PJ - Connected Products Azure Pipeline Documentation

Docupedia Export

Author:Venkatesan Kannan (MS/EDB7-XC)
Date:07-Jan-2025 06:35

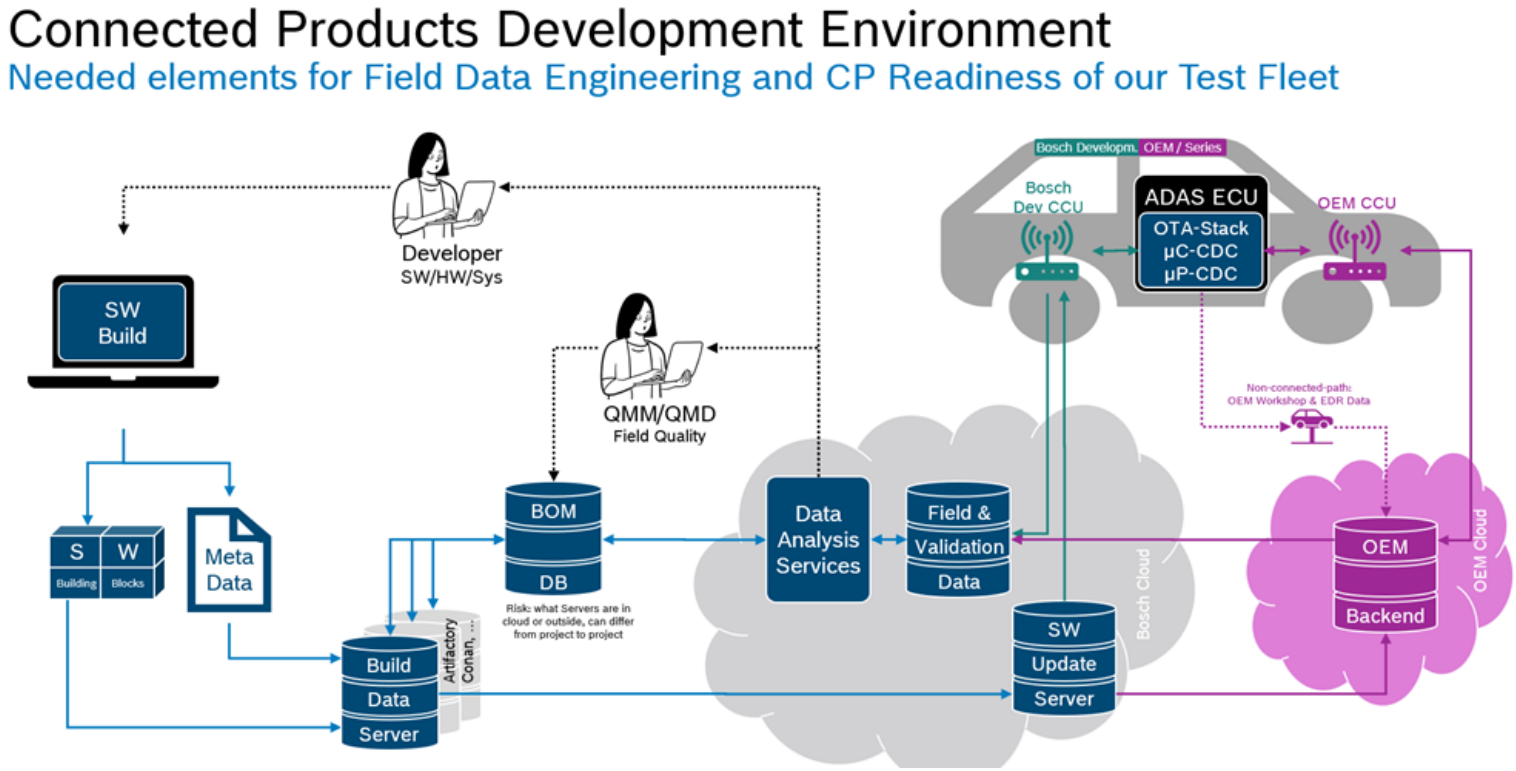
Table of Contents

PJ - Connected Products overview of pipeline structure

PJ - Connected Products:
The goal of our project **Connected** Products is to connect our products to (HW, SW, and Functions).
→ In the field → together with the OEMs
→ During the Development Phase.

