An IBM Document from

IBM Z DevOps Acceleration Program

September 2022 - Release 1.0

Implementing a Release-based Development Process for Mainframe Applications

Dennis Behm
dennis.behm@de.ibm.com

Mathieu Dalbin
mathieu.dalbin@fr.ibm.com

Lauren Li
lauren.k.li@ibm.com

Abstract

This document provides guidance on implementing a release-based delivery workflow using Git, IBM Dependency Based Build, and a deployment solution

Table of contents

1 Introduction

This document describes a blueprint implementation of a release-based development process for mainframe applications. This setup leverages a standardized development toolset based on an enterprise-wide Git provider and continuous integration/continuous deployment (CI/CD) toolchain.

The purpose of streamlining both the DevOps solutions and the delivery workflow is to simplify the process for development teams to deliver quality product releases on time. This enables Agile development practices that allow the teams to respond more effectively to changes in the market and customer needs.

The first part of the document will cover a high-level overview of the release-based approach, and details of the technical implementation will be discussed in the second part. It is important to note that the branching model discussed in this document is intended to serve as a guide rather than a set of rules. Because development teams and applications vary in size, complexity, and requirements, the approach can be adjusted as needed depending on the application and team.

2 Overview of a simplified release-based development process

2.1 Introduction

As Git turned into the de-facto version control system in today's IT world, new terminologies such as *repositories*, *branches*, and *merges* arose. By agreeing upon a central Git server to integrate and consolidate changes, development teams were able to collaborate more efficiently and effectively. Building upon the open-source vanilla Git implementation, popular Git providers including GitHub, GitLab, and Bitbucket have implemented additional workflow features to facilitate a secure and stable development process. These include features such as *Pull Requests* (sometimes referred to as *Merge Requests*) to support coordination with Git in larger teams. The term *Pull Request* will be used throughout this document to designate the operation of merging one branch into another.

In 2010, Vincent Driessen blogged about his experience in software development projects. His thought process and experiences leveraging Git in large development projects led to the documentation of a Git branching model called *git-flow*[1], which became widely adopted throughout the development community.

In many organizations, mainframe development teams typically follow a release-based process to deliver incremental updates to a pre-defined production runtime. When adopting Git as a source control management (SCM) solution for doing mainframe development, *git-flow* is a perfect foundation for designing a safe development process.

2.2 Mainframe development specificities in a git-flow approach

In the traditional development lifecycle of mainframe applications, only modifications, either what has changed or what is impacted by a change, are built, released, and deployed. The process avoids the time-consuming, full build and deployment of the entire application.

This raises the requirement to have one configuration (branch) in the version control system representing the current source code running in production. The main branch of the Git repository will typically represent that reference. Representing the production state of the application with a branch can be considered as a variation of the original git-flow concept.

