Design for SQL and AVRO Tags pipelines (CI/CD)

Table of Contents

Introduction

The scope of this is to create a workflow for the deployment of source schema and SQL DDLs. Currently, in the on-prem setup (deploying Avro & HQL DDLs), there were few pain points that were mentioned for eg:

- There is no standard process for the development and ways of working
- Developers have to create the tags manually
- The commits/tags are merged to a branch only when the deployment is happening in the production

In this design/workflow, we have tried to mitigate those.

Development Workflow

There are 2 options to develop the SQL DDLs:

Option-1

The developers will:

1. Use the Databricks workspace in DEV
2. Clone their code & create a feature branch
3. Test their changes in that workspace
4. Merge their changes to master

Option-2

The developers will:

1. Login to Orange terminal server
2. Use IDE, clone their code & create a feature branch
3. Test their changes in that workspace
4. Merge their changes to master

Note: This option will require the configuration of Databricks Connect and port openings.

We will go with **Option-1** for now

CI/CD Workflow

Build (CI)

1. Create a CI build with the required stages for the source schema/SQL DDLs files to publish the changes to the latest Artifacts by building a package
2. The CI build is associated with the Tag version
3. The build needs to have built-in code analysis (Optional)
4. Build can be automated/manual run on feature branch before merging the code to master and then master merge is associated with automated CI run.

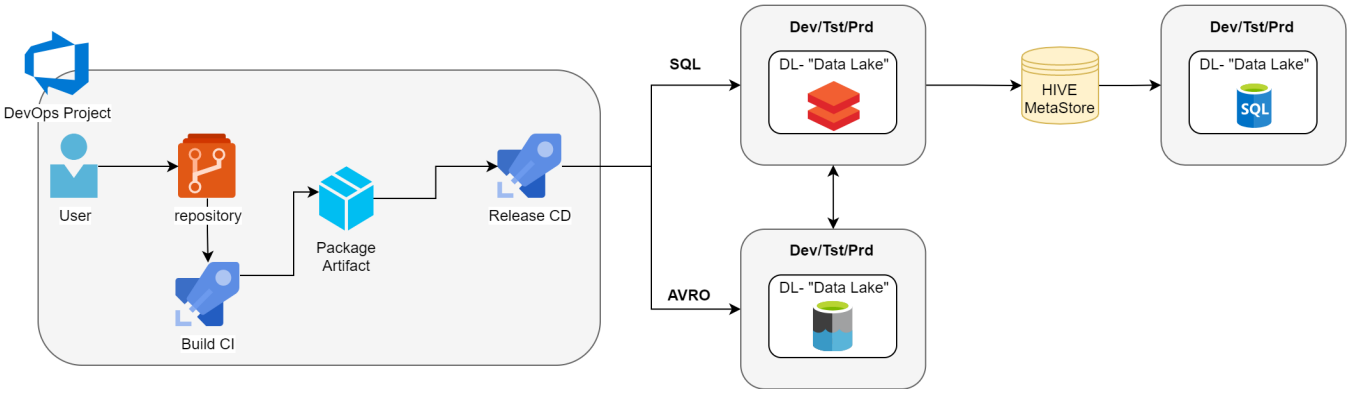
Release (CD)

1. Create a YML/Classic release with the required stages
2. Link the associated build artifact with the release
3. Add the required tasks in the pipeline

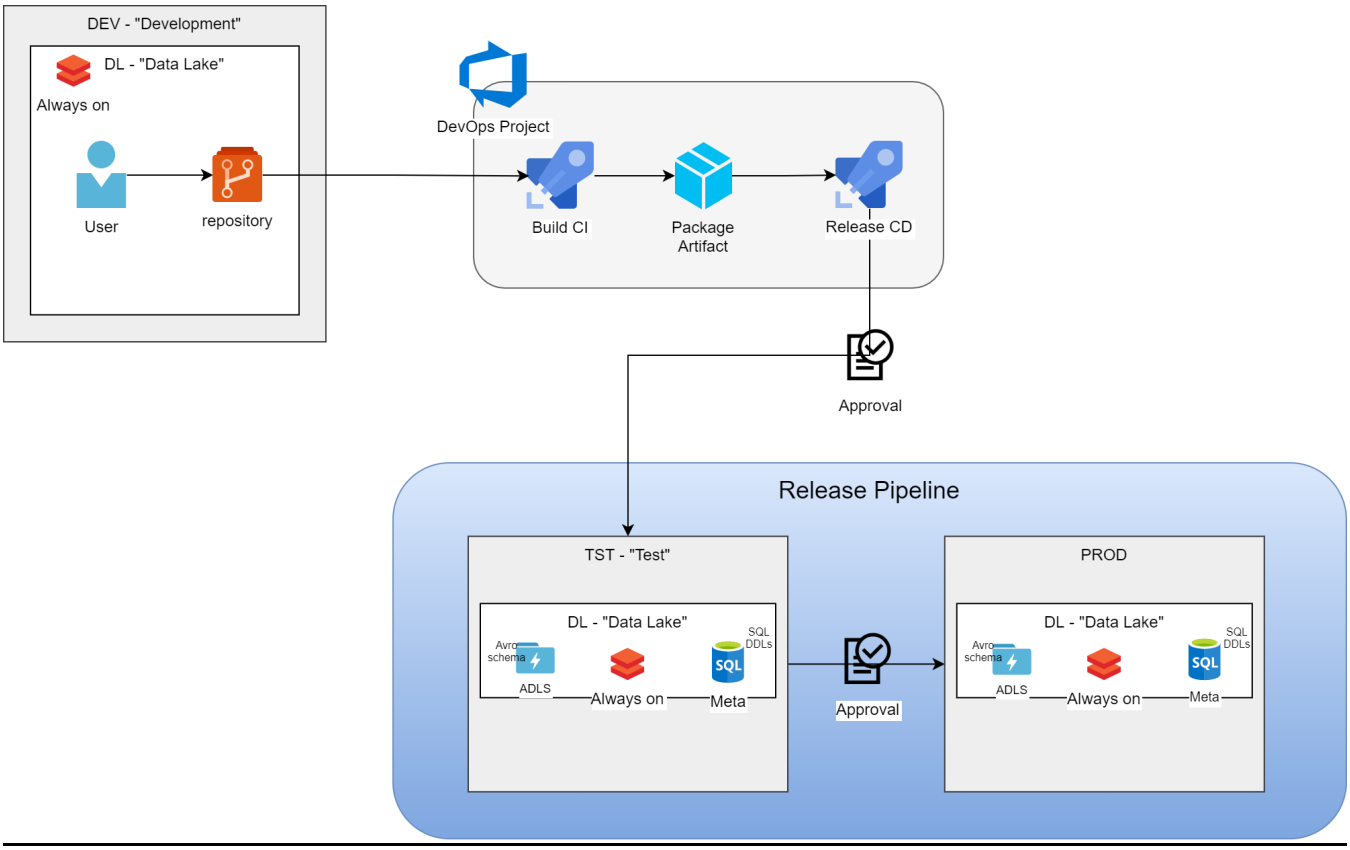
Repo Structure

```
sourcing_setup
  |-sql_ddls
  |         |- databases
  |                 |- database-1.sql
  |                 |- database-2.sql
  |         |- tables
  |                 |- table-1.sql
  |                 |- table-2.sql
  |         |- views
  |                 |- view-1.sql
  |                 |- view-2.sql
  |                 |- view-3.sql
  |-source_schema (avro/json)
        |- *.avsc
     |- *.json
```