Connecting refers to
→ **Getting Data** Over the air from our products for the improvement of quality **(Predictive diagnosis**) & **KPIs**
→ Understanding the function behavior and learning about the interaction with its Environment (incl. User)
→ Gathering inputs for **data-driven development**.
→ Offering the capability for **shadow mode & and shadow Validation**.

Enhancing and maintaining our products in the field by **updates over the air (aka FOTA)**
→ Directly to the user and customers In a safe and secure manner



Overview of Pipeline Structure

The primary objective of our Azure Pipeline is to automate the creation of Conan packages, facilitating a streamlined process for building, packaging, and pushing artifacts to our Artifactory repository. This automated pipeline ensures efficient and consistent package generation, ultimately providing a reliable and centralized source for dependencies. The produced Conan packages are designed to be easily consumed by various projects within our ecosystem, fostering seamless integration and collaboration.

- Compile and build software for different variants (e.g., x86_gcc_debug, fdc_armv7_r5f_vfpv3_d16_release).
- Generate Conan packages for these variants.
- Perform unit testing, code analysis, and documentation generation.
- Push the generated Conan packages to the Artifactory repository.
- Facilitate the consumption of Conan packages by other projects within the XC-AS Software Factory.

Key Components:

The pipeline is comprised of the following key components:

Source Code Repository:
The source code repository holds the Conan package configurations and relevant source code.
Repo URL: https://github.boschdevcloud.com/PJ-CP/prj.field_data_collection.git

Build Stages:

Compiles source code for different variants - This stage involves taking the source code of your software project and translating it into machine-readable code. **The term "variants" suggests that our pipeline can handle different configurations or flavors of the software, each tailored for specific purposes or environments.**
Functionality: This stage ensures that the source code is transformed into executable binaries suitable for the specified variants **(e.g., x86_gcc_debug, fdc_armv7_r5f_vfpv3_d16_release).**

Dependencies: Relies on source code configurations, build presets, and any external dependencies needed for successful compilation.

Generates Conan packages - After compiling the source code, **this stage involves creating Conan packages. Conan is a package manager that helps in managing and distributing dependencies of your software project.** A Conan package contains all the necessary files and information required to use a specific version of a library or tool.

Functionality: This step ensures that the dependencies of your software are packaged in a standardized way, making it easy for other projects to consume them.

Dependencies: Depends on the compiled artifacts from the previous build stage.

Unit Testing and Code Analysis:

Conducts unit tests and static code analysis - This stage involves running unit tests **to verify that individual units of code function correctly. It also includes static code analysis, which is the process of analyzing the source code without executing it, aiming to find potential issues or improvements.**

Functionality: Ensures the reliability and **quality of the code** by identifying bugs, potential security vulnerabilities, or **areas for improvement.**

Dependencies: Depends on the compiled artifacts from the Build Stages.

Dependencies Tools:

Utilizes tools like QAC, module checker, and compiler warnings - Various tools, such as **QAC** (a **static code analyzer**) and **module checker**, are employed to analyze the codebase further. Compiler warnings are messages generated by the compiler indicating potential issues in the code.
Functionality: These tools provide **additional layers of code analysis to catch issues** that might not be captured by unit tests.

Dependencies: Depends on the compiled artifacts from the Build Stages.

Artifactory Repository:

Serves as the central repository for the produced Conan packages - This repository is a **centralized storage location for all the Conan packages produced during the pipeline execution**. It acts as a hub where other projects can access and retrieve the required dependencies.
Functionality: Provides a single source of truth for Conan packages, making it easy for projects within the XC-DX Software Factory **to consume and share dependencies**.
Dependencies: Receives Conan packages from the Deployment Stage.

