Software Requirements Specification (SRS)

1. Document Control

Project Name: Study Buddy – Clemson Student Study Session Scheduler (CLI)

Version: 1.0 (Final; no changes allowed after approval)

2. Purpose

This document specifies the functional and non-functional requirements for a simple, command-line “Study Buddy” application that allows Clemson students to create profiles, record enrolled courses and weekly availability, find classmates in the same course with overlapping availability, schedule study sessions, and confirm/cancel those sessions. It will be implemented in standard C++17 (front-end CLI to back-end core), using only the C++ standard library and simple local file storage. The SRS will serve as the single source of truth for design, implementation, and acceptance.

3. Scope

- **In Scope**: Profile creation, course enrollment tracking, weekly availability management, search/filter by exact course code, overlap-based match suggestions, session scheduling, participant confirmation, session cancellation, persistence to local files, and basic error handling via CLI prompts and messages.
- **Out of Scope**: GUI/desktop/mobile apps, networking, cloud services, university authentication/SSO, calendar integrations, advanced optimization/scheduling algorithms, complex time-zone handling, and high-scale multi-user concurrency.

4. Definitions, Acronyms, Abbreviations

- **CLI**: Command-Line Interface.
- **Course Code**: Department and number (e.g., `CPSC 2120`).
- **Availability Slot**: A weekly recurring time block defined by **day of week** and **start hour**/**end hour** (24-hour clock).
- **Session**: A scheduled meeting for a specific course at a specific weekly time slot, with participants and a status.

- **Status**: `PROPOSED`, `CONFIRMED`, `CANCELLED`.
- **User/Student**: A Clemson student using this app.

5. References

- IEEE 830/29148 style SRS structure (general guidance).
- C++17 ISO/IEC 14882:2017 standard (language and STL).

---

6. Overall Description

6.1 Product Perspective

Standalone, offline CLI utility. Single-binary application with local file persistence (CSV-like text files). No external services. Assumes one user operating the program at a time on a local machine.

6.2 Product Functions (High-Level)

1. Create and manage a student profile (name, email, optional passcode). //good
2. Add/remove courses the student is enrolled in. //good
3. Add/remove weekly availability slots.
4. Search classmates by **exact** course code and overlapping availability.
5. Suggest matches (list classmates + candidate overlapping time slots).
6. Create a study session proposal; invite classmates.
7. Confirm or cancel scheduled sessions.
8. View upcoming sessions and invitations.

6.3 User Classes and Characteristics

- **Student (Primary)**: CLI user with basic command familiarity. Expects simple prompts and helpful error messages.

6.4 Operating Environment

- OS: Windows 10+, macOS 12+, or a modern Linux distribution.
- Compiler: Any C++17-compliant compiler (e.g., g++, clang++, MSVC).
- Storage: Local filesystem (text files).

6.5 Design and Implementation Constraints

- **Language:** C++17, STL only.

- **Interface:** CLI only.
- **Storage:** Local text files (CSV-like).
- **Time Representation:** Weekly recurring schedule using day-of-week (`0=Sun..6=Sat`) and hour integers `0..23`with **1-hour granularity**.
- **IDs:** Integer IDs assigned sequentially at runtime and persisted.
- **No exotic libraries** (performance enhancers, JSON libs, ORMs, etc.).

6.6 Assumptions and Dependencies

- Students know their exact course codes.
- Local single-user access; no concurrent writes.
- Times are local machine time; no DST/time-zone adjustments.
- Email format is free-text; minimal validation only.

---

7. System Features (Functional Requirements)

Numbered requirements are **binding**. Each requirement is testable.

7.1 Profiles

R-P1 The system shall allow creation of a student profile with fields: `name`, `email`, and optional `passcode`.
R-P2 The system shall persist new and updated profiles to `students.csv`.
R-P3 The system shall prevent creation of duplicate profiles with the same email.
R-P4 The system shall allow viewing and editing of profile `name` and `email`.
R-P5 If a `passcode` is set, the system shall store only a non-reversible hash (e.g., `std::hash<std::string>` result) and compare by hash. *(Informational: not production-grade security.)*

7.2 Courses

R-C1 The system shall allow adding an exact course code (e.g., `CPSC 2120`) to the student's enrollment list.
R-C2 The system shall allow removing a course from the enrollment list.
R-C3 The system shall prevent duplicate course entries per student.
R-C4 Enrollments shall be persisted to `enrollments.csv`.

7.3 Availability

R-A1 The system shall allow adding a weekly availability slot defined by `day_of_week`, `start_hour`, `end_hour` with `0 ≤ start < end ≤ 24` and integer hours.
R-A2 The system shall merge or reject overlapping/adjacent availability slots for the same day to maintain normalized ranges.
R-A3 The system shall allow removing an availability slot by exact match or index.
R-A4 Availability shall be persisted to `availability.csv`.

7.4 Search & Match Suggestions

R-S1 The system shall let a student search classmates by **exact** course code.
R-S2 The system shall compute overlap between the searching student's availability and each classmate's availability.
R-S3 The system shall list suggested matches as `(classmate_id, classmate_name, overlapping time slots)`sorted by soonest available slot, then by name.
R-S4 If no overlaps exist, the system shall state that no matches were found.

7.5 Session Scheduling & Management

R-M1 The system shall allow creating a `PROPOSED` session for a given course and time slot `(day, start_hour, duration=1)`.
R-M2 The system shall prevent creating sessions that conflict with the organizer's existing confirmed sessions.
R-M3 The system shall allow inviting one or more classmates enrolled in the same course.
R-M4 A session becomes `CONFIRMED` when the organizer and **all** invited participants mark `confirm`.
R-M5 Any participant may set a `PROPOSED` or `CONFIRMED` session to `CANCELLED`; the system shall record a reason code (free text allowed).
R-M6 The system shall prevent inviting the organizer as a separate participant and prevent duplicate invites.
R-M7 The system shall block proposing a session in a slot that is outside the organizer's availability.
R-M8 Sessions and participants shall be persisted to `sessions.csv` and `session_participants.csv`.

