

*IBM Z DevOps Guide*



**Edition 1.0.3 (September 2024)**

**© Copyright International Business Machines Corporation 2024.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

---

# Contents

<b>Notices.....</b>	<b>i</b>
<b>Chapter 1. Overview.....</b>	<b>1</b>
Applying DevOps to IBM Z.....	1
IBM Z DevOps Acceleration Program.....	1
CI/CD for z/OS applications.....	2
CI/CD for z/OS applications.....	2
Integrated development environment.....	5
Source code management.....	5
Build.....	10
Artifact repository.....	13
Deployment manager.....	14
Pipeline orchestrator.....	14
<b>Chapter 2. Getting started.....</b>	<b>17</b>
Understanding the journey to a CI/CD pipeline.....	17
DevOps transformation approaches.....	17
Roles.....	17
Transformation journey milestones.....	22
Installing and configuring your tools.....	24
Integrated development environment.....	24
Source code management.....	24
Build.....	26
Artifact repository.....	30
Deployment manager.....	30
Pipeline orchestrator.....	30
Resources.....	30
<b>Chapter 3. Working practices.....</b>	<b>33</b>
Introduction to working practices for teams using Git.....	33
Aims and Assumptions.....	33
Choosing a workflow and branching model.....	34
The Git branching model for mainframe development.....	34
Characteristics of trunk-based development with feature branches.....	34
Workflows in this branching model.....	39
Learn more.....	50
Working practice variations.....	50
Working with overlapping releases.....	50
Audit and compliance.....	51
What is an audit?.....	51
Application audit requirements.....	51
Application audit compliance practices.....	52
Further reading.....	53
<b>Chapter 4. Tutorials and courses.....</b>	<b>55</b>
Tutorials.....	55
IBM Z Systems software trials (IBM Z Trial).....	55
CI/CD pipeline tutorials.....	55
Courses.....	55
Learn to code and develop in a modern integrated development environment.....	56

Gain insights into your z/OS applications with analysis tooling.....	56
Use an intelligent build tool to compile and link your z/OS applications.....	56
Streamline quality assurance with IBM Z testing tools.....	56
Modernize with cloud native DevOps tooling.....	56
<b>Chapter 5. Designing the CI/CD pipeline.....</b>	<b>57</b>
Planning repository layouts and scopes.....	57
Basic components in a Git SCM layout.....	57
Working with Git.....	57
Visualizing the SCM layout.....	58
Guidelines for Git repository scope and structure.....	60
Managing dependencies between applications in separate repositories.....	62
Advanced topic: Options for efficient use of Git repositories.....	63
Defining dependency management.....	65
Layout of dependencies of a mainframe application.....	65
Applications and programs.....	67
Applications and application groups.....	68
Cross-cutting interfaces.....	69
Managing and life cycle of shared interfaces.....	70
Resources.....	72
Designing the build strategy.....	72
User build.....	73
Pipeline build.....	73
Resources.....	75
Architecting the pipeline strategy.....	75
Building, packaging, and deploying in a pipeline.....	75
Package content and layout.....	76
Package strategy and scope.....	77
Specialty of Mainframe Packages.....	79
Resources.....	79
Outlining the deployment strategy.....	79
Introducing deployment managers.....	79
Resources.....	80
<b>Chapter 6. Migrating data from z/OS to Git.....</b>	<b>81</b>
Migrating data from z/OS to Git.....	81
Introduction to migrating data.....	81
Steps for migrating data from z/OS to Git.....	81
Managing code page conversion.....	81
Introduction.....	81
Understanding the code page translation process.....	82
Defining the code page of files in Git.....	86
Managing the code page on z/OS UNIX System Services.....	87
Using the DBB Migration Tool.....	88
Summary.....	88
Resources.....	88
Methods for migrating data.....	88
DBB Migration Tool.....	88
SCLM-to-Git Migration Tool.....	99
Manual migration.....	100
<b>Chapter 7. Implementing CI/CD.....</b>	<b>101</b>
Implementing a pipeline for the branching model.....	101
Configurations to support working with feature branches.....	101
The Basic Build Pipeline for main, epic, and release branches.....	104
The Release Pipeline with Build, Package, and Deploy stages.....	107
Deployment to production.....	109

Conclusion.....	110
Implementing pipeline actions.....	110
Common Back-end Scripts.....	110
Specific guides for each orchestrator.....	110
<b>Chapter 8. Resources.....</b>	<b>113</b>
Additional resources.....	113
<b>Legal information.....</b>	<b>115</b>
Trademarks.....	115
Privacy policy considerations.....	115



## Notices

---

This information was developed for products and services offered in the US. This material might be available from IBM® in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing, IBM Corporation, North Castle Drive, MD-NC119, Armonk, NY 10504-1785, US*

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

### COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under

all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.



---

# Chapter 1. Overview

## Applying DevOps to IBM Z

---

Our prescriptive guidance for applying DevOps to IBM Z® aims to align tools, working practices, and outcomes with other platforms in your enterprise as much as we can.

Its foundation is a continuous integration/continuous delivery (CI/CD) pipeline consisting of several key components including a Git-based service for source code management and IBM® Dependency Based Build (DBB).

The recommended components are loosely coupled and highly customizable, allowing you to implement a CI/CD pipeline that maximizes the tools and processes shared by all your development teams, mainframe or not.

There are multiple advantages to adopting a standardized CI/CD pipeline across your enterprise:

- Equip developers for productivity
  - Open source tools such as Git, Jenkins, and Artifactory have become the de facto industry standard for powering continuous integration and continuous delivery in the enterprise. These tools support today's best practices and provide advanced functionality in version control, automation, and more.
- Collaborate and communicate more effectively across the enterprise
  - Standardizing the CI/CD pipeline and development workflows where possible between mainframe and other teams makes it easier to communicate and collaborate across the enterprise. This is especially beneficial for hybrid applications, which consist of both mainframe and non-mainframe components.
- Enable modern approaches such as DevOps and Agile for mainframe applications
  - Industry-standard tools such as version control with Git and automation with CI/CD pipelines enable smaller, high-quality development increments and deliverables. This lets your z/OS® application teams respond faster and more effectively to changing market pressures or customer needs, while also building towards a larger development goal.
- Reduce the overhead of onboarding the next generation of z/OS application developers.
  - Newer developers are likely to be more familiar with industry-standard tools, and being able to use these technologies in the z/OS workplace lowers the entry barrier for them to get started with contributions to the team.

This documentation site aims to help your enterprise succeed in the journey from legacy development processes and library managers to a modern CI/CD pipeline.

## IBM Z DevOps Acceleration Program

The IBM Z DevOps Acceleration Program (DAP) can help your enterprise get started on the journey to adopt this modern and agile approach to z/OS applications. DAP is a no charge value-add early adoption program designed to partner with clients during four distinct stages that are necessary for any DevOps transformation:

- Value Stream Assessment
- Training
- Deployment
- Adoption

To learn what's involved in migrating your z/OS applications to our recommended toolchain and working practices, see [Understanding the migration effort](#).

To learn more about the CI/CD pipeline itself for z/OS applications, see the [CI/CD pipeline introduction](#).

## CI/CD for z/OS applications

---

### CI/CD for z/OS applications

#### Introduction to CI/CD

CI/CD is a development process that empowers enterprises to develop and deliver higher quality software more frequently. The "CI" in "CI/CD" stands for "continuous integration", while the "CD" can stand for either "continuous delivery" and/or "continuous deployment".

- "Continuous integration" means that new code is merged into the shared code base at more frequent intervals (for example, daily or weekly). This allows for more frequent builds of the code base, enabling more frequent automated testing against the application builds, which can help identify integration issues earlier in the development cycle, when they are easier to fix, and subsequently reduce issues closer to release.
- "Continuous delivery" is when the changes to an application are automatically tested and uploaded (delivered) to a repository, where it can then be deployed to different environments. This practice can reduce time to deployment due to the integration of continuous testing and development, and can thus reduce the cost of development without compromising quality. Although the deployment itself is automated, manual approval is still required to authorize the deployment.
- "Continuous deployment" is when the changes to an application are automatically deployed to a production environment for use by your customers. It is considered to be one step further than continuous delivery, and might or might not be implemented depending on business needs. With continuous deployment, the developers and automated tests in production-like test environments are trusted enough that the new code is considered approved when it passes the tests. Deployment no longer requires manual approval and is also automated to speed the time to customer value.

CI/CD is powered by a set of tools that automates processes in the software development lifecycle. As a whole, the CI/CD pipeline can enable teams to adopt Agile and DevOps workflows so that application teams can deliver changes more frequently and reliably, and respond more efficiently to feedback from operations. For this reason, CI/CD is considered a best practice in today's industry standards. IBM®'s general [DevOps](#) page provides additional information on CI/CD in the context of DevOps.

#### Applying CI/CD to z/OS applications

Legacy z/OS® development tools and processes have traditionally made it difficult for z/OS applications to participate in modern CI/CD pipelines. However, our recommended toolchain allows you to integrate z/OS applications into the same CI/CD pipelines used by teams developing non-mainframe applications. That means z/OS teams can now use the same modern, industry standard, open source technologies for integrated development environments (IDEs), version control, automation, and more. You can learn more about how IBM supports DevOps for the mainframe at the [IBM Z® DevOps](#) and [hybrid cloud continuous integration](#) pages.

Using a Git-based source code management and the IBM Dependency Based Build (DBB) tool, you can integrate with other industry-standard [CI/CD pipeline tools](#), so that the developer workflow looks something like the [Day in the life of a developer](#) described in the following section.

#### *Day in the life of a developer*

With the IBM Z DevOps CI/CD pipeline in place, an application developer would typically use the following (or similar) workflow when completing a development task (for example, a bugfix or feature enhancement):

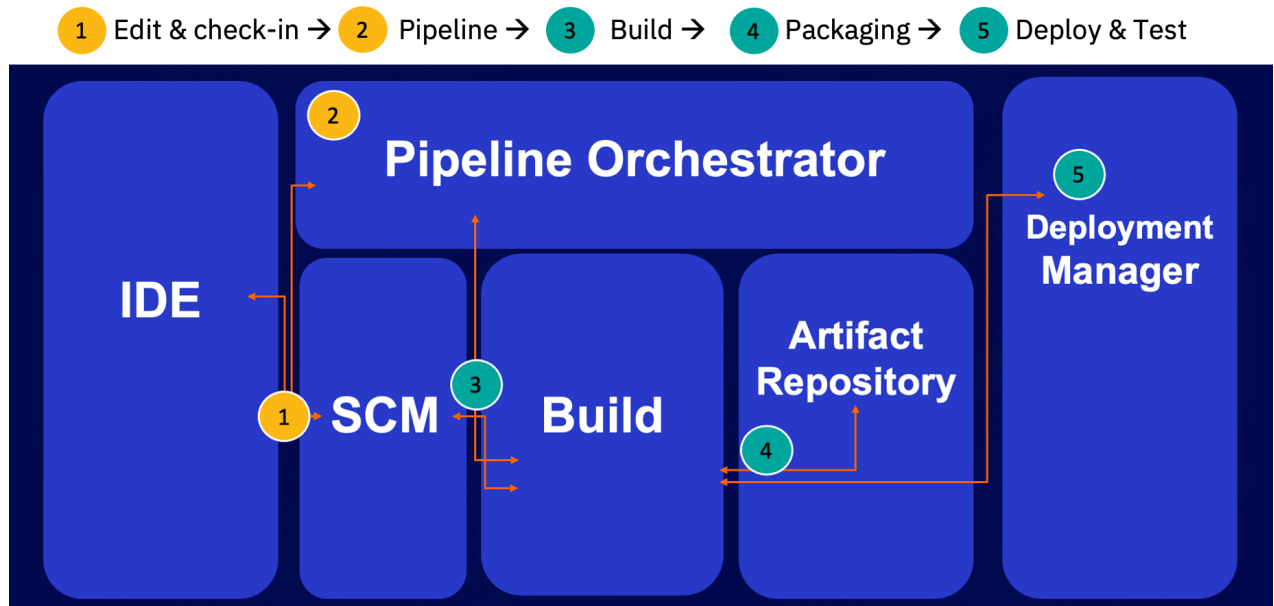
**Tip:** The IBM Z DevOps CI/CD pipeline is based on Git as the source code management tool. If you are new to Git and its concepts (such as branches and pull/merge requests), you can check out the [Source code management](#) page to get familiar with the terminology so you can better understand the workflow below.

1. The developer will start by cloning or pulling a copy of the application code from a central Git repository down to her local workstation.
2. She can then create a new personal branch of that code for her specific task. This will allow her to work on the task in isolation, in parallel with her team, without having to worry about other development activities disturbing her work.
3. Once she has made her code changes that she is ready to test, she can use IBM's Dependency Based Build (DBB) tool to build the program so that she can test it individually and verify that her fix works and does not cause regressions.
4. Once she is happy with her changes, she can commit them to her personal branch of code, and then push her personal branch with her code changes to the central Git repository.
5. Now, she can open a pull request to have the changes in her personal branch of code be merged into the shared, common branch of code for her team. Her team has set up an automated pipeline to run builds of the code in the pull request, which also include tests and code scans.
  - This is also the point at which her team has an approvals process where she can add teammates to review her changes and approve them before merging them into their common branch of code.
6. Once her pull request is approved and her changes are merged into the common branch of code, the personal branch of code where she originally did her development work can be deleted, and a full or impact pipeline build can run on the common branch of code to move the changes forward or associate them with a release.

### **Components of a CI/CD pipeline for z/OS applications**

The developer's workflow in the previous section is enabled by CI/CD pipeline tooling, the major components of which are summarized in this section. The following image depicts steps in the CI/CD pipeline with two different colors, yellow and green. Yellow steps highlight the steps performed by the developer, while green steps are performed by the pipeline. For the developer, their steps include check-out and check-in of code, but also include triggering of a pipeline job. Developer-level operations end at this point.

All subsequent steps, in green, are performed by the pipeline. These steps include building, publishing to an artifact repository, and deploying to an execution environment.



You can click on each component in the following list to learn more about it and see common technology options:

- **Integrated development environment (IDE):** The IDE is what the developer uses to check out and edit her code, as well it check it back into the version control system. Many modern editors have

features that enhance development capabilities, such as syntax highlighting, code completion, outline view navigation, and variable lookups, as well as integrations such as debugging and unit testing.

- **Source code management (SCM, Version control):** The SCM is used to store and manage different versions of source code files, as well as application configuration files, test cases, and more. This is what enables the application development team to do parallel development. We recommend a Git-based SCM. For more information about Git and why it is foundational to our recommendations, as well as an explanation of the Git concepts, see the [SCM](#) documentation.
- **Build:** The build component takes care of understanding dependencies, and then compiling and linking programs to produce the executable binaries such as load modules and DBRMs. When running this component, you can also integrate automated steps for unit testing and code quality inspection (although these are sometimes considered as separate components in the CI/CD pipeline). We recommend that the build is handled by IBM Dependency Based Build (DBB), which has intelligent build capabilities that enable you to perform different types of build to support various steps in your workflow. Some examples of these build types include single-program user builds, full application builds, and impact builds.
- **Artifact repository:** Once the build component has created the executable binaries, they are packaged together and uploaded into the artifact repository, along with metadata to help trace those binaries back to the source. This component is crucial for decoupling the source code management from the runtime environments, enabling the key DevOps practice of "Build once, deploy many".
- **Deployment manager:** The deployment manager is the tool that rolls out the application packages. When it is time to deploy the application, the deployment manager downloads the package from the artifact repository and uploads the contents to the target libraries. If there are other steps to perform, such as installation steps like CICS® NEWCOPY or PHASE-IN, or a bind step when Db2® is involved, the deployment manager also handles those. Importantly, it also keeps track of the inventory of execution environments so that you can know what each environment is running.
- **Pipeline orchestrator:** The pipeline orchestrator oversees all the automated processes in the pipeline. This component integrates the steps from the different tools together and ensures they all run in the correct order.

Although it might seem CI/CD requires developers to learn and work with a lot of different tools, they are primarily just working with the IDE for code editing, the SCM for version control, and performing some individual user builds. Once development gets to point where they want to integrate their code changes into their team's shared codebase, the pipeline is largely automated via the pipeline orchestrator. This means that once the CI/CD pipeline is in place, if the developer has to interact with any of the automated components at all, they would mostly just be checking a dashboard or status, performing any intentionally manual approval steps, and/or verifying the results of the pipeline job.

*How do you select what tool to use for each component?*

You will find for many of the CI/CD pipeline components, multiple tools are available to perform the functionality. However, to reap the benefits of standardization across the enterprise, we generally recommend that clients pick the option that other parts of their organization are already using. Popular tooling options for each component in this section are listed on the respective component's dedicated page, although it should be noted that these lists are not necessarily all-inclusive. For some more common combinations of technologies, the IBM Z DevOps Acceleration Team has produced detailed documentation on the setup and implementation.

### ***What does it mean to work with a CI/CD pipeline?***

The CI/CD pipeline building block components supply facilities to the development and delivery teams, such as:

- Isolation capabilities to reduce coordination efforts in larger groups
- Integration workflows and traceability from tasks to code changes, binaries, and deliveries
- Standardized development practices across teams and platforms
- Automation of build, packaging, and deployment tasks

When it comes to isolation techniques, we refer to the branching strategies for the version control system. When using Git, branches are created to allow parallel development of the code, with each branch dedicated to a particular purpose, such as stable integration of the team's code or an individual developer's work on a hotfix or new feature. Developers can now integrate and share changes with the team through the enterprise's central Git provider. Accordingly, additional workflows like merge or pull requests for code review and approvals are expected to be used. These concepts are further introduced in the [SCM component overview page](#).

Compared to a library manager, which relies on concatenation for compilation and linking, Git and its branches provide complete isolation and remove dependencies on the areas under current development by other development teams. The developer therefore works within the scope of the entire application. This also implies that a branch does not represent the contents of an execution environment. The branch is in fact fully decoupled from the environment via the artifact repository, ensuring a complete separation of the build and deployment phases. This decoupling enables developers to adopt previously impossible provisioning practices such as spinning up an isolated test execution environment with the push of a button.

Defining an application package in a CI/CD pipeline will be different from the largely manual ad-hoc approaches seen in traditional mainframe library managers. With a CI/CD pipeline, the outputs of the build process are preconfigured to automatically be packaged together upon generation. These application packages are the inputs to the deployment manager, and the deployment manager is responsible for the installation of the packages to the execution environment.

## Resources

This page contains reformatted excerpts from [Packaging and Deployment Strategies in an Open and Modern CI/CD Pipeline focusing on Mainframe Software Development](#).

## Integrated development environment

An integrated development environment (IDE) provides check-out and check-in capabilities to the [source code management \(SCM\)](#) system. Typically, it supports the language and coding practices of the developer, enables building the modified code within the sandbox, and drives unit tests as part of the coding phase.

We embrace a "Bring Your Own IDE" approach, meaning we want to enable flexibility in choice of IDE.

### Common IDE options for z/OS application development

- Eclipse-based: IBM® Developer for z/OS® (IDz)
- Visual Studio Code (VS Code) extension: IBM Z® Open Editor (and IBM Z Open Debug)
- Browser-based: Wazi for Dev Spaces

## Resources

This page contains reformatted excerpts from [Packaging and Deployment Strategies in an Open and Modern CI/CD Pipeline focusing on Mainframe Software Development](#).

## Source code management

A source code management (SCM) tool manages and stores different versions of your application configuration such as source code files, application-specific configuration data, test cases, and more. It provides capabilities to isolate different development activities and enables parallel development.

In our described continuous integration/continuous delivery (CI/CD) implementation, we showcase Git as the SCM when applying DevOps to IBM Z®.

## Why move to Git?

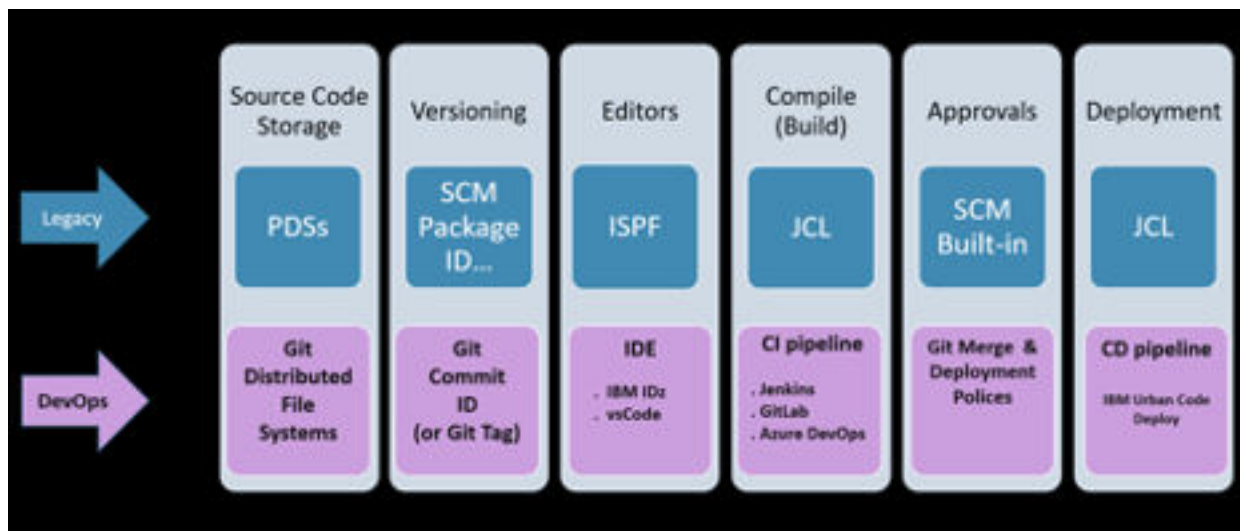
Git is the de facto industry standard SCM for the open source community and is growing in popularity among major organizations. It is a central part of the modern developer's toolkit, and provides a common SCM tool for hybrid application architectures that can span across components ranging from those implemented in traditional mainframe languages such as COBOL, PL/I, or Assembler, to components for the service layer such as z/OS® Connect, and components in Java™ and Go, to reflect the architecture of the business application.

Git integrates with most modern DevOps tools and pipeline processes to support the full development lifecycle from continuous integration (CI) to continuous delivery (CD). By migrating to Git as the enterprise SCM, mainframe application development teams can then take advantage of the open source community's modern tooling.

## Git in the software development lifecycle

The following diagram draws analogies between legacy mainframe SCM processes and DevOps with Git on z/OS.

It shows the key features of an SCM process, starting with source code storage, and ending with the deployment to the development or other upper environments such as Q/A and Production.



## Git basics

Git is a distributed "version control system" for source code. It provides many features to allow developers to check in and check out code with a full history and audit trail for all changes.

Source is stored in repositories (also known as "repos") on hierarchical file systems on Linux®, MacOS, Windows, or z/OS UNIX System Services.

The team stores a primary copy of the repository on a service running Git on a server (see [Common Git provider options](#)). Such services provide all the resilience required to safeguard the code and its history. Once source code is moved into a repository on the server, that becomes the primary source of truth, so existing processes to ensure the resilience of copies on z/OS are no longer required.

An application repo can be cloned from the team's chosen Git server (known as the "remote") to any machine that has Git, including a developer's local computer using popular integrated development environments (IDEs) such as IBM® Developer for z/OS (IDz) and Microsoft's Visual Studio Code (VS Code). By default, clones contain all the files and folders in the repository, as well as their complete version histories. (Cloning provides many options to select what is copied and synchronized.)

All Git operations that transfer the data held in the repository (`clone`, `push`, `fetch`, and `pull`) use SSH or HTTPS secure communications. Pros and cons of each protocol are discussed in ["Git on the Server - The Protocols"](#).

Info:

z/OS UNIX System Services includes [OpenSSH](#). z/OS OpenSSH provides the following z/OS extensions:

- System Authorization Facility (SAF) key ring: z/OS OpenSSH can be configured to allow z/OS OpenSSH keys to be stored in SAF key rings.
- Multilevel security: This is a security policy that allows the classification of data and users based on a system of hierarchical security levels combined with a system of non-hierarchical security categories.
- System Management Facility (SMF): z/OS OpenSSH can be configured to collect SMF Type 119 records for both the client and the server.
- Hardware Cryptographic Support: OpenSSH can be configured to choose Integrated Cryptographic Service Facility (ICSF) callable service for implementing the applicable SSH session ciphers and HMACs.

The developer can then create "branches" in the repository. Branches allow developers to make and commit changes to any files in the repository in isolation from other developers working in other branches, or for an individual developer to work on multiple work items that each have their own branch.

For each task the developer has (such as a bug fix or feature), the developer would generally do their development work on a branch dedicated to that task. When they are ready to promote their changes, they can create a "pull request", (also known as a "merge request") which is a request to integrate (or "merge") those changes back into the team's common, shared branch of code.

With Git's branching and merging features, changes can be performed in isolation and in parallel with other developer changes. Git is typically hosted by service providers such as GitHub, GitLab, Bitbucket, or Azure Repos. Git providers add valuable features on top of the base Git functionality, such as repository hosting, data storage, and security.

In Git, all changes are committed (saved) in a repo using a commit hash (unique identifier) and a descriptive comment. Most IDEs provide a Git history tool to navigate changes and drill down to line-by-line details in Git diff reports. The following image of an Azure Repos example setup shows the Git history on the right panel, and a Git diff report on the left.



As part of comprehensive integrity assurance, developers can [cryptographically sign their commits](#).

## Git branching

A Git "branch" is a reference to all the files in a repo at a certain point in time, as well as their history. A normal practice is to create multiple branches in a repo, each for a different purpose. Typically, there will be a "main" branch, which is shared by the development team. The team's repository administrator(s) will usually set up protections for this branch, requiring approval for any change to be merged into it. The team might also have additional shared branches for different purposes, depending on their branching strategy. The repository administrator(s) can also set up branch protections for these branches, as well as any other branch in the repository.

All Git actions are performed on a branch, and a key advantage of Git is that it allows developers to clone a repo and create (check out) a new branch (sometimes called a "feature branch") to work on their own

changes in isolation from the main source branch. This lets each developer focus on their task without having to worry about other developers' activities disturbing their work or vice versa.

When a developer wants to save their code changes onto a branch in Git, they perform a Git "commit", which creates a snapshot of the branch with their changes. Git uniquely identifies this snapshot with a commit hash, and attaches a short commit message from the developer describing the changes. The developer (and any other teammates with access to the branch) can then use this commit hash as a point-in-time reference for the set of committed changes. They can later check out the commit hash to view the code at that commit point. Additionally, the code can also be rolled back (or "reverted", in Git terminology) to any prior commit hash.

## ***Git merge***

Feature branching allows developers to work on the same code, and work in parallel and in isolation. Git merge is how all the code changes from one branch get integrated into another branch. Once developers complete their feature development, they initiate a pull request asking to integrate their feature changes into the team's shared branch of code.

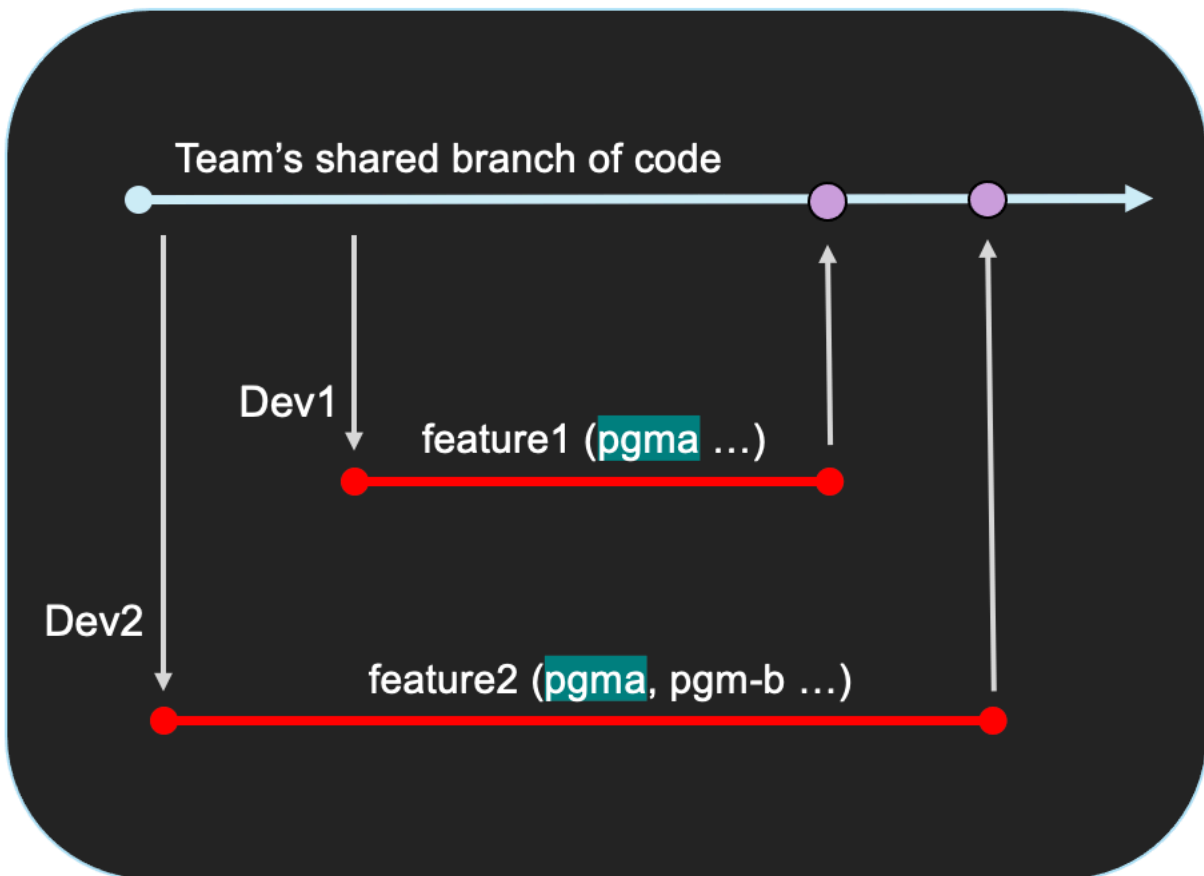
The pull request process is where development teams can implement peer reviews, allowing team leads or other developers to approve or reject changes. They can also set up other quality gates such as automated testing and code scanning to run on the PR. Git will automatically perform merge conflict detection to prevent the accidental overlaying of changes when the pull request is merged in. Development teams often have a CI pipeline that is triggered to run upon pull request approval/merge for the integration test phase.

### *Merge conflict detection: parallel development use case*

One of the biggest benefits of using Git is its merge conflict detection. This is Git's ability to detect when there are overlaps in the code changes during a merge process, so that developers can stop the merge and resolve the merge conflict. This merge conflict detection means that team members can merge their changes to the same program while avoiding unintentionally overlaying each other's code.

To illustrate this example of parallel development, in the following diagram, Developer 1 (Dev1) and Developer 2 (Dev2) have each created their own feature branch from the same version of their team's shared branch of code. Note that there are no commits (indicated by purple dots) on the team's shared branch between when Dev2 and Dev1 created their respective feature branches. Now, each developer can work on their own feature in isolation: Dev1 has his `feature1` branch where he is working on his copy of the code, and Dev2 has her `feature2` branch where she is working on her copy of the code.





Doing this kind of parallel development is complicated on legacy systems, especially with PDSs, because developers have to figure out how to merge the code at the end, especially when working on the same files. Additionally, legacy SCMs typically lock files that are being worked on. In contrast, Git branching allows the developers to work on the files at the same time, in parallel.

In the Git example illustrated above, Dev1 and Dev2 agreed to work on different parts of the same program, and they then each make their own pull request to integrate their respective changes back into the team's shared branch of code when they are ready. Dev1 has done this before Dev2, so his changes have been approved and merged in first. When Dev2 later makes her request to merge her code changes into the team's shared branch of code, Git does a line-by-line check to make sure the changes proposed in Dev2's pull request do not conflict with any of the changes in the shared branch of code (which now include Dev1's changes). If any issues are found, Git will stop the merge and alert the developers of the merge conflict. Git will also highlight the conflicting code so that the developers know where to look and can resolve the conflict, most likely via another commit in Dev2's branch.

### **Git tags**

A Git tag references the repo with a specific, unique commit point. Tags are optional but are strongly recommended and broadly used in modern development practices with Git.

### **Common Git provider options**

- [GitLab](#)
- [GitHub](#)
- [Atlassian Bitbucket](#)
- [Azure Repos](#)

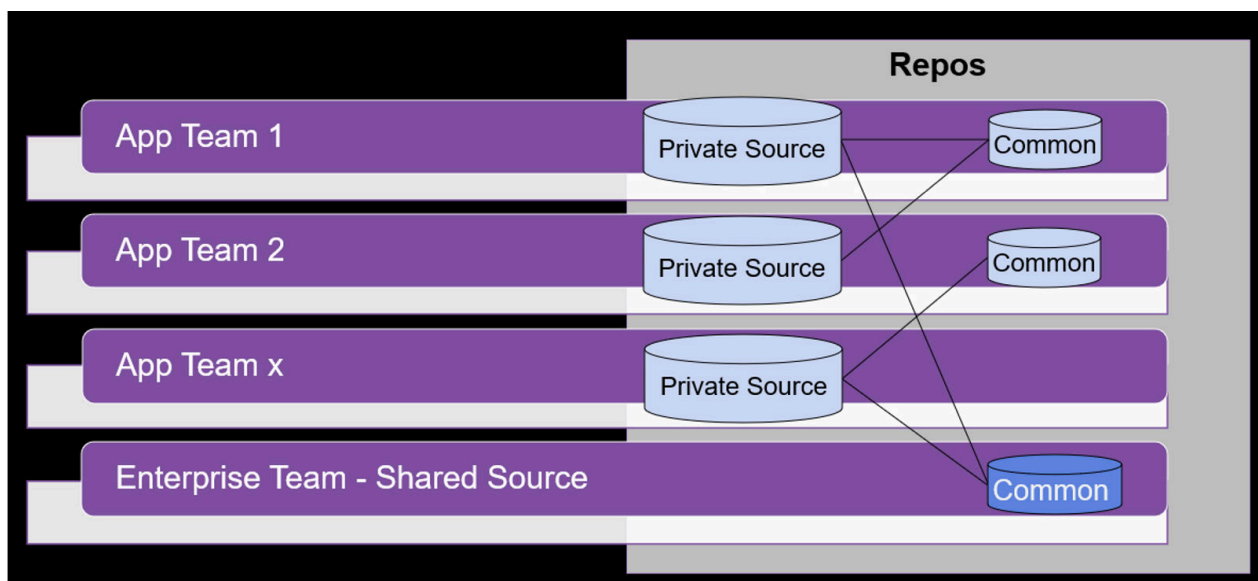
## Best practices

### Sharing code

It is a common practice that mainframe applications share common code. For example, COBOL copybooks are typically shared across applications that process similar data.

The following diagram illustrates how teams can define repos to securely share common code. In this example, App Team 1 has common code that App Team 2 can clone and use in their build.

Another example (also illustrated in the following diagram) is that an enterprise-wide team can maintain source that is common across many applications.



## Resources

This page contains reformatted excerpts from [Git training for Mainframers](#).

## Build

The Build component of a continuous integration/continuous delivery (CI/CD) pipeline converts the source code into executable binaries. It supports multiple platforms and languages. In mainframe environments, it includes understanding dependencies, compile, link-edit, and unit test. The build can include the inspection of code quality to perform automated validation against a set of coding rules. In some cases, code quality inspection could also be a component of its own in the pipeline.

While many of the steps in the DevOps flow for mainframe applications can be performed using the same tooling used by other development teams, the build step in particular needs to remain on z/OS®. Therefore, this step is primarily handled by IBM® Dependency Based Build (DBB). DBB has intelligent build capabilities where it can not only compile and link z/OS programs to produce executable binaries, but it can also perform different types of builds to support the various steps in an application development workflow. This includes the ability to perform an "impact build", where DBB will only build programs that have changed since the last successful build and the files impacted by those changes, saving time and resources during the development process.

DBB is a set of APIs based on open-source Groovy and adapted to z/OS. This enables you to easily incorporate your z/OS application builds into the same automated CI/CD pipeline used by other teams. It is possible to use DBB as a basis to [write your own build scripts](#), but we recommend starting with the [zAppBuild framework](#) to provide a template for your build, and then customizing it as necessary for your enterprise and applications.

The zAppBuild framework helps facilitate the adoption of DBB APIs for your enterprise and applications. Rather than writing your own Groovy scripts to interact with the DBB APIs, you can fill in properties to define your build options for zAppBuild, and then let zAppBuild invoke DBB to perform your builds.

## DBB features

- Perform builds on z/OS and persist build results
- Persist metadata about the builds, which can then be used in subsequent automated CI/CD pipeline steps, as well as informing future DBB builds
- Can be run from the command line interface, making it easy to integrate into an automated CI/CD pipeline

## zAppBuild features

- Framework template facilitates leveraging DBB APIs to build z/OS applications, letting you focus on defining the build's properties separately from the logic to perform the build
- High-level enterprise-wide settings that can be set for all z/OS application builds
- Application-level settings for any necessary overrides in individual application builds
- Includes out-of-the-box support for the following languages:
  - COBOL
  - PL/I
  - BMS and MFS
  - Link Cards
  - PSB, DBD
  - See zAppBuild's [Supported Languages](#) for a full list of out-of-the-box supported languages.
- Supported build actions:
  - Single program ("User Build"): Build a single program in the application
  - List of programs: Build a list of programs provided by a text file
  - Full build: Build all programs (or buildable files) of an application
  - Impact build: Build only the programs impacted by source files that have changed since the last successful build
  - Scan only: Skip the actual building and only scan source files for dependency data
  - Additional supported build actions are listed in zAppBuild's [Build Scope](#) documentation.

## zAppBuild introduction

zAppBuild is a free, generic mainframe application build framework that customers can extend to meet their DevOps needs. It is available under the Apache 2.0 license, and is a sample to get you started with building Git-based source code on z/OS UNIX System Services (z/OS UNIX). It is made up of property files to configure the build behavior, and Groovy scripts that invoke the DBB toolkit APIs.

