Using CA Endevor SCM in Agile Software Development

Mainframe application development is evolving from the traditional waterfall process to the agile process. To transition to agile, development teams need a source code management tool that helps them manage concurrent development activities, while keeping the main development branch in a buildable state. CA Endevor® Software Change Manager (CA Endevor SCM) has been, and continues to be, the mainframe source code management tool of choice, with its superior support for traditional development as well as for agile development.

This paper summarizes:

- Key agile software development practices: continuous integration; continuous delivery; and continuous deployment.
- Current trends in agile workflows that support these practices, as illustrated by the Git Branch-per-Feature flow (GitHub flow) and the GitFlow models.
- How CA Endevor SCM can fully support these key agile practices, with workflows that are comparable to Git models.

Agile Practices

Agile supports rapid and flexible response to change. As a result of implementing agile, new development practices have emerged, including continuous integration, continuous delivery, and continuous deployment.

Continuous Integration

Continuous integration is a software development practice where developers integrate code into a shared repository frequently, preferably several times a day. This provides the opportunity to verify each integration by an automated test and automated build. Continuous integration helps to keep the production build stable. As a result, continuous integration has become a standard of agile software development.

In addition, the continuous integration approach supports best practices that keep your application deployable anytime. Continuous integration can even enable automatic deployment to any stage including production whenever any change is made. These practices are known as continuous delivery and continuous deployment.

Continuous integration significantly speeds up development without any trade-off in code quality and allows the development team to spend more time on developing new features. The speed-up is achieved by leveraging frequent automated builds and deployments (as verifications). In addition, because of frequent code integrations, it is easier and faster to find out the root cause of potential integration conflicts and errors. The faster that conflicts and errors are corrected, the more time is available for new feature development.

Benefits

- Reduces time and effort for integrations of code changes
- Allows earlier identification and prevention of defects
- Reduces overhead across the development and deployment process
- Allows a quick feedback on every change
- Reduces manual testing effort
- Helps collaboration between team members
- Prevents conflicts in different branches as they are frequently integrated

Best Practices

- Maintain a single source repository
- Automate the build; make the build fast and tested automatically
- Integrate frequently to the baseline
- Test in a clone of the production environment
- Provide easy access to the latest artifacts and deliverables
- Ensure that every team member can see the results of the latest build

Continuous Delivery

Continuous delivery is a software development practice that helps development teams produce software in short cycles and keeps the software product in a state where it can be reliably released at any time. Continuous delivery is based on top of continuous integration – it leverages automation to have a code which is releasable after every change. Together with continuous integration, it enables building, testing, and releasing the software faster, more frequently, and with minimal overhead.

Continuous Deployment

Continuous deployment is the next step of continuous delivery. Continuous delivery ensures that the software could be released at any time, but it is up to the team when the software will go to production. Continuous deployment ensures that the process of shipping the software to production is automated, so that deployment to production does not require any manual steps.

Workflows – Using Git

In agile software development, one important focus is the support of effective development, while always maintaining the main development branch (that is, the trunk or master) in a buildable state.

There are different guidelines and best practices for fulfilling both requirements, each of which has its benefits and drawbacks. Typically, implementation of any such strategy needs to respect the technical and environmental constraints of the organization. Thus, it is always necessary to consider the pros and cons of any best practice in the context of the particular organization, rather than to blindly follow it.

Typical Git Workflow

Today, the best practices associated with the open source version control system Git are considered as standard for agile development. In short, the typical workflow used when working with Git leverages branching capabilities. For any new feature or fix, a new branch is created where isolated development takes place. This branch can be either a local branch for a developer or a remote public branch. This means, that other developers are not distracted when working on different parts of the software. Also, it ensures that the master branch cannot be broken by changes made in a feature branch. To follow continuous integration concepts, feature branches should be short-lived and frequently merged into the master branch.

