BROADCOM
MAINFRAME SOFTWARE

Endevor

Endevor Bridge for Git: Getting Started

White Paper

Table of Contents

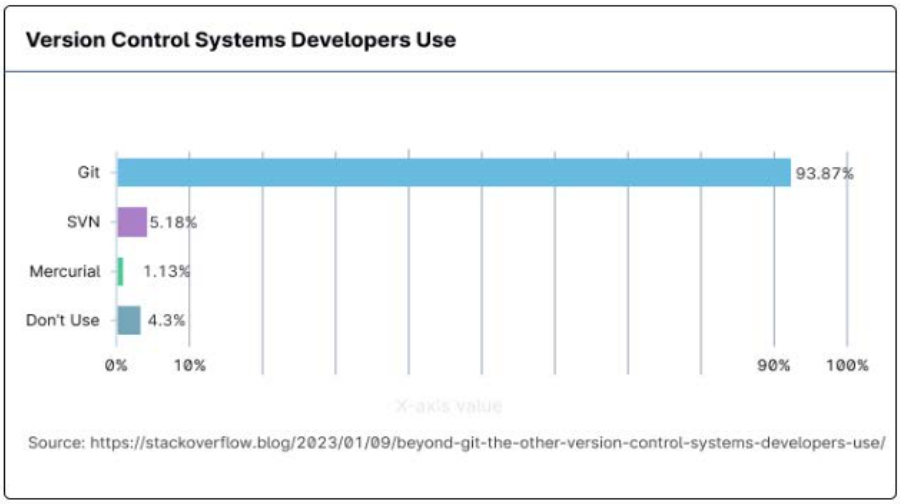
Chapter 1: Introduction

In many companies, a new generation of developers is finding its place to work alongside experienced mainframe developers. Have you been faced with the challenges stemming from the different backgrounds, habits, and needs of the two groups? Established mainframe developers who are familiar with traditional mainframe tools such as Endevor® and are content working on the green screen stand apart from the young developers coming out of university or previous projects. These young developers are familiar with development centered on Git, and modern IDEs such as Visual Studio Code. Imagine their surprise when they first encounter the world of mainframe, and they are asked to edit their code using ISPF. The Broadcom strategy in this dichotomy is to enable choice, rather than force a change. The goal is to merge the two worlds and connect the current silos of mainframe and distributed application development. Enabling the modern developer experience on the mainframe means three things:

- Git for collaboration
- Freedom of choice regarding the IDE for code editing
- CLI to access the mainframe

Even though CLI might not appear to be the first choice for most modern developers, it has been regaining popularity thanks to its cross-platform flexibility and ability to be used in scripts. Popular tools such as Git rely on a CLI, and a console is available in most modern IDEs. Achieving cross-platform consistency means that the same set of tools can be used, no matter the target platform. A consistent set of tools should make life easier for developers spanning distributed applications and mainframe applications, front end and back end.

Figure 1: Version Control, Adapted from The Stack Overflow Blog[1]



This approach delivers the following benefits:

- Attract the new generation of developers to mainframe (easy on-board, common skill-set, standard collaboration practices)
- Support common practices for end-to-end enterprise application development (mobile to mainframe)
- Offer a similar experience across all platforms (distributed, cloud, mobile, mainframe)
- Leverage popular, proven collaboration practices
- Protect investment in mainframe software asset management and lifecycle automation

1. Donovan, Ryan, “Beyond Git: The other version control systems developers use,” *The Stack Overflow Blog*, January 9th, 2023, https://stackoverflow.blog/2023/01/09/beyond-git-the-other-version-control-systems-developers-use.

Chapter 2: Design Thinking

When designing solutions, Broadcom teams want to understand the users and their everyday challenges, expectations, frustrations, and desires. Looking at the individual profiles of people working in Broadcom customer organizations, and observing patterns has enabled the creation and definition of specific personas. These personas fall under two categories of users: developers and tool administrators.

2.1 Developers

The primary developer personas are Michelle and Rob. They are set apart in their background and experience on the mainframe. Michelle is a bright young developer who knows modern languages such as TypeScript, Java, .Net, and C++. She also knows some COBOL, and has enough knowledge of Endevor to get by. From her perspective, she does not understand why mainframe user interface evolution stopped at the green screen. Even in the case of modern mainframe UIs, the user experience is still inspired to a great extent by the green screen. Her previous experience involved Git and modern IDEs. Her colleague, Rob, has been working with Endevor for ages. His interaction with ISPF is so smooth that he does not want to even hear about any modern UIs. He knows what works best for him and is rather skeptical about new solutions claiming an improved user experience.

Persona: Michelle

Profile: Application Developer

- Programs in TypeScript, Java, Net, C#, C++, SQL.
- **Uses Git.**
- Uses Visual Studio Code and other modern IDEs.
- **Not interested in working with mainframe tools or mainframe environments.**
- Interested in building new apps, extending existing applications, and modernization.

Persona: Rob

Profile: Experienced Application Developer

- Has extensive COBOL and/or PL/I programming experience.
- Understands mainframe database application concepts.
- Writes code to create software applications or update existing software applications.
- Enjoys working with **traditional mainframe tools**.

2.2 Tool Administrators

Carl, the Endevor DevOps engineer and administrator, manages and maintains the development environment for Rob. He understands the specifics of that particular Endevor implementation. He will naturally want to protect the investment put into Endevor processes over the years because he can appreciate the complexity of potential conversion to a new system.

Conversely, Todd, the senior operations engineer, does not have as much insight into mainframe development tools and believes moving all application development under Git is the way to go.

Persona: Carl

Profile: Endevor Administrator

- Creates and manages software development build and lifecycle automation.
- Participates in product upgrade and product maintenance to some extent.

Persona: Todd

Profile: Senior Operations Engineer

- Aligns Git setup with development processes.
- **Helps to move engineering teams into Git.**
- **Creates and initializes Git repositories.**

2.3 Michelle versus Rob and Todd versus Carl

Different species of IT professionals from different backgrounds with different approaches, yet they need to work together on common projects. The challenge is to find common ground for all of them, so that they can be as productive and comfortable in their work environment as possible.

The concept of personas is a constant reminder of who we are working for. During a recent validation exercise with Endevor customers, more than 80% confirmed that the current challenge they are facing is how to onboard young talent and have them smoothly collaborate with existing mainframe teams. So, the question we are looking at is, “How can we help Michelle and Rob to work together and be efficient?” Related to that question, “How can we support Todd and Carl in their efforts to provide an ideal development environment to their programmers, both modern and traditional?”

