1. Design Plan (15 points)

Objective: Ensure the trainee creates a structured design before implementation.

Criterion	Points	Description
High-Level Overview	3	Provides a clear summary of how the calculator will be structured.
Modular Design	3	Breaks functionality into distinct modules (e.g., parsing, operations, testing).
Data Flow Description	3	Describes how data moves through the system (from input parsing to computation and output).
Error Handling Strategy	3	Documents how errors (e.g., bad input, divide-by-zero) will be detected and handled.
Scalability Considerations	3	Discusses how the design can be extended (e.g., adding 64-bit support).

2. Create the Project Structure and Setup (10 points)

Objective: Ensure a well-organized project structure with a proper build system.

Criterion	Points	Description
Project Structure	3	Includes `src/`, `include/`, and `references/` directories.
CMake Configuration	3	`CMakeLists.txt` is correctly structured and generates a working build system.
Build Script	2	`build.sh` exists and properly automates build steps (`mkdir build`, `cmake ..`, `make`).
Git Version Control	2	Repository is properly initialized, commits follow good practices (descriptive messages, logical commits).

3. Parse Command-Line Arguments (15 points)

Objective: Correctly handle command-line arguments and input validation.

Criterion	Points	Description
Correct Parsing	4	Properly extracts `operand1`, `operator`, and `operand2` from arguments.
Input Validation	4	Rejects malformed input (e.g., incorrect number of arguments, missing spaces).
Handling Special Characters	3	Accounts for shell-escaped operators (`<<`, `>>`, `<<<`, `>>>`, `&`, `
Error Messaging	2	Provides meaningful error messages for invalid input.
Graceful Exit	2	Ensures program does not crash on bad input.

4. Build a Library of 32-bit Math Functions (25 points)

Objective: Implement mathematical operations in a modular way with overflow/underflow detection.
Trainees cannot cast operands to `int64_t` for overflow checks.

Criterion	Points	Description
Separation of Concerns	4	Each operation is implemented in a separate function.

Criterion	Points	Description
Modularity for Portability	4	Functions are structured to support easy transition to 64-bit or other sizes.
Correct Implementation	5	Implements all required operations (`+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `
Overflow Detection (Arithmetic Ops)	4	Properly detects integer overflow/underflow without relying on `int64_t` casting.
Bitwise Operations Handling	4	Correctly implements logical and circular bitwise operations.
Error Handling & Edge Cases	4	Handles divide-by-zero, left shift overflow, and other corner cases safely.

5. Build a Test Suite Using CUnit (20 points)

Objective: Write and execute unit tests to validate mathematical operations.

Criterion	Points	Description
Test Coverage	6	Includes test cases for all arithmetic and bitwise operations.
Edge Case Handling	5	Tests cover overflow, underflow, zero division, and boundary conditions.
Correct Use of CUnit	4	Uses `CUnit` properly (setup, teardown, assertions).
Automated Testing	3	Tests are included in the build system (`make test` or similar).
Error Reporting in Tests	2	Fails test cases gracefully and provides meaningful output.

6. Write a README (10 points)

Objective: Provide documentation for usage, compilation, and testing.

Criterion	Points	Description
Overview of the Project	3	Clearly explains project purpose and functionality.
Build & Run Instructions	3	Provides detailed instructions for compiling and running `simplecalc`.
Usage Examples	2	Includes command-line examples and expected outputs.
Testing Instructions	2	Explains how to run tests and interpret results.

7. Formatting and Standards Compliance (15 points)

Objective: Ensure code adheres to best practices in style and maintainability.

Criterion	Points	Description
Follows CSD-T Style Guide	5	Code adheres to the **Cyber Solutions Development - Tactical** style guide.
Follows SRP (Single Responsibility Principle)	3	Each function and module has a well-defined, singular purpose.
Separation of Concerns	3	Input parsing, computation, and output are properly separated.
Consistent Formatting	2	Uses proper indentation, naming conventions, and whitespace.
Code Documentation	2	Includes meaningful comments and function headers.

Total: 110 points

- **Excellent (90-100%)** → Code is clean, modular, well-documented, and fully functional.
- **Good (80-89%)** → Mostly complete, with minor errors or missing edge cases.
- **Satisfactory (70-79%)** → Some missing functionality or major issues in one or more components.
- **Needs Improvement (60-69%)** → Significant issues with implementation, testing, or structure.
- **Failing (<60%)** → Major structural flaws, missing core functionality, or improper handling of errors.