Microsoft | Microsoft Fabric

This presentation is the property of Microsoft and is intended for informational and educational purposes only. You may use, copy, and distribute this presentation for your personal, non-commercial purposes. You may not modify, alter, or create derivative works from this presentation without the prior written consent of Microsoft. You may not use this presentation to misrepresent, defame, or disparage Microsoft or its products, services, or affiliates. You may not use this presentation to endorse or promote any other products, services, or organizations without the prior written consent of Microsoft.

By using this presentation, you agree to abide by these terms. If you do not agree, you must not use this presentation. Microsoft reserves the right to change these terms and conditions at any time without notice. Microsoft disclaims any and all warranties, express or implied, relating to this presentation, including but not limited to the accuracy, completeness, timeliness, or suitability of the information contained herein. Microsoft is not liable for any damages, losses, or liabilities arising from your use of or reliance on this presentation.

Please review the terms of use posted in the content library.

#FABCONSQLCON2026

FABCON
Microsoft Fabric
COMMUNITY CONFERENCE

SQLCON
Microsoft SQL
COMMUNITY CONFERENCE

ATLANTA MARCH 16 - 20, 2026

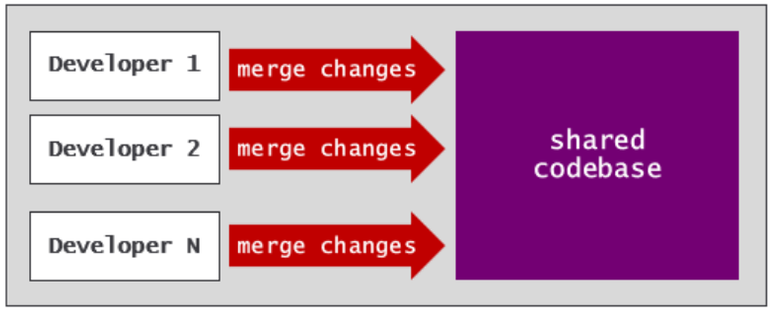
Patterns and Best Practices with Fabric CI/CD

Session Agenda

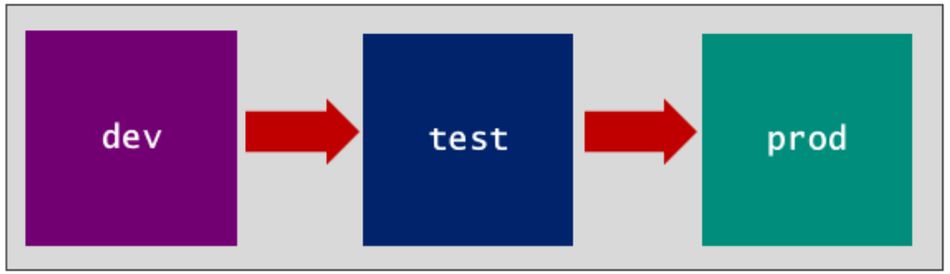
- Essential Concepts and Capabilities
- Planning the Fabric CI/CD Project Lifecycle
- Creating Project Infrastructure using Terraform
- Building a Development Process
- Building a Release Process

Motivation for CI/CD

- What is CI/CD?
  - **Continuous Integration (CI)** is practice of frequently merging code changes to shared repository
  - **Continuous Deployment (CD)** is practice of automating deployment to target environments
- CI provides **development process** where multiple folks work on same codebase

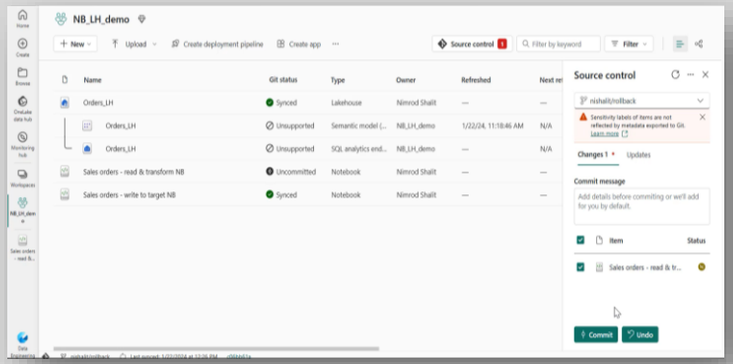


- CD provides **release process** to move changes through a set of stages

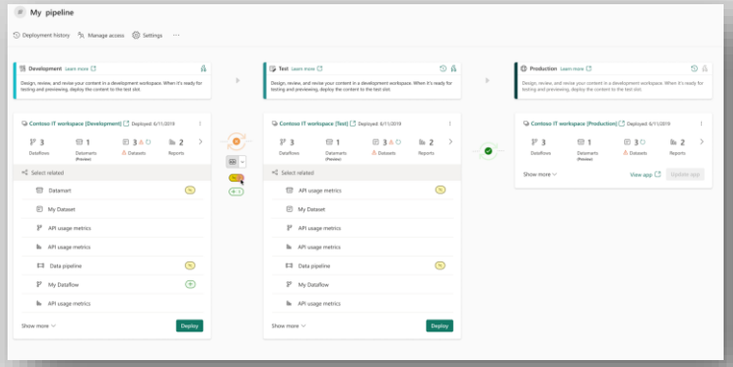


Fabric CI/CD Platform

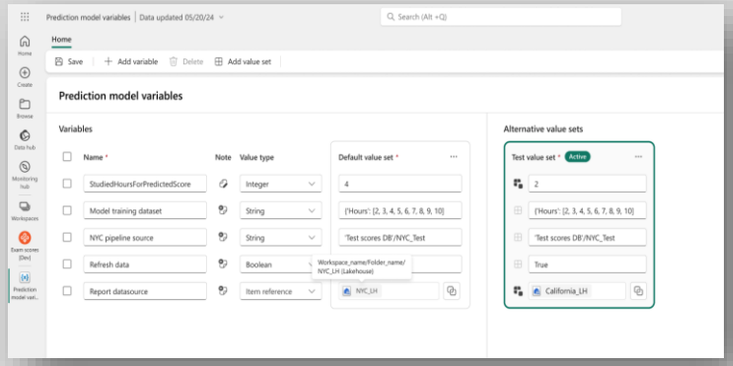
Built-in Git integration



Deployment pipelines



Variable Library



Fabric REST APIs

HTTP Copy

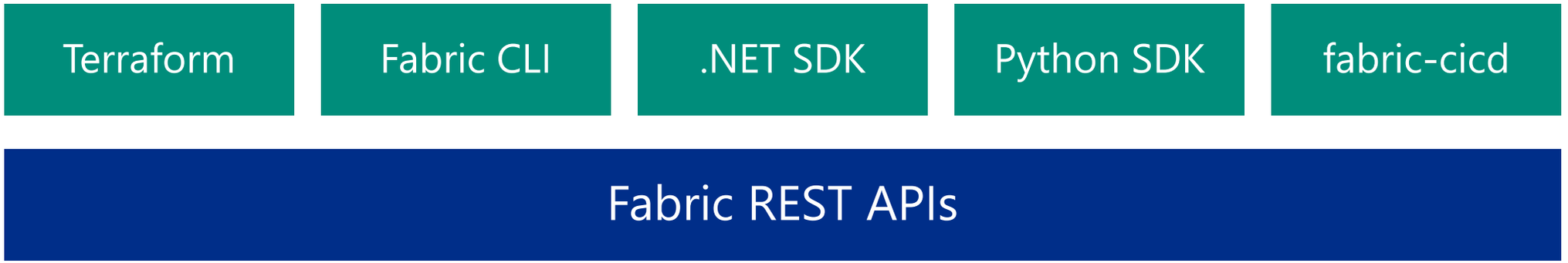
```
POST https://api.fabric.microsoft.com/v1/connections

{
 "connectivityType": "ShareableCloud",
 "displayName": "MyGitHubPAT",
 "connectionDetails": {
  "type": "GitHubSourceControl",
  "creationMethod": "GitHubSourceControl.Contents",
  "parameters": [
   {
    "dataType": "Text",
    "name": "url"
```

