Salesforce CI/CD Pipeline - Developer Guide

# Overview

This document provides a step-by-step guide for developers to use our Azure DevOps pipeline to commit, compare, and deploy changes to Salesforce environments using sfdx-git-delta and the Salesforce CLI. The pipeline supports delta deployments, destructive changes, and automated testing.

# Prerequisites

* Access to the repository and the Azure DevOps project:

<https://obs-isd.visualstudio.com/Salesforce%20-%20Internal%20Projects/_git/Salesforce%20-%20Internal%20Projects>

* Salesforce credentials are stored in the Azure DevOps variable group.
* Feature branch should be up to date with develop.
* Release branch must be created from prod branch.

# Pipeline Structure

The pipeline is triggered on:

* Pushes to develop, test, uat, or main.
* Manual execution with a commit hash parameter.

Stages include:

1. Checkout repository
2. Install dependencies
3. Determine commit range
4. Generate delta and destructiveChanges package
5. Run validation
6. Full deployment

# Committing Changes

1. Pull latest changes from develop branch.
2. Make metadata changes locally.
3. Stage and commit only your changes.
4. Push and create PR to target branch.

# Running the Pipeline

* **Automatic**: Triggered on push to deployment branches.
* **Manual**: Trigger via UI with CommitHash parameter.

# Required Variable Groups

Each Salesforce environment must have a variable group in Azure DevOps with the following:

* SF\_CLIENT\_ID
* SF\_CLIENT\_SECRET
* SF\_USERNAME
* SF\_PASSWORD
* SF\_LOGIN\_URL

These groups are:

* obsglobal--sfdev
* obsglobal--sftest
* obsglobal--sfuat
* obsglobal—sfprod
* obsglobal—sfprod-predeployment

# Deployment Logic

* **Delta Package**: Only modified metadata is deployed.
* **Destructive Changes**: Deletes removed metadata.
* **Validation Run**: Includes dry-run and test execution.

# Release Strategy

Our branch structure includes develop, test, uat, and main. The main branch represents the source of truth for production metadata.

# Release Strategy

1. A dedicated release branch is created from main.
2. The uat branch is merged into the release branch, containing only the features that will go live.
3. Commit each go-live feature individually.
4. Run Salesforce – Validate Release pipeline.
5. The release branch is then merged back into main upon successful validation.

# Rollback Strategy

1. Revert the problematic changes from the main branch directly.
2. The pipeline detects the reverted commit and automatically generates the correct delta and destructiveChanges packages.
3. The rollback changes are deployed to production via the regular pipeline.

# Flow Diagram

## A diagram of a company AI-generated content may be incorrect.

# Branch Naming Convention

* **Lowercase and Hyphen-separated**: Use lowercase for branch names and hyphens to separate words, such as *feature/new-login* or *bugfix/header-styling*.
* **Alphanumeric Characters**: Use only alphanumeric characters (a-z, A-Z, 0–9) and hyphens. Avoid punctuation, spaces, underscores, or any non-alphanumeric character.
* **No Continuous Hyphens**: Do not use continuous hyphens. feature--new-login can be confusing and hard to read.
* **No Trailing Hyphens**: Do not end your branch name with a hyphen. For example, feature-new-login- is not a good practice.
* **Descriptive**: The name should be descriptive and concise, ideally reflecting the work done on the branch.

# Branch Prefixes

* **Feature Branches**: These branches are used for developing new features. Use the prefix feature/. For instance, *feature/login-system*.
* **Bugfix Branches**: These branches are used to fix bugs in the code. Use the prefix bugfix/. For example, *bugfix/header-styling*.
* **Hotfix Branches**: These branches are made directly from the production branch to fix critical bugs in the production environment. Use the prefix hotfix/. For instance, *hotfix/critical-security-issue*.
* **Release Branches**: These branches are used to prepare for a new production release. They allow for last-minute dotting of i’s and crossing t’s. Use the prefix release/. For example, *release/v1.0.1*.
* **Documentation Branches**: These branches are used to write, update, or fix documentation eg. the README.md file. Use the prefix docs/. For instance, *docs/api-endpoints*.

# Sample Branch Names

* feature/T-456-user-authentication
* bugfix/T-789-fix-header-styling
* hotfix/T-321-security-patch
* release/v2.0.1
* docs/T-654-update-readme

# Best Practices

* Run local validation with sf project deploy preview. Running the command below with the metadata that was created or modified in the environment, before opening the PR: sf project deploy start --dry-run --manifest manifest/package.xml --test-level RunLocalTests --target-org "[sfdev alias name]"
* Do not commit \*meta.xml unless changed.
* Use meaningful commit messages.
* Coordinate on shared components.
* Commit small and related changes all at once. This allows reverting changes without removing unrelated changes.

# Troubleshooting

|  |  |
| --- | --- |
| **ISSUE** | **SOLUTION** |
| Authentication failure | Check secrets and variable setup |
| Invalid metadata errors | Clean package.xml and retry |
| Variable is empty | Ensure env block passes it |

# Creating a Release

It's possible to simplify the manual process of merging ready-made resources from uat branch into a release branch using VS Code, but this still involves Git operations. However, integrating VS Code with Git Graph can help the process.

# Prerequisites

* **VS Code Installed:** Ensure you have VS Code installed.
* **Git Installed:** Git must be installed on your system and accessible by VS Code.
* **Project Open:** Open your Azure DevOps repository folder in VS Code.
* **Git Graph Extension:** Install the "Git Graph" extension from the VS Code Marketplace. This extension provides a rich visual representation of your Git history and makes cherry-picking very intuitive.
* **(Optional) GitLens Extension:** GitLens is another incredibly powerful Git extension that enhances VS Code's built-in capabilities. It can also help visualize history and commits.