7.6 Viewing & Notifications (CLI)

R-V1 The system shall list a student's sessions grouped by status (`PROPOSED`, `CONFIRMED`, `CANCELLED`).
R-V2 The system shall list pending invitations requiring confirmation.
R-V3 The system shall provide clear CLI messages for actions taken and errors encountered.

7.7 Data Persistence & Integrity

R-D1 All mutations (create/update/delete) shall be immediately written to disk.
R-D2 File writes shall be atomic via write-to-temp then rename to prevent corruption.
R-D3 The system shall validate on load and skip malformed records with a warning.

---

8. External Interface Requirements

8.1 User Interface (CLI)

- Simple verb-noun commands with arguments; interactive prompts when arguments are missing.
- Example commands (exact names are binding):
  - `create_profile --name "Jane Doe" --email janed@clemson.edu [--passcode ****]`
  - `edit_profile --name "Jane Q. Doe"` / `edit_profile --email janeqd@clemson.edu`
  - `add_course --code "CPSC 2120"` / `remove_course --code "CPSC 2120"` / `list_courses`
  - `add_availability --day 2 --start 14 --end 16` / `remove_availability --day 2 --start 14 --end 16` / `list_availability`
  - `search_matches --course "CPSC 2120"`
  - `schedule_session --course "CPSC 2120" --day 2 --start 15 --invite 7,12`
  - `list_sessions` / `list_invitations`
  - `confirm_session --id 31` / `cancel_session --id 31 --reason "Conflict"`
  - `help` / `exit`

8.2 Hardware Interfaces

- None (standard keyboard/terminal I/O only).

8.3 Software Interfaces

- Filesystem for persistence via standard `fstream`.

8.4 Communications Interfaces

- None (offline single-user).

9. System Data Model

9.1 Entities

Student
`id:int`, `name:string`, `email:string`, `pass_hash:optional<size_t>`

Enrollment
`student_id:int`, `course_code:string`

Availability
`student_id:int`, `day:int(0..6)`, `start:int(0..23)`, `end:int(1..24)` *(1-hour granularity; exclusive end)*

Session
`id:int`, `course_code:string`, `day:int`, `start:int`, `duration:int(=1)`, `organizer_id:int`, `status:{PROPOSED,CONFIRMED,CANCELLED}`, `cancel_reason:string?`

SessionParticipant
`session_id:int`, `student_id:int`, `confirmed:bool`

9.2 File Formats (CSV-like)

- `students.csv`: `id,name,email,pass_hash?`
- `enrollments.csv`: `student_id,course_code`
- `availability.csv`: `student_id,day,start,end`
- `sessions.csv`: `id,course_code,day,start,duration,organizer_id,status,cancel_reason`
- `session_participants.csv`: `session_id,student_id,confirmed`

(Values containing commas will be quoted. The program will escape quotes inside quoted fields.)

10. Detailed Behavior & Algorithms

10.1 Availability Normalization

- On `add_availability`, if a new slot overlaps or is adjacent to an existing slot for the same day, merge into a single `[min_start,max_end]` range.
- Reject if `start<0`, `end>24`, or `start≥end`.

10.2 Overlap Computation

- Two slots overlap if `max(a.start, b.start) < min(a.end, b.end)`.
- Overlapping interval can yield candidate session start hours in the integer range `[overlap_start, overlap_end-1]`.

10.3 Match Suggestion

- For each classmate in `course_code`, compute overlaps with the current user's availability.
- Produce tuples `(classmate_id, name, list<day,start>)`.
- Sort by `(day ascending from today's day index), start ascending, then name)`.

10.4 Session Lifecycle

- `schedule_session` creates a `Session{status=PROPOSED}` with participants `invited={ids}` (confirmed=false). Organizer is implicitly confirmed if they run `confirm_session` (explicit confirmation recommended for uniformity).
- `confirm_session` toggles participant's `confirmed=true`. When all invited participants **and** organizer are `confirmed=true`, set session `status=CONFIRMED`.
- `cancel_session` marks session `status=CANCELLED` and clears pending confirmations.
- Prevent new sessions that time-conflict with any `CONFIRMED` session for the organizer. *(A conflict is same `day` and `start`.)*

---

11. Error Handling & Edge Cases

Input Validation

- Invalid command → print `help` with usage.
- Missing required args → interactive prompt for the missing piece.
- Invalid `day` (`<0` or `>6`) or hour (`<0` or `>24`) → reject with message.
- Non-existent IDs (student, session) → reject with message.
- Duplicate email or course enrollment → reject with message.

- Availability overlap/adjacency without merge → auto-merge and inform user.
- Removing a slot that doesn't exist → warn, no-op.

Scheduling

- Propose session outside organizer's availability → reject.
- Invitee not enrolled in the course → reject.
- Inviting organizer or duplicate invitees → reject.
- Organizer already has a confirmed session at that slot → reject.
- Confirmation for a `CANCELLED` session → reject with message.
- Confirmation by a user who isn't a participant → reject.

Persistence

- File not found on first run → create empty files.
- Malformed record on load → skip and log warning to console (with line number).
- Write failure → print error, abort the operation (in-memory state unchanged).
- Atomic write: write to temp (e.g., `.tmp`), then `rename`.

Edge Cases

- User removes availability that contains a `PROPOSED`/`CONFIRMED` session → warn the user; allow removal, but keep the session unchanged.
- User removes course with future sessions → reject unless all related sessions are cancelled first.
- All invitees decline (never confirm) → session remains `PROPOSED`; organizer may cancel.
- Multiple availability blocks in the same day → normalized by merging.
- Empty profile (no courses or availability) → search/schedule commands fail with instructive messages.

---

12. Non-Functional Requirements

N-F1 Usability: Clear commands, consistent messages, `help` command with examples.
N-F2 Reliability: Atomic writes; tolerant of malformed input files.
N-F3 Performance: Operations complete within ~100ms for datasets up to: 1,000 students, 5,000 availability slots, 2,000 sessions.
N-F4 Portability: Builds on Windows/macOS/Linux with a C++17 compiler.
N-F5 Maintainability: Modular code; separation of concerns between CLI, domain, and storage layers.
N-F6 Security (Minimal): Passcode hashed with `std::hash` (not cryptographically secure). Files are plaintext.