High-Level workflow



Workflow (Environment Mapping)



The data flows through the scenario as follows:

1. A developer creates a feature branch based on the repository master branch which contains ***source schema*** & ***SQL ddls***.
2. A developer changes particular values as source code in a particular structure of ***source schema*** or ***SQL ddls*** and creates a pull request with required approvers to review.
3. An automated build is run which creates a package and tags it, once the pull request is merged.
4. Application code including the changed file is committed to the source code repository in Azure Repos in the master branch
5. The build publishes the artifacts for release tagged with specific tags for changes done in ***source schema*** or ***SQL ddls***. For eg. - If a developer changes values in source schema then a specific tag after the build gets associated with the package such as ***package-1.0-sourceschema*** or ***package-1.0-sqlddls*** depending on the changes.
6. Continuous deployment within Azure Pipelines triggers an automated deployment of application artifacts *with environment-specific configuration values along with specific values* as to which environment the pipeline is referred to, after the test environment deployment there is an approval flow to push to the production environment.
7. The artifacts are deployed to Databricks cluster / ADLS. The databricks cluster is referenced to ADLS for fetching desired data and then pushed to SQL DB using hive.
8. Developers monitor and manage health, performance, and usage information.

Suggestions for Development

This subsection includes suggestions for development, specifically for running notebooks in a given folder in a traceable way at Databricks.

The script written for this task can include the following steps.

1. Check for the notebooks existing in a given directory. See Databricks Workspace API List
2. Check for jobs existing in a given directory. See Databricks Jobs API List
3. Optional: Next step will be to create jobs. If they need to run on already existing clusters, new clusters need to created at this step. This can be done using Databricks Clusters API. Additional checks can be added that instead of creating a new cluster, existing clusters can be modified.
4. If there is no job present for a given notebook (combining results of step 1 and 2), programmatically create a job for that notebook. If the existing job needs to be updated, update that job. See Databricks Jobs API Create. Note that there can be multiple jobs that can have the same name with different IDs. Job ID needs to be used as a comparison point.
5. Run the newly created jobs via an API call.

A timeout / retry policy should be added between steps if that involves creation and starting new clusters as this may take some time.

Alternative to the this suggestion, one time job can be created and run if there is no need to have traceability at Databricks with simplifying the development. The above example first creates jobs and then runs them.

FAQs

Q. Which project will be used for the creation of Azure Repo and Pipeline?

Repos and pipelines will be created in the **AppFramework** project

Q. Will the pipelines create a new cluster?

The cluster is an "always-on" cluster and has hive metastore configurations in place. Pipelines won't create a new cluster.

Q. Git identity?

TBD

Q. Archive type?

tar.gz?

Q. ADLS update - update already existing files or create new ones?

The files will not be update, if update - there will be new files created.

Q. Service principle? Databricks token?

1.Service Connection tst: tst-dl-abinitio-serviceconnection

Secret Id found stored in keyvault: tst-dl-codl-kv, secretName: tst-dl-abinitio-sp

Tentant Id found stored in keyvault: tst-dl-codl-kv, secretName: tenantId

2. Service Connection prd: prd-dl-abinitio-serviceconnection

Secret Id found stored in keyvault: prd-dl-codl-kv, secretName: prd-dl-abinito-sp

Tentant Id found stored in keyvault: prd-dl-codl-kv, secretName: tenantId

3. Databricks config token tst stored in keyvault: tst-dl-codl-kv , secretName: DatabricksTestAbinitio-ConfigToken

4. Databricks config token prd stored in keyvault: prd-dl-codl-kv , secretName: DatabricksProdAbinitio-ConfigToken