering using substring matching on NAME/PROGRAMME and relational operators on MARK, with constraints to prevent malformed or duplicated filters and to fail gracefully with feedback.
Checksum Integrity Check	`checksum.c`	Detects unsaved changes by computing CRC32 checksums for the in-memory database and last saved file, prompting users appropriately during OPEN and EXIT.
Event Log	`event_log.c`	Records each operation, its status code, and timestamp in a dynamically growing log that users can review via the SHOW LOG command for transparency and traceability.

Table 4. *Overview of the enhancement and unique features implemented in the CMS.*

Results and Insights

Testing Strategy

Our testing approach combined white-box and black-box methods to ensure correctness across normal and exceptional scenarios. White-box unit tests targeted key modules including the parser, database, sorting, statistics, advanced query, checksum, and event log, verifying internal behaviour through status codes and edge-case handling. Black-box tests exercised full command sequences (e.g., OPEN → INSERT → UPDATE → SAVE), validating menu flow, input handling, and error messaging

from an end-user perspective. Together, these tests provided broad coverage and strong confidence in system reliability.

Test Cases and Validation

Representative test cases from our suite demonstrate broad coverage of system functionality. Below is a condensed selection of cases illustrating validation logic, core functionality, and enhancement features.

Test Case ID	Description	Input	Expected Output	Result
TC01	Open valid database file	OPEN	Records loaded, checksum set	Pass
TC02	Reject malformed record	Corrupted line	Parse error reported	Pass
TC03	Insert valid record	INSERT with valid fields	Record added	Pass
TC04	Reject duplicate ID	Insert ID already in table	Duplicate error	Pass
TC05	Query existing student	QUERY 12345	Displays full record	Pass
TC06	Query non-existent student	QUERY 99999	Not found message	Pass
TC07	Update student mark	UPDATE	Modified record saved in memory	Pass
TC08	Delete student	DELETE with confirmation	Record removed cleanly	Pass
TC09	Sort by ID ascending	SORT ID A	Records ordered by ID	Pass
TC10	Compute statistics	STATISTICS	Total, average, highest & lowest printed	Pass

Table 5. *Each result aligns with the system's functional expectations.*

Performance and Complexity Considerations

Performance tests were run on datasets of 100, 500, and 1000 records, and all core operations completed within roughly 0.006–0.013 seconds as seen in Table 6. The small differences between dataset sizes indicate that fixed overheads such as startup, I/O, and timing precision dominate at this scale, so the CMS appears instantaneous during normal use.

In terms of complexity, core operations such as querying, inserting, updating, deleting, and computing statistics run in O(n) time, while sorting uses a stable in-place bubble sort with $O(n^2)$ complexity. The Advanced Query Pipeline applies several linear filtering stages, making its cost proportional to the number of records and active filters. Memory usage is O(n), dominated by the dynamic array of student records and the event log. All allocations are freed on shutdown, and the system maintains a small footprint due to its reliance solely on standard C libraries.

Dataset Size (records)	SHOW_ALL (seconds)	QUERY_WORST (seconds)	SORT_MARK_ASC (seconds)	ADV_QUERY_3_FILTERS (seconds)
100	0.012550	0.007574	0.007195	0.007771
500	0.007347	0.006723	0.006413	0.006407
1000	0.006476	0.006418	0.006490	0.006354

Table 6. *Execution time in seconds for key operations at different dataset sizes*[1].

Quality Attributes

The CMS demonstrates several desirable software quality attributes. The modular code organisation improves readability and maintainability, with each module handling a clearly defined responsibility and exposing its interface through well-structured header files. Consistent naming conventions: `lower_snake_case` for functions and variables, `CamelCase` for types, and `UPPER_SNAKE_CASE` for constants enhance clarity. Command handlers are isolated in their own directory for traceability and ease of extension. Error handling follows consistent patterns across modules, and reliability is strengthened through the event log and checksum mechanisms, which track user actions and detect unsaved changes before potentially destructive operations.

Insights and Interpretation

Overall, the testing process demonstrated that the CMS reliably meets all core requirements, including loading, inserting, querying, updating, deleting, and saving records. The advanced features—sorting, statistics, checksum verification, and the advanced query pipeline—functioned correctly and added significant value beyond the project baseline. The error handling system proved effective, especially in parsing and record validation, where invalid or unexpected inputs were consistently identified and rejected without compromising system stability.

One notable insight is the importance of input sanitisation in complex features like the advanced query pipeline. Early tests revealed that malformed or repeated filter stages could produce inconsistent behaviour, leading to improvements in the validation logic. Additionally, the checksum mechanism proved valuable during black-box testing, preventing accidental data loss during early exit attempts. The event log provided a clear audit trail of actions, which proved useful during debugging and greatly enhanced the system's transparency.