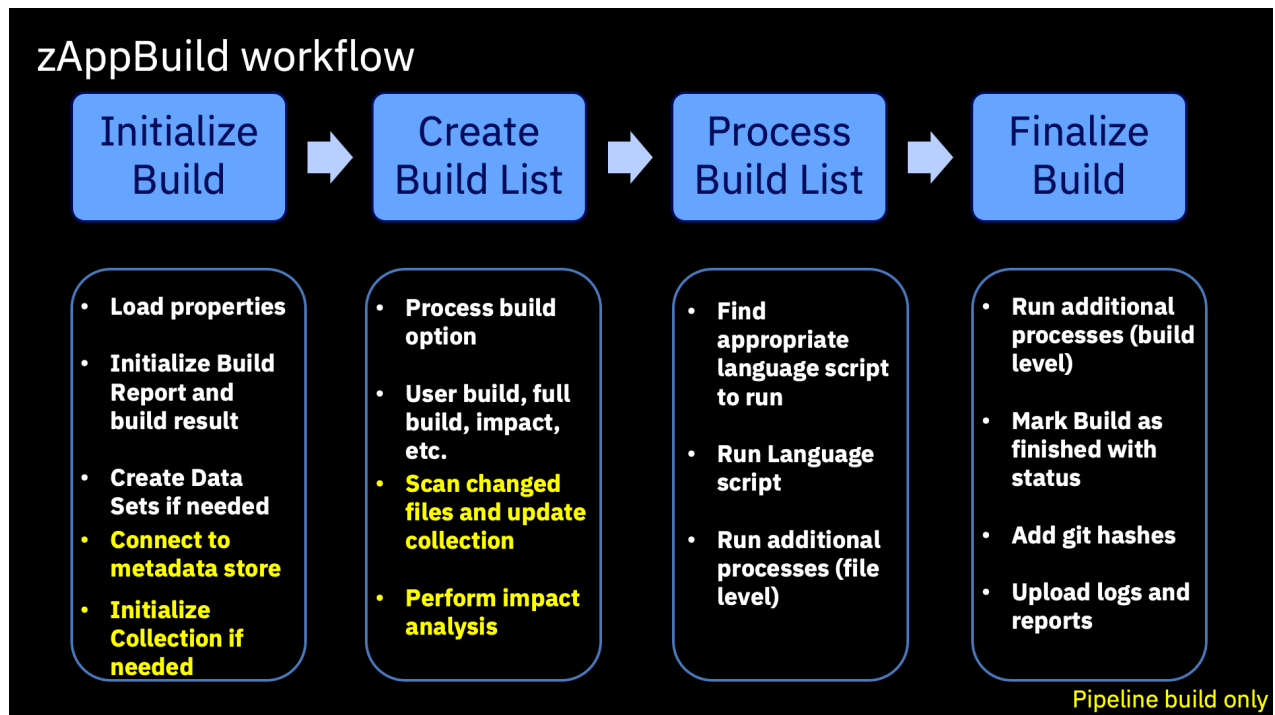
Build properties can span across all applications (enterprise-level), one application (application-level), or individual programs. Properties that cross all applications are managed by administrators and define enterprise-wide settings such as the PDS name of the compiler, data set allocation attributes, and more. Application- and program-level properties are typically managed within the application repository itself.

The zAppBuild framework is invoked either by a developer using the "User Build" capability in their integrated development environment (IDE), or by an automated CI/CD pipeline. It supports different [build types](#).

The main script of zAppBuild, `build.groovy`, initializes the build environment, identifies what to build, and invokes language scripts. This triggers the utilities and DBB APIs to then produce runtime artifacts.

The build process also creates logs and an artifact manifest (`BuildReport.json`) for deployment processes coordinated by the deployment manager.

The following chart provides a high-level summary of the steps that zAppBuild performs during a build:



### **zAppBuild architecture**

The zAppBuild framework is split into two parts. The core build framework, called `dbb-zappbuild`, is a Git repository that contains the build scripts and stores enterprise-level settings. It resides in a permanent location on the z/OS UNIX file system (in addition to the central Git repository). It is typically owned and controlled by the central build team.

The other part of zAppBuild is the `application-conf` folder that resides within each application repository to provide application-level settings to the central build framework. These settings are owned, maintained, and updated by the application team.

#### *dbb-zappbuild folder structure overview*

- `build-conf` contains the following enterprise-level property files:
  - `build.properties` defines DBB initialization properties, including location and of the DBB metadata store (for storing dependency information) and more.
  - `dataset.properties` describes system datasets such as the PDS name of the COBOL compiler or libraries used for the subsystem. You must update this properties file with your site's data set names.
  - Several language-specific property files that define the compiler or link-editor/binder program names, system libraries, and general system-level properties for COBOL, Assembler, and other languages.
- `languages` contains Groovy scripts used to build programs. For example, `Cobol.groovy` is called by `build.groovy` to compile the COBOL source codes. The application source code is mapped by its file extension to the language script in `application-conf/file.properties`.
- `samples` contains an `application-conf` template folder and a reference sample application, `MortgageApplication`.
- `utilities` contains helper scripts used by `build.groovy` and other scripts to calculate the build list.
- `build.groovy` is the main build script of zAppBuild. It takes several required command line parameters to customize the build process.

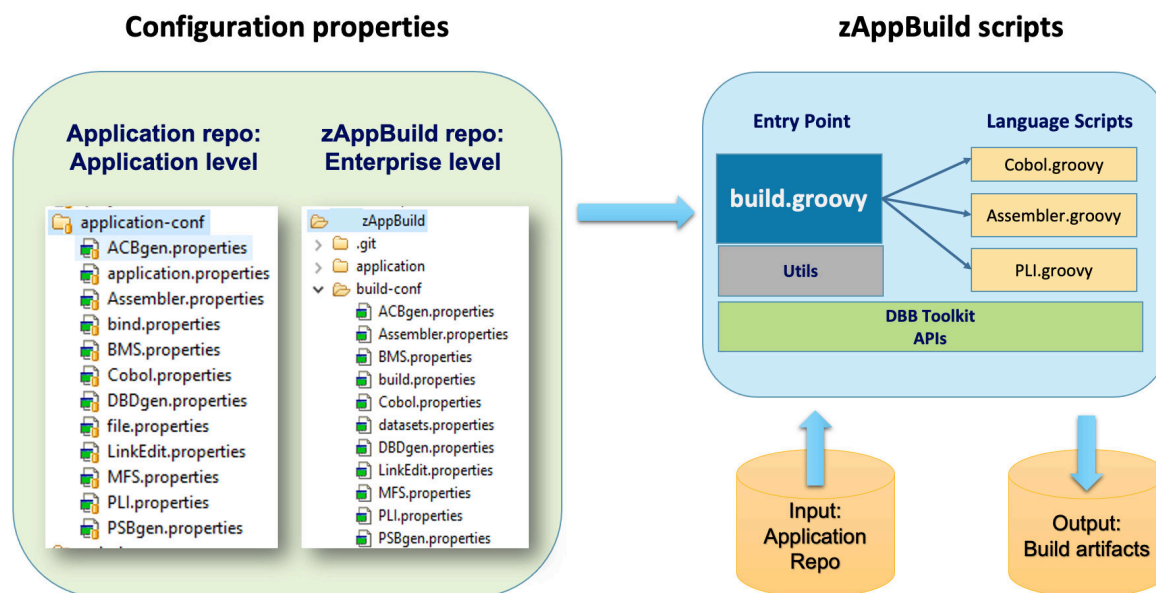
## application-conf overview

This folder is located within the application's repository, and defines application-level properties such as the following:

- `application.properties` defines various directory rules, default Git branch, impact resolution rules such as the copybook lookup rules, and more.
- `file.properties` maps files to the language scripts in `dbb-zappbuild/languages`, and provides file-level property overrides.
- Property files for further customization of the language script processing. For example, `Cobol.properties` is one of the language properties files to define compiler and link-edit options, among other properties.

The following diagram illustrates how zAppBuild's application- and enterprise-level configurations feed into its build scripts to generate build artifacts from an application repository:

## zAppBuild Architecture



## Resources

- [IBM documentation for DBB](#)
- [IBM Dependency Based Build Fundamentals course](#)

This page contains reformatted excerpts from the following documents:

- [DBB zAppBuild Introduction and Custom Version Maintenance Strategy](#)
- [Packaging and Deployment Strategies in an Open and Modern CI/CD Pipeline focusing on Mainframe Software Development](#)

## Artifact repository

Once building occurs, the pipeline then publishes and stores the build outputs as a package in the artifact repository. This package contains any artifact that will need to be deployed, such as load modules, DBRMs, DDL, and the configuration files for the subsystems. Importantly, the package also contains the build artifacts' metadata and other necessary pieces of information that enable any changes to be traced back to the version control system. Depending on the system, the package can be a WAR, EAR files, a Windows installer package, among others. The artifact repository can also be used as the publishing platform to store intermediate files needed in the build phase.

The artifact repository contains a complete history of packages, and therefore also provides access to older versions. This feature is especially important in cases where a rollback or audit is required. The artifact repository is meant to be the single point of truth for binaries, much in the same way that a SCM is the single point of truth for source files.

It is expected that a package will be deployed to several execution environments, each of them being used for different testing phases. Ultimately, some packages will be deployed to production. In this arrangement, the artifact repository acts like a proxy for the deployment manager, which is responsible for deploying the artifacts produced by the build system to one or more runtime environments.

The key mission and benefit of an artifact repository is to decouple source code management (SCM) configurations from runtime environments. This supports the fundamental DevOps principle of "build once, deploy many". Once you build and test a set of binaries to verify it, then that is the same set of binaries that you will want to deploy to the production environment. By ensuring you can use the same set of executables between your deployment environments, from testing to production, you not only reduce the risk of build time issues going undetected into your production environments, but it also becomes much easier to determine if a deployment problem is the result of a build time issue or a runtime environment issue.

## Common artifact repository options

- JFrog Artifactory
- Sonatype Nexus
- UrbanCode® Deploy (UCD) Codestation
- Azure Artifacts

## Resources

This page contains reformatted excerpts from [Packaging and Deployment Strategies in an Open and Modern CI/CD Pipeline focusing on Mainframe Software Development](#).

## Deployment manager

The deployment manager is responsible for understanding the execution environments and maintains an inventory of the environment's deployed content. It is used to rollout application packages. For many runtimes, copying artifacts is not enough to actually make them executable. There are numerous installation steps to perform. A good example of this would be a CICS® NEWCOPY/PHASE-IN, or, when Db2® is involved, a bind against the database of the environment.

## Common options

- UrbanCode® Deploy (UCD)
- Wazi Deploy (Python or Ansible®)
- Ansible z/OS® modules

## Resources

This page contains reformatted excerpts from [Packaging and Deployment Strategies in an Open and Modern CI/CD Pipeline focusing on Mainframe Software Development](#).

## Pipeline orchestrator

Also known as the CI (Continuous Integration) Orchestrator - This is where automation happens. The CI Orchestrator provides connectors to version control, build systems, and packaging and deployment. Its goal is to remove manual and repetitive tasks as much as possible. It also drives the building of the application package, includes automated unit tests, and publishes the results in an artifact repository to make them available to the provisioning and deployment practices.

## **Common pipeline orchestrator options**

- GitLab CI
- Jenkins
- GitHub Actions
- Azure Pipelines

## **Resources**

This page contains reformatted excerpts from [Packaging and Deployment Strategies in an Open and Modern CI/CD Pipeline focusing on Mainframe Software Development](#).





---

# Chapter 2. Getting started

## Understanding the journey to a CI/CD pipeline

---

What is involved in the DevOps adoption effort from green screen to CI/CD?

- Have the right tools (CI/CD pipeline tools): The major components are summarized in the [CI/CD for z/OS® applications page](#).
- Have the right people (Roles): The DevOps transformation journey includes a range of roles to help accomplish the various milestones along the way. For example, the enterprise will need roles to architect the CI/CD pipeline and workflows, implement the infrastructure, understand how to use the CI/CD tools, and so on. You can read more about each role in the [Roles](#) section.
  - IBM® and business partners can also help support you through various milestones of your journey to the CI/CD pipeline. Available resources are described for each milestone in the [Transformation journey milestones](#) section.
- Have the right mindset: Migrating from legacy tooling and development processes to DevOps and CI/CD often requires work to bring about cultural change in the organization. In fact, many successful DevOps adoption efforts involve a change transformation specialist to help the teams impacted by this initiative understand the concepts and workflows behind the DevOps change and the benefits they bring.

### DevOps transformation approaches

The adoption of a Git-based CI/CD pipeline can be approached in a couple different ways. A "big-bang" approach involves designing and implementing the future state, migrating code from z/OS to the Git provider, and training all the developers on the new way of working while the legacy system is still in place and active. Then, at a given conversion date, all developers will switch to using the new CI/CD pipeline, with the caveats that they will have received training in a short timeframe and that the pipeline will support all workflows from day one.

However, many enterprises prefer a "phased" adoption approach, with the iterative migration of applications from the legacy system to the new CI/CD pipeline. This approach begins by designing the future state, and then training and migrating just one (or a couple) complete application team(s) at a time to the new CI/CD pipeline. Additional applications are subsequently onboarded to the new CI/CD pipeline over several iterations. This iterative process allows the organization to find and iron out wrinkles in the migration process and/or pipeline on a smaller scale, and then apply the experience and improve for later iterations. By the time the enterprise is mostly using the CI/CD pipeline, the migration process will be more familiar, and the switch from the legacy system to the modern CI/CD pipeline becomes much less disruptive. Application dependencies across the new CI/CD pipeline and the legacy system are expected during the iterative adoption process, but the IBM Z® DevOps Acceleration Team (DAT) can provide resources that help you establish processes to address this period of co-existence.

### Roles

Having a team with the right skills and mindset is critical to a successful DevOps transformation effort. While the following roles each have their own specific skillsets and tasks, an individual can perform more than one role if it makes sense for their team and organization. You can click on each role to learn about it.

#### Architect

The architect helps define the new software delivery process.

Generally, the architect will be someone with strong z/OS skills who understands the infrastructure and current build processes. This deep background knowledge about the current z/OS infrastructure state and mainframe application build processes is important for understanding how to translate those processes into the more modern DevOps pipeline.

A key task is to condense existing mainframe workflows and design the to-be state in the CI/CD pipeline. For this, the architect collaborates with the other teams to create common enterprise software delivery processes. Additionally, the architect is involved in defining necessary integration points of the pipeline, as well as designing the migration process.

- Background skills and knowledge:
  - Strong z/OS skills (average of about 10 years of experience)
  - Knowledge about mainframe development processes and workflows
- Skills and concepts to learn:
  - CI/CD and DevOps principles
  - Git concepts and architecture
- Tasks:
  - Collaborate between the mainframe development team(s) and other teams to transform the existing mainframe workflows into the to-be CI/CD pipeline
- Job positions that you might find filling this role:
  - Enterprise architect
  - Enterprise application architect
  - IT architect

## **Build specialist**

The build specialist develops and maintains the build scripts for the new pipeline.

This is a developer type of role that focuses on turning the source code into a deployable artifact, so familiarity with z/OS build processes is required. The build specialist might adapt a non-mainframe example of build scripting to z/OS.

- Background skills and knowledge:
  - Mainframe build fundamentals (for example, JCL/REXX, understanding of compile/link/bind options, and so on)
- Skills and concepts to learn:
  - Git concepts
  - IBM Dependency Based Build (DBB) architecture (for example, dependency management and build results)
  - Groovy scripting
- Tasks:
  - Plan and perform migrations
  - Develop and maintain the customized build framework
- Job positions that might fill this role:
  - Build engineer
  - z/OS build administrator

## **Pipeline specialist**

The pipeline specialist assembles and administers the pipeline via the CI/CD orchestrator.

This is a developer type of role that focuses on building, scaling, and maintaining the CI/CD pipeline structure. The pipeline specialist does not need to be as z/OS-aligned as the build specialist. Rather than being concerned with building COBOL programs (or other z/OS languages), the pipeline specialist is more concerned about integrating tools together. This role often already exists elsewhere in the enterprise.

Typically, the pipeline specialist is the first role to adopt the DevOps tooling, and then teaches the tools and workflows to other teams in the organization (for example, z/OS application development teams).

- Background skills and knowledge:
  - CI/CD specialist
  - Git concepts
  - Change management integrations
- Skills and concepts to learn:
  - Foundational IBM DBB concepts
  - Groovy, Shell scripting
- Tasks:
  - Develop and maintain customized integration scripts between the different pipeline building blocks (for example, using Groovy and Shell scripting)
- Job positions that might fill this role:
  - DevOps engineer
  - DevOps pipeline administrator
  - DevOps team

## **Change transformation specialist**

The change transformation specialist drives the cultural and organizational change required for a successful modernization journey.

This role is more of a consulting and people-focused role rather than a technical one. Enterprises sometimes hire an individual specifically for this role when embarking on the DevOps transformation journey - for example, someone with specialized training in coaching DevOps/Agile methodologies, who has experience in helping teams make the transformation succeed from a cultural and cross-team point of view.

- Background skills and knowledge:
  - Strong communication
  - Planning and organizing
  - Change transformation
  - Understanding of DevOps and Agile concepts
- Skills and concepts to learn:
  - Understand the needs and concerns of all groups, in order to be the "voice of the transformation"
- Tasks:
  - Effectively communicate to teams the motivation and purpose behind the transformation journey
  - Collaborate with teams to coordinate training for the cultural and organizational change
- Job positions that might fill this role:
  - Change transformation specialist
  - DevOps/Agile coach
  - Transformation enablement team

## **Deployment specialist**

The application deployment specialist implements the deployment solution.

This developer type of role may be part of the DevOps team (with the pipeline specialist), and might already be using a deployment manager with other teams. It is helpful for them to have some

understanding of the mainframe subsystem and infrastructure interfaces, as those will also be involved in the z/OS application deployment processes.

- Background skills and knowledge:
  - Deployment management
  - Git concepts
- Skills and concepts to learn:
  - Mainframe subsystem and infrastructure interfaces
- Tasks:
  - Develop and maintain central deployment processes
  - Collaborate with the build specialist and pipeline specialist to design the to-be solution
- Job positions that might fill this role:
  - DevOps engineer
  - DevOps team

## **Integrated development environment specialist**

The integrated development environment (IDE) specialist is a developer type of role that helps implement the workflows within the IDE.

Since the IDE is a central tool used by application developers, it is important that someone in the organization is trained on how to use the IDE effectively, and can also guide others on using it. The IDE specialist understands (or learns) how to use the IDE, and shares this knowledge with others in their organization.

- Background skills and knowledge:
  - Software development tasks and use cases
- Skills and concepts to learn (if not already acquired):
  - IDE customization
  - Git concepts
- Tasks:
  - Customization and documentation of the IDE
  - IDE installation and deployment to developer workstations (including upgrades)
  - Training and coaching others on using the IDE
- Job positions that might fill this role:
  - Software developer (or application developer)
  - Software engineer

## **Middleware specialist**

The middleware specialist role is an umbrella term that covers different technical roles that help install and configure the tools for the CI/CD pipeline.

This role might be handled by more than one individual, as it can cover setup tasks on both Linux® and mainframe environments, depending on the enterprise's needs.

- Background skills and knowledge:
  - Background in managing or administering the requisite middleware system
- Skills and concepts to learn (if not already acquired):
  - Initial install and configure steps for DevOps tooling

- Tasks:
  - Assist with installation and configuration of Linux-based components and/or z/OS host components (depending on selected DevOps technology stack)
- Job positions that might fill this role:
  - Middleware system programmer or system administrator (for example, CICS® administrator and/or Db2® administrator)
  - Infrastructure team

## Migration specialist

The migration specialist is typically a transitional role that focuses on facilitating the migration from the legacy development tools and processes to the modern CI/CD pipeline.

This role can either be handled by a selected team in the enterprise, or by a business partner.

- Background skills and knowledge:
  - Mainframe data fundamentals
  - Understanding of the legacy development system
- Skills and concepts to learn (if not already acquired):
  - Git concepts
  - IBM Dependency Based Build fundamentals (for example, [DBB Migration Tool](#))
- Tasks:
  - Help move data from legacy z/OS application development systems to Git
- Job positions that might fill this role:
  - DevOps implementation architect
  - Build engineer and DevOps team

## Testing specialist

The testing specialist is technical role that focuses on quality assurance in the software.

While testing in legacy development workflows is often manual and time consuming, the move to a modernized DevOps toolchain allows the testing specialist to create tests that can be automatically run by the developer, and/or as part of a CI/CD pipeline. The scope of these tests can range from individual unit tests to larger-scale integration tests on dedicated testing platforms.

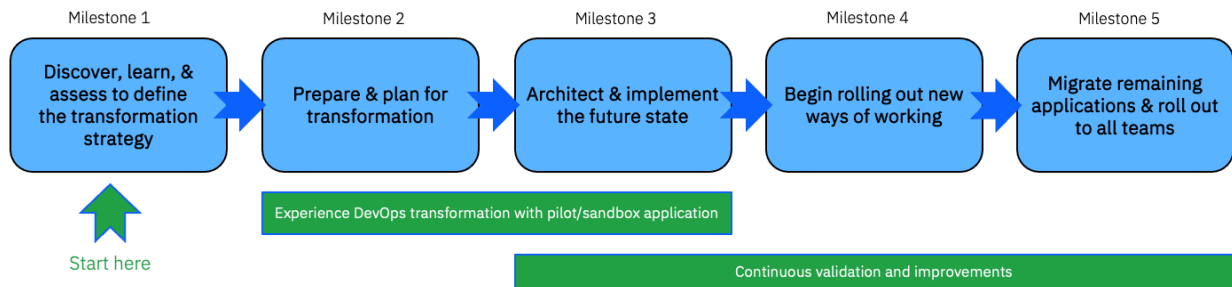
- Background skills and knowledge:
  - Understanding of the z/OS application functionality and use cases
  - Experience testing z/OS applications
- Skills and concepts to learn (if not already acquired):
  - Git concepts
  - IBM Dependency Based Build fundamentals (for example, running a DBB User Build)
  - Modern z/OS testing tools such as zUnit, IBM Virtual Dev and Test for z/OS (ZVDT), and/or IBM Z Virtual Test Platform (VTP)
- Tasks:
  - Create and automate testing processes for the CI/CD pipeline (for example, unit and/or integration testing)
- Job positions that might fill this role:
  - Quality engineer
  - Quality assurance team

- Testing team

## Transformation journey milestones

As you migrate from the green screen and adopt CI/CD, there are several key milestones along the way. The following blueprint is a high-level overview of the milestones in a CI/CD journey. If you would like a deep dive session on this blueprint, please reach out to us via the "Contact IBM" link at the bottom of this page. IBM teams and business partners are available to help you at each step of the way:

## DevOps Transformation – Blueprint



- Milestone 1: Discover, learn, & assess to define the transformation strategy
  - This is where you learn about and validate the pipeline approach. IBM conducts a no-charge Value Stream Assessment (VSA) that uses IBM Design Thinking principles in order to capture your requirements. It involves sketching out a high-level design of the to-be future state, and understanding the technical composition of the pipeline. In this milestone, building a proof-of-concept (POC) will help further validate the approach and build up skills, either to implement the pipeline with in-house resources or to be able to make informed decisions together with a services delivery organization. The POC will also help prove the benefits of the new approach and establish support from technical champions and management/leadership, which is key for a successful DevOps transformation.
  - Milestone 1 resources:
    - At the start of Milestone 1, free online resources and training provided by IBM can help you explore the opportunities to adopt DevOps for IBM Z.
    - Engage with the IBM Sales and Tech Sales teams to learn more about how adopting DevOps for IBM Z can benefit your enterprise.
    - When your team is ready to conduct a POC, the DAT can help with additional training and design sessions, and will include a primer on how to analyze your current application as well as provide enablement on the new way of working.
- Milestone 2: Prepare & plan for transformation
  - This milestone is about understanding your system's current state. Your system has evolved over time, perhaps over decades. Therefore, it requires a detailed assessment of the current build setup, the current repository layouts, and how ownership of code is defined. By assessing and understanding your system's current state, you can clarify the designs for your future state, as well as the steps for how to get there. This is also when you will want to consider the best way for your organization to adopt the new CI/CD pipeline, whether that would be to take a phased migration approach (which most customers do) or a big-bang approach.

Additional important activities in Milestone 2 include developing a migration strategy based on application types (for example, all COBOL-only applications), expanding the POC efforts with a pilot application using a subset of your own application code, and defining the target architecture and branching model that your organization will follow. The ultimate objective in this phase is to validate the business value and obtain approval and budget to proceed with the transformation project.

- Milestone 2 resources: The following IBM teams and business partners can help with analysis and refinement, leading to a proposal to move off of the legacy library manager:
  - The IBM Z DevOps Acceleration Program (which contains the DAT) can help you conduct a pilot for further evaluation of applying DevOps with a subset of your own z/OS application code.
  - IBM services such as the Software Migration Project Office (SMPO), Expert Labs, or IBM Consulting
  - IBM business partners
  - Global Systems Integrators
- Milestone 3: Architect & implement the future state
  - Milestone 3 starts with implementation (installation/configuration) of the production-level toolchain. It also gets into the details of the new working practices by refining designs and ideas from the previous milestones to create a low-level design of the future workflow. This includes planning the timing of different parts of the DevOps transformation journey to prioritize core pieces. If the design for the pipeline is in a stable state, then the necessary pipeline pieces can go forward with implementation. This is when the "heavy lifting" and "plumbing" work for the pipeline begins, integrating the different CI/CD components together in the context of your selected technologies (for example, implementation of the modern IDE, build scripts, pipeline definitions, and deployment automation).
 

In Milestone 3, it is also important to communicate the DevOps transformation journey to the intended users of the new CI/CD pipeline. This can include demos, training, and provisioning of sandbox systems where the users can explore hands-on the future state. Other important activities are identifying education requirements and defining and publishing a rollout plan.
  - Milestone 3 resources:
    - Build framework samples for zAppBuild can be found in zAppBuild's [GitHub documentation](#).
    - First-class implementation support and guidance are available from IBM services (SMPO, Expert Labs, IBM Consulting), IBM business partners, and Global Systems Integrators.
- Milestone 4: Begin rolling out new ways of working
  - In Milestone 4, the plan is put into action. This means the chosen application development and operation teams are trained and onboarded with the new way of working, and via migration plan, the core DevOps team begins to move applications of a particular type from the legacy system to the new CI/CD pipeline. This should be done in an iterative way (that is, a phased migration approach), allowing the organization to continuously incorporate feedback for improving the working practices and migration process. This is also when DevOps coaches and technology Subject Matter Experts (SMEs) can be engaged to educate the development and operations staff. By the end of this milestone, a CI/CD pipeline is in place, active, and considered to be "in production".
  - Milestone 4 resources:
    - Launch the new pipeline and make the change stick using IBM services (SMPO, Expert Labs, IBM Consulting), IBM business partners, and Global Systems Integrators.
- Milestone 5: Migrate remaining applications & roll out to all teams
  - Milestone 5 continues to educate and move the remaining application teams to the new pipeline and wraps up the migration part of the DevOps transformation journey, with teams now adopting a new way of working. If needed, Milestones 3 through 5 can be repeated for other application types (for example, all PL/I applications or all Assembler applications or another set of more complex applications). After all the application teams are moved to the new CI/CD pipeline, independent software vendor (ISV) licenses can be returned, and the organization can continue to improve developer productivity by enabling new capabilities and optimizing the DevOps workflow in the pipeline.
  - Milestone 5 resources:
    - "Rinse and repeat" by moving the remaining applications and continuously enhance the pipeline using IBM services (SMPO, Expert Labs, IBM Consulting), IBM business partners, and Global Systems Integrators.

## Installing and configuring your tools

---

This page contains installation and setup instructions for the components of our recommended CI/CD pipeline (described in our [Introduction to CI/CD for z/OS®](#)), as well as links to installation documentation for specific products. The CI/CD pipeline is based on Git for source code management (SCM), while IBM® Dependency Based Build (DBB) and zAppbuild are required for the Build component. For the remaining components of the CI/CD pipeline, enterprises will generally select one technology for each component category, although sometimes more than one IDE option is selected (depending on developer preference).

**Note:** We have listed common options for each CI/CD component below. However, if you do not see the option your enterprise has selected for a particular component, that does not necessarily mean it cannot align with our guidance. For most tools being used elsewhere, you can integrate them into the pipeline as long as they are compatible with z/OS application development and bring value to your enterprise.

Not sure what tool to pick for certain CI/CD component? See our documentation on [How do you select what tool to use for each component](#).

### Integrated development environment

Who needs to install the integrated development environment (IDE), and where?

- The integrated development environment (IDE) will need to be installed by z/OS application developers on their local workstations.
- Some IDE options have host components that require a system programmer to configure.

#### IDE options

- [IBM Developer for z/OS \(IDz\)](#): Eclipse-based IDE
  - Additional useful tools: [Groovy development environment in IDz](#)
- [Wazi for VS Code](#): Visual Studio Code IDE with IBM Z® Open Editor and Debug extensions)
- [Wazi for Dev Spaces](#): Browser-based IDE
  - Note: If you are using Wazi Dev Spaces (browser-based IDE), developers will not have to install the IDE.

### Source code management

We recommend using Git-based source code management. Clients can pick from several Git providers, including those listed in the following section for [Initializing a repository in the Git provider for mainframe application](#).

Who needs to install the source code management (SCM), and where?

- Regardless of the Git provider selected, Git will need to be installed on both developer machines, and on z/OS.
  - Developer machines: Developers can obtain Git the same way they would for non-mainframe development.
  - z/OS: The Rocket distribution of Git is specifically ported for z/OS, and can be installed by a system programmer.

Once Git is set up on the developer machines and z/OS build environments, it works together with the enterprise Git provider for the organization. The enterprise's team that handles the Git provider will usually assist to provide the basic organization or repository setup for the z/OS application team(s), and will also provide guidance on the recommended protocol for communicating with the central Git provider. The z/OS application teams will then maintain these repositories.



## Installing and configuring Git on z/OS

IBM provides an SMP/E install of Rocket Git for z/OS, which is documented at [Program Directory for Rocket Git for z/OS](#). For the latest maintenance and updates you should also review the [Fix List for Rocket Git for z/OS](#) and apply them on a regular and timely basis.

## Initializing a repository in the Git provider for mainframe applications

The following tabs describe how to initialize a repository in common Git provider options.

### *Initialize a repository in Azure Repos*

1. In Azure DevOps' web interface, follow the Azure documentation to [Create a new project in Azure](#).
2. Initialize a new repo in Azure:
  - a. From your new Project page select Repos and the "Initialize" option. Your repo is created with the name of your project with a "main" branch.
  - b. From your new Repo's page select "Clone", "SSH", and "Manage SSH Keys". This takes you to the "New Key" page where you can add your z/OS SSH public key.
    - Azure documentation on adding SSH keys can be found at [Using SSH key authentication](#).
  - c. Once the SSH key is added, go back to the repo's clone button to cut & paste the SSH-based URL for cloning on UNIX System Services. For example, a project named "Azure-Mortgage-SA" would look like `git@ssh.dev.azure.com:v3/Azure-Repo-DBB/AzDBB/`.
  - d. On UNIX System Services run `git clone` followed by the URL. Your new local repo is ready for the next step – PDS migration.

**Note:** Instead of cloning with SSH you can clone with HTTPS. For example, you can clone using a [personal access token \(PAT\)](#) that is generated from the Azure UI. Apply the PAT to the clone command, for example: `-https://<USER>:<PAT>@<my.azure.com/repo uri>`. Other alternatives are Azure's OAuth token, adding `extraHeaders` in the clone command to enable the `basicSecurity` option, Azure's `System.AccessToken` pipeline variable, or Git's built-in [Credential Manager](#).

### *Initialize a repository in GitLab*

1. Initialize a new repo in GitLab:
  - a. In GitLab's web interface, follow GitLab documentation on [creating a project](#).
  - b. Follow GitLab documentation to [clone the repository](#) you created in the previous step to UNIX System Services. If you are using SSH, you would add your z/OS SSH public key to your GitLab account.
    - GitLab provides guidance on adding SSH keys in their documentation at [Use SSH keys to communicate with GitLab](#).

### *Initialize a repository in GitHub*

1. Initialize a new repo in GitHub:
  - a. In GitHub's web interface, follow GitHub documentation on [creating a new repository](#).
  - b. Follow GitHub documentation to [clone the repository](#) you created in the previous step to UNIX System Services. If you are using SSH, you would add your z/OS SSH public key to your GitHub account.
    - GitHub documentation on adding SSH keys can be found at [Generating a new SSH key and adding it to the ssh-agent](#)

### *Initialize a repository in Bitbucket*

1. Initialize a new repo in Bitbucket:
  - a. In Bitbucket's web interface, follow Bitbucket documentation on [creating a new repository](#).

- b. Follow Bitbucket documentation to [clone the repository](#) you created in the previous step to UNIX System Services. If you are using SSH, you would add your z/OS SSH public key to your Bitbucket account.

- Bitbucket documentation on adding SSH keys can be found on their [Git SSH page](#).

To migrate a mainframe application to your newly-initialized repository, configure the DBB Migration Tool, and finalize the changes and push to Git. Information on these steps can be found on the [DBB Migration Tool page](#).

## Build

IBM Dependency Based Build (DBB) is the recommended build tool for z/OS applications. This is complemented by the zAppBuild framework, which helps facilitate your build process using DBB APIs. Many clients start by using zAppBuild and enhancing it to their needs, for example by adding new language scripts, or by modifying the existing build processing logic.

This section provides a set of instructions for how you can make zAppBuild available in your Git provider and how to synchronize new features of zAppBuild into your customized fork.

**Note:** zAppBuild releases new versions through the main branch. New contributions are added first to the develop branch, which then will be reviewed and merged to the main branch.

The [IBM DBB samples repository](#) contains additional utilities that enhance or integrate with the other DBB build processes.

Who needs to install DBB, and where?

- System programmers install DBB toolkit on z/OS.
  - Set up Db2<sup>®</sup> for z/OS or Db2 for LUW (Linux<sup>®</sup>, UNIX, and Windows) for the DBB metadata store.
  - See IBM Documentation on [Installing and configuring DBB](#).
- Developers using IDz as their IDE must add the [DBB integration](#) to their installation of IDz in order to use DBB's user build feature.

Who needs to set up zAppBuild (and the IBM DBB samples repository), and where?

- The build engineer and/or DevOps team (in [DAT roles](#): Build specialist and/or Pipeline specialist) should set this up with the enterprise's Git provider.
  - Steps for making a copy of the zAppBuild repository available in your enterprise's preferred Git provider are provided in the [following section](#).
  - If the IBM DBB samples repository is needed, it can be copied from its [IBM GitHub page](#) to your Git provider in a similar manner to the zAppBuild repository.

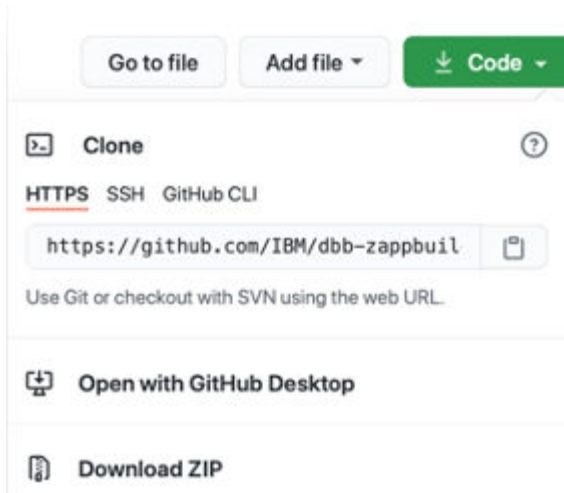
## Making zAppBuild available in your Git provider

Before you start your customization of zAppBuild, you must first create a clone of IBM's zAppBuild repository and store the clone in your Git provider of choice. This could be any Git provider, such as GitHub, GitLab, Bitbucket or Azure Repos, and so on. If you have done this already, feel free to move to the next section.

Here are the steps to make the zAppBuild repository available in a central repository on your Git provider:

1. On your local workstation, use your browser and log on to your Git provider. Follow the instructions in your Git provider to create a new repository, which will be the new “home” of your customized version of zAppBuild.
  - a. We suggest "dbb-zappbuild" as the new repository/project name, but you can use another name if you prefer.
  - b. Set the repository's visibility according to your needs.
  - c. Do not initialize the repository yet.

- Your Git provider will create the repository, but it is not yet initialized. On most Git providers, the repository creation process will end on a page with information on how to share an existing Git repository. Leave the browser open.
2. Clone IBM's public [zAppBuild](#) repository to your local workstation. You can use your local workstation's terminal to complete this step (for example, Git Bash in Windows, or Terminal on MacOS).
    - If you are using IBM Developer for z/OS (IDz) as your IDE, you can use its [Local Shell](#). Wazi for VS Code and Wazi for Dev Spaces also both have Terminal windows. (We documented the steps in this guide using a terminal.)
      - a. In the terminal, navigate to the folder where you would like to clone the repository.
      - b. Retrieve the Git repository URL or SSH path from IBM's public [zAppBuild](#) repository:



- c. In your terminal, enter the command for cloning the repository. (The following command uses the Git repository URL, but the SSH path can also be used if you have SSH keys set up.):

```
git clone https://github.com/IBM/dbb-zappbuild.git
```

- Example Git clone command with output in a terminal:

```
dennisbehm@Denniss-MacBook-Pro ~/git/buildframework ▶ git clone https://github.com/IBM/dbb-zappbuild.git
Cloning into 'dbb-zappbuild'...
remote: Enumerating objects: 604, done.
remote: Total 604 (delta 0), reused 0 (delta 0), pack-reused 604
Receiving objects: 100% (604/604), 217.50 KiB | 898.00 KiB/s, done.
Resolving deltas: 100% (395/395), done.
```

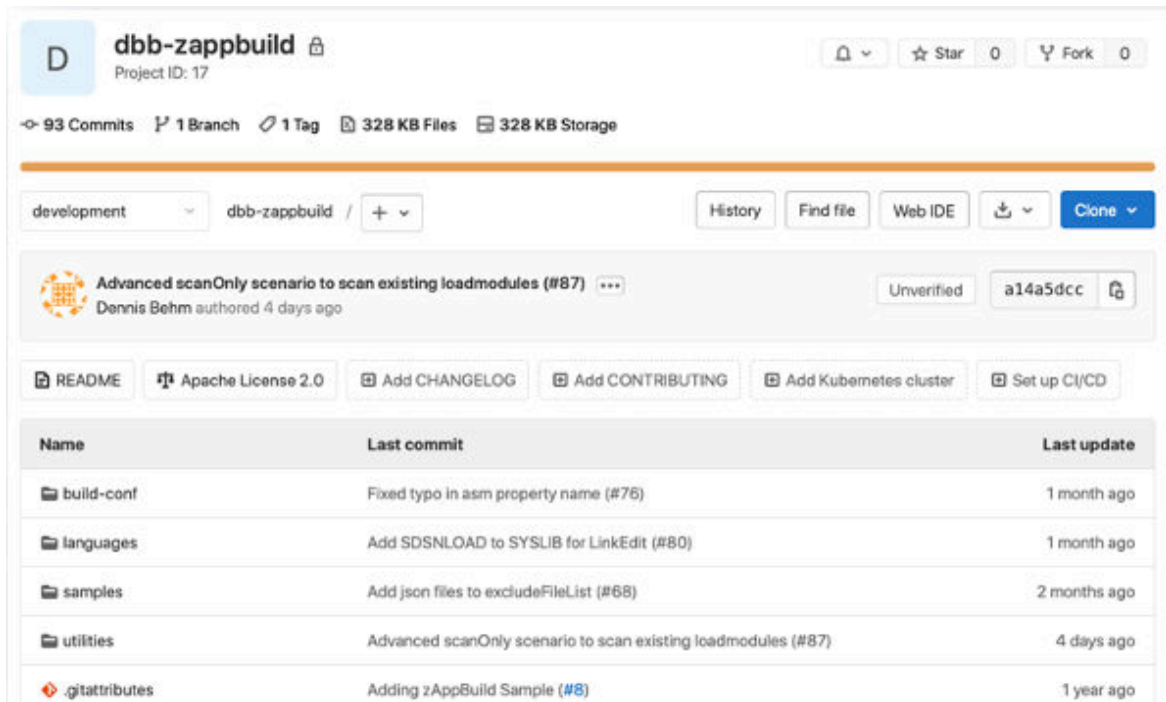
3. Follow the instructions of your Git provider to push the contents of your newly-cloned local repository (from Step 2) to the central repository on your Git provider (created in Step 1). (Note: Exact instructions may vary from Git provider to Git provider.)