The following sections describe the solution Broadcom offers along with Endevor.

Chapter 3: Modern Deployment Options

The Endevor Bridge for Git is a part of the Broadcom Mainframe DevOps Suite. To enable developers to benefit from the advanced team collaboration capabilities, Bridge for Git connects Git to Endevor through an Enterprise Git Server, such as GitHub, Bitbucket, GitLab, Gitea, or Azure DevOps. Only code that is pushed from the local Git repository to the Enterprise Git repository on designated branches is recognized and propagated to Endevor by the integration.

In this integration, the branching and merging capabilities of Git complement Endevor purely as source control and a collaboration tool. For Rob, Endevor keeps the role of source control, and serves as a build machine and life cycle automation system for the entire application. The user experience remains exactly the same for both Rob on Endevor and for Michelle in Git, including their related tools.

The Bridge for Git provides three primary methods for deployment, catering to traditional environments, private cloud, and Kubernetes orchestration. These versatile deployment options simplify installation, maintenance, and scaling, ensuring the solution seamlessly fits into modern enterprise cloud and CI/CD pipelines.

3.1 Traditional ZIP Archive

For standard server deployments, the application is available as a ZIP archive. This archive contains all necessary components for running the application on a standalone server or virtual machine:

- The application executable (JAR file)
- All required dependencies (`/lib` directory)
- Configuration templates for initial setup

3.2 Docker Deployment

For AMD64 architecture environments utilizing container technology, Bridge for Git 3.0 supports a dedicated Docker image. This allows faster, more consistent deployment across different testing, staging, and production environments, reducing configuration drift and simplifying dependency management.

3.3 Helm Chart for Kubernetes

For enterprises leveraging cloud-native orchestration platforms, a Helm Chart certified for use on Kubernetes environments, including OpenShift, is provided.

The Helm Chart simplifies the deployment and management of complex applications on Kubernetes clusters. It manages the configuration, scaling, and updates of the Bridge for Git application automatically, enabling high availability and resilience.

Chapter 4: System and Migration Considerations

Administrators must review the platform requirements in this chapter and follow the recommended migration process to ensure a successful upgrade to version 3.0.

4.1 Mandatory Platform Requirements

The new architecture requires the following component updates for optimal performance and security:

- **Java Version:** Bridge for Git 3.0 requires Java 17 installed where the application is run. This is a significant change from prior software versions and must be addressed before upgrading.
- **Hardware Minimum**: For a standard server deployment, the minimum recommended setup is a 64-bit architecture with 2 CPU cores and 4 GB of RAM (1 MB stack, 2 GB heap). Capacity must also be provisioned for the operating system.

4.2 Migration Best Practices

Bridge for Git is designed for the automatic migration of existing configurations. However, the following best practices are recommended:

- Always perform a full backup before initiating any upgrade, paying special attention to external databases or the working directory containing the embedded database.
- Run the upgrade process on a test environment first to validate all workflows.
- Review breaking changes and make any necessary adjustments.
- Verify that all connected client software, such as the Zowe CLI Plugin and Visual Studio Code Extensions, are using versions that are compatible with Bridge for Git 3.0.

NOTE: The first synchronization after migration will involve migrating the repository structure to the new 3.0 standard and may take longer than usual.

If you have any concerns regarding your migration, contact Broadcom support for assistance.

Chapter 5: Git versus Endevor: Differences and Challenges

Main takeaways:

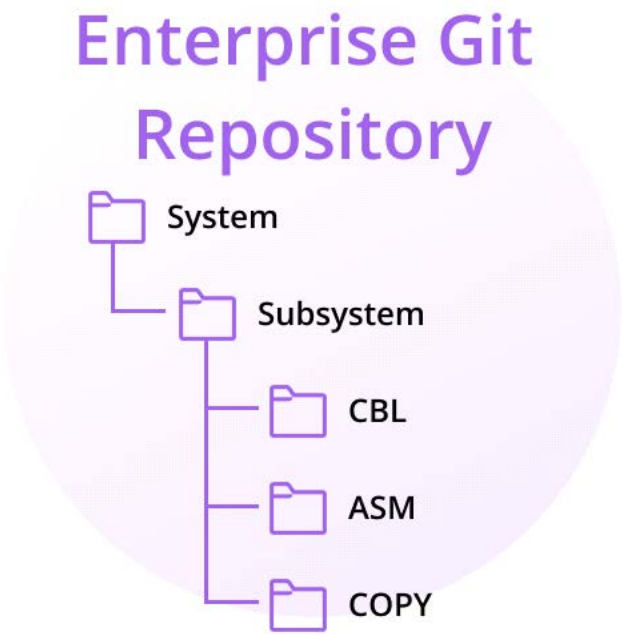
- *Git and Endevor are different in several ways:*
  - *Data organization: hierarchy, partitioning, and size.*
  - *Location of application files and elements:*
    - *Git has all files available in the working directory so developer builds can be performed.*
    - *Endevor only has the elements that are being modified in the working stage—the Endevor map is used to find the other elements needed at build time.*

Git and Endevor in many aspects are inherently different. There is not a direct way to link them one to one. Some differences stem from the underlying operating system, others are due to the SCM design. Before comparing Git and Endevor, let us start with a quick terminology clarification for the purpose of this document.

Endevor manages business applications. These applications can often be independent from one another. In other cases, applications do not have clear boundaries and are melted or blended together. This document uses the word *application* with the assumption that there is a clear understanding of the application *boundaries*, such as the set of all artifacts that make the application including source code, binaries, JCL/Rexx scripts, and so on. The granularity of what is meant by an application depends on the environment.

Now, back to Git and Endevor. In the following section we will first outline the main differences between the two systems, and then review the recommended approaches

Figure 2: Git Repository



The file structure in Git allows for an arbitrary organization into projects with files in a folder hierarchy of virtually infinite complexity and depth of folders, sub-folders, sub-folders of sub-folders, and so on. Alternatively, inventories in Endevor are organized in a much flatter structure that uses systems, subsystems, and these elements are further defined by types. As for the size of projects, usually a Git project can range from a few files to a few thousand, and in some cases it can scale up to 50K files. In Endevor, there is a vast range in the size of systems and subsystems, which can contain hundreds of thousands of elements. It is important to understand that copying Endevor elements to Git will maintain the Endevor-like flat structure. This needs to be considered when deciding on a synchronization strategy for a particular Endevor application with respect to the number of elements.

