PACKAGING AND DEPLOYMENT STRATEGIES IN AN OPEN AND MODERN CI/CD PIPELINE FOCUSING ON MAINFRAME SOFTWARE DEVELOPMENT

Dennis Behm
Nicolas Dangeville

Abstract
Learn about the different concepts and strategies of application packaging and deployments in traditional mainframe software delivery lifecycles

Release 1.0
Table of Contents

1. **Introduction and Motivation** .......................................................... 3
   - The big picture .................................................................................. 3
   - What does it mean to work with a CI/CD pipeline? ......................... 6
   - Parallel development within an application team .......................... 7
   - Categorization of elements and the Scope your Git repository ....... 8

2. **Packaging and Deployment characteristics of the pipeline** .......... 10
   - Introduction of Artifact Repositories ............................................. 10
   - Building, Packaging and Deploying in a Pipeline ......................... 11
   - What’s new with transferring objects outside of datasets on IBM Z .. 11
   - Package Content and Layout ....................................................... 12
     - Package Strategy and Scope ..................................................... 13
     - Specialty of Mainframe Packages .......................................... 14
   - Introduction of Deployment Managers ......................................... 15
   - Relationship between Branches, Build, Package and Execution Environments ...... 16

3. **Evaluating different packaging and deployment strategies** ............. 18
   - Reference Application Scenario .................................................. 19
     - Strategy: Partial Packages ....................................................... 19
       - Packaging Phase ..................................................................... 19
       - Deployment phase ............................................................... 20
       - Implications of partial packages ........................................... 21
       - Embedding the package process into your Git workflow .......... 23
       - Options to create cumulative packages ................................ 24
     - Strategy: Full Packages .......................................................... 26
       - Packaging phase ..................................................................... 26
       - Deployment phase ............................................................... 27
       - Implications of full packages ................................................. 28
       - Additional requirements on the package ................................. 29
     - Advanced Deployment Strategy: Incremental Deployment with full packages .... 30
       - Deployment phase ............................................................... 31
       - Implications of advanced deployment strategy ...................... 33
   - Rollback ......................................................................................... 34
About the authors:
Dennis Behm, dennis.behm@de.ibm.com, IBM DevOps Acceleration Team, IBM Germany
Nicolas Dangeville, dangeville.n@fr.ibm.com, IBM DevOps Acceleration Team, IBM France

Thank you to the following reviewers:
Rosalind Radcliffe, IBM U.S.
Mark Boyd, IBM U.S.
Bruce Green, IBM U.S.
Sherri Hanna, IBM U.S.
Chris Hoina, IBM U.S.
Flemming Petersen, IBM Denmark
François Dumont, IBM France
Gatien Dupré, IBM France
Senthil Nathan, HCL
1. Introduction and Motivation

This is the second whitepaper in a series of publications focusing on applying modern and/or open source technologies in traditional mainframe software delivery lifecycles. These papers focus on the tools and how they are used for mainframe applications. It is important to recognize a transformation to modern tools alone will not be a complete solution and may in fact make things harder. It is important to transform to modern ways of working and bring the modern tools in support of the process changes.

Our first publication discussed utilizing Git as a version control manager for mainframe software development, the various adoption processes of common interfaces, the management of copybooks and includes shared between application systems, and the concepts of building applications. The focus of this document will be on the packaging and deployment strategies of the CI/CD pipeline.

Although we will introduce you to different concepts and design patterns, assisting you in your decision-making process, we will limit detail on actual implementation. We will reintroduce some topics addressed in our first publication, but we would encourage you to take a look at our first whitepaper in this series if you have not yet already.

Our first whitepaper can be found here: Develop Mainframe Software with Opensource Source Code Managers and IBM Dependency Based Build

The big picture

A generic CI/CD pipeline setup includes several building blocks, each serving a dedicated purpose.

A key goal of applying modern technologies in the mainframe development environment is the standardization of delivery processes and practices across platforms and leverage synergistic effects across all development teams. On one hand the teams will use an almost identical toolset, and on the other they will also streamline the development/delivery practices in terms of automation, quality and productivity.

In general, the delivery process is not performed in sequence: we see iterations as there are loops and interactions (sign-off, reviews, approvals) that are involved in the process. The pipeline provides an opportunity to introduce new practices alongside well-known existing ones. These new opportunities can take many forms: the provisioning of an execution environment with the push of a button, the triggering of a pipeline during key development events, or the leveraging of artifact repositories to fully decouple continuous integration components from delivery.