[1] https://nvie.com/posts/a-successful-git-branching-model/

2.3 High-level workflow of the release-based approach

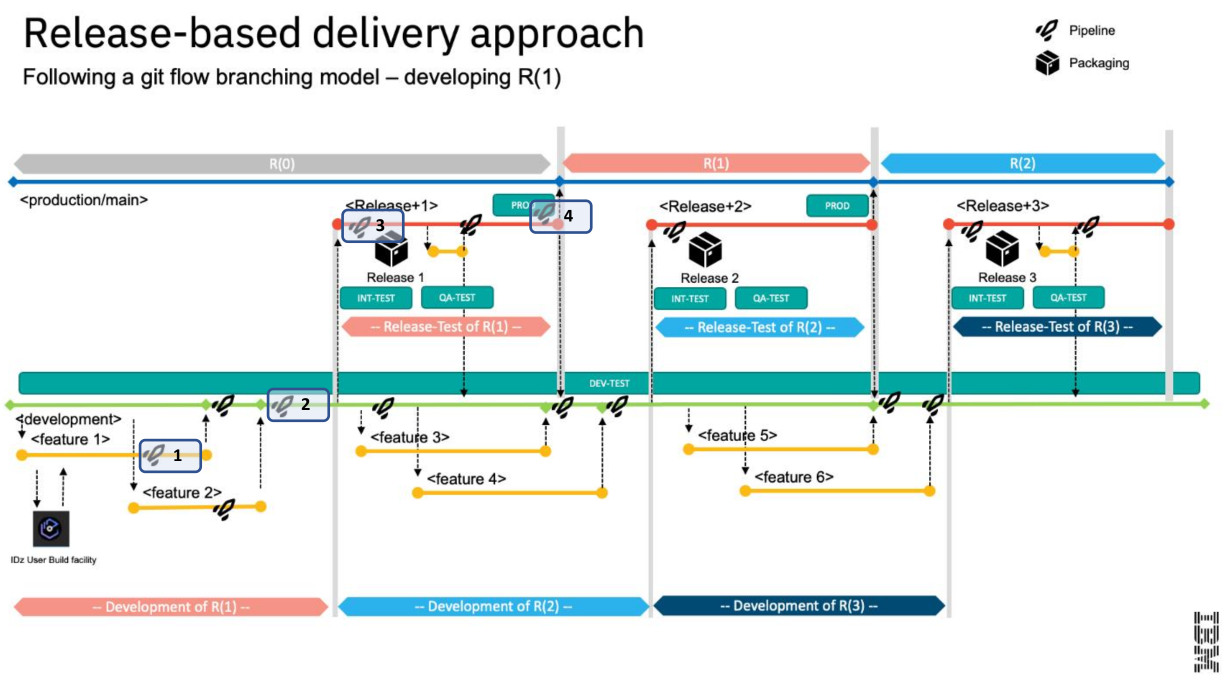

Figure 1 - Git-flow supporting a Release-based delivery approach

The above diagram depicts a simple release-based delivery process. This methodology allows developers to work on two or more concurrent releases, with the option to deliver hotfixes to the current production runtime.

To support this workflow, a distinction must be made between long-living branches such as the *Development* and the *Production/Main* branches, and short-living feature branches (also known as "topic branches"). Developers implement their changes on short-living feature branches and integrate those via Pull Requests into the long-living branches, which are configured to be protected branches. *Release* branches are assumed to be protected as well.

At a high level, the development team works through the following tasks:

1. Developers accept new work items and plan them according to a loosely defined release schedule. The release-based workflow provides some flexibility to developers, as it is not required to work against a formal release schedule.
2. For each independent development task for the current release (release *R(1)* in this case), a feature branch is created according to pre-defined naming conventions, allowing the assigned developers to have a copy of the codebase on which they can work in isolation from other concurrent development activities.
3. The developers typically fetch the feature branch from the central Git repository (often called *origin*) into their local clone of the repository and start making the necessary modifications for their development task. They use the Dependency Based Build (DBB) *User Build* facility of their integrated development environment (IDE) to validate the changes before committing them and pushing them back to their feature branch on the central Git repository.
4. 1 Optionally, the CI/CD pipeline orchestrator runs a pipeline for the feature branch on the central Git server each time the developers push their committed changes to it. This process will start a consolidated build that can also include the build of impacted programs within the application scope. Unit tests can be automated for this pipeline, as well.

5. Once the developers feel their code changes are ready to be integrated back into the shared Development branch, they create a Pull Request to integrate the changes from their feature branch into the shared Development branch. The Pull Request process often provides the capability to add peer review and approval steps before allowing the changes to be merged.
6. 2 Once the Pull Request is merged into the Development branch, the next pipeline execution for the Development branch will build all the changes since the last successful build on the Development branch. This pipeline can include a step to deploy the built artifacts (load modules, DBRMs, etc.) into a shared test environment, as highlighted by the green *DEV-TEST* icon in the diagram. In this *DEV-TEST* environment, the development team can validate their combined changes. This first test environment helps support a shift-left testing strategy by providing a sandbox with the necessary setup and materials for developers to test their changes early.
7. When the development team agrees to progress further in the delivery process of R(1), they enter the release phase of the workflow by creating a Release branch, which is based off the Development branch. This can be an automated action or performed manually. The purpose of the release phase is to validate the changes in the next test environment.
8. Along with the creation of the Release branch, the team creates a Pull Request from the Release branch to the Production/Main branch. This creates a place for collaboration between team members and a vehicle to track and propagate the changes with transparency.
9. 3 The first execution of the pipeline for the Release branch will rebuild the contributed changes for the release, with the optimized compile options. Compared to the previously described pipeline on the Development branch, this pipeline includes an additional step to package build outputs and create a release candidate package. The package includes all the identified changes since the last release and is passed to the deployment solution. Related metadata is also propagated to enable traceability across the version control system, pipeline orchestration, and deployment solution.
10. The release package is promoted through the various test stages and can take a predefined route. The process can be controlled either through the pipeline orchestrator itself, or the development team can leverage the interfaces of the deployment solution.
11. At this point in time, the development team can already start working on the following release *R(2)*, based on the Development branch. This approach allows work on two (or more) active deliveries at the same time.
12. In the event of a defect being found in the new code of the candidate package for release R(1), the developer will create a feature branch from the Release branch and fix the issue. It is expected that the new package with the fix is required to pass all the quality gates and to be tested again.
13. 4 When the release is ready to be shipped, the development team proceeds with the finalization of the release. They must ensure that during the development process of release R(1), no code changes (such as hotfixes) were delivered into the Production/Main branch, or if delivered, the development team must ensure that the code changes have been integrated into the Release branch (and Development branch), as well. All quality gates should have passed successfully and approvals issued by the appropriate reviewers. When the deployment solution installs the release package into the production environment, the content of the Release branch is merged into the Production/Main branch and the associated Pull Request is closed accordingly. All other long-living branches, such as the Development branch and any other active Release branches, should be subsequently updated with the changes of the delivered release.