N-F7 Testability: Deterministic functions (e.g., overlap detection) exposed for unit tests; sample seed data supported.

---

13. System Architecture & Modules

- **CLI Controller**: Parses commands/flags, routes to services, formats output.
- **ProfileService**: Create/edit profiles; uniqueness by email; passcode hashing.
- **CourseService**: Manage enrollments; validation; list courses.
- **AvailabilityService**: Add/remove slots; normalization/merge logic.
- **MatchService**: Compute overlaps; produce suggestions.
- **SessionService**: Create/confirm/cancel sessions; enforce constraints.
- **StorageAdapter**: Load/save CSV files; atomic writes; simple escaping.
- **Validation**: Input validators for email, course code, day/hour ranges.
- **Util**: ID generation, string split/trim, safe conversions.

Design Notes / C++ Best Practices

- Prefer `std::vector`, `std::unordered_map`, `std::string`, `std::optional`.
- `const`-correctness; pass by `const&` for large objects.
- No raw owning pointers; use values or RAII objects.
- Keep exceptions for unrecoverable I/O; otherwise return error codes/`optional`.
- Avoid global mutable state; inject services where practical.
- Separate headers/implementations; minimal inline in headers.

---

14. Data Validation Rules

- **Email**: Must contain `@` and a domain; recommend `.edu`, but not required.
- **Course Code**: Uppercase dept (2–5 letters) + space + 3–4 digits (e.g., `MATH 1080`, `CPSC 2120`).
- **Availability**: `0 ≤ start < end ≤ 24`, integers only.
- **Session**: Duration fixed at 1 hour; `start ∈ [0,23]`.

---

15. Use Cases

UC-1 Create Profile
Actor: Student

Pre: None
Main Flow: User runs `create_profile` → enters fields → profile persisted.
Post: Student exists with unique email.
Alt: Duplicate email → error message.

UC-2 Manage Availability
Main Flow: Add or remove slots; normalization ensures no overlaps.
Alt: Invalid ranges → rejection.

UC-3 Search Classmates (Exact Course)
Main Flow: `search_matches --course "CPSC 2120"` → system shows classmates and overlapping slots.
Alt: No overlaps → message.

UC-4 Schedule Session
Main Flow: Organizer proposes a slot, invites classmates; session `PROPOSED`.
Alt: Slot outside availability or conflicts → rejection.

UC-5 Confirm/Cancel Session
Main Flow: Participants confirm until all confirmed → `CONFIRMED`.
Alt: Any participant cancels → `CANCELLED`.

UC-6 View Sessions/Invitations
Main Flow: List pending invites and all sessions by status.

---

16. Sample CLI Flows

- > create_profile --name "Avery Tiger" --email averyt@clemson.edu
- Profile created: id=1
-
- > add_course --code "CPSC 2120"
- Course added.
-
- > add_availability --day 2 --start 14 --end 17
- Availability added (Tue 14:00–17:00). Merged if adjacent.
-
- > search_matches --course "CPSC 2120"
- Matches:
-   #7 Jordan Lee: Tue 15:00, Tue 16:00
-
- > schedule_session --course "CPSC 2120" --day 2 --start 15 --invite 7
- Session PROPOSED (id=31). Waiting for confirmations: [7, organizer].

-
- > confirm_session --id 31
- You have confirmed session 31. Status: CONFIRMED.
-
- > list_sessions
- CONFIRMED:
-   [31] CPSC 2120 Tue 15:00–16:00 Participants: you, #7 Jordan Lee

---

17. Acceptance Criteria

- All functional requirements (R-P*, R-C*, R-A*, R-S*, R-M*, R-V*, R-D*) demonstrably satisfied via CLI commands.
- Persistence verified by inspecting CSV files and reloading state.
- Error cases produce clear messages; invalid operations are rejected.
- Overlap logic correct on representative test data.
- Build passes on at least one Windows and one UNIX-like environment with C++17.

---

18. Testing Plan (High-Level)

- **Unit Tests**: Overlap detection, availability normalization, duplicate detection, session conflict detection.
- **Integration Tests**: CLI flows for profile, courses, availability, matching, scheduling, confirm/cancel, and persistence round-trip.
- **Negative Tests**: Invalid inputs (hours, days, unknown IDs), conflicting schedules, non-enrolled invitees, duplicate invites.
- **Data Corruption Simulation**: Malformed CSV lines are skipped with warnings.

---

19. Traceability Matrix (Excerpt)

Req	Feature	Module	Test

R-P1..P5	Profiles	ProfileService, StorageAdapter	UT-Profile-Create/Edit/Hash
R-C1..C4	Courses	CourseService, StorageAdapter	IT-Courses-Add/Remove/Persist
R-A1..A4	Availability	AvailabilityService	UT-Avail-Merge/Validate
R-S1..S4	Search/Match	MatchService	IT-Match-Overlaps
R-M1..M8	Sessions	SessionService	IT-Session-Lifecycle
R-V1..V3	Views	CLI Controller	IT-List-Sessions/Invites
R-D1..D3	Persistence	StorageAdapter	IT-Atomic-Write/Load-Skip

---

20. Risks & Mitigations

- **Risk**: Data loss on write. **Mitigation**: Atomic write via temp + rename.
- **Risk**: Weak passcode security. **Mitigation**: Documented as educational only; store hashes; local usage only.
- **Risk**: Confusing CLI. **Mitigation**: Consistent naming; `help` with examples.

---

21. Maintenance & Support

- Code organized by modules; minimal dependencies; clear build instructions (Makefile or CMake).
- Logging to stdout/stderr only.

