ASPL – Introduction Assignment

Part 1:

Configurations and Build Files

Pyproject.toml (root): Provides project's metadata required for installation of a python project – common for projects designed to be widely installed. Lists the two main packages – caf and libcaf

Dockerfile: Defines a complete development environment, installing all necessary tools (Python, C++ compiler, CMake, OpenSSL, pytest, etc.) on top of a chosen image (ubuntu in our case). meant to facilitate supporting and easy to use environment for users, independent of a specific OS.

Makefile: Provides convenient make targets for various commands the user might do in the project, like building the Docker development environment, testing, installing the project etc. Acts as the primary interface for users to interact with the project.

CMakeLists.txt: CMake build configuration for the C++ extension module, defines compilation targets, source files, linking requirements and output location. Can be described as a cross-platform automation tool that generates build files.

Core Source Files

cli_commands: -High level implementation of the project's main commands. It defines the API for each function and receives it from the cli file after the parsing of the user input.

- Manages the error cases and provides user feedback.
- Called by cli.py after argument parsing and delegates the actual execution of those functions to repository.py.

libcaf/libcaf/init.py: Marks libcaf as a Python package, giving constants.py, ref.py and plumbing.py the same namespace. Defines what users see when they import libcaf.

Repository.py: The main class of the project. Implements the core logic of CAF's repository and encapsulates all repository operations. It manages initialization, commit creation, tree and blob storage, branch handling, and computing diffs between commits. provides a single, cohesive abstraction that knows how to interact with the repository's data and state. It uses plumbing.py for low-level operations and ref.py to manage references such as branches and HEAD. In turn, the CLI layer calls methods from repository.py to execute user requests like init, commit, or diff.

ref.py: Defines reference types (HashRef, SymRef) and provides functions for reading, writing, and resolving references. Handles the distinction between direct refs (pointing to commits) and symbolic refs (pointing to other refs). Used by repository.py for all ref operations.

Constants.py: Keeps all fixed names that are widely used in the project, avoiding mismatches with different modules and allowing a clean code.

Plumbing.py: Provides low-level helper functions (save_commit, load_commit, save_tree, load_tree, hash_object). Prevents higher level functions dealing with serialization\deserialization, file descriptor, and additional under-the-hood operations. Some of those functions are wrappers to C++ functions – defined in pybind module (bind.cpp).

Core library

Caf.cpp: collects performance-critical C++ operations - hashing, opening content, deleting content, and saving objects – so bind.cpp can expose them to python as clean API.

Bind.cpp: C++ file that uses pybind11 to expose C++ functions and classes (Blob, Commit, Tree, save/load, hash) to Python as a module.

hash_types: Implements the hashing functions for CAF's object types (Blob, Tree, Commit). Each object type has a specific serialization format that's hashed to produce its content-addressed identifier. Used by repository operations when creating objects.

Object_io: Implements serialization and deserialization for complex objects (Commit, Tree). Provides functions that read/write binary representations to the object database (save_commit, load_commit, save_tree, load_tree). Uses functions from caf.cpp and hash_types.cpp to compute storage location and writing content.

Blob.h, commit.h, tree.h, tree_record.h: Define the data structures that represent CAF objects: blob.h – file content, tree.h and tree_record.h – directory structure and commit – a pointer to a tree. All members are const (immutable), preventing accidental modification.

Test Infrastructure

tests/conftest.py: Pytest configuration file defining fixtures used across all tests. Provides reusable test utilities and reduces code duplication in tests.

CLI Command Tests: Ensures CLI commands work correctly and handle errors gracefully. Validates the user-facing behavior of CAF. Each file tests one command's behavior, including:

- success cases (command works as expected)
- error cases (invalid arguments, missing repository, etc.),
- edge cases (empty branches, corrupted data, etc.)
- output verification (checking what's printed to stdout).

Library Tests: Unit tests for the core libcaf library. Tests internal functionality without going through CLI and focuses on correctness of individual components, including:

- Error injection - tests like test_log_corrupted_commit_raises_error deliberately corrupt data to verify error handling.
- Boundary testing: tests that validate input such empty strings, invalid refs, None values etc.