Developer Tools and Libraries

```
Creating a new workspace...
* 'ws2.Workspace' created
fab:/$ cd ws2.workspace
* Switched to 'ws2.Workspace'
fab:/ws2.Workspace$ create lh2.lakehouse -P enableSchemas=true
Creating a new Lakehouse...
* 'lh2.Lakehouse' created
fab:/ws2.Workspace$ cp /_ws_cli.workspace/nb1.notebook nb1.notebook -f
Copying '/_ws_cli.Workspace/nb1.Notebook' → '/ws2.Workspace/nb1.Notebook'...
* Copy completed
fab:/ws2.Workspace$ job start nb1.notebook
Starting job (async) for 'nb1.Notebook'...
* Job instance 'e00356cc-0187-4e2a-a148-17995310374b' created
→ To see status run 'job run-status /ws2.Workspace/nb1.Notebook --id e00356cc-0187-4e2a-a148-17995310374b'
fab:/ws2.Workspace$
```

Automation in Fabric

- Tools and libraries for interacting with the Fabric REST APIs
  - **Terraform** - IoC tool used to create project infrastrucutre
  - **Fabric CLI** - interactive and scripting tool with ease-of-use experience
  - **.NET SDK** - productivity layer on Fabric REST APIs for C# developers
  - **Python SDK** - productivity layer on Fabric REST APIs for Python developers
  - **fabric-cicd** - Python library used to build release process



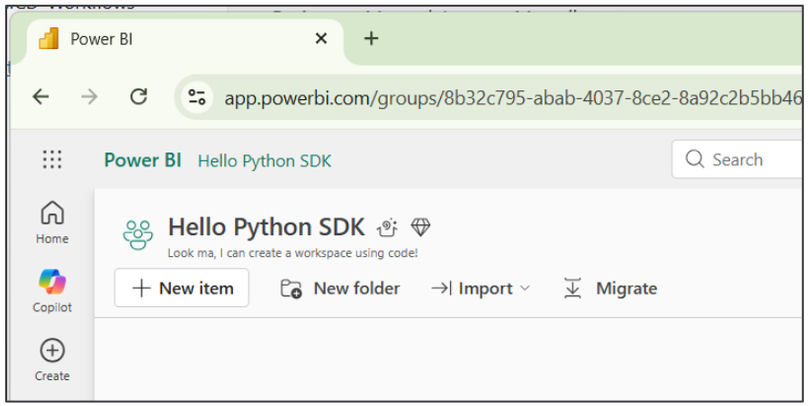
Fabric CI/CD Best Practices Discussed In This Session

- Use service principals for automation
- Use Terraform to provision Fabric CI/CD environment infrastructure
- Use variable libraries to parameterize settings that change across environments
- Abstract logic for ETL jobs into workspace items such as notebooks and pipelines
- Implement continuous integration using feature workspaces and pull requests
- Develop workflows to automate syncing workspace items as changes are merged
- Develop workflows to automate continuous deployment
- Leverage **fabric-cicd** to implement release processes for continuous deployment

Python SDK (new)

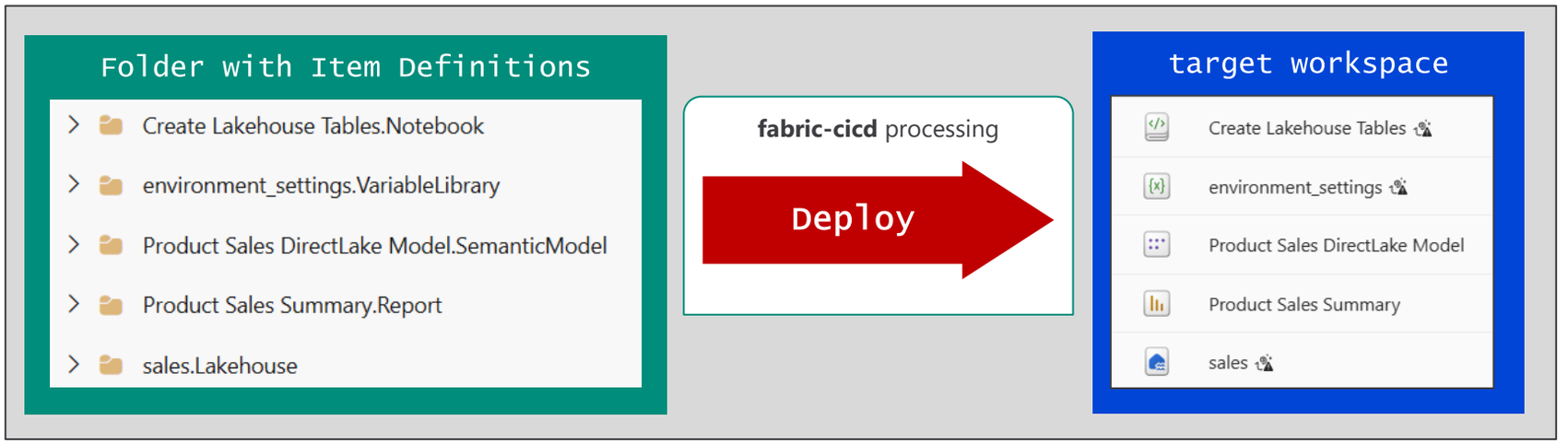
listing01.py U

```python
"""Hello world for Fabric REST API Python SDK"""

import os
from azure.identity import ClientSecretCredential
from microsoft_fabric_api import FabricClient

credential = ClientSecretCredential(
    os.getenv('AZURE_TENANT_ID'),
    os.getenv('AZURE_CLIENT_ID'),
    os.getenv('AZURE_CLIENT_SECRET')
)

fabric_client = FabricClient(credential)

# Call Create Workspace API
create_workspace_request = {
    "display_name": 'Hello Python SDK',
    "description":  'Look ma, I can create a workspace using code!',
    "capacity_id":  os.getenv('FABRIC_CAPACITY_ID')
}

fabric_client.core.workspaces.create_workspace(create_workspace_request)

# Call List Workspace API
workspaces = fabric_client.core.workspaces.list_workspaces()
for workspace in workspaces:
    print(f"{workspace.display_name} - [{workspace.id}]")
```