---

22. Appendices

22.1 Build & Run (Example)

- # build
- c++ -std=c++17 -O2 -o study_buddy src/*.cpp
-
- # run
- ./study_buddy

22.2 Sample CSV Snippets

- # students.csv
- 1,Avery Tiger,averyt@clemson.edu,1084729381
- 7,Jordan Lee,jlee3@clemson.edu,
-
- # enrollments.csv
- 1,CPSC 2120
- 7,CPSC 2120
-
- # availability.csv
- 1,2,14,17
- 7,2,15,17
-
- # sessions.csv
- 31,CPSC 2120,2,15,1,1,CONFIRMED,
-
- # session_participants.csv
- 31,1,true
- 31,7,true

---

23. Sign-Off

By signing below, stakeholders agree that this SRS (Version 1.0) fully and accurately captures the required functionality. No changes are permitted after approval without a formal change request outside the scope of this assignment.

Instructor / Product Owner: ___________________________ Date: //____
Engineering Lead: ____________________________________ Date: //____

-

Study Buddy (CLI, C++17) — Pseudocode

The following pseudocode specifies program structure, data flow, algorithms, and error handling for the Study Buddy application, consistent with the previously approved SRS. It is organized by modules and requirements. Names are indicative; implementation in C++17 should follow similar decomposition.

---

1. Conventions

- `// comment` explains steps.
- `PROC` denotes a procedure (no return) and `FUNC` denotes a function (with return).
- `RAISE error(code, message)` prints an error and returns an error result to caller.
- All time values use weekly recurrence: `day ∈ {0..6}` (0=Sun), integer `hour ∈ {0..24}`, exclusive `end`.
- Duration for sessions is fixed at 1 hour (`start ∈ {0..23}`).

---

2. Data Structures (in-memory)

```
TYPE Student:
    id: INT
    name: STRING
    email: STRING
    pass_hash: OPTIONAL<SIZE_T>

TYPE Enrollment:
    student_id: INT
    course_code: STRING   // format: "DEPT ####"

TYPE Availability:
    student_id: INT
    day: INT              // 0..6
    start: INT            // 0..23
    end: INT              // 1..24, end > start

TYPE SessionStatus = { PROPOSED, CONFIRMED, CANCELLED }

TYPE Session:
    id: INT
    course_code: STRING
    day: INT
    start: INT
    duration: INT = 1
    organizer_id: INT
    status: SessionStatus
    cancel_reason: OPTIONAL<STRING>

TYPE SessionParticipant:
    session_id: INT
    student_id: INT
    confirmed: BOOL

TYPE DataStore:
    students: MAP<INT, Student>               // by student.id
    studentsByEmail: MAP<STRING, INT>         // email -> student.id
    enrollments: MULTIMAP<INT, Enrollment>    // key student_id
    enrollmentsByCourse: MULTIMAP<STRING, INT>// course_code ->
student_id
    availability: MULTIMAP<INT, Availability> // key student_id
    sessions: MAP<INT, Session>               // by session.id
    participants: MULTIMAP<INT, SessionParticipant> // key session_id
    nextStudentId: INT
    nextSessionId: INT
```

---

3. Persistent Storage (CSV-like files)

3.1 File Names

```
students.csv
enrollments.csv
availability.csv
sessions.csv
session_participants.csv
```

3.2 Atomic Write Helper

```
PROC atomic_write(path: STRING, lines: LIST<STRING>):
    tmpPath ← path + ".tmp"
    WRITE_ALL(tmpPath, JOIN(lines, "\n")) OR RAISE error("IO_WRITE",
"Failed to write temp")
    RENAME(tmpPath, path) OR RAISE error("IO_RENAME", "Failed to
rename temp")
```

3.3 Load/Save

Each file has a corresponding `load_*` and `save_*` routine:

```
PROC load_all(ds: INOUT DataStore):
    IF NOT FILE_EXISTS(students.csv): CREATE_EMPTY_FILES()
    load_students(ds)
    load_enrollments(ds)
    load_availability(ds)
    load_sessions(ds)
    load_participants(ds)
    recompute_indices(ds)
    set_next_ids(ds)

PROC save_all(ds):
    save_students(ds)
    save_enrollments(ds)
    save_availability(ds)
    save_sessions(ds)
    save_participants(ds)
```

- Each `load_*`:
    - Read file line by line.
    - Parse CSV fields (respecting quotes).
    - On parse failure: `PRINT "Warning: malformed line N in <file>; skipping."` and continue.
- Each `save_*`:
    - Serialize records to lines.
    - Call `atomic_write`.

```
PROC recompute_indices(ds):
    ds.studentsByEmail ← {}
    FOR EACH s IN ds.students: ds.studentsByEmail[s.email] ← s.id

    ds.enrollmentsByCourse ← {}
    FOR EACH e IN ds.enrollments: APPEND
ds.enrollmentsByCourse[e.course_code], e.student_id

PROC set_next_ids(ds):
    ds.nextStudentId ← (MAX key of ds.students) + 1 OR 1 IF empty
    ds.nextSessionId ← (MAX key of ds.sessions) + 1 OR 1 IF empty
```

---

4. Validation Utilities

```
FUNC is_valid_email(email: STRING) -> BOOL:
    RETURN CONTAINS(email, "@") AND CONTAINS(email, ".") AND NOT
STARTS_WITH(email, "@")

FUNC is_valid_course(code: STRING) -> BOOL:
    // Pattern: UPPER letters (2–5), space, digits (3–4)
    RETURN MATCH_REGEX(code, "^[A-Z]{2,5} [0-9]{3,4}$")
```

```
FUNC is_valid_day(d: INT) -> BOOL: RETURN 0 ≤ d ≤ 6
FUNC is_valid_hour(h: INT) -> BOOL: RETURN 0 ≤ h ≤ 24
FUNC is_valid_avail_range(start: INT, end: INT) -> BOOL:
    RETURN is_valid_hour(start) AND is_valid_hour(end) AND start < end
```