Another thing to consider is how application code is distributed across systems and subsystems in Endevor. In simple cases, one application corresponds to a system, and subsystems correspond either to individual components, or to releases. However, depending on the particular Endevor implementation, it is common to have one application spanning multiple systems and subsystems, or the other way round (one system or subsystem being shared among multiple applications), or even a combination of both. In this last case, file naming conventions may be used to distinguish which elements belong to which application. These complex situations must be analyzed before synchronizing with Git.

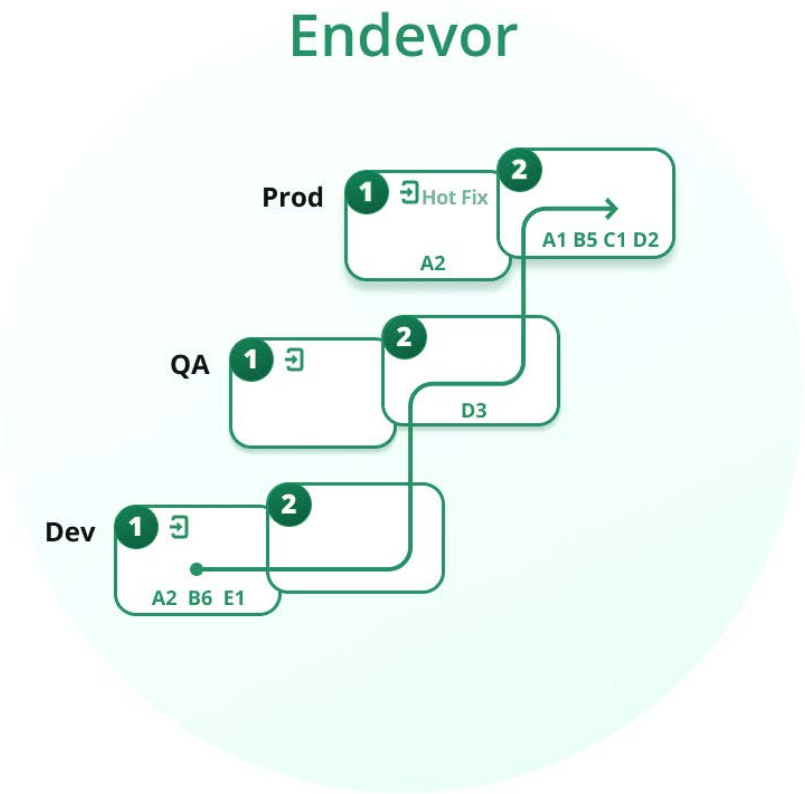
In an Enterprise Git server, it is natural to have all of the files that are related to an application or a component available in one or more repositories for Michelle to clone and to work on locally. All files are then present in her local working directory so it is possible to perform a build at any time. In Endevor, the user experience is not that different. Rob can also access all the elements that he needs to work with. However, the elements are not all physically located in one place. Most elements that are not currently being modified are placed in the last environment and stage of the Endevor life cycle, often named *Production*. When Rob wants to edit an element, he retrieves the element from the working stage, which we will call the *Development environment stage* for the purpose of this white paper. As the modified element progresses through its life cycle, it is promoted from Development to later stages (for example, to QA and to Production) using Endevor packages. As for building, Endevor performs a build of each element after every change, and the build processors are able to search through the Endevor map for the dependencies that are required to generate that element. Because of this, it is not necessary to have all elements in one location. This concept is used in the following sections when outlining a recommended solution.

Chapter 6: Endevor Map and SDLC

It is important to have an understanding of the concept of the Endevor map and the software development life cycle (SDLC). The Endevor map consists of any number of environments and stages that are linked together to create a path where software changes can progress from a development work area to a production deployment.

In the following example, there are three environments: Dev, QA, and Prod. Each Endevor environment usually has two active stages (S1 and S2), of which one is an entry stage (marked with a little door icon in the following figure). The thin green arrow shows how the life cycle is defined. In this case, it follows all the stages one after another up until QA S2. Changes from QA S2 are then promoted directly to Prod S2. Prod S1 is dedicated to hotfix changes, which are allowed to skip the regular life-cycle path and are promoted directly to Prod S2.

Figure 3: Example Environment



The figure also shows how elements can be distributed along the Endevor life-cycle map:

- Version 1 of element C (C1) is only present in Prod S2. If Rob wants to modify this element, he must retrieve it from Prod S2, modify it, and then check it in at Dev S1
- E1 is a new element that Rob has created and added to Dev S1. It is not present in Prod S2.
- Element D2 has been modified and has been promoted, and its version 3 (D3) is now in QA S2 waiting for final testing to complete before it can be promoted to Prod S2.

- Element B5 is currently being modified in Dev S1 as B6. When Rob completes his changes, B6 is promoted through the life cycle as unit testing and integration testing are successfully completed.
- Element A1 is currently being modified in the Development environment Dev S1. At the same time, it is also modified as a hotfix in Prod S1. These two versions of element A are both labeled A2, however they differ in content. Let's say that the version from hotfix stage Prod S1 is the first promoted to Prod S2. In this case, the version at Dev S1 must base its changes on the new A2 version coming from Prod S1 (this corresponds to the Git concept for merge). After that the element coming from Dev S1 becomes A3.

NOTE: The Endevor map can include elements from multiple systems and subsystems.

Chapter 7: Synchronizing Endevor with Git

Main takeaways:

- *The Endevor to Git Synchronization using the Bridge for Git involves the following actions:*
  - *Creating Endevor Connections*
  - *A branch in a Git repository is associated to a location in Endevor defined by environment, stage, system, and subsystem*
  - *A Git branch is associated to an Endevor entry stage for editable branches, or any stage for read-only branches*
  - *There are multiple mapping types:*
    - *Simple mapping*
    - *Basic multi-branch mapping*
    - *Advanced multi-branch mapping*
  - *Three primary synchronization options:*
    - *Synchronize all stages in Endevor map*
    - *Synchronize only the elements from the work environment*
    - *Mirror the work environment exactly*
  - *The following additional synchronization options:*
    - *Generate after update*
    - *CCID, Comment, and From-location values*
    - *Commit Message*
    - *Add Element*
    - *Git Cache*
    - *Webhooks*

To enable Rob and Michelle to collaborate on a project, Carl and Todd need to work together to connect Endevor and Git, associating one Enterprise Git repository to a location in Endevor. This association is done by creating a Git-Endevor mapping using the Endevor Bridge for Git. Based on the structure of the selected Endevor location and on the team's development processes, one of the following mapping types will be used.