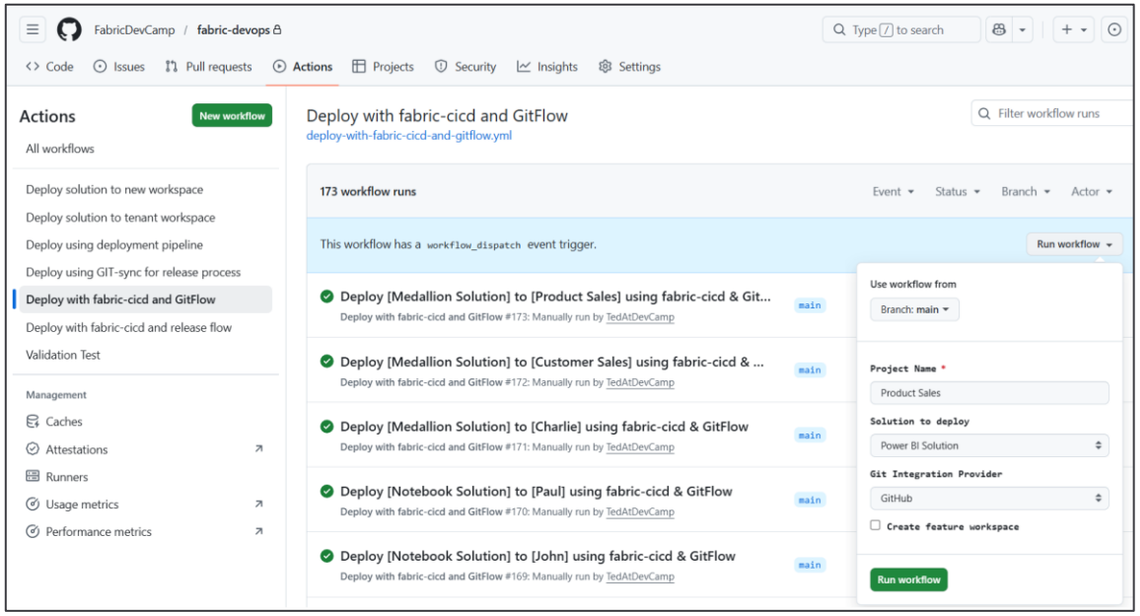
The fabric-cicd Python Library

- **fabric-cicd** library provides code-first deployment strategy for Fabric solutions
  - Provides code for Fabric deployment and building release processes
  - Deploys Fabric solution from folder of item definitions to set of items in target workspace
  - Eliminates developers need to program directly with Fabric REST APIs
  - Recently extended with support for configuration-based deployment
  - An open source project that is also fully supported by the Fabric product team



fabric_devops developer sample

- Sample project as accessible as public GitHub repository
  - Provides sample code for setting up Fabric CI/CD projects with end-to-end lifecycle workflow
  - Automates setting up Fabric CI/CD projects using GitHub workflow actions
  - Leverages fabric-cicd library for deploying Fabric solutions
  - Includes support for creating & configuring repositories in Azure Dev Ops and GitHub
  - GitHub repo at **https://github.com/FabricDevCamp/fabric-devops**



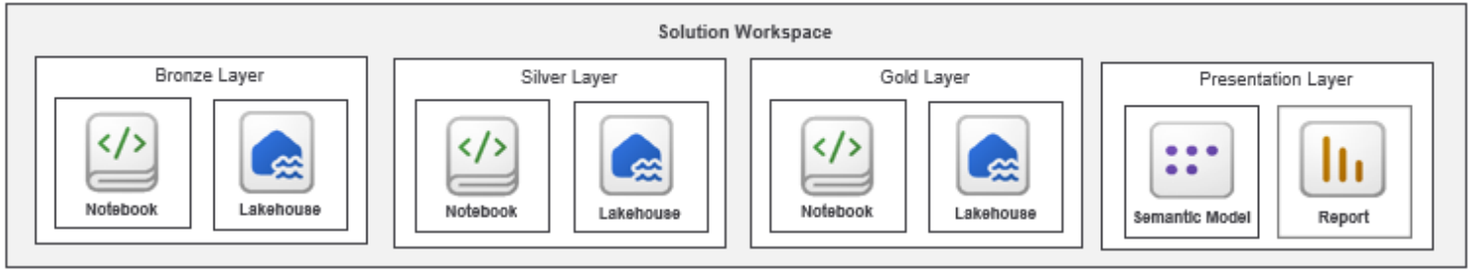
Planning the Fabric CI/CD Project Lifecycle

Planning the Fabric CI/CD Project Lifecycles

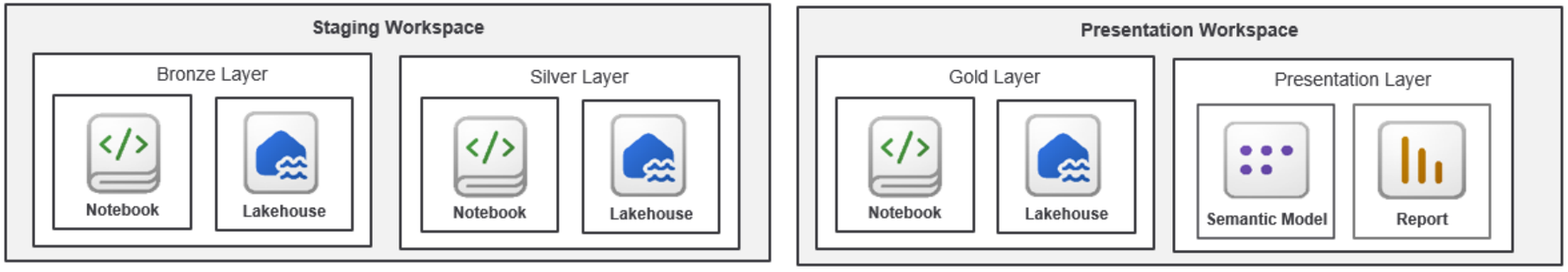
- Steps in the planning process
  - Plan the Fabric solution
  - Plan the project environments
  - Plan the GIT branching strategy
  - Plan the development process (CI)
  - Plan the release process (CD)

Plan the Fabric Solution

- Plan and prototype the Fabric solution
  - Fabric solution is composed on Fabric items
  - Fabric solution usually includes workspace items which run ETL logic
- Fabric solution can be designed to run in a single workspace

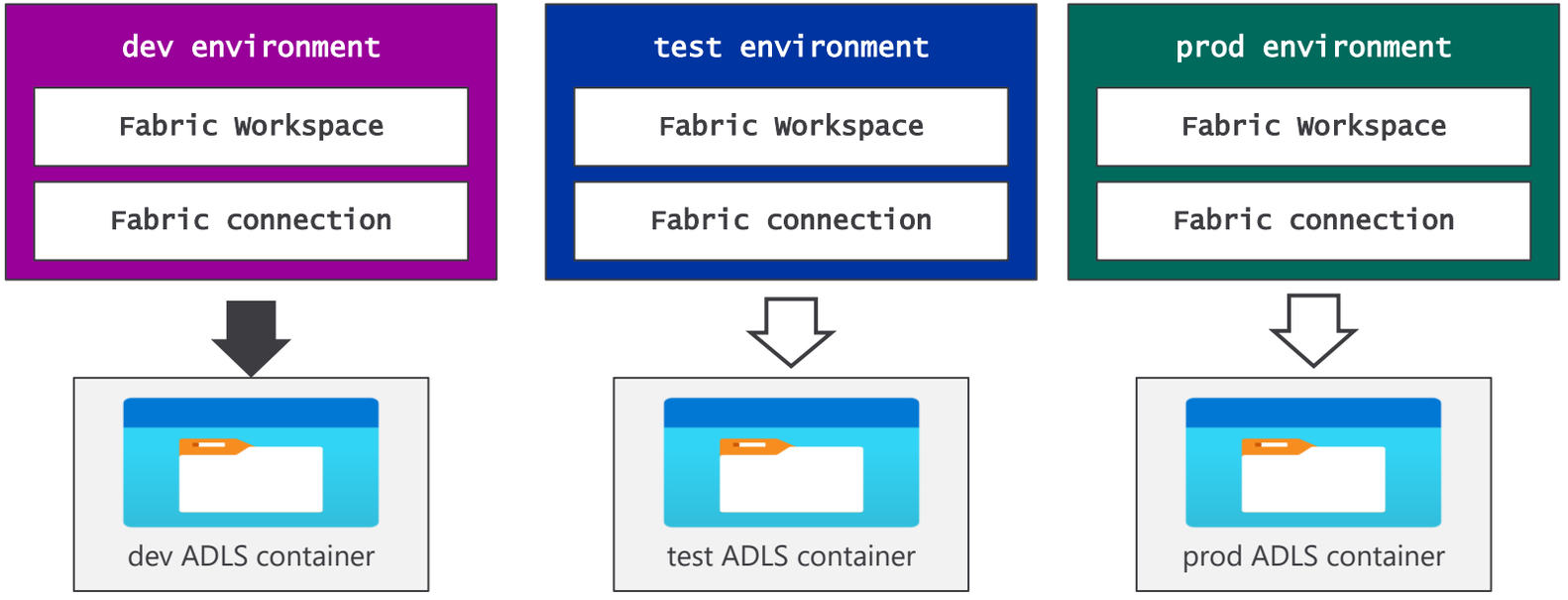


- Fabric solution can be spread across multiple workspaces



Plan the Project Environments

- Fabric CI/CD project requires multiple environments such as **dev**, **test** and **prod**
  - Each environment requires one or more workspaces
  - Each workspace must be associated with a capacity
  - It is common to create a separate capacity for each environment
  - Environments might require their own Azure resources such as a storage account or key vault
  - Determine which environment-specific settings (e.g. datasource URLs) need parameterization