---

5. Availability Normalization & Overlap

```
// Merge a new slot into existing slots for student/day, coalescing
overlaps and adjacencies.
PROC add_normalized_availability(ds, student_id, day, start, end):
    IF NOT is_valid_day(day) OR NOT is_valid_avail_range(start, end):
        RAISE error("BAD_RANGE", "Invalid day/hour range")

    slots ← LIST of Availability where availability.student_id =
student_id AND availability.day = day
    newStart ← start; newEnd ← end
    merged ← EMPTY LIST

    // Step 1: separate non-overlapping; extend range where
overlapping/adjacent.
    FOR EACH slot IN slots:
        IF slot.end < newStart OR slot.start > newEnd:
            APPEND merged, slot
        ELSE:
            // Overlap or adjacency: extend new range
            newStart ← MIN(newStart, slot.start)
            newEnd ← MAX(newEnd, slot.end)

    // Step 2: remove old day slots for student; reinsert merged + new
    REMOVE all availability entries for (student_id, day)
    FOR EACH m IN merged: INSERT availability(m)
    INSERT availability(student_id, day, newStart, newEnd)

    save_availability(ds)   // persist immediately
    PRINT "Availability added/merged: day=", day, " ", newStart, "-",
newEnd
```

```
FUNC overlap_interval(a_start, a_end, b_start, b_end) -> OPTIONAL<(INT, INT)>:
    s ← MAX(a_start, b_start)
    e ← MIN(a_end, b_end)
    IF s < e: RETURN (s, e) ELSE RETURN NONE
```

---

6. Match Suggestion Logic

```
// Returns list of (classmate_id, classmate_name, list of (day, candidate_start_hours))
FUNC suggest_matches(ds, current_id: INT, course_code: STRING) -> LIST<Match>:
    IF NOT is_valid_course(course_code): RAISE error("BAD_COURSE", "Invalid course code")
    IF NOT enrolled(ds, current_id, course_code): RAISE error("NOT_ENROLLED", "You are not enrolled in the course")

    result ← EMPTY LIST
    mySlotsByDay ← availability slots for current_id grouped by day
    classmates ← UNIQUE student_ids from ds.enrollmentsByCourse[course_code] EXCLUDING current_id

    FOR EACH mate_id IN classmates:
        mateSlotsByDay ← availability slots for mate_id grouped by day
        candidates ← EMPTY LIST<(INT day, LIST<INT> hours)>

        FOR day IN 0..6:
            FOR EACH mySlot IN mySlotsByDay[day]:
                FOR EACH mateSlot IN mateSlotsByDay[day]:
                    ov ← overlap_interval(mySlot.start, mySlot.end, mateSlot.start, mateSlot.end)
                    IF ov EXISTS:
                        (s, e) ← ov
                        // candidate start hours for 1-hour session
                        hours ← LIST of integers h where s ≤ h < e AND 0 ≤ h ≤ 23
                        IF hours NOT EMPTY:
                            APPEND candidates, (day, hours)

        IF candidates NOT EMPTY:
            SORT candidates by (day ASC, min(hours) ASC)
            APPEND result, (mate_id, ds.students[mate_id].name,
candidates)

    // Sort result by earliest candidate slot, then name
    SORT result BY (min(day,hour) ASC, name ASC)
    RETURN result
```

Helper:

```
FUNC enrolled(ds, student_id, course_code) -> BOOL:
    RETURN EXISTS e IN ds.enrollments WHERE e.student_id = student_id
AND e.course_code = course_code
```

---

7. Session Scheduling & Lifecycle

7.1 Conflict Detection

```
FUNC has_conflict(ds, student_id, day, start) -> BOOL:
    FOR EACH sess IN ds.sessions:
        IF sess.status = CONFIRMED AND (sess.organizer_id = student_id
OR
            EXISTS p IN ds.participants WHERE p.session_id = sess.id
AND p.student_id = student_id AND p.confirmed = TRUE):
            IF sess.day = day AND sess.start = start: RETURN TRUE
    RETURN FALSE
```

7.2 Schedule Session (PROPOSED)

```
PROC schedule_session(ds, organizer_id, course_code, day, start,
invitees: LIST<INT>):
    // Validation
    IF NOT is_valid_course(course_code): RAISE error("BAD_COURSE")
    IF NOT is_valid_day(day) OR NOT (0 ≤ start ≤ 23): RAISE
error("BAD_TIME")
    IF NOT enrolled(ds, organizer_id, course_code): RAISE
error("NOT_ENROLLED_ORG")
    IF NOT within_availability(ds, organizer_id, day, start, start+1):
RAISE error("OUTSIDE_AVAIL_ORG")
    IF has_conflict(ds, organizer_id, day, start): RAISE
error("ORG_CONFLICT")

    // Normalize invitees
    uniqueInvitees ← UNIQUE(invitees) EXCLUDING organizer_id
    IF uniqueInvitees EMPTY: RAISE error("NO_INVITEES", "At least one
classmate is required")

    // Validate invitees
    FOR EACH uid IN uniqueInvitees:
        IF NOT ds.students CONTAINS uid: RAISE error("INV_ID",
"Unknown invitee id")
        IF NOT enrolled(ds, uid, course_code): RAISE
error("INV_NOT_ENROLLED", "Invitee not enrolled")
        // Note: Do not check their availability here; they will
confirm only if free.

    // Create session
    sid ← ds.nextSessionId; ds.nextSessionId ← ds.nextSessionId + 1
    sess ← Session{ id=sid, course_code, day, start, duration=1,
organizer_id, status=PROPOSED, cancel_reason=NULL }
    ds.sessions[sid] ← sess

    // Participants: organizer (confirmed=false initially for
uniformity), invitees (confirmed=false)
    INSERT ds.participants, SessionParticipant{sid, organizer_id,
FALSE}
    FOR EACH uid IN uniqueInvitees:
        INSERT ds.participants, SessionParticipant{sid, uid, FALSE}

    save_sessions(ds); save_participants(ds)
    PRINT "Session PROPOSED id=", sid, " awaiting confirmations."
```