7.1 Creating Endevor Connections

An Endevor Connection consists of a base URL and a configuration of Endevor Web Services. Administrators such as Carl and Todd must work together and set up the correct Endevor Connections so that Mapping Administrators use the correct configuration of Endevor when mapping an inventory.

Additionally, both Mapping Administrators and Developers (or anyone who works with elements in mappings) need to ensure that they provide their credentials for the particular Endevor Connection. Their credentials are used for the synchronization of changes from Git to Endevor.

NOTE: When a Mapping Administrator creates a mapping, their credentials are used to synchronize changes from Endevor to Git. If you do not want to use individual credentials for this communication from Endevor to Git, you can use certificates that are associated with a service account for such connections.

7.2 Types of Git-Endevor Mappings

When creating a new Git-Endevor mapping, you can choose from three primary templates depending on your architectural needs:

- Simple mapping: A streamlined configuration that restricts a single system and subsystem to a single branch.
- Basic multi-branch mapping: An intermediate setup for managing one or more Endevor systems and subsystems with a Git repository.
- Advanced multi-branch mapping: The most flexible and commonly used option. It is the recommended choice for production environments as it provides full control over complex workflows.

NOTE: The simple and basic multi-branch mapping templates are generally intended for Proof of Concept (PoC) purposes due to their limited scope.

Beyond the standard templates, you can also define a mapping by uploading a custom JSON definition file. The most efficient workflow for this is as follows:

1. Download an existing mapping configuration.
2. Modify the parameters within the JSON file to meet your requirements.
3. Upload the edited file to initialize the new mapping.

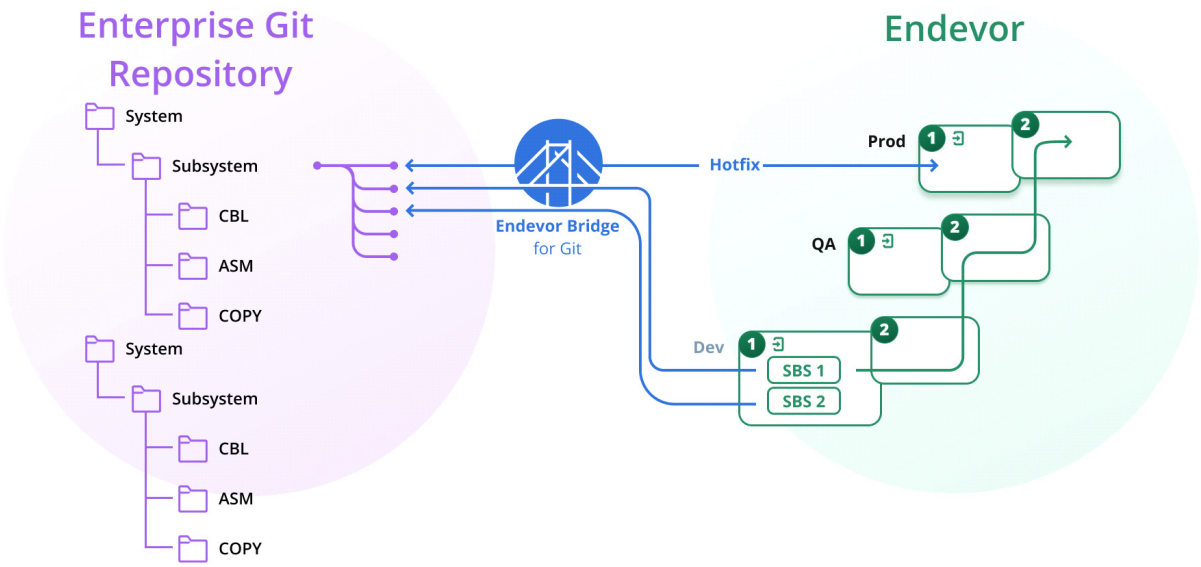
7.2.1 Advanced Multi-branch Mapping

Advanced multi-branch mapping associates one enterprise Git repository to any number of Endevor systems and subsystems. Multiple synchronized branches can be created, corresponding to different Endevor environments, stages, systems and subsystems. This mapping type should be used in most production use cases, as it supports most typical Endevor life cycle configurations. Teams who want to synchronize sandboxes (an Endevor sandbox is defined as a subsystem) to branches will benefit from advanced multi-branching.

This mapping type is the most flexible in terms of branch configuration. Some synchronized branches might only differ in environment and stage (similar to the previous hotfix example), while other synchronized branches will differ in the system or subsystem definition (the advanced branches).

NOTE: To keep the following visual simple, subsystems are only displayed in the Dev environment, but they exist in all environments and are not necessarily the same across the life cycle.

Figure 4: Advanced Multi-Branch Mapping: Multi-System Definition per Branch

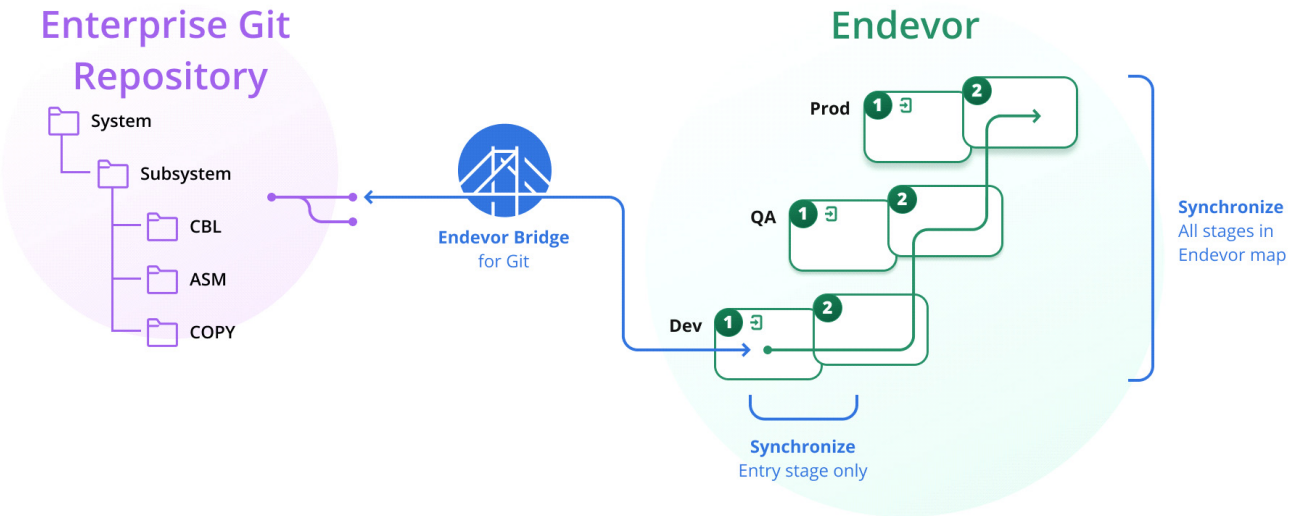


7.3 Synchronization Options

To address the differences and challenges outlined in the section Git versus Endevor: Differences and Challenges, and to enable synchronization of very large Endevor inventories, the Bridge for Git offers the following options:

- Synchronize all stages in the Endevor map, or
- Synchronize only elements from the working environment, either as an exact mirror of the environment or as an incrementally growing repository of files starting with the initial contents of the environment.