The developer can perform comparable steps in CA Endevor SCM. The following table lists Git terms used to describe the workflow and equivalent terms used in CA Endevor SCM:

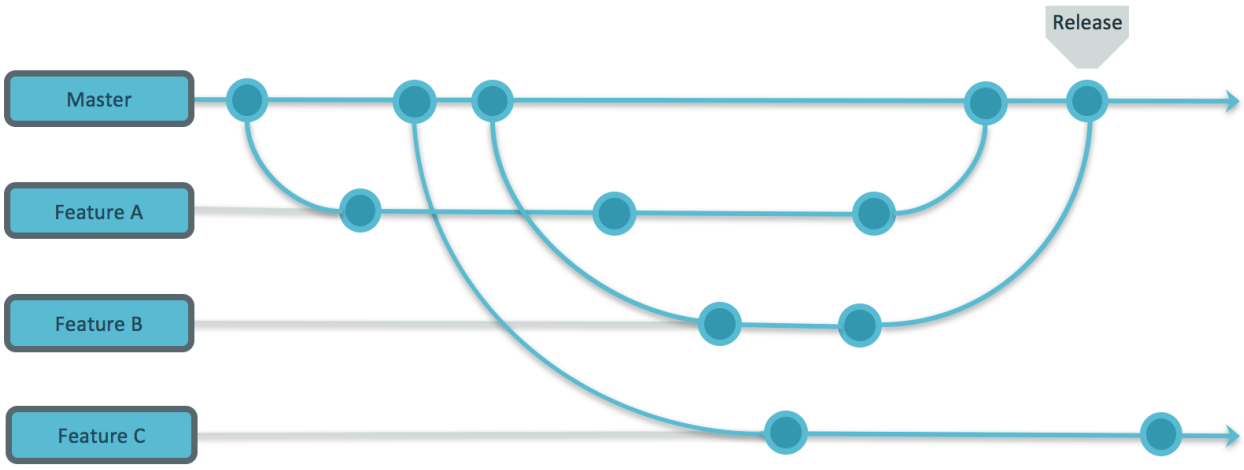
Git term	CA Endevor SCM equivalent
Branch	Branch Subsystem
Pull request	Package approval

Branch-per-Feature Flow

The Branch-per-Feature flow (sometimes known as GitHub flow) follows the guidelines previously described for the typical Git flow. For any new feature or fix, a new branch is created. Once the feature or fix is done, the branch is merged into the master and deleted.

The following graphic illustrates the Branch-per-Feature flow. Horizontal lines represent the branches. Each circle on a branch line represents the point that a change is committed to that particular branch – it simply means that there is some activity happening in the particular branch. Features A and B are developed within their respective branches, completed and merged back to the Master. Feature C is not completed and does not influence release of the product.