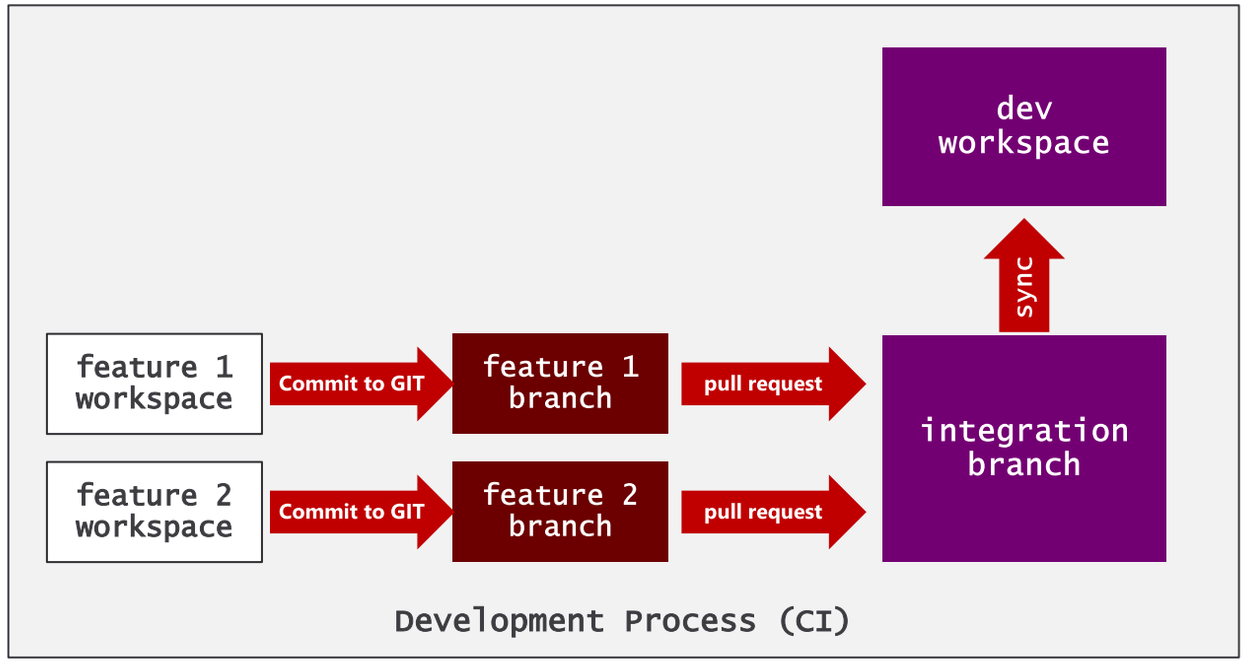


Plan the GIT repository

- Select a GIT provider
  - Choose between **Azure Dev Ops**, **GitHub** and **GitHub Enterprise**
- Adopt a branching strategy to determine what long-lived branches to create
  - When using **GitFlow** branching, plan on three branches named **dev**, **test** and **main**
  - When using **trunk-based development** branching, plan on single branch named **main**
  - Determine what branch policies to use on specific branches
- Plan on who will be developing workflows in GIT provider to control CI/CD lifecycle
  - When using Azure Dev Ops, workflows are developed using **Azure pipelines**
  - When using GitHub, workflows are developed using **GitHub Workflow Actions**

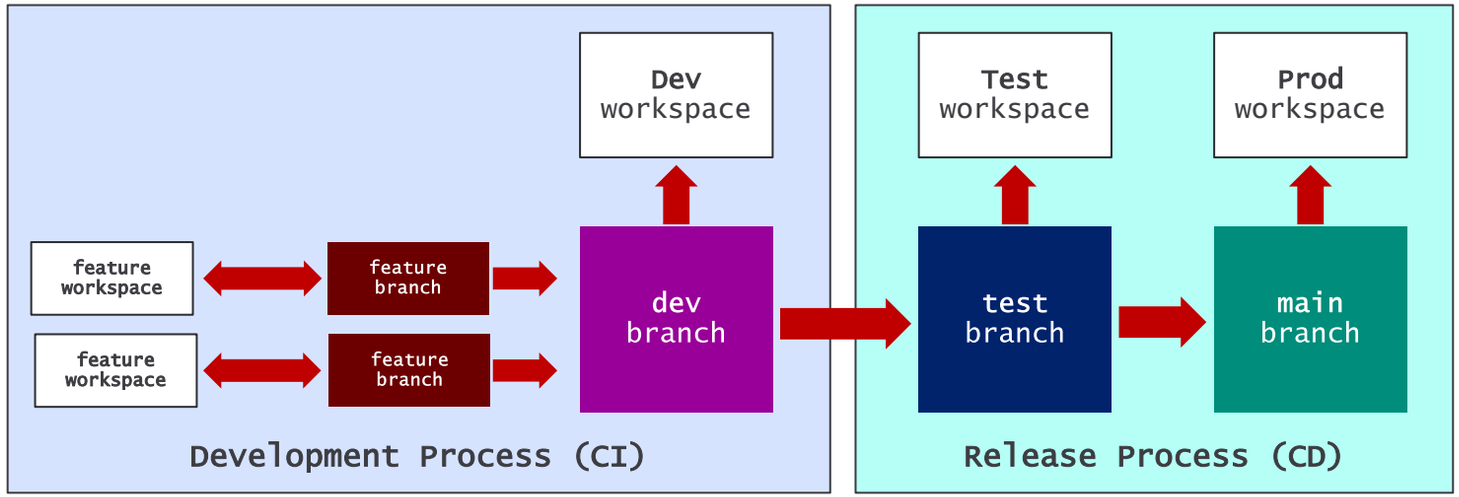
Plan the Development Process for Continuous Integration

- Feature branches and feature workspaces recommended for development process
  - Determine what GIT branch will be the Integration branch (typically it is either **main** or **dev**)
  - Feature branches created from integration branch
  - Feature branch is paired with feature workspace
  - Updates to feature workspace items are committed to GIT
  - Updates committed to feature branch are merged to integration branch using pull requests



Plan the Release Process for Continuous Deployment

- Release process built to deploy updates from integration branch to **test** and **prod**
  - Deploy to **test** workspace for testing and, once approved, deploy to **prod** workspace