Deployment Stage:

Pushes Conan packages to the Artifactory repository - In this stage, the Conan packages generated **earlier are pushed or deployed to the Artifactory repository**. This step makes the packages available for other projects to use.
Functionality: Enables the distribution of the packaged dependencies to the central repository.
Dependencies: Depends on the successfully generated Conan packages.

Conan Build Stages:

Executes Conan builds for specified variants - Conan builds are executed for **specific variants (e.g., x86_gcc_release, r5f_ghsarm).** This stage ensures that the Conan packages are correctly configured and built according to the specified requirements.
Functionality: Manages the Conan configurations and builds Conan packages for different environments or platforms.
Dependencies: Relies on Conan configurations and dependencies specified in the Conan packages.
Manages Conan configurations and Conan package uploads.

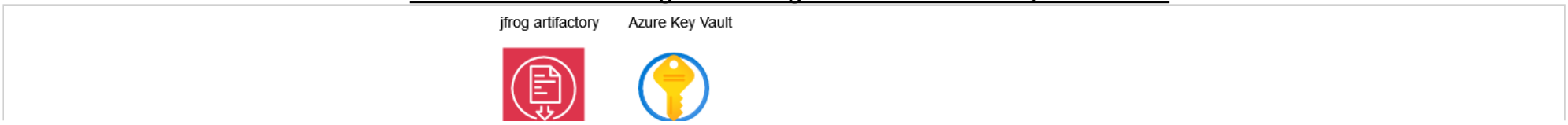
Containerization:

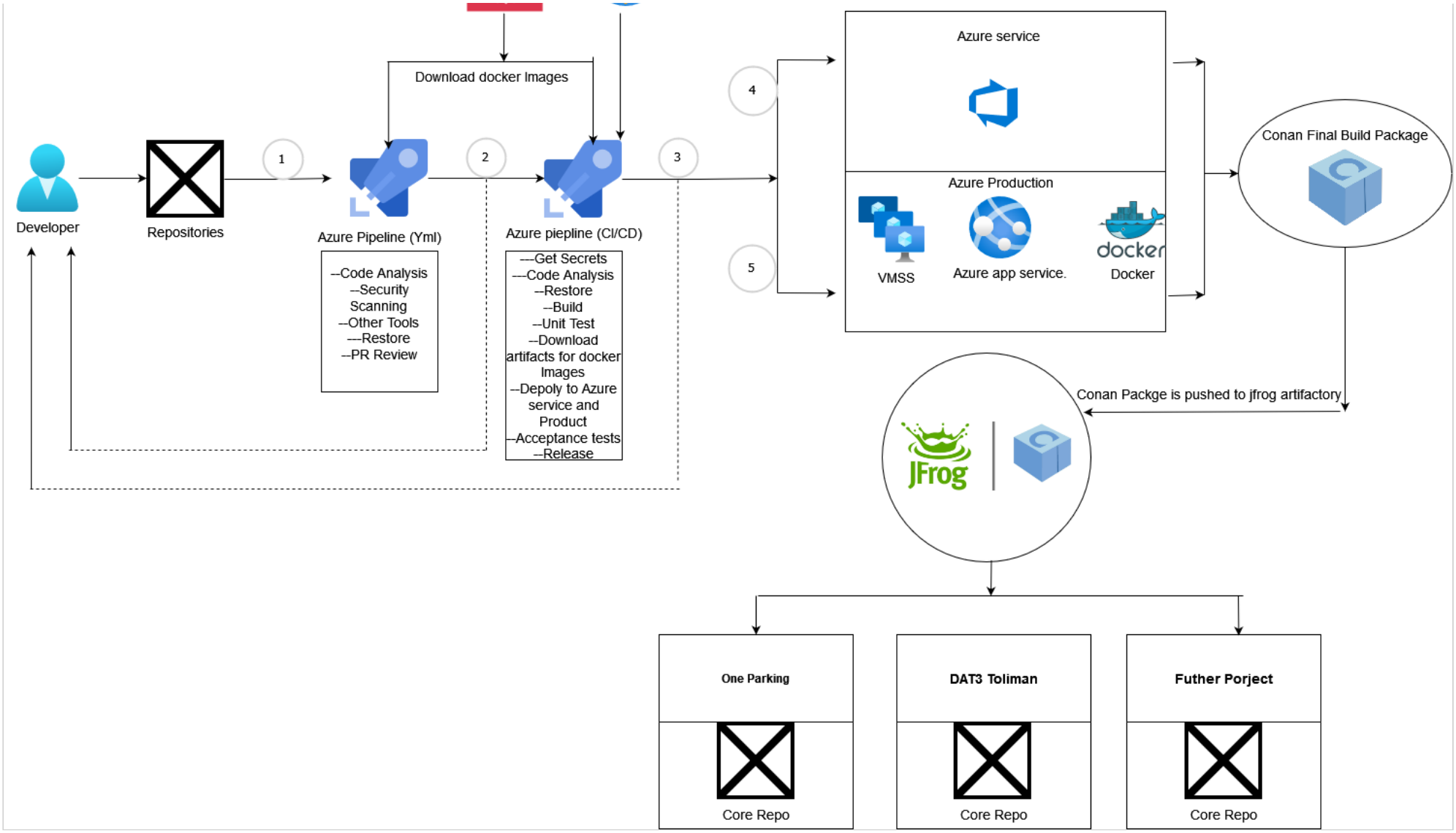
Utilizes Docker containers for consistent and reproducible builds - Docker containers provide a standardized and isolated environment for the entire build process. This ensures that builds are consistent across different development machines and environments.