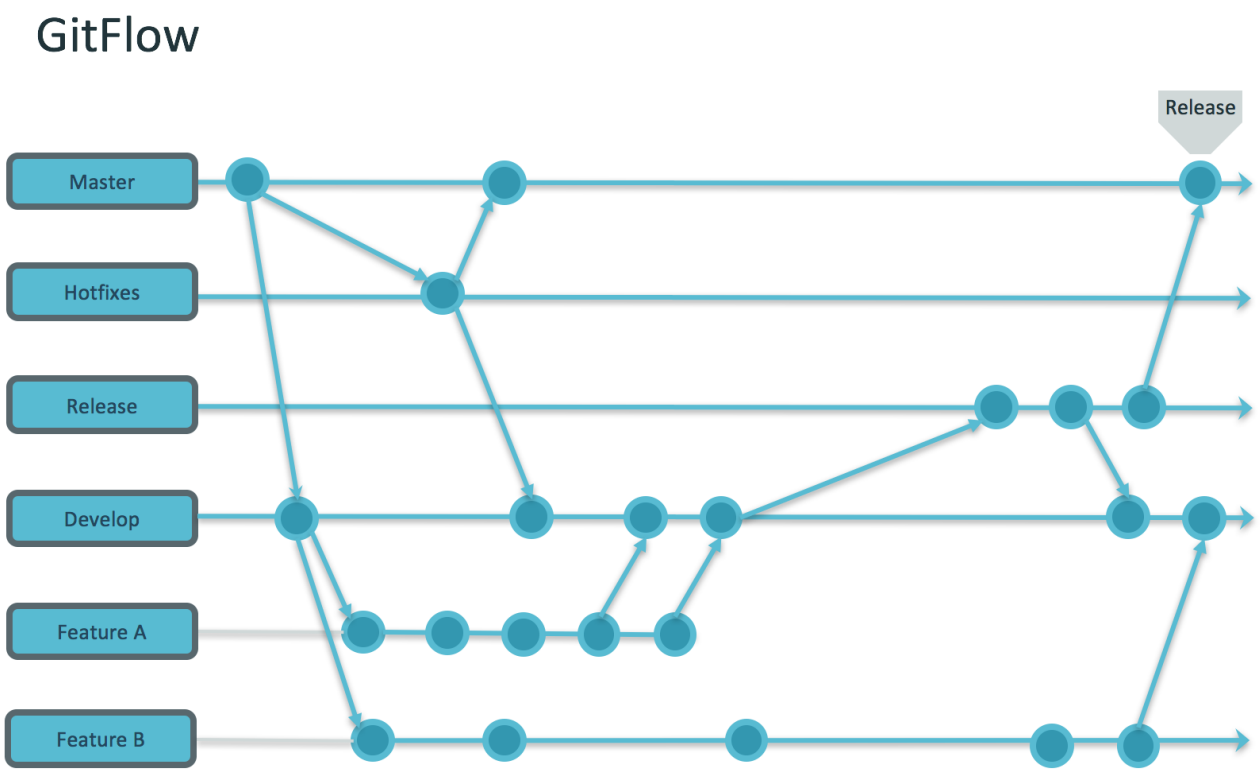
Branch-per-Feature flow



Best Practices

- Branches should be labeled with descriptive names to make it clear which feature, story, or fix the branch was created for. Again, the important requirement here is that the master branch needs to stay stable so that it could be deployed for production at any time.
- Pull request reviews are recommended before any merge from the feature branch to the master happens.
- In addition, to embrace continuous integration practices, the event merge, or eventually the commit, should be followed by automation routines such as unit, integration, or smoke tests.
- To follow one of the key principles of continuous integration, the feature and fix branches should be short-lived so that the integration could run frequently (at least on a daily or nightly basis).

GitFlow



This GitFlow model takes special care to protect the stability of the master branch. This model includes the following branches:

- *Master* -- The master is used only for the final GA version of the code. During development, developers working on features do not access the master.
- *Develop* -- Instead, a develop branch is the branch used for integration of feature branches. For the development team, the develop branch is the master. From the continuous integration perspective, the develop branch (in addition to the master) should be the branch that is covered by continuous integration practices.
- *Feature branches* -- For the new features, special feature branches are created. The workflow there is the same as the Feature flow, with the difference that the develop branch acts as the master. For each new feature, a feature branch is created and the develop branch acts as the master.
- *Hotfixes* -- If a fix is required for a previous release, the hotfixes branch is used. The fix is developed here (or integrated from a fix branch). Then it is merged with the master branch. This makes the fix part of the official production release in case production needs to be updated immediately. Also, the fix is merged to the develop branch, so that the currently developed release contains the fix as well.

- *Release* -- Just before the release, the release branch is used for the purposes of final testing, finalizing documentation, and so on. Development of features for future releases can still continue in the develop branch. If any issue is found in final testing in the release branch, the issue is fixed there and the fix is merged with the develop branch. After everything is done and approved in the release branch, the changes are merged to the master branch and the release is now ready to be deployed for production from the master.

In addition, to embrace continuous integration practices, the event merge, or eventually the commit, should be followed by automation routines like unit, integration, or smoke testing. Also, the feature and fix branches should be short-lived so the integration can run frequently (at least on a daily or nightly basis).

Together with any merge of branches, a pull request review should be performed to ensure that the code is of good quality and follows best practices. For example, the code change meets particular code standards, quality checks, and the unit test results are successful.