3 Guidance for designing and implementing a release-based development pipeline

In this section, the technical implementation of the release-based methodology will be detailed, focusing on the major steps of the workflow.

3.1 Implement changes on the Feature branch

When the developers start working on a new task, they will first create a feature branch based on the Development branch to start working in isolation. If the feature branch was created on the central Git repository, the developers can use a terminal or another Git interface on their local workstation to clone or fetch the new feature branch from the central Git repository down to their local working tree in which the changes will be implemented.

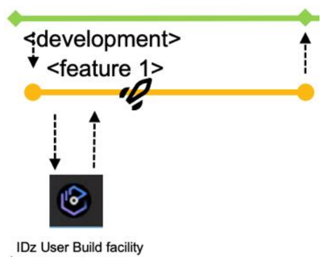

Figure 2 - Feature branching workflow

Integrated development environments (IDEs) supported by IBM allow developers to perform a DBB User Build to quickly gather feedback on the implemented changes. This feature is expected to be used before the changes are committed and pushed to the central Git server, where a pipeline can process changes automatically.

3.1.1 User Build setup

User Build is a feature provided by IBM-supported IDEs that compiles and link-edits modified source code located in the local, checked-out Git working tree on the developer's workstation. This capability is available in the following IBM IDEs:

- IBM Developer for z/OS[2]
- VS Code with the IBM Z Open Editor extension[3]
- IBM Wazi for Dev Spaces[4]

The developer configures the User Build process to point to the central build framework implementation, such as zAppBuild[5], provided by the Mainframe DevOps Team. The build option *--userBuild* is passed to the build framework.

Because the operation is performed with the credentials of the currently logged-in user, it is recommended for each developer to reuse the high-level qualifier of their personal datasets. It is the developer's responsibility to regularly clean up the mainframe datasets and sandbox directories on USS that are used for User Build. Automated cleanup of the files can be established based on a defined naming convention for datasets or with a specific storage management policy.

[2] IBM Developer for z/OS - https://www.ibm.com/docs/en/developer-for-zos
[3] IBM Z Open Editor - https://www.ibm.com/docs/en/cloud-paks/z-modernization-stack/2022.2?topic=ide-option-2-developing-vs-code
[4] IBM Wazi for Dev Spaces - https://www.ibm.com/docs/en/cloud-paks/z-modernization-stack/2022.2?topic=ide-option-1-developing-codeready-workspaces
[5] https://github.com/IBM/dbb-zappbuild

User Build is a convenient way to compile and link-edit source code without committing the changes into the Git version control system. Therefore, build outputs of User Builds are not assumed to be installed into a runtime environment. To be able to perform simple and rudimentary tests on User Build-generated outputs, the developer should modify the test JCLs to point to the personal libraries used in User Builds.

Alternatively, the setup of a pre-concatenated runtime library can be implemented to perform more tests in the context of a (shared) test runtime environment. A dedicated pre-concatenated library in the runtime system (for example, batch, IMS and CICS) into which the developers can write allows a separation of the modules produced by User Builds, and enables regular cleanup of these intermediate versions that are not yet registered in the central Git provider.

External dependencies to other components, such as include files (for example, copybooks or object decks) which are not managed within the application repository, but are required for building the application, can either be pulled in via a dataset concatenation[6] or by the usage of Git submodules, depending on the repository organization.

Developers can commit and push their changes to their feature branch on the central Git provider at any time.

3.1.2 Pipeline build for Feature branches

Using feature branches is a common practice to isolate development activities and focus on a particular feature, fix, or enhancement. Feature branches relate back to the change request (or issue) from the planning phase and their name should be based on a meaningful convention that reflects this. Typically, one or several developers operate on the feature branch to contribute with the necessary changes on the code base.

Optionally, a feature branch pipeline can be configured by the DevOps team to build the codebase of the application and perform code reviews. The purpose of this pipeline is also to expand the scope of the build and include changed and impacted programs to the list of artifacts to be produced. It operates on the source code of the feature branch stored in the central Git provider.

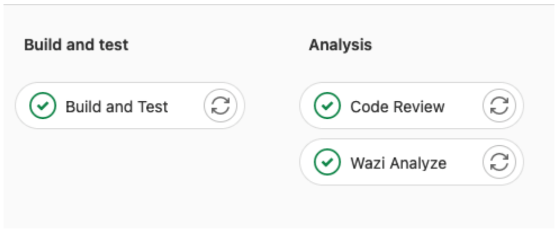

Figure 3 - Pipeline steps for the feature branch

The pipeline leverages the dependency metadata managed by IBM Dependency Based Build (DBB) via DBB collections, which are consumed by the build framework, zAppBuild. At the first execution of the build process for the feature branches, zAppBuild will duplicate this metadata by cloning the related collections for efficiency purposes. This cloning phase ensures the accuracy of the dependency information for this pipeline build. To be able to clone the collection, zAppBuild needs to understand which collection contains the most accurate information and must be duplicated. As collection names are derived from the name of the branch, it is easy to identify which collection should be cloned.

