DevOps CI/CD Challenge

Design and implement a fully automated CI/CD pipeline for a trivial **C++ library**, following the release process described in the challenge. The library itself is intentionally trivial; the difficulty is in implementing the release flow correctly in CI/CD.

The objective of this challenge is to evaluate your ability to translate a release process into a working CI/CD implementation. The application itself is intentionally trivial; the complexity lies in the CI/CD workflow. We are assessing your understanding of the release process (including how you would handle different scenarios), as well as the quality, correctness, and clarity of the automation you implement.

Project Scope

- Create a tiny C++ library (e.g., int sum(int a, int b)).
- Use **CMake**.
- Include **unit tests** (GoogleTest).
- Package and publish the library via **Conan**.
- Use GitHub as the source repository.
- Use GitHub Actions for CI/CD.
- Identify and extract reusable workflows/actions into a separate “shared” repo.

Platform targets: Linux, Windows, MacOS

Conan packaging requirements (Conan 2)

- Provide conanfile.py (Conan 2) that produces an installable package:
    - headers in include/
    - libraries in lib/
    - CMake package metadata so consumers can use find_package(mylib CONFIG REQUIRED)
- Use Conan 2 profiles per platform/toolchain (Linux/macOS/Windows). Profiles must capture at least: os, arch, compiler, build_type
- Publishing target:
    - Use two Conan remotes:
        - conan-rc (release candidates / pre-merge)
        - conan-stable (final releases / post-merge)
    - Package reference format:
        - RC: mylib/<X.Y.Z>-dev-<short-sha>
        - Release: mylib/<X.Y.Z>

Required CI/CD Capabilities

Your implementation must include:

Pull Request Verification, on every pull request:

- Enforce up-to-date branch with main
- Enforce linear history
- Run:
    - Format/lint (e.g., clang-format check; optional clang-tidy)
    - CMake configure + build
    - Unit tests (ctest)
    - Dependencies locked / pinned:
        - pin Conan dependencies (lockfile or explicit versions + profiles)
        - ensure deterministic builds
    - The PR cannot be merged unless all checks pass

Label-Driven Workflow

- verify label:
    - Triggers integration/E2E tests (trivial is fine but must exist). Example:
        - Build a tiny “consumer” CMake project that does:
            - installs the dependency via Conan: conan install --requires=mylib/<ref> ...
            - uses find_package(mylib CONFIG REQUIRED)
            - links against the package and runs a small executable test
- publish label:
    - Before merge, it must:
        - Build and upload a release-candidate Conan package to the conan-rc remote
        - Block if the release version already exists in conan-stable
        - Produce publishable artifacts before merge (RC packages for each platform in the matrix)

Release on Merge, when a PR with the publish label is merged:

- Publish the Conan package(s) to the remote
- Create a Git tag with the version (vX.Y.Z)
- Create a GitHub Release

Versioning

- Use semantic versioning
- The version must be explicitly bumped in the PR
- Pre-merge builds must include a version suffix (e.g. 1.0.0-dev-<short-sha>)

Conan mapping suggestion (Conan 2)

- Release candidate package reference:

    - mylib/X.Y.Z-dev-<short-sha> uploaded to remote: conan-rc
- Release package reference:
    - mylib/X.Y.Z uploaded to remote: conan-stable

Constraints

- You must **automate the branch protection set up** (e.g. with gh or importing settings)
- **No manual steps** allowed after PR approval
- The **reusable actions** must be identified and extracted into another generic repo

Deliverables

- GitHub repository URLs (main a reused actions)
- Short README explaining how the pipeline and release works (labels, etc.)
- Short recording showing it in action and explaining the steps