7.3 Confirm Session

```
PROC confirm_session(ds, session_id, actor_id):
    IF session_id NOT IN ds.sessions: RAISE error("NO_SESSION")
    sess ← ds.sessions[session_id]
    IF sess.status = CANCELLED: RAISE error("CANCELLED", "Cannot
confirm a cancelled session")

    part ← FIND participant WHERE session_id = session_id AND
student_id = actor_id
    IF part NOT FOUND: RAISE error("NOT_PARTICIPANT")

    // Validate actor's constraints at confirm-time
    IF has_conflict(ds, actor_id, sess.day, sess.start): RAISE
error("TIME_CONFLICT")
    IF NOT within_availability(ds, actor_id, sess.day, sess.start,
sess.start+1): RAISE error("OUTSIDE_AVAIL")

    part.confirmed ← TRUE
    UPDATE ds.participants

    // If all participants confirmed → CONFIRMED
    IF ALL participants with session_id = session_id HAVE confirmed =
TRUE:
        sess.status ← CONFIRMED
        UPDATE ds.sessions

    save_participants(ds); save_sessions(ds)
    PRINT "Session ", session_id, " status: ", sess.status
```

7.4 Cancel Session

```
PROC cancel_session(ds, session_id, actor_id, reason: STRING):
    IF session_id NOT IN ds.sessions: RAISE error("NO_SESSION")
    sess ← ds.sessions[session_id]

    // Only participants can cancel
    IF NOT EXISTS p WHERE p.session_id = session_id AND p.student_id =
actor_id:
        RAISE error("NOT_PARTICIPANT")

    sess.status ← CANCELLED
    sess.cancel_reason ← reason
    UPDATE ds.sessions
    save_sessions(ds)
    PRINT "Session ", session_id, " cancelled."
```

7.5 Helpers

```
FUNC within_availability(ds, student_id, day, start, end) -> BOOL:
    slots ← availability slots for student_id where slot.day = day
    FOR EACH slot IN slots:
        IF slot.start ≤ start AND end ≤ slot.end: RETURN TRUE
    RETURN FALSE
```

---

8. Profile & Enrollment Management

```
PROC create_profile(ds, name, email, passcode_optional):
    IF NOT is_valid_email(email): RAISE error("BAD_EMAIL")
    IF email IN ds.studentsByEmail: RAISE error("DUP_EMAIL")
    id ← ds.nextStudentId; ds.nextStudentId ← ds.nextStudentId + 1
    pass_hash ← HASH(passcode_optional) IF passcode_optional EXISTS
ELSE NULL
    s ← Student{id, name, email, pass_hash}
    ds.students[id] ← s
    ds.studentsByEmail[email] ← id
    save_students(ds)
    PRINT "Profile created id=", id

PROC edit_profile_name(ds, id, new_name):
    IF id NOT IN ds.students: RAISE error("NO_STUDENT")
    ds.students[id].name ← new_name
    save_students(ds)

PROC edit_profile_email(ds, id, new_email):
    IF NOT is_valid_email(new_email): RAISE error("BAD_EMAIL")
    IF new_email IN ds.studentsByEmail: RAISE error("DUP_EMAIL")
    old_email ← ds.students[id].email
    ds.students[id].email ← new_email
    REMOVE ds.studentsByEmail[old_email]
    ds.studentsByEmail[new_email] ← id
    save_students(ds)

PROC add_course(ds, student_id, course_code):
    IF NOT is_valid_course(course_code): RAISE error("BAD_COURSE")
    IF enrolled(ds, student_id, course_code): RAISE
error("DUP_COURSE")
    INSERT ds.enrollments, Enrollment{student_id, course_code}
    APPEND ds.enrollmentsByCourse[course_code], student_id
    save_enrollments(ds)

PROC remove_course(ds, student_id, course_code):
    // Reject removal if any non-cancelled future sessions exist for
this course
    IF EXISTS sess WHERE sess.course_code=course_code AND sess.status
≠ CANCELLED AND (
        sess.organizer_id=student_id OR EXISTS participant where
participant.session_id=sess.id AND participant.student_id=student_id):
        RAISE error("SESSIONS_EXIST", "Cancel related sessions first")

    REMOVE all Enrollment entries for (student_id, course_code)
    REMOVE student_id from ds.enrollmentsByCourse[course_code]
    save_enrollments(ds)
```

---

9. CLI Controller

9.1 Startup

```
MAIN():
    ds ← DataStore()
    load_all(ds)
    current_user ← NULL
    PRINT "Study Buddy CLI — type 'help' for commands."
    LOOP:
        INPUT line
        IF line EMPTY: CONTINUE
        cmd, args ← parse_command(line)

        SWITCH cmd:
            CASE "help": print_help()
            CASE "exit": BREAK

            CASE "create_profile":
                name ← arg_or_prompt(args, "--name")
                email ← arg_or_prompt(args, "--email")
                pass ← optional_arg(args, "--passcode")
                create_profile(ds, name, email, pass)
                current_user ← ds.studentsByEmail[email]

            CASE "login":                      // optional convenience
                email ← arg_or_prompt(args, "--email")
                current_user ← ds.studentsByEmail[email] OR RAISE
error("NO_SUCH_USER")

            CASE "edit_profile":
                REQUIRE_LOGGED_IN(current_user)
                IF "--name" in args: edit_profile_name(ds,
current_user, args["--name"])
                IF "--email" in args: edit_profile_email(ds,
current_user, args["--email"])

            CASE "add_course":
                REQUIRE_LOGGED_IN(current_user)
                code ← arg_or_prompt(args, "--code")
                add_course(ds, current_user, code)

            CASE "remove_course":
                REQUIRE_LOGGED_IN(current_user)
                code ← arg_or_prompt(args, "--code")
                remove_course(ds, current_user, code)
            CASE "list_courses":
                REQUIRE_LOGGED_IN(current_user)
                list_courses(ds, current_user)

            CASE "add_availability":
                REQUIRE_LOGGED_IN(current_user)
                day ← INT arg_or_prompt(args, "--day")
                start ← INT arg_or_prompt(args, "--start")
                end ← INT arg_or_prompt(args, "--end")
                add_normalized_availability(ds, current_user, day,
start, end)

            CASE "remove_availability":
                REQUIRE_LOGGED_IN(current_user)
                day ← INT arg_or_prompt(args, "--day")
                start ← INT arg_or_prompt(args, "--start")
                end ← INT arg_or_prompt(args, "--end")
                remove_exact_availability(ds, current_user, day,
start, end)

            CASE "list_availability":
                REQUIRE_LOGGED_IN(current_user)
                list_availability(ds, current_user)

            CASE "search_matches":
                REQUIRE_LOGGED_IN(current_user)
                code ← arg_or_prompt(args, "--course")
                matches ← suggest_matches(ds, current_user, code)
                print_matches(matches)

            CASE "schedule_session":
                REQUIRE_LOGGED_IN(current_user)
                code ← arg_or_prompt(args, "--course")
                day ← INT arg_or_prompt(args, "--day")
                start ← INT arg_or_prompt(args, "--start")
                inviteStr ← arg_or_prompt(args, "--invite")    //
e.g., "7,12"
```

```
                invitees ← parse_id_list(inviteStr)
                schedule_session(ds, current_user, code, day, start,
invitees)

            CASE "confirm_session":
                REQUIRE_LOGGED_IN(current_user)
                sid ← INT arg_or_prompt(args, "--id")
                confirm_session(ds, sid, current_user)

            CASE "cancel_session":
                REQUIRE_LOGGED_IN(current_user)
                sid ← INT arg_or_prompt(args, "--id")
                reason ← arg_or_default(args, "--reason", "No reason
provided")
                cancel_session(ds, sid, current_user, reason)

            CASE "list_sessions":
                REQUIRE_LOGGED_IN(current_user)
                list_sessions(ds, current_user)

            CASE "list_invitations":
                REQUIRE_LOGGED_IN(current_user)
                list_invitations(ds, current_user)

            DEFAULT:
                PRINT "Unknown command. Type 'help'."
```