- a. Within the terminal session, execute the following commands to push an existing Git repository:

- Replace `<existing_repo>` with the path to your newly-cloned local repository.
- Replace `<Your central Git repository>` with the URL to the new central repository on your Git provider. (For example, with GitLab as the Git provider, the URL might look similar to `git@gitlab.dat.ibm.com:DAT/dbb-zappbuild.git`.)

```
cd <existing_repo>
git remote rename origin old-origin
git remote add origin <Your central Git repository>
git push -u origin -all
git push -u origin --tags
```

- b. On the Git provider's webpage for your new central repository in the browser, you will find that the repository is now populated with all of zAppBuild's files and history, just like on IBM's public [zAppBuild repository](#).
- The following screenshot shows an example of a populated central zAppBuild repository with GitLab as the Git provider:



## Updating your customized version of zAppBuild

To update your customized version of zAppBuild with latest official zAppBuild enhancements, you can integrate the latest official zAppBuild features into your version of zAppBuild. So, let's get started.

1. Locate the internal Git repository and create a new Git branch. This is a good practice to validate the changes first. In this example, the new branch is called update-zappbuild.
2. Add a new Git remote definition to connect to IBM's official public zAppBuild GitHub repository. (Note: This step requires internet connectivity.)
  - a. First, list the remotes by issuing `git remote -v`:

```
dennisbehm@Denniss-MacBook-Pro ~/git/gitlab/dbb-zappbuild 35-update-zappbuild git remote -v
origin http://gitlab.dat.ibm.com/dat/dbb-zappbuild.git (fetch)
origin http://gitlab.dat.ibm.com/dat/dbb-zappbuild.git (push)
```

- For more on Git remotes, see the [git-remote documentation](#).
- b. Add a new remote named zappbuild-official to connect to GitHub by issuing the following command:

```
git remote add zappbuild-official https://github.com/IBM/dbb-zappbuild.git
```

- c. Verify that the new remote is available by issuing the command to list the remotes again: `git remote -v`:

```
dennisbehm@Denniss-MacBook-Pro ~/git/gitlab/dbb-zappbuild 35-update-zappbuild git remote -v
origin http://gitlab.dat.ibm.com/dat/dbb-zappbuild.git (fetch)
origin http://gitlab.dat.ibm.com/dat/dbb-zappbuild.git (push)
zappbuild-official https://github.com/IBM/dbb-zappbuild.git (fetch)
zappbuild-official https://github.com/IBM/dbb-zappbuild.git (push)
```

- d. Fetch the latest information from the official repository, by executing a Git fetch for the official dbb-zappbuild repository:

```
git fetch zappbuild-official
```

```
dennisbehm@Denniss-MacBook-Pro ~/git/gitlab/dbb-zappbuild 35-update-zappbuild git fetch zappbuild-official
remote: Enumerating objects: 33, done.
remote: Counting objects: 100% (33/33), done.
remote: Total 51 (delta 33), reused 33 (delta 33), pack-reused 18
Unpacking objects: 100% (51/51), 21.58 KiB | 223.00 KiB/s, done.
From https://github.com/IBM/dbb-zappbuild
* [new branch]      development -> zappbuild-official/development
* [new branch]      master -> zappbuild-official/master
* [new tag]         2.0.0 -> 2.0.0
```

- e. Make sure that your feature branch is checked out, before attempting to merge the changes from zappbuild-official. To merge the changes run into your branch update-zappbuild, run the following command:

```
git merge zappbuild-official/main
```

```
dennisbehm@Denniss-MacBook-Pro ~/git/gitlab/dbb-zappbuild 35-update-zappbuild git merge zappbuild-official/development
Auto-merging utilities/ImpactUtilities.groovy
CONFLICT (content): Merge conflict in utilities/ImpactUtilities.groovy
Auto-merging utilities/GitUtilities.groovy
Auto-merging utilities/BuildUtilities.groovy
Auto-merging samples/MortgageApplication/application-conf/application.properties
Automatic merge failed; fix conflicts and then commit the result.
```

Potentially, you face merge conflicts. In the above case, the merge processor could not automatically resolve the `utilities/ImpactUtilities.groovy`.

Run the command `git status` to see which files changed:

```
dennisbehm@Denniss-MacBook-Pro ~/git/gitlab/dbb-zappbuild 35-update-zappbuild | merge git status
On branch 35-update-zappbuild
Your branch is up to date with 'origin/35-update-zappbuild'.

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Changes to be committed:
  modified:   BUILD.md
  modified:   README.md
  modified:   build-conf/Assembler.properties
  modified:   build.groovy
  modified:   languages/Assembler.groovy
  modified:   languages/BMS.groovy
  modified:   languages/Cobol.groovy
  modified:   languages/PLI.groovy
  modified:   samples/MortgageApplication/application-conf/application.properties
  modified:   samples/application-conf/application.properties
  modified:   utilities/BuildUtilities.groovy
  modified:   utilities/GitUtilities.groovy

Unmerged paths:
  (use "git add <file>..." to mark resolution)
  both modified:   utilities/ImpactUtilities.groovy
```

- f. Open the unmerged files and resolve them manually. Either use the terminal, or an IDE for this task.

Tip:

The Git integration in many modern IDEs (for example, VS Code) is able to provide a side-by-side comparison highlighting the diff between your feature branch and the incoming changes from the merge attempt (in this case, from zappbuild-official). This can help make manual resolution of any merge conflicts much easier.

- g. Commit the changes and verify them with a sample application before committing it (or opening a pull request to commit it) to your main branch that is used for all your production DBB builds.

## Artifact repository

The artifact repository is often already in-place as part of the enterprise's non-mainframe CI/CD pipeline.

Who needs to set up the artifact repository, and where?

- Generally, the DevOps team (pipeline specialist) will work to set this up for z/OS application teams, as well.

### Artifact repository options

- Azure Artifacts
- JFrog Artifactory
- Sonatype Nexus
- UrbanCode® Deploy (UCD) CodeStation

## Deployment manager

Who needs to install the deployment manager, and where?

- Depending on the software selected, the deployment manager might require an agent on the z/OS side, which can be set up by a system programmer (infrastructure team). (Alternatively, the pipeline orchestrator could SSH into z/OS.)

### Deployment manager options

- UrbanCode Deploy (UCD)
- Wazi Deploy

## Pipeline orchestrator

The pipeline orchestrator is often already in-place as part of the enterprise's non-mainframe CI/CD pipeline.

Who needs to set up the pipeline orchestrator, and where?

- Generally, the DevOps team (pipeline specialist) will work to set this up for z/OS application teams, as well.
- The pipeline orchestrator often requires an agent or runner on the z/OS side, which can be set up by a system programmer (infrastructure team). (Alternatively, the pipeline orchestrator could SSH into z/OS.)

### Pipeline orchestrator options

- Azure Pipeline
- GitHub Actions
- GitLab CI
- Jenkins

## Resources

This page contains reformatted excerpts from the following documents:

- DBB zAppBuild Introduction and Custom Version Maintenance Strategy





---

# Chapter 3. Working practices

## Introduction to working practices for teams using Git

---

How a team employs Git's fundamental feature to create branches, commit changes to them, and flow sets of changes for review and inclusion into deployed systems determines how they gain the advantages of applying DevOps to IBM Z®.

Git branching models are patterns to enable development teams using Git to manage their code in a streamlined manner. Since Git is established as a de facto standard for source code management (SCM) in the developer community, several approaches were designed to fulfill developers' requirements and manage the application source code lifecycle, with advantages and drawbacks depending on use cases. Based on the experience gained designing branching strategies, the pages in this section describe a blueprint implementation of a trunk-based development approach for mainframe applications using feature branches with an early integration pattern. This setup uses a standardized development toolset based on an enterprise-wide Git provider and a continuous integration/continuous delivery (CI/CD) toolchain.

A characteristic of this integration pattern is that developers are implementing changes for a planned release and integrate their changes into a common permanent branch (the shared configuration) that is built, tested, and released together as one consistent entity.

The purpose of streamlining both the working practices 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 Git branching model for mainframe development](#) introduces the branching model and outlines the development workflow from the developer's perspective. The details of the technical implementation with IBM Dependency Based Build (DBB) and zAppBuild, as well as packaging and deployment, are discussed in [Implementing a pipeline for the branching model](#). All branching models are adaptable to the needs of specific teams and their applications. Our branching approach advocates for best practices and indicates where variations can be applied.

The target audience of this branching model documentation is mainframe DevOps architects and SCM specialists interested in learning how to design and implement a CI/CD pipeline with a robust and state-of-the-art development workflow.

### Aims and Assumptions

Some aims and assumptions that guide our recommendations include:

- The workflow and branching scheme should both scale up and scale down.
  - Small teams with simple and infrequent changes will be able to easily understand, adopt, and have a good experience.
  - Large, busy teams with many concurrent activities will be able to plan, track, and execute with maximum agility using the same fundamental principles.
- Planning and design activities as well as code development aim to align to a regular release cadence.
- There is no magic answer to managing large numbers of "in-flight" changes, so planning assumptions should aim as much as possible to complete changes quickly, ideally within one release cycle.

#### Tip

DevOps/Agile practices typically encourage that, where possible, development teams should strive to break down larger changes into sets of smaller, incremental deliverables that can each be completed within an iteration. This reduces the number of "in-flight" changes, and allows the team to deliver value (end-to-end functionality) more quickly while still building towards a larger development goal.

- We know it is sometimes unavoidable for work to take longer than one release cycle and we accommodate that as a variant of the base workflow.

## Choosing a workflow and branching model

Your choice of workflow and the branching model that supports it need to take into account your team's needs and characteristics.

Aspects to consider include:

- Size of the team
- Frequency of change
- Granularity of change
- Amount of parallel development
- Formality of release process

The workflows of our recommended [Git branching model for mainframe development](#) are flexible enough to scale from small teams with an infrequent pattern of small changes, to large and busier teams with many concurrent projects and projects spanning multiple cycles of a formal release cadence. These workflows are supported by the CI/CD pipeline implementation described in [Implementing a pipeline for the branching model](#).

## The Git branching model for mainframe development

---

As Git became 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 page to designate the operation of reviewing and merging one branch into another.

Many mainframe development teams follow a release-based or iteration-based process to deliver incremental updates to a pre-defined production runtime.

## Characteristics of trunk-based development with feature branches

As mentioned in the [Introduction to working practices for teams using Git](#), our recommended approach scales very well to support the needs of a range of team sizes, frequency and size of changes, and degrees of concurrent working.

### Starting simple

The trunk-based development approach with short-lived feature branches is a simple and structured workflow to implement, integrate, and deliver changes with an early integration process flow using a single long-living branch: `main`. Developers work in isolation in feature branches to implement changes to the source code, and ideally test the changes in a specific environment. Each feature branch (sometimes referred to as a "[topic branch](#)") is dedicated to a specific developer task such as a feature or bug fix.

A similar workflow is also documented by Microsoft without giving it a name.

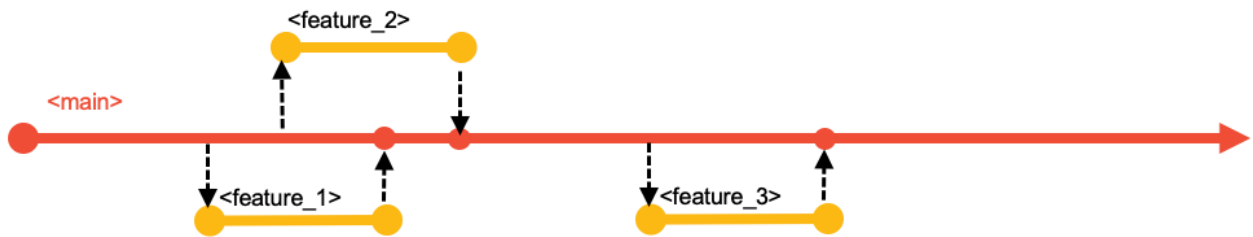
The `main` branch is the point of reference for the entire history of the mainline changes to the code base, and should be a protected branch. All changes should originate on a separate branch created to hold

---

<sup>1</sup> Trunk-based development approach: <https://trunkbaseddevelopment.com/#scaled-trunk-based-development>

<sup>2</sup> Git branching model documented by Microsoft: <https://learn.microsoft.com/en-us/azure/devops/repos/git/git-branching-guidance?view=azure-devops>

them until they are ready to be merged. Using a branch-and-merge approach is natural in a Git-based development workflow, and is very lightweight both in terms of resource consumption and developer experience.



### **All changes start on a dedicated branch**

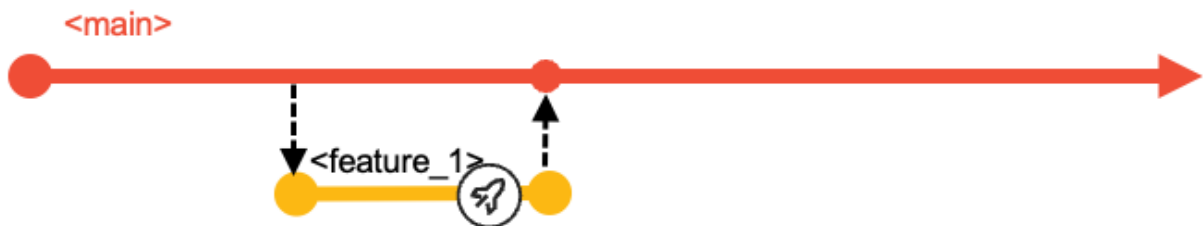
Although all Git-based DevOps services provide direct access to the server copy of the code repository, developers will typically use an integrated development environment (IDE) of their choice to work locally on a clone of the repository. Standard practices ensure developers' local clones are synchronized with the server copy (typically known as the "remote") through actions known as "pull" and "push". (For those unfamiliar with Git terminology, these terms are explained in finer detail in [Source code management](#).)

In a small team where there is almost never more than one change in progress at a time, using a branch enables the developer to make the changes in a series of commits rather than as one atomic change as they edit source and save files. Of course, the "one-line" change might mean there genuinely is only one commit that needs merging, but the process is so natural and light-weight that it is not worth making that a special case.

Such branches are able to be built prior to merging - this can eliminate the possibility of the merge breaking the build of `main`, thus reducing the risk in making changes.



Build pipeline



### **Merging a branch**

A branch holds all the commits for a change - be that a single commit for a one-liner or a sequence of commits as the developer refined the change while making it ready for review and merging into `main`.

The request to merge a branch is made explicitly, but can be as formal or informal as needed by the team. Protection of `main` can mean that only certain people can perform the merge, or that a review and approval of the change is required before merging it, or both.

The action of merging can either simply take all the commits from the branch and add them to main, or that multiple commits in the branch can be "squashed" into one commit - the latter can keep the overall history on main "cleaner" if that's important to the team.

main should always build successfully, enabling the team to choose when to package and deploy.

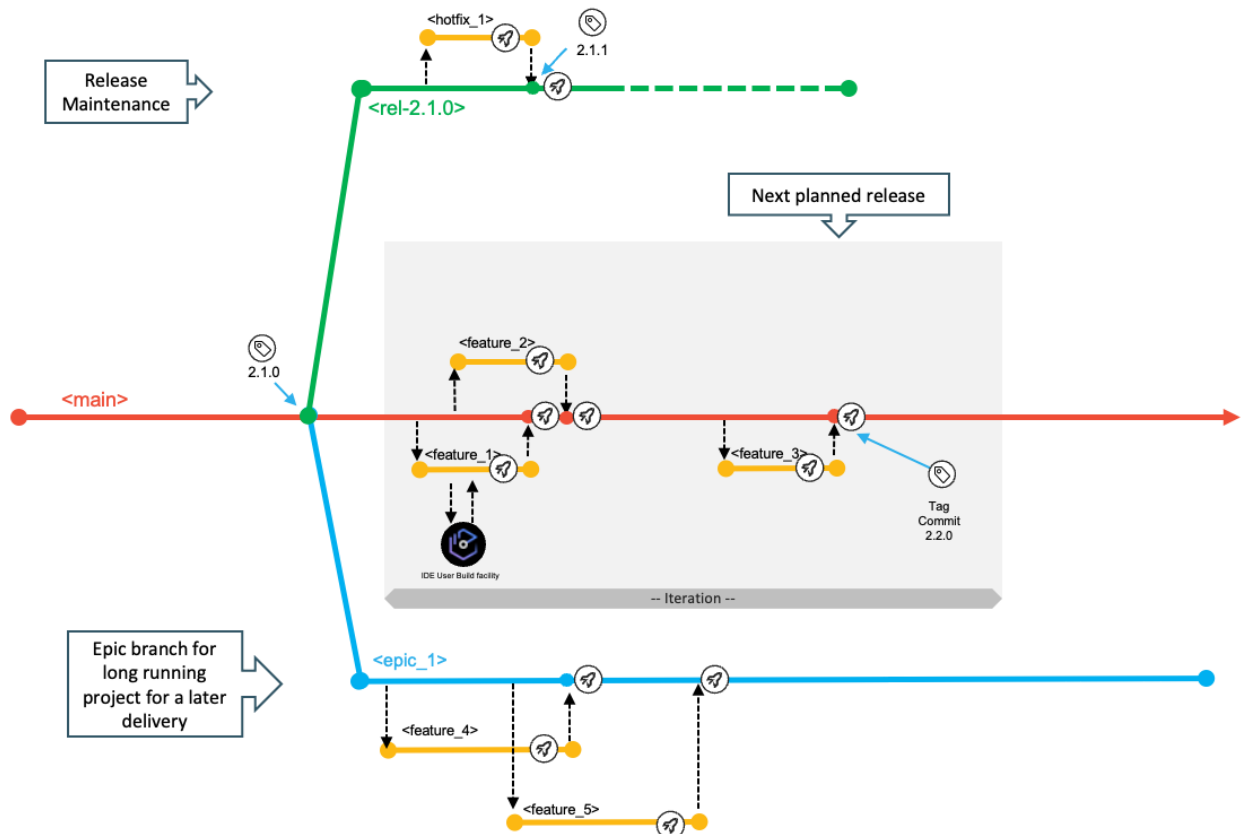
## Scaling up

The use of branches for concurrently planned activities scales extremely well for busier teams. Additionally, epic and release maintenance branches accommodate specific development workflows and allow the model to scale even further. The latter two branches exist for the duration of the epic or release maintenance and are short-living branches.

The implemented changes of the iteration are then delivered collectively as part of the next release. Each development team decides how long an iteration is. We advocate for working towards smaller, quicker release cycles, but this model can also be used with longer iterations. Due to business or technical reasons, the merging of features into the main branch can also be delayed. Although this scenario is a discouraged practice, the recommendation in this case is to group such features into a specific epic branch, as described later.

This branching model uses Git tags to identify the various configurations/versions of the application, such as a release candidate or the version of the application repository that is deployed to production.

Depending on the type of change, the development workflow can vary. In the standard scenario, developers use the main branch to deliver changes for the next planned release, while the release maintenance branches allow fixing of the current release running in the production runtime environment(s). Using epic branches is optional for development teams, but is a grouping mechanism for multiple features that should be built and tested together, thus allowing teams to increase the concurrency of working on multiple, larger development initiatives of the application. The epic branch also represents a way to manage the lifecycle of features that are not planned for the next planned release. In this way, it is a vehicle to delay merging the set of features into the main branch for a later time.



The `main`, `epic`, and `release` branches are assumed to be protected branches, meaning that no developer can directly push changes to these configurations. It requires developers to make changes on a feature branch and go through the pull request process. Before merging the feature branch into a shared branch (whether it is the `main` branch or an `epic` 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 target branch. In a continuous integration workflow, integrations are expected to happen early to avoid delaying merge conflicts or merges leading to an unstable build.

Notice that `main` is the only long-running branch. In particular, there are no branches aligned to environments which may be deployed to - such as `prod`, `production`, `QA` or `Test` branches. We would expect builds to be deployed to environments known by similar names, but there is no compelling reason to have such branches. The build and deployment tools ensure clear traceability from the point in the history of `main` (or a feature/epic branch before merging) a package was produced. The deployment manager installs the package into the various testing environments and, upon successful completion of testing and sign-off, into the production environment as a release. The deployment manager maintains the inventories of the deployed packages for the various runtime environments.

Additional long-lived branches will each represent alternative *histories* and give rise to possible ambiguity as sequences of commits that will need to be merged to multiple branches rather than just to `main`.

The single consolidated history on `main` serializes the commits of merged feature or fix branches - and then is punctuated with explicit release tags and/or release branches. Since a branch can be easily created from any previous commit even if a release candidate build needs a specific fix this can be achieved whenever it is needed.

## Naming conventions

Consistent branch naming conventions help indicate the context for the work that is performed. Throughout this document, the following naming patterns are used:

- `main`: The only long-living branch which is the only branch from which every release is initially derived
- `release/rel-2.0.1`: The release maintenance branch for an example release named `rel-2.0.1`
- `epic/ai-fraud-detection`: An epic branch where "aiFraudDetection" is describing the initiative context (in this example, an initiative to adopt AI technology for fraud detection)

Feature branches also need to relate back to the change request (or issue) from the planning phase and their context. Some examples are shown in the following list:

- `feature/42-new-mortgage-calculation` for a planned feature for the next planned release.
- `hotfix/rel-2.0.1/52-fix-mortgage-calculation` for a fix of the current production version that is running the `rel-2.0.1` release.
- `feature/ai-fraud-detection/54-introduce-ai-model-to-mortgage-calculation` for a contribution to the development initiative for adopting AI technology for fraud detection.

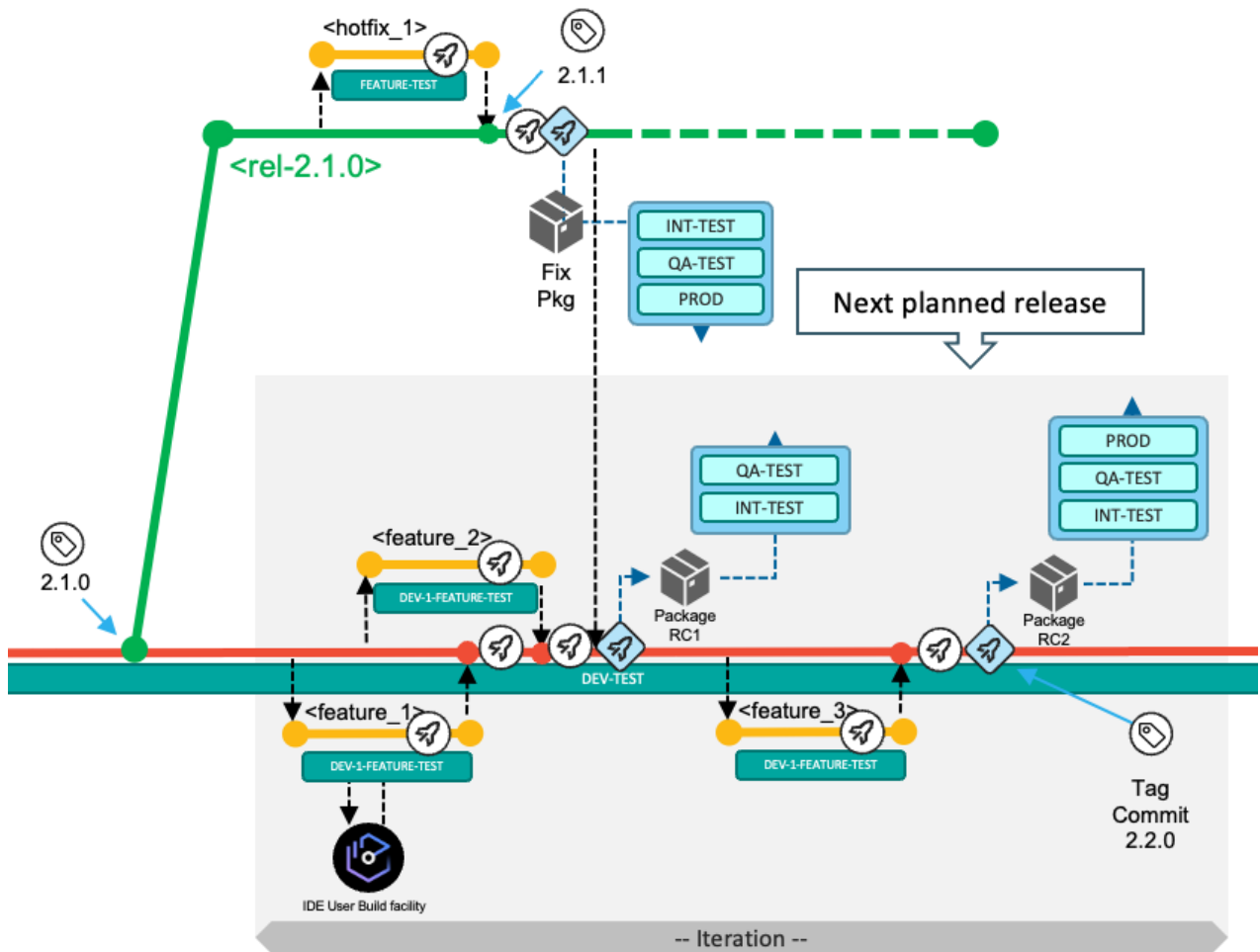
The above naming conventions help to easily identify the category of the branch you are looking at. The last segment in feature or hotfix branch names may include a prefix of the work item identifier.

**Note:** In the diagrams visualizing the workflows, the branch names are abbreviated to their contexts for readability. For example, for the feature branch `feature/42-new-mortgage-calculation`, the diagrams will show `feature_1`.

## Integration branches

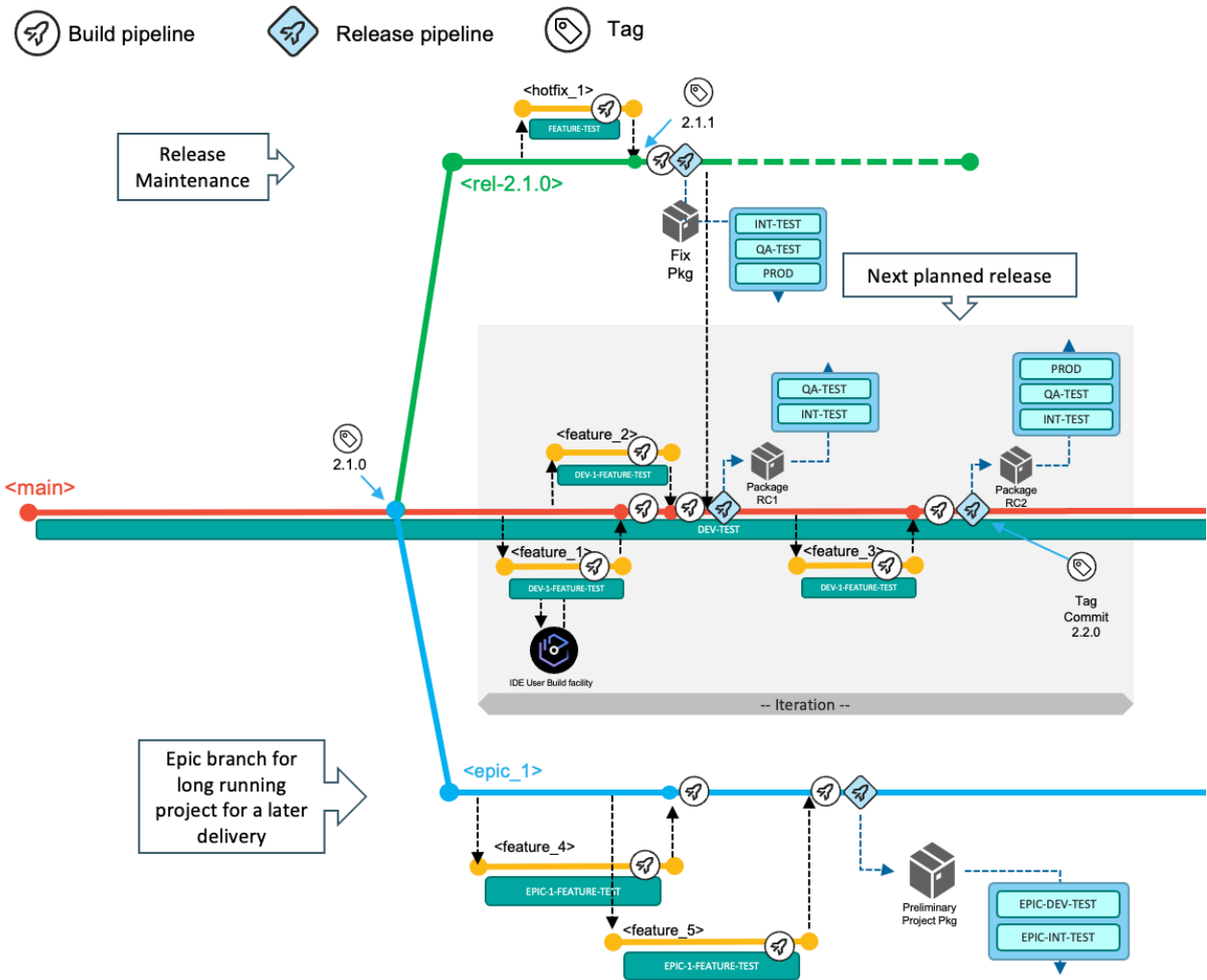
Specific branches, such as `main`, `epic`, and `release` branches can be seen as integration branches, because their purpose is to integrate changes from other branches (typically feature branches). To drive the integration process of changes into a shared branch of code, mechanisms like pull requests are a convenient way as they guide the developers with a streamlined workflow. The number of integration branches required for your development process depends on the needs of the application team. However, while the cost of creating new branches is low, keeping them up-to-date, for instance by integrating release bugfixes from the stabilization phase into concurrent epic branches, can be expensive.

For application teams who want to embrace an agile development methodology and who sequentially deliver new releases with limited parallel development initiatives, they can use the main branch and, optionally, the release maintenance branch as integration branches to implement the next planned release and potential bug fixes. The following diagram illustrates a branching model for a Git-based development process with sequential release deliveries.



A common, recommended practice is to squash the different commits created on the feature branch into a single new commit when merging, which keeps the Git history from becoming cluttered with intermediate work for the feature. This also helps to maintain a tidy history on the main branch with only the important commits.

If the development teams need to work on a significant development initiative in parallel to the standard scenario of the next planned release, this model allows isolation using the epic branch workflow. The epic branch (epic\_1) in the following branching diagram represents an additional integration branch that is used to implement, build, and test multiple features that are planned for the development initiative and can be merged into the main branch at a later time. The team decides which commit/tag of the codebase in the main branch will be used as the base for the epic branch, although it is recommended to start from the last tag for the main branch.



When the work items implemented on the epic branch are planned and ready to be delivered as part of the next planned release, the development team merges the epic branch into the main branch.

Epic branches can be used to compose various styles of development processes. The documentation for [Working practice variations](#) provides additional samples.

## Workflows in this branching model

This branching model facilitates three different types of development workflows:

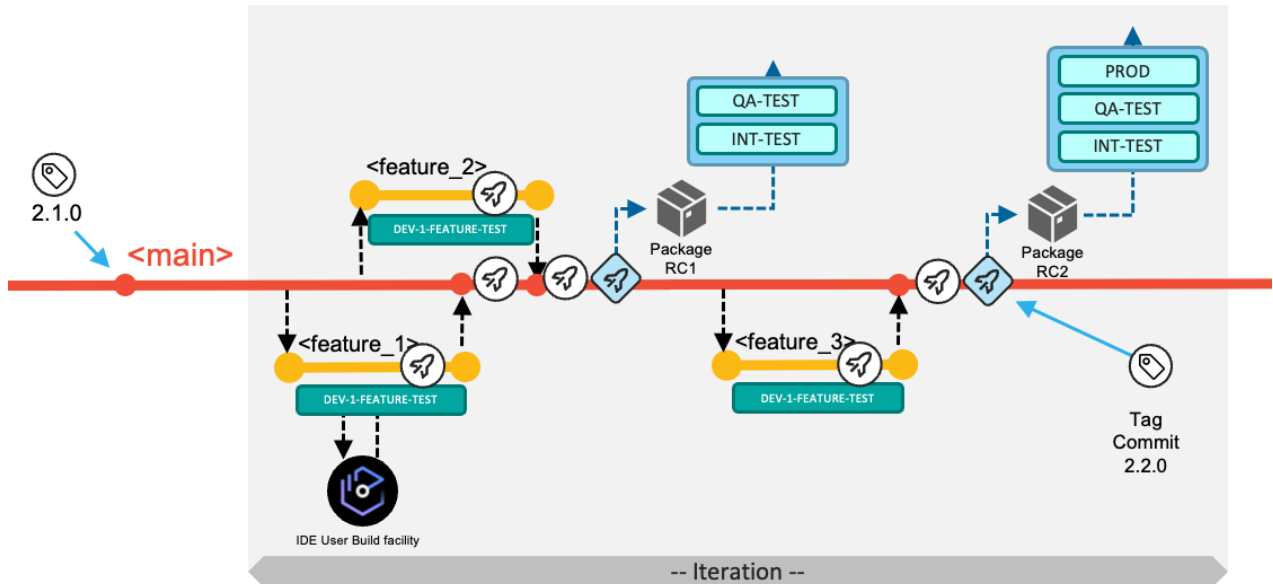
1. (Default development workflow) Deliver changes with the next planned release: With a single long-living branch, the development process allows developers to work and focus on the next planned release. After planning the work items for the next release, the development team is adding changes to the main branch by merging in pull requests for feature branches.
2. Implement a fix for the current production state: This workflow enables development teams to resolve a production problem in the currently-released version of the application by leveraging a release branch that is used for maintenance purposes.
3. Use an epic branch for a significant development initiative: Concurrent development activities for a significant solution development initiative that includes multiple planned work items for a later delivery (which could even be starting the development of a future release) are supported by creating an epic branch from a commit point in the history of main.

Git tags are used throughout this process to indicate and label important commits, such as the commit of a release that is built from the main branch, or a maintenance release created from a release maintenance branch.

The next sections outline the various tasks and activities performed by the development team in the context of the above three scenarios.

## Deliver changes with the next planned release

The following diagram depicts the typical workflow to deliver changes for the next planned release. In the default workflow, the development team commits changes to the head of the main branch. The changes of the next planned release are built, packaged, and released from the main branch.



Developers implement their changes by committing to short-living feature branches (visualized in yellow), and integrate those via pull requests into the long-living main branch (visualized in red), which is configured to be a protected branch.

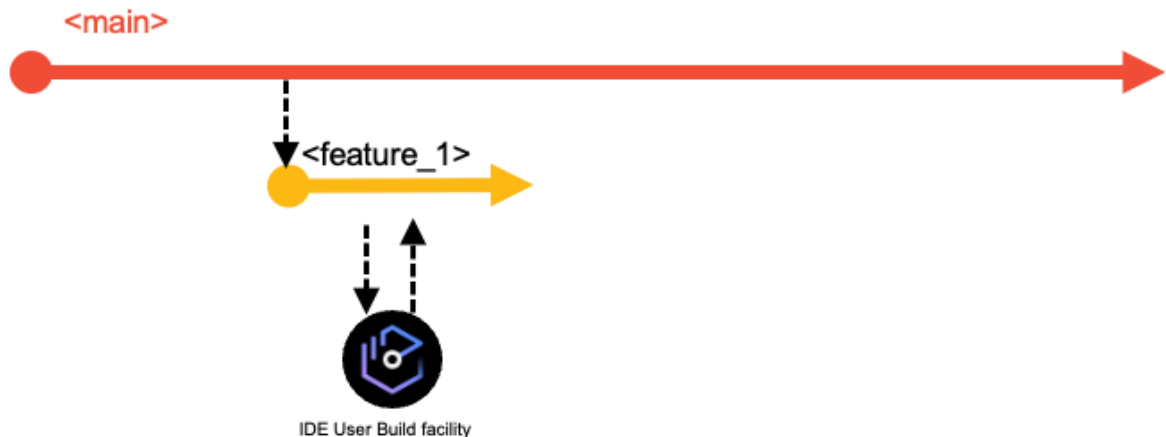
At a high level, the development team works through the following tasks:

1. New work items are managed in the backlog. The team decides which work items will be implemented in the next iteration. Each application team can decide on the duration of the iteration (which can also be seen as the development cycle). In the above diagram, three work items (Feature 1, Feature 2, and Feature 3) were selected to be implemented for the next iteration. The development team is responsible for coordinating if features are required to be implemented in a specific order.
2. For each work item, a feature branch is created according to pre-defined naming conventions, such as `feature/42-new-mortgage-calculation`. The feature branch allows the assigned developers to work on their changes in isolation from other concurrent development activities.
  - **Note:** In the diagrams for this workflow, multi-segmented branch names are abbreviated to their contexts in the diagrams for readability (for example, `feature_1` or `hotfix_1`).





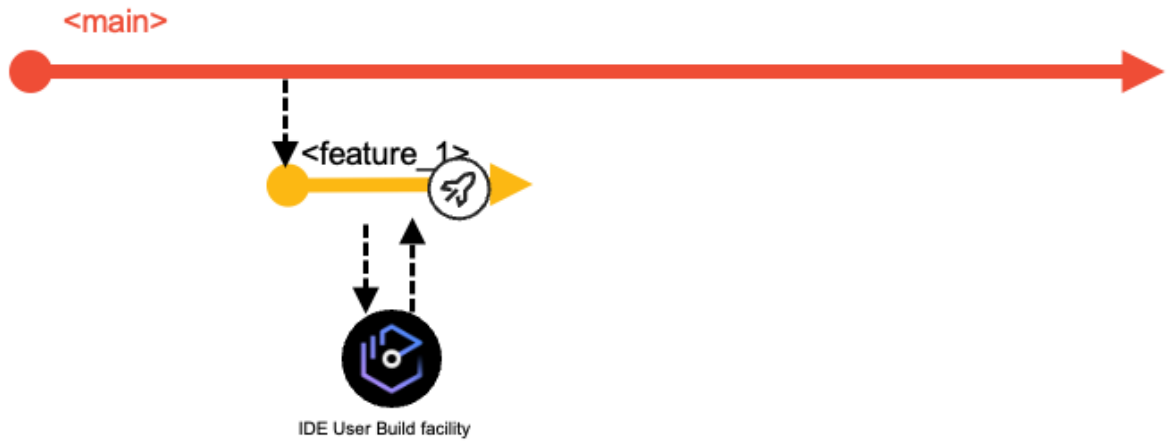
3. To start making the necessary modifications for their development task, developers create a copy of the Git repository on their local workstations through Git's clone operation. If they already have a local clone of the repository, they can simply update their local clone with the latest changes from the central Git repository by fetching or pulling updates into their local clone. This process makes the feature branch available for the developers to work with on their local workstation. They can then open their local clone of the repository in their integrated development environment (IDE), and switch to the feature branch to make their code changes.
4. Developers use the Dependency Based Build (DBB) User Build facility of their IDE to validate their code changes before committing the changes to their feature branch and pushing the feature branch with their updates to the central Git repository. (Tip: Feature branches created locally can also be pushed to the central Git server).