Workflows – Using CA Endevor SCM

CA Endevor SCM fully supports the concepts of agile software development and can be used without any trade-offs in an organization that is following the principles of agile.

Software Lifecycle (Environment Mapping)

In CA Endevor SCM, workflows are known as lifecycles, which consist of a set of Environments that are mapped together to define the flow of code from Environment to Environment. A lifecycle can be understood as a sequence of the change maturity levels.

A lifecycle can be configured in many ways with many Environments. However, creating a lifecycle structure of more than three Environments could lead to a less transparent workflow, unnecessary overhead, and a long path for the code changes to reach production.

For agile purposes, a three-Environment or even a two-Environment lifecycle is recommended. The following three or two Environment lifecycles are flexible enough to fulfill implementation, testing, and release needs.

Three-Environment Lifecycle -- Here is an example of a three-Environment software lifecycle:

Implementation –> Validation –> Production

This lifecycle includes the following Environments:

- Implementation (IMPL) Environment, which is where active development occurs. Development leverages the concept of code branches created for specific feature or fix development. Debugging and unit testing are the other activities happening in the IMPL Environment.
- Validation (VLD) Environment, which is dedicated to certification and validation of the changes. Regression and integration testing is happening here, both leveraging the automation capabilities of CA Endevor SCM.

- Production (PROD) Environment, which might be used for real production from where the application is running. However, as real production is usually running in a remote site, the Package Ship feature is used to push production-ready code to the remote site.

To move groups of related software changes along the map, developers could use Promotion Packages.

Two-Environment Lifecycle — An even simpler lifecycle can be used to support agile development.

Implementation –> Production

The two-Environment lifecycle includes the following Environments:

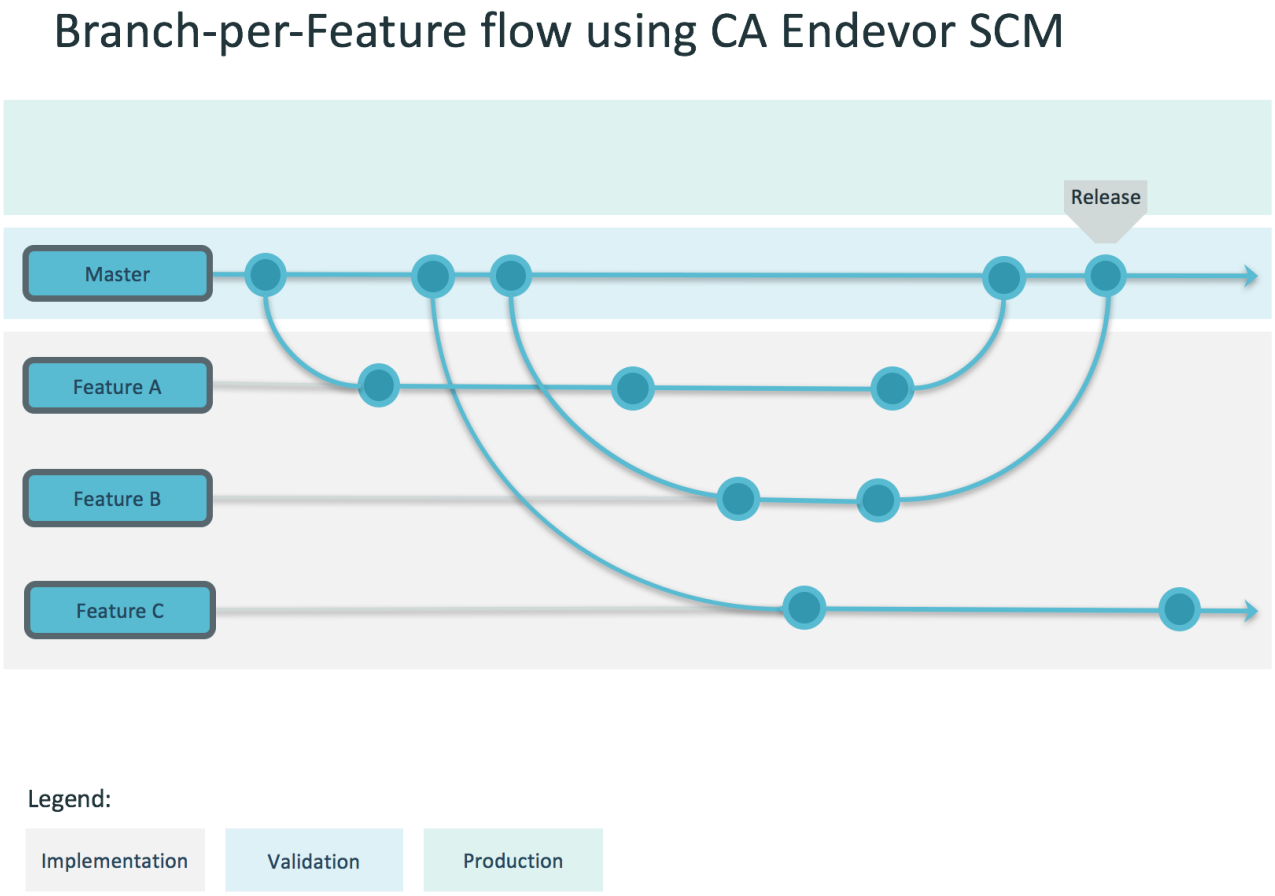
- Implementation (IMPL) Environment, which would include a Stage that plays the role of the Validation Environment. It means that the development as well as the validation procedures are handled in the Implementation Environment.
- Production (PROD) Environment, which receives the output from the Implementation Environment and contains the production ready application.