The following image (Figure 1) represents the different capabilities involved in an end to end pipeline, and highlights the interactions and loops in the process:

---

1 http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102772
Let’s first focus on the continuous integration phase of the pipeline. As is custom, an IDE allows developers to make code changes. The files are managed by a source configuration manager (SCM), which provides configuration and version control management. These source files are built, and the binaries and outputs are created. These outputs can be managed, as part of an archive that we will call a package, in an artifact repository. The artifact repository makes outputs available to the deployment manager. Deployment to various environments can begin.

The automation and its rules during all these steps are provided by an orchestrator.
Let’s review the building blocks (**Figure 2**) and map them to the practices managed in a pipeline. Additional blocks can contribute to the pipeline, but we will limit our focus to the key ones.

**IDE**

An IDE (Integrated Development Environment) provides check-out and check-in capabilities to the version control 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.

**Pipeline Orchestration**

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.

**Version Control (Source Control Management)**

The SCM is used to manage and store different versions of your application configuration like source code files, application specific configuration data, test cases etc. It provides capabilities to isolate different development activities and enables parallel development.

**Build**

The build component 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.

**Artifact Repository**

Once building occurs, the pipeline then publishes and stores the build outputs in the form of a package. This package contains any artifact that will need to be deployed, possible examples are load modules, DBRMs, DDL, and the configuration files for the subsystems. 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. Please refer to our first whitepaper for more details.

The key mission and benefit of an artifact repository is to decouple SCM configurations from runtime environments.

**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; when DB2 is involved, it’s a bind against the database of the environment.
This pipeline is notable because each of the building blocks share a loose coupling with one another. Scalable and modular by design, oftentimes several products and technologies can be employed for each block in the pipeline. Regardless, whatever technology is chosen, it is important to preserve traceability and audit across the entire pipeline.

In the image below, we describe a possible combination of products that can be used in a pipeline that supports the mainframe.

**Figure 3 Sample Pipeline Overview**

IBM Developer for z/OS and IBM Z Open Development provide the editors and tools a mainframe developer needs in order to write their code. This image (Figure 3) depicts steps with two different colors, yellow and green. Yellow highlights the steps performed by the developer while green is 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. All that is left for the user to do is verify the results of the pipeline job and review the different dashboards provided by the automated processes.

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

The pipeline building blocks 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

When it comes to isolation techniques, we refer to the branching strategies for the version control system. When using Git - a popular workflow called ‘git flow’ assumes that each developer creates a topic branch. Another option, the ‘feature branch’ workflows, isolates the work of a sub team in its own specific branch.

Compared to a library manager, which relies on concatenation for compilation and linking, a SCM 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 the 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.

Another feature of decoupling is that it enables developers to adopt previously impossible provisioning practices such as spinning up an isolated test execution environment with the push of a button.

Developers can now integrate and share changes with the team through a previously agreed upon central Git provider. Accordingly, additional workflows like merge or pull requests for code review and approvals are expected to be used.

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. 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.

But before we dive into packaging and deployment, we must first quickly revisit the key topics (discussed in detail in the first whitepaper) of repository layouts.

Parallel development within an application team

The following image (Figure 4) describes a simple set of branches that can be found in a Git repository. Master represents the code running in production.

Next release is the main branch that integrates the various features, each developed in isolation in their short-lived branches.

Hotfix branches, derived from master, are created to resolve production defects. Once validated the hotfix branch is merged in master and next release, then can be removed.

The yellow branches, seen as features, display the short-lived branches which are created for a specific task and are deleted once the work is completed.
If needed, two release branches can be created. For example, once a release has been declared: the development of the next one can proceed in parallel.

**Categorization of elements and the Scope your Git repository**

To provide greater isolation for each team, the source code of the entire IT landscape should be stored in several Git repositories, rather than just one.

The scope of the Git repository will vary, but let’s assume that each functional area has its own repository. We’ll loosely refer to these areas as components.

While identifying their scope might not be obvious, you will encounter components on the mainframe as you would elsewhere. As an example, you can identify the elements which are used solely within a component. Very often the component provides some well-defined interfaces (in COBOL, they are expressed by copybooks). The component itself, can again consume copybooks (aka interfaces) of other components.

Translating this to the architecture of COBOL applications, some of these copybooks represent the interface description of how a program should be invoked using the linkage section; while others are purely internal to the component. When the various application systems interact with each other, the copybooks acting as interfaces of these application systems require special attention. A similar concept applies to PL/I applications and their *include files*. Details on the different adoption workflows, and how to handle these public interfaces can be found in our Part One whitepaper.
With the introduction and Part One recap complete, let’s discuss the Continuous Deployment aspect of the pipeline.
2. Packaging and Deployment characteristics of the pipeline

First, we’ll introduce artifact repositories, their purpose, and their contents. Then we’ll discuss the need for a package descriptor; a component that contains metadata about the artifacts of a package.