Figure 5: Bridge for Git Options



7.3.1 Option 1: Synchronize All Stages in Endevor Map

All files will be available in the Enterprise Git repository

This first option is suitable for applications that have a reasonable number of Endevor elements that all can be synchronized in the enterprise Git repository, and consequently made available in the local working directory of Michelle. Regardless of the life-cycle stage the elements in the Endevor map are placed in, they are all synchronized in an enterprise Git repository through the Development entry stage. The elements will be placed in folders in the repository organized by type.

7.3.2 Option 2: Synchronize Only Elements from Work Environment

Only elements from Development entry stage will be in the enterprise Git repository

For scenarios where the number of elements in Endevor is too large and you do not want them all synchronized to the Enterprise Git repository, Bridge for Git provides the ability to synchronize elements from only the Endevor Development entry stage, ignoring all the other elements that are located in higher stages of the Endevor map.

This option ensures synchronization of a reasonable amount of data while providing Git users with access to the elements that they require. In general, the Development entry stage only holds around 5% or less of the total number of elements.

If Michelle needs an element that is in one of the higher stages of the Endevor map, she can get it in one of the following ways:

- **(Recommended)** Michelle can preempt synchronization by using the Bridge for Git plug-in for Zowe CLI to copy elements from up the map directly into the synchronized repository, with or without its dependencies.

  Example Syntax: `zowe ebg mapping add-element` *`elementName`* `--repo-context` *`ORGANIZATION`* `--repo` *`REPONAME`* `--type` *`TYPENAME`*
- **(Recommended)** Michelle does everything directly from Code4z (Visual Studio Code). Using the Bridge for Git Explorer extension, she is able to see the elements that exist up the Endevor map from her work environment and can add the required element to the synchronized repo, optionally with dependencies. Once it is added to the remote synchronized repository, Michelle can pull the missing element to her local working directory. When she makes changes in a subsequent commit and pushes them back to the Enterprise repo, the element is added to the Development entry stage.
- Michelle uses the Endevor plug-in for Zowe CLI to copy the element to the Endevor Development entry stage, using the command generate with copyback.

  Example Syntax: `zowe endevor generate element` *`elementName`* `--env` *`DEV`* `--sn 1 --sys` *`SYSNAME`* `--sub` *`SUBNAME`* `--type` *`TYPENAME`* `--copy-back -i` *`ENDEVOR`*

  The next time a synchronization runs in the Bridge for Git, it will automatically bring this element to the synchronized repository from where Michelle can pull the elements to her local repository.
- Michelle uses the Endevor plug-in for Eclipse to retrieve the element to a local project and select the option to immediately add it to the Endevor Development entry stage. The next time a synchronization runs in the Bridge for Git, it will automatically bring this element to the synchronized repository from where Michelle can pull the elements to her local repository.

As a general rule, with 7.3.2 Option 2: Synchronize Only Elements from Work Environment, the Enterprise Git repository only contains elements from the Development entry stage. On top of that rule, the Bridge for Git identifies elements which Michelle has already modified or pulled into the repository on demand, and those elements remain synchronized in the Enterprise Git repository even if they are promoted to a higher stage in the Endevor lifecycle and disappear from the Development Entry stage. Unlike 7.3.3 Option 3: Mirror-Mapping Mode, this ensures that Michelle's user experience is more

consistent in the sense that her elements are not removed by the synchronization process in the background without her knowledge. Her elements remain in the Enterprise Git repository unless she or another Git user removes them (in which case they are still present in one of the higher stages in Endevor) or unless they are removed from all Endevor stages completely.

7.3.3 Option 3: Mirror-Mapping Mode

Only elements present in the selected stage will be in the Enterprise Git repository

Mirror-mapping mode maintains a strict, 1:1 reflection of a specific Endevor inventory location within a Git repository. Unlike standard mapping modes that include inherited elements from higher environments, mirror-mapping isolates only the elements physically residing at a designated Stage and System to create a high-fidelity, exact image of that environment. This mode may be beneficial for precise environment auditing or code location detection, as it ensures that the Git branch remains a perfect, real-time replica of the mainframe source—automatically reflecting every addition, modification, or deletion within the targeted Endevor stage.

7.4 Using Filters

In the Mappings, Synchronization options, and scenarios described previously, the Bridge for Git allows for a reduction in the number of synchronized elements through the use of filters that are based on element type and element name. When creating a mapping, Bridge for Git makes a request to Endevor and provides a list of the available element types for each system or subsystem that is included in the inventory of the mapping. At the same time, mapping administrators can specify names or parts of names using wildcards (standard Endevor wildcards apply, * [asterisk] and % [percent]) in an entry field for an element name.

For example, if Michelle only needs to work on COBOL files and copybooks with the name prefix of FIN, filtering by COBOL and CPY type (depending on the file extension setting in Endevor) and `FIN*` is applied. Filtering can substantially reduce the number of synchronized elements, which improves the performance of the initial synchronization process, makes the Git repository less cluttered, and provides a better user experience to Michelle.

NOTE: If a developer wishes to add an element or elements of a different type than is present in a mapping that uses filters, they can. The element will be added and synchronized in the appropriate folder type.

7.5 Bi-directional Synchronization

For read-write branches, in all mapping types and synchronization options, once initialized, the synchronization goes both ways between Git and Endevor. In case of read-only branches, the synchronization only goes from Endevor to Git.

7.5.1 Synchronization from Git to Endevor

This Git-Endevor forward synchronization makes Michelle's changes (done in Git) available to Rob and subsequently have them managed by Endevor. The synchronization is triggered immediately after Michelle's changes get merged into the synchronized branch in the synchronized enterprise Git repository.