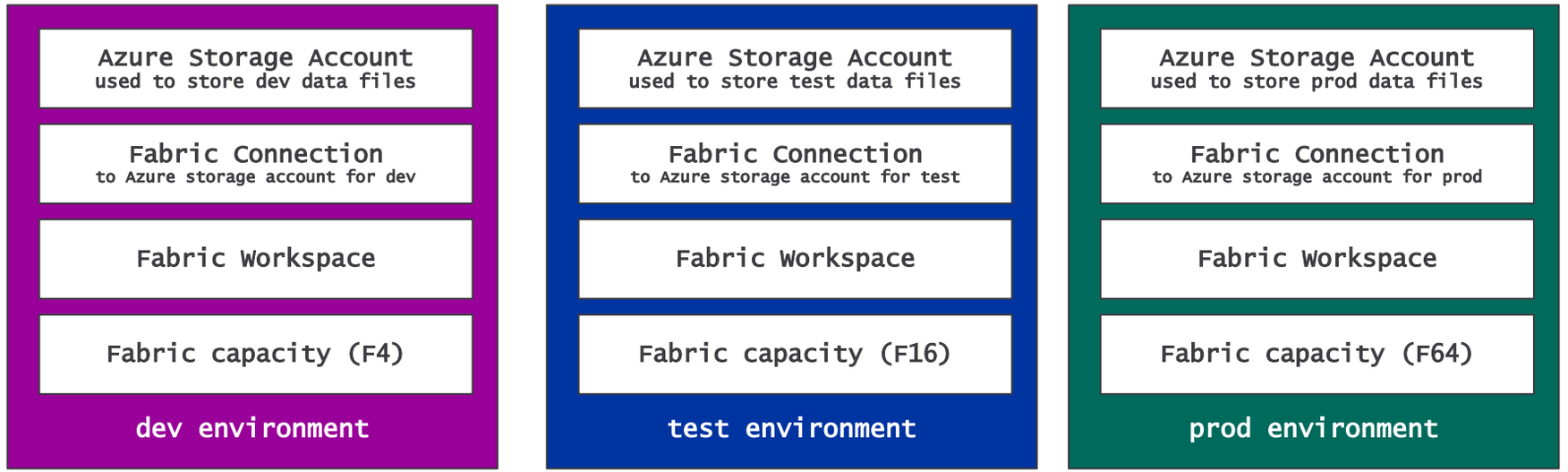


- Choose between several common options for building a release process
  - **Option 1**: Build the release process using **deployment pipelines**
  - **Option 2**: Build the release process using **GIT synchronization**
  - **Option 3**: Build the release process using **fabric-cicd** and **GitFlow** branching
  - **Option 4**: Build the release process using **fabric-cicd** and **release flow** branching

Creating Project Infrastructure using Terraform

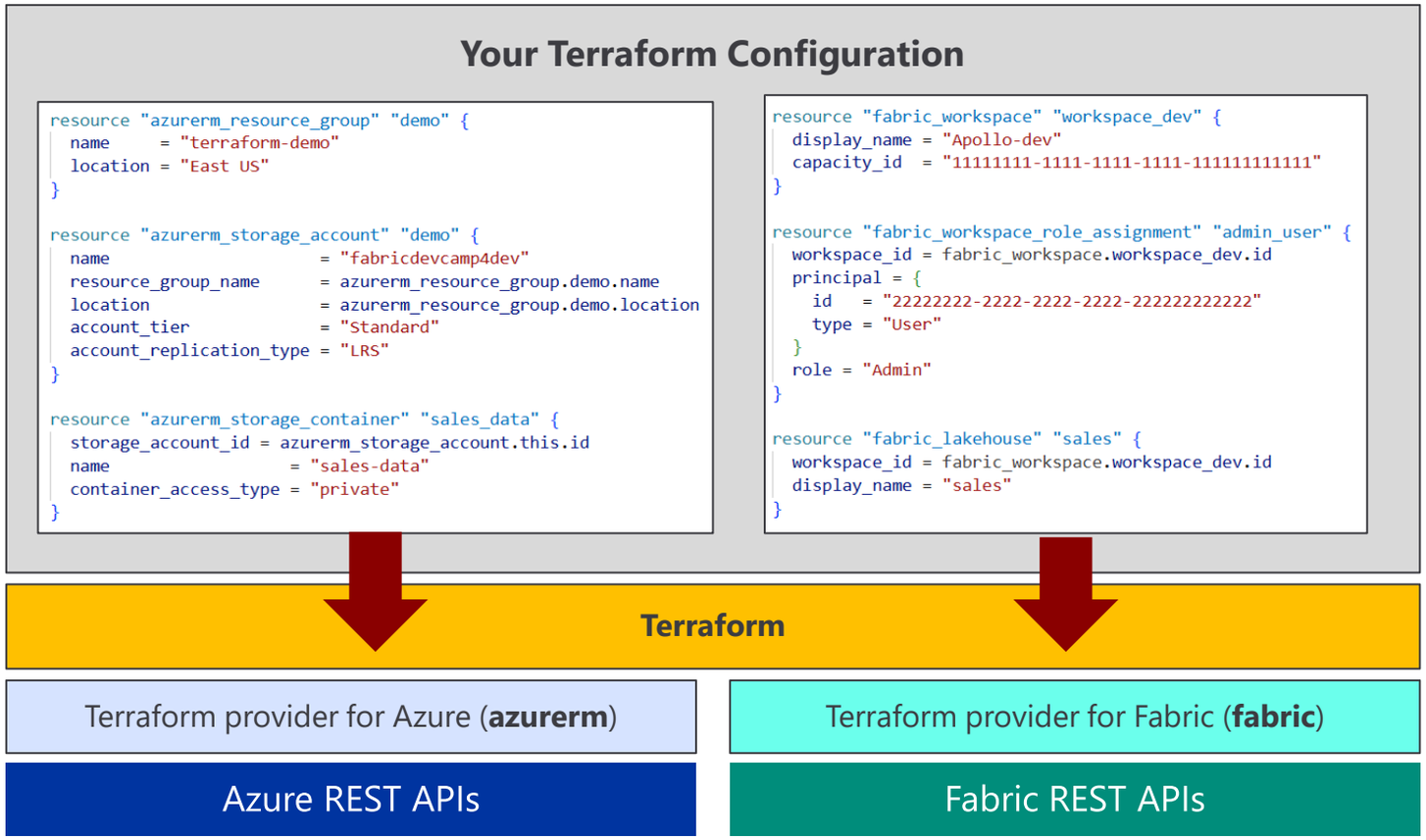
Using Terraform to Create Project Infrastructure

- Terraform is open-source tool built on the principles of **infrastructure as code (IaC)**
  - IaC pattern based on defining the infrastructure for software project using configuration files
  - Provides significant benefits over manual setup processes or procedural programming logic
  - Terraform deployment process is versionable and repeatable
- Use Terraform to create resources needed for Fabric CI/CD project environments
  - Use Terraform to create tenant-level items in Fabric such as **workspace**, **capacities** & **connections**
  - Use Terraform to create Azure resources such as **storage accounts**, **key vaults** & **service principals**



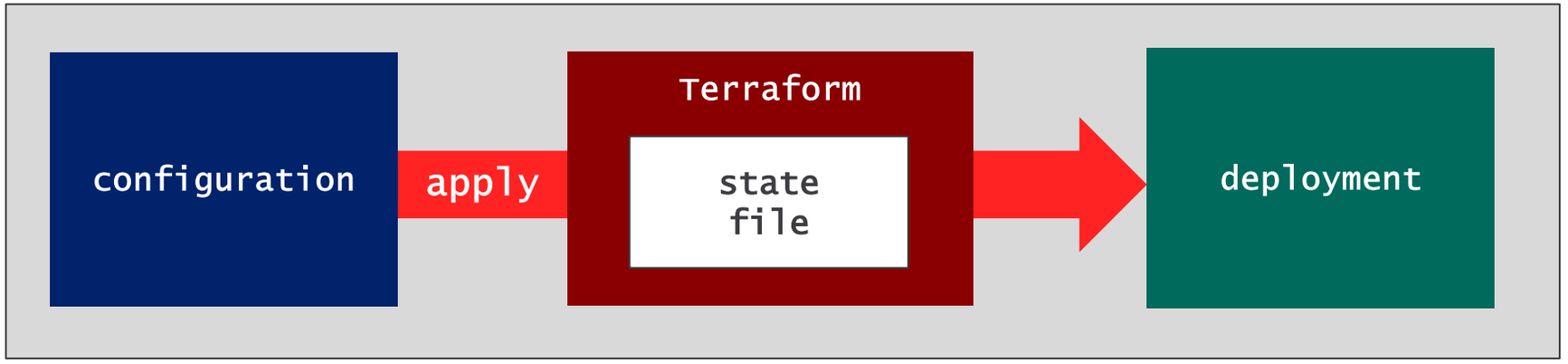
Terraform Providers

- Terraform delegates API calls to Terraform providers
  - **azurerm** provider used to provision and manage Azure resources
  - **fabric** provider used to provision and manage fabric resources