**Introduction of Artifact Repositories**

Artifact repositories focus on storing and versioning the build artifacts (executable, deployable artifacts and additional files) along with their metadata and other necessary pieces of information which allow tracing back of changes to the version control system.

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.

The artifact repository acts like a proxy for the deployment manager. The deployment manager is responsible for deploying the artifacts produced by the build system to one or more runtime environments.

The artifact repository is a key element when implementing a CI/CD pipeline. It decouples version control systems from the actual execution environments. And it also enables deployment of application code into newly provisioned test environments.

Today, popular artifact repository (Figure 6) implementations include products such as JFrog Artifactory, Sonartype Nexus or IBM UrbanCode Deploy.

*The artifact repository consolidates application components*

Let’s assume a typical application (Figure 7) consists of several application components such as: a front-end, middle-tier and back-end. We would expect to see one entry in the artifact repository for the entire application, with subfolders for each application component.

Note that mainframe executables, i.e. program objects, DBRM, etc. are being transferred to the central artifact repository manager along with all the other platform artifacts.
Building, Packaging and Deploying in a Pipeline

A pipeline removes the manual tasks of defining a package. This task is now fully automated and occurs once the application is built.

You should expect to have multiple packages. And each package is expected to have passed automated quality gate testing. Not all packages will make it to production due to discovered defects.

The creation of a package most likely occurs at the application component level. Therefore, the SCM layout also has an impact on the packaging.

Technically, a package is first composed of a list of binaries or files. But it also includes metadata for the deployment manager. Using the metadata and a 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 that of 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.

What’s new with transferring objects outside of datasets on IBM Z

Historically load modules and program objects could not be moved off IBM Z datasets without the loss of file attributes. But in mid-2018 this changed. At that time the z/OS operating system was enhanced to support the moving of load modules and program objects from z/OS datasets to a Unix System Services file system.

You might be asking: why is this important information? Well, consolidation and standardization of your shop’s artifact repository could lead you to host the package repository on a distributed platform, so you would move load modules to the artifact repository off IBM Z.

During this roundtrip, moving a binary from the mainframe into an artifact repository and back to the desired execution environment, a copy between the PDSE and USS is performed. You might see that the binary content of a program object can change. The copy between USS and PDSE involves the Z/OS binder program behind the scenes. The binder internal information area in the program object may change, but not the executable pieces. The binder stamp stays the same, therefore the executable program’s integrity is preserved, and compliance is insured.

Please also see Z/OS PTF OA55299: New Function - Enhance CP to support Copy files together with aliases.

https://www-01.ibm.com/support/docview.wss?crawler=1&uid=isg1OA55299
Package Content and Layout

Of course, it is imperative that you consider **build outputs** (items highlighted in yellow) as part of your package. But it’s equally important that we 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.

In the following image (**Figure 8**), you see lists of parts, which 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 distributed 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 its contents (which is metadata)

For the mainframe, a package will contain executables required to run your application like program objects, DBRM, JCL, control cards – as well as a manifest file.

**Hint:** 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 (i.e. programs), while the other repositories provide additional files (e.g. 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 – all executables and configuration files belonging to an application component – and partial packages – containing just the updated executables and configurations. You may 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 - which are produced immediately after each build and cumulative packages – those that include outputs produced by several builds.

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

As you’ve seen already, there are many factors you’ll need to consider when selecting your strategy. As a practical exercise, let’s walk through the following mainframe scenario using delta packages. In this scenario (Figure 10) 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.

![Figure 10 Execution environments are not updated at the same pace](image-url)
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.

But 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’ll encounter long running, existing ones - 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 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 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, the retrieval of-, and the deployment of all increments till the desired state is achieved (in our example, package 1, 2, and 3). In some cases, packages will overlap. Although, the deployment manager may 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 (e.g. 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.

Let’s examine the two options that we have identified and relate them to the different IBM Dependency Based Build types:

- **Strategy: Partial packages**
  Contains the outputs of an 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, we 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, e.g. 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, etc. Considering the direction of containers with Kubernetes using helm charts and OpenShift templates using yaml, using yaml for the meta data will make it more consistent with other industry work and make it clearly understandable by z/OS and non z/OS developers.

![Figure 11 Schema of an Application Package Manifest](image)

**Introduction of Deployment Managers**

Deployment managers support the automated application rollout of the application binaries and of the application configurations across platforms, that is managing distributed and mainframe 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 of a given environment.