In practical use, the CMS offers strong reliability for small to medium datasets. While sorting may be less efficient for extremely large tables, this limitation does not affect typical academic use cases. Overall, the system performs robustly, handles erroneous inputs gracefully, and provides a rich feature set that supports both administrative workflows and analytical tasks.

[1] Benchmark timings were obtained using a custom script available at: https://github.com/timothyckl/INF1002-P1-08-CMS/blob/main/benchmark.py

Conclusion

Summary of Achievements

The CMS developed in this project effectively addresses the challenges of managing student information in academic settings. It replaces unstructured, error-prone record keeping with a structured, validated solution in which data is consistently stored, easily retrieved, and safely modified. The system meets the original objectives by providing reliable core operations, structured file-based persistence, and strict input validation to protect data integrity

Beyond the baseline requirements, the CMS includes enhancement features that strengthen its analytical and operational capabilities. Sorting organises records meaningfully, summary statistics offer insight into class performance, the checksum mechanism guards against accidental data loss, and the event log provides transparency through action tracking. The multi-stage Advanced Query Pipeline further extends the system by supporting complex filtering scenarios for administrative and analytical use. Overall, the CMS delivers a robust, extensible, and well-architected solution for student record management.

Limitations

Despite its strong feature set, the CMS has several limitations arising from technical and contextual constraints. It uses an array-based in-memory structure, which is simple and efficient for small datasets but scales poorly as record counts reach the tens of thousands, due to O(n) searches and $O(n^2)$ bubble-sort performance. The system also runs solely in a command-line environment, which may be less accessible to users unfamiliar with text interfaces. Its plain-text file format is transparent but less robust than binary or database-backed storage. Finally, the CMS is a single-user, single-session application; it lacks concurrency control and multi-user access, limiting its suitability for larger institutional deployments.

Future Work

Several avenues for future improvement could enhance the scalability, usability, and feature richness of the CMS. Migrating from an array-based structure to a more sophisticated data organisation, such as hash tables or balanced trees which would increase efficiency for large datasets. Alternatively, integrating a lightweight database engine could offer both improved performance and expanded query capabilities. A graphical user interface would significantly improve accessibility for non-technical users and create a more intuitive user experience.

The Advanced Query Pipeline could be extended to support additional operators, compound conditions, logical branching, and nested expressions, thus enabling more expressive filtering.

Introducing user authentication, session profiles, or role-based access control would provide security-enhanced workflows suitable for real-world deployment. Finally, transitioning the file format to a more resilient structure (such as JSON, CSV with schema enforcement would increase portability and reduce the risk of corruption. These enhancements would move the CMS closer to a production-ready information management system.

Team Contribution

Team Member	Major Contributions
Aamir	Implemented the INSERT and DELETE commands; designed and implemented input validation for new records; developed safe deletion logic and the sorting subsystem, enabling ordering of records by ID or Mark in ascending/descending order.
Timothy	Implemented OPEN and SHOW ALL commands; developed the checksum verification module; handled remaining command implementations; built the command registry and dispatching mechanism; authored the automated unit test suite; managed code merging, architectural integration, and final system assembly.
Dalton	Implemented the UPDATE command; contributed to the event log module for recording operations during the session.
Gin	Implemented the SAVE command; developed the summary statistics module for computing record count, average marks, and highest/lowest scores.
Hasif	Implemented the QUERY command; designed the Advanced Query Pipeline (GREP and MARK filtering logic, validation mechanisms); integrated it into the command interface; authored tests for QUERY and ADV QUERY.
AI Teammate	Assisted in refining system architecture, structuring the report, improving documentation clarity, and articulating technical descriptions for complex features such as the advanced query pipeline, checksum behaviour, and system flow diagrams.

References

Chia, T., Bin Mohd Faisal, M. H., Chng Cheng Hao, D., Aamir, M., & Gin Lim, L. (2025). Class Management System [Computer software]. https://github.com/timothyckl/INF1002-P1-08-CMS

Database Management System (DBMS) | Encyclopedia of Computer Science. (2003). *DL Books*. https://doi.org/10.5555/1074100.1074317

GeeksforGeeks. (2025a, July 23). *Bubble Sort Algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/bubble-sort-algorithm/

GeeksforGeeks. (2025b, May 24). *Cyclic Redundancy Check and Modulo 2 division*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/modulo-2-binary-division/

Google C++ Style Guide. (n.d.). https://google.github.io/styleguide/cppguide.html

International Organization for Standardization. (2024). *ISO/IEC 9899:2024: Programming languages — C*. https://www.iso.org/obp/ui/en/#iso:std:iso-iec:9899:ed-5:v1:en