Sure! Below is the complete **Confluence page documentation**, including all the ideas from our conversation, with a focus on reorganizing notebooks for deployment.

**📘 Reorganizing Notebooks for Deployment**

**Author: [Your Name]**

**Last Updated: {{auto-update}}**

**Tags: Databricks, ADF, Notebook Versioning, Deployment, CI/CD, Best Practices**

**🧭 Overview**

As part of our Databricks notebook deployment strategy, we need to ensure that the process is scalable, maintainable, and avoids unnecessary manual updates across various environments. This document highlights three potential strategies for managing notebook versioning and deployment in a way that reduces the need for constant adjustments in ADF pipelines or interactive workflows.

**Objective:**

* To **minimize manual intervention** in ADF pipelines after deployment.
* To **manage versions** of Databricks notebooks and jobs effectively.
* To **automate notebook versioning** without modifying each individual pipeline.

**📂 Folder Structure for Notebook Versioning**

To keep things organized and ensure the deployment process can scale, we propose splitting notebooks into different folders based on their **execution context**:

main\_repo/

├── notebooks/

│ ├── job\_cluster/ # Notebooks triggered via Databricks jobs

│ ├── interactive\_cluster/ # Notebooks used in ADF or interactive workflows

│ ├── shared\_utils/ # Common logic/utilities reused across notebooks

│ └── routing/ # Launcher logic for routing versioned execution

│

├── library/ # Python modules used across notebooks

├── configs/

│ ├── version\_map.json # Maps notebook names to active versions

│ └── cluster\_map.json # Optional - map notebooks to cluster configs

└── ci-cd/

└── deploy.yml # CI/CD logic for selective deployment

**🚀 Deployment Strategy**

**1. Global Parameters for Versioning (Problem: Frequent Updates)**

**Issue:**

Every time a new version is deployed, the **global parameters** (such as notebook paths or versions) need to be updated manually within ADF. This requires manual intervention across multiple ADF pipelines.

**Potential Solution:**

* Introduce a **dynamic parameter** that reads notebook versions from a file or an external source (like ADLS, SQL, or web activity).
* This way, the ADF pipeline can dynamically read the version for each notebook instead of hardcoding the paths or versions.
* This avoids the need to manually change every single pipeline, but it still requires a way to ensure the pipeline always reads the correct version.

**2. Adding a New Activity to Handle Notebook Path Changes (Problem: Changing Multiple Pipelines)**

**Issue:**

Another approach is adding a **new activity** at the beginning of all interactive cluster calls, which reads the current version from an external source (like ADLS, SQL, or web service). However, this requires a change in multiple pipelines (potentially **16+ pipelines**).

**Potential Solution:**

* Use an **external service** or database (e.g., ADLS, SQL) to store the **notebook version**.
* A **web activity** or **SQL lookup activity** can fetch the current notebook version, which is then passed into the interactive cluster call.

For example:

* 1. Store the version details in **ADLS** or **SQL Database**.
  2. In the ADF pipeline, add a **Lookup Activity** to read the version from the database.
  3. Dynamically update the notebook path to the correct version based on the value fetched from the lookup.

This approach ensures that ADF pipelines remain dynamic and version-agnostic, but the cost is that each interactive pipeline would require a change to incorporate this versioning step.

**3. Dividing Notebooks into Subcategories for Easier Deployment (Preferred Strategy)**

**Folder Organization:**

main\_repo/

├── notebooks/

│ ├── job\_cluster/ # Notebooks triggered via Databricks jobs

│ ├── interactive\_cluster/ # Notebooks used in ADF or interactive workflows

│ ├── shared\_utils/ # Common logic/utilities reused across notebooks

│ └── routing/ # Launcher logic for routing versioned execution

**Solution:**

**Strategy**: Create separate folders for **job clusters** and **interactive clusters**, each with its own versioning system. The interactive notebooks should be deployed based on a versioning approach, whereas job cluster notebooks are only managed in the job clusters. This way, you can ensure that:

1. **Job Cluster Notebooks**:
   * Are deployed using simple job configurations.
   * Versioning can be handled by the folder structure or a dynamic dispatcher.
2. **Interactive Cluster Notebooks**:
   * A launcher notebook can dynamically route calls to the correct version, based on configurations stored in a **version map** (e.g., JSON, database, etc.).
   * A **routing notebook** will read the configuration and use it to determine which notebook version to call.
   * This prevents the need for hardcoded notebook paths, which would otherwise require constant updating in multiple places.

**Example version\_map.json**

This configuration file will hold all the mapping information for different notebook versions and their paths.

{

"weather\_pipeline": "v1.3",

"etl\_runner": "v2.0"

}

**Routing Logic Example: notebook\_launcher.py**

import json

def get\_version(notebook\_name):

with open("/dbfs/configs/version\_map.json") as f:

version\_map = json.load(f)

return version\_map.get(notebook\_name, "default")

notebook\_to\_run = "etl\_runner"

version = get\_version(notebook\_to\_run)

notebook\_path = f"/Workspace/notebooks/job\_cluster/{version}/{notebook\_to\_run}"

dbutils.notebook.run(notebook\_path, timeout\_seconds=3600)

By using this approach:

* The ADF pipeline points to a **static launcher notebook**.
* The **launcher notebook** handles the dynamic path and versioning logic.
* This avoids manual intervention in ADF pipeline paths, and all deployments can be managed in a centralized manner using version configurations.

**🧩 Benefits of the Proposed Approaches**

**1. Versioning via External Sources**

* **Advantages**: ADF pipelines are dynamic and flexible, as they fetch versions from a central source.
* **Challenges**: Requires managing an external source like ADLS or a SQL database.

**2. Using a Lookup Activity**

* **Advantages**: This solution centralizes version management and can be dynamic.
* **Challenges**: It can increase complexity and requires changes across all impacted pipelines (potentially a large number).

**3. Folder Structure Based Versioning**

* **Advantages**: Simplifies the versioning process by keeping everything organized. A single launcher notebook handles dynamic path selection.
* **Challenges**: This is the preferred approach but may require restructuring your repository if it's not already set up this way.

**💡 Future Improvements**

* **Dynamic Versioning Storage**: Use a **centralized versioning service** (e.g., an internal API or cloud service) to fetch and cache versions dynamically.
* **Environment-Specific Versioning**: Extend the version\_map.json or external configuration to handle environment-specific versions, enabling better control across development, staging, and production.

**📅 Conclusion**

This page presents multiple strategies for reorganizing your Databricks notebook deployment to avoid frequent manual updates in ADF pipelines. By leveraging folder-based versioning, dynamic launcher notebooks, and central configuration management, you can streamline your deployment process and make it more efficient for both job and interactive workflows.

Feel free to edit and expand on this documentation based on your project needs!