The deployment process can be implemented by scripts or with some 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.

**Relationship between Branches, Build, Package and Execution Environments**

Due to the nature of Git, we expect to see branches appearing and disappearing. We mentioned short-lived topic branches earlier. Pipeline orchestrators can dynamically discover new branches in the central Git repository and create new pipeline jobs.

Each short-lived branch can potentially produce its own set of packages, that should to be tested in its own isolated execution environment.

You can see, that depending on the branch involved and the pipeline job, a package might not be allowed to be deployed to a QA or production environment. For example, a development package may contain artifacts built with debug options, which are not allowed in production. It is also worth recognizing that integration with other application systems generally has not happened in a topic branch. But of course, these programs need to be tested, even if they will be rebuilt when merged in a release branch.

At this point the pipeline could certainly leverage runtime provisioning technology to allocate an isolated execution environment. For those on IBM Z, please have a look at the capabilities of z/OSMF Cloud Provisioning, z/OS Provisioning Toolkit or IBM Z Development & Test environment.

![Diagram](image)

*Figure 12 The pipeline orchestrator can automatically identify new configurations*

To summarize, there should be a single pipeline (Figure 12), along with a dedicated pipeline job for each application configuration. Running the job will produce deployable packages, consistent
with the scope of the application component. By convention, the published package in the artifact repository manager tracks information about where it originated from.

This branching strategy leads to merging the topic branches into an integration or release branch. The packages produced by the build of these branches have more of an official status than the ones produced from feature branches. Only packages from the release branch, are the ones that will be deployed in multiple environments (QA, pre-production, then production), but only after they have been tested and approved for deployment. If testing fails, then the cycle returns to development and new packages will be created.
3. Evaluating different packaging and deployment strategies

In this section we will provide an overview of possible scenarios that include different combinations for building, packaging and deploying an application. The table below shows these different combinations.

Of course, the typical mainframe strategy is to build incrementally and to create partial packages. Therefore, partial deployment is often favored.

<table>
<thead>
<tr>
<th>S</th>
<th>Build scope / DBB mode</th>
<th>Package Scope</th>
<th>Deployment Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dependency Based Build (Impact Build)</td>
<td>Partial Package based on build outputs / scope</td>
<td>Deployment of the partial package or deployment of several partial packages</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Dependency Based Build (Impact Build)</td>
<td>Full Package based on description of the entire application component</td>
<td>Deployment of a full package (all the artifacts)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Dependency Based Build (Impact Build)</td>
<td>Full Package based on description of all application component</td>
<td>Incremental Deployment of a calculated list of deployable artifacts</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Full build (full Build)</td>
<td>Full package</td>
<td>Deployment of a full package (all the artifacts)</td>
</tr>
</tbody>
</table>

We will now introduce a simple scenario of an application component which will be used to explore the different scenarios.
Reference Application Scenario

Let’s consider application system ‘ACME’ (Figure 13).

It is comprised of

- 100 COBOL programs - PGM001 to PGM100
- 200 COBOL copybooks
- and 50 JCLs

But the application ACME is not standalone:
In the build phase, ‘ACME’ relies on shared copybooks, which are provided by other applications, along with copybooks used for central functions e.g. exception handling to cleanly compile.

And at runtime, ‘ACME’ also calls programs and subprograms from other applications.

Strategy: Partial Packages

Packaging Phase

Let’s consider the following setup (Figure 14): the pipeline processes Application 1, which requires additional parts like copybooks provided by other application teams. Then, the pipeline builds Application 1. This creates a set of binaries and outputs, along with a build report. Finally, a package is then created and published in an artifact repository.

In this scenario:
1. The developer modifies PGM001 and PGM002, which are both BATCH DB2 programs.
2. An **incremental build** is performed at the scope of the application component which updates the build libraries at CICD.APP.FEAT1.LOAD and CICD.APP.FEAT1.DBRM.
3. After the successful build, the **produced binaries** are stored in the artifact repository in a package along with a package manifest.
4. As a result, a partial, delta package is now available in the artifact repository.

**Deployment phase**

![Diagram showing deployment process]

*Figure 15 Deployment of a partial package*

The packaging step creates a partial package with the build outputs of an incremental dependency based build. The package manifest provides metadata to the deployment manager to treat the outputs according to their type.

The deployment manager refreshes the execution environment and updates PGM001 & PGM002, including binding the DBRM members to the database.

At this stage, the deployment manager must capture the entire deployment history, i.e. which partial packages were deployed into a given execution environment and in what order.

At the same time, the inventory provides information about each installed artifact along with their relationships to the relevant packages.

