CI/CD Documentation for Advanced Tic Tac Toe Project

Introduction

This document provides a comprehensive explanation of the Continuous Integration/ Continuous Deployment (CI/CD) pipeline configured for the Embedded Systems Project. The CI/CD pipeline automates the processes of building, testing, and deploying the application, ensuring code quality, reliability, and efficient delivery. This automation is very important in embedded systems projects because it helps ensure the code is always tested properly and deployed in a consistent way.
The CI/CD pipeline is set up in the **ci.yml** file and uses GitHub Actions to run all the steps automatically. It includes important stages such as setting up the environment, checking code quality, scanning for security issues, running tests, measuring performance, and getting the project ready for deployment.

CI/CD Workflow Overview

The CI/CD workflow is triggered on any push or pull request to any branch.
This ensures that changes across all branches, not just main or master, are automatically validated. This flexible trigger strategy enhances testing coverage and early detection of issues during development. The workflow consists of two main jobs: build and deploy.

Build Job

The build job is responsible for compiling the application, running various checks, and preparing artifacts for deployment. It runs on an “ubuntu-22.04” environment and includes the following steps:

1. Checkout code

This step uses the `actions/checkout@v4` action to retrieve the source code from the repository. This is the foundational step for any CI/CD pipeline, ensuring that the latest version of the code is available for subsequent build and test processes.

2. Install Qt 6 and dependencies

This crucial step installs all necessary development tools and libraries required for the project. Given that the project utilizes Qt 6, the pipeline ensures that the correct version of Qt, along with essential build tools like “cmake”, “g++”, and “build-essential”, are present. Additionally, it installs “libgl1-mesa-dev” and “libglu1-mesa-dev” for graphics, “libfontconfig1”, “libsqlite3-dev”, “cppcheck” for static analysis, and “valgrind” for memory debugging.

This comprehensive installation ensures a consistent and reproducible build environment. No separate installation is required for QTest, as it is included with the Qt framework.

4. Code Quality Analysis

This step integrates cppcheck for static code analysis. cppcheck is a powerful tool that detects various types of errors, including memory leaks, buffer overflows, and uninitialized variables, without executing the code. The output is generated in XML format (cppcheck_report.xml), allowing for automated parsing and reporting of code quality metrics.

5. Security Scan

This step performs a basic security scan using grep to search for keywords such as "password", "secret", or "key" inside .cpp and .h files. Though basic, this helps prevent accidental credential exposure.

6. Build Project with qmake6

This is the core build step. The pipeline uses qmake6 Embedded.pro to generate the Makefile and then compiles the code using make -j$(nproc), utilizing all available processor cores.

7. Run Unit and Integration Tests

This step executes both unit and integration tests using QTest. It checks for the presence of dedicated directories Unit-tests_tests and Integration_Test, and builds their .pro files using qmake6. After compilation with make, the test executables are run using xvfb-run in headless mode, and their outputs are saved in structured XML format under the test_results/ directory. This detailed and automated process ensures robust test coverage and enables easy parsing of results.

8. Prepare Deployment Package

This step structures the necessary deployment files in a deployment_package directory. It includes the Embedded executable and relevant configuration files (.conf, .ini, .db), and generates a build_info.txt file containing the build date, commit hash, and branch name for traceability.

9. Upload Comprehensive Artifacts

Artifacts such as the compiled executable, deployment package, test results, performance reports, cppcheck results, object files, and Makefile are uploaded using actions/upload-artifact@v4. Artifacts are versioned using GitHub run number and retained for 30 days.

Deploy Job

The deploy job is responsible for taking the artifacts generated by the build job and deploying them. This job is conditional and only runs when a push event occurs on the main branch. This ensures that only thoroughly tested and validated code is deployed to production. The deploy job includes the following steps:

1. Checkout code

This step checks out the repository code using actions/checkout@v4, just like in the build job. Although deployment primarily relies on the artifacts produced by the build job, having the full source code available allows the deployment process to generate versioning metadata, release tags, or future release notes.

2. Download build artifacts

This step uses `actions/download-artifact@v4` to retrieve the complete-build artifacts that were uploaded by the build job. This ensures that the deployment process operates on the exact same tested binaries and files that were produced during the build phase, keeping everything consistent and avoiding any differences or unexpected issues.

3. Prepare Deployment Directory

This step organizes the downloaded artifacts into a structured `deployment` directory. It copies the `Embedded` executable and the contents of the deployment_package into this directory. It also sets execute permissions on the `Embedded` executable and creates a `DEPLOYMENT_INFO.txt` file with details like the build date, version (derived from the GitHub run number), and commit hash. This preparation ensures that the application is ready for deployment in a well-defined structure.