[6] zAppBuild Configuration - https://github.com/IBM/dbb-zappbuild/blob/main/samples/application-conf/Cobol.properties#L91-L97

In the zAppBuild configuration, the originating collection is defined via the *mainBuildBranch* property.[7] Depending on the source branch a given feature branch was created from, the *mainBuildBranch* property may be dynamically passed to the build framework.

The pipeline for the feature branch performs the build step, which typically uses a unique high-level qualifier that can be derived from the feature branch name. However, it is not required to package the generated artifacts and deploy them, unless you want to install them in target environments for testing purposes. Similar to the User Build scenario, the deployment phase can leverage the pre-concatenation approach, as discussed earlier.

A house-keeping strategy should be implemented when the feature branch is no longer needed and therefore removed from the central Git provider. This includes the clean-up of the DBB collections as well as the build workspace. Specific scripts can be integrated into the pipeline to delete collections and build groups[8], or unnecessary build datasets[9]. When leveraging GitLab CI/CD as the pipeline orchestrator, the use of GitLab environments helps to automate these steps when a branch is deleted. An implementation sample is provided via the published technical document *Integrating IBM z/OS Platform in CICD Pipelines with Gitlab*[10]. Generally, webhooks and other extensions of the pipeline orchestrator can be leveraged to perform these clean-up activities when a branch is being deleted.

3.1.3 Integrating features into the Development branch

When developers are finished with the implementation of their changes in their feature branch, the next step is to integrate their contribution into the shared codebase for the given stage of the development cycle. This process is usually performed with the help of the Pull Request mechanism of the central Git provider.

In the simplified model of the release-based approach, creating a Pull Request is a convenient way to declare the changes to be delivered with this iteration. However, if developers need to hold their changes off until a later iteration, they still have the capability to create a Pull Request without going through the process of merging their branch into the shared Development branch.

The Development branch is assumed to be a protected branch, meaning that no developer can directly push changes to this configuration. It requires and forces developers to go through the Pull Request process. Before merging the feature branch into the shared Development branch, some evidence should be gathered to ensure quality and respect of the coding standards in the enterprise. Peer-reviewed code, a clean pipeline execution, and approvals are examples of such evidence, allowing the development team to confidently merge the feature branch into the Development branch.

A common practice is to squash the different commits created on the feature branch into a single new commit, which keeps the Git history from becoming cluttered with intermediate work for the feature. This also helps to maintain a clean history on the Development branch. You can read more about squash merges on the documentation sites of the different Git providers.

[7] IBM DBB zAppBuild Sample Build Framework Implementation – Impact Builds for topic branches https://github.com/IBM/dbb-zappbuild/blob/main/BUILD.md#perform-impact-build-for-topic-branches
[8] IBM DBB Community Repository - https://github.com/IBM/dbb/tree/main/Utilities/WebAppCleanUp
[9] IBM DBB Community Repository - https://github.com/IBM/dbb/tree/main/Utilities/DeletePDS
[10] IBM Whitepaper - https://www.ibm.com/support/pages/integrating-ibm-zos-platform-cicd-pipelines-gitlab

3.2 Pipeline steps for the Development branch

After the feature branch is merged into the Development branch, a new pipeline is kicked off to build the integrated changes in the context of the Development branch configuration, as shown in Figure 5 (1).

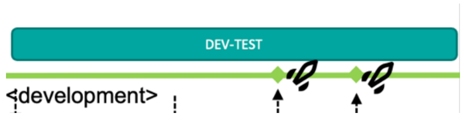

Figure 4 - Automated pipelines after merging feature into the shared codebase

Additional steps such as automated code reviews or updates of application discovery repositories can be included in this pipeline process. When the development team moves from the development phase into the release phase, they start by creating a Release branch from the current state of the Development branch (Figure 5, 2).

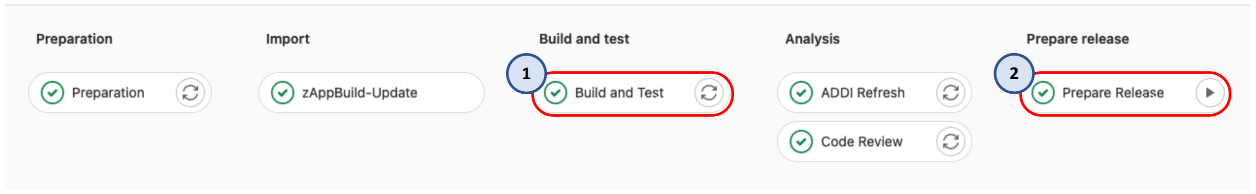

Figure 5 - Pipeline steps for the Development branch

3.2.1 Development pipeline: Build and Test

The purpose of the Build step of the pipeline for the Development branch is to ensure that multiple merged features can be built and then tested together. It might happen that some features have indirect dependencies on other features planned for the same release. This early point of integration along with the impact build capability of the build framework[11] ensures consistency and transparency of the upcoming release.