Software Lifecycle Configuration Options

CA Endevor SCM does not limit the organization from following current standards for workflows. It is important to realize that there are two perspectives.

- The first perspective is that CA Endevor SCM is highly customizable. This allows a wide variety of workflows by leveraging branches (Subsystems), the Parallel Development option and Promotional Packages.
- The second perspective is about the mindset behind the different workflows. It is not only about mimicking the best practices, but about applying the concepts behind the best practices.

Following Branch-per-Feature Flow Using CA Endevor SCM



The Branch-per-Feature flow approach can be smoothly implemented in CA Endevor SCM.

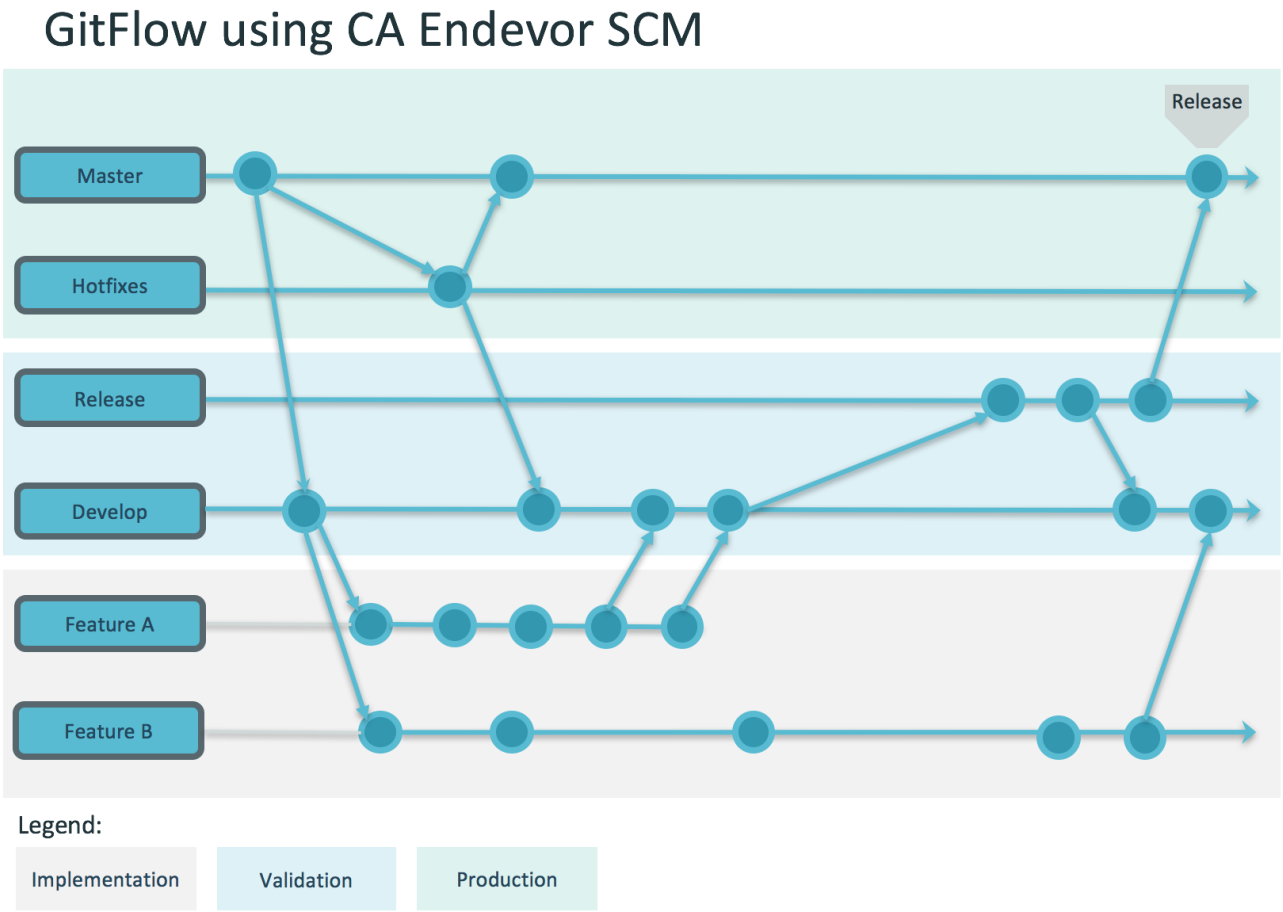
To illustrate the flow activities, the graphic includes the addition of the horizontal background rectangles representing CA Endevor SCM Environments.

For any new feature or fix, a new branch is created in the *Implementation Environment*. Once the change is completed and unit tested, a Package is created and promoted to the Validation Environment where the automated testing procedures should be leveraged. Again, the important requirement here is that the Validation Environment needs to stay stable so that it could be deployed to production any time.

Package approval reviews are recommended before any promotion from the feature branch to the *Validation Environment* happens.

Alternatively, it is possible to use the lifecycle with two Environments – *Implementation and Production*. In this case, the *Implementation* Environment will use one Stage where active development happens and a second Stage where testing will take place. The validated code could go directly to the *Production* Environment.