7.5.2 Synchronization from Endevor to Git

This synchronization from Endevor to Git, also called sync-back, makes Rob's changes (done in Endevor) available to Michelle in Git. The Bridge for Git offers two mechanisms for the sync-back:

- **Regular (polling) sync-back**: Runs at a desired frequency, scans the relevant Endevor inventory and retrieves all new changes back to the Git repository.
  - An administrator can set a sync-back interval with a specific start date and time for the whole application.
  - Alternatively, mapping administrators can set sync-back intervals for their mappings.
- **Event-based sync-back**: Using the Mainframe Webhook server, actions happening in Endevor are monitored and the sync-back is immediately triggered for any new change in the synchronized inventory.

7.6 Additional Synchronization Options

The mapping administrator can configure several options that affect the synchronization of changed elements to Endevor.

7.6.1 Generate after Update

The mapping administrator can toggle `Generate after Update` on for a mapping so that elements from the mapping that are successfully synchronized to Endevor will be generated. Generate actions are performed with the same user ID, CCID, and comment as the original synchronization.

7.6.2 CCID, Comment, and From-location

The mapping administrator can also set expressions or string literals that are used for the CCID, Comment, and From-location for elements changes in the mapping. These values will be used instead of the default behavior, which is described in the following section.

7.6.2.1 Default Behavior

Bridge for Git allows developers to provide CCID and Comment values in the commit field for their changes using the “CCID=” pattern. The first eight characters following “CCID=” are used for the CCID, while other values are populated to the comment field. The first seven characters of the Comment field are reserved for the first seven characters of the git hash for a commit. The from-location value uses “Bridge for Git” plus the identifier that is specified in the application.yml.

7.6.3 Commit Message

The Commit Message option provides enhanced traceability for elements synchronized from Endevor back to your Git repository. This setting has transitioned from defining a complete message to a prefix-based system, allowing you to append specific tracking data to your defined prefix. For administrators, this ensures a consistent audit trail in Git while still allowing for custom identifiers (even emojis [Unicode characters]) to be included in the synchronization logs.

7.6.4 Add Element

Add Element functionality allows developers to bring elements from higher stages of the Endevor map into their local synchronized Git repository on demand. This function may be useful when only a subsection of Endevor inventory is mapped. You can limit the synchronization scope using three different mechanisms: mapping mode ('Work environment only' mode), type filter, and name mask.

Within the **Synchronization Options** tab, administrators can now explicitly enable or disable this functionality for specific branches or for the entire mapping. This gives teams control over whether developers can use the Endevor Explorer or Bridge for Git plug-in for Zowe CLI to manually pull elements into their workspace.

7.6.5 Git Cache

This infrastructure option allows users to toggle between a single local Git repository and a double local Git repository.

- **Single Channel:** One local repository is used for all mapping synchronizations
- **Double Channel:** Two local repositories are used, one for each synchronization direction (default for Bridge for Git v3 mappings). While it requires more storage, it significantly improves the throughput of changes.

7.6.6 Webhooks

Mapping administrators can also configure webhooks for a particular mapping to send information about events and their results in a mapping to a specified endpoint. One practical use case of the webhooks is to create a webhook that sends information of successful synchronization events to a Jenkins pipeline that kicks off automated testing. In this way, more of the development process is driven through the developer's actions in a Git-Endevor mapping.

7.7 Enhanced Multi-Branching

With the new Bridge for Git 3.0, the same code change is now represented by a single, unique commit as it progresses through the Endevor lifecycle (for example, from DEV to QA). This simplifies the merge process and makes branch comparisons more accurate and reliable.

7.7.1 A Single Commit Per Version

The most critical change is generating a single commit per unique element version. When an element is promoted in Endevor (for example, from Stage 1 to Stage 2), the commit history remains traceable. This delivers the following immediate benefits:

- Git's native tools (Pull/Merge Requests) accurately compare branches synchronized to different Endevor stages, eliminating irrelevant changes and highlighting only true deltas.
- Standard Git merge functionality can be used without manual selection. The single-commit history ensures clean merges and promotions using the Git server, which triggers the corresponding Endevor action.
- Auditing tools only process the change once, simplifying compliance and improving performance for downstream systems.

7.7.2 Configuration and Mapping

The advanced multi-branching capability uses the following enhanced configuration:

- Aliases are integral to managing comparisons. They ensure consistent naming of systems and subsystems in the Git repository structure regardless of the Endevor stage, which is vital for accurate branch comparisons across the mapping.
- The architecture supports multiple developers working in parallel Endevor sandboxes that are mapped to separate Git branches. This ensures that conflicts are detected and resolved in the familiar Git environment before synchronizing back to Endevor.

7.8 Separation of Sync and Refresh

Updating the local Git cache and communicating with the mainframe are logically split, optimizing the process for faster change throughput and improved manageability visible in the **Mapping Activity** dashboard. This separation divides the process into two phases:

- **Evaluation of Endevor Changes**: Calculates Endevor deltas and isolates mainframe processing time
- **Endevor-to-Git Synchronization**: Pushes identified changes to Git for faster throughput

The new architecture also includes a Git Cache option. You can now select either a single local Git repository or a double local Git repository. The double local repository requires more disk space, but improves the throughput of changes by preventing sync from blocking refresh operations. The overall handling of mappings with large numbers of branches has been significantly optimized, leading to improved system performance and reliability.

7.9 Building Local Changes

Main takeaways:

- *What is the equivalent of Generate for a developer working in Git?*
- *Extract build information from Endevor into scripts, alongside application code*
- *Developers can make changes, compile in USS, and view the compilation listings*

With Git, Michelle is used to having all files available in her local working directory. This availability enables her to build and test her changes at any time before committing them and sharing them with the rest of the team. In Endevor, a generate action can automatically run to build the given element immediately whenever an element is changed and the changes are saved. Even though the experience is different, both Michelle and Rob are used to having timely feedback on their changes. The challenge here is that in most cases, Endevor applications cannot be built outside Endevor. So how can Michelle validate her changes in Git before committing and pushing them to Endevor?

Enter Team Build. Team Build enables teams to extract their Endevor-defined build processes into scripts that they can manage alongside the source code in their repository. Using exportz, teams specify the inventory location where they want to generate build scripts.