# Git Graph

* Click the "Git Graph" icon (usually in the status bar or findable via the Command Palette: Ctrl+Shift+P -> "Git Graph: View Git Graph").
* This will open a visual log of your repository. You can see your main, uat, and new release branches.
* Locate the uat branch in the Git Graph.
* Browse its history. Each "feature" likely consists of one or more commits. If features are merged via Pull Requests (PRs), you might look for those merge commits or the sequence of commits associated with that feature.
* Tip: Good commit messages and potentially tagging PR merges make this step much easier.

# Checrry-Pick the Features

* **Crucially, ensure you are currently on your release branch.** You can verify this in the VS Code status bar or at the top of the Git Graph.
* In the Git Graph view, find the first commit of the feature you want on the uat branch.
* **Right-click** on the commit.
* Select **"Cherry Pick..."** from the context menu.
* A confirmation dialog will appear. Click "Yes, cherry pick".
* **To select multiple commits:** You can often hold Ctrl (or Cmd on Mac) while clicking multiple commits in Git Graph, then right-click and choose "Cherry Pick...". It's often safer to cherry-pick one feature (or even one commit) at a time, especially if you anticipate conflicts.
* Repeat this process for all the "ready features" you want in this release.

# Features

* If a cherry-picked commit conflicts with changes already on your release branch (or previous cherry-picks), VS Code will flag it.
* The conflicting files will appear in the Source Control panel under "Merge Changes".
* Click on a conflicting file to open the **Merge Editor**.
* VS Code's Merge Editor provides a three-way view (Incoming, Current, Result) and allows you to accept incoming changes, current changes, or both, and manually edit the result.
* Once you've resolved the conflicts in a file, click "Accept Merge" in the editor.
* After resolving all conflicts, **stage** the resolved files (click the + icon in the Source Control panel).

## Push the Release Branch

* Once all desired features are successfully cherry-picked and any conflicts are resolved, push your release branch to Azure DevOps: git push
* You can also use the "Publish Branch" or "Push" options in the VS Code Source Control panel or Command Palette.

# Branch Policies

develop:

* Minimum 2 reviewers.
* Allow requestors to approve their own changes.
* When new changes are pushed, it resets all code reviewer votes.
* Optional linked work items
* Require comment resolution
* Optional reviewers: Francis Lopez, Gabriel Santos and Carlos Manrique

test:

* Minimum 2 reviewers.
* Allow requestors to approve their own changes.
* When new changes are pushed, it resets all code reviewer votes.
* Optional linked work items
* Require comment resolution
* Optional reviewers: Francis Lopez, Gabriel Santos and Carlos Manrique

uat:

* Minimum 2 reviewers.
* Allow requestors to approve their own changes.
* When new changes are pushed, it resets all code reviewer votes.
* Optional linked work items
* Require comment resolution
* Required reviewers: Francis Lopez

main:

* Minimum 1 reviewer.
* Allow requestors to approve their own changes.
* When new changes are pushed, it resets all code reviewer votes.
* Optional linked work items
* Require comment resolution

# Pipeline Library Approvals

osglobal--sfprod:

* Approvers: AdminAgents
* Allow approvers to approve their own runs
* Timeout: 30 days

# Pipelines Stages

## Deploy to Non-Prod Orgs

This stage is responsible for building, validating, and preparing the Salesforce deployment for non-production environments (Develop, Test, UAT).

1. Checkout Code:

Retrieves the latest code from the current branch (e.g., develop, test, etc.).

1. Install Salesforce CLI:

Sets up the Salesforce CLI environment for further commands.

1. Determine Commit Hash:

• If a CommitHash is passed as a parameter, it uses it directly.

• If not, it finds the last successful run for the same branch and extracts the commit hash.

1. Install Plugin:

Installs sfdx-git-delta to generate delta packages for selective deployments.

1. Generate Delta:

Compares the commit hash with the current branch to generate the deployment package and destructive changes.

1. Generate Test Command:

Dynamically builds the sf command to run relevant Apex test classes.

1. Login to Salesforce:

Authenticates into the appropriate org using environment-specific credentials.

1. Dry-Run Deployment:

Performs a pre-deployment (dry-run) to validate changes and metadata compatibility.

1. Test Deployment Validation:

Runs all local tests to verify the code's integrity.

1. Deploy to Non-Prod (if not on main):

Executes the actual deployment to the target Salesforce sandbox (DEV, TEST, or UAT).

## Deploy to PROD

This stage is responsible for deploying to the **Salesforce Production Org**. It only runs when the pipeline is triggered from the main branch.

1. Checkout Code:

Fetches the latest code from the main branch.

1. Install Salesforce CLI & Plugin:

Same setup steps as Stage 1.

1. Determine Commit Hash:

Reuses the logic to get the commit hash, just like in the first stage.

1. Generate Delta:

Prepares the production deployment package based on the latest validated changes.

1. Login to Production:

Logs in to the Salesforce Production Org using secure credentials.

1. Deploy to PROD:

Executes the full deployment using the delta package, including running all local tests.

# Contacts

* CI/CD Maintainer:
* DevOps Team:
* Salesforce Admin:

This document is maintained by the Salesforce DevOps team.

Last updated: May 29, 2025