Decoupling the build step from the deploy step in the pipeline is important to ensure that only outputs from successful builds are installed into the test environment, rather than directing the build framework to update the libraries of the test environment directly.

In this phase of the development lifecycle, the build typically operates with the compile options to enable testing and debugging of the programs. As most organizations restrict the deployment to the production environments with optimized code only, these build artifacts can be seen as temporary and only for initial testing and debugging purposes.

To deploy the generated artifacts to the shared development test system (represented by the *DEV-TEST* icon in Figure 4), a post-build script in a subsequent step is leveraged to process the build report documenting the generated build outputs. It is typically the role of the deployment solution to create a package and deploy it into different environments (this step will be discussed in detail in the description of the pipeline for the release). However, given the temporary nature of the build outputs created for the Development branch, a valid strategy would be to implement an ad-hoc method to install those temporary deliverables to a *DEV-TEST* environment and circumvent the formal packaging and deployment process. The major drawback of this approach is a lack of traceability and understanding of what runs on the *DEV-TEST* environment.

To implement this simplified process, a post-build script can be used to copy the output artifacts from the build libraries directly to the associated target runtime libraries.

[11] See zAppBuild --impactBuild - https://github.com/IBM/dbb-zappbuild/blob/main/BUILD.md#perform-impact-build

3.2.2 Development pipeline: Automated creation of the Release branch

When the development team has completed their tests on the *DEV-TEST* environment and agreed to enter the release phase, the development team creates a new Release branch from the current state of the Development branch. This Release branch would now include all the contributing changes for this release.

The creation of the Release branch can be easily automated to ensure consistent naming conventions across projects. This creation step is typically the last action of the pipeline for the Development branch, and is generally a manually-triggered step for more control. This also allows the development team to specify and pass a name for the release in accordance with their organization's standard conventions. One option is to allow the team to specify the type of release: major, minor or patch, rather than passing in the name. By automating this step, standards are correctly and consistently applied: based on the release type the development team has chosen, the pipeline step automatically computes the name of the Release branch following a semantic versioning model[12]. For instance, releases can follow a naming convention like *rel-x.x.x*, where *x.x.x* represents the semantic versioning of the release. Using the *rel* prefix also helps to easily identify releases in the pipeline processing when adding steps into the workflow. Of course, different naming conventions can be applied.

To automate the creation of the new Release branch, the DevOps engineer can leverage the REST APIs offered by the different Git providers. REST APIs can be reached with the *curl* utility, which is available on most platforms, including z/OS Unix Systems Services (USS).

For reference, GitHub documents its REST interface to create a new reference[13], while GitLab's documentation to create new branches can be found under the Branches documentation[14].

After the Release branch is created, a Pull Request is created to document the changes for this release, which are now flowing towards production. The creation of the Pull Requests can be automated with both GitHub[15] and GitLab[16]. The Pull Request process provides the capability to add peer reviews and approvals before allowing the changes to be merged.

3.3 Pipeline steps for the Release branch

The pipeline for the Release branch includes additional steps and differs from the previously discussed pipelines.

The goal of this pipeline, outlined in Figure 7, is to build a release candidate with all the contributed changes 1 , and to create the release package of the generated build outputs 2 , which is then deployed through all the quality gates towards production (3 4).

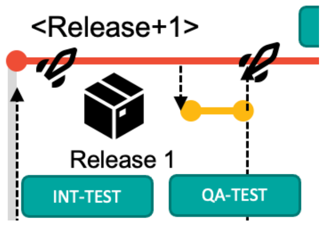

Figure 6 - Building and packaging a release candidate

[12] Semantic Versioning - https://en.wikipedia.org/wiki/Software_versioning#Semantic_versioning
[13] GitHub REST API - Create Branch - https://docs.github.com/en/rest/git/refs#create-a-reference
[14] GitLab REST API - Create Branch - https://docs.gitlab.com/ee/api/branches.html#create-repository-branch
[15] GitLab REST API - Pull Requests - https://docs.github.com/en/rest/pulls/pulls
[16] GitLab REST API - Merge Requests - https://docs.gitlab.com/ee/api/merge_requests.html

Figure 7 outlines the steps of a GitLab pipeline for the Release branch:

