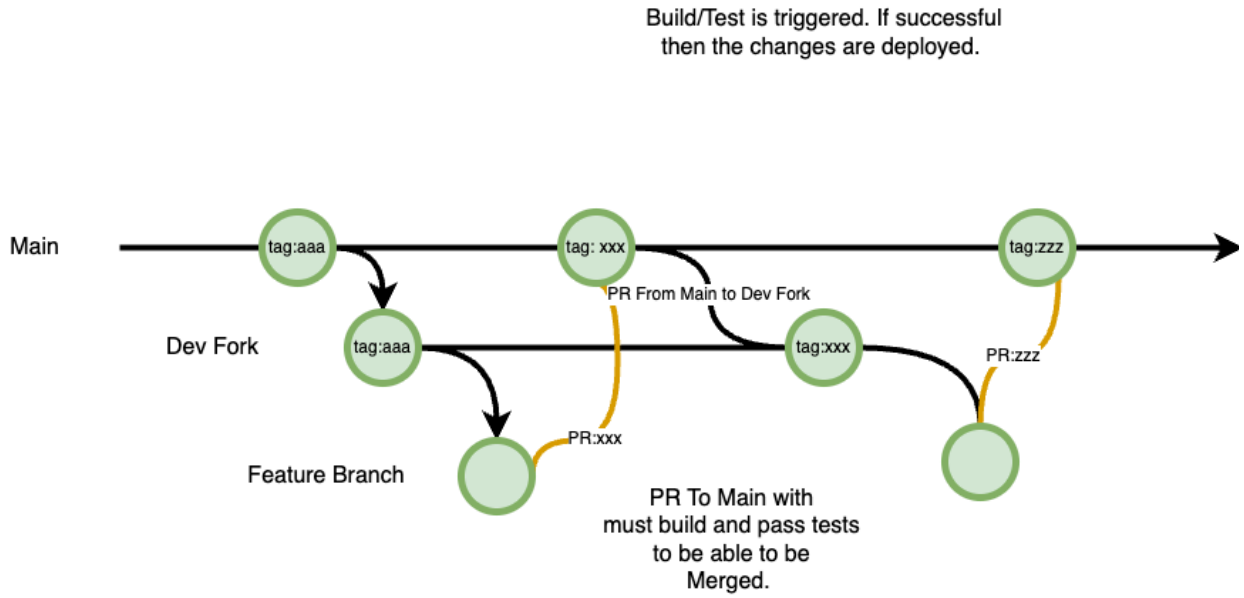
CI/CD



This is a standard type of gitflow that uses trunk/main based development. This is not a recommendation, just a way that CI/CD can be implemented.

The Key to CI CD is defining at what points does the code need to be deployed and interacted with by other actors in the application. This is usually when the code needs to be tested or ultimately deployed.

Using the above as a talking point the following steps may be relevant to the process.

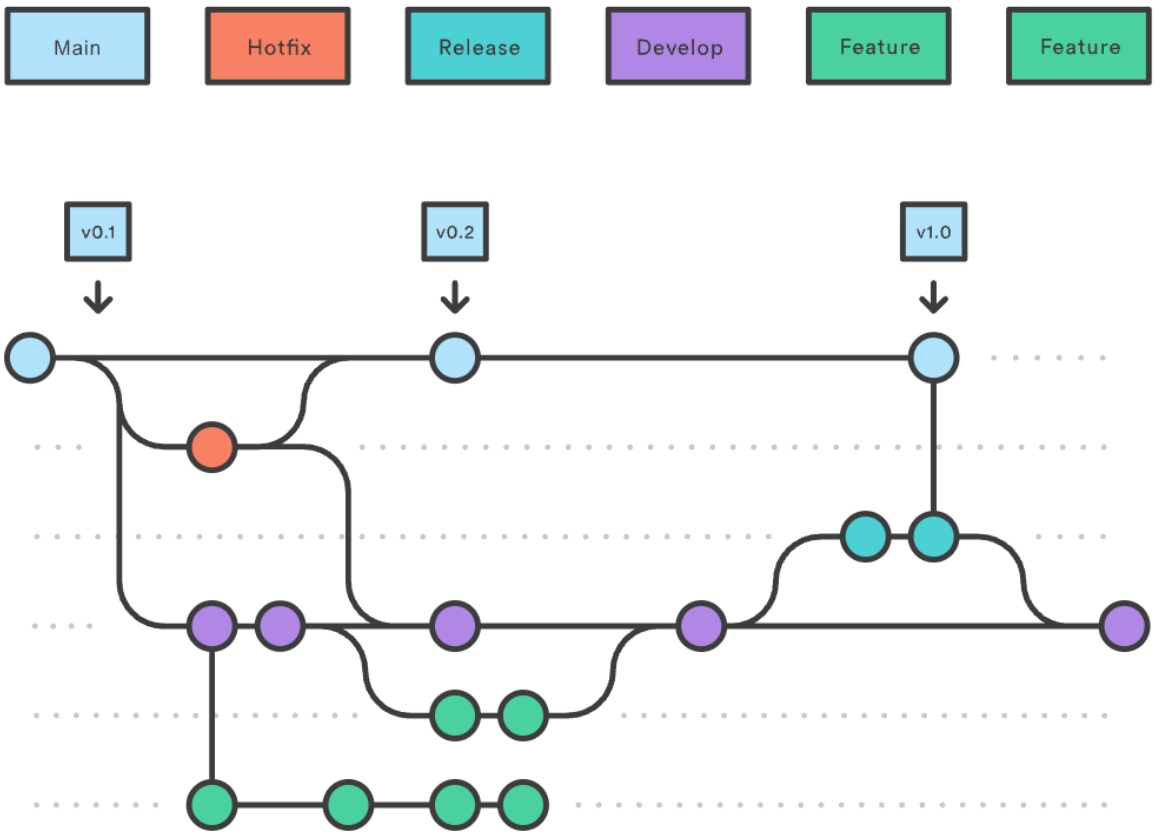
1. Developer Makes a Fork of the Main/Trunk branch
    a. On future development a pr from main to the dev fork is preferred.
2. From their isolated Dev Fork the developer makes a branch.
3. Dev work is done against that branch.
4. When dev work is completed the Developer creates a PR from their Feature Branch to the Main branch
    a. The Devops service(ADO) listens for the PR to be created
    b. The service executes any build/deployment scripts against the test environment
    c. The service then executes any unit tests, integration tests
    d. QA checks off on the feature
5. If all of the above succeed then the pr is approved for merging into main.
6. Once the PR is merged into main, the same processes that were outlined in step 4 are applied again to the production environment.