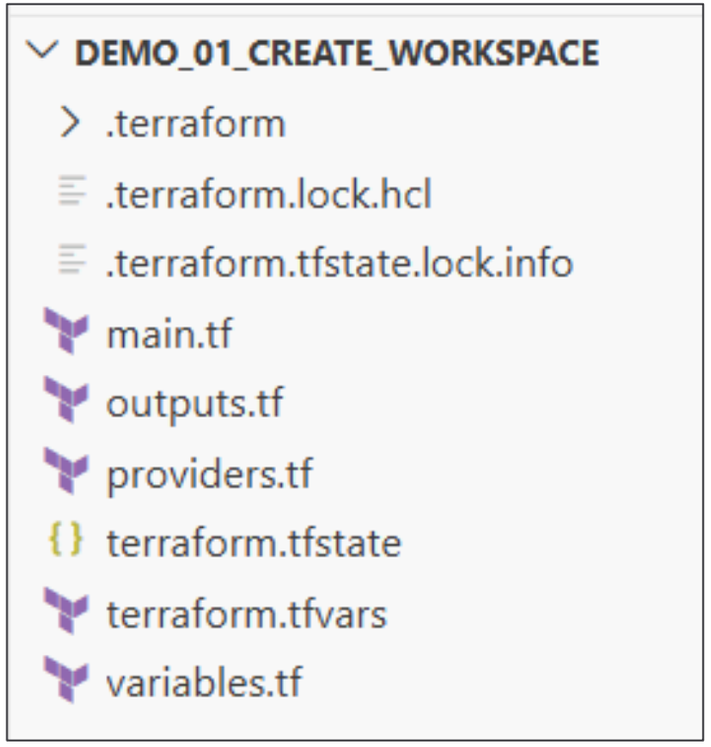


Terraform State File

- Steps for using Terraform in a Fabric CI/CD project?
  - Create a Terraform configuration in in a folder using one or more configuration files
  - Run init command
  - Run **plan** command to review planned lists of the steps required to deploy configuration
  - Run **apply** command to start job to create resources needed to match your configuration
  - Terraform creates state file to track properties for each resource as it's created or updated
  - By tracking resource state, Terraform can compare the configuration against the deployment
  - Terraform can always determine what needs to be created, updated or destroyed



Terraform Configuration Project Files



Common files added to Terraform configurations

- **providers.tf** used to indicate what provider(s) to use
- **variables.tf** used to define variables for coniguration
- **terraform.tfvars** used to assign variable values
- **main.tf** used to provision and manage resources
- **outputs.tf** used to retrieve and display resource properties
- **.terraform** folder created when you run **terraform init** command
- **terraform.tfstate** created when you run **terraform apply** command

Authenticating as a Service Principal

- With Terraform, you can set service principal credentials at provider level
  - Include tenant id, client id and client secret to authenticate using client credential flow
  - Also possible to authenticate as managed identity if required
  - When creating Azure resources, you also need to supply Azure subscription Id

provider.tf

```hcl
terraform {
  required_version = ">= 1.8, < 2.0"
  required_providers {
    azurerm = {
      source = "hashicorp/azurerm"
      version = "4.63.0"
    }
    fabric = {
      source  = "microsoft/fabric"
      version = "1.8.0"
    }
  }
}

provider "azurerm" {
  features {}
  tenant_id     = var.tenant_id
  client_id     = var.client_id
  client_secret = var.client_secret
  subscription_id = var.subscription_id
}

# Configure the Microsoft Fabric Provider
provider "fabric" {
    tenant_id     = var.tenant_id
    client_id     = var.client_id
    client_secret = var.client_secret
    preview = true
}
```

Creating a Fabric Workspace

- Use **fabric_workspace** resource to create new workspace
  - You must provide values for **display_name** and **capacity_id**
  - Add workspace role assignments using **fabric_workspace_role_assignment** resource
  - Also possible to add workspace items such as lakehouse using **fabric_lakehouse** resource

```
resource "fabric_workspace" "workspace_dev" {
  display_name = var.workspace_name_dev
  capacity_id = var.capacity_id
}

resource "fabric_workspace_role_assignment" "admin_user" {
  workspace_id = fabric_workspace.workspace_dev.id
  principal = {
    id   = var.admin_user_id
    type = "User"
  }
  role = "Admin"
}

resource "fabric_workspace_role_assignment" "developers_group" {
  workspace_id = fabric_workspace.workspace_dev.id
  principal = {
    id   = var.developers_group_id
    type = "Group"
  }
  role = "Member"
}

resource "fabric_lakehouse" "sales" {
  display_name = "sales"
  workspace_id = fabric_workspace.workspace_dev.id
}
```

Create a Fabric Capacity

- Fabric capacity created using **azurerm_fabric_capacity** resource
  - You must first create Azure resource group in which to create new capacity
  - You must provide valid capacity values for **name** and **location**
  - Creating new workspace using new capacity requires **fabric_capacity** datasource to retrieve capacity id

```
data "azurerm_client_config" "current" {}

locals {
  spn_object_id = data.azurerm_client_config.current.object_id
}

resource "azurerm_resource_group" "main" {
  name     = var.resource_group_name
  location = var.capacity_location
}

resource "azurerm_fabric_capacity" "main" {
  name                = var.capacity_name
  resource_group_name = azurerm_resource_group.main.name
  location            = azurerm_resource_group.main.location
  administration_members = [
    local.spn_object_id,
    var.admin_user_upn ]
  sku {
    name = var.capacity_sku_size
    tier = "Fabric"
  }
}
```

```
data "fabric_capacity" "main" {
  display_name = var.capacity_name
  depends_on = [ azurerm_fabric_capacity.main ]
}

resource "fabric_workspace" "main" {
  display_name = var.workspace_name
  capacity_id = data.fabric_capacity.main.id
  depends_on = [ data.fabric_capacity.main ]
}

resource "fabric_workspace_role_assignment" "admin_user" {
  workspace_id = fabric_workspace.main.id
  principal = {
    id   = var.admin_user_id
    type = "User"
  }
  role = "Admin"
}
```

Creating an Azure Storage Account

- Create Azure storage account using **azurerm_storage_account** resource
  - You must create Azure resource group in which to create new storage account
  - You must provide valid account values for account tier and replication type
  - This sample also demonstrates adding a container nd generating a SaS token

```
resource "azurerm_resource_group" "main" {
  name     = var.resource_group_name
  location = var.location
}

resource "azurerm_storage_account" "main" {
  name                     = var.storage_account_name
  resource_group_name      = azurerm_resource_group.main.name
  location                 = azurerm_resource_group.main.location
  account_tier             = "Standard"
  account_replication_type = "LRS"
}

resource "azurerm_storage_container" "sales_data" {
  storage_account_id = azurerm_storage_account.main.id
  name                  = "sales-data"
  container_access_type = "private"
}
```

```
data "azurerm_storage_account_blob_container_sas" "sales_data_sas" {
  connection_string = azurerm_storage_account.this.primary_connection_string
  container_name    = azurerm_storage_container.sales_data.name
  https_only        = true

  start  = timestamp()
  expiry = timeadd(timestamp(), "8760h") # 8760 hours = 1 year duration

  permissions {
    read   = true
    list = true
    add    = false
    create = false
    write  = false
    delete = false
  }
}
```