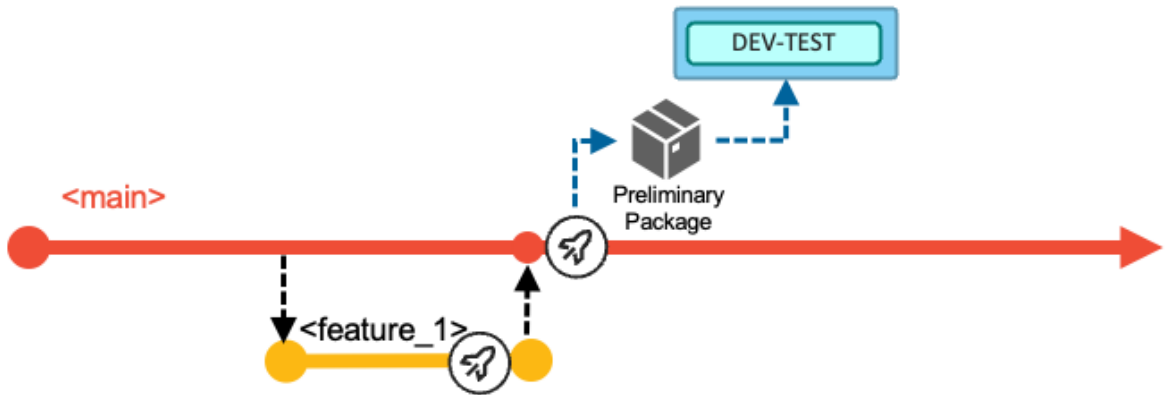
tip

This branching model is also known as a continuous integration model to reduce merge conflicts. While developing on the feature branch, a common practice is for developers to regularly sync their feature branch with the main branch by merging the latest changes from the main branch into their feature branch. This ensures that developers are operating based on a recent state of main, and helps to identify any potential merge conflicts so that they can resolve them in their feature branch.

5. Developers test their changes before requesting to integrate them into the shared codebase. For example, they can test the build outputs of the User Build step. For a more integrated experience, the CI/CD pipeline orchestrator can be configured to run 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 includes the changed and impacted programs within the application scope. Unit tests can be automated for this pipeline, as well. To continue even further testing the feature branch, the developer might want to validate the build results in a controlled test environment, which is made possible by an optional process to create a preliminary package for the feature branch.

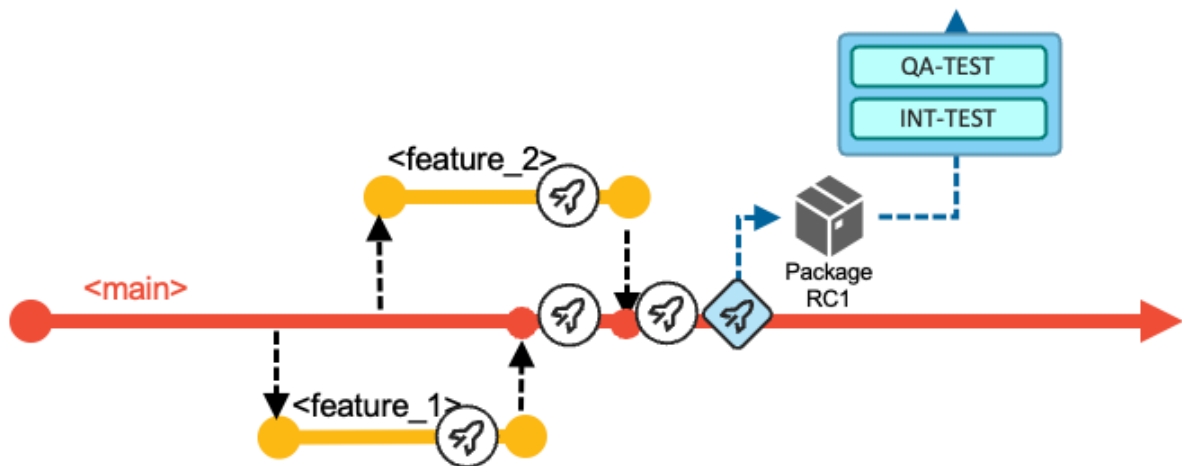


6. When developers feel their code changes are ready to be integrated back into the shared main branch, they create a pull request asking to integrate the changes from their feature branch into the main branch. The pull request process provides the capability to add peer review and approval steps before allowing the changes to be merged. As a basic best practice, the changes must be buildable. If the pull request is associated with a feature branch pipeline, this pipeline can also run automated builds of the code in the pull request along with tests and code quality scans.
7. Once the pull request is merged into the main branch, the next execution of the Basic Build Pipeline will build all the changes (and their impacts) of the iteration based on the main branch.



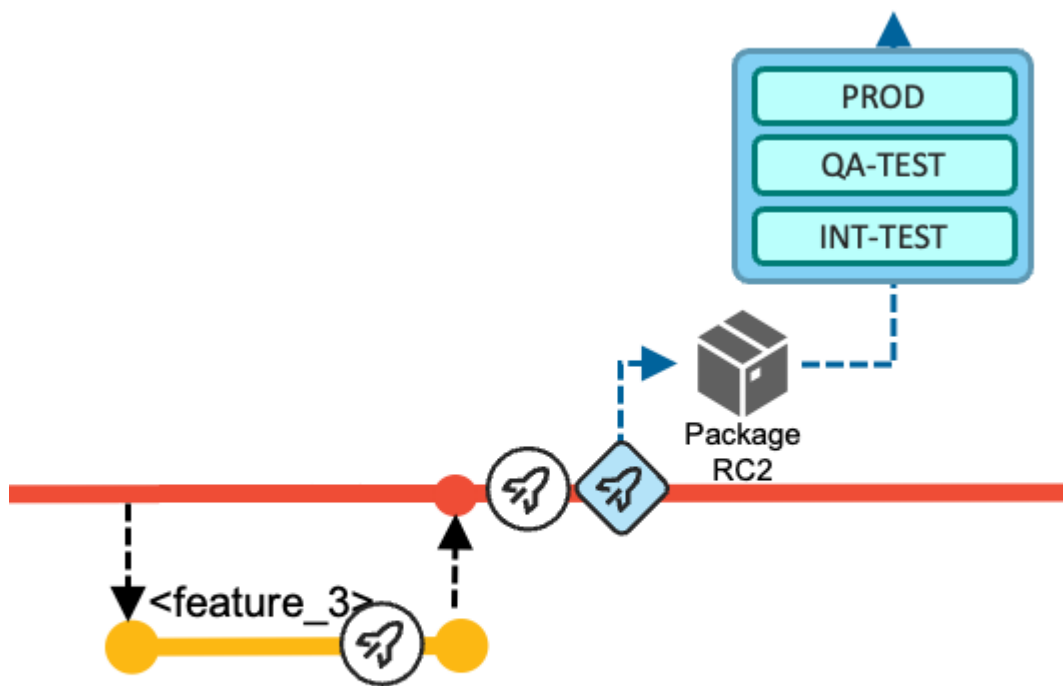
The pipeline can optionally include a stage to deploy the built artifacts (load modules, DBRMs, and so on) into a shared test environment, as highlighted by the blue DEV-TEST icon in the above diagram. In this DEV-TEST environment, the development team can validate their combined changes. This first 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. The installation happens through the packaging and deployment process of a preliminary package that cannot be installed to higher environments (because it is compiled with test options), or alternatively through a simplified script solution performing a copy operation. In the latter, no inventory and no deployment history of the DEV-TEST system exist.

8. In the example scenario for this workflow, the development team decides after implementing Feature 1 and Feature 2 to progress further in the delivery process and build a release candidate package based on the current state of the main branch. With this decision, the development team manually runs the Release Pipeline. This pipeline rebuilds the contributed changes for this iteration - with the compiler options to produce executables optimized for performance rather than for debug. The pipeline includes an additional stage to package the build outputs and create a release candidate package (Package RC1 in the following diagram), which is stored in a binary artifact repository.

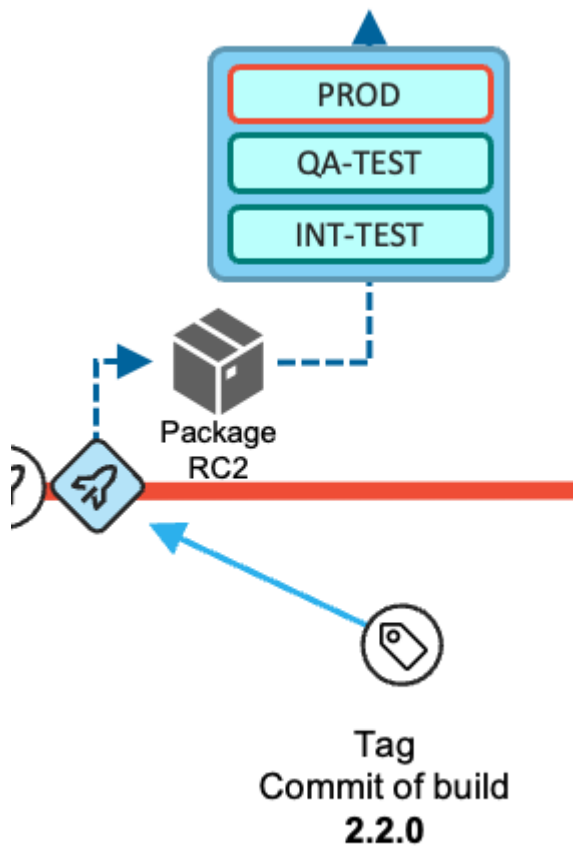


Although not depicted in the above diagram, this point in `main`'s history can be tagged to identify it as a release candidate.

9. The release candidate package is installed in the various test stages and takes a predefined route. The process can be assisted by the pipeline orchestrator itself, or the development team can use the deployment manager. In the event of a defect being found in the new code of the release candidate package, the developer creates a feature branch from the `main` branch, corrects the issue, and merges it back into the `main` branch (while still following the normal pull request process). It is expected that the new release candidate package with the fix is required to pass all the quality gates and to be tested again.
10. In this sample walkthrough of an iteration, the development of the third work item (Feature 3) is started later. The same steps as above apply for the developer of this work item. After merging the changes back into the `main` branch, the team uses the [Basic Build Pipeline](#) to validate the changes in the DEV-TEST environment. To create a release candidate package, they make use of the [Release Pipeline](#). This package (Package RC2 in the following diagram) now includes all the changes delivered for this iteration -- Feature 1, Feature 2 and Feature 3.



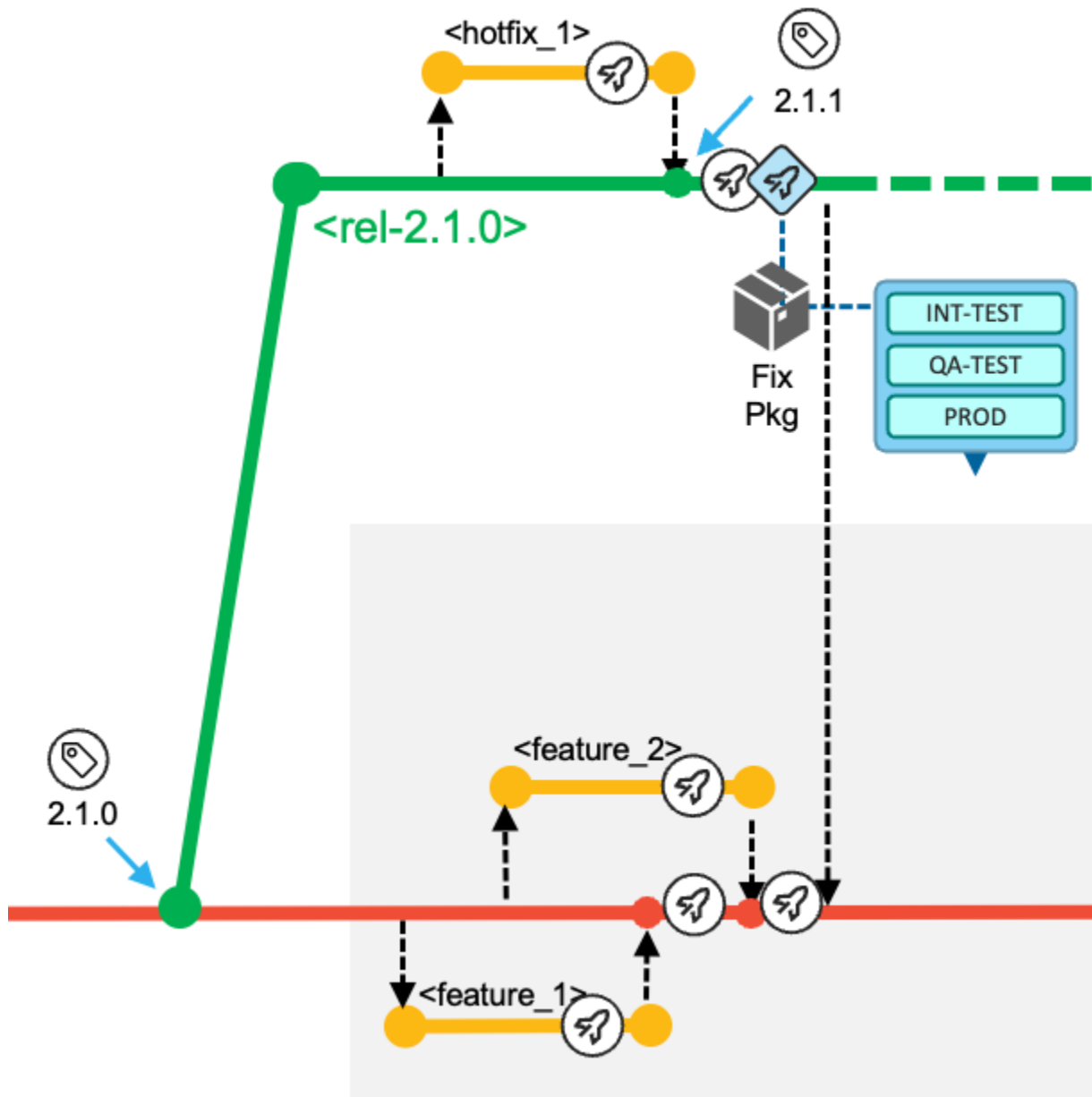
11. When the release is ready to be shipped after all quality gates have passed successfully and the required approvals have been issued by the appropriate reviewers, the deployment of the package from the binary artifact repository to the production runtime environment is performed via the deployment manager or is initiated from the [Release Pipeline](#).
12. Finally, during the release process to the production environment, the state of the repository (that is, the commit) from which the release candidate package was produced is tagged following a [semantic versioning strategy](#). This serves to identify what version is currently in production, and also serves as the baseline reference for the calculation of changes for the next release.



### Implement a fix for the current production state

The process of urgent fixes for modules in the production environment follows the fix-forward approach, rather than rolling back the affected modules and reverting to the previous deployed state.

The following diagram depicts the maintenance process to deliver a fix or maintenance for the active release in production for the application. The process uses a release maintenance branch to control and manage the fixes. The purpose of the branch is to add maintenance to a release that is already deployed to the production environment. It does not serve the process to add new functionality to a future release, which is covered by the default workflow or the usage of an epic branch.



Tip (if you only tagged a release...)

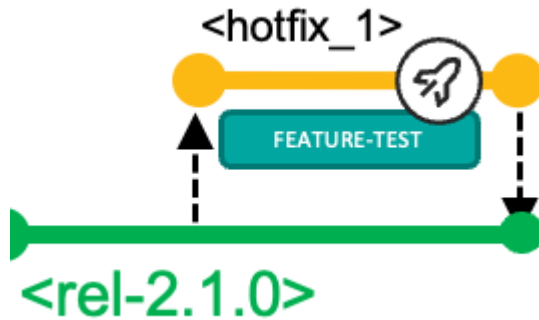
If at the point you built a release candidate package you only marked a release with a tag and did not create a release maintenance branch, you can create that branch at a later time when you find you need it.

For example, `git switch -c release/rel-2.0.1 2.0.1` creates a new branch `release/rel-2.0.1` from `git tag 2.0.1` on `main` and makes it the current branch.

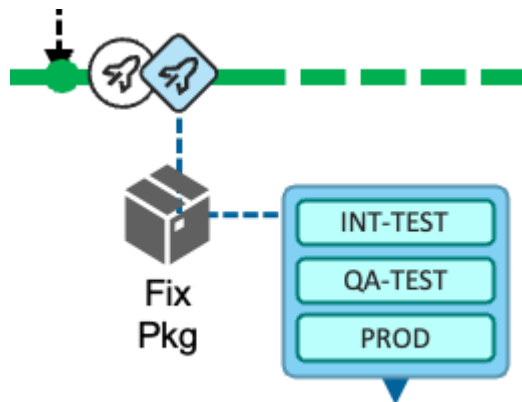
When implementing a fix for the current production state, the development team works through the following tasks:

1. In the event of a required fix or urgent maintenance for the production runtime, which in this example is currently running the 2.1.0 release, the development team creates a `release/rel-2.1.0` branch based on the existing Git tag in the central Git server. The release branch is a protected branch and does not allow developers to directly push commits to this branch.
  - **Note:** In the diagrams for this workflow, the `release/rel-2.1.0` branch is abbreviated to `rel-2.1.0` for readability. Other multi-segmented branch names are similarly abbreviated to their contexts in the diagrams (for example, `feature_1` or `hotfix_1`).

- For each necessary fix, a feature branch is created according to the defined naming conventions `hotfix/rel-2.1.0/52-fix-mortgage-calculation`, represented by the `hotfix_1` based on the `release/rel-2.1.0` branch. This allows the assigned developer to have a copy of the codebase on which they can work in isolation from other development activities.



- The developers fetch the feature branch from the central Git repository into their local clone of the repository and switch to that branch to start making the necessary modifications. They use the user build facility of their IDE to vet out any syntax issues. They can use a feature branch pipeline to build the changed and impacted files. Optionally, the developer can prepare a preliminary package, which can be used for validating the fix in a controlled test environment.
- The developer initiates the pull request process, which provides the ability to add peer review and approval steps before allowing the changes to be merged into the `release/rel-2.1.0` release maintenance branch.
- A Basic Build Pipeline for the release maintenance branch will build all the changes (and their impacts).
- The developer requests a Release Pipeline for the `release/rel-2.1.0` branch that builds the changes (and their impacts), and that includes the packaging process to create the fix package for the production runtime. The developer will test the package in the applicable test environments, as shown in the following diagram.



- After collecting the necessary approvals, the fix package can be deployed to the production environment. To indicate the new state of the production runtime, the developer creates a Git tag (`2.1.1` in this example) for the commit that was used to create the fix package. This tag indicates the currently-deployed version of the application.
- Finally, the developer is responsible for starting the pull request process to merge the changes from the `release/rel-2.1.0` branch back to the main branch to also include the fix into the next release.
- The `release/rel-2.1.0` branch is retained in case another fix is needed for the active release. The release maintenance branch becomes obsolete when the next planned release (whose starting point is represented by a more recent commit on the main branch) is deployed to production. In this event, the new commit point on the main branch becomes the baseline for a new release maintenance branch.

## Use an epic branch for a significant development initiative

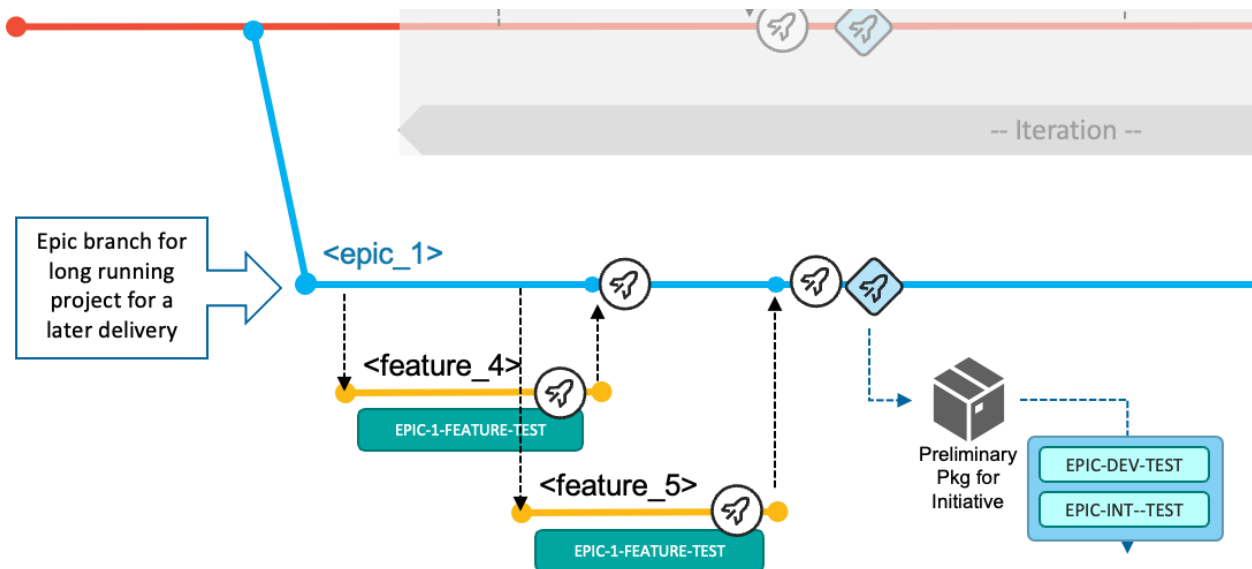
Let us now focus on change requests that represent significant work effort and require major changes, for instance, due to updates in regulatory frameworks in the banking or insurance industry, or the need to already kick off the development phase of features not planned to be delivered in the very next release.

In these situations, the development team cannot follow the business-as-usual workflow to deliver functionality with the next planned release, because the time and work required breaks out of the traditional durations of one release. For each of these scenarios, the development team is using an epic branch to keep the changes in multiple features separated from the other development activities. It is an integration branch to group and integrate multiple features that are planned for this initiative. Ideally, the team has a dedicated test environment assigned (such as EPIC-DEV-TEST and EPIC-INT-TEST in the following diagram), to also plan and implement any infrastructure updates such as Db2® schema changes.

Trunk-based development suggests using feature flags within the code to implement complex features via the main workflow while allowing the delay of their activation. Feature flags are often employed so that a given business functionality can be activated at a given date, but be implemented and deployed earlier (whether to dev/test or production environments). We do not see this as a common practice for traditional mainframe languages such as COBOL or PL/I, although some development organizations might apply this practice in mainframe development.

All these scenarios lead to the requirement on the development process to implement changes independently from the [main workflow](#).

Note that the epic branch workflow described in this section is not meant to be used for a single, small feature that a developer wants to hold back for an upcoming release. In those smaller cases, the developer retains the feature branch until the change is planned to be released.

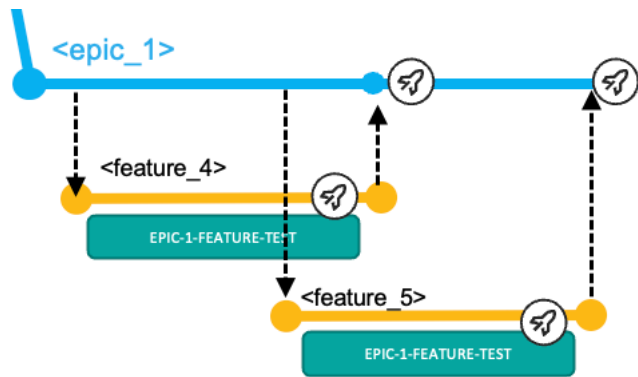


The development tasks for a development initiative are:

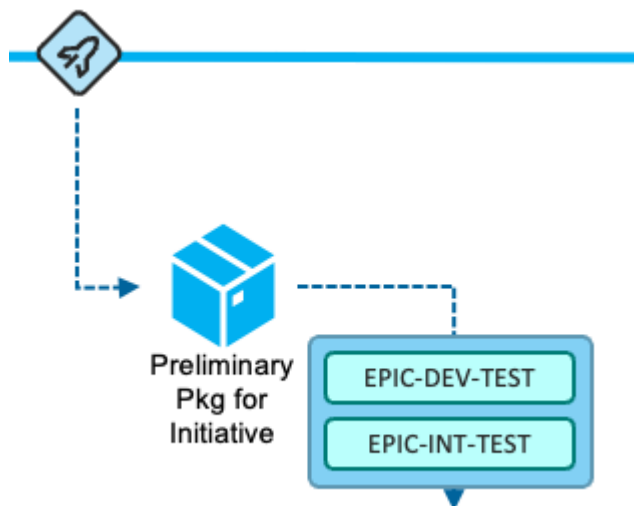
1. The team creates an epic branch from the Git tag representing the current production version of the application, which is at this point the most stable configuration. This process provides them isolation of the codebase from any other ongoing changes for the next iteration(s). In this workflow, the epic branch for the the larger AI fraud detection development initiative is named `epic/ai-fraud-detection`. In the diagrams, it is abbreviated as `epic_1`.
2. Based on how the work items are distributed between the developers, a feature branch is created based on the epic branch according to the pre-defined naming conventions. For example, the feature branch `feature/ai-fraud-detection/54-introduce-ai-model-to-mortgage-calculation` is based off the epic branch `epic/ai-fraud-detection`.
3. The developers fetch the feature branch from the central Git repository into their local clone of the repository and switch to that branch to start making the necessary modifications. They use the user



build facility of their IDE for building and testing individual programs. They can also use a feature branch pipeline to build the changed and impacted files. Optionally, the developer can prepare a preliminary package, which can be used for validating the fix in a controlled test environment, such as the EPIC-1-FEATURE-TEST environment shown in the following diagram.

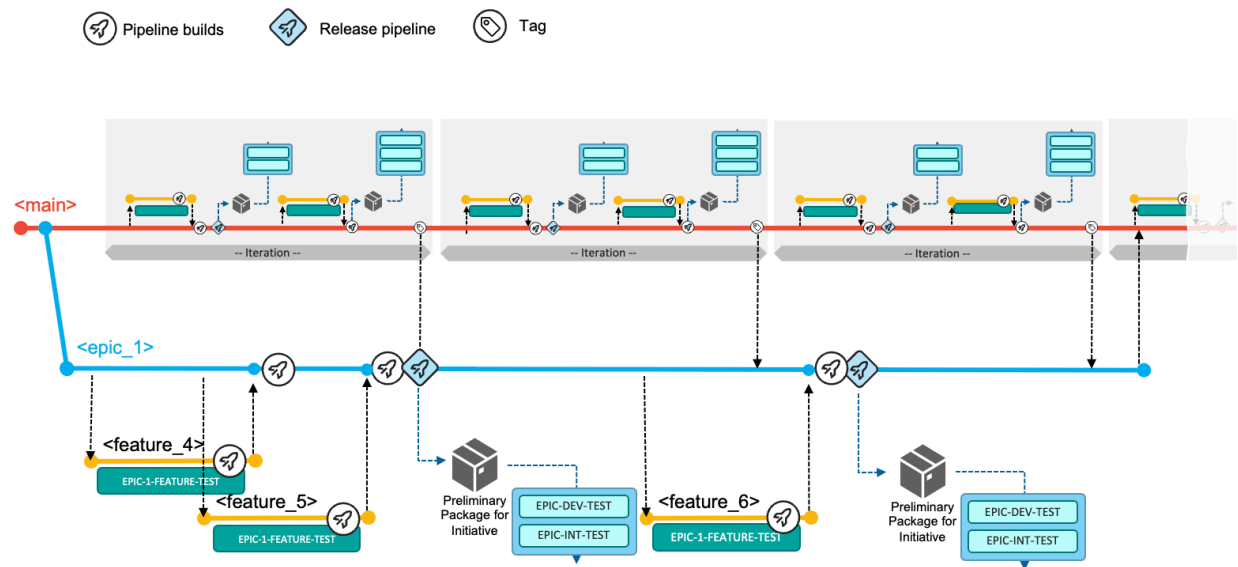


4. The developer initiates the pull request process, which provides the ability to add peer review and approval steps before allowing the changes to be merged into the epic branch.
5. A Basic Build Pipeline for the epic branch will build all the merged features (both the changes and their impacts) from the point where the epic branch was branched off.
6. It is important that the team frequently incorporates updates that have been implemented for the next release and/or released to production via the default development workflow (with the main branch) into the epic branch to prevent the configurations from diverging too much and making the eventual merge of the epic branch into main difficult. A common practice is to integrate changes from main into the epic branch at least after each completion of a release via the default workflow, in order to merge in the latest stable version updates. More frequent integrations may lead to pulling intermediate versions of features that might not be fully implemented from a business perspective; however, this should not deter developers since the main branch should always be in a buildable state.
7. When the development team feels that they are ready to prototype the changes for the development initiative in the initiative's test environment, they request a Release Pipeline for the epic branch that builds the changes (and their impacts) and includes the packaging process to create a preliminary package. This preliminary package can then be installed into the initiative's test environment (for example, the EPIC-DEV-TEST environment). The team will test the package in the assigned test environments for this initiative, as shown in the following diagram.



8. Once the team is satisfied with their changes for the development initiative, they plan to integrate the changes of the epic branch into the main branch using the pull request process. This happens when

the changes should be released towards production with the next planned iteration. The following diagram depicts of the process of integrating the changes implemented for epic\_1 in parallel with the default workflow after three releases.



## Learn more

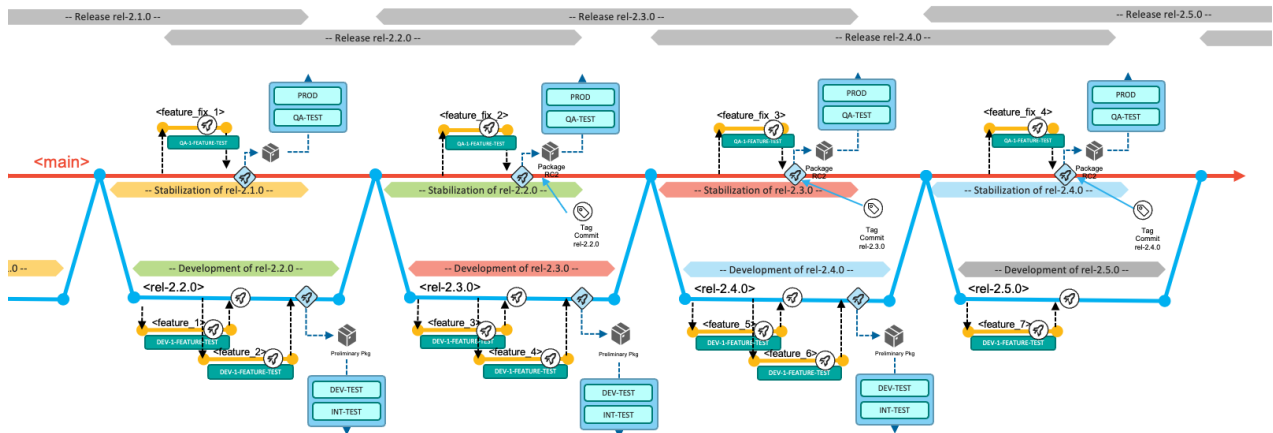
This page describes our recommended Git branching model and workflows for mainframe development. This model is intended to be used as a template, and can be adjusted, scaled up, or scaled down according to the needs of the development team. Additional variations for the branching strategies and workflows can be found in [Working practice variations](#).

For recommendations on designing and implementing the workflows described in this branching model, please refer to [Implementing a pipeline for the branching model](#).

## Working practice variations

### Working with overlapping releases

In traditional mainframe development workflows, teams follow a practice of overlapping releases. In that situation, the team uses the *main* and *epic* branches following a rolling wave pattern: The team decides which commit/tag of the main codeline will be used to baseline the overlapping release -- most likely when the previous release moves into its release stabilization phase. The development phase of the subsequent release then occurs on the epic branch and is merged back into main when entering the stabilization phase. This leads to the below composition of the *main* and *epic* workflow:



## Audit and compliance

This section outlines key application audit requirements and compliance practices within a z/OS® DevOps workflow.

### What is an audit?

"An Information Technology audit is the examination and evaluation of an organization's information technology infrastructure, applications, data use and management, policies, procedures, and operational processes against recognized standards or established policies. Audits evaluate if the controls to protect information technology assets ensure integrity and are aligned with organizational goals and objectives." — Definition provided by [Harvard University](#)

### Application audit requirements

The above definition encompasses all aspects of an audit. For our purposes, we will focus on what an auditor looks for after an application development cycle.

Two key areas of an application audit are access controls and the software development lifecycle. The general requirements for these areas are described in the following subsections.

#### Access controls

Auditors ensure teams are authorized to work on a given application. Access control layers include permitting:

- Development team members to make changes to an application
- Peers to review and approve changes
- Team leads to give final approval and integrate changes
- Administrators to have general oversight and optionally approve releases to upper environments
- Auditors to have general access for oversight and compliance review

#### Software development lifecycle

Auditors review software development compliance along the following key areas:

- **Release management**  
Assess the controls around changes, ensuring they are documented, tested, reviewed, and approved.
- **Patch management**

Ensure patches (hot fixes) are applied to address vulnerabilities or outages.

- **Separation of duties**

Prevent errors and fraud by ensuring no single individual has control over all aspects of the development process.

- **Traceability and reporting**

Ensure secured and immutable compliance evidence (audit trails) like logs and reports across the development lifecycle that include:

- What was changed and why
- When it was changed
- Who made the change
- Who reviewed and approved it
- How artifacts are tracked from development to production

- **Records retention**

In some cases, audit report retention of up to 7 years is required as defined under the [Sarbanes-Oxley Act \(SOX\)](#) in the USA. Other countries may have different requirements.

## Application audit compliance practices

A core component of audit compliance is the rules that define and enforce the activities that users have within the development lifecycle.

All major DevOps service providers and tools include access controls, typically enforced by Role-Based Access Control (RBAC) policies under the Lightweight Directory Access Protocol (LDAP). These access controls can be extended to integrate with z/OS security facilities such as [Resource Access Control Facility \(RACF\)](#).

Examples and key practices that administrators can implement to comply with the audit requirements outlined above include:

- **Source code access:**

Using GitHub as an example Git provider, RBAC is implemented through a system of permissions and roles that manage individual user and team access to repositories and other organization resources.

- **Git branch protection:**

Administrators define access rules to prevent unauthorized updates to, for example, production-level branches. For more details on branching models and protection, see the IBM Z® DevOps Acceleration Program's recommended [Git branching model for mainframe development](#).

- **Separation of duties:**

Separation of Duties (SoD) establishes a broad set of access controls through various practices and tools:

- RBAC is used to enforce SoD by assigning different roles and permissions to different users. Each role has specific responsibilities and limitations, ensuring that no single person can perform all critical tasks.
- Workflow automation tools provide features that enforce SoD by requiring multiple approvals and checks at different stages of the development lifecycle.
  - Git pull requests: Developers submit pull requests for code changes, which must be reviewed and approved by one or more peers before merging.
  - Approval gates: Continuous integration/continuous delivery (CI/CD) pipelines include approval gates where designated individuals must approve a build and release.
  - Trusted users: In z/OS, CI/CD processes run under a privileged account, also known as a "protected account" in RACF. For example, this account is authorized to run automated builds of any mainframe application using tools like IBM® Dependency Based Build or to release any package from development up to production using tools like IBM Wazi Deploy.

- **Traceability and reporting:**

- Git history: Track who made changes, when, and what was changed.
  - CI pipeline logs: Logs from CI tools like Azure DevOps CI, which show what was built, by whom, and where the output artifacts were stored.
  - CD pipeline logs: Logs from CD tools like IBM Wazi Deploy, showing where artifacts were downloaded and deployed on a z/OS host.
- **Records retention:**  
DevOps administrators ensure that logs and reports are protected and retained based on the organization's policies.

## Further reading

While this page highlights key aspects of software development audit requirements, there are many more topics to consider. DevOps tooling vendors may also have their own specific documentation on implementing compliance using their products. The following examples are vendor-specific references with further details:

- [Azure® DevOps and RBAC](#)
- [GitLab® administration and compliance](#)



---

# Chapter 4. Tutorials and courses

## Tutorials

---

### IBM Z Systems software trials (IBM Z Trial)

If you are new to DevOps for z/OS® applications, you might want to explore the workflow and tooling without having to first install and configure an entire technology stack on your own environment. [IBM Z® Systems software trials](#) (also known as IBM Z Trial) allows you to try out a variety of IBM Z software experiences in a provisioned environment for free, meaning you can get right to learning how to use these tools/technologies. The following IBM Z Trial experiences are particularly relevant to DevOps and CI/CD pipelines for z/OS applications:

- **Bring Your Own (BYO) IDE for Cloud Native Development:** Explore the integrated development environment (IDE) functionality in the Eclipse-based IBM® Developer for z/OS® (IDz) and/or Microsoft® VS Code™ with IBM Z extensions, and learn how these capabilities fit into the DevOps workflow with other tools such as Git, DBB, and Jenkins.
- **IBM Application Delivery Foundation for z/OS®:** Explore the range of features in the IBM Application Delivery Foundation for z/OS (ADFz) suite of integrated tooling that can help you analyze, understand, debug, and modify your COBOL programs.
  - **Further reading:** [ADFz Resources](#) contains links to standalone enablement resources for ADFz, IDz, and IBM Z DevOps including videos, blog links, new release announcements, PDFs, and other deep-dive learning content.

### CI/CD pipeline tutorials

Once you have the prerequisite DevOps tools installed on your environment, you can follow CI/CD pipeline tutorial(s) that cover your selected technologies.

**Note:** Some tutorials might use older product versions, but are included on this list because the information is still useful reference material.

- **Build a pipeline with GitLab CI, IBM Dependency Based Build, and IBM UrbanCode® Deploy:** Learn how to configure GitLab CI for a CI/CD pipeline that includes builds with IBM Dependency Based Build (DBB), packaging and deploy phases with IBM UrbanCode Deploy (UCD), automated unit testing with the ZUnit feature of IBM Developer for z/OS (IDz), and code coverage and code review with IDz.
- **Build a pipeline with Jenkins, Dependency Based Build, and UrbanCode Deploy:** Learn how to configure UrbanCode Deploy (UCD) and Jenkins to build a pipeline that uses DBB and Artifactory to streamline the development of applications up through the deployment phase. This tutorial also covers how to include automated unit testing, code coverage, and IBM Developer for z/OS (IDz) code review in the pipeline by using the ZUnit feature of IDz.

For a deeper dive on specific IBM tools and technology stacks, check out our [Courses](#) page.

## Courses

---

The DevOps Acceleration Team (DAT) offers free courses that are beneficial to learners who want to increase their DevOps skills in a holistic and engaging manner. The available courses are a mix of self-paced and remote instructor-led courses, and cover a variety of topics for different roles. An IBM®-issued Credly badge is awarded to the learner upon successful completion of each course.

- Self-paced courses: These courses can be taken at your own pace, whenever it is convenient for you.
- Remote instructor-led courses: Instruction occurs with a remote class and instructor over web conference on a set schedule, offering interaction opportunities.

For a CI/CD learning plan organized by role, you can check out [DevOps Transformations for IBM zSystems™](#) and [CICD pipelines with DBB Git](#).