Functionality: Enhances reproducibility by encapsulating all the necessary dependencies and tools within a Docker container.

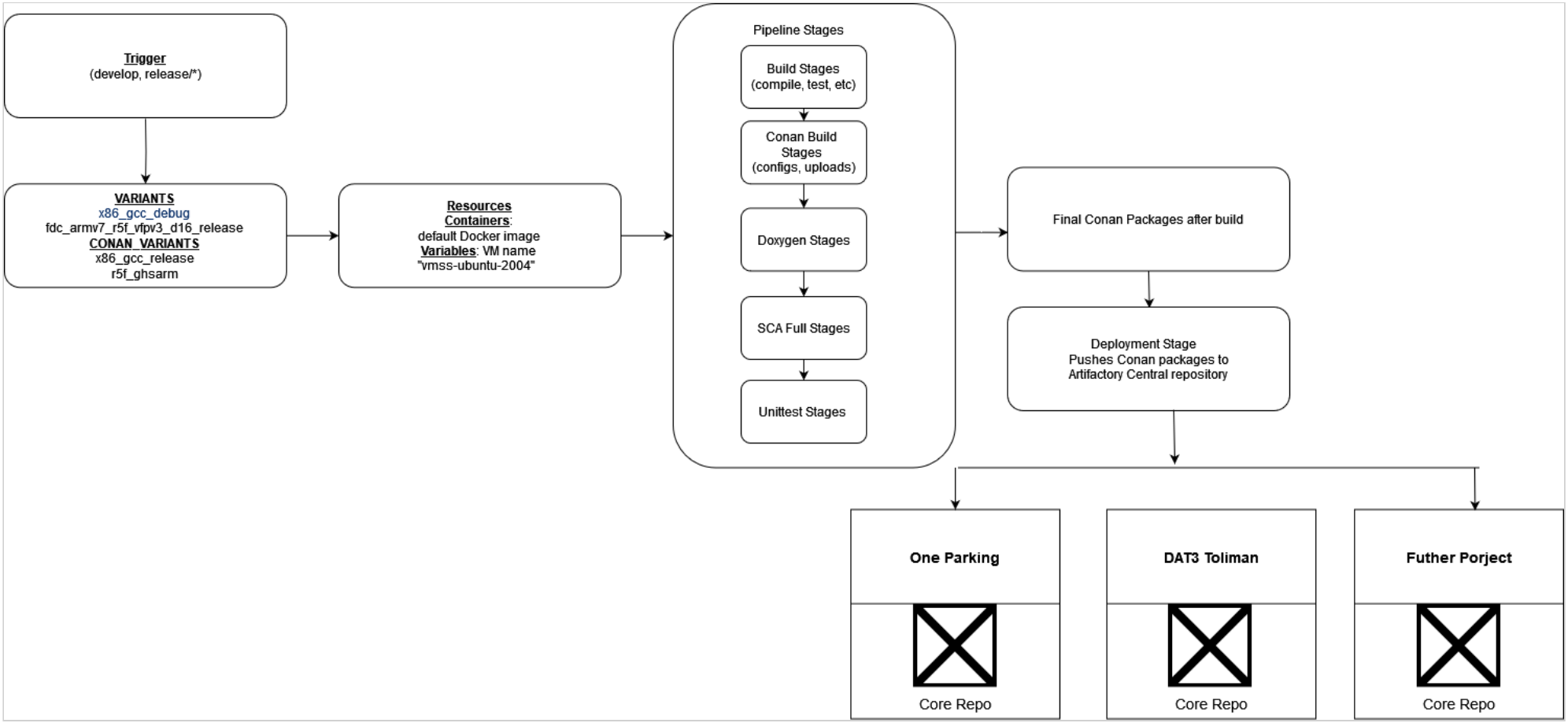
Dependencies: Relies on the Docker image specified in the pipeline configuration.

