

Branching Strategy


- [General Information](#)
- [CI/CD](#)
 - [Process](#)
 - [Pull Requests](#)
 - [Tags](#)
- [In-depth branch information](#)
 - [Feature](#)
 - [Develop](#)
 - [Release](#)
 - [Hotfix](#)
 - [Main](#)


General Information

Using branches for development is widely accepted as the standard approach across all areas of development. Without proper structure, things can quickly become messy and, in the worst-case scenario, jeopardize the quality of the deliverables—especially when multiple developers work on the same project over an extended period.

Implementing a well-defined branching strategy is an excellent way to ensure development security and make sure no work is being overwritten, missing or straight up harmful. At Contica, we have structured our branches as follows: **with a more detailed description for each branch provided below.**

 **Feature:** This branch serves as a container for small additions to a larger project.

 **Release:** Extracted from the develop branch, this represents a development version and typically indicates the end of the current development phase.

 **Hotfix:** Extracted from the main branch, acts as a container for changes needed to be made quickly to production without interfering with ongoing development.

 **Main:** This branch should exclusively hold the latest relevant release.

By adhering to this strategy, releases should only be deployed to the test environment, and the develop branch should only be deployed to the dev environment. Once testing is completed for the relevant release, a pull request should be made into main. Main should solely contain the code intended for deployment to production, and nothing else.

The inspiration for this approach is the well-established [Git Flow](#) branching structure, with slight modifications to fit the deployment of Azure resources to the cloud.

CI/CD

Another way to secure your pipelines and ensure that no harmful code is developed or deployed is by using a well-defined CI/CD workflow. [Read more about CI/CD here](#). By utilizing this branching strategy, we enable a CI/CD workflow where each branch triggers a separate step in the chain. A recommended approach is to initiate a deployment pipeline using branch triggers. This means that when a release is created, a deployment should automatically be made to the test environment.

This chain can also incorporate automated testing, which will restrict a deployment to an environment if all unit tests are not successfully completed.

Disclaimer: Logic Apps Consumption is difficult to test using pipelines. It is possible but very complicated.

Process

Here follows a short step-by-step description of the CI/CD process we aim to maintain.

1. **Continuous Integration (CI):** Developers frequently merge their code changes into a shared repository, usually using version control systems like Git. The CI process is triggered whenever a new code change is pushed to the repository.
2. **Build and Test Automation:** Once triggered, the CI system pulls the latest code from the repository and builds the software. Automated tests, including unit tests and integration tests, are executed to verify that the code changes are working correctly.
3. **Continuous Deployment (or Continuous Delivery):** If the CI process is successful, the code changes are ready for deployment. In Continuous Deployment, the changes are automatically deployed to a production-like environment. In Continuous Delivery, the changes are prepared for deployment but require manual approval before being released.
4. **Release and Deployment:** The deployment process can involve various stages, such as deploying to staging environments for additional testing or performing canary releases where the changes are gradually rolled out to a subset of users.
5. **Iterative Improvement:** The CI/CD process encourages iterative development and continuous improvement. Developers receive feedback, make adjustments, and repeat the process, ensuring that the software remains in a releasable state at all times.

This branching strategy enables controlled deployments to different environments based on the branch being used.

- **Feature branch:** Does not trigger a deployment.
 - **Develop branch:** Triggers a deployment to the Development environment.
 - **Release branch:** Triggers a deployment to the Test environment.
 - **Hotfix branch:** Does not trigger a deployment.
 - **Main branch:** Triggers a deployment to the Production environment.
-

Pull Requests

Utilizing pull requests is an effective approach to implementing the [four-eye principle](#) in code collaboration. By leveraging pull requests as the mechanism for merging code across deployment-triggering branches, we can realize the following advantages:

1. **Code Review:** Enables thorough review and feedback on proposed code changes.
2. **Collaboration and Feedback:** Facilitates collaboration and constructive feedback among team members.
3. **Quality Control:** Maintains code quality by adhering to standards and best practices.
4. **Version Control:** Provides a history of changes for traceability and rollbacks.
5. **Conflict Resolution:** Helps resolve conflicts when multiple team members work on the same codebase.
6. **CI/CD Integration:** Seamlessly integrates with CI/CD pipelines for automated testing.
7. **Project Management:** Enhances transparency, accountability, and coordination in development.