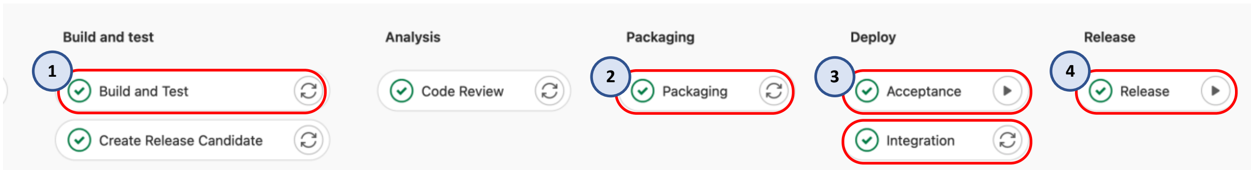

Figure 7 - Implemented release pipeline for the Release branch

3.3.1 Release pipeline: Build and Test

The Build and Test stage of the pipeline for the Release branch needs to build all the incorporated changes of all merged features with the *Optimize* options. To identify the list of changes contributing to the release, the build step of the pipeline leverages the *--baselineRef* option of zAppBuild for incremental builds, which is used to specify a baseline hash when calculating the list of changes. The option *--baselineRef* is a sub-parameter of the *--impactBuild* option in zAppBuild, and sets the base Git hash upon which the *git diff* command calculates changes for the repository[17].

Although it can be done manually, it is recommended to let the pipeline calculate the baseline reference for impact builds in the Release branch. Instead of dealing with Git hashes to set the baseline reference, the model leverages Git tags to identify specific points in the Git history. When a new Release branch is created, a Git tag is also automatically created to easily refer to the state of the repository. When calculating the baseline reference, the pipeline can search for the Git tag of the previous release candidate in the history of Git tags and define this tag as the baseline reference.

In this model, tagging is automated by leveraging the respective APIs for the Git provider (for example, GitLab[18] or GitHub[19]).

3.3.2 Release pipeline: Packaging

The Packaging step runs after the Build and Test step and creates a package of the generated build outputs (for example, load modules, DBRMs, and JCLs). This incremental, yet cumulative package includes the build outputs obtained from all the contributed changes (including the files impacted by the changes) for this release. It represents a release candidate, which can be deployed into the various test environments along the existing staging hierarchy.

The Packaging step not only creates the binaries package, but it also carries information about the source code, such as the Git commit and additional links (including references to the Pull Request and the pipeline execution), which are helpful for understanding the context of the creation of the package.

[17] zAppBuild implementation to set baselineRef - https://github.com/IBM/dbb-zappbuild/blob/main/BUILD.md#perform-impact-build-by-providing-baseline-reference-for-the-analysis-of-changed-files

[18] GitLab REST API – Create Git tag - https://docs.gitlab.com/ee/api/tags.html#create-a-new-tag

[19] GitHub REST API – Create Git tag - https://docs.github.com/en/rest/git/refs#create-a-reference

The Dependency Based Build community repository[20] contains two sample scripts that implement this Packaging step. If IBM UrbanCode Deploy is used as the deployment solution, the *CreateUCDComponentVersion* script[21] can be leveraged to create a UCD component version. Alternatively, if a scripted deployment is being set up, the *PackageBuildOutputs* script[22] can be used instead to store artifacts in an enterprise binary artifact repository.

Both sample scripts leverage data from the DBB Build Report to extract and retain the additional metadata, allowing traceability between the build and deployment activities.

3.3.3 Release pipeline: Deployment

The deployment process of a release package can either be triggered from the CI/CD pipeline or driven through the user interface of the deployment solution. The implementation can vary, based on the capabilities offered by the CI/CD orchestrator and the deployment solution.

IBM UrbanCode Deploy (UCD) provides a rich web-based interface, powerful REST APIs, and command-line interfaces. Typically, the pipeline execution requests the deployment of the application package into the defined test environments automatically, after successful completion of the preceding building and packaging steps. These requests are performed through the REST APIs provided by UCD. However, if the application team prefers to set up manual triggers for the deployments to the specific environments, this can be performed through the Web interface of UCD. In that case, the pipeline is primarily used for continuous integration and packaging.

The DBB Community repository provides a sample *DeployUCDComponentVersion* script[23] to request a UCD application deployment leveraging UCD's REST APIs.

In a GitLab CI/CD implementation, a pipeline can stay on hold and wait for user input. This allows the pipeline to automatically trigger the deployment of the application package into the first configured environment and lets the application team decide when to deploy to the next environment through a manual step (for instance, deployment to the Acceptance environment).