This relational information is key. We need to be able to trace back to the source files that were used to build the binary that is running in the execution environment. If we know of the deployed packages, then the deployment inventory will provide the traceability information and relate the artifact to its source code files.

When building a release, you’ll have consolidated all the requirements and enhancements made during the development of the release. Next, you’ll want to update the acceptance environment. This is where insight into inventory comes in handy as it can show you the active packages in an environment. The deployment manager can calculate the set of packages that need to be applied to the acceptance environment to consistently deploy the release.
Implications of partial packages

As you can see, partial packages must be handled carefully during the deployment phase. It is equally important to understand a package’s contents and also be able to track a package once it is deployed. Doing so will provide you with oversight on inventory status, and the overall environment.

Now let’s assume that a pipeline job is triggered for the release branch after each commit to the central Git server. The developer has developed and tested his modification within his isolated topic branch and most likely has not created a package yet. A build runs, and an incremental package is created. However, given the number of developers working on the release, we can expect several packages. In most cases, packages are independent of each other, don’t overlap and can be deployed and taken forward independently.

In some cases, packages depend on each other and may also overlap. So, when deploying these packages to a runtime environment, it has to be done in the correct order. Given the tight link between the builds and the packages that get created, you cannot skip a package, as doing so would potentially result in a gap in your runtime environment.

Let’s use an example to further clarify this point. The image below (Figure 16) reminds us that the order of the deployment to an execution environment can be crucial. The first flow diagram shows the correct deployment order (the packages get deployed in the order they were built); first APP.FEAT1.20190724001, then APP.FEAT1.20190724002 and finally APP.FEAT1.20190724003.

Figure 16 Correct deployment order of partial packages
So now let’s review what happens when packages are deployed incorrectly. This flow diagram (Figure 17), shows us that the APP.FEAT1.20190724002 package is deployed after the APP.FEAT1.20190724003 the package. And because of this, the most recent update on PGM003 is lost as it has been overwritten by a previous version of itself.

![Figure 17 Incorrect deployment order of partial packages](image)

When execution environments are synchronized / updated regularly, they are much more manageable as are the number of packages being deployed. But when the number of packages differ significantly between environments, it becomes tricky and inefficient to track, order, and apply them.

However, there are different patterns which aim tackle this problem. But first, let’s review the following technical solutions:

- **Strategy 1**
  
  Reduce the number of packages by merging packages into a single new package; merging capabilities must be supported by the deployment manager; this process creates a cumulative package.

- **Strategy 2**
Dedicate a new job (not tied to a commit) to create packages of a wider scope – to include several changes; rather than building after each commit, you instead wait until your tests are positive, and then trigger this now larger job/build; a single cumulative package is created with all of the changed programs from the previous “larger” build; this “larger” build also provides an opportunity to perform the final compile with full optimization options (aka optimize, no debug) of the compiler to create the binaries.

It is likely a combination of strategies will be used for different types of branches and the different types of work going on. Making sure you plan the strategies and have the system handle it based on the type of branch and the phase of development will help simplify the process. Let’s explore applying these strategies in detail in the next paragraphs.

**Embedding the package process into your Git workflow**

We have seen that there are several implications when tying the build and packaging steps together. So far, we’ve proposed the need for a dedicated build for packaging.

But it is still important for us to consider the context in which we create partial packages.

In many cases, developers will use DBB user builds to compile and test their program before committing them to the topic branch on the central Git server. Because we are using DBB user builds, we would expect the quality of the code to have reached a reasonable level before committing. At this time, no packages have been created yet.

Let’s return to our previous Git branch layout (*Figure 18*) and explore some areas where packaging can be implemented:

*Figure 18 Packaging in context of sample Git branch layout*

1. While a development team works on a requirement in a feature branch, it can regularly build and package. The development team uses an environment to test their feature work, by deploying these packages.
Packaging explicitly in this early stage is a matter of taste and can be seen as optional but recommended. We have observed some teams require that all deployments to go through the deployment manager, as this provides clear traceability. However, it may slow down cycle times when the developer iterates in this phase. In contrast to this, some teams choose to apply a lighter, less controlled deployment process for the environment related to development; and instead install artifacts as part of the build.

You may even use a combination of these setups to support both strategies. For instance, a fast path may be well suited for DBB user builds: directly install to an execution environment. Whereas a more controlled process involves the deployment manager for development pipeline builds.

Because of the very nature of this process, traceability will be lost in a shared execution environment across features.