DevOps

GitFlow-based DevOps best practices

GitFlow branching strategy

Gitflow Workflow defines a strict branching model designed around the project release, and assigns very specific roles to different branches and defines how and when they should interact.



- main branch stores the official release history. Each commit to the main branch is tagged with a version number to facilitate version tracking.
- develop branch functions as an integration branch for features. This branch contains the entire commit history of the project, while the main branch contains an abridged version.
- feature branches are used to encapsulate development of a single feature. In contrast to trunk/main-based development, develop servers as the parent branch and feature branches are merged back to develop. These branches should be short-lived to minimize drift and the

need to integrate changes from other completed feature branches. It is recommended to have a naming convention, eg feature/some-feature or username/some-feature
- release branches are used to prepare a release. Once all features intended to be included in a release are ready, a release branch is forked off develop. After this point no new features should be added to this branch, only bug-fixes, while feature development for the next release can continue on develop. When a release is ready to ship, it is merged to main as well as back to develop, and the release branch is deleted. A naming convention is also recommended, eg release/v1.0.2
- hotfix branches are used to patch releases. They are forked from main, and when complete should be merged back to main and develop. A naming convention could be hotfix/some-bug

GitFlow and trunk-based development are both very popular branching strategies. Each development team should decide on a strategy or combination of strategies that best fits their needs.

For example, some teams prefer to not use a develop branch, and both feature and hotfix branches are created directly off main. This facilitates faster deployment of new features to production, but makes delineating a release more difficult.

Pull Requests and Quality Assurance

Regardless of branching strategy, it is crucial that code is thoroughly tested before deployment, when a pull request is opened. While a developer can run unit tests and integration tests locally or in a non-production Databricks environment, it is also important to test the behaviour of the code as it will run in production - using a Service Principal. This will help surface any permission or configuration issues.

When a PR is opened:

a) The Devops service(AzDO) listens for the PR to be created
b) The service executes any build/deployment scripts against a test environment
c) The service then executes any unit tests, integration tests, test DABs, etc.
d) QA checks off on new features and overall functionality

It is possible to only execute a subsection of the steps when merging to develop, and reserve the full suite for PRs to main. This is sometimes done when testing takes a long time, but the drawback is that some issues may be overlooked until a PR to main is attempted.

When a PR is merged, all the automated steps outlined above should be applied again to the production environment. This applies both to the ingestion framework itself and any projects that utilize it. When ingestion framework code is merged to main, code should be packaged in a wheel, versioned, and uploaded to artifact storage.

Environment isolation

Isolation between testing environments needed for development and PRs becomes important when two or more developers are working in the same codebase.

Ideally, a temporary Databricks workspace should automatically be created when a PR is opened, and all tests executed in this environment. Developers may open a draft PR when first starting development, and run any manual testing in that environment. The workspace should be torn down when the PR is closed. However, it is also necessary to comply with all security requirements when creating the temporary workspaces, and typically the initial automation requires close coordination with IT teams.

Alternatively, prefixes/suffixes can be used to try to isolate simultaneous PRs. With this approach developers need to know each object that is created (files, catalogs, workflows, etc.) and use the prefix/suffix to avoid conflicts. Additionally, these objects need to be destroyed when the PR is closed.

Leveraging DABs

Testing the ingestion framework

Integration tests should run on test data which has been curated to exercise the full functionality of the codebase, and with as many test DABs as needed to test all the variations in DLTs that the framework supports.

DABs template for projects

Bundle templates enable users to create bundles in a consistent, repeatable way, by establishing folder structures, build steps and tasks, tests, and other DevOps infrastructure-as-code (IaC) attributes common across a development environment deployment pipeline.

A custom template can be leveraged to ensure all users of the ingestion framework create consistent DABs. Variables can be defined in a databricks_template_schema.json file, and specified through command line prompts. Alternatively, the variables can be specified in a json configuration file and supplied with --config-file option.

This documentation describes how to create and test a custom template - link. The default-python and mlops-stacks templates can be used as examples.

Important!

A bundle's identity consists of the bundle name, the bundle target, and the workspace. More specifically, the root_path property is what determines a bundle's unique identity, and it defaults to ~/.bundle/${bundle.name}/${bundle.target}

If two bundles are configured with the same root path they will conflict with each other and cause issues that may be difficult to troubleshoot.

Building PySpark packages

PySpark packages can be built the same way as any other Python package. Two common tools are Poetry and setuptools. Both use files to define how the package should be built. Poetry offers a basic usage guide and setuptools have a quickstart page.

Wheels can be stored in Databricks as workspace files or in volumes, or more commonly in artifact storage such as Azure Artifacts.

To install private packages, such as in Azure Artifacts, in Databricks follow this documentation.