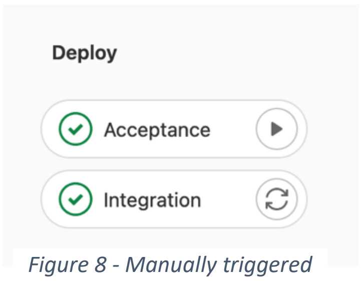

Figure 8 - Manually triggered jobs to deploy release candidate to Acceptance environment

With Jenkins as the CI/CD orchestrator, it is not common to keep a pipeline in progress over a long time. In this case, the pipeline engineering team might consider the approach of requesting the deployments through the user interface of the deployment solution. Alternatively, a deployment pipeline in Jenkins can be designed and set up to combine the deployment with any automated tests or other automation tasks.

3.3.4 Release pipeline: Ready for finalizing the release

Once the application and test teams complete the entire testing process and have validated and approved all changes for the release, the application package is ready to be released into production. The next step is to create the tag in Git to identify the release. This creation can be manually performed, or it can be automated through the pipeline execution.

---

[20] https://github.com/IBM/dbb
[21] Create UCD component version - https://github.com/IBM/dbb/tree/main/Pipeline/CreateUCDComponentVersion
[22] Package Build outputs - https://github.com/IBM/dbb/tree/main/Pipeline/PackageBuildOutputs
[23] Request Application Deployment Sample Script - https://github.com/IBM/dbb/tree/main/Pipeline/DeployUCDComponentVersion

For the automated approach in GitLab, a REST request is submitted through a pipeline step to create the Git tag for the current state of the Release branch[24]. (A similar API is available for GitHub[25].) A naming convention can be used to help identify the final state of the release, for instance *rel-x.x.x-FINAL* .

The pipeline orchestration is set up to run another pipeline when the Git tag is created. This helps to finalize the deployment of the release to the production environment.

3.3.5 Implementing changes in the Release branch

During the release phase, additional changes to the codebase might be required, such as corrections for a fix. To implement those changes, developers will typically create a feature branch from the Release branch. When the change is implemented, a Pull Request is used to integrate these changes into the Release branch. A strategy must be defined to handle those corrections for the release. Several options can be considered:

- Option A: The package containing corrections is kept separate, but needs to be handled together with the initial release candidate package. This leads to additional coordination efforts.
- Option B: A new cumulative binary package containing the baseline build outputs and the output of the corrections is formed and passed to the artifact repository. The documentation of the packaging scripts, *CreateUCDComponentVersion* and *PackageBuildOutputs*, which can implement this strategy by accepting multiple build reports, provides more information.
- Option C: Rather than using an incremental build of the contributed changes for the correction, perform a cumulative impact build that leverages the *--impactBuild* and *--baselineRef* options together to build and package the next release candidate. This strategy might be the most expensive one, as it will produce a new set of binaries that might need to pass all test and quality gates.

3.4 Pipeline steps for the Final Release package

The pipeline for the Final Release package consists of three steps. The first step defines a release in the central Git server, which adds additional information on top of the Final Release Git tag. The next step requests the deployment of the final package to the production runtime, and the last step automates the merging of the changes into the Production/Main branch, through the Pull Request.

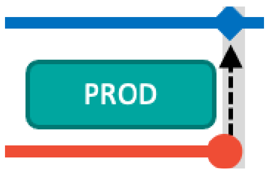

Figure 9 - Process to deploy to production

These steps can either be managed through the pipeline orchestrator or be performed via the deployment solution. In both cases, the sequence of steps remains similar. The design decision might be impacted by the roles and the organization responsible for these actions in the development and release lifecycle.

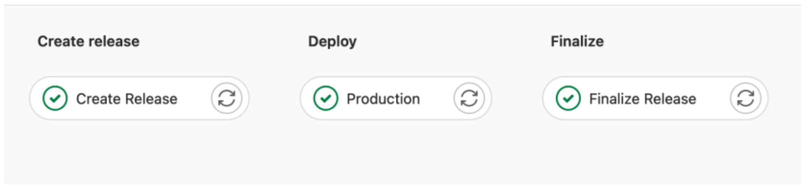

Figure 10 - Pipeline steps for the Final Release package

[24] GitLab REST API – Create Tag - https://docs.gitlab.com/ee/api/tags.html
[25] GitHub REST API – Create Tag - https://docs.github.com/en/rest/git/tags

3.4.1 Creating a release in a central Git server

Most Git providers allow for the creation of a release to provide a summary of the changes, as well as additional documentation. GitLab[26] and GitHub[27] offer a REST API to create a release. These requests can either be driven from the pipeline orchestrator or the deployment manager.

As an example, zAppBuild also declares releases to identify stable versions: https://github.com/IBM/dbb-zappbuild/releases/tag/2.5.0

3.4.2 Deployment to the Production environment

The next step triggers the deployment to the production environment. For IBM UrbanCode Deploy, the previously mentioned *DeployUCDComponentVersion* script can be reused.

In this phase of the lifecycle, the process can potentially integrate additional approvals. For instance, the Operations team can be asked to approve the deployment request and potentially validate the Pull Request on the central Git server.