Below, you will find courses organized by where in the IBM Z® DevOps transformation journey a customer would typically first encounter the product(s) covered in that course. You can learn more about each course by clicking on its hyperlinked course title. While these courses are primarily targeted towards enabling end user developers, additional roles that can benefit are called out under each course category.

## Learn to code and develop in a modern integrated development environment

- **[IBM Developer for z/OS® \(IDz\)](#)**: The IBM Developer for z/OS (IDz) basics course is self-paced and designed to provide the audience with an introduction to IDz, a modern integrated development environment (IDE) for Z development.
- **[DevOps Distance Learning Program](#)**: This remote instructor-led course provides an interactive learning experience with both entry-level and deeper-dive focus sessions for IDz and other useful tools in the larger IBM Application Delivery Foundation for z/OS (ADFz) product suite.

Related role(s): [IDE specialist](#)

## Gain insights into your z/OS applications with analysis tooling

- **[IBM Application Discovery and Delivery Intelligence Essentials](#)**: This self-paced course introduces IBM Application Discovery and Delivery Intelligence (ADDI), an analytical platform for z/OS application modernization.
  - **Further reading:** The [ADDI Accelerator](#) contains materials and resources designed to help customers accelerate their adoption of ADDI and enable their organizations to successfully deploy this analysis and discovery tool.

Related role(s): [Architect](#), [IDE specialist](#)

## Use an intelligent build tool to compile and link your z/OS applications

- **[IBM Dependency Based Build Fundamentals](#)**: This self-paced course provides the audience with an introduction to building mainframe applications in a DevOps pipeline with IBM Dependency Based Build (DBB).

Related role(s): [Build specialist](#)

## Streamline quality assurance with IBM Z testing tools

- **[IBM Z DevOps Testing Fundamentals](#)**: This self-paced course is designed to teach how z/OS application developers can shift-left and accelerate testing using IBM Z DevOps testing tools such as ZUnit, a powerful unit testing framework specifically tailored for testing z/OS applications and mainframe environments.

Related role(s): [Testing specialist](#)

## Modernize with cloud native DevOps tooling

- **[Cloud native development with IBM Z and Cloud Modernization Stack Fundamentals](#)**: This self-paced course introduces IBM Wazi for Red Hat® OpenShift Dev Spaces (IBM Wazi), a cloud native productive development environment that fully integrates with any enterprise-wide standard DevOps pipeline.

Related role(s): [Architect](#), [Build specialist](#), [Pipeline specialist](#), [Deployment specialist](#), [IDE specialist](#), [Testing specialist](#)



---

# Chapter 5. Designing the CI/CD pipeline

## Planning repository layouts and scopes

---

Because the traditionally large and monolithic nature of enterprise applications can slow down development both at the organizational level and the individual developer level, many enterprises take the migration to Git as an opportunity to examine how they can break their monolithic applications down into more agile functional Git repositories. The repository layout in the source code management (SCM) component of the continuous integration/continuous delivery (CI/CD) pipeline affects other pipeline components in several ways:

- Repository layout affects the developer experience, as it impacts the level of isolation for development activities - in other words, how much of your development can happen independently of code or components managed in other repositories.
- The scope of the build is typically defined by what is included inside a repository versus outside of it. For shared resources such as copybooks, other repositories might need to be pulled in.
- The scope of the package sent to the artifact repository will depend on the outputs produced by the build step.

### Basic components in a Git SCM layout

At a high level, the Git SCM can be thought of in two components: the central Git provider, which is the source from which developers will be pulling code from and pushing code to when they do their development work, and the clone on the developer workstation.

Within the central Git provider, there can be multiple organizations:

- Each organization can have multiple repositories inside it.
- In turn, each repository can have multiple branches (or versions) of the code in the repository.
- Branches enable updates to be isolated within a branch until they are ready to be included in other branches. This allows each developer to focus on their development task on a dedicated branch, only needing to worry about upstream and downstream impacts when they are ready to integrate those with their work. By convention, there is one branch that contains the latest, greatest, and most up-to-date agreed state of the code - this is conventionally known as `main`. Other branches can exist that represent a past state of the code or new changes that are not yet ready to be included in `main`.
  - Branching in Git is lightweight both in storage space and time - no replication of the entire set of files is required to make a new branch; rather, it is a *copy-on-update* scheme that only requires the storage of the differences in files.

On the developer's workstation, the developer will have however many repositories they have decided to clone down onto their local computer to work with. By default, the clone of the repo from the server is an entire copy of the repo which, for example, includes the same history of changes and all the same branches that exist on the server. When working on changes, the developer will switch their working tree to be the contents of a particular branch.

### Working with Git

Git is a distributed version control system, so rather than the files all staying in the mainframe during development, developers will clone the repository they are interested in down to their local workstation. Your organization's administrator(s) can control who has what permissions, or which repositories each developer has access to.

Working with a distributed version control system like Git also means that when planning the SCM layout, one factor to consider is size. Generally, you want to avoid doing something like migrating the entire codebase of a large application into one repository.

Large repositories can be cumbersome for a variety of reasons (although there are organizations that successfully manage their large codebases in so-called *monorepos*). For example, without using more sophisticated Git commands and options, when developers need to clone the repo it will take longer.

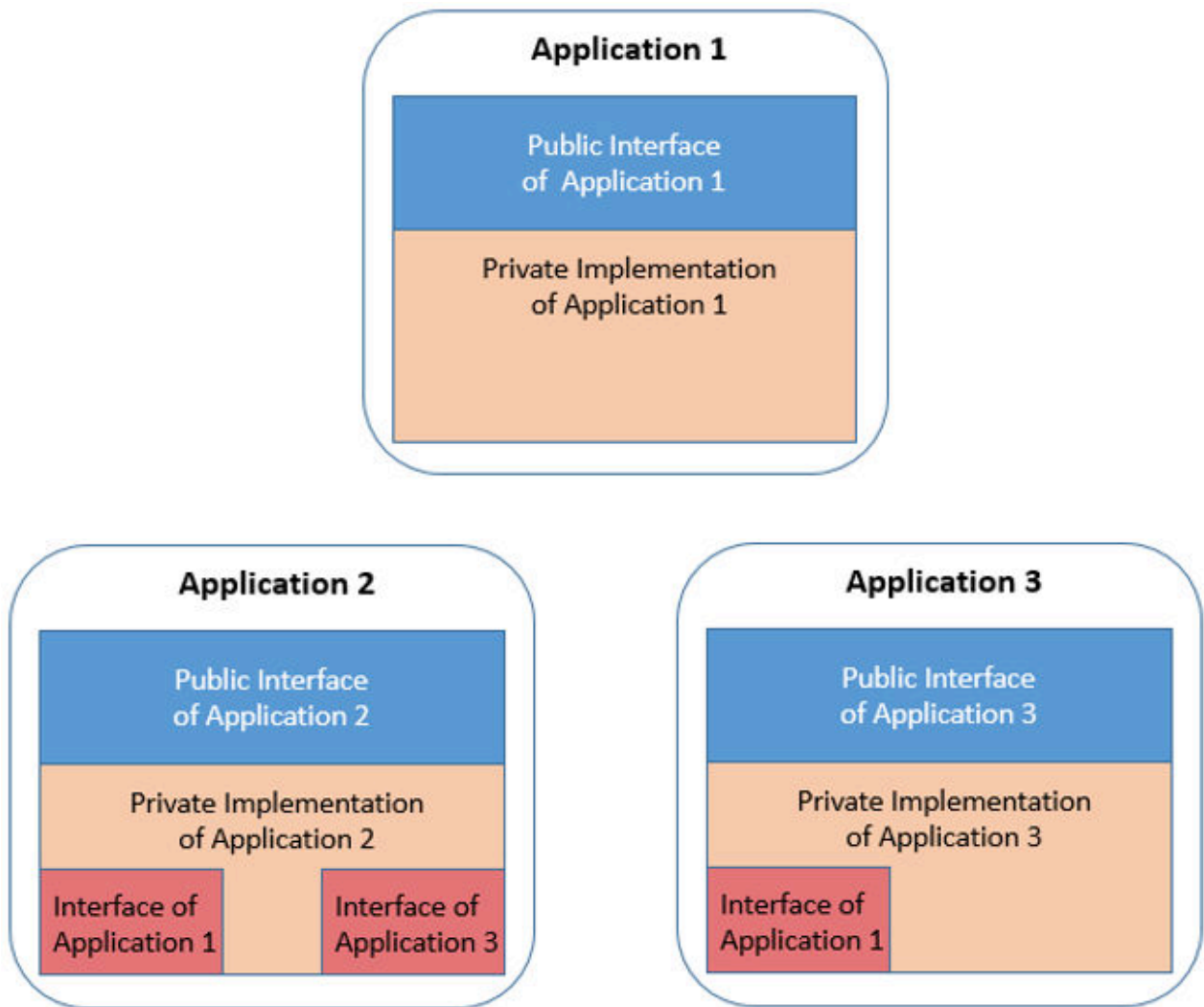
## Visualizing the SCM layout

Traditionally, mainframe applications are often seen as a monolith, but as mentioned previously, it is generally not preferable to have all of your codebase in one giant repository. To begin thinking about how to separate an application "monolith" into different repositories and designing your SCM layout, you can start to think about boundaries within your application, which are formed by things such as responsibilities and ownership.

To use an analogy, you can think of the application like a large capital city. On the surface, it might seem like this city is one big entity - a big monolith. However, upon closer inspection, the city is broken up into districts. And while each district has its own workings (for example, each district has its own school system), all of these districts are connected by a broader infrastructure in a controlled way. There might be public transport, such as a metro or buses, which run on predefined paths and schedules. We can think of this transport infrastructure like the data flowing between application boundaries within a larger system.

In a similar sense, the districts within the city could be likened to groupings of related files and programs, which can be formed into their own smaller applications (in a loose sense of the term) with data flowing between them.

The following diagram illustrates the point more literally. In this example, there are three applications, which we can again think of like the city districts:



- Each application has its own inner workings in the private implementation (represented by the tan parts of the above diagram). In the city analogy, an example of this could be the school system each district has. In the mainframe context, an example of these inner workings could be the COBOL programs for that application.
- Additionally, each application needs to be able to communicate with other applications, which is accomplished via its public interface (shown in blue on the diagram). Using the city analogy again, the bus stop for each district could be considered as the district's "public interface". In the mainframe context, the public interface could be the copybooks defining data structures that are passed between programs.
- Because each application needs to know how to interact with the other applications, the application also includes usage of interfaces from the other applications (shown in red on the diagram). This can be likened to the application needing to know the bus stops and schedule from the other applications.

Typically, with this layout, each application would have its own Git repository with its own scope. Compared to putting the whole system in a single large Git repository, having these smaller repositories enables more agile development practices. Companies choose a central, web-based repository server where they can manage their repositories, and from which their developers will clone from (for example, Git providers such as GitHub, GitLab, Bitbucket, Azure Repos, and so on). The following section provides some guidance to consider when breaking down a monolithic codebase to achieve this kind of setup.

## Guidelines for Git repository scope and structure

The guidelines in this section can be used when considering how to cluster your codebase into Git repositories in a way that makes sense and encourages a streamlined workflow for your organization. There are several factors to balance when scoping repositories, including the following:

- Access control
  - Consider who (or which teams) need access to which files. Read/write, read-only, and restricted permissions need to be preserved, and this can be achieved by separating the files into different Git repositories and then managing access accordingly through your Git provider.
- Functional areas of the code
  - In your current system, if different teams work on different components of your application, it might make sense to make boundaries based on that. An individual can be part of one or several teams.
- Decoupling
  - Consider decoupling applications where it makes sense to do so. This allows different functional areas to have separate release cycles, which can help promote more agility in the overall organization.
- Interfaces
  - When scoping for Git repository layouts, consider if or how changes performed by a team in one repository might impact other teams. In other words, within a team, if a developer on that team makes a breaking change to their repository (for example, to the application's private implementation), they can just work with their team to fix it. However, if a developer makes breaking changes impacting another team (for example, to the application's public interface), then resolving it can become more complicated.
- Size
  - Consider the scope of changes for a typical change request (or pull request/merge request). For example, it is typically preferable to avoid change requests spanning across multiple repositories. Git providers usually have built-in review and approval mechanisms, so it is recommended to set up the repository layout in a way where these approval and review systems will make sense.
- Performance
  - Git performance can be a consideration, but it should not be the primary driver to make decisions when organizing repository layouts.

These guidelines impact how application teams implement change, and must be carefully considered.

## Recommended internal Git repository layout

A efficient internal layout - that is, the layout of folders within Git repositories - helps application developers easily navigate the codebase of an application. The internal layout also affects the CI/CD pipeline setup, particularly the build and packaging stages. Establishing standardized conventions helps provide a consistent developer experience and simplifies the pipeline implementation.

Generally, the internal repository layout should cater for various artifact types:

- z/OS mainframe artifacts such as COBOL, PL/I or Assembler programs and include files and JCL,
- Service definitions (in the wider sense of APIs) such as IBM z/OS Connect projects,
- Java source code implementing application components that can be run on the mainframe or a distributed runtime environment,
- Test artifacts:
  - Standardized unit tests to test single modules,
  - Manually triggered test drivers,
  - Integration test configurations and scripts,
- Documentation and README,

- and everything that makes up the application configuration, such as Infrastructure as Code definitions, for instance CICS Resource builder definitions.

There is no common industry standard that existing open source projects follow. The internal repository layout is rather determined by how it has been established when the project started, how the development team performs change, as well as how the application architecture evolves over time.

The following list is not exhaustive but represents some of the important factors to consider when designing the internal Git repository layout:

- Navigation through the components and functional areas of the application
- Accommodation of changes in the application architecture
- Addition or change of programming languages

Typically, the first level of folders in the repository is driven by the various application components. The second level provides an architectural breakdown on the applications' code base. In the sample layout below, the docs folder allows the application team to store the documentation of the application under version control. The cbsa folder is named after the application and groups the artifacts by their functional areas, such as services (APIs), z/OS source code, tests, and Java:

```
docs/
cbsa/
|- application-conf/
|- api/
|  |- src/
|  |- test/
|- zos/
|  |- src/
|     |- cobol/
|     |- copybooks/
|     |- copybooks_gen/
|     |- interfaces/           (containing services to other mainframe applications)
|  |- test/                   (e.g. Mainframe Unit tests)
|- test/                       (e.g. Galasa tests)
|- java/
|  |- src/
|     |- com/acme/cbsa/packages
|  |- test/ (JUnit)
```

Each component folder contains `src` and `test` subfolders when applicable, to segregate the source code from the test artifacts. For the mainframe artifacts, the most common layout is to group by artifact type, such as the programming languages. Inside the `zos/src` folder, the structure indicates the different purposes of the application's artifacts. For instance, the folder `interfaces` contains include files owned by the application that can be referenced by other applications.

For the pipeline setup, the `clone` and `checkout` phase can be enhanced to retrieve external dependencies from other Git repositories. These steps can also be configured to only retrieve the shared include files that reside in the `interfaces` folder. Additionally, standardized dependency resolution rules can be applied in the build framework configuration.

Having a standardized layout across applications' repositories helps in using some features that Git provides usually provide. One of these features is the capability to implement a policy-based control of who needs to review and approve changes (known as CODEOWNERS or branch policies). Such a mechanism is easier to setup when based on the repository's structure.

To facilitate the navigation in the codebase, it is possible to further group source code by business functionality or by architectural composition of the application. In the following sample layout, the mainframe component, `zos`, is split into core account management functions (such as `create`, `update`, and `delete`), payment functions (to handle payments between accounts), and shared functions.

```
docs/
cbsa/
|- application-conf/
|- api/
|  |- src/
|  |- test/
|- zos/
|  |- src/
```

```

|- account.management /
  |- cobol/
  |- copybooks/
  |- copybooks_gen/
  |- interfaces/
|- account.payments /
  |- cobol/
  |- copybooks/
  |- copybooks_gen/
  |- interfaces/
|- account.shared.functions /
  |- cobol/
  |- copybooks/
  |- copybooks_gen/
  |- interfaces/
|- test/ (e.g. Mainframe Unit tests)
|- test/ (e.g. Galasa tests)
|- java/
  |- src/
  |- com/acme/cbsa/packages
  |- test/ (JUnit)

```

Combining various application components into a single repository allows changes to be processed by a single pipeline with a streamlined build, packaging, and deployment process as part of a cohesive planning and release cadence. However, if application components are loosely coupled and/or even maintained by different teams, it often makes more sense to maintain them in separate repositories to allow for independent planning and release cycles. The following section provides guidance on how to manage dependencies between separate repositories.

## Managing dependencies between applications in separate repositories

One of the challenges in designing the scope of applications and their repositories is about the dependency management, inside and across applications. For mainframe applications, this implies thinking about how the files are arranged. If a program and its subprogram are located in the same application's repository, then there is an 'internal' dependency between these artifacts, that does not impact other applications. If they are located in different applications' repositories, the dependency between these repositories can be qualified as 'external'.

Languages such as Java™ allow you to access other external applications by referencing their application programming interfaces (APIs). However, with COBOL (and other mainframe languages), the build process needs to pull in the external dependencies as source, via concatenation. Therefore, this is something you will want to consider when designing your SCM layout.

To help guide the SCM layout design process, we can approach the SCM design with a two-step process:

1. Define which files make up each application: This is like defining the "districts" in your city, if we go back to the city analogy. You can consider looking at boundaries that have already naturally arisen based on ownership and responsibility around different parts of the codebase, in addition to the other [guidelines for repository scope](#).
  - You might already have an idea of which files make up each application based on a data dictionary.
  - IBM® Application Discovery & Delivery Intelligence (ADDI) can assist in visualizing call graphs and clusters of dependencies in your codebase. These visualizations can be useful when you are analyzing the code to figure out how to organize it.
2. Understand the internal versus external elements (such as copybooks) for each application: Here, the objective is to identify interfaces and shared items across application scopes - in other words, we want to determine which application is using which copybooks. In the city analogy, this is like identifying the bus stop names. Based on the repository layout, you can minimize how much changes in one repository will impact other repositories (and in turn, other teams and applications). Ideally, you will end up with files that have a lot of references within the same repository.

The result of the analysis in the steps above can be reflected within each application's repository. In the following example, you can see that the file structure is organized to indicate private versus public files and interfaces.

```

/BOOKING-TAXES
  |_ COPYBOOK
    |_ COPY01.cpy
  |_ COPYBOOK_PUBLIC
    |_ COPY02.cpy
  |_ COBOL_PROGRAMS
    |_ LO28050.cbl
  |_ ASM_PROGRAMS
    |_ AM28050.asm

/BOOKING-TAXES-FOREIGN

```

....

More details about managing dependencies can be found in the documentation for [Defining dependency management](#).

## Advanced topic: Options for efficient use of Git repositories

Migrating z/OS® applications from PDSs to Git takes planning and analysis. If you are new to Git, the earlier sections of this page provide an overview of general guidelines to keep in mind. The following advanced section explores finer details that can be considered when optimizing Git repositories for performance.

Modeling applications by size is a good first step in managing performance and capacity. The following table is an example that models applications by number of files (PDS members) times average total lines at 80 bytes each. Use it to compare with your application repos to determine what, if any, optimization may be needed.

The table also includes total size compressed by Git. Compression reduces the amount of disk space objects take up, which is important for efficient storage and transfer of Git repositories. Smaller objects mean less data to transfer, which can lead to faster clone, fetch, and push operations.

App size	File count	Average lines per program at 80 bytes per line	Size (MB)	Size at 50% compression (MB)
small	1,500	1,000	114	57
med	5,000	1,500	572	286
large	50,000	5,000	19,073	9,537
Very Large	140,000	5,000	53,406	26,703

Total size is half the story. Managing a Git repository's history also plays an important part in maintaining optimum performance.

The size of a Git repository's history can vary depending on the number of commits, the size of the files, and how often the repository is modified. In some cases, the history can become larger than the total size of all the files in the repository, especially if there are many commits that modify the same files over time.

Git employs various compression and storage optimization techniques to manage the size of the history, such as storing file differences instead of full copies for each commit and using delta compression for objects in the object database.

When comparing the performance of Git operations on repositories with large histories versus smaller histories, several metrics can be considered, including network transfer speed and CPU usage. Here's a general comparison:

- Clone speed: Cloning a repository with a large history will typically take longer than cloning one with a smaller history. This is because Git needs to download and process more data for the larger history.
- Fetch speed: Similarly, fetching updates from a remote repository can be slower for repositories with large histories, as Git needs to transfer and process more data.
- Commit speed: Committing changes to a repository may be slower for repositories with large histories, especially if there are many files or changes that need to be processed.
- Branching and merging: Creating branches and performing merges can be slower for repositories with large histories, as Git needs to analyze the history to determine the correct merge base.
- CPU usage: Git operations on repositories with large histories may consume more CPU resources than those on repositories with smaller histories. This is because Git needs to process more data and perform more calculations for the larger history.
- Memory usage: Git's memory usage can also be higher for repositories with large histories, especially during operations that require loading and processing a significant portion of the history.
- Disk I/O: Operations on repositories with large histories may result in more disk I/O compared to operations on repositories with smaller histories, due to the larger amount of data that needs to be read from and written to disk.

Overall, repositories with large histories can require more time and resources to perform Git operations compared to repositories with smaller histories. However, the exact impact on performance will depend on factors such as the size of the history, the number of files, the nature of the changes, and the capabilities of the system running Git.

## Strategies to manage large Git repositories with extensive histories

Managing large Git repositories with extensive histories can be challenging, but there are several strategies you can use to make the process more manageable:

- Split repositories: If parts of your repository are independent or have different access requirements, consider splitting them into separate repositories. This can make each repository smaller and more manageable.
- Baseline versus merge: Consider the trade-offs between maintaining a baseline branch (such as `master` or `main`) with a linear history versus using feature branches and merging them back into the main branch. A linear history can make it easier to track changes and understand the evolution of the codebase, but it may require more frequent rebasing and rewriting of history. On the other hand, using feature branches and merging them back into the main branch can make it easier to work on multiple features simultaneously and collaborate with other team members.

Tip:

To learn about recommended Git branching strategies for mainframe applications, check out our [Git branching model for mainframe development](#).

- Code review practices: Establish code review practices and guidelines to ensure that changes to the repository are reviewed and approved by team members before being merged into the main branch. Maintaining code quality in this way can help teams avoid regressions in the codebase, as well as the extra time and commit history that might come with fixing regressions later in the development cycle.



- Regular maintenance: Regularly review and clean up the repository, removing any unnecessary files and branches that are no longer needed.
- Squash commits: Squashing commits can also help reduce the size of the repository, as each commit contains metadata and changesets that take up space. By squashing commits when merging into the long-living protected branch, you maintain relevant commits in the Git history while avoiding the potential noise of intermediate commits, which can make operations such as cloning, fetching, and browsing the history faster and more efficient. Git provider interfaces typically have an option to facilitate squashing commits when merging a pull request.

By incorporating these additional strategies into your Git workflow, you can further improve the management of large repositories and enhance collaboration within your team.

## Defining dependency management

---

An IT system is developed by many teams and composed of different applications driven by the line of businesses and consumers. Applications need to interact to provide the overall system and interact through defined interfaces. Using well-defined interfaces allows the parts of the application to be worked on independently without necessarily requiring a change in other parts of the system. This application separation is visible and clear in a modern source code management (SCM) system, allowing clear identification of each of the applications. However, in most traditional library managers, the applications all share a set of common libraries, so it is much more difficult to create the isolation.

This page discusses ways to componentize mainframe applications so they can be separated and the boundaries made more easily visible.

### Layout of dependencies of a mainframe application

From a runtime perspective in z/OS<sup>®</sup>, programs run either independently (batch programs) or online in a middleware (CICS<sup>®</sup>, IMS) runtime environment. Programs can use messaging resources like MQ queues or data persistence in the form of database tables, or files. Programs can also call other programs. In z/OS, called programs can either be statically bound or use dynamic linking. If a COBOL program is the first program in a run unit, that COBOL program is the main program. Otherwise, the COBOL program and all other COBOL programs in the run unit are subprograms. The runtime environment involves various layers, including dependencies expressed between programs and resources or programs and subprograms.

There are multiple types of relationships to consider. The source files in the SCM produce the binaries that run on z/OS. To create the binaries, a set of source level dependencies must be understood. There is also a set of dependencies used during run time. These multiple levels of dependencies are defined in different ways, and in some cases not clearly defined at all. Understanding and finding the dependencies in source files is the first challenge.

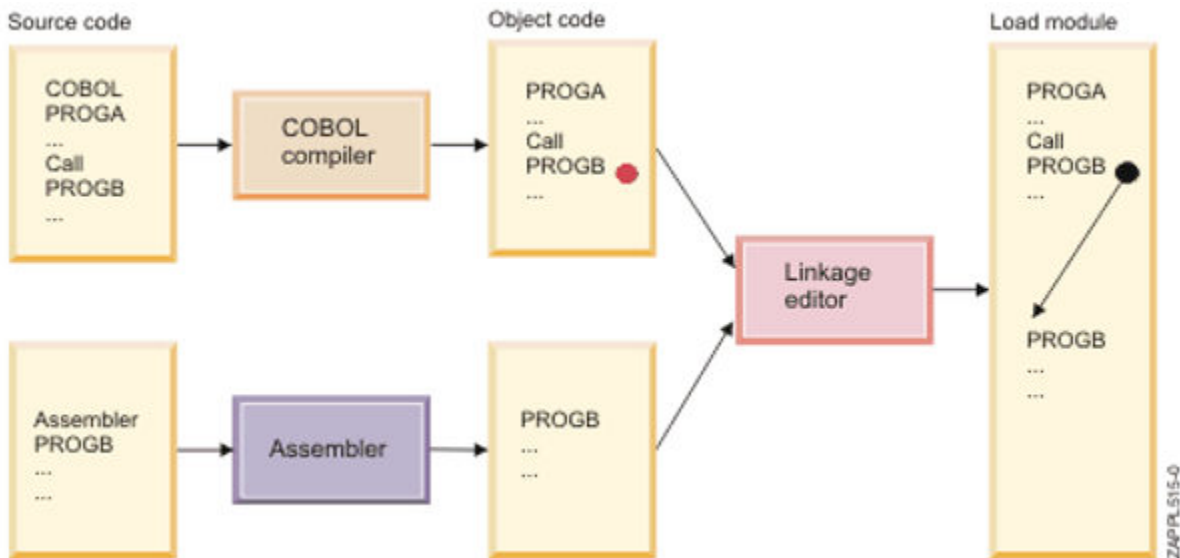
Building a program involves different steps:

1. Compilation including any pre-compilation steps, defined as explicit steps or as option of the compiler, creates a non-executable binary (object deck) file.
2. Link-edit, which assembles the object deck of the program with other objects and runtime libraries as appropriate. Link-edit can be driven by instructions (a link card) from the SCM or as dynamically defined in the build process.

These steps are summarized in the following diagram.

---

<sup>1</sup> See the "Using subprograms" chapter of the [Enterprise COBOL for z/OS documentation library programming guide](#).



As part of the traditional build process, some additional steps, such as binds to databases, are sometimes included. The function of these steps is to prepare the runtime for a given execution environment. These should not be included in the build process itself, but should instead be included in the deployment process.

## Source dependencies during the build differ from runtime dependencies

Most of the time when people think about an application, it is from a runtime point of view. Several components are required for an application to run. Some of these are required as dependencies, such as the database or middleware and its configuration, while others are required as related, such as other applications that might be called.

Everything running in a runtime environment starts as source from an SCM - or at least, this should be the goal when you consider infrastructure as code. Some source files represent definitions or are scripts that are not required to be built. Those that do require being built generally require other source files such as copybooks, but might not require the CICS definition, for example. Some of the source files are also included in many different programs - for example, a copybook can be used by many programs to represent the shared data structure. It is important to understand the relationships and dependencies between the source files, and when those relationships or dependencies have importance. The copybook is required to build the program, so it is required at compile time, but it is not used during run time. The configuration for a program such as the CICS transaction definition or the database schema is related to the application, but is required only for the runtime environment.

A concrete dependency is the interface description when calling a program. A copybook defines the data structure to pass parameters to a program. So, the copybook is important to be shared while the program is part of the implementation.

## Programs call each other either dynamically or statically

On z/OS, there are two ways programs are generally called: dynamically and statically. Statically called programs are linked together at build time. These dependencies must be tracked as part of the build process to ensure they are correctly assembled. For dynamic calls, the two programs are totally separate. The programs are built and link-edited separately. At runtime, the subprogram is called based on the library concatenation.

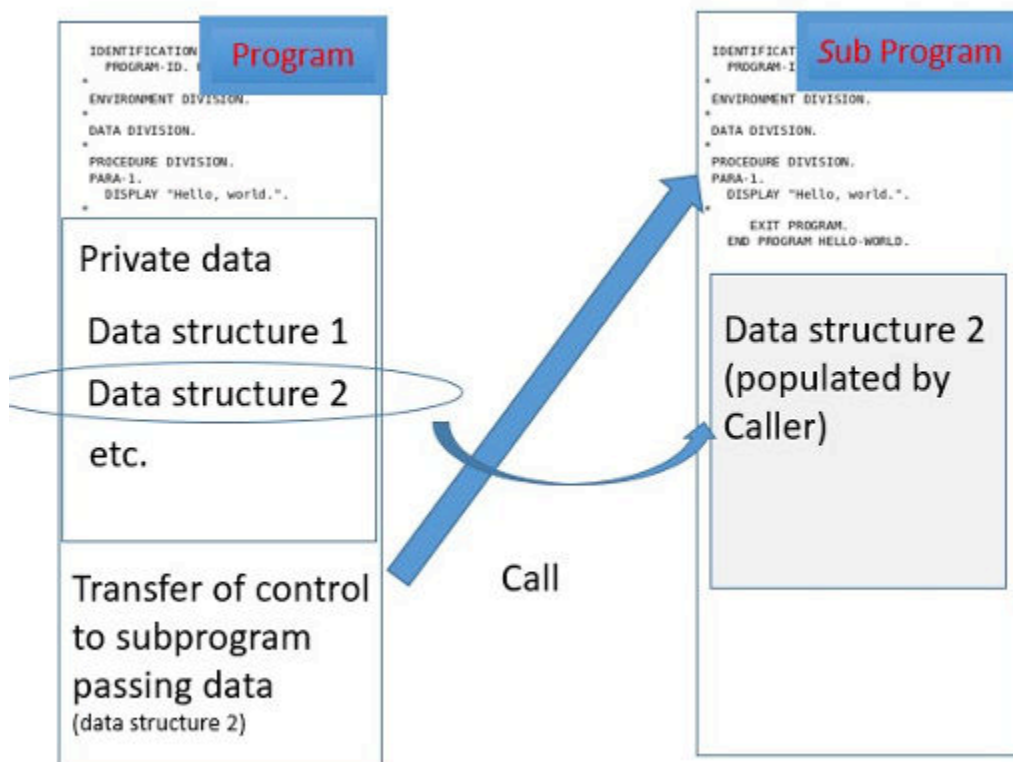
Many organizations have been moving to increased usage of dynamic calls as that approach reduces the complexity at build time. However, this approach means that the runtime dependencies need to be tracked and understood if any changes are made that require updates in both program and subprogram.

These programs and subprograms are interdependent even when using dynamic calls. When a program calls another program, generally they share data. A transfer of control occurs between the program and

the subprogram with the main program passing a set of data to the subprogram and generally expecting some data in response.

Different mechanisms exist to share pieces of data based on the language or the runtime. However, there is a need for the caller and the called program to define the data structure to be shared. The call of a subprogram is based on a list of transfer parameters, represented in the interface description like an application programming interface (API), but it is more tightly coupled than today's APIs that are based on representational state transfer (REST).

Commonly, shared data structure is defined in an included source file - for example, COBOL uses copybooks.



It is very common to define multiple copybooks for programs in order to isolate data structures and reuse them in other areas of an application component. Using copybooks allows more modularity at source level and facilitates dealing with private and shared data structures, or even private or shared functions.

## Applications and programs

In a web application, it is relatively easy to define an application because the artifact that is deployed is the complete representation of such an application: the EAR or WAR file. In the Windows world, it is more complicated since an application can be made of several executables and DLLs, but these are generally packaged together in an installable application or defined by a package manager.

An application is generally defined by the function or functions it provides. Sometimes there is a strong mapping between the parts that are shipped, and sometimes it is a set of parts that run the application.

In the mainframe, we fall closer to the second case where applications are defined by functions. However, based on the way the applications have grown over the years, there may be no clear boundary as to where one application ends and another one begins. An application can be defined by a set of resources (load modules, DBRMs, definitions) that belong together as they contribute to the same purpose: the calculation of health insurance policies, customer account management, and so on.

At the source file level, the relevant files contributing to an application are derived from the runtime of an application. These files can usually be identified by different means: a set of naming conventions, the

ownership, information stored in the SCM, etc. It may not seem obvious at first glance, but most of the time it is possible to define which source files contribute to a given application.

Scoping your source files to an application has many benefits. It formalizes the boundaries of the application, and therefore its interfaces; it allows you to define clear ownership; and it helps with the inventory of the portfolio of an organization. Planning of future features to implement should be more accurate based on this scoping.

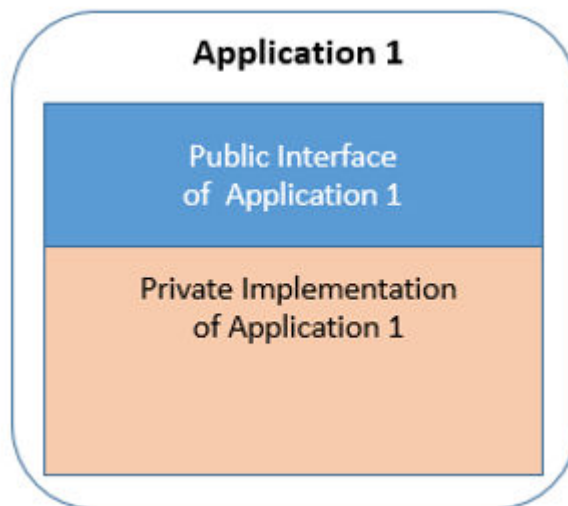
## Applications and application groups

Within an organization, multiple applications generally make up the business function. An insurance company may have applications dedicated to health insurance, car insurance, personal health, or group health policies. These applications may be managed by different teams, but they must interact. Teams must define the interfaces or contracts between the applications. Today, many of these interactions are tightly coupled with only a shared interface defining the relationship.

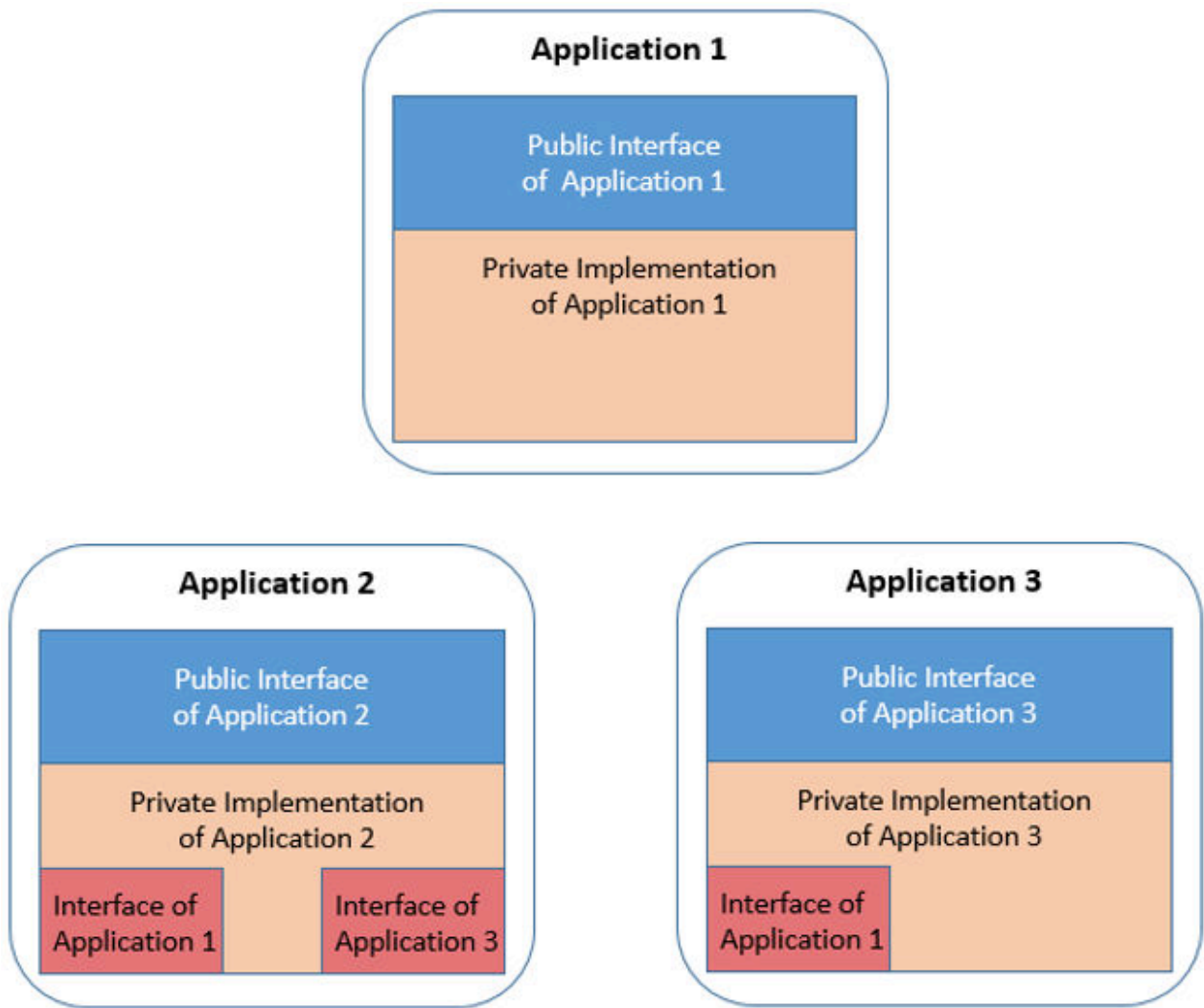
As we have seen so far, for traditional z/OS applications, the interface is not separate but defined in source via a shared interface definition, generally a copybook or include. This source must be included in each program build for them to be able to interact. With this information, an application can be defined by two main components: shared interfaces that are used to communicate with other programs and the actual implementation of the programs.

It is important to note that shared copybooks could be shared not only within an application, but also across programs or across applications. The only way other programs or applications can interact with the program is by including the shared interface definition. A z/OS load module does not work like a Java™ Archive (JAR) file, because it does not expose interface definitions.

The following diagram illustrates the concept of an application having a private implementation (its inner workings, represented in tan), and a public interface to interact with other programs and applications (represented in blue).