Please find below the diagram showing an overview of the Project workflow.





Please find the below diagram that shows how DATA flows in the Project.



Each block represents a stage or component in our PJ-CP pipeline, and arrows indicate the flow between them.

Build Stages (e.g., x86_gcc_debug, fdc_armv7_r5f_vfpv3_d16_release):

Tools and Technologies:

Compilation: GCC (GNU Compiler Collection), GHS (Green Hills Software Compiler).
Static Code Analysis: QAC.
Module Checker: (disabled).
Unit Testing: (enabled).
Documentation: Doxygen.
Code Coverage: (disabled).
Splunk Integration: (disabled).
Programming Languages: Source Code: Could include C, C++.
Build Configuration: Scripts or configuration files for build setup.

Conan Build Stages (e.g., x86_gcc_release, r5f_ghsarm, r5f_ghsarm_with_interface):

Tools and Technologies:
Package Manager: Conan Package Manager.
Programming Languages: Conan configurations may involve specifying dependencies and settings for C/C++ packages.

Containerization:

Tools and Technologies:
Containers: Docker Containers.
Programming Languages: Not applicable (containers encapsulate the environment).

Artifactory Repository:

Tools and Technologies:
Artifactory (Docker Registry) – Jfrog Artifactory.
Programming Languages: Not applicable (repository management).

General Environment Setup:

Tools and Technologies:
Programming Languages: Custom Environment Setup Scripts **(setup_environment.yml).**

Additional Configurations and Variables:

Tools and Technologies:
Environment: Azure DevOps Variables, Parameterized Configurations.
Programming Languages: Not applicable (configuration management).

Pipeline Execution Environment (e.g., VM Image):

Tools and Technologies:
VMSS: (Virtual Machine Scale Sets), Ubuntu 20.04.
Programming Languages: Not applicable (infrastructure configuration).

In summary, our pipeline involves a **mix of C/C++ programming languages** and utilizes various **tools such as GCC, GHS, Conan, Docker, and Artifactory to compile, test, package,** and deploy our software components. The configuration files and scripts play a crucial role in setting up the environment and managing the pipeline execution flow.