Creating ADLS Gen2 Connections

- Fabric connections can be created using **fabric_connection** resource
  - Connection **display_name** must be unique across all cloud connections within Entra Id tenant
  - **connection_details** includes connector-specific settings for **type**, **creation_method** and **parameters**
  - **credential_details** include **credential_type** and credentials specific to that credential type

```
locals {
  adls_server = azurerm_storage_account.this.primary_dfs_endpoint
  adls_container = data.azurerm_storage_account_blob_container_sas.sales_data_sas.container_name
  adls_target_path = "${local.adls_server}${local.adls_container}"
}

resource "fabric_connection" "alds_sas" {
  display_name      = "ADLS-SAS-[${local.adls_target_path}]"
  connectivity_type = "ShareableCloud"
  privacy_level     = "Organizational"
  connection_details = {
    type            = "AzureDataLakeStorage"
    creation_method = "AzureDataLakeStorage"
    parameters = [
      {
        name  = "server"
        value = local.adls_server
      },
      {
        name  = "path"
        value = local.adls_container
      }
    ]
  }
  credential_details = {
    credential_type       = "SharedAccessSignature"
    skip_test_connection  = false
    shared_access_signature_credentials = {
      token_wo           = data.azurerm_storage_account_blob_container_sas.sales_data_sas.sas
      token_wo_version   = 1
    }
  }
}
```

Managing Connections in Fabric

- Tenant-level scope makes it harder to manage connections than workspace items
  - Some connections are designed to be shared across workspaces
  - Some connections are only intended for use within a single workspace
  - Terraform provides easy way to track connections and manage their lifecycle
- Adopting connection naming conventions assists maintenance, reuse and readability
  - Create display name using pattern such as [**connector**]-[**credential-type**]-[**location**]
  - Adding workspace Id into display name used to indicate connection is only used in one workspace

Name ↑	Connection type
ADLS-SAS-[https://fabricdevcamp.dfs.core.windows.net/sampledata/ProductSales/Dev/]	Azure Data Lake Storage Gen2
ADLS-SAS-[https://fabricdevcamp.dfs.core.windows.net/sampledata/ProductSales/Prod/]	Azure Data Lake Storage Gen2
ADLS-SAS-[https://fabricdevcamp.dfs.core.windows.net/sampledata/ProductSales/Test/]	Azure Data Lake Storage Gen2
GIT-ADO-SPN-[https://dev.azure.com/fabricdevcamp/Product Sales/_git/Product Sales/]	Azure DevOps - Source control
GIT-GitHub-PAT-[https://github.com/fabricdevcampdemos/Customer-Sales]	GitHub - Source control
Workspace[1e4ffa23-ada7-4382-95e9-8f9a269aa2b4]-Lakehouse[sales]-SqlEndpoint	SQL Server

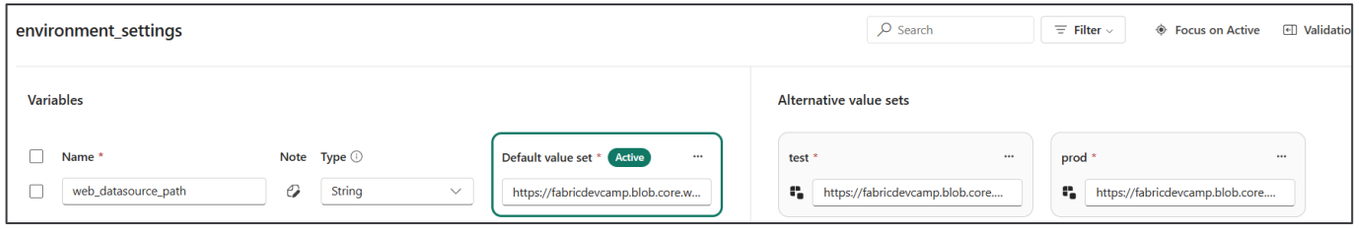
Building a Development Process

Build the Development Process for Continuous Integration

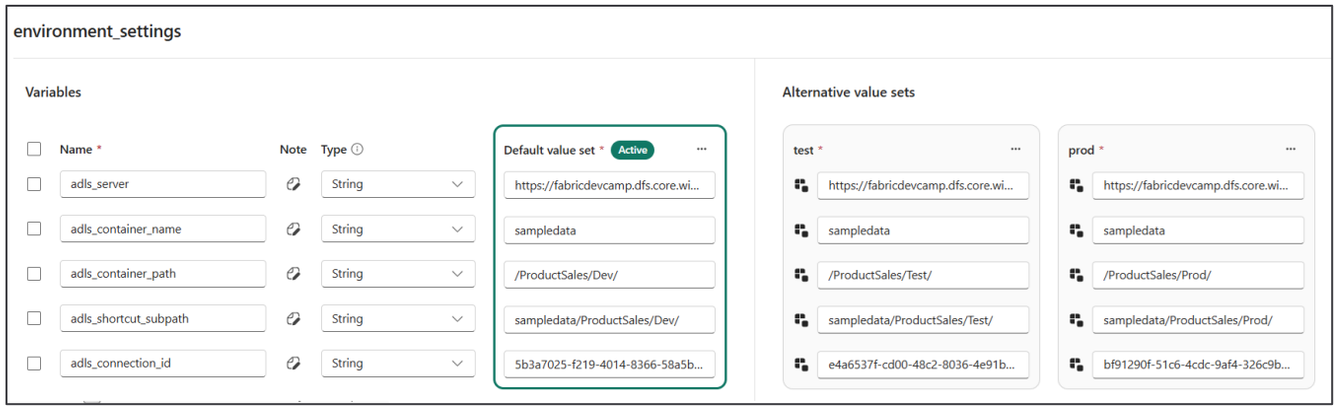
- Steps in building the development process
  1. Get the solution up and running in the **dev** workspace
  2. Create variable library to parameterize environment-specific settings
  3. Create and configure a GIT repository
  4. Connect the **dev** workspace to the GIT integration branch
  5. Create feature branches and feature workspaces
  6. Engage in development to create and update workspace items
  7. Create pull requests to merges changes to the integration branch

Creating a Variable Library for Parameterization

- Example 1 - parameterizing a datasource URL for a notebook
  - Variable library contains single variable name **web_datasource_url**
  - Variable library extended with two additional value sets named **test** and **prod**

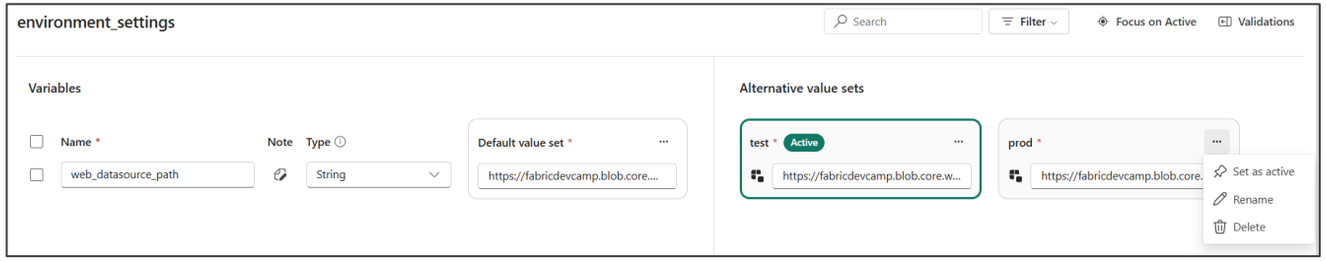


- Example 2 - parameterizing a ADLS Gen2 connection for shortcuts or pipelines
  - Contains variables with datasource path as well as variable for **connection_id**