4. Create Release Archive (.zip)

To facilitate easy distribution and versioning, this step creates a `.zip` archive of the entire `deployment` directory. The archive is named `tic-tac-toe-v$ {{ github.run_number }}.zip`, clearly indicating the version based on the GitHub run number. This compressed package serves as a portable release artifact.

5. Package Test Results

An additional step compresses all test results into test_results_package.zip for easier sharing, release integration, and archiving. This provides a centralized, portable bundle of all test outcomes.

6. Create GitHub Release Tag

This step automates the creation of a Git tag for the release. It configures Git with a bot user and then creates an annotated tag (`v${{ github.run_number }}`) with a descriptive message. This tag is then pushed to the origin, marking a specific point in the repository's history as a release version. This is crucial for version control and tracking deployed versions.

7. Deploy Application

This step simulates the actual deployment of the application to a target environment. In this configuration, it copies the contents of the deployment directory to a temporary production-like directory (/tmp/tic-tac-toe-production). While this is a simplified example, in a real-world scenario, this step would involve deploying to a server, cloud platform, or embedded device. The message "Application deployed successfully" indicates the completion of this phase.

8. Verify Deployment

After deployment, this step performs a basic verification to ensure the application is correctly deployed and runnable. It checks for the presence of the Embedded executable in the deployment directory and attempts to run it with the --version flag. A successful execution confirms that the deployed application is functional. If the executable is not found or fails to run, the deployment is marked as failed, preventing further issues.

9. Upload Release Artifacts

This step uploads the newly created release artifacts, specifically the .zip archive, the deployment directory, and the test_results_package.zip as artifacts of the GitHub Actions run. This makes the release package easily accessible from the GitHub Actions interface for manual downloads or further automated processes.

10. Upload to GitHub Release

This final step leverages the softprops/action-gh-release@v1 action to create a formal GitHub Release. It uses the previously created tag, sets the release name, and provides a body message. Crucially, it attaches the .zip archive, the contents of the deployment directory, and the zipped test results package to the GitHub Release. This provides a user-friendly interface for accessing release assets and release notes directly from the GitHub repository, completing the CI/CD process with a well-documented and accessible release.

Main Functions of the Code

The provided `ci.yml` file defines a robust CI/CD pipeline with several key functions:

- **Automated Build and Test:** The pipeline builds the C++ application using Qt 6 and runs both unit and integration tests in headless mode using xvfb-run. Test outputs are stored in XML format, enabling structured test result reporting and easier integration with future dashboards.

- **Code Quality and Security Assurance:** Integration of `cppcheck` for static analysis and a basic `grep` -based security scan helps identify potential issues early, promoting higher code quality and reducing the risk of security problems.

- **Performance Benchmarking:** The inclusion of performance measurement provides valuable insights into the application's resource consumption, which is critical for optimizing embedded systems.

- **Artifact Management:** The pipeline carefully gathers and uploads various artifacts, including the compiled executable, deployment packages, test results, and reports. This ensures traceability, facilitates debugging, and provides all necessary components for deployment.

- **Structured Test Packaging**: All test results are bundled into a zipped package (test_results_package.zip) for release inclusion and external analysis.

- **Automated Deployment and Release Management:** For changes pushed to the `main` branch, the pipeline automates the deployment process, including the creation of a release archive, Git tags, and a formal GitHub Release with attached assets. This streamlines the release cycle and ensures consistent deployments.

- **Branch-Agnostic Validation:** The CI/CD pipeline triggers on all branches for building and testing, while deployments are restricted to the main branch for production safety. This approach ensures that all code changes are validated early during development, improving code quality and preventing bugs from reaching the main branch. At the same time, restricting deployment to the main branch helps prevent unstable or incomplete features from being deployed, ensuring that only reviewed and approved changes are released to users.

- **Environment Consistency:** By defining all dependencies and build instructions in the pipeline itself, the workflow ensures a consistent environment across all builds — reducing errors caused by local setup differences.

- **Version Tracking and Auditability**: Each release is tagged with a unique version and commit hash. This improves traceability and helps identify exactly what code was included in each deployment.

Conclusion

The CI/CD pipeline defined in ci.yml provides a reliable and automated workflow for building, testing, and deploying the Advanced Tic Tac Toe game. By integrating automated compilation, quality checks, unit and integration testing, performance analysis, and release packaging, the pipeline enhances development efficiency and reduces the risk of errors.

This approach ensures that every code change is consistently validated in a controlled environment, while only stable code from the main branch is deployed. As a result, the team can focus more on improving the game's features and logic, with confidence that the system is continuously verified, versioned, and delivered in a stable and repeatable way.