3.4.3 Automatically merge code into the Production/Main branch

Before merging the code into the Production/Main branch, it is the responsibility of the development team to ensure that changes can be seamlessly integrated: during the development process of the release, emergency fixes may have appeared, and their corresponding changes should be merged into the relevant branches first, including the Release branch.

In the last step of the pipeline for the Release package, the changes of the release are integrated into the Production/Main branch on the central Git server. The changes should be merged from the Release branch into the Production/Main branch through a Pull Request. Again, GitLab[28] and GitHub[29] provide similar APIs to automate the merge of the Pull Request programmatically.

In this merge operation, it is not recommended to squash the commits, as each commit allows the traceability back to the explicit request contributing to the changes. At this point in the workflow, to ensure that the Development branch and all other long-living branches such as active Release branches remain updated with the latest content and Git history from the Production/Main branch, Pull Requests can be created from the Production/Main branch into the Development branch and any other active Release branches.

3.5 Pipeline execution for the Production/Main branch

After the Pull Request is merged into the Production/Main branch, a pipeline for this branch is executed to refresh DBB's dependency metadata for the Production/Main collection. This is achieved by running a build with the *--impactBuild --scanAll* options. The purpose of this scan is to keep DBB's dependency information for this branch up to date for any potential emergency fixes that would need to be implemented from the Production/Main configuration.

[26] GitLab REST API – Create Release https://docs.gitlab.com/ee/api/releases/
[27] GitHub REST API – Create Release https://docs.github.com/en/rest/releases/releases
[28] GitLab REST API – Merge a Merge Request https://docs.gitlab.com/ee/api/merge_requests.html#merge-a-merge-request
[29] GitHub REST API – Merge a Pull Request https://docs.github.com/en/rest/pulls/pulls#merge-a-pull-request

4 Conclusion

This document provides guidance for implementing a release-based development process for mainframe applications. The CI/CD pipeline configurations that were outlined at various steps of the release development cycle can be adjusted depending on the application team's existing and desired development processes and philosophy. Factors that might impact the design of the pipelines and workflow include test strategies, the number of test environments, and potential testing limitations.

When architecting a CI/CD pipeline, assessment of current and future requirements in the software delivery lifecycle is key. As CI/CD technologies continue to evolve and automated testing using provisioned test environments becomes more common in mainframe application development teams, the release-based branching strategy can also evolve to maximize the benefits from these advances.

5 Appendix

5.1 Use curl to trigger actions in the central Git provider

As previously highlighted, some steps of the pipeline can be automated through the REST APIs or command-line interface provided by the Git provider. Depending on the Git server and pipeline orchestrator, different approaches exist to manage the credentials to perform these actions.

The GitLab platform uses tokens to identify users and provides the capability to define bots, which help automate some steps of the CI/CD pipeline[30]. Bots can possess their own tokens, which can be provided when REST API requests are performed. Authentication tokens are passed through the header of the REST API request.

The following snippets are examples of automated steps in the GitLab CI/CD pipeline orchestrator:

- Calculation of the release number depending on the argument passed in for the *releaseType* parameter:

```
# evaluate the provided release type
if [ "$releaseType" == "patch" ]; then export releaseName=rel-`git tag --sort=-committerdate | head -n 1 | sed 's/^[rel-]*//g' | sed 's/-[a-zA-Z0-9]*//g' | awk -F. -v OFS=. '{$3 += 1 ; print}'`; fi;
if [ "$releaseType" == "minor" ]; then export releaseName=rel-`git tag --sort=-committerdate | head -n 1 | sed 's/^[rel-]*//g' | sed 's/-[a-zA-Z0-9]*//g' | awk -F. -v OFS=. '{$2 += 1 ; $3 = 0; print}'`; fi;
if [ "$releaseType" == "major" ]; then export releaseName=rel-`git tag --sort=-committerdate | head -n 1 | sed 's/^[rel-]*//g' | sed 's/-[a-zA-Z0-9]*//g' | awk -F. -v OFS=. '{$1 += 1 ; $2 = 0; $3 = 0; print}'`; fi;
```

- Sending a REST Request to the GitLab server to create a new branch:

```
curl --request POST --header "PRIVATE-TOKEN: ${AutomationToken}" "${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/repository/branches?ref=${CI_COMMIT_REF_NAME}&branch=${releaseName}"
```

- Sending a REST Request to the GitLab server to create the Merge Request:

```
curl --request POST --header "PRIVATE-TOKEN: ${AutomationToken}" "${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/merge_requests?source_branch=${releaseName}&target_branch=production&title=Merge%20Request%20for%20Release%20${releaseName}%20deployment&remove_source_branch=true&squash=false"
```

Note: The above snippets are simplified fragments and do not include the evaluation of return codes from the REST request.

[30] GitLab Token Overview - https://docs.gitlab.com/ee/security/token_overview.html