Using variables in a notebook

- Set active value set to match environment of target workspace

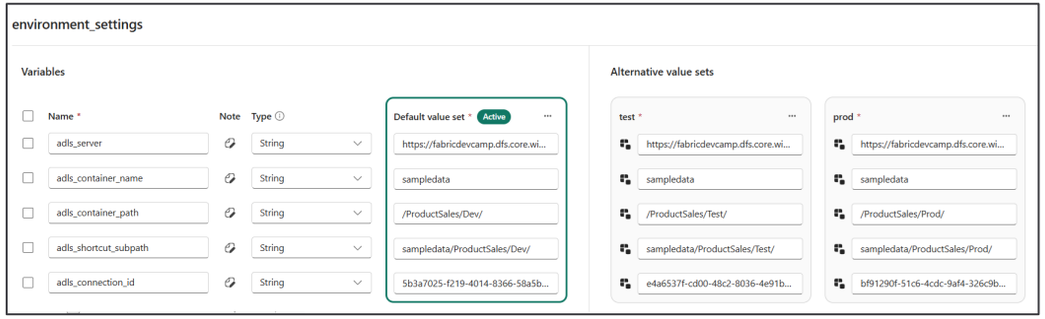


- Notebook uses expression syntax of **$(/**/** {LIBRARY_NAME} **/** {VARIABLE_NAME} **)** to read variable value

```
# copy CSV files to lakehouse to load data into bronze layer
import requests

csv_base_url = notebookutils.variableLibrary.get("$(/**/environment_settings/web_datasource_path)")

print(f'Coping files from {csv_base_url}')

csv_files = { "Customers.csv", "Products.csv", "Invoices.csv", "InvoiceDetails.csv" }

folder_path = "Files/sales-data/"

for csv_file in csv_files:
  csv_file_path = csv_base_url + csv_file
  with requests.get(csv_file_path) as response:
    csv_content = response.content.decode('utf-8-sig')
    mssparkutils.fs.put(folder_path + csv_file, csv_content, True)
    print(" - " + csv_file + " copied to Lakehouse file in OneLake")
```

Using variables in a ADLS Gen2 shortcut

- Using variables to create ADLS Gen2 shortcut
  - Variables required for server location, subpath and connection id

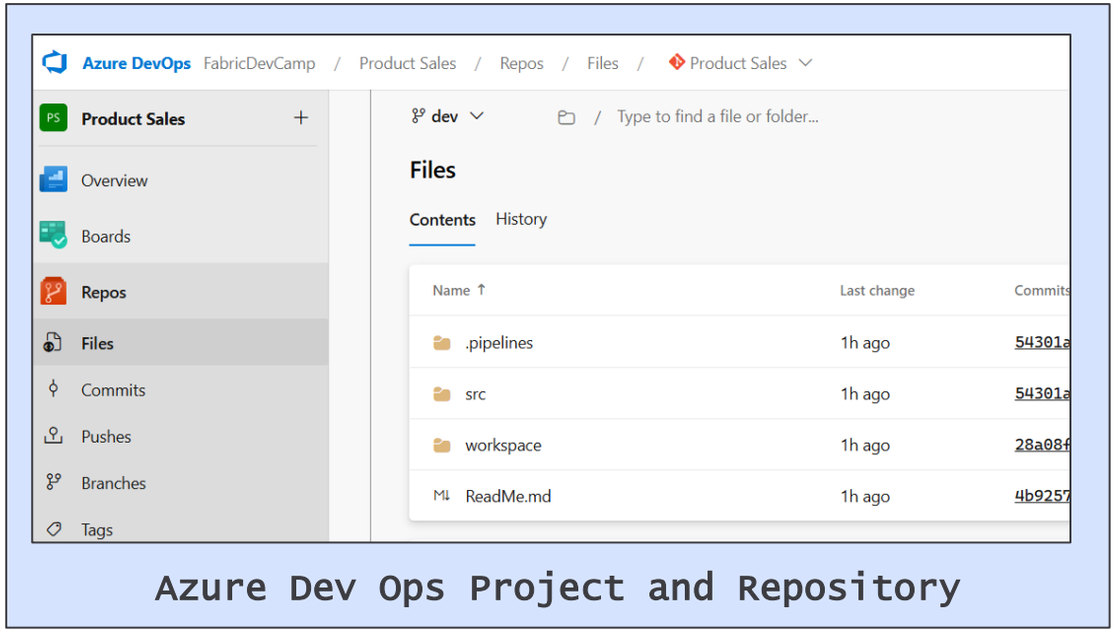


  - Shortcut can be created by hand or by using Create Shortcut API using variable expression syntax

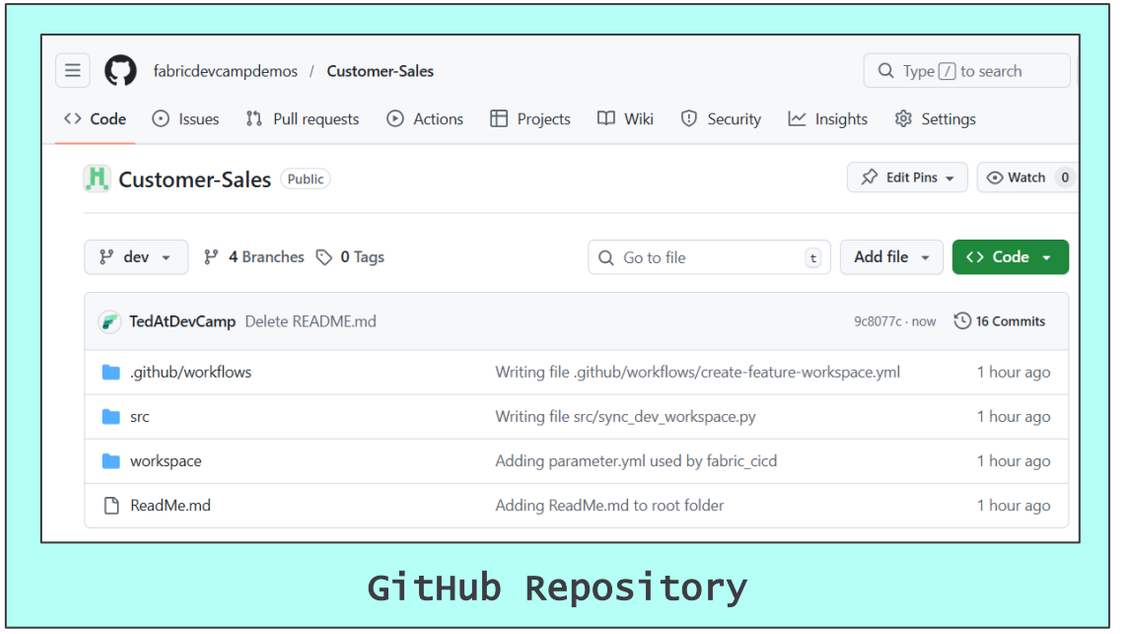
```
[
  {
    "name": "sales-data",
    "path": "/Files",
    "target": {
      "type": "AdlsGen2",
      "adlsGen2": {
        "connectionId": "$(/**/environment_settings/adls_connection_id)",
        "location": "$(/**/environment_settings/adls_server)",
        "subpath": "$(/**/environment_settings/adls_shortcut_subpath)"
      }
    }
  }
]
```

Create and configure a GIT repository

- Select a GIT provider supported by Fabric
  - Choose **Azure DevOps** or **GitHub** or **GitHub Enterprise**
- Create branches and configure branch policies
- Configure the GIT repository with secrets and variables
- Extend the GIT Repository with Workflows