9.2 List/Print Helpers

```
PROC list_courses(ds, student_id):
    FOR EACH e IN ds.enrollments WHERE e.student_id = student_id:
        PRINT e.course_code

PROC list_availability(ds, student_id):
    SLOTS ← availability for student_id
    GROUP by day; SORT by start
    FOR EACH day in 0..6:
        FOR EACH slot IN day group: PRINT "Day", day, ":", slot.start,
"-", slot.end

PROC print_matches(matches):
    IF matches EMPTY: PRINT "No matches found."
    ELSE:
        FOR EACH (mate_id, name, cands) IN matches:
            DISPLAY name and earliest few candidate hours per day

PROC list_sessions(ds, student_id):
    COLLECT sessions where participant or organizer is student_id
    GROUP by status; SORT by (day, start)
    PRINT grouped lists with participant names and confirmation states

PROC list_invitations(ds, student_id):
    FOR EACH p IN ds.participants WHERE p.student_id=student_id AND
p.confirmed=FALSE:
        sess ← ds.sessions[p.session_id]
        IF sess.status = PROPOSED: PRINT details
```

9.3 Remove Exact Availability

```
PROC remove_exact_availability(ds, student_id, day, start, end):
    IF NOT is_valid_day(day) OR NOT is_valid_avail_range(start, end):
        RAISE error("BAD_RANGE")
    removed ← FALSE
    FOR EACH slot IN availability for (student_id, day):
        IF slot.start = start AND slot.end = end:
            DELETE slot
            removed ← TRUE
            BREAK
    IF NOT removed: PRINT "No matching slot found to remove."
    save_availability(ds)
```

---

10. Error Messages (standardized)

```
PROC RAISE error(code: STRING, message: STRING):
    PRINT "[ERROR] ", code, ": ", message
    RETURN
```

Additionally, for invalid command usage, invoke:

```
PROC print_help():
    PRINT "Commands:"
    PRINT "  create_profile --name <str> --email <str> [--passcode
<str>]"
    PRINT "  login --email <str>"
    PRINT "  edit_profile [--name <str>] [--email <str>]"
    PRINT "  add_course --code <DEPT NUM>"
    PRINT "  remove_course --code <DEPT NUM>"
    PRINT "  list_courses"
    PRINT "  add_availability --day <0..6> --start <0..23> --end
<1..24>"
    PRINT "  remove_availability --day <0..6> --start <..> --end <..>"
    PRINT "  list_availability"
    PRINT "  search_matches --course <DEPT NUM>"
    PRINT "  schedule_session --course <DEPT NUM> --day <0..6> --start
<0..23> --invite <id,id,..>"
    PRINT "  confirm_session --id <session_id>"
    PRINT "  cancel_session --id <session_id> [--reason <text>]"
    PRINT "  list_sessions"
    PRINT "  list_invitations"
    PRINT "  help | exit"
```

---

11. Testing Hooks (unit-friendly pure functions)

Implement the following as standalone functions to enable deterministic unit tests:

```
FUNC normalize_slots(existing_slots_for_day: LIST<Availability>,
new_slot: Availability)
        -> LIST<Availability>   // returns fully merged list

FUNC compute_overlaps(my_slots: LIST<Availability>, mate_slots:
LIST<Availability>)
```

```
        -> LIST<(day: INT, hours: LIST<INT>)>

FUNC conflict_exists(sessions: LIST<Session>, participants: LIST<SessionParticipant>, student_id: INT, day: INT, start: INT)
        -> BOOL
```

---

12. Performance & Persistence Notes

- After any mutation, call targeted `save_*` (not necessarily `save_all`) to minimize I/O while ensuring durability.
- For small datasets (as per SRS), linear scans are acceptable. Use `unordered_map` and grouping to keep operations ≤ O(n) per command.

---

13. Security Notes (educational scope)

- `pass_hash` computed with standard library hash; used only to check equality if a passcode feature is activated. Do not claim cryptographic security.
- Files are plaintext. No encryption or access control beyond host OS.