Pipeline Execution:

Triggering Mechanism: From Pull Request –(PR)

As per the Azure Pipeline code:

Trigger: none: This line means that the pipeline is not triggered by the typical code push (none means no automatic triggering).
PR: This section specifies the pull request (PR) triggering configuration.
Branches: It further specifies the branches for which pull requests trigger the pipeline.
Include: Specifies the branches to include for triggering.
Develop: This means that pull requests targeting the develop branch will trigger the pipeline.
Release/*: This is a wildcard pattern, indicating that pull requests targeting any branch starting with "release/" will trigger the pipeline.
So, in summary, our pipeline is triggered when pull requests are opened or updated for the develop branch and any branch starting with "release/". This is a common setup for pipelines that need to validate changes before merging them into specific branches.

Here are the key points discussed:

Pipeline Overview: The pipeline is triggered on specific branches (develop, release/*) and utilizes a structured YAML configuration to define stages, jobs, and tasks.
Build Stages: Stages include compiling source code, generating Conan packages, conducting unit tests and static code analysis, and deploying packages to the Artifactory repository.

Conan Build: The pipeline effectively manages Conan builds for different variants, ensuring consistent and reproducible builds.

Containerization: Docker containers are employed to achieve consistent and reproducible builds, promoting portability across different environments.

Technologies Used: Various tools and technologies, such as GCC, GHS, and Docker containers, are utilized in different stages of the pipeline. Programming languages like C and C++ are predominant in the source code.

Pipeline Execution: The pipeline is triggered by specific events, including pull requests targeting the develop branch and release branches. Monitoring and logging are integral parts of the pipeline, providing insights into the health of the execution.

IDM Roles:

IDM2BCD_BDC_Artifactory_01_perm566_user: Artifactory 01 - Permission Target Application_coordinator_423 - Delete/Overwrite permission.

IDM2BCD_BDC_Github_01_org397_member: GitHub Enterprise Org PJConnectedProducts - Member.

IDM2BCD_BDC_Github_01_org397_owner: GitHub Enterprise Org PJConnectedProducts - Owner.

BoschDevCloud_user: This role enables the user to see the Bosch Development Cloud in the applications overview and order roles via IdM user self-service. It is limited to all internal and external users with a Bosch AD user account.

URS Groups:

rb_urs_pjcp_contributer: URS group for Developers in the PJCP domain.

rb_urs_pjcp_devopsadmin: URS group for DevOps administrators in the PJCP domain.

rb_urs_pjcp_systemuser: URS group for system users in the PJCP domain.

GitHub Organization: PJ-Connected Products:

Member role: **IDM2BCD_BDC_Github_01_org397_member**.

Owner role: **IDM2BCD_BDC_Github_01_org397_owner.**

Cloud Access:

BDC_Cloud521_admin: Access group for **Cloud space 00062 admin rights.**

Azure subscription access: To access the Azure infra.

Notes:

Ensure that the user is not directly added to any resource; the system user is a member of **rb_urs_pjcp_systemuser**. The URS Group is added via Bring Your Own Groups (also in GitHub)

If you are working on the FDC-Core as a developer please request the following Contributor role: **rb_urs_pjcp_contributer**

If you are working as a Dev-Ops Admin/Batman/etc., please request the following role: **rb_urs_pjcp_devopsadmin**

License information:

We need to request license access for your project: Please reach out to **License-Service-Inquiry@de.bosch.com & SOFA Team**

1) 5065@rb-lic-prqa-cloud.bosch.tech
2) 2009@rb-lic-ghs-cloud.bosch.tech
3) 8224@rb-lic-armlmd-xc-cloud.bosch.tech
4) 3059@rb-lic-covlicd-cloud.bosch.tech
5) GCC License
6) Coverity License we need to raise the URS group: **CCLICENSE_1712_227_ua** is the license group for **Access for CC-AD Coverity user licenses.**