2. When all changes have been implemented and passed all development tests, the team would then continue forward and release the changes. To accomplish this, the team can use a Pull Request with the assembled and integrated changes. In this phase of development, the software is expected to be in a stable state. The build can also be configured to produce a single cumulative package, which deduces all changes made over the lifetime of the feature branch, as part of the pull request. Although, this package may not fully reflect the merged content in release, it can still be used to test the scope of the Pull Request.

3. The build of the “Next Release” branch is triggered after merging. We would expect regular merges to occur between each feature and the “Next Release” branch. As a result, several partial packages would be created.

4. When all features are implemented, fully tested and merged in the branch for the “Next Release”, a “larger” build can be triggered at the end of the release cycle. This larger build could be where the compiled binaries with optimized compile options are produced. This newly created cumulative package will contain the new binaries, i.e. the development done for feature 1, feature 2 and feature 3. In some cases, additional fixes will need to be added, causing another merge in the next release branch. This merge will result in an additional partial package.

Options to create cumulative packages

We previously shown that a cumulative package can be created by merging different packages. However, for this to occur, the functionality needs to exist within the deployment manager. Absent this functionality, let us explore other alternatives to merging of packages.

Dedicated build

We have already explained that we can establish a dedicated build job to assemble the release (see option 4 in previous paragraph). This dedicated build would be triggered prior to declaring the release. The build would compile all changed files including their impacts with the optimize compile options for that release and package the build outputs.
The build and package scope are defined by comparing the current configuration state with that of the previous successful build result – of the dedicated build job. It follows the described approach for incremental builds and partial packages; the only difference being that the build is triggered much later in the process.

But, in some cases, you have to add additional changes like integrating a small fix to the previously built cumulative package of the dedicated build; in this case a partial package will be created on top of the larger cumulative package and both will need to be deployed.

Alternate Calculation of package contents
So far, we have strongly linked the scope of a package to the scope of a build. Remember, a build is incremental, it will process all changes made since a previous successful build, and create various binaries to be packaged, just like in the image below (Figure 19).

![Diagram showing build and packaging have the same reference](image)

*Figure 19 Build and packaging have the same reference*

Let’s now change that pattern and consider different references to calculate the changed files for the build and the files for the package.

The build calculation will reference the changed files that have occurred since the previous successful build – as usual. Whereas the calculation used for the packaging scope references an earlier time period than the one used in the build calculation: Build 0 - the expected package scope is highlighted in yellow:
Figure 20 Build and packaging have different references

In cases such as these, cumulative packages would be created during build 1, 2 and 3, with a reference to the starting point of the release (Build 0).

The starting point for defining the package scope occurs explicitly by the first commit or if a tag in the history of the release branch is set. This tag will most likely be placed when the release is first initiated. Given the calculated list of changed files, DBB would then be able to determine which artifacts have been produced and which need to be added to the package.

An alternative option would be to identify the outputs based on a comparison of the last commit in the master branch. By comparing the “Next Release” branch with the “Master”, we also obtain all the source files that have changed for this release.

We can therefore create cumulative packages, independent of the build scope.

Task oriented creation of packages

Cumulative packages can also be created with information derived from a work item tracking system. If we consider a feature task, we know that it is divided into multiple implementation tasks. If a task can track files that have been changed during its implementation, then it is possible to gather the complete list of changed source files for a given feature. We can therefore create a cumulative package, which scope is the feature.

Strategy: Full Packages

First introduced in chapter two, full packages are another packaging strategy. Here each time we create a package, we also capture the entire content of the application, within the scope of the build.

Packaging phase

As in the previous scenario, the developer again modifies PGM001 and PGM002 (Figure 19).
1. The developer has created a modified version of PGM001 & PGM002
2. The developer then triggers the pipeline with either an impact or a full build. This updates the build libraries at CICD.APP.FEAT1.LOAD and CICD.APP.FEAT1.DBRM.
3. After the successful build, the **entire outputs belonging to the application component** within the build environment are packaged
4. The package is sent and stored in the artifact repository in a package along with a package manifest

The packaging step creates a full package. This includes all build outputs for all 100 programs of the application ACME. The easiest way to identify all outputs is to perform a full build with a clean build environment. Since a full build takes more time, a simulation build, or an application descriptor (which defines the entire scope of an application and its outputs), is recommended.

**Deployment phase**

Let’s now follow the steps performed by the deployment manager when deploying a full package:
The packaging step has created a full package which includes all outputs of all 100 programs. And the package manifest provides metadata to the deployment manager for how it should treat outputs according to their type.

The deployment manager refreshes the execution environment and updates all binaries from PGM001 to PGM100; this includes the DBRMIs, etc.

While this is a more theoretical approach for deploying mainframe applications, it does clearly follow the commonly used strategies of distributed applications. Later on, we will see that deployment of full packages can be done much more intelligently.