Tags

Tags serve as identifiers that signify specific states in the codebase, comparable to branching out to a new release. Incorporating tags allows for efficient navigation within a large codebase encompassing multiple releases over an extended duration. This streamlines the process of identifying the current production release and the subsequent release slated for deployment, aiding new developers in grasping the project's status.

In our repositories, we adopt a dual-tagging approach:

`in-production`

This tag signifies the release currently deployed in the production environment, aligning precisely with the main branch.

`ready-for-production`

This tag designates the release that has successfully undergone testing and is scheduled to be merged into the main branch for subsequent deployment to the production environment.

In-depth branch information

At the core, the development model is greatly inspired by existing models out there. The central repo holds two main branches with an infinite lifetime:

- `main`
- `develop`

The `main` branch at `origin` should be familiar to every Git user. Parallel to the `main` branch, another branch exists called `develop`.

We consider `origin/main` to be the main branch where the source code of `HEAD` always reflects a *production-ready* state.

We consider `origin/develop` to be the main branch where the source code of `HEAD` always reflects a state with the latest delivered development changes for the next release.

Feature

May branch off from:

`develop`

Must merge back into:

`develop`

Branch naming convention:

`feature/related-release(x.y.z)/identifier/a-descriptive-name`

Between the version and the description is an identifier for the type of working being done on the feature branch.

New - New development.

Bug - *A minor or major change that resolves an issue in an already released integration.*

- Should only implement what is described in the branch name. Prefer to use more but smaller feature branches instead of one large.
- Can be merged directly to "Develop" locally, merge conflicts should be handled locally before pushing to remote repository.

- On successful release to production, all related feature branches should be deleted to maintain a clean branch structure.
-

Develop

Contains code from ongoing development. When no active development is being done, this branch should be an exact match of the latest release branch and also the source for any new feature branches.

No development should be made directly on the develop branch itself, code should only be merged from ongoing or completed feature branches.

Release

May branch off from:

`develop`

Must merge back into:

`develop` and `main`

Branch naming convention:

`release/x.y.z`

Release branches support preparation of a new production release. They allow for last-minute dotting of i's and crossing t's. Furthermore, they allow for minor bug fixes and preparing meta-data for a release (version number, build dates, etc.). By doing all of this work on a release branch, the `develop` branch is cleared to receive features for the next big release.

The key moment to branch off a new release branch from `develop` is when `develop` (almost) reflects the desired state of the new release. At least all features that are targeted for the release-to-be-built must be merged in to `develop` at this point in time. All features targeted at future releases may not—they must wait until after the release branch is branched off.

X - Indicates a major/initial release. Primarily used on releases aimed for production and major logical changes.

Example: *1.0.0 - Initial production release*

Example: *2.0.0 - Customer decided to move from consumption Logic Apps to Logic App Standard*

Y - Indicates a change in logic, however no source or destination system should be affected by a change occurring here.

Example: *1.1.0 - Refactored to use event grid trigger instead of Service Bus polling.*

Example: *2*.1.0 - Moved XSLT Mapping to Logic App Standard Integration Account.**

Z - Minor changes in logic or structure. Can be used on name changes, parameterizations, url changes and other non critical changes.

Example: *1.1.2 - Renamed Compose action and moved email receivers to a parameter.*

Example: *1.2.1 - Rename Logic App Standard workflow*

Hotfix

May branch off from:

`main`

Must merge back into:

`develop` and `main`

Branch naming convention:

`hotfix-*`

Hotfix branches are very much like release branches in that they are also meant to prepare for a new production release, albeit unplanned. They arise from the necessity to act immediately upon an undesired state of a live production version. When a critical bug in a production version must be resolved immediately, a hotfix branch may be branched off from the corresponding tag on the main branch that marks the production version.

The essence is that work of team members (on the `develop` branch) can continue, while another person is preparing a quick production fix.

Main

Should **ONLY** contain the latest production release. No commits should be allowed to the main branch directly. All changes should be made by Pull Requests from tested and reliable release branches.

Preferably policies should be set up to actually prohibit direct work on the main branch along with review policies on pull requests.

Everything that is currently in production should be on the main branch and should be idempotent.