Then, with some basic user information configuration, developers can make changes to elements and build those changes, along with dependencies, in a USS directory. Subsequently, they receive listing information to their workspace and can take action to fix any issues with their element changes. Following a successful build, Michelle and her fellow team members will have the assurance that they can push their changes for merging and synchronization to Endevor.

7.10 A Day in the Life of Michelle

Main takeaways:

- *Michelle's one-time setup tasks:*
  - *Provide credentials for the Endevor connection that is used by the mapping*
  - *Clone the enterprise Git repository*
  - *(Optional) Set up a local pre-check to help prevent synchronization issues*
- *Michelle's workflow:*
  1. *Create a feature branch*
  2. *Edit Code with VS Code*
  3. *Build with Team Build*
  4. *Test*
  5. *Git pull to integrate other users' changes*
  6. *Iterate the above as needed*
  7. *Git commit*
  8. *Git push with a pre-check*
  9. *Enterprise Git server: Create a Pull request*

Michelle is joining Rob's team to help in a long-term project developing new features for a mainframe application. She usually works on front-end, but she also has knowledge of COBOL that she can use in this assignment.

After Todd and Carl set up the Git-Endevor mapping for the application, Michelle is directed to Bridge for Git where she adds her credentials to the Endevor Connection that is used by the mapping for synchronization. She can alternatively add her credentials through the Endevor Bridge for Git plug-in for the Zowe CLI. Michelle views the details of the mapping and the options that are set, and then she navigates to the Git repository.

This Git repository is managed in an enterprise Git server from which Michelle can clone. She creates a development branch in her local Git repository and updates her Zowe CLI user profiles. Michelle can now edit elements in her working directory.

Michelle uses her preferred IDE, for example Visual Studio Code, and she makes the necessary changes on selected files. She can optionally set a non-default processor group for any of the elements by navigating to the `bridge.json` file in the element type folder and changing the processor group entry for the particular elements. The processor group will not only be used for the build action, but also for the generate action (if the generate after update feature is enabled for her mapping).

Now that she has edited the files, Michelle can run a build by using the Team Build scripts that are incorporated into her repository. To do this, she runs the required command in her CLI. For example, if she wants to run a build of the changed files and any dependencies, she will type `syncz task build` in the command line. Once the build finishes, she can view the listing by selecting the link in the terminal output or in the listings directory that is generated in her repository.

Following these steps, Michelle is ready to commit and push to the Enterprise Git repository. In the Git commit message field she provides a CCID and comment that will be propagated to Endevor by specifying `CCID=<value> <comment>`. Alternatively, she does not enter any value and the mapping uses the expressions her mapping administrator set for the mapping. She then commits and pushes, with the push validated by a precheck. Her changes are synchronized to the enterprise Git server and there she is able to create a pull request so that her changes can be reviewed and merged in the synchronized branch. The Bridge for Git then immediately propagates her changes to Endevor and the changes are available to the rest of her team, regardless of whether they use Git or Endevor.

Michelle's experience working on a mainframe application using a repository synchronized with Endevor is no different from what she does when doing code changes on the front end.

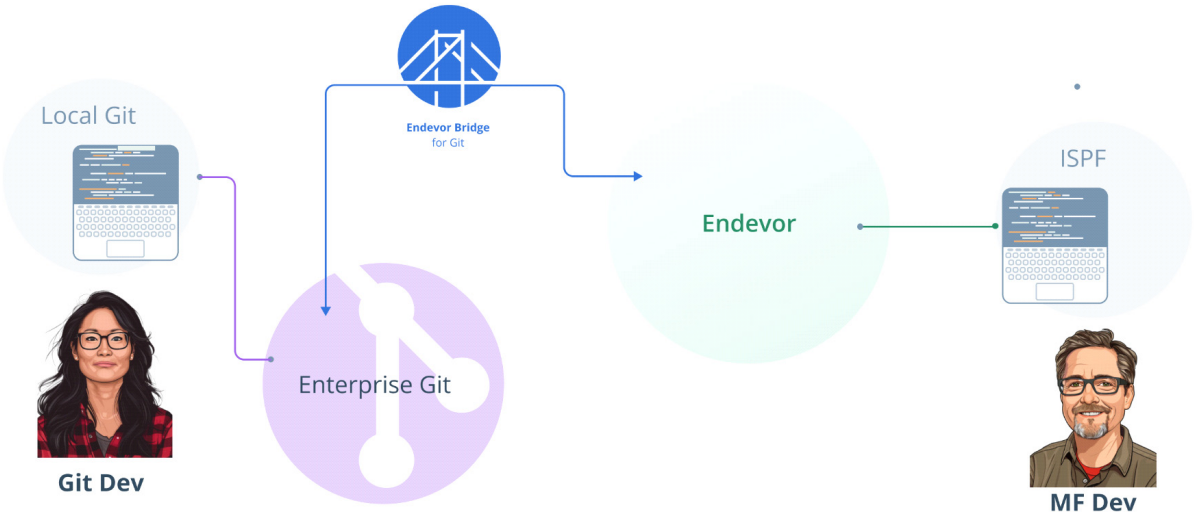
7.11 Out-of-Sync Changes Management

Main takeaways:

- *When Michelle's pushed changes are in conflict with Rob's changes done in Endevor or contain a user error:*
  - *Endevor is the single source of truth.*
  - *Out-of-sync changes are reverted and placed in a revert branch.*
  - *Michelle can fix any errors and/or use Git merging capability to merge with the latest changes from Endevor and push again.*

Let's now take a look at how collaboration between Rob and Michelle will work. Rob edits his code using the green screen or Eclipse UI and saves his changes directly in Endevor the way he is used to. Michelle works alongside him, in her favorite IDE, and pushes her changes in the enterprise Git repository.

Figure 6: Collaboration



To prevent out-of-sync issues when pushing her changes to the synchronized branch, Michelle is encouraged to use common Git practices—always fetch the latest changes from the enterprise Git repository and merge her code with them before pushing. The frequency of Endevor to enterprise Git server synchronization determines how up to date the synchronized branch will be most of the time. The more frequent synchronization, the smaller the probability of a conflict will be.

An issue in synchronization still can occur, be it because of a conflicting change performed in Endevor after the last synchronization, or because it is rejected by Endevor due to a user error (for example, an incorrect type or record length). In these cases, Endevor remains the single source of truth. The Bridge for Git delegates the resolution to the enterprise Git server and leverages its powerful merging capability by always reverting conflicting changes back to a consistent state. A new revert branch gets created containing Michelle's changes, so that she can pull the latest version, correct anything if necessary and finally merge before pushing again.