In a full deployment scenario, the deployment manager only tracks which packages are deployed to the execution environment in the interest of ensuring traceability. In this scenario an entire reinstall of unmodified artifacts occurs and leads to longer deployment execution times. This scenario does work (even if there are implications for configuration files, as explained later) but this strategy is not optimized. This is because it replaces binaries, that don't need to be changed, in the execution environment.

Implications of full packages

Full packages are best when refreshing out of date environments or when populating fresh environments. Full package makes sense when its artifacts represent an application component that serve a functional area. New releases of the application component will be tested, validated and could ultimately be deployed into production.

Two options exist when creating a full package, both impacted by on the scope of the build.

- Option 1: Full Build

  This is the easiest option when creating a complete / consistent package with all build outputs. A full build can be as simple as: build, then package all artifacts from the libraries, along with the build report. An important consideration for full build feasibility is the reliance on the size of the application system, aka this option should be considered only for small application components.
Option 2: Incremental Build:

Despite full builds being undesirable in most cases, it is still possible to create full packages in an incremental build scenario. The source files are stored in the Git repositories, and the build process holds descriptions of the application outputs, at the metadata level. As an example, DBB can perform a full ‘preview’ build to provide this description of the application outputs. Thus, it becomes possible to create full packages by collecting the outputs of the build, even if they haven’t been produced by the last build. Full packages are indiscriminate, as they can be created from any build type used.

Full packages show their true value when dynamically provisioning new execution environments. There you can just deploy the full package and equip the newly provisioned execution environment.

Also remember, a full package does not necessarily imply a full deployment. It is the inventory of an environment that provides the knowledge to incrementally refresh the execution environment, replacing only the modified modules. Later on, we’ll discuss this further in the Advanced Deployment Strategy section.

Additional requirements on the package

Full packages demand specific requirements for managing application configuration elements, ones such as: DDL, CICS, IMS, MQ definitions, etc. These packages should contain the configuration files that describe the creation of these resources. For example, a database schema should represent the entire creation schema for an application component. This is needed because when deploying a full package, it does not imply that the previous packages have been deployed. Nor does it imply that it is the first deployment for the execution environment.

So, when the deployment of another full package occurs, the database and its tables are already there. The creation script cannot be applied as-is in an existing database schema. Rather, an optimized deployment process will need to occur in order to calculate the updated statements between existing schemas and subsequent ones.

These same modification considerations also apply to definitions for CICS, IMS, and MQ as well as definitions for other runtimes. Depending on the state of the system and at deployment time, only partial modifications may need to be applied.

Therefore, the configuration files need to be treated carefully. One way of ensuring care is for the files to be handled by dedicated packages, ones different from those containing only non-configuration files.

For calculating the delta of the middleware configuration, the deployment manager can make use of additional services like IBM Db2 Object Compare or IBM Db2 DevOps Experience. These services make generating the delta script between the current and the future state of a database schema simple. When dynamic calculation is not possible or not desired, then incremental update scripts will need to be provided. At the same time, full creation scripts will need to be maintained.
Advanced Deployment Strategy: Incremental Deployment with full packages

In previous sections we showed that it is possible to optimize the deployment process for full packages. But in order to do this, a key element is required: the package metadata needs to carry a unique identifier for each artifact. An example of this could be a build ID. When combined with an incremental dependency based build, the new full package will contain modified artifacts alongside unchanged artifacts.

The deployment manager needs to be aware of the inventory in an execution environment; it knows which artifacts are deployed and what their unique identifiers are. By comparing an artifact’s metadata, the deployment manager identifies those artifacts of a package that have not been applied to the environment. Once detected, the deployment manager will then proceed to deploy these artifacts, omitting those ones that have already been deployed and don’t need to be changed.

It’s worth noting that this improved strategy will lead to new requirements for the package and the deployment manager. But to further illustrate incremental deployment with full packages, let’s take a closer look at the elements of this process:

- We know that within the package, the manifest file is very important. This file contains metadata about the artifacts included in the package. In addition to the already described metadata, the manifest stores a unique identifier for each artifact. This unique identifier is later being used as an indicator as to whether the element actually was modified.
- The deployment manager manages the current inventory of the execution environments, knows which artifacts are deployed, and their unique identifiers.

Let’s revisit our package stored in the artifact repository (Figure 21). In addition to the actual binaries, a package will carry a manifest file. This file contains a unique identifier for each artifact, along with the additional metadata, as described earlier.

The output files are what receive a unique identifier.

When selecting this unique identifier, several choices are available.

We have established that the build ID is a possibility.