---

14. Acceptance Mapping (traceability)

- **Profiles (R-P1..P5):** `create_profile`, `edit_profile_*`, `students.csv`, hash on set.
- **Courses (R-C1..C4):** `add_course`, `remove_course`, enrollment persistence, dup prevention.
- **Availability (R-A1..A4):** `add_normalized_availability`, `remove_exact_availability`, validation.
- **Search/Match (R-S1..S4):** `suggest_matches`, overlap computation, empty-result handling.
- **Sessions (R-M1..M8):** `schedule_session`, `confirm_session`, `cancel_session`, constraints.
- **Viewing (R-V1..V3):** `list_sessions`, `list_invitations`, standardized messages.
- **Persistence (R-D1..D3):** immediate saves, `atomic_write`, load-skip on malformed lines.

Study Buddy CLI — Test Plan and Results

Test Plan (Functional Requirements)

Scope

Validate the **core functional requirements** of the Study Buddy CLI per the approved SRS. In scope: profile creation, course enrollment/listing, availability add/remove, match suggestion (exact course + overlapping availability), session scheduling, session confirmation, and invalid/duplicate input handling.

Out of Scope

Non-functional testing (performance, security hardening, concurrency), GUI, networking.

Test Environment

- OS/Terminal with a C++17 binary built from `make`.
- Data directory: `./data/` (CSV files created on first run). Seed data may be used from the repo.

Assumptions

- IDs auto-increment and persist in CSVs.
- Commands are entered with correct syntax except in negative tests.
- Time granularity is 1 hour; `day ∈ [0..6]`, `start ∈ [0..23]`, `end ∈ [1..24]` (exclusive end).

Test Data & Setup (minimal)

1. Start with empty or provided seed CSVs in `data/`.
2. Create/login as **User A** (organizer). Optionally create **User B** to act as a classmate.
3. Use sample course code `"CPSC 2120"` and availability on Tuesday (`day=2`).

Test Cases & Results

Legend: **PASS** means behavior matched SRS; errors shown are expected messages/codes from CLI.

ID	Area	Precondition / Steps (abridged)	Expected Result	Actual	Outcome
T01	Create profile	`create_profile --name "Avery Tiger" --email avery@clemson.edu`	New student created, ID assigned; record in `students.csv`	Created with ID; persisted	PASS
T02	Duplicate email	Run T01, then `create_profile --name "Dup" --email avery@clemson.edu`	Error: duplicate email (no new record)	Error shown; no new row	PASS
T03	Add course	Logged in as User A → `add_course --code "CPSC 2120"`	Course added; `enrollments.csv`updated	Added & persisted	PASS
T04	List courses	`list_courses`	Course appears in list	Displayed as enrolled	PASS
T05	Duplicate course	After T03 → `add_course --code "CPSC 2120"`	Error: duplicate course	Error shown	PASS
T06	Add availability	`add_availability --day 2`	Slot stored/merged in `availability.csv`	Added/merged	PASS

		`--start 14 --end 17`			
T07	Remove availability	`remove_availability --day 2 --start 14 --end 17`	Slot removed	Removed	PASS
T08	Remove non-existent availability	Remove same slot again	Message: "No matching slot found"	Message shown	PASS
T09	Suggest matches (overlap exists)	Setup: User A & User B both enrolled in `CPSC 2120`; A has `2:14–17`, B has `2:15–17` → `search_matches --course "CPSC 2120"`	List User B with overlapping hours (e.g., Day 2 [15,16])	Shown with hours	PASS
T10	Suggest matches (no overlap)	B has availability on different day/time	Message: "No matches found"	Message shown	PASS
T11	Schedule valid session	A → `schedule_session --course "CPSC 2120" --day 2 --start 15 --invite <B_id>`	Session created `PROPOSED`, participants pending	Created as PROPOSED	PASS
T12	Schedule outside availability	A lacks `2:15–16` slot → schedule at 15	Error: `OUTSIDE_AVAIL_ORG`	Error shown	PASS

T13	Confirm session (participant)	B → `confirm_session --id <session_id>`; A also confirms	B confirmation recorded; when all confirm → `CONFIRMED`	Status transitions to CONFIRMED	PASS
T14	Confirm by non-participant	Another user (not invited) confirms	Error: `NOT_PARTICIPANT`	Error shown	PASS
T17	Invalid course format	`add_course --code "cs 2120"` (wrong case/format)	Error: `BAD_COURSE`	Error shown	PASS
T18	Schedule with non-enrolled invitee	Invite user not enrolled in course	Error: `INV_NOT_ENROLLED`	Error shown	PASS
T19	Cancel session	Any participant → `cancel_session --id <session_id> --reason "Conflict"`	Status set to `CANCELLED`, reason saved	Cancelled with reason	PASS

Traceability (Core)

Requirement Area	Test IDs
Profile create	T01–T02
Add/List courses	T03–T05

Add/Remove availability	T06–T08
Suggest matches	T09–T10
Schedule sessions	T11–T12,, T18
Confirm sessions	T13–T14
Invalid/duplicates	T02, T05, T08, T12, T14–T18

Summary

- All **core functional** tests executed against the CLI build.
- All positive paths **PASSED**.
- Negative tests correctly returned expected error messages/codes.
- The deliverable meets SRS functional requirements for the Waterfall Testing Phase (no requirement changes).

Test Output:

```
TestID,Title,Outcome,Detail
T01,Create profile,PASSED,
T02,Duplicate email rejected,PASSED,
T03,Add course CPSC 2120,PASSED,
T05,Duplicate course rejected,PASSED,
T04,List courses includes CPSC 2120,PASSED,
T06,Add availability 2:14-17,PASSED,
T08,Remove non-existent availability reports message,PASSED,
T07,Remove existing availability,PASSED,
T09,Suggest matches (overlap exists),PASSED,
T10,Suggest matches (no overlap),PASSED,
T11,Schedule valid session,PASSED,
T12,Schedule outside availability rejected,PASSED,
T18,Schedule with non-enrolled invitee rejected,PASSED,
T13,Confirm session transitions to CONFIRMED,PASSED,org=1 inv=1 confirmed=1
T14,Confirm by non-participant rejected,PASSED,
T19,Cancel session sets CANCELLED,PASSED,
```