Following GitFlow Flow Using CA Endevor SCM



In like manner to the Feature flow model, the GitFlow model can be smoothly implemented in CA Endevor SCM.

The GitFlow model takes special care to protect the stability of the master branch, and so GitFlow should be implemented using three Environments:

Implementation → Validation → Production

- The Production Environment is used only for the final release of the code.
- Where GitFlow introduces a develop branch whose purpose is an integration of feature branches, the CA Endevor SCM Validation Environment should be used for this purpose.
- For the new features, special feature branches are created in the Implementation Environment. The finalized changes are promoted using packages to Implementation Environment.
- For the final testing just before the release, the GitFlow model defines the release branch. For this purpose, a special stage in the Validation Environment can be created. If any issue is found in final testing in this stage of the Validation Environment, the issue is fixed here. Once the code is tested and ready to be released, the changes are promoted to the Production Environment.

- The hotfixes branch suggested by the GitFlow model can be handled by a dedicated stage in the Production Environment. This stage is where the fixes for the previous release are developed.

Package approval reviews are recommended before any promotion from the feature branch to the *Validation* Environment happens.

Conclusion

Following agile software development practices for mainframe application development enables organizations to deliver software more quickly and of the highest quality. CA Endevor SCM fully supports agile processes and thus helps organizations to leverage the advantages of agile software development.

To learn more, visit ca.com/endevor