Testing with Prod Data in Dev

TLDR;

It is highly recommended that you use copies of non-sensitive data sets in dev and testing phases of the SDLC. The reasoning behind this is simple

1. Prod and Dev data NEVER match
    a. This makes testing pretty much impossible
    b. Need to ensure the test data captures all the fidelity and use cases that are represented in the production datasets
2. Prod Data is MUCH Larger than dev
    a. Causes problems with testing
    b. The way you would test and build a job that works on smaller datasets is different than how you would process a large dataset.

These two high level reasons are why there are bugs that show up in production that never were discovered until deployment.

How to address this is fairly simple. Create multiple catalogs, one each for dev qa/test prod. Each of these tiers would allow read access to each of the lower tiers to copy data. This will allow you to snapshot production data and use it for testing and development purposes.

Why?

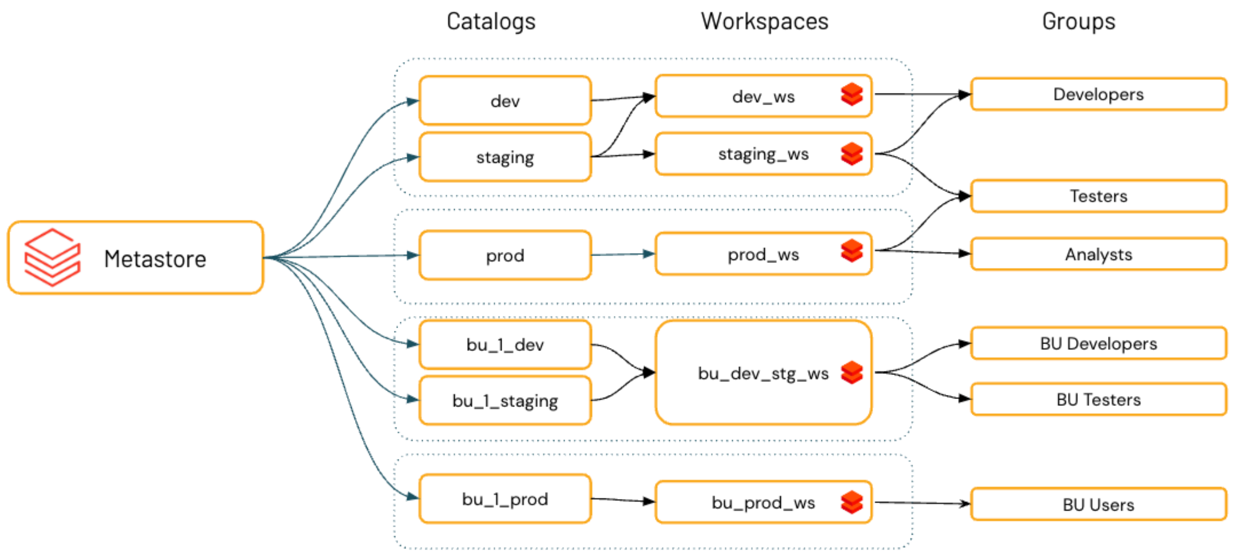
Debugging in production is not good. Not to mention it's unstable and causes even more problems as you try to “fix” stuff in prod without being able to accurately test it in a lower environment. As a general rule of thumb you want to build your systems on data that is representative of the Size, Scope and Variance of the data that is in production. This is the only way to ensure that jobs are functional when they have been delivered from the dev team.

Rough Implementation

Databricks documentation for UC Best Practices on Azure:

https://learn.microsoft.com/en-us/azure/databricks/data-governance/unity-catalog/best-practices

To allow production data to be used in lower environments using Unity Catalog, you can follow these steps:



1. **Create Separate Catalogs for Different Environments**:

    - Create separate catalogs for production and lower environments (e.g., development, testing). This helps in isolating data and managing access control effectively.
    - Example:

```
Unset
databricks catalogs create --json '{

  "name": "prod_catalog",

  "storage_root":
"abfss://<container>@<storage-account>.dfs.core.windows.net/prod",

  "comment": "Production catalog"

}' --profile <profile-name>
```

2. **Create Storage Credentials**:

    - Create storage credentials to manage access to the storage accounts.
    - Example:

```
Unset
databricks storage-credentials create --json '{

  "name": "prod_storage_credential",
  "azure_managed_identity": {

    "access_connector_id": "<access-connector-id>",

    "managed_identity_id": "<managed-identity-id>"

  }

3. }' --profile <profile-name>
```

4. **Set Up External Locations**:

   - Create external locations for each environment to manage where the data is stored.
   - Example:

```
Unset
databricks external-locations create --json '{

  "name": "prod_external_location",

  "url": "abfss://<container>@<storage-account>.dfs.core.windows.net/prod",

  "credential_name": "<credential-name>"

}' --profile <profile-name>
```

5. **Grant Access to Managed Identities**:

   - Ensure that the managed identities have the necessary permissions to access the storage accounts.
   - Example:

```
Unset
az role assignment create --role "Storage Blob Data Contributor" --assignee
<managed-identity-id> --scope <storage-account-resource-id>
```

6. **Create and Manage Schemas**:

   - Create schemas within the catalogs to organize data by use case, project, or team.

- Example:

```
Unset
databricks schemas create --json '{

  "catalog_name": "prod_catalog",

  "name": "prod_schema",

  "comment": "Production schema"

}' --profile <profile-name>
```

7. **Control Access with Dynamic Views**:
    - Use dynamic views to control access to specific columns or rows of data, ensuring sensitive data is masked or filtered appropriately.
    - Example:

```
Unset
CREATE VIEW prod_catalog.prod_schema.prod_table_view AS

SELECT

  id,

  CASE WHEN is_member('data_engineers') THEN email ELSE 'REDACTED' END AS
email,

  country,

  product,

  total

FROM prod_catalog.prod_schema.prod_table;
```

8. **Use Service Principals for Jobs**:
    - Run jobs using service principals to automate processes and reduce the risk of accidental overwrites by users.
    - Example:

```
Unset
databricks jobs create --json '{
  "name": "prod_job",

  "new_cluster": {

    "spark_version": "10.4.x-scala2.12",

    "node_type_id": "Standard_DS3_v2",

    "num_workers": 2

  },

  "libraries": [],

  "spark_jar_task": {

    "main_class_name": "com.example.Main"

  },

  "timeout_seconds": 3600,

  "max_retries": 1

}' --profile <profile-name>
```

9. **Audit and Monitor Access**:

   - Enable audit logs to monitor access and actions performed on the data.
   - Example:

```
Unset
databricks clusters create --json '{

  "cluster_name": "audit_cluster",

  "spark_version": "10.4.x-scala2.12",

  "node_type_id": "Standard_DS3_v2",

  "num_workers": 2,

  "autotermination_minutes": 120,

  "spark_conf": {

    "spark.databricks.dataLineage.enabled": "true"
  }
}' --profile <profile-name>
```

By following these steps, you can effectively manage and use production data in lower environments while ensuring data governance and security using Unity Catalog.