One might consider using the Git hash; another unique identifier. However, this would not make for a good candidate as the Git hash does not change for a rebuild unless there is an actual change to the source code.
A recommended option could therefore be a combination of unique identifiers, such as: 
\([\text{git hash + build timestamp}]\) or \([\text{git hash + build ID}]\).

When using the build timestamp as part of the unique identifier, we assume, that an incremental dependency based build has been used. In a full build scenario, all binaries will have a new unique identifier, even if the source itself has not changed.

As already stated, even if artifacts have not changed, they will be part of the full package – providing a full representation of the application component.

In the below diagram (Figure 24), we see three full packages stored in the artifact repository. The manifest file carries a unique identifier for each file, which is highlighted in the illustration in the column UUID.

![Diagram of full packages]

**Figure 24 Full packages produces by an incremental build carrying a UUID**

**Deployment phase**

Now let’s take a look at how the deployment manager performs the deployment in an existing execution environment; one that already contains a previous package of the application.

Here the deployment manager captures the current inventory of the execution environment – our target execution environment is currently running APP.FEAT1.20190724001. But let’s assume that the development team wants to deploy the following package: APP.FEAT1.20190724003.

Instead of deploying the entire package to the environment, the deployment manager calculates the delta between those two packages, based on the unique identifiers.

Through its calculation, the deployment manager finds that deploying the modified artifacts PGM001, PGM002, PGM003 and PGM004 will allow the environment to reach the following future state: APP.FEAT1.20190724003.

New files or deletions can also be handled in this calculation phase.
Figure 25 Deployment to an existing environment using delta calculation

Now, let’s imagine that the development team has just setup a new execution environment using infrastructure and middleware provisioning capabilities. Using these exact steps, this improved deployment scenario will support the deployment of the entire application package of your application component to an empty execution environment.

In such a scenario the deployment manager will know the current state of the execution environment, however it would not observe any existing packages being deployed. Because of this, it would acknowledge the absence and deploy the entire application package to the empty execution environment.
Implications of advanced deployment strategy

The strategy of using incremental builds, full packages with manifest file and delta deployments, removes the need to deploy all packages all the time. This strategy also optimizes the deployment phase, as shown in the previous scenario, by dynamically calculating the differences between existing and future states.

Calculating the required delta - those elements that actually have changed and need to be deployed to maintain consistency – gets you the same result as merging several partial packages, but it is a simpler way to calculate and to manage packages.

Conversely, creating and storing full packages, requires more time and more storage in the artifact repository.
Rollback

Backing out a change from an environment is not a trivial task. In many cases it is quite risky to rollback. Very often organizations chose to move forward and make the appropriate fixes to the environment. Despite this, rollbacks do still happen. Let’s examine a rollback in the context of partial and full packages.

For the partial package scenario, the deployment manager requires a thorough understanding of the inventory. The deployment manager will need to prepare a rollback package for each deployment and be able to revert to the previous state of an artifact if necessary. This is because uninstalling an incremental package will result in restoring elements from several other incremental packages.

In the below scenario, rolling back 20190724003 (1) requires either a restoration of PGM003 from a backup package of 20190724003 in a pre-deployed state or a retrieval of PGM003 from 20190724002 (2).

![Rollback scenarios](image)

Figure 27 Rollback scenarios

In a full package scenario, you can easily rollback to a previous version by deploying the full package of this desired version. This full package is stored in the artifact repository.
4. **Summary**

Development and deployment strategies will vary from site to site. However, the trends we continue to see among sites are the standardization and harmonization of development practices among distributed and mainframe development teams. This in effect will lead to more homogenous toolsets and similar practices, without the need for specialized skills.

This whitepaper has illustrated a series of packaging and deployment strategies; some resembling existing mainframe approaches while others are more progressive in their standardization.

In all cases, we saw that the deployment manager continues to deploy to several execution environments, representing the traditional testing stages. However, packages are only defined once and are deployed to several environments. The packages should be treated as a first-class object.

We also highlighted the importance of the artifact repository. The repository is a key element when implementing a CI/CD pipeline. The artifact repository decouples the continuous integration practices from the continuous deployment practices and makes building independent from the deployment strategy you’ve chosen.

You now know that the demand for well-defined delivery pipelines will increase as the use of flexible and dynamic execution environments becomes more widespread. We must recognize that, because manually maintaining and supporting these often short-lived environments can be challenging and inefficient.

Know this: mainframe development teams will continue to encounter new scenarios, ones that would be difficult to support with traditional mainframe development practices. And because of this it will be important for your organization to begin its transformation by first identifying those scenarios that are most important.