Figure 7: Revert Branch Gets Created

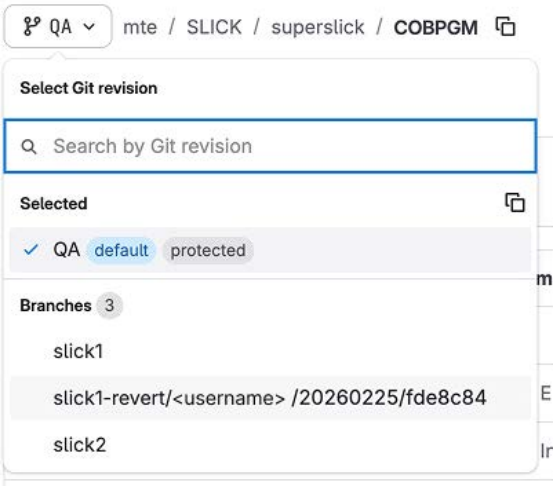


Figure 8: Information About Reverted Commits

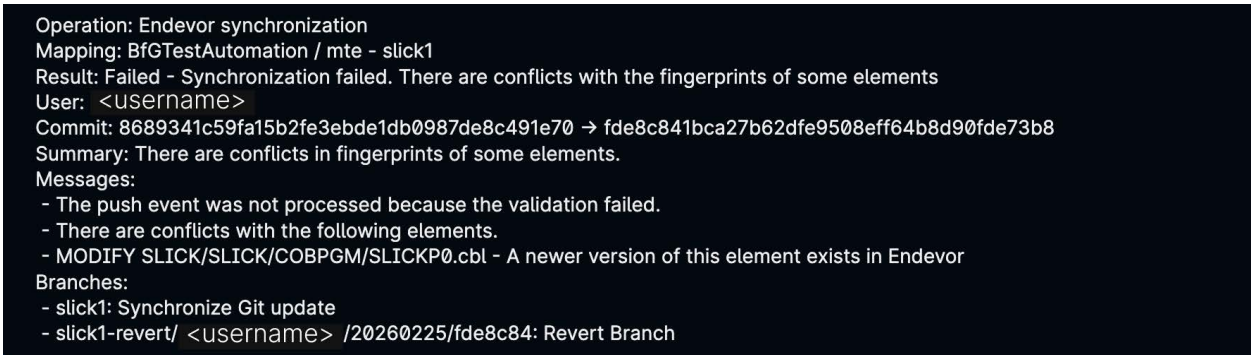
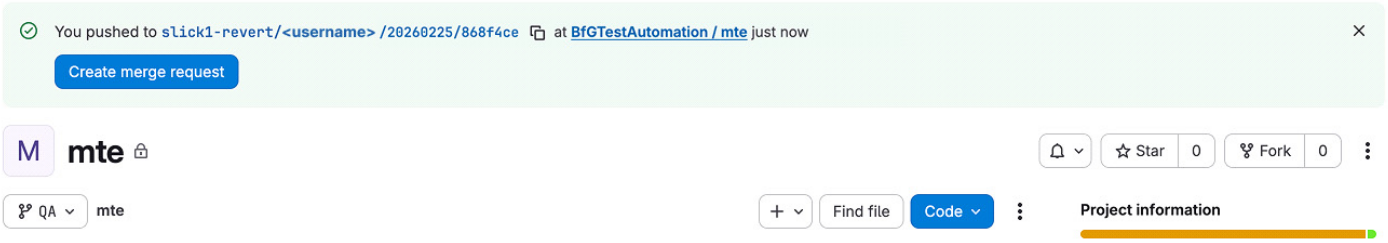


Figure 9: Squash and Merge the Commits from the Revert Branch



Revision History

Endevor-SCM-WP105; March 23, 2026

- Renamed document to *Endevor Bridge for Git: Getting Started*.
- Made the following revisions for the Bridge for Git v3.0 release:
  - Added Chapter 4, System and Migration Considerations
  - Added 7.3.3 Option 3: Mirror-Mapping Mode
  - Added 7.6.3 Commit Message
  - Added 7.6.4 Add Element
  - Added 7.6.5 Git Cache
  - Added 7.7 Enhanced Multi-Branching
  - Added 7.8 Separation of Sync and Refresh
  - Updated Figure 1, Version Control, Adapted from The Stack Overflow Blog
  - Updated Chapter 2, Design Thinking, including persona images.
  - Updated Chapter 3, Modern Deployment Options.
  - Updated Figure 2, Git Repository
  - Updated Figure 3, Example Environment
  - Updated Chapter 7, Synchronizing Endevor with Git
  - Updated 7.1 Creating Endevor Connections
  - Updated 7.2 Types of Git-Endevor Mappings
  - Updated 7.3 Synchronization Options
  - Updated 7.3.2 Option 2: Synchronize Only Elements from Work Environment
  - Updated 7.9 Building Local Changes (retitled from *Developer's Build and the Use of Work Areas*)
  - Updated 7.10 A Day in the Life of Michelle
  - Updated 7.11 Out-of-Sync Changes Management
  - Updated Figure 6, Collaboration
  - Updated Figure 7, Revert Branch Gets Created
  - Updated Figure 8, Information About Reverted Commits
  - Updated Figure 9, Squash and Merge the Commits from the Revert Branch

Endevor-SCM-WP104; December 29, 2025

- Updated Figure 7, Revert Branch Gets Created
- Updated Figure 8, Information About Reverted Commits
- Updated Figure 9, Squash and Merge the Commits from the Revert Branch

Endevor-SCM-WP103; June 23, 2022

- Updated Chapter 5: Git versus Endevor: Differences and Challenges
- Updated Chapter 7: Synchronizing Endevor with Git
- Updated 7.1 Creating Endevor Connections
- Updated 7.4 Using Filters
- Updated 7.5.2 Synchronization from Endevor to Git
- Updated 7.6 Additional Synchronization Options
- Updated 7.9 Building Local Changes

- Updated 7.10 A Day in the Life of Michelle

Endevor-SCM-WP102; February 1, 2021

- Updated 2.2 Tool Administrators
- Updated Chapter 7: Synchronizing Endevor with Git
- Updated 7.3 Synchronization Options
- Updated Chapter 4, Summary

Endevor-SCM-WP101; March 20, 2020

- Updated Chapter 1, Introduction
- Updated Chapter 2, Design Thinking
- Updated Chapter 3, Modern Deployment Options
- Updated Chapter 4, Summary


BROADCOM®