FPAC Release Process for Automated Deployments

This document describes the release process that will be utilized to support the automated CI/CD pipelines.

1. Executive Summary

FPAC is transitioning from a predominantly manual software deployment model to a fully automated CI/CD release process designed to support consistent, traceable, and repeatable deployments of data pipeline software. This modernization introduces structured Git branching rules, enforced pull-request workflows, automated quality and security checks, and controlled promotion of code across four environments—DEV, CERT, STAGE, and PROD.

In the new model, all development work begins on short-lived feature branches tied to Jira tickets. These changes are validated through automated scans and deployed to the DEV environment on every commit. Approved changes are merged into the `dev` branch, which drives deployments to CERT, where integration testing and certification occur. Once a collection of features is ready for release, the `dev` branch is merged into `main`, triggering deployment to STAGE for user acceptance testing. Production releases are promoted exclusively through Git tags applied to the `main` branch, ensuring immutable release artifacts, auditability, and rollback capability.

This approach eliminates ad-hoc deployment practices and introduces governance, transparency, and automation into every stage of the deployment process. The result is a standardized and scalable release mechanism that reduces risk, accelerates delivery cycles, and aligns FPAC's development and operations teams around a common, enforceable process.

2. Automated Release Process Overview

The automated CI/CD pipeline process will implement a set of rules that enable the effective delivery of data pipeline software through the environments. The current process does not have automated CI/CD, thus the rules for code deployment have been ad-hoc agreements. This section will provide an overview of the automation and the rules that enable this automation.

2.1 Tools and Systems Overview

2.1.1 Current State

The data pipeline software is maintained in a series of BitBucket repositories. There are approximately thirty repositories used for maintaining the software used by FPAC. The data pipelines are scheduled and run as batch jobs from Jenkins. There are currently three Jenkins environments, development (DEV), certification (CERT) and production (PROD). The scheduling of the jobs is implemented by manually changing the schedule in Jenkins. The master schedule is maintained as a Confluence document. There is no release process for the changes to the schedule and currently the schedule is not part of source code control.

2.1.2 Future State Code Repository

The data pipeline software will be migrated to a BitBucket mono-repo, which will have a directory for each project. This solution will support sharing common CDK constructs across data pipelines. It will also support sharing common Lambda layers across projects. These layers include both binary third-party builds as well as common FPAC code, shared across pipelines. The goal of this is to provide the CI/CD automation from a single repository, rather than having the automation replicated across many repositories.

2.1.3 Future State Scheduling

The schedule of data pipeline jobs is critical to the success of these data pipelines. There will be an application that will display and maintain the schedule. Changes made in this application will directly impact the scheduling software, whether that is Jenkins or other tools. The goal is that this scheduling application will have the ability to display what is going to run, what is running and what has run. Within this display, the user should be able to obtain the status and any relevant messages. The deployment of this application code, will follow the GIT branching and CI/CD process, defined in this document.

2.2 CI/CD Process Automation

The CI/CD process automation has the following features:

- Deployment is via AWS CDK using an automated deployment pipeline triggered from BitBucket
- Jenkins will be used to deploy software to one of four environments (DEV,CERT,STAGING, PROD)
- The existing CERT Jenkins system will support both CERT and STAGING
- The AWS CDK implementation is idempotent - ensuring that the AWS CDK command can be run as often as required, without causing the deployment pipeline to break
- The promotion of a release to PROD will be a manual step that is prepared using automation, but must be manually triggered to deploy to the PROD environment
- Other steps are prepared via automation and automatically triggered
- Manual triggering of a deployment, can be added to any step, as required

3. GIT Branching and CI/CD Automation

Automated CI/CD pipelines require the implementation of a well-understood branching strategy. Branch naming conventions and protected branch rules enable the automation system to execute deployment logic based on merges, commits, and tags.

3.0.1 CERT and STAGE Environments

During the transition period, the STAGE environment will serve temporarily as CERT for code moving through automated processes. After the transition:

- **CERT** is deployed from the `dev` branch
- **STAGE** is deployed from the `main` branch

CERT represents the certified state of development for the next release, while STAGE provides a pre-production validation area. After a release, STAGE becomes the environment used for hotfix validation, as CERT moves forward with changes for the next version.

3.0.2 `main` Branch Release Overview with Tags

The `main` branch contains production-ready code and acts as the release branch. Releases are deployed using Git tags that represent specific pull requests merged into `main`. Tags serve as immutable checkpoints, enabling rollback and supporting simultaneous development of future releases.

3.0.3.1 Branch Protection

The `main` and `dev` branches are protected. Engineers cannot commit directly to either branch—changes must be introduced through feature branches and merged via pull requests.

3.0.3.2 Branch Protection Exceptions

Administrators may directly commit to protected branches for specific purposes:

- Ensuring `main` and `dev` match immediately after a production release
- Resolving complex merge conflicts across multiple feature branches
- Updating automated pipeline definitions

3.1 `dev` branch process on DEV environment

The `dev` branch will contain the current state of development. It is a series of PRs (pull requests), from work done for features that are defined a Jira tickets. Each feature should be a branch from `dev` with the name of the Jira ticket. The DEV environment will be a series of changes related to commits made on these feature branches, directly correlated to Jira tickets. The CERT environment will have all the approved PRs that have been merged into the `dev` branch. These actions will be automated via the CI/CD pipeline.

- Jira ticket example called `DDAA–2400`
- Create a branch called `DDAA–2400` from the `dev` branch
- Changes made to `DDAA–2400` based on the Jira ticket
- Commit changes to `DDAA–2400`
- Updates are automatically pushed to DEV environment via CI/CD pipeline
- All code quality and security scans must be passed for each commit, otherwise the deployment to the DEV environment will fail
- It is at this point that all automated scans are run and these must be passed for every commit

3.2 `dev` branch process on CERT environment

The process to promote work to the CERT environment. The feature branch will utilize a PR to merge changes to the `dev` branch. The PR process will involve a manual code review, to ensure that the code should be promoted to the CERT environment. Once the PR is approved, the CI/CD process will automatically deploy the updated `dev` branch to the CERT environment.

- Data Engineer or Lead creates a PR for `DDAA–2400` to merge to `dev`
- The PR is reviewed and any changes are resolved
- Any merge conflicts must be corrected
- The PR is approved and feature branch is merged to `dev` branch
- The `dev` branch changes are automatically pushed to the CERT environment once the PR is approved
- Automated scans are not run, to ensure that the PR is successfully merged
- The changes are available for QA testing
- The feature branch `DDAA–2400` is deleted after the successful merge
- Changes based on QA testing should be done on a new feature branch - starting the process back to section 3.1

3.3 `dev` branch to `main` deployed to STAGE environment

Based on criteria external to this process, a PR will be created to merge the current state of `dev` to `main`. The STAGE environment will always contain the most current state of the `main` branch.

- Once the PR from `dev` to `main` is approved, the updated `main` branch is automatically deployed to STAGE
- The CI/CD deployment automation is trigged by the PR from `dev` to `main`

- User testing can cause changes to flow back through the process, so that multiple release candidates can exist

3.4 `main` branch deployed to PROD environment

The process to deploy a release to the PROD environment utilizes a tag. A tag is created on the `main` branch, for the final commit that represents a release candidate. This tag name should follow a common release naming convention. FPAC appears to be using a date-based versioning solution, so the tag should have the convention [PI]YYYY.MM.PROD[-N], where -N is the incremental number of the next tag for this year and month combination, starting with 1. The created tag will automatically start the process to for a deployment to production. However, it is not completely automatic. The creator of the tag will need to login to the production Jekins server to run the job, that is created.

- Tag is created on the `main` branch
- *EXAMPLE: The first release deployed to PROD for 2026 in January would be tagged as PI2026.01.PROD.01*
- The PROD tag creation will trigger automation - a job is created on the production Jenkins server
- The deployment process must be manually completed on the Jenkins server
- Each PROD tag is a release to the PROD environment

4. Summary: Branching Strategy Overview

This documentation describes a **GitFlow-inspired branching model** adapted to support automated CI/CD pipelines and environment-based deployments. While it does not implement the full canonical GitFlow process, it utilizes GitFlow's core branching concepts and enhances them with environment-driven promotion rules and tag-based production releases.

Key characteristics of the identified strategy include:

- **Two long-lived protected branches**
  - `dev` – the integration branch where active development occurs
  - `main` – the production-ready branch representing release candidates and deployed code
- **Short-lived feature branches**
  - Created from `dev`, named after Jira tickets, and merged back through PRs
  - Eliminated after merge, ensuring clean history and reduced branch clutter
- **Tag-driven production releases**
  - Tags on the `main` branch represent immutable release artifacts
  - Tags follow a date-based versioning scheme and initiate production deployment workflows
  - Tags also provide rollback points

Overall, this is best described as:

A modified GitFlow (GitFlow-lite) branching strategy with environment-based deployments and tag-driven production releases.

---

5. Why This Strategy Resembles GitFlow

The structure closely aligns with GitFlow's foundational components:

- `main` ≈ GitFlow's master (production branch)
- `dev` ≈ GitFlow's develop (integration branch)
- Feature branches per task → merged via PRs → deleted after merge
- Releases identified by tags on `main`

These elements establish a recognizable GitFlow model while omitting the full complexity of GitFlow's optional release and hotfix branches.

---

6. Differences From Standard GitFlow

Although inspired by GitFlow, this implementation contains notable customizations:

6.1 No Dedicated Release Branches

Traditional GitFlow uses temporary release/* branches. Here, the merge of `dev` into `main` functions as the release preparation step, and a **tag** denotes the finalized production build.

6.2 Hotfix Behavior Is Implied, Not Formalized

GitFlow defines explicit hotfix/* branches. In this model, hotfixes are tested in STAGE and integrated through standard feature branching rather than a distinct hotfix flow.

6.3 CI/CD Pipelines Tied to Branches and Tags

Unlike standard GitFlow, this process tightly couples:

Branch/Tag	Deployment Target
Feature branch	DEV environment

Branch/Tag	Deployment Target
`dev` branch	CERT environment
`main` branch	STAGE environment
Tag on `main`	PROD environment

This mapping is highly opinionated and optimized for automated promotion.

---

7. CI/CD Behavior in This Strategy

The branching model aligns tightly with CI/CD automation:

1. **Feature Development**
   - Work happens on a Jira-specific branch from `dev`
   - Commits undergo verification scans and deploy automatically to DEV
2. **Certification**
   - Feature branches merge into `dev` through PR review
   - Merging triggers automated deployment of `dev` to CERT
3. **Staging**
   - Approved PRs from `dev` into `main` deploy mainline code to STAGE
4. **Production**
   - Creating a tag on `main` initiates a controlled deployment to PROD
   - Tags act as immutable release artifacts and rollback points

This process enables incremental approvals, gated promotion, and consistent release reproducibility.