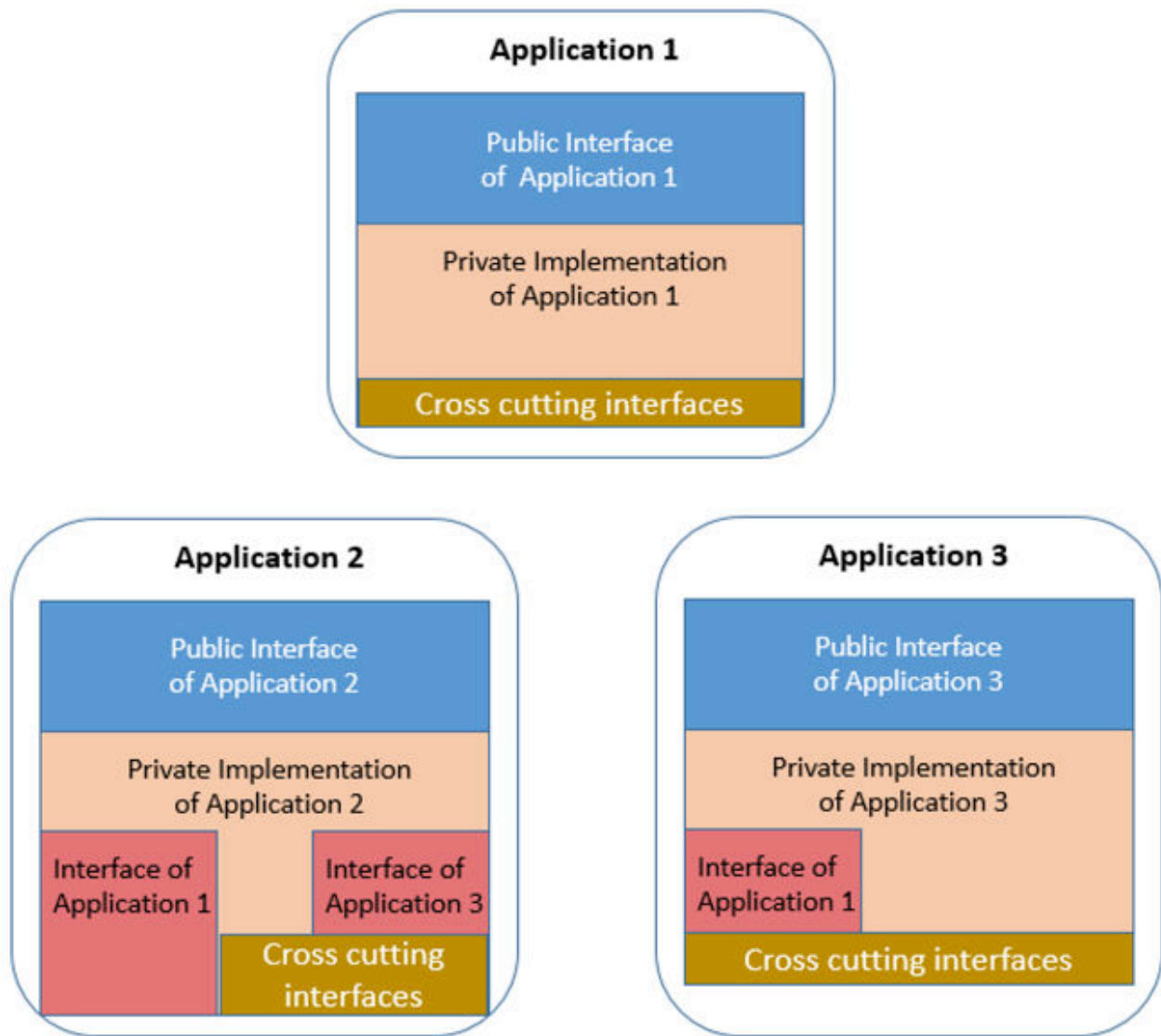


As applications communicate, their implementation consumes the public interface of the applications with which they interact. This concept of a public interface is common in Java programs and the way the communication between applications is defined. This principle can also be applied to existing COBOL and PL/I programs to help explain the structure required for a modern SCM, and is illustrated in the following diagram, with the applications' usage of other applications' interfaces indicated in red.



## Cross-cutting interfaces

There are additional capabilities that might need to be shared in addition to sets of data structures for application communication. These capabilities might include standard security or logging functions and can be considered cross-cutting, infrastructure-level interfaces (represented in brown in the following diagram). These capabilities may be developed once and then included in many different programs. It would be very helpful if these additional included capabilities could also be handled as shared components with their own application lifecycle. The challenge comes when these components change in a non-compatible way. These types of changes are generally infrequent but might be needed at times.



In the preceding sections, we have laid out some of the key factors when considering the source code of traditional mainframe applications. The environment generally consists of many different applications that can provide shared interfaces and could consume shared components, or cross-cutting interfaces.

The knowledge of these factors and their respective lifecycles can guide the desired structure of source files in the SCM. Several patterns are possible to provide appropriate isolation, but to also provide appropriate sharing based on different requirements.

## Managing and life cycle of shared interfaces

As discussed in previous sections, applications often provide interfaces and rely on other applications' interfaces to communicate. Each application has its own life cycle of changes. In this process, changes are made. The changes are stabilized, and then go to production. In the end, there is a single production execution environment where all the applications will run. The applications must integrate, but the most common breaking point is their interfaces. If an application's interface changes in a non-backward compatible manner and the other applications do not react to this change (to at least recompile the modules affected by the interface change), then the production application will break. Therefore, most changes are implemented in such a way so that the structure of the data that is shared is not changed, but instead filler fields are used, or data is added to the end where only applications who need to react to the change are required to respond. In discussing the management of shared interfaces, we will refer to the process of reacting to changes as "adoption", and the applications consuming the interface as "consumers".

The main types of changes that can be made to an interface are as follows:

- **Backward compatible change:** The interface changes, but the change is done such that all applications do not need to react. A typical example is when a field is added to a data structure, but this field does not have to be handled by most applications except a selected few, which requested this new field. Usually, fillers are used in a data structure and when people declare a new field, the overall structure of the data structure stays the same. Only the consumers interested in the new field need to adopt by using the new element in the interface. The others should use the new interface to stay current, but do not need to adopt it at this point. A standard practice is to adopt the change at least for those programs which are changing and recompiling for other reasons. Many organizations do not do this type of adoption until the change has made it to production, to be sure they will not be dependent on a change that gets pulled from the release late in the cycle.
- **Breaking change:** A data structure has changed in such a way that the overall memory layout of the data is impacted. It could be because of an array in a data structure. There are other cases, as well (such as changes in arrays or condition name changes also known as level 88). In these cases, all of the consumers need to rebuild, whether they have functional changes to their code or not.

Enterprises typically already have their own rules in place for how to manage cross-application dependencies and impacts. It is most common for shared interface modifications to be done in a backward-compatible manner, meaning a change of the interface does not break the communication. Or, if there would be a breaking change, the developers create a new copy rather than extending and breaking the existing interface. Thus, applications consuming that interface can continue to work even without adopting the new change. The only consumers that would need to rebuild would be those that will use the new feature (or other change that was made to the interface). This way of making backward-compatible changes depends on the developers' expertise in implementing them.

Especially with breaking changes, a process needs to be defined for planning and delivering application interfaces' changes. Traditionally, the adoption process is often handled by an audit report generally run late in the cycle. However, having early notification and understanding of these changes helps improve and speed delivery of business function. Understanding your enterprise's current rules for managing shared interfaces is important for being able to accommodate them when implementing build rules in your new CI/CD pipeline system.

## **Adoption strategies for shared interfaces**

Changes to public interfaces should be coordinated between the different applications. Various levels of coordination for the adoption process can be defined:

- **No process (in other words, immediate adoption):**
  - In this case, the application owning the modified interface shares it by publishing it to a shared file system (for example, the COPYLIB). The next time applications consuming that interface compile, they will get impacted by the changes immediately. This is the unfortunate reality of many organizations today. Developers can be hit by unexpected changes that break their code. To minimize the risk, developers start to develop and test in isolation before sharing the updated interfaces. With Git and DBB, they can develop in isolation, build within their isolation, and fully test before they integrate with the rest of the team.
  - Usually this scheme does not scale (that is, it works for small teams, or small sets of applications only) and slows down development teams as they grow. It can be applied for application internal interfaces without impacts to other applications.
- **Adoption at a planned date:**
  - The application providing the shared interface announces (for example, via a meeting, email, or planning tool) that they will introduce a change at a given date, when the consuming applications can pick it up. This allows the consuming applications to prepare for the change, and is part of the resource planning. An outline of the future version might also be made available with the announcement.
- **Adoption at own pace:**

- The providing application publishes the new version of the interface to a shared location, which could be a file system, a separated Git repository, or an artifact repository. The teams using the interface can then determine themselves when to accept the change into their consuming application(s), or if they want to work with an older version of the interface.

Whichever adoption method is selected, we need to ensure there is only one production runtime environment. On the source code level, applications need to consolidate and reconcile development activities to ensure changes are not accidentally picked up before being ready to move forward. The most common method for achieving this is to use an agreed-upon Git branching strategy to organize and manage development work feeding into the CI/CD pipeline.

## Pipeline build strategies for shared interfaces

When implementing rules for the build process in the CI/CD pipeline, different build strategies exist with different implications. There are two typical approaches you can consider:

- Automated build of all consumers: After changing and making a shared interface available to others, the build pipelines will automatically build the programs which include the copybook.
  - Example: When a shared copybook gets modified and made available to others, the build pipeline will automatically build programs that include that copybook.
  - Questions to consider with this approach:
    - Who owns the testing efforts to validate that the program logic did not break?
    - Who is responsible of packaging, staging and deploying the new executable? (This can only be the application owner.)
- Phased adoption of changed shared interfaces: If the modifications to the copybooks are backward-compatible, then consumers will incorporate the changes the next time they modify the programs. Thus, adoption of the external copybook change happens at different times.
  - Since consumers of the shared interface are not forced to rebuild with this approach, this means that while some production load modules might run with the latest copybook, others might run with older versions of the copybook. In other words, there will be multiple versions of this shared interface included in the load modules in production. As long as the copybooks are modified in a backward compatible way, this should not be a problem.

The build strategy you pick for shared interfaces should depend on what your current workflow is, and if you want to continue with a similar workflow, or change that flow. General strategies for designing different builds with DBB and zAppBuild are described in [Designing the build strategy](#).

For pipeline builds using DBB, zAppBuild's `impactBuild` capability can detect all changed files, such as modified copybooks or subroutines, and then based on that, it can find all the programs (in Git repositories) that are impacted by that change and build them. The most common use case for pipeline build automation with DBB is an automated impact build within an application (rather than between repositories). However, if you would like to force a rebuild of all consumers of a shared interface, including those in other repositories, then some considerations should be made to understand how the consumers include the external module, and subsequently, how to configure the build scope for external dependencies. Several scenarios for implementing specific build scopes depending on development practices and applications' requirements are described in the document [Managing the build scope in IBM® DBB builds with IBM zAppBuild](#).

## Resources

This page contains reformatted excerpts from [Develop Mainframe Software with OpenSource Source code managers and IBM Dependency Based Build](#).

## Designing the build strategy

---

The build step contains all the steps to compile and link the source files into executable code. This page focuses on the different build strategies to support build scenarios at different phases of application



development, and points to the relevant sections within the shared sample scripts via IBM®'s [zAppBuild repository](#).

There are several different types of possible builds. At a high level, there is a user build, which is a build done by a developer on a single program, and a pipeline build, which will build the single or multiple changes pushed to the Git server.

Due to the nature of mainframe languages, source files need to be associated to a build type. For the build process, it is important to know what type of processing the file needs so that the correct build process can be associated with it. The correlation can be tracked through a fixed mapping list, through a folder organization, or a file extension.

Within the zAppBuild samples on GitHub, a mapping in the file `file.properties` provides the association of the file to a certain build script.

## User build

In an early development phase, developers need the ability to build the single program they are working on, as well as the unit test programs being created.

The following integrated development environment (IDE) tools provide an option to enable the developer to compile a selected program in an easy and fast way, without the need to commit or push the change to the repository, while still using the same build options defined for the pipeline build. The purpose is to build fast for the personal testing scenario:

- [IBM Developer for z/OS® \(IDz\)](#)
- [Wazi for Visual Studio Code](#)
- [Wazi for Dev Spaces](#)

Additional information about performing a user build can be found in the documentation for [IBM Dependency Based Build](#) and for the IDEs listed above.

## Pipeline build

A pipeline build is generally a build of changes in one or more commits on a branch in the remote repo. It can be triggered automatically (for example, when developers push their feature branch to the remote repo, or when commits are merged to `main`), or it can be triggered manually (such as for release candidates). It produces the official binaries, outputs that can be packaged and deployed to different environments, including production. By having the clear definition of what went into each build and the official build outputs, this ensures there are audit records.








### Scope of pipeline builds

The continuous integration/continuous delivery (CI/CD) pipeline can kick off builds for many different scenarios, providing different overrides for each. The build scripts need to handle all these scenarios. Examples of the multiple possible build scenarios include a full build, a dependency build, and a scoped build.

A pipeline build is also possible for the short-lived topic (or feature) branches. However, the binaries resulting from this particular scenario cannot go to production because they typically lack a review and approval process.

### **Full build**

A full build compiles the full defined configuration. The full list of source files is provided as input from the pipeline to the build scripts. One strategy is to allow the specification of the build type in the pipeline orchestrator, as demonstrated in the following screenshot of a Jenkins sample pipeline. The build script would then need to handle this input as the build type.

-  Back to Dashboard
-  Status
-  Changes
-  Workspace
-  Build with Parameters
-  Delete Project
-  Configure

## Project Mortgage-Sample-App

This build requires parameters:

buildType  Please select the appropriate build type

### **Impact build**

Instead of a full build, most builds are dependency-based impact builds, only building the changed files and the files that depend on (or are impacted by) those changed files. However, there are different qualities of defining the scope of an impact build, which is reflected in the impact calculation phase of a build.

#### *Impact calculation for dependency-based builds*

The easiest strategy is to only build the modified files. How to calculate this depends on the scenario. If it is a topic branch build, this can simply be the modified files identified in the Git commit. If this is a release build, you can use a diff of the current baseline to the previous successful build baseline to calculate the modifications.

The next consideration is if you want to include the directly impacted files in the list of buildable files. This calculation for direct dependencies occurs on the source file level. If a copybook is changed, all files using this copybook are directly impacted. This level of impact calculation is managed by the IBM Dependency Based Build server. The server contains a collection representing all direct dependencies in a configuration. Your build script needs to query the dependency-based build server to retrieve the list of impacted files and pass this along to the build script. The use cases can vary. If you want to automatically rebuild programs including a modified copybook, you should pursue the strategy above. Instead of rebuilding all dependent programs, you might want to provide a function for the user to select which, if any, of these impacted files should be rebuilt.

The `ImpactUtilities.groovy` sample script for IBM Dependency Based Build in the public [zAppBuild GitHub repository](#) provides an example of this impact calculation.

### **User-defined build scope**

While dependency-based impact builds cover strategies with a good level of automation for the build process, the user-defined build scope provides full flexibility to the user. It is up to the user to provide the list of files being considered in the build. The list could also reference a work item and its related changes. As a variation, the list can include the files from where the impact calculation should be performed.

A file containing the build list is stored as part of the repository.

```
1 MortgageApplication/bms/epsmlis.bms
2 MortgageApplication/bms/epsmort.bms
3 MortgageApplication/cobol/epsmlist.cbl
4 MortgageApplication/cobol/epsmpmt.cbl
5 MortgageApplication/cobol/epsnbrv1.cbl
6 MortgageApplication/cobol_cics/epsdsmrd.cbl
7 MortgageApplication/cobol_cics/epsdsmrt.cbl
8 MortgageApplication/cobol_cics_db2/epscmort.cbl
9 MortgageApplication/link/epsmlist.lnk
10 MortgageApplication/mfs/dfsiv1.mfs
11 MortgageApplication/copybook/epsmortf.cpy
12 MortgageApplication/copybook/epsmtcom.cpy
```

Similar to the build scripts, we recommend storing the user-defined build list as part of the Git repository. Additionally, you need to consider a parameterized Jenkins job including the relevant logic, which can serve all three build types: full build, dependency-based impact build, and user-defined build.

## Strategies for orchestrated builds

As we now have shown, there are different possible build scenarios. It is important to select the right build types for your setup. Using a mixture, depending on the level, should serve your needs. At each given level the choices will be limited, with the most flexibility at the development level. If you intend to maintain the traditional library population strategy, you can implement user builds to build development activities within the development stage and set up a dependency-based impact build in the integration stage to ensure consistency across the application.

It is important to recognize that by moving to Git and using an artifact repository for outputs, you are no longer tied to a library structure. You can now use the same build configuration strategy while pulling any already-built parts out of the artifact repository.

## Resources

This page contains reformatted excerpts from [Develop Mainframe Software with OpenSource Source code managers and IBM Dependency Based Build](#).

## Architecting the pipeline strategy

---

### Building, packaging, and deploying in a pipeline

A continuous integration/continuous delivery (CI/CD) pipeline removes the manual tasks of defining a package. This task is now fully automated and occurs once the application is built.

It is typical to have multiple packages, with each package being expected to have passed automated quality gate testing. Not all packages will make it to production due to discovered defects.

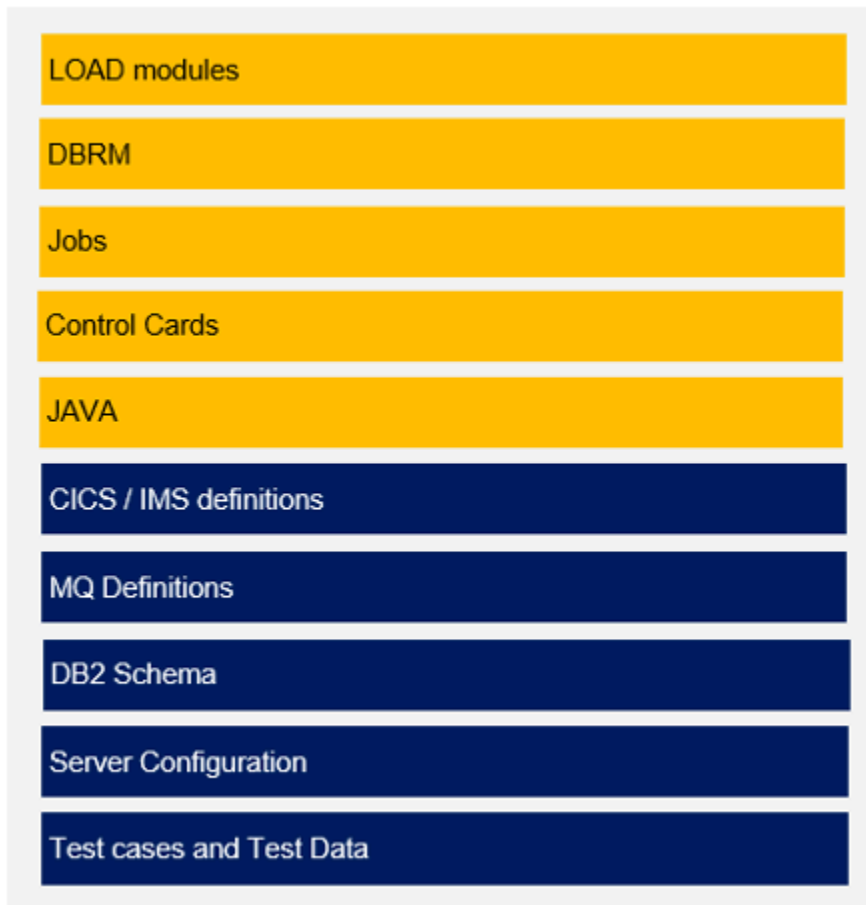
Because the creation of a package most likely occurs at the application component level, the source code management (SCM) layout also has an impact on the packaging.

Technically, a package is mainly composed of a list of binaries or files. However, it also includes metadata for the deployment manager. Using the metadata and a defined process, the deployment manager knows how to treat each element of the package during rollout.

It is important, especially for build outputs on mainframes, to keep track of each's element type. For example, during the deployment phase, a CICS® program will require different handling than a plain batch program. The same applies for programs using Db2®. Therefore, the metadata associated with each artifact should provide a "deploy type". This type indicates the nature of the artifact and is used to drive the appropriate process when deploying the artifact.

## Package content and layout

It is imperative that you consider build outputs as part of your package. Examples of these items are highlighted in yellow in the following image. However, it is equally important to also consider the application-specific configuration artifacts (items highlighted in blue). Doing so will help teams avoid a flood of change requests, limit constant back and forth communication, and will enable the continued use of deployment automation. Application-specific configurations should be treated as source code, in the same way you manage your application source code (although, not everything needs to end up in a single package; we can also consider configuration and application packages). The following image shows a list of package items that are typical in the mainframe domain.

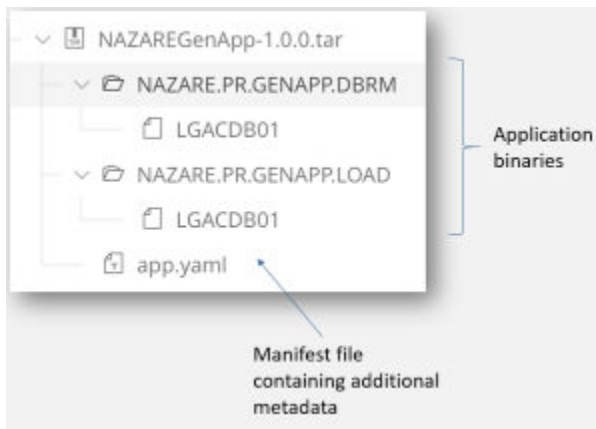


The package is represented in an archive format such as `.tar` (common in the UNIX world). This format is consistent with non-mainframe applications, where teams usually work with full application packages in archives such as a WAR or a JAR file.

In all cases, the package consists of two items:

- The actual binaries and files produced by the build manager
- A manifest describing the package's contents (that is, metadata)

For mainframe applications, a package will contain executables required to run your application, such as program objects, DBRM, JCL, and control cards – as well as a manifest file. An example of package contents in a typical mainframe application package is shown in the following image.



**Tip:**

The package can also carry listings or debug files which you use during debugging. By doing so, you ensure that your listing files' version matches the version of your binary.

## Package strategy and scope

Package creation occurs after a build. Here, binaries are created, but other files are also produced during the build process. The build process takes inputs from source files stored in one or several Git repositories. Usually, when several repositories are involved, one repository will be responsible for providing the parts to build (that is, programs), while the other repositories provide additional files (for example, shared copybooks). The scope of the build, derived from the scope of the main Git repository used during the build, defines the possible content of the package.

We need to distinguish between a "full package" – containing all executables and configuration files belonging to an application component – and a "partial package" – containing just the updated executables and configurations. You might be familiar with "incremental packages"; oftentimes, this term may be used interchangeably with the partial package term.

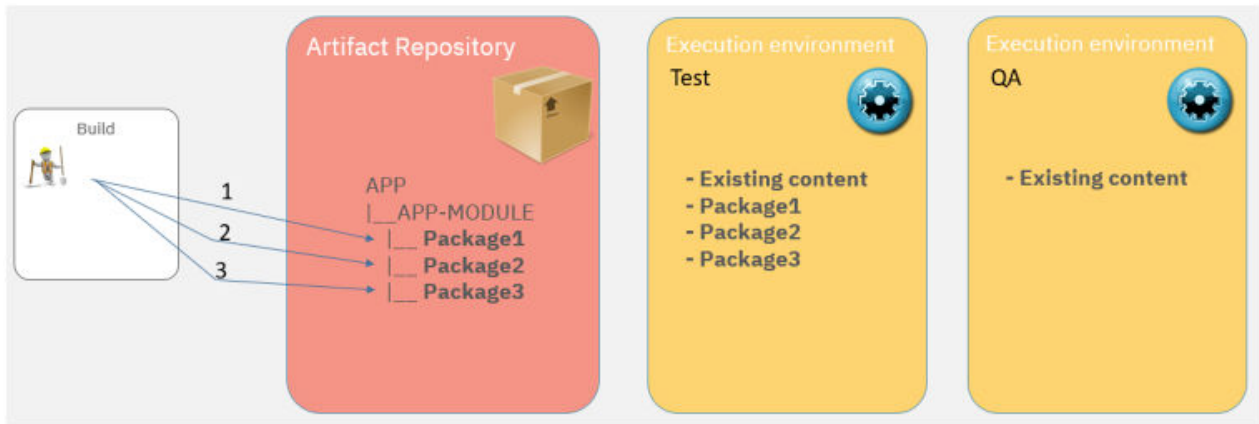
Partial packages can be divided into two types:

- "Delta packages" are produced immediately after each build
- "Cumulative packages" include outputs produced by several builds.

Mainframe applications typically work with incremental updates of the execution environment using partial packages.

As you might have seen already, there are many factors you will need to consider when selecting your strategy. As a practical exercise, we will walk through the following mainframe scenario using delta packages. In this scenario, summarized in the following figure, there are several execution environments in our system, and they are not updated at the same pace.

- The build is incremental and produces a partial package containing the newly built artifacts.
- The CI/CD pipeline automatically deploys the package produced by the build in a Test environment. We might see some tests failing, so developers iterate.
- The build runs three times. It produces three packages. Each of them is deployed in Test. However, QA is not updated yet.
- The next phase of tests is performed in the QA environment, when a first level of validation has occurred in Test.



Currently, most mainframe development practices only work within a static and fixed set of known execution environments. Introducing a new execution environment is, today, a significant task in mainframe shops. It prohibits provisioning of test environments for specific projects and/or sprints; a requirement for most agile teams.

In contrast, when provisioning capabilities are applied to your mainframe CI/CD pipeline, it becomes possible to make execution environments available more dynamically. A new test environment could be provisioned and used on-demand, and then recycled when the feature's test has been completed.

You may have noticed that the deployment pipeline has several types of execution environments to manage. At times, you will encounter long running, existing environments (that is, those that are updated but not at the same time), and also “fresh” environments - those that are either empty or need to be significantly updated.

In the first case (existing execution environments), we see that the environment is updated regularly by the deployment manager. Here it is easy and convenient to work with increments as each increment is deployed.

In the second case (“fresh” execution environments), the environment is updated less frequently or sometimes later in the CI/CD process. In this case, working with increments now requires an understanding of the last deployed increment, and the retrieval and the deployment of all increments until the desired state is achieved (in our example, package 1, 2, and 3). In some cases, packages will overlap, although the deployment manager might be smart enough to deploy only the latest version of an artifact.

In that latter case (“fresh” execution environments), working with partial packages is even more challenging. We miss out on more than just the latest packages; we miss out on some (if not all) significant history too. Thus, it becomes useful to maintain the complete content of the application, along with complete configuration scripts (for example, CICS declarations, full DDL). If a sound knowledge of the environment's inventory is maintained, then as we deploy, it will be possible to correctly calculate and apply the delta.

We can now examine the two packaging options that we have identified and relate them to the different IBM® Dependency Based Build types:

- Strategy: Partial packages
  - Contains the outputs of a DBB impact build. The build has identified the impacted elements by querying the collection in the IBM Dependency Based Build server.
- Strategy: Full packages
  - Contains all the outputs of an application configuration. The content either needs to be determined through a full build, a simulation build, or an application descriptor. The application descriptor defines the entire scope of an application and its outputs.

## Specialty of Mainframe Packages

Due to the nature of mainframe deployments, there is a need to capture additional metadata, such as the type of the object, for each binary in the package. We call this type of metadata the “deploy type”. It gives explicit criteria to follow a series of steps that are appropriate for the artifact to deploy.

There is thus a need for a manifest file. This file describes the contents of the application package and adds meta information to each of its artifacts. This information will then also be used by the deployment manager.

Additionally, the manifest file captures traceability information about the configuration used to create the package - in particular, a Git hash to trace back to the actual version of the source code. The manifest file will also capture the package type: full or partial.

The limits of which environment a package may or may not go is another piece of meta-information that the manifest of a package should contain.

The format of the manifest is more of a secondary consideration: it can be .yaml, .json, .xml, and so on. Considering the direction of containers with Kubernetes using Helm charts and OpenShift templates using .yaml, using .yaml for the metadata will make it more consistent with other industry work and make it clearly understandable by z/OS® and non z/OS developers. The following image shows a sample schema of an application package manifest.

```
<Package Name>
|_ build reference
|_ packageType (full, incremental)
|_ git-hash (the state of the branch)
|_ contents
| Container
  PGM          TYPE          PARMS
  CICD.APP.FEAT1.LOAD  PGM001  BATCH_DB2
  CICD.APP.FEAT1.LOAD  PGM002  BATCH_DB2
  CICD.APP.FEAT1.LOAD  PGM003  BATCH
  CICD.APP.FEAT1.LOAD  PGM004  BATCH
  CICD.APP.FEAT1.LOAD  PGM005  CICS
  CICD.APP.FEAT1.LOAD  PGM006  CICS_DB2
  CICD.APP.FEAT1.LOAD  PGM100  BATCH
  CICD.APP.FEAT1.DBRM  PGM001  DBRM
  CICD.APP.FEAT1.DBRM  PGM002  DBRM
  CICD.APP.FEAT1.DBRM  PGM006  DBRM
```

## Resources

This page contains reformatted excerpts from [Packaging and Deployment Strategies in an Open and Modern CI/CD Pipeline focusing on Mainframe Software Development](#).

## Outlining the deployment strategy

---

### Introducing deployment managers

Deployment managers support the automated application rollout of the application binaries and of the application configurations across platforms (that is, managing both mainframe and other systems). Typically, a deployment manager would perform the following tasks when deploying a package into an environment:

- Verify that the package is applicable to the environment

- Sort and order the artifacts to process
- For each artifact of a given type, run the appropriate process
- Perform post deployment activities

Approval processes may be embedded. This approach will likely be necessary due to audit and legal requirements.

Deployment managers are aware of the execution environments. They generally also provide an inventory of the packages and of the current deployed binaries in a given environment. The deployment process can be implemented with scripts or with other more sophisticated techniques. These more sophisticated processes manage return codes and facilitate the use of the APIs of the different middleware systems. However, in both cases, and in order to install the package, the deployment process will consume the manifest file, which contains metadata about contents of the package.

## Resources

This page contains reformatted excerpts from [Packaging and Deployment Strategies in an Open and Modern CI/CD Pipeline focusing on Mainframe Software Development.](#)



---

# Chapter 6. Migrating data from z/OS to Git

---

## Migrating data from z/OS to Git

---

### Introduction to migrating data

Migrating your source code data from z/OS® to Git is one of the first steps in implementing a modern CI/CD pipeline. Once you have planned your to-be Git repository layouts, you can use the steps in the following section to migrate your data from z/OS to Git. During the migration process, it is important to be aware of potential issues with migrating non-printable or non-roundtrippable characters, and to have a plan for how to handle them if you encounter them. You can read more about how to handle these characters in [Managing code page conversion](#).

### Steps for migrating data from z/OS to Git

At a high level, the steps for migrating data from z/OS to Git are as follows:

1. Unload the source code from the legacy version control system to PDS libraries: Legacy version control systems often store data in a proprietary format, which IBM®'s Dependency Based Build (DBB) cannot directly access. Therefore, the source code should first be unloaded to PDS libraries.
2. Load files from PDS libraries to a Git repository on z/OS UNIX System Services (z/OS UNIX) using one of the following methods:
  - [DBB migration tool](#): The DBB migration tool is provided with DBB, and is the most commonly-used method for migrating data from z/OS to Git.
  - [SCLM-to-Git migration tool](#): You can use the SCLM-to-Git migration tool if you are moving away from SCLM as your current library manager.
  - [Manual migration](#): Manual migration of source code from z/OS to Git is possible, but is generally not recommended since it tends to be slower, more tedious, and prone to human error.
3. Synchronize the new repository on z/OS UNIX with a remote repository on your chosen enterprise Git service provider.

---

## Managing code page conversion

---

### Introduction

Choosing Git to manage enterprise-level software assets implies a migration task of the z/OS® artifacts. Whether the source code is under the control of a legacy SCM solution or not, these artifacts are typically stored as members within partitioned data sets (PDSs) with a specific encoding. Depending on locale, country languages, regulations or work habits, the encoding of the source code can differ by various specificities such as accents, locale typography and characters. To support these peculiarities, many different EBCDIC code pages were developed in z/OS and are now widely used.

In the Git world, the standard encoding is UTF-8, which supports most (if not all) of the characters used in printing worldwide. When migrating artifacts from z/OS to Git, a translation must occur to convert the EBCDIC-encoded content of source files to UTF-8. This process is transparent to the user, whereby the translation is performed by Git, as long as the necessary information on how to perform the conversion is provided. Git uses a specific file called `.gitattributes` to describe and dictate the translation behavior. This page contains details on the encoding that should be used when hosting the artifacts on z/OS or on Git repositories.

Here we describe some specific migration situations and guidance that must be taken into account in order to ensure a successful migration of z/OS artifacts to Git. This page cannot cover all the country-

related specifics, nor the complex encodings found worldwide. Rather it is aimed at presenting general use cases and how to prevent/circumvent potential pitfalls.

## Understanding the code page translation process

Code page translation has always been a challenge since z/OS is exchanging data with the non-EBCDIC systems. To completely understand the effect of encoding, it is important to take a step back and analyze the binary content of files. This analysis will help to better visualize the transformation process that is occurring behind the scenes

### Looking inside a file

A file itself can be summarized as being a series of bytes, often expressed in the hexadecimal notation (from x'00' to x'FF'). This file can represent binary content which is not meant to be human-readable (like an executable file), or text content. For the latter and from a user standpoint, this series of bytes doesn't mean anything if the content is not associated with a code page, which will link each byte to a printable character (or a control character). Printable characters are the formation of a character set.

The combination of the contents of a file and the code page used defines the human-readable content. That said, the contents of a file will be displayed slightly differently when combined with different code pages. Let's explore a couple of examples using the ISPF Editor with the "HEX VERT" option enabled to show the hexadecimal values of a file.

Consider the following example where a file containing the French sentence "Bonjour à tous !" (which translates to "Hello everyone!") is displayed with two different code pages. The file which contains this sentence was saved using IBM-1147 encoding. When the file is then displayed using a different encoding of IBM-1047, although the contents of the file is the same from a binary standpoint, some characters are displayed in a different way:

- Using the IBM-1147 code page:

```
Bonjour à tous !  
C9999A9474A9AA44  
26516490C036420F
```

- Using the IBM-1047 code page:

```
Bonjour @ tous |  
C9999A9474A9AA44  
26516490C036420F
```

To be correctly displayed with the IBM-1047 code page, the contents of this file must be transformed. Please note how the hexadecimal codes for the à and the ! characters changed, respectively from x'7C' to x'44' and from x'4F' to x'5A':

- Transforming to the IBM-1047 code page:

```
Bonjour à tous !  
C9999A9444A9AA45  
265164904036420A
```

It is important to understand that the code page plays a determining role not only when displaying a file, but also when editing it. To ensure the content of a file is consistent when used by different people, the code page used for editing and displaying will likely be the same for all the users. If Alice edits a file with the IBM-1147 code page and introduces characters (like accents) specific to that code page, then Bob will need to use the IBM-1147 code page to display the content of this file. Otherwise, he may experience the situation depicted earlier, where accents are not displayed correctly. If Bob uses the IBM-1047 code page

to edit the file and change the content to replace the special characters by the corresponding accents, then Alice will not be able to display the content of this file like she entered it on her terminal.

Consider this next example. A program file, such as C/370, employs the left ([]) and right (]) brackets in some of the language statement constructs. In many US-based legacy applications, the original encoding used to create source files was IBM-037. If these files are then displayed using a different encoding of IBM-1047, again, although the content of the files is the same from a binary standpoint, the brackets are not displayed the correct way:

- Using the IBM-037 code page:

```
void main(int argc, char *argv[])
A98849889489A4898864888945898ABB5
569404195D95301973B038190C1975ABD
```

- Using the IBM-1047 code page:

```
void main(int argc, char *argvŸ")
A98849889489A4898864888945898ABB5
569404195D95301973B038190C1975ABD
```

To be correctly displayed with the IBM-1047 code page, the contents of this file must be transformed. The hexadecimal codes for the [ and the ] characters must be changed from x'BA' and x'BB' to x'AD' and x'BD', respectively:

- Transforming to the IBM-1047 code page:

```
void main(int argc, char *argv[])
A98849889489A4898864888945898AAB5
569404195D95301973B038190C1975DDD
```

Again, it is very important that anyone and everyone who displays or edits the file uses a consistent code page. This can sometimes be a challenge, as the code page to be used is generally specified in the 3270 Emulator (TN3270) client session set-up. Another challenge is trying to determine the original encoding used to create the file.

To summarize, the binary content of a file must be transformed to ensure consistency when displayed with another code page. This process is known as the code page translation and is key when migrating your source from your z/OS platform to a non-EBCDIC platform, which is using different code pages (and most likely today, UTF-8).

## Vocabulary

In this document, we will interchangeably use the coded character set ID (CCSID), its equivalent name, and the code page to designate an encoding. Although not strictly equivalent, they are all often interchanged for the sake of convenience. For instance, the IBM-1047 code page is equivalent to the 1047 coded character set (CCSID) and is usually named IBM-1047. The code page for IBM-037 is equivalent to coded character set (CCSID) 037 and is usually named IBM-037. The CCSID for UTF-8 is 1208 and is linked with many code pages.

To simplify, the code pages will be designated by their common name, for instance IBM-037, IBM-1047, IBM-1147, and UTF-8.

For reference, a list of code pages is available in the [Personal Communications documentation site](#).

## The challenge of migrating to UTF-8

If you have ever tried transferring a z/OS data set to a machine running an operating system such as Microsoft Windows or Linux®, you probably ended up with similar content when opening the file:



The content is unreadable because the transfer was performed without a code page conversion. This generally happens when the file was transferred as binary, which leaves the content of the file untouched. Most of the transfer utilities offer the capability to transfer the files as binary or as text. In the latter option, code page conversion occurs, and the file should be readable on non-EBCDIC platforms.

The same conversion must be performed for text files when migrating to Git. Generally, Git uses the UTF-8 code page and, as such, a translation must occur on the content of the transferred files. Likewise, when bringing the files back from Git to z/OS, a backward conversion must be performed to ensure the files are correctly read with the code page in use. Failure in performing this conversion may break the contents of the files, which could become damaged and unreadable. Recovering from this failure is difficult, and the best solution is usually to restart the conversion all over again from the beginning, which often means deleting the files in Git and transferring the same files from z/OS again.

## Managing non-printable and non-roundtripable characters

In the traditional EBCDIC code pages, some characters exist to control the different devices themselves or the way text should be displayed. These characters are located in the first 64 positions of the EBCDIC characters tables and can be classified into 2 categories: the non-printable characters and the non-roundtripable characters.

- Non-printable characters: These characters can be translated from EBCDIC to UTF-8 and back to EBCDIC without breaking the content of the file. However, editing files which contain these characters with editing tools on a UTF-8 platform, can cause display issues. Non-printable characters include all EBCDIC characters below 0x40.
- Non-roundtripable characters: The translation is still possible from EBCDIC to UTF-8, but the translation back to EBCDIC is not possible. In this case, the translated file is no longer identical to its original version. This specific situation must be considered and managed prior to the migration to Git. For these special characters, the conversion from EBCDIC to UTF-8 is generally feasible, but the resulting content of the files may be scrambled or displayed in a wrong way. Moreover, when transferring the files back to z/OS, the files could be damaged, and the transfer could even fail. Non-roundtripable characters include the following:
  - EBCDIC
    - Newline: 0x15
    - Carriage return: 0x0D
    - Line feed: 0x25
    - SHIFT\_IN/SHIFT\_OUT: 0x0F & 0x0E

In a migration process, when either non-printable or non-roundtripable characters are found, two options are presented:

- Keep these characters “as-is” with the risk of damaging the source files.
- Modify the characters to a conversion-friendly format.

For the first option, it is possible to keep the content of the files intact, but the cost of this is to manage the file as a binary artifact in Git. The benefit of this configuration is that the content of the file is not altered by Git when the file is transferred. Because it is flagged as binary, the contents are not converted to UTF-8 and remain the same across the entire workflow (that is, from z/OS PDS to Git to z/OS PDS). The major drawback of this method is that the modification of these binary files with a non-EBCDIC-aware tool, such as an integrated development environment (IDE) or Notepad, will be extremely difficult. As these files were not converted to UTF-8, their contents will still be in EBCDIC, which is rarely supported outside of the z/OS platform. Browsing or editing the content of these files would require to bring them back to the z/OS platform. Still, this option could be a valid strategy for files that rarely (or never) change, but this implies setting up a strategy when a modification is required (that is, these files will need to be modified on z/OS, rather than the non-EBCDIC-aware tool - this strategy should remain as an exception).

The alternative is to opt for a transformation of the content that contains non-printable or non-roundtripable characters before migrating the files. Programmers have often employed elaborate strategies based on the use of these characters in literals or constants. Through the ISPF editor, developers have been able to insert hexadecimal values (when enabling the editing of raw data with the hex on feature) in their source code. This use case is very common and can easily be corrected by replacing the hexadecimal coded characters with their corresponding hexadecimal values as shown in the picture below (taken from a CICS® COBOL copybook):

- Initial values:

```
30      01      DFHBMSCA.
31      02      DFHBMPER PICTURE X VALUE IS '0'.
32      02      DFHBMPNL PICTURE X VALUE IS ' '.
33      02      DFHBMPFF PICTURE X VALUE IS '0'.
34      02      DFHBMPER PICTURE X VALUE IS ' '.
35      02      DFHBMASK PICTURE X VALUE IS '0'.
36      02      DFHBMUNP PICTURE X VALUE IS ' '.
```

- Final values after transformation:

```
34      01      DFHBMSCA.
35      02      DFHBMPER PICTURE X VALUE IS X'19'.
36      02      DFHBMPNL PICTURE X VALUE IS X'15'.
37      02      DFHBMPFF PICTURE X VALUE IS X'0C'.
38      02      DFHBMPER PICTURE X VALUE IS X'0D'.
39      02      DFHBMASK PICTURE X VALUE IS '0'.
40      02      DFHBMUNP PICTURE X VALUE IS ' '.
```

This transformation could even be automated (albeit cautiously) through scripting. Other use cases for these characters should be analyzed carefully, and developers should be encouraged to write their source code in a way that allows for code page conversion.

Fortunately, IBM® Dependency Based Build (DBB) provides a feature in its migration script to detect these special characters and helps manage their code page conversion or their transfer as binary.

In any case, the decision about using binary-tagged files in Git, refactoring these files to transform the non-printable and non-roundtripable characters into their hex values, or not changing the content of the files should be taken prior to performing the final migration from datasets to Git repositories. If the migration is started with files that contain either non-printable or non-roundtripable characters, there is a high risk that files cannot be edited using other editors. Once the files are migrated, it is often very difficult to resolve the situation after the fact, as there can be information lost during the code page conversion. In that situation, the best option is to restart the migration from datasets, assuming the original members are still available until the migration is validated.

## Defining the code page of files in Git

Since Git is a distributed system, each member of this “network” should be able to work with the files stored in the Git repositories. Over the last decade, Git has become the de facto standard for the development community, being used in all-sized enterprises, internal, open-source and/or personal development projects. Because Git emerged in the Linux world, it was first designed to support technologies from this ecosystem, and it all started with the file encoding. Although it first appeared in the 90’s, UTF-8 took off in the 2010’s through the increased interconnectivity between heterogeneous systems. UTF-8 is now recognized as the universal, common language for exchanging data and files. Naturally, UTF-8 also became the standard for encoding source files for the non-EBCDIC world.

Since mainframe source resides on the host side under the control of a Source Code Management solution, this source is encoded with an EBCDIC code page and could potentially contain national characters. In order to store these source files in Git, a code page conversion must occur between the z/OS source file EBCDIC code page and UTF-8. Git for z/OS will perform that conversion, which is transparent for the developers.

However, Git for z/OS must be instructed on the conversion operation to perform. These instructions are defined in a specific file that belongs to the Git repository. The `.gitattributes` file is a standard file used by Git to declare specific attributes for the Git Repository that it belongs to. With Git for z/OS, it has been extended to describe the code page that should be used for files when they are stored in Git and on z/OS UNIX System Services (z/OS UNIX).

The structure of the file is straightforward. One or several attributes are associated to each path and file extension listed in the file. Wildcards can be used for generic definitions. This file must be encoded in ASCII (ISO-8859-1), as it is processed by Git on both z/OS and other operating systems. Here is an example of such a file, which uses two parameters: `zos-working-tree-encoding` and `git-encoding`:

```
# line endings
* text=auto eol=lf

# file encodings
*.cpy zos-working-tree-encoding=ibm-1047 git-encoding=utf-8
*.cbl zos-working-tree-encoding=ibm-1047 git-encoding=utf-8
*.bms zos-working-tree-encoding=ibm-1047 git-encoding=utf-8
*.pli zos-working-tree-encoding=ibm-1047 git-encoding=utf-8
*.mfs zos-working-tree-encoding=ibm-1047 git-encoding=utf-8
*.bnd zos-working-tree-encoding=ibm-1047 git-encoding=utf-8
*.lnk zos-working-tree-encoding=ibm-1047 git-encoding=utf-8
*.txt zos-working-tree-encoding=ibm-1047 git-encoding=utf-8
*.groovy zos-working-tree-encoding=ibm-1047 git-encoding=utf-8
*.sh zos-working-tree-encoding=ibm-1047 git-encoding=utf-8
*.properties zos-working-tree-encoding=ibm-1047 git-encoding=utf-8
*.asm zos-working-tree-encoding=ibm-1047 git-encoding=utf-8
*.jcl zos-working-tree-encoding=ibm-1047 git-encoding=utf-8
*.mac zos-working-tree-encoding=ibm-1047 git-encoding=utf-8
*.json zos-working-tree-encoding=utf-8 git-encoding=utf-8
```

The `zos-working-tree-encoding` parameter specifies the code page used for encoding files on z/OS UNIX and in PDSs. It must be consistent with file tags for the files that are under the control of Git. The `git-encoding` parameter specifies the encoding for files stored in Git outside of the z/OS platform. The typical value for this parameter is UTF-8.

This file is used by Git for z/OS to understand which conversion process must be performed when files are added to Git and when files are transferred between a Git repository and the z/OS platform. This file plays an important role in the migration process of source files from PDSs to Git repositories, and its content must be thoroughly planned.

An example for this file can be found in the [dbb-zappbuild repository](#). This sample contains the major definitions that are typically found in the context of a migration project.

## Managing the code page on z/OS UNIX System Services

During the migration of source code files from z/OS PDSs to Git, a transitional state exists as the source members are copied to z/OS Unix System Services (z/OS UNIX). This step is important and must be correctly performed to ensure the files are imported into Git in the right way.

z/OS UNIX supports the use of different code pages for the files stored on zFS filesystems. As z/OS UNIX introduced the use of ASCII in the z/OS environment, there are some concepts to understand before diving into this section. These concepts are grouped under the term “Enhanced ASCII” in the [z/OS documentation](#).

Files on z/OS UNIX are just a sequence of bytes, like members of PDSs. When members are copied from PDSs to files on z/OS UNIX, the raw content is not altered, and the byte sequence doesn’t change. However, since z/OS UNIX can deal with files coming from non-EBCDIC systems, which are often encoded in ASCII, additional parameters are introduced to facilitate the management of the file encoding.

### File tagging

One of the additional parameters is the tagging of files to help determine the nature of the content. By default, files are not tagged on z/OS UNIX, and their content is assumed to be encoded in IBM-1047 by most of the z/OS utilities, unless stated otherwise. If the original content of those files is created with a different code page, some characters may not be rendered or read by programs properly.

A required step to ensure that files are correctly read and processed is to tag them. The tagging of files on z/OS UNIX is controlled by the `chtag` command. With `chtag`, you can print the current tagging information for a file (`-p` option, or the `ls -T` command also displays the tag of listed files) or change this tagging (`-t/-c/-b/-m` options). It is important to understand that this tagging information is then used during the migration process into a remote repository by Git for z/OS, which uses that information to correctly convert the file to the standard Git encoding of UTF-8. Having the correct tagging set for files on z/OS UNIX is a major step in the migration process to ensure a successful conversion.

The following example shows the output of the `ls -aLT` command for a folder that contains COBOL source code:

```
ls -aLT
total 448
          drwxr-xr-x  2 USER1  OMVS      8192 Feb 22 14:47 .
          drwxr-xr-x  7 USER1  OMVS      8192 Jan 12 14:06 ..
t IBM-1047  T=on  -rw-r--r--  1 USER1  OMVS      8930 Feb 22 13:04 epscmort.cbl
t IBM-1047  T=on  -rw-r--r--  1 USER1  OMVS    132337 Jan 12 14:06 epscmrd.cbl
t IBM-1047  T=on  -rw-r--r--  1 USER1  OMVS      7919 Feb 22 13:04 epsmlist.cbl
t IBM-1047  T=on  -rw-r--r--  1 USER1  OMVS      5854 Jan 12 14:06 epsmpmt.cbl
t IBM-1047  T=on  -rw-r--r--  1 USER1  OMVS      6882 Jan 12 14:06 epsnbrvl.cbl
```

### Automatic conversion

Although not directly involved in the migration process, the Enhanced ASCII feature introduces an in-flight automatic conversion for files stored in z/OS UNIX. This automatic conversion is mainly controlled by the `_BPXK_AUTOCVT` environment variable or by the `AUTOCVT` parameter defined in a `BPXPRMxx PARM LIB` member. By default, programs on z/OS UNIX are operating in EBCDIC. When the `_BPXK_AUTOCVT` parameter is activated to either `ON` or `ALL`, along with the correct tagging of files, programs executing in z/OS UNIX can transparently and seamlessly work with ASCII files without converting them to EBCDIC.

During the migration process to Git, the `.gitattributes` file is used to describe the conversion format for all the files under the control of Git. To avoid any manual tasks, it is recommended to enable the automatic conversion for any thread working with Git on z/OS and the `.gitattributes` file. This file is discussed in further detail in the section [Defining the code page of files in Git](#).

Other options can interfere in the automatic conversion process. Without being exhaustive, this list provides some parameters which can also influence the global behavior of tools ported to z/OS, including Git for z/OS:

Parameter	Documentation
_TAG_REDIR_IN _TAG_REDIR_OUT _TAG_REDIR_ERR	<a href="https://www-40.ibm.com/servers/resourceink/svc00100.nsf/pages/zosV2R4SA232280/\$file/bpxa500_v2r4.pdf">https://www-40.ibm.com/servers/resourceink/svc00100.nsf/pages/zosV2R4SA232280/\$file/bpxa500_v2r4.pdf</a>
FILETAG	<a href="https://www.ibm.com/docs/en/zos/2.3.0?topic=options-filetag-cc-only">https://www.ibm.com/docs/en/zos/2.3.0?topic=options-filetag-cc-only</a>

These parameters can affect the overall behavior of the z/OS UNIX environment, so they should be configured with caution. Detailing the impacts of these parameters is outside the scope of this document. For more information on how they can affect your configuration, please reference the above documentation. Additionally, the recommended values for these parameters are described in the [DBB configuration documentation page](#).

## Using the DBB Migration Tool

IBM® Dependency Based Build (DBB) provides a migration tool to help facilitate the copying of source artifacts to Git from your z/OS development software configuration management (SCM). This tool can be used to copy the source code contained in your z/OS partitioned data sets (PDSs) to a local Git repository on z/OS UNIX that will later be committed and pushed to a Git server. The DBB Migration Tool has built-in functionality that automates the detection of non-printable and non-roundtripable characters, copies the files to z/OS UNIX, tags them on z/OS UNIX, and generates a `.gitattributes` file. More information on the DBB Migration Tool and its usage in common migration scenarios can be found in [DBB Migration Tool](#).

## Summary

This document highlighted some pitfalls to avoid when migrating source code from z/OS PDSs to Git: correctly determining the original code page used when editing and reading source code members in z/OS is a key activity to ensure a smooth migration of these elements to Git. The second aspect is about managing the specific EBCDIC characters which are not easily converted to their UTF-8 counterparts. For these specific characters, known as non-printable and non-roundtripable characters, a decision must be taken to either refactor the source code to eliminate those characters, or transfer the files as binary. Both options have drawbacks that should be evaluated prior to the final migration to Git, as there is no easy way back.

The DBB Migration Tool shipped with IBM Dependency Based Build helps perform this migration activity by automating the detection of non-printable and non-roundtripable characters, copying the files to z/OS UNIX, tagging them on z/OS UNIX, and generating a `.gitattributes` file. You can learn more about this utility in [DBB Migration Tool](#).

## Resources

This page contains reformatted excerpts from [Managing the code page conversion when migrating z/OS source files to Git](#).

## Methods for migrating data

---

### DBB Migration Tool

#### DBB Migration Tool overview

IBM® Dependency Based Build (DBB) provides a migration tool that facilitates the copying of source code PDS members into a pre-existing local Git repository in z/OS® UNIX System Services (z/OS UNIX), stored on the z/OS File System (zFS).

The commits to the local Git repository can then be pushed to the chosen remote repository in your enterprise Git server.



During the copy process, the DBB Migration Tool will:

- perform the necessary code page conversion from the z/OS code page to the encoding used in Git,
- when applicable, create (or update) the `.gitattributes` file with the correct encoding mappings,
- tag the z/OS UNIX files with the appropriate code page values,
- and (potentially) report on any issues encountered during the migration.

The DBB Migration Tool is bundled as part of the SMP/e installation of the Dependency Based Build toolkit (FMID HBGZ110) and is typically found under the `migration/bin` sub-directory of the DBB installation folder (which by default is `/usr/lpp/IBM/dbb`, unless customized during installation).

The setup, general usage, and various options provided by this tool to assist in the migration can be found on the [IBM Documentation website](#).

The following sections of this page will showcase various migration scenarios using the context that has been described in [Managing code page conversion](#). The intent is not to provide an exhaustive set of scenarios supported by the DBB Migration Tool, but rather to focus on common use cases. The scenarios are:

1. [Migration using the default settings](#)
2. [Migration using the pdsEncoding Mapping Rule](#)
3. [Detection of non-roundtripable characters](#)
4. [Detection of non-printable characters](#)

## Example setup

To illustrate the scenarios, the following sample PDSs were constructed to highlight some specific migration situations that may be encountered and how to mitigate potential issues:

- [MIGRATE.TECHDOC.SOURCE](#)
- [MIGRATE.TECHDOC.COPYBOOK](#)

### ***MIGRATE.TECHDOC.SOURCE***

Content of the `MIGRATE.TECHDOC.SOURCE` dataset:

Member	Description
IBM037	Member that has been created using the code page of IBM-037. Example: IBM-037 Code Page <pre>void main(int argc, char *argv[])</pre> <b>Note:</b> Under the IBM-037 code page, the hexadecimal codes for the <code>[</code> and the <code>]</code> characters are <code>x'BA'</code> and <code>x'BB'</code> , respectively.
IBM1047	Member that has been created using the code page of IBM-1047. Example: IBM-1047 Code Page <pre>void main(int argc, char *argv[])</pre> <b>Note:</b> Under the IBM-1047 code page, the hexadecimal codes for the <code>[</code> and the <code>]</code> characters are <code>x'AD'</code> and <code>x'BD'</code> , respectively.

### ***MIGRATE.TECHDOC.COPYBOOK***

Content of the `MIGRATE.TECHDOC.COPYBOOK` dataset:

Member	Description
NROUND	<p>Member that contains non-roundtripable characters.</p> <p>Example: Non-roundtripable characters</p> <pre> Line with CHAR_NL (0x15)  ¤ Line with CHAR_CR (0x0D)  ¯ Line with CHAR_LF (0x25)  ¸ Line with CHAR_SHIFT_IN (0x0F)  _ Line with CHAR_SHIFT_OUT(0x0E)  _ Line with empty CHAR_SHIFT_OUT(0x0E) and CHAR_SHIFT_IN (0x0F)  __ </pre>
NPRINT	<p>Member that contains non-printable characters.</p> <p>Example: Non-Printable Characters</p> <pre> Line with (0x06)  ? Line with (0x07) Line with (0x1B) </pre>
HEXCODED	<p>Member that contains non-printable and non-roundtripable characters.</p> <p>Example: Hexadecimal Coded Characters</p> <pre> 01  DFHBMSCA. 02 DFHBMPPEM  PICTURE X VALUE IS '_''. 02 DFHBMPNPL  PICTURE X VALUE IS '¸''. 02 DFHBMPFFF  PICTURE X VALUE IS '_''. 02 DFHBMPCCR  PICTURE X VALUE IS '¸''. 02 DFHBMASK  PICTURE X VALUE IS '0'.'. 02 DFHBMUNP  PICTURE X VALUE IS ' ' '. </pre>

Member	Description
HEXVALUE	<p>Member that contains the suggested transformation of the non-printable or non-roundtripable characters contained in the “HEXCoded” member in a more suitable format. Example: Hexadecimal Values</p> <pre> 01  DFHBMSCA. 02  DFHBMPPEM  PICTURE X VALUE IS X'19'. 02  DFHBMPNL  PICTURE X VALUE IS X'15'. 02  DFHBMPFF  PICTURE X VALUE IS X'0C'. 02  DFHBMPCR  PICTURE X VALUE IS X'0D'. 02  DFHBMASK  PICTURE X VALUE IS '0'. 02  DFHBMUNP  PICTURE X VALUE IS ' '. </pre>

## Migration scenarios

### Migration using the default settings

In this scenario, we will be migrating all the source members in the `MIGRATE.TECHDOC.SOURCE` PDS using the default settings into a local z/OS UNIX Git Repository under `/u/user1/Migration`. This is the most simplistic form of invoking the DBB Migration Tool and, in most cases, satisfies most needs.

```

$DBB_HOME/migration/bin/migrate.sh -r /u/user1/Migration -m Mapping
Rule[h1q:MIGRATE.TECHDOC,extension:SRC,toLower:true] SOURCE

Setting dbb.file.tagging = true
Local GIT repository: /u/user1/Migration
Mapping: MappingRule[h1q:MIGRATE.TECHDOC,extension:SRC,toLower:true]
MappingRuleId: com.ibm.dbb.migration.MappingRule
MappingRuleAttrs: [h1q:MIGRATE.TECHDOC,extension:SRC,toLower:true]
Using mapping rule com.ibm.dbb.migration.MappingRule to migrate the data sets
Migrating data set SOURCE
Copying MIGRATE.TECHDOC.SOURCE(IBM037) to /u/user1/Migration/source/ibm037.src using default
encoding
Copying MIGRATE.TECHDOC.SOURCE(IBM1047) to /u/user1/Migration/source/ibm1047.src using default
encoding
** Build finished

```

Note that the DBB Migration Tool is using a default encoding, which is IBM-1047.

An examination of the files on the local z/OS UNIX Git repository will reveal that the files were copied and were tagged with the default code page of IBM-1047.

```

ls -alT /u/user1/Migration/source
total 64
          drwxr-xr-x  2 USER1  OMVS      8192 May  4 12:33 .
          drwxr-xr-x  4 USER1  OMVS      8192 May  4 12:33 ..
t IBM-1047  T=on  -rw-r--r--  1 USER1  OMVS       61 May  4 12:33 ibm037.src
t IBM-1047  T=on  -rw-r--r--  1 USER1  OMVS       61 May  4 12:33 ibm1047.src

```

Additionally, the `.gitattributes` file was created (or updated) with the correct encoding mappings. All source artifacts, except those tagged as binary (to be discussed later), will be stowed in the remote repository using the UTF-8 code page (as defined by the `git-encoding=utf-8` parameter), whereas any artifacts that are copied from the remote repository to z/OS will be translated to the IBM-1047 code

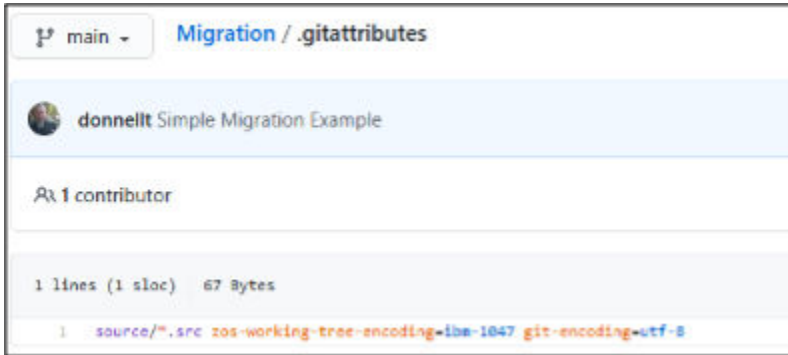
page (as defined by the `zos-working-tree-encoding=ibm-1047` parameter). The documentation on [Defining the code page of files in Git](#) provides more information.

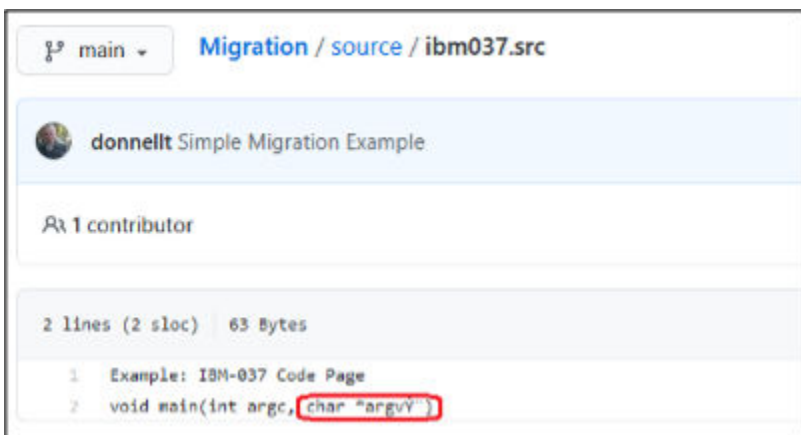
```
cat /u/user1/Migration/.gitattributes
source/*.src zos-working-tree-encoding=ibm-1047 git-encoding=utf-8
```

At this point, Git actions such as `add`, `commit` or `push` can be performed on the migrated source artifacts to introduce them to the remote repository.

```
git add .
git commit -m "Simple Migration Example"
[main 6436b92] Simple Migration Example
3 files changed, 5 insertions(+)
 create mode 100644 .gitattributes
 create mode 100644 source/ibm037.src
 create mode 100644 source/ibm1047.src
git push
Counting objects: 6, done.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 634 bytes | 211.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To github.ibm.com:user1/Migration.git
 3d2962a..6436b92 main -> main
```

Once the Git push command has completed to the remote repository on your Git server, the resulting files should be translated into the correct UTF-8 code page.





However, as indicated in the last picture above, the `ibm037.src` file reveals an encoding issue. This will be discussed in the next scenario.

### ***Migration using the `pdsEncoding` Mapping Rule***

In this scenario, we will be migrating a single source member from the `MIGRATE.TECHDOC.SOURCE` PDS using the `pdsEncoding` keyword in the mapping rule to override the default encoding. Recall that from the previous migration scenario, there was an encoding issue with the final copy of the `ibm037.src` file in the remote repository. This occurred because the z/OS file was written using the IBM-037 code page instead of the default IBM-1047 code page. The problem is not in how the file was encoded, but rather how the Git push action converted the file when sending it to the remote repository.

This can be a common occurrence for source files that pre-date the introduction of z/OS Unix System Services (z/OS UNIX), and where high-level languages, such as C/370, were utilized. As stated previously, determining the original encoding can be a challenge since the code page used to create the file is generally specified in the 3270 Emulator (TN3270) client session set-up. Therefore, an analysis of the z/OS source should be performed to determine the original code page used to create the source. To determine the code page used to create files through ISPF, an alternate option is to ask the developers which code page they are using to edit the files through their 3270 connections.

To correct the encoding issue identified in the previous scenario, we will use the `pdsEncoding` keyword of the mapping rule to override the default IBM-1047 code page with IBM-037 for the offending member.

```
$DBB_HOME/migration/bin/migrate.sh -r /u/user1/Migration -m
MappingRule[h1q:MIGRATE.TECHDOC,extension:SRC,toLower:true,pdsEncoding:IBM-037] "SOURCE(IBM037)"

Setting dbb.file.tagging = true
Local GIT repository: /u/user1/Migration
Mapping: MappingRule[h1q:MIGRATE.TECHDOC,extension:SRC,toLower:true,pdsEncoding:IBM-037]
MappingRuleId: com.ibm.dbb.migration.MappingRule
MappingRuleAttrs: [h1q:MIGRATE.TECHDOC, extension:SRC, toLower:true, pdsEncoding:IBM-037]
Using mapping rule com.ibm.dbb.migration.MappingRule to migrate the data sets
Migrating data set SOURCE(IBM037)
Copying MIGRATE.TECHDOC.SOURCE(IBM037) to /u/user1/Migration/source/ibm037.src using IBM-037
** Build finished
```

Note that the DBB Migration Tool is using the override encoding of IBM-037 for a named member. This override does not necessarily have to be performed on a member-by-member basis, as the DBB Migration Tool supports the ability to override the encoding for an entire PDS being migrated.

An examination of the files on the local z/OS UNIX Git repository will reveal that the file was copied and tagged with the override code page of IBM-037.

```
ls -alT /u/user1/Migration/source
total 64
      drwxr-xr-x  2 USER1  OMVS      8192 May  4 13:10 .
      drwxr-xr-x  4 USER1  OMVS      8192 May  4 13:10 ..
t IBM-037      T=on  -rw-r--r--  1 USER1  OMVS        61 May  4 14:54 ibm037.src
t IBM-1047    T=on  -rw-r--r--  1 USER1  OMVS        61 May  4 13:10 ibm1047.src
```

Additionally, the `.gitattributes` file was updated with the correct encoding mappings:

```
cat /u/user1/Migration/.gitattributes
source/*.src zos-working-tree-encoding=ibm-1047 git-encoding=utf-8
source/*.src zos-working-tree-encoding=IBM-037 git-encoding=utf-8
```

However, in this example you will notice a slight anomaly in that there are two (2) entries for the same sub-folder `source/*.src`. This will cause an encoding conflict during the Git add action. To correct this situation, the `.gitattributes` file must be manually updated to add the file name. Wild cards can be used in the file name should there be more than one member that matches this situation. The order of these entries is important, with the last entry taking precedence. In some cases, additional wild carding may be required to prevent further conflicts.

```
cat /u/user1/Migration/.gitattributes
source/*.src zos-working-tree-encoding=ibm-1047 git-encoding=utf-8
source/ibm037.src zos-working-tree-encoding=IBM-037 git-encoding=utf-8
```

Once the correction has been made to the `.gitattributes` file, the Git commit and push actions can be performed on the updated files to the remote repository:

```
git add .
git commit -m "IBM037 Code Page Fix"
[main 107c86c] IBM037 Code Page Fix
 2 files changed, 2 insertions(+), 1 deletion(-)
git push
Counting objects: 5, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 485 bytes | 485.00 KiB/s, done.
Total 5 (delta 2), reused 0 (delta 0)
To github.ibm.com:user1/Migration.git
 6436b92..107c86c main -> main 3d2962a..6436b92 main -> main
```

Now when examining the offending file in the remote repository, the contents of the file should be translated correctly:



The probability that members of a single PDS were written using a different code page, though possible, is extremely low. However, it is worth pointing out that it could expose an issue in how the DBB Migration Tool generates the `.gitattributes` file.

### ***Detection of non-roundtripable characters***

In this scenario, we will examine how the DBB Migration Tool can assist in the detection of what is known as non-roundtripable characters. The documentation on [Managing non-printable and non-roundtripable characters](#) provides more conceptual background information. To illustrate this, we will be migrating the single source member `MIGRATE.TECHDOC.COPYBOOK(NROUND)`, which contains both types of characters.

During the migration of a PDS member to a z/OS UNIX file, the DBB Migration Tool will scan the content of the file to see if it detects any non-roundtripable characters. These characters are defined in the `migrate.groovy` script and are:

```
@Field def CHAR_NL = 0x15
@Field def CHAR_CR = 0x0D
@Field def CHAR_LF = 0x25
@Field def CHAR_SHIFT_IN = 0x0F
@Field def CHAR_SHIFT_OUT = 0x0E
```

If detected, the DBB Migration Tool will emit a diagnostic message in the console log and will copy the member to z/OS UNIX as binary and therefore no code page conversion will be performed:

```
$DBB_HOME/migration/bin/migrate.sh -r /u/user1/Migration -m
MappingRule[h1q:MIGRATE.TECHDOC,extension:CPY,toLower:true,pdsEncoding:IBM-037]
"COPYBOOK(NROUND)"

Local GIT repository: /u/user1/Migration
Using mapping rule com.ibm.dbb.migration.MappingRule to migrate the data sets
Migrating data set COPYBOOK(NROUND)
[WARNING] Copying MIGRATE.TECHDOC.COPYBOOK(NROUND) to /u/user1/Migration/copybook/nround.cpy
! Possible migration issue:
  Line 2 contains non-roundtripable characters:
    Char 0x15 at column 27
  Line 3 contains non-roundtripable characters:
    Char 0x0D at column 27
  Line 4 contains non-roundtripable characters:
    Char 0x25 at column 27
  Line 7 contains non-roundtripable characters:
    Empty Shift Out and Shift In at column 74

! Copying using BINARY mode
** Build finished
```

Note that the DBB Migration Tool has detected numerous non-roundtripable characters on various lines and has performed the copy as binary.

An examination of the files on the local z/OS UNIX Git Repository will reveal that the file was copied. The file should automatically be tagged as binary by the DBB Migration Tool, but if not, the `chtag -b` command can be used to add the binary tag prior to performing the Git add command.

```
ls -alT /u/user1/Migration/copybook
total 48
          drwxr-xr-x  2 USER1  OMVS      8192 May  8 13:10 .
          drwxr-xr-x  5 USER1  OMVS      8192 May  8 13:12 ..
b binary  T=off -rw-r--r--  1 USER1  OMVS      560 May  8 13:10 nround.cpy
```

Additionally, the `.gitattributes` file was automatically updated by the DBB Migration Tool to indicate that the file is mapped as binary:

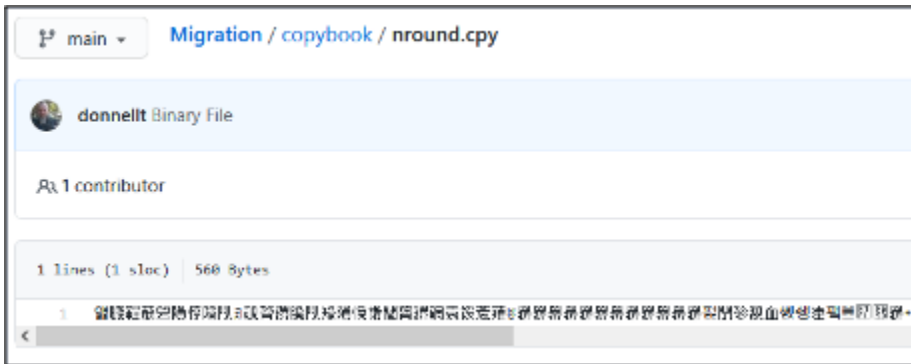
```
cat /u/user1/Migration/.gitattributes
copybook/nround.cpy binary
```

During the Git push to the remote repository, Git will treat this as a binary file and no conversion to UTF-8 will take place. In essence, the resulting file in the remote repository will be the original contents of the PDS member, in EBCDIC.

```
git add .
warning: copybook/nround.cpy added file have been automatically tagged BINARY because they were
untagged yet the .gitattributes file specifies they should be tagged
git commit -m "Binary File"
[main 0213795] Binary File
 2 files changed, 1 insertion(+)
 create mode 100644 .gitattributes
 create mode 100644 copybook/nround.cpy
git push
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (5/5), 598 bytes | 598.00 KiB/s, done.
Total 5 (delta 0), reused 0 (delta 0)
```

```
To github.ibm.com:user1/Migration.git
e901992..0213795 main -> main
```

Once the Git push action has completed to the remote repository, the resulting file will be treated as binary:



This may not be an ideal situation as described in the documentation on [Managing non-printable and non-roundtripable characters](#), and should be corrected/reconciled before continuing with the migration.

### **Detection of non-printable characters**

In this final scenario, we will examine how the DBB Migration Tool can assist in the detection of what is known as non-printable characters. The documentation on [Managing non-printable and non-roundtripable characters](#) provides more conceptual background information. To illustrate this, we will again migrate the single source member `MIGRATE . TECHDOC . COPYBOOK (NPRINT)`, which only contains non-printable characters.

During the migration of a PDS member to a z/OS UNIX file, the DBB Migration Tool will scan the content of the file to see if it detects any non-printable characters. These characters are defined as any hexadecimal values that are an EBCDIC `x'40'` or less, and not one of the five (5) non-roundtripable characters.

The DBB Migration Tool provides three (3) options on how to handle and report on these characters:

1. **Do Not Check:** The non-printable characters are not researched; the member is simply copied as text. Code page conversion will occur.
2. **Info:** The script will emit an Informational diagnostic message in the console log if a non-printable character is detected and the file is copied as text. Code page conversion will occur.
3. **Warning:** The script will emit a Warning diagnostic message in the console log if a non-printable character is detected and the file is copied as binary. No code page conversion will occur, and the offending characters will be treated like non-roundtripable characters.

It should be noted that for the “Do Not Check” and “Info” options, although code page conversion is taking place, this could cause an issue later. The file may not be easily maintained with a non-EBCDIC-aware editor, and you run the risk of having corrupted files.

Controlling what level of checking should be performed during the migration is done via the optional scan level `-np, --non-printable <level>` parameter switch passed to the DBB Migration Tool. If the scan level parameter is not specified, the non-printable characters are not checked (equates to the “Do Not Check” option).

*Scan level set to info*

Example output for migration performed with scan level set to `info`:

```
$DBB_HOME/migration/bin/migrate.sh -r /u/user1/Migration -np info
-m MappingRule[h1q:MIGRATE.TECHDOC,extension:CPY,toLower:true,pdsEncoding:IBM-037]
"COPYBOOK(NPRINT)"

Non-printable scan level is info
Local GIT repository: /u/user1/Migration
Using mapping rule com.ibm.dbb.migration.MappingRule to migrate the data sets
Migrating data set COPYBOOK(NPRINT)
```



```
[INFO] Copying MIGRATE.TECHDOC.COPYBOOK(NPRINT) to /u/user1/Migration/copybook/nprint.cpy using
IBM-037
! Possible migration issue:
  Line 2 contains non-printable characters:
    Char 0x00 at column 19
  Line 3 contains non-printable characters:
    Char 0x06 at column 19
  Line 4 contains non-printable characters:
    Char 0x07 at column 19
  Line 5 contains non-printable characters:
    Char 0x1B at column 19

** Build finished
```

Note that the DBB Migration Tool has detected numerous non-printable characters on various lines and has performed the copy as text and will be tagged on z/OS UNIX using the supplied encoding of IBM-037:

```
ls -aLT /u/user1/Migration/copybook
total 64
          drwxr-xr-x  2 USER1  OMVS      8192 May  6 15:39 .
          drwxr-xr-x  4 USER1  OMVS      8192 May  6 14:55 ..
t IBM-037  T=on  -rw-r--r--  1 USER1  OMVS       114 May  6 15:39 nprint.cpy
```

### Scan level set to warning

Example output for migration performed with scan level set to warning:

```
$DBB_HOME/migration/bin/migrate.sh -r /u/user1/Migration -np warning
-m MappingRule[hfq:MIGRATE.TECHDOC,extension:CPY,toLower:true,pdsEncoding:IBM-037]
"COPYBOOK(NPRINT)"

Non-printable scan level is warning
Local GIT repository: /u/user1/Migration
Using mapping rule com.ibm.dbb.migration.MappingRule to migrate the data sets
Migrating data set COPYBOOK(NPRINT)
[WARNING] Copying MIGRATE.TECHDOC.COPYBOOK(NPRINT) to /u/user1/Migration/copybook/nprint.cpy
! Possible migration issue:
  Line 2 contains non-printable characters:
    Char 0x06 at column 19
  Line 3 contains non-printable characters:
    Char 0x07 at column 19
  Line 4 contains non-printable characters:
    Char 0x1B at column 19

! Copying using BINARY mode
** Build finished
```

Note that the DBB Migration Tool has detected numerous non-printable characters on various lines and has performed the copy as binary. The file may be untagged on z/OS UNIX System Services, and if it is the case, the tagging of the file is still required and should be performed manually:

```
ls -aLT /u/user1/Migration/copybook
total 48
          drwxr-xr-x  2 USER1  OMVS      8192 May  6 17:17 .
          drwxr-xr-x  4 USER1  OMVS      8192 May  6 17:17 ..
- untagged  T=off  -rw-r--r--  1 USER1  OMVS       320 May  6 17:17 nprint.cpy
```

## Recommendations for using the DBB Migration Tool

A strategy must be decided on how to handle both non-printable and non-roundtripable characters found in source members that are to be migrated from z/OS PDSs to Git. The HEXCODED member, though not demonstrated in the above scenarios, is a common occurrence in many older legacy applications. Source code members such as this need to be identified and transformed to that shown in the HEXVALUE member if the goal is to manage the source seamlessly using the modernized tooling provided through Git. For more conceptual background information, please refer to [Managing non-printable and non-roundtripable characters](#).

The DBB Migration Tool provides an option to perform a scan of the z/OS PDSs to assist in the analysis and correction/reconciliation of these situations prior to performing the copy to the local Git repository.

Invoking this scan is done via the optional preview -p, --preview parameter switch and will bypass the copy. The default is “No Preview”. An example of the preview follows:

```
$DBB_HOME/migration/bin/migrate.sh -r /u/user1/Migration -np warning -p -m
MappingRule[hlq:MIGRATE.TECHDOC,extension:CPY,toLower:true,pdsEncoding:IBM-037] COPYBOOK

Non-printable scan level is warning
Preview flag is specified, no members will be copied to HFS
Local GIT repository: /u/user1/Migration
Using mapping rule com.ibm.dbb.migration.MappingRule to migrate the data sets
Migrating data set COPYBOOK
[WARNING] Previewing MIGRATE.TECHDOC.COPYBOOK(HEXCoded)
! Possible migration issue:
  Line 3 contains non-printable characters:
    Char 0x19 at column 37
  Line 4 contains non-roundtripable characters:
    Char 0x15 at column 37
  Line 5 contains non-printable characters:
    Char 0x0C at column 37
  Line 6 contains non-roundtripable characters:
    Char 0x0D at column 37

! Will copy using BINARY mode
Previewing MIGRATE.TECHDOC.COPYBOOK(HEXVALUE). Using IBM-037.
[WARNING] Previewing MIGRATE.TECHDOC.COPYBOOK(NPRINT)
! Possible migration issue:
  Line 2 contains non-printable characters:
    Char 0x06 at column 19
  Line 3 contains non-printable characters:
    Char 0x07 at column 19
  Line 4 contains non-printable characters:
    Char 0x1B at column 19

! Will copy using BINARY mode
[WARNING] Previewing MIGRATE.TECHDOC.COPYBOOK(NROUND)
! Possible migration issue:
  Line 2 contains non-roundtripable characters:
    Char 0x15 at column 27
  Line 3 contains non-roundtripable characters:
    Char 0x0D at column 27
  Line 4 contains non-roundtripable characters:
    Char 0x25 at column 27
  Line 5 contains non-printable characters:
    Char 0x0F at column 33
  Line 7 contains non-roundtripable characters:
    Empty Shift Out and Shift In at column 74

! Will copy using BINARY mode
```

With the -l, --log option, a log file can be created to contain all the messages about the migration process, including the non-printable and non-roundtripable characters encountered during the scan. This log file can be used by the developers to perform the necessary changes in their original source code members prior to the real migration process.

Many other options of the Mapping Rule parameter can be used to control the behavior of the DBB Migration Tool. These options are described on the [IBM Documentation website](#).

## Migrating from IBM Engineering Workflow Management to GIT

For specific information on using the DBB Migration Tool to migrate from IBM Engineering Workflow Management (EWM) to Git, please reference [Migrating from IBM Engineering Workflow Management \(EWM\) to GIT](#).

## Resources

- This page contains reformatted excerpts from [Managing the code page conversion when migrating z/OS source files to Git](#).

# SCLM-to-Git Migration Tool

## Introduction

Software Configuration Library Manager (SCLM) is primarily a library manager and configuration manager. Although it also has some change management and impact analysis capabilities, the functionality is basic by today's standards. To help customers migrate from SCLM to the more modern Git software configuration management solution, IBM® provides the [SCLM-to-Git Migration Tool](#).

The SCLM-to-Git Migration Tool is a set of scripts that extract SCLM data and place it in the target Git repository. The migration is performed within z/OS® UNIX System Services (z/OS UNIX), using Shell scripts that subsequently invoke Groovy scripts. The SCLM-to-Git Migration Tool also creates the `.gitattributes` file for code page conversion, although the tool's generated sample build scripts are not currently compatible with the zAppBuild sample DBB build framework.

Before getting started with the migration, we recommend reviewing your current SCLM setup and removing any unused or redundant definitions to help make the migration process smoother.

## What gets migrated

The SCLM-to-Git Migration Tool migrates more than just source code from SCLM to Git. The following list details what is included in the migration:

- Project definition data
  - The project definitions define the steps a source member needs to go through to be built (for example, precompile, compile, and link edit).
  - This is mainly language definitions, but also includes allocation information of any files used in the language definitions.
- Member-level metadata
  - This includes member-level overrides, which are set by an SCLM construct called an ARCHDEF. This metadata provides the ability to override compile options, change output member names, and change output file destinations.
- Generation of link decks from link-edit control (LEC) ARCHDEFs
  - LEC ARCHDEFs are how SCLM controls the compile and link-edit process. These LEC ARCHDEFs need to be converted to standard link-edit decks for use by DBB.
- Current version of the source
  - The `zimport.sh` script is used to migrate the current production version of the source to Git.
- (Optional) Previous version of the source
  - Previous versions can also be migrated by starting with the oldest version first and working backwards to the current version.

## Migration process

The SCLM-to-Git Migration Tool moves source members from SCLM to Git in three phases. The steps and outputs for each phase are detailed in the tool's [Migration Process](#) documentation on GitHub. You can view a video guide for how to perform each phase in the following list.

The three-phase migration process used by the SCLM-to-Git Migration Tool:

1. Extract the SCLM metadata and source: Watch [Migrating SCLM to Git Part 1](#) to learn more.
2. Migrate source members to a Git repository in z/OS UNIX: Watch [Migrating SCLM to Git Part 2](#) to learn more.
3. Create sample DBB Groovy build scripts: Watch [Migrating SCLM to Git Part 3](#) to learn more.

- **Note:** The sample build scripts created in this step are not currently compatible with the zAppBuild sample DBB build framework. If you would like to use zAppBuild, you can skip this step and simply use the migrated code from the previous step (Step 2) with zAppBuild.

The above video guides are part of the [IBM Dependency Based Build Foundation Course](#). More information on the context, configuration, and steps for the SCLM-to-Git migration tool can be found in Module 5 (Migrating Source Members to Git) of this course.

## Resources

- [SCLM-to-Git migration tool GitHub repository](#)
- [Module 5 - Migrating Source Members to Git, IBM Dependency Based Build Foundation Course](#)
- [Migration made easy - host SCLM to Git on Z \(webinar\)](#)

## Manual migration

Manual migration of source data from z/OS® to Git is generally not recommended, as it tends to be slower, more tedious, and prone to human error. However, it is possible, and can be done several ways, including the following:

- Copy the files to z/OS UNIX System Services (z/OS UNIX) via the Interactive System Productivity Facility (ISPF).
- Copy the files to z/OS UNIX via IBM® Developer for z/OS (IDz).
  - Drag and drop members from IDz's Remote System Explorer (RSE) to a local project.

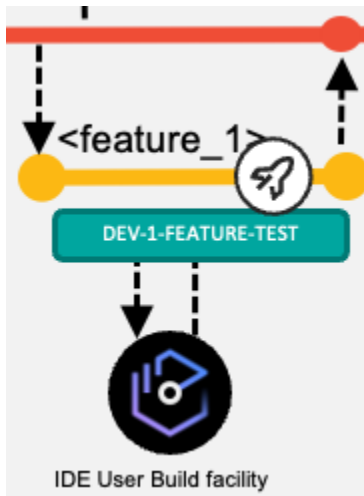
Although manual migration is not recommended, if you do proceed with it, then it is important to remember that you must also manually create the `.gitattributes` file used for code page translation between z/OS and the Git server, and also manually detect and manage code page conversion issues.

# Chapter 7. Implementing CI/CD

## Implementing a pipeline for the branching model

This page details the technical implementation of the different continuous integration/continuous delivery (CI/CD) pipeline types used in the [Git branching model](#) for mainframe development. If a branching model workflow demands a specific configuration, it will be covered within the same section.

### Configurations to support working with feature branches



When developers start working on a new task, they will first create a feature branch. Feature branches are created off the latest code state of the source configuration, whether that is the main branch or an epic or release maintenance branch.

If the feature branch was created on the central Git repository, the developers can use the integrated development environment (IDE), a terminal, or another Git interface on their local workstation to clone or pull the new feature branch from the central Git repository. They then switch to the feature branch to implement their changes.

IDEs supported by IBM® allow developers to perform a Dependency Based Build (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 remote repository, where a pipeline can process changes automatically. Developers regularly commit and push their changes to synchronize with their feature branch in the remote repository.

### User Build setup

User Build is a capability provided by IBM-supported IDEs that uploads the modified source code and its dependencies from the local, checked-out Git working tree on the developer's workstation to a personal directory on IBM z/OS® UNIX System Services, and then invokes the build framework to execute the compile and link-edit steps. This capability is available in the following IDEs:

- IBM Developer for z/OS
- [Microsoft Visual Studio Code \(VS Code\) with the IBM Z® Open Editor extension](#)
- [IBM Wazi for Dev Spaces](#)

The developer configures the User Build process to point to the central build framework implementation, such as [zAppBuild](#), provided by the Mainframe DevOps Team. The build option `--userBuild` is passed to the build framework along with the reference to the file the developer would like to build.

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 (- -h1q) of their personal datasets. It is the developer's responsibility to regularly clean up the mainframe datasets and sandbox directories on z/OS UNIX System Services 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.

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 or as a build output in the artifact repository.

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

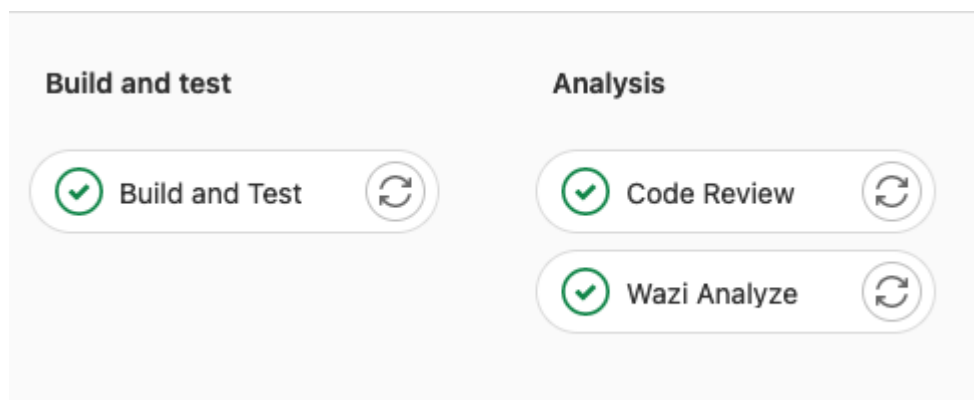
## Pipeline build of feature branches

It is a common practice to use a feature branch pipeline that builds the codebase of a feature branch and runs automated code inspections using the IDz Code Review feature and/or updates the static code analysis repository such as in IBM Wazi Analyze.

This pipeline expands the scope of the build past that of the user build and makes sure all changed and impacted programs are included in the list of artifacts to be produced by leveraging the - -impactBuild option of zAppBuild. The developer must make sure to have pushed the feature branch with all their committed changes from their local clone of the repository to the central Git provider so that those changes are available to the feature branch pipeline process.

The pipeline configuration requires processing logic to compute a dedicated high-level qualifier to guarantee that build datasets are exclusively used for the provided branch. The computed value is passed into the build command via the - -h1q parameter. zAppBuild allocates the datasets automatically.

The following screen capture shows the stages included a sample pipeline build for a feature branch.



The build uses the dependency metadata managed by IBM Dependency Based Build via DBB collections, which are consumed by the build framework, zAppBuild. At the first execution of the build process for 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.

<sup>1</sup> zAppBuild documentation: <https://github.com/IBM/dbb-zappbuild/blob/3.2.0/docs/BUILD.md#perform-impact-build-for-topic-branches>

To be able to clone the collection, zAppBuild needs to understand which collection contains the most accurate information and should be duplicated. As collection names are derived from the name of the branch, it is easy to identify which collection should be cloned. In the zAppBuild configuration, the originating collection reference is defined via the `mainBuildBranch`<sup>1</sup> property.

Depending on the [branching model workflow](#) being used, the `mainBuildBranch` property might need to be overridden for the feature branch pipeline:

- In the [default development process](#) working towards the next planned release based on the head of the main branch, the default configuration for the `mainBuildBranch` is accurate and does not need to be overridden.
- When [implementing a fix in context of a release maintenance branch](#), the `mainBuildBranch` must be set to the name of the release maintenance branch to correctly clone the dependency information.
- When [implementing changes on a feature branch in an epic branch context](#), the `mainBuildBranch` property must be set to the name of the epic branch.

Instead of manipulating the property file that defines the `mainBuildBranch` setting and is part of the repository, the pipeline automation can compute the correct setting and pass the overriding property via the [override command-line option](#) of zAppBuild.

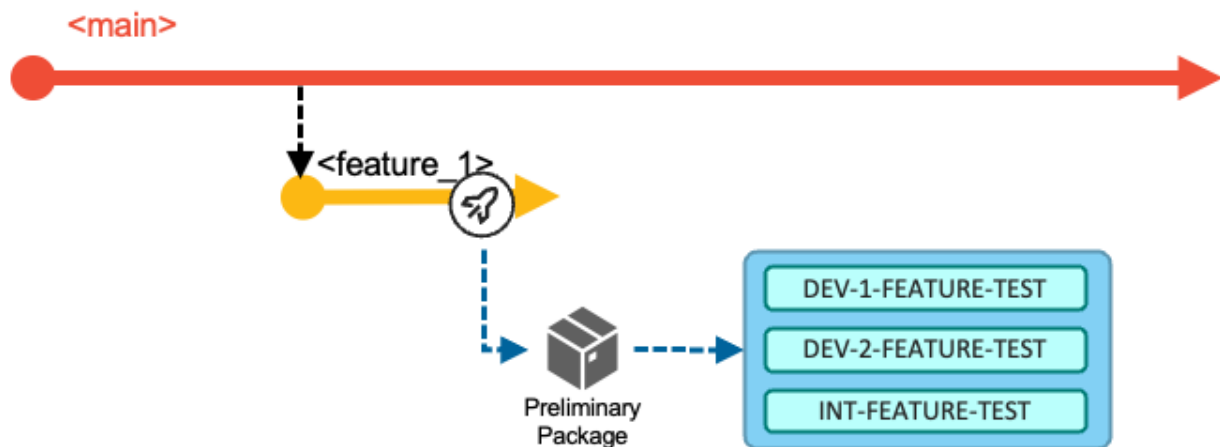
## Package and Deploy a feature for testing in controlled test environments

Today's mainframe development workflows can allow developers to install their changes into controlled test environments before these changes get assigned into a release, for instance when the developer would like to prototype/pilot a new feature. On lower environments, there might be multiple CICS regions that developers can use, which provide a level of isolation from other ongoing development work. The pipeline process can be extended to provide a similar functionality as an optional step for the developer.

This strategy is supported by feature branch packaging and deployment of a preliminary package. It is implemented as a dedicated pipeline that developers request on demand for their feature branch. The pipeline performs the following actions:

1. Build all the changes of the feature branch that were implemented, including their impacts as outlined in [Basic Build Pipeline: Build and Test stage](#), using the commit point at which the feature branch was branched off as the baseline reference for calculating the changes.
2. Package the generated build outputs as outlined in [Release Pipeline: Package stage](#).

The deployment process must ensure that these preliminary packages cannot be deployed into any production environment.



Often, these controlled development test environments are used as shared test environments for multiple application teams. To use the same runtime environment, such as a CICS region, for both prototyping and for testing integrated changes, we recommend separating the preliminary (feature) packages from the planned release packages by separating these types into different libraries. The package for the

prototyping workflow is deployed via its dedicated deployment environment model, illustrated in the above diagram as DEV-1-FEATURE-TEST.

Because preliminary packages are intended to be short-lived and temporary, they can be deployed to a library via the deployment automation process to a pre-concatenated library. Housekeeping strategies must be established to ensure that either automation routines or developers are cleaning up the preliminary packages when the testing is done.

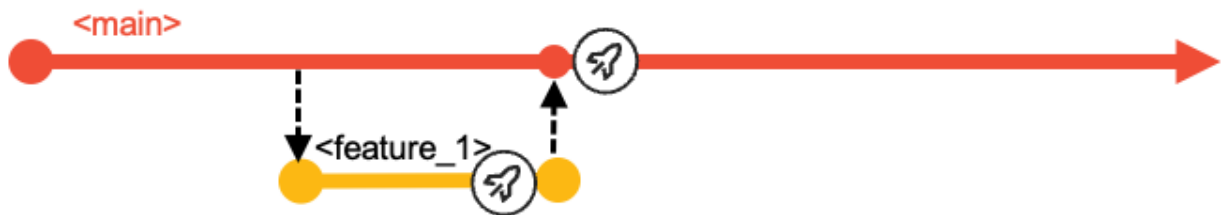
This strategy should be designed with the infrastructure engineering team to prepare the test environments to support this workflow.

## Housekeeping recommendations

A housekeeping strategy should be implemented when the feature branch is no longer needed and therefore removed from the central Git provider. Successful merging adds commits from one branch to the head of another. Once complete, the branch the commits were merged from can be safely deleted. (Keeping old branches can cause confusion and does not contribute to the traceability of the history.) This housekeeping strategy should include the cleanup of the DBB collections, the build workspace on z/OS UNIX System Services, and the build datasets.

Specific scripts can be integrated into the pipeline to delete collections and build groups, or remove unnecessary build datasets. 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 CI/CD Pipelines with Gitlab. Generally, webhooks and other extensions of the pipeline orchestrator can be used to perform these cleanup activities when a branch is deleted.

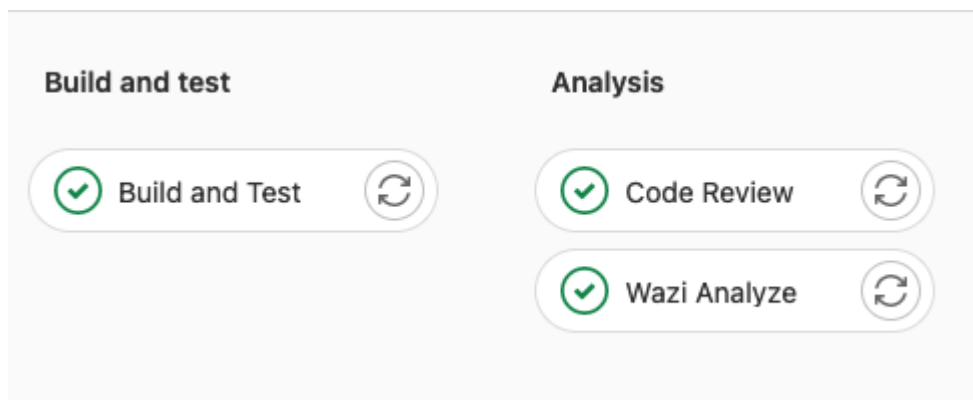
## The Basic Build Pipeline for main, epic, and release branches



It is common practice to build every time the head of the main, epic, or release branch is modified.

When a feature branch is merged into a shared integration branch, a new pipeline is kicked off to build the merged changes in the context of the configuration of the integration branch.

Additional steps such as automated code reviews or updates of application discovery repositories can be included in the pipeline process, as shown in the sample pipeline setup in the following screen capture.





## Basic Build Pipeline: Build and Test stage

The purpose of the Build and Test stage of the pipeline for an integration branch is to ensure that the branch can be built and then tested together. It might happen that some features have indirect dependencies on other features planned for the same deliverable. This early point of integration along with the [impact build capability](#) of the zAppBuild build framework ensures consistency and transparency for the upcoming deliverable.

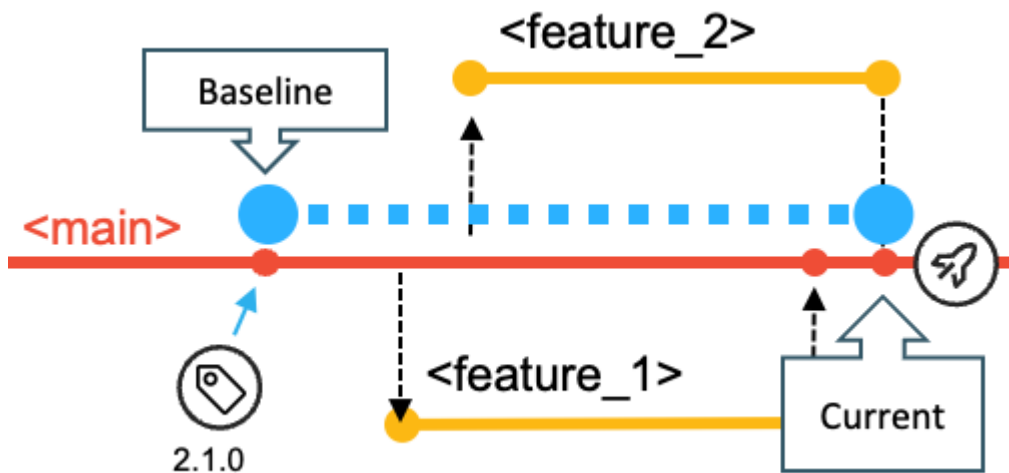
Conceptually speaking, the build step of a CI/CD pipeline decouples building from deploying. This is important to ensure that only outputs from successful builds are installed into the test environment, rather than directing the build framework to update the libraries of the test environment directly.

The Build and Test stage of the pipeline for the integration branch builds all the incorporated changes that have so far been merged for the deliverable. To identify the list of changes contributing to the next planned release, the release maintenance, or the epic, the build step of the pipeline uses the `--baselineRef` option of zAppBuild for incremental builds, which is used to specify a baseline hash or point in the commit history for calculating the list of changes. Using this approach of incremental builds avoids unnecessarily building parts of the application that are unaffected by any of the changes in the commits to the base branch since the last release.

Additionally, the pipeline configuration requires a dedicated high-level qualifier to guarantee that build data sets are exclusively used for the provided branch. The value is passed to the zAppBuild command via the `--hlq` parameter.

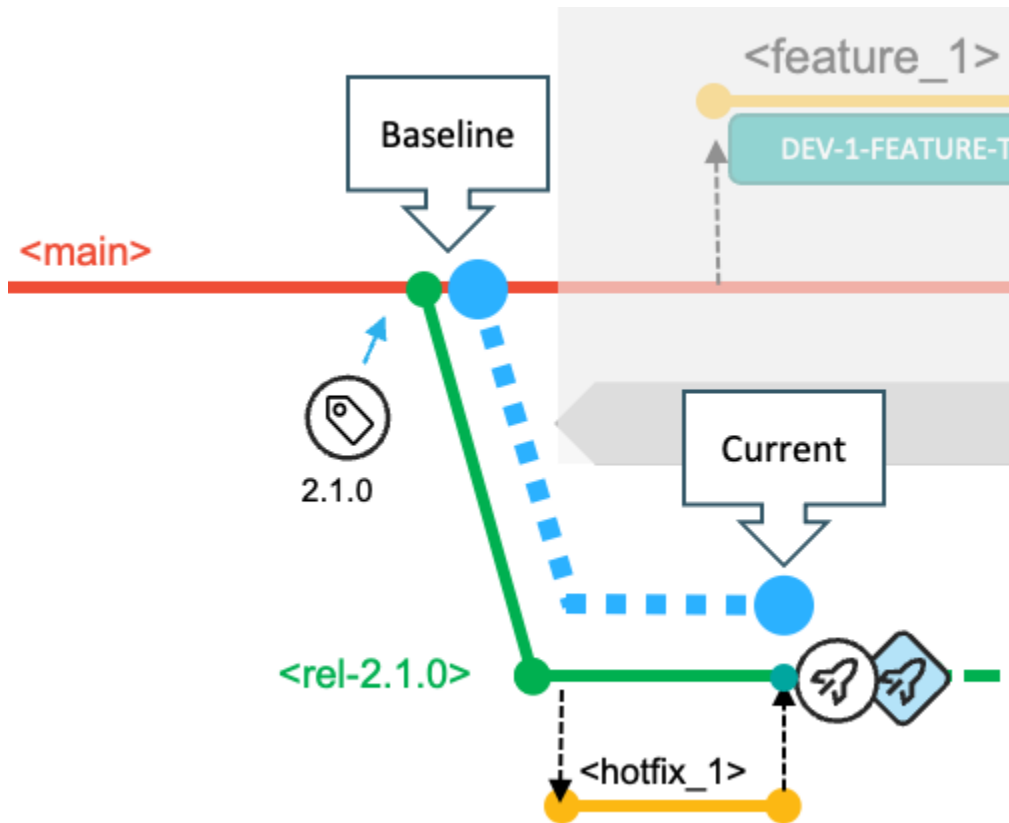
The option `--baselineRef` is a sub-parameter of the `--impactBuild` option in zAppBuild, and sets the base Git hash upon which the `git diff` command calculates changes for the repository.

In the [default workflow](#) with `main` as the base branch, the baseline reference is defined by the commit hash (or the Git tag) of the previous release (that is, the release currently in production). In the following diagram, the blue dotted line shows the changes calculated from the baseline to the point at which the `feature_2` branch is merged in.



For the [hotfix workflow](#), the hotfixes are planned to be implemented from a release maintenance branch whose baseline reference is the commit (or Git tag) that represents the state of the repository for the release. This is also the commit from which the respective release maintenance branch was created, as depicted in the below diagram.

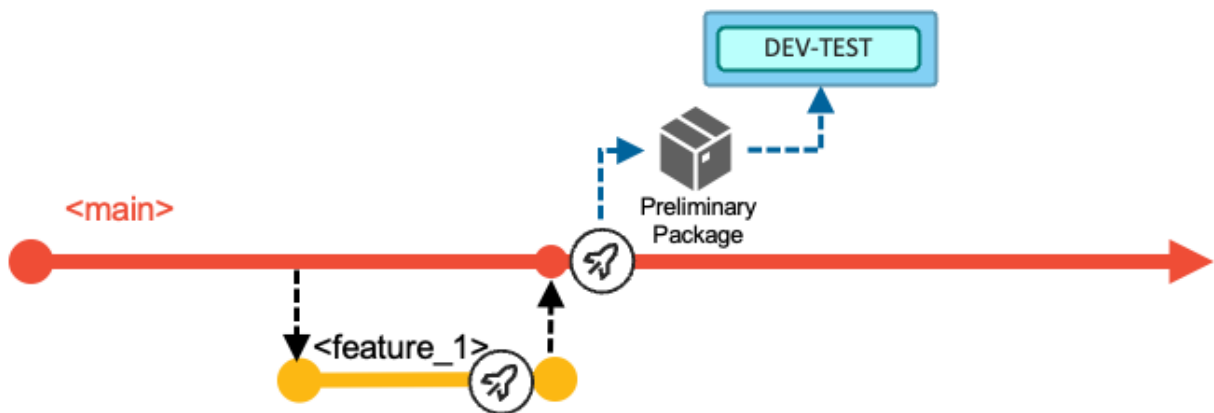
<sup>2</sup> zAppBuild implementation to set `baselineRef`: <https://github.com/IBM/dbb-zappbuild/blob/3.2.0/docs/BUILD.md#perform-impact-build-by-providing-baseline-reference-for-the-analysis-of-changed-files>



For the epic branch workflow, the baseline reference for the build pipeline is the commit (or Release tag) from which the epic branch was created, also referred to as the fork point.

### Basic Build Pipeline: Install outputs to a development and test environment

In this phase of the development lifecycle for the default workflow implementing and delivering changes for the next planned release, the build typically operates with the compile options to enable testing and debugging of 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.



There are two options to deploy the generated artifacts to the shared development test system - represented by the blue DEV-TEST shape in the above figure.

(Recommended) Option A: Extend the pipeline with a packaging stage and a deployment stage to create a preliminary package similar to Release Pipeline: Package stage. It is traditionally the responsibility of the deployment solution to install the preliminary package into different environments. Doing so in this

phase of the workflow will give the necessary traceability to understand which versions are installed in the development and test environment.

Option B: Use a post-build script to copy the output artifacts from the build libraries to the associated target runtime libraries and manually run the necessary activation steps such as a Db2® bind process or an online refresh. However, even given the temporary nature of the outputs created by this build, this circumvents the formal packaging and deployment process. The major drawback of this approach is a lack of traceability and understanding of what runs on the development and test environment.

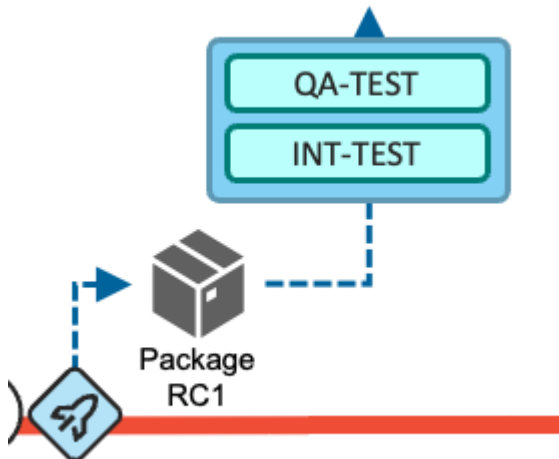
This step of the build pipeline is also applicable for pipelines for the epic or the release maintenance branches.

### Basic Build Pipeline: Analyze stage

An optional Analyze stage after building the most current state of the main branch can include steps to perform automated code inspections using the [IDz Code Review feature](#) and/or to update the static code analysis repository such as in [IBM Wazi Analyze](#).

Submitting a Sonarqube scan at this point of the workflow can also help the development team to keep an eye on the maintainability and serviceability of the application.

## The Release Pipeline with Build, Package, and Deploy stages



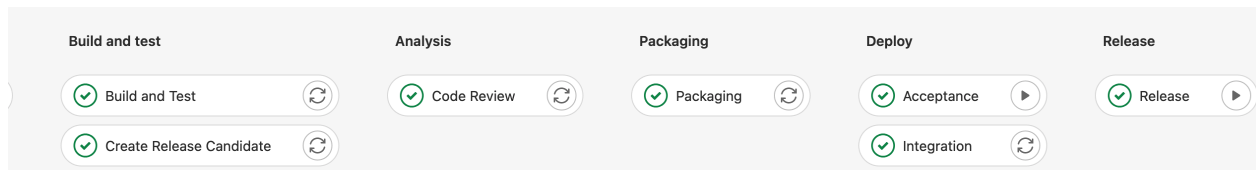
The Release Pipeline is used by the development team when they want to create a release candidate package that can be deployed to controlled test environments. The development team manually requests the pipeline to run. The pipeline is not expected to be used for every merge into the main branch.

The Release Pipeline differs from the previously-discussed pipelines and includes additional steps: after the stages of building and code scans have successfully completed, the pipeline packages all the incorporated changes of all merged features for this deliverable to create a package.

The package can be an intermediate release candidate version that can already be tested in the managed test environments, as outlined in the [high-level workflows](#). When the development team has implemented all the tasks planned for the iteration, this same pipeline is used to produce the package that will be deployed to production.

The following diagram outlines the steps of a GitLab pipeline for the Build, Package, and Deploy stages.

The Deploy stage can only be present in the pipeline for the default workflow (with main) when delivering changes with the next planned release, because the pipeline is unaware of the assigned environments for the epic and release maintenance workflows.



## Release Pipeline: Build stage

Similar to the build pipeline outlined in [Basic Build Pipeline: Build and Test stage](#), the Build stage of the Release Pipeline builds all the incorporated changes of all merged features. To identify the list of changes contributing to the deliverable, the Build stage of the pipeline uses the `--baselineRef` option of `zAppBuild` to detect all contributed changes based on rules outlined in [Basic Build Pipeline: Build and Test stage](#).

For the main and the release maintenance workflows, this build performs the compilation with the compiler options to produce executables optimized for performance. The pipeline must use its dedicated set of PDS libraries to not interfere with the [Basic Build Pipeline](#).

For the epic branch workflow, the build can occur with test options, as the package is only deployed into the initiative's test environment and will be rebuilt when the epic branch is integrated into the main branch.

## Release Pipeline: Package stage

The Package stage runs after the Build and Test stage, and creates a package of the generated build outputs (for example, load modules, DBRMs, and JCLs). This package includes the build outputs of all the contributed changes (including the files impacted by the changes) for the deliverable. It represents a release candidate that can be deployed into the various test environments along the existing staging hierarchy. As outlined in the high-level workflows, this can even happen when only a subset of the features for the deliverable is implemented.

The name or associated metadata of the package allows the development team to relate the package to the development workflow. Based on naming conventions of the package, different rules need to apply to its lifecycle. Some examples of this are shown below:

- Use `rel-2.1.0_RC01` for a release candidate package for the next planned release.

The package name represents the name of the next planned release. This package can make it to production, as it contains build outputs produced with options for optimization.

- Use `rel-2.0.1-patch_RC01` for an urgent fix package of the current production version.

This package is allowed to bypass any concurrent development activities and can take shortcuts in the route to production. For example, if it can only be tested on the QA-TEST environment, the developers can bypass lower test environments based on an "emergency" flag of the deployment package.

- Use `epic1-prelim_pkg01` for a preliminary package of a long-running epic branch workflow.

This package can only be deployed to the assigned test environments available to the initiative, and cannot be deployed to production.

The Package stage 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 pipeline job), which are helpful for understanding the context of the creation of the package.

The [DBB community repository](#) contains two sample scripts that implement the Package stage. If IBM UrbanCode® Deploy (UCD) is used as the deployment solution, the [CreateUCDComponentVersion](#) script can be used to create an IBM UrbanCode Deploy component version. Alternatively, if a scripted deployment is being set up, the [PackageBuildOutputs](#) script can be used instead to store artifacts in an enterprise binary artifact repository used by [IBM Wazi Deploy](#).

Both sample scripts use data from the DBB build report to extract and retain the metadata, allowing traceability between the build and deployment activities as outlined above.

## Release Pipeline: Deploy stage

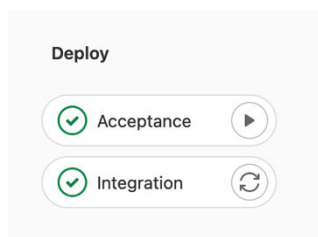
The deployment process of a release package for the [default workflow with main](#) can either be triggered from the CI/CD pipeline or driven through the user interface of the deployment manager. The implementation can vary based on the capabilities offered by the CI/CD orchestrator and the deployment solution. Deployment manager options for z/OS include IBM UrbanCode Deploy (UCD) and IBM Wazi Deploy.

IBM UrbanCode Deploy provides a rich web-based interface, powerful REST APIs, and a command-line interface. Typically, the pipeline execution requests the deployment of the application package into the defined test environments automatically, after successful completion of the preceding Build and Package stages. 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 UCD's web interface. In that scenario, the pipeline is primarily used for continuous integration and packaging.

The [DBB community repository](#) provides a sample [DeployUCDComponentVersion](#) script that can be included in a pipeline process to request a UCD application deployment leveraging UCD's REST APIs.

IBM Wazi Deploy is a deployment manager for z/OS artifacts and comes with a command-line interface that can be easily invoked from a pipeline orchestrator and does not require a wrapper script. After retrieving the package to deploy from the artifact repository, the `wazideploy-generate` step generates the deployment instructions (also known as the deployment plan) for the artifacts of the package. This plan is then passed to the `wazideploy-deploy` step, which installs the contents of the package into the specified runtime environment.

The following screen capture shows the Deploy stage of a sample Release Pipeline.



Implementation details of the Deploy stage can vary based on the pipeline orchestrator being used. 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).

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 manager, or alternatively, they can design and set up a deployment pipeline in Jenkins that can combine the deployment with any automated tests or other automation tasks.

## Deployment to production

When the release candidate package has passed all quality gates and received all the necessary approvals, it is ready to be deployed to the production environment.

The release manager takes care of this step of the lifecycle and will use the user interface of the deployment manager, such as UCD's browser-based interface. In the case of a deployment manager solution with a command-line interface such as Wazi Deploy, the user interface of the pipeline orchestrator is used by the release manager to drive the deployment to production. A deployment pipeline definition needs to be configured to roll out the package.

---

<sup>3</sup> IBM Wazi Deploy static deployment workflow: <https://www.ibm.com/docs/en/developer-for-zos/16.0?topic=deploy-getting-started-static-deployment>

Deploying to production consists of two tasks:

1. Invoke the deployment to the production runtime environment through either the deployment manager interface or a deployment pipeline definition.
2. Tag the commit in the remote repository by assigning a Git tag to the commit that was used to build the release package.

Most Git providers allow for the creation of a release to provide a summary of the changes, as well as additional documentation. [GitLab](#) and [GitHub](#) offer REST APIs to create a new tag/release. This action should be automated as part of the deployment to production.

As an example of using Git tags, [zAppBuild](#) also declares releases to identify stable versions.

## Conclusion

This page provides guidance for implementing a [Git branching model for mainframe development](#) with IBM Dependency Based Build and zAppBuild.

The CI/CD pipeline configurations that were outlined at various stages 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 designing 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 outlined branching strategy can also evolve to maximize the benefits from these advances.

## Implementing pipeline actions

---

### Common Back-end Scripts

Popular choices for continuous integration/continuous delivery (CI/CD) pipeline orchestration include Azure Pipelines, Github Actions, Gitlab CI, and Jenkins. Each orchestrator has its specific way of defining the configurations to coordinate pipeline actions. For instance, Jenkins has its Jenkinsfile, and GitLab CI has its `.gitlab-ci.yml` file.

To help provide a consistent CI/CD pipeline experience regardless of your selected pipeline orchestrator, the Dependency Based Build (DBB) community repository provides a set of [Common Back-end Scripts for pipeline implementations](#). These scripts cover orchestration tasks throughout the CI/CD pipeline, and can be run from any pipeline orchestrator by including them in the orchestrator's configuration as shell script invocations. You can learn more about how to use these Common Back-end Scripts to simplify your pipeline configuration and setup in the [Common Back-end Scripts README](#).

To see a sample implementation of the Common back-end scripts, watch the video [Using the Common Backend Scripts with a Microsoft Azure DevOps \(ADO\) Pipeline](#).

### Specific guides for each orchestrator

The following guides for the common pipeline orchestrators may help you get started with implementing your CI/CD pipeline. Although written before the creation of the Common Back-end Scripts, these guides can be used to integrate the Common Back-end Scripts into the specific orchestrators.

**Note:** These guides cover commonly-selected combinations of tools for the different CI/CD pipeline components, but they are intended as examples and are therefore not all-inclusive. If you do not see the specific technology or combination of tools your enterprise has selected, that does not necessarily mean it cannot align with our guidance. For most tools being used elsewhere, you can integrate them into the pipeline as long as they are compatible with z/OS® application development and bring value to your enterprise.

## **Azure Pipelines**

- [Azure DevOps and IBM® Dependency Based Build Integration](#)

## **GitHub Actions**

- [Using IBM Dependency Based Build \(DBB\) with Github Actions](#)

## **GitLab CI**

- [Build a pipeline with GitLab CI, IBM Dependency Based Build, and IBM UrbanCode® Deploy](#)
- [Integrating IBM z/OS platform in CI/CD pipelines with GitLab](#)

## **Jenkins**

- [Build a Pipeline with Jenkins, DBB and UCD](#)
- [Managing git credentials in Jenkins to access the central git provider](#)
- [POC Cookbook – Building a modern pipeline in Mainframe](#)
- [Setting up the CI Pipeline Agent on IBM Z as a Started Task](#)





---

# Chapter 8. Resources

## Additional resources

---

Follow the following links for additional information:

1. [DBB Documentation](#)
2. [DBB community samples repository](#)
3. [zAppBuild repository](#)
4. [Discover and plan for z/OS hybrid applications](#)
5. [CI for the z/OS DevOps experience](#)



# Legal information

---

## Trademarks

---

IBM®, the IBM logo, and [ibm.com](http://ibm.com)® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" [here](#).

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux® is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Java™ and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Red Hat®, JBoss®, OpenShift®, Fedora®, Hibernate®, Ansible®, CloudForms®, RHCA®, RHCE®, RHCSA®, Ceph®, and Gluster® are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.

## Privacy policy considerations

---

IBM® Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at <http://www.ibm.com/privacy> and IBM's Online Privacy Statement at <http://www.ibm.com/privacy/details> the section entitled "Cookies, Web Beacons and Other Technologies" and the "IBM Software Products and Software-as-a-Service Privacy Statement" at <http://www.ibm.com/software/info/product-privacy>.

This doc site provides content search capability by using [DocSearch by Algolia](#). For data that DocSearch collects, check [DocSearch FAQ](#). This doc site is a subscriber of Algolia's service, and you can refer to the applicable section of [Privacy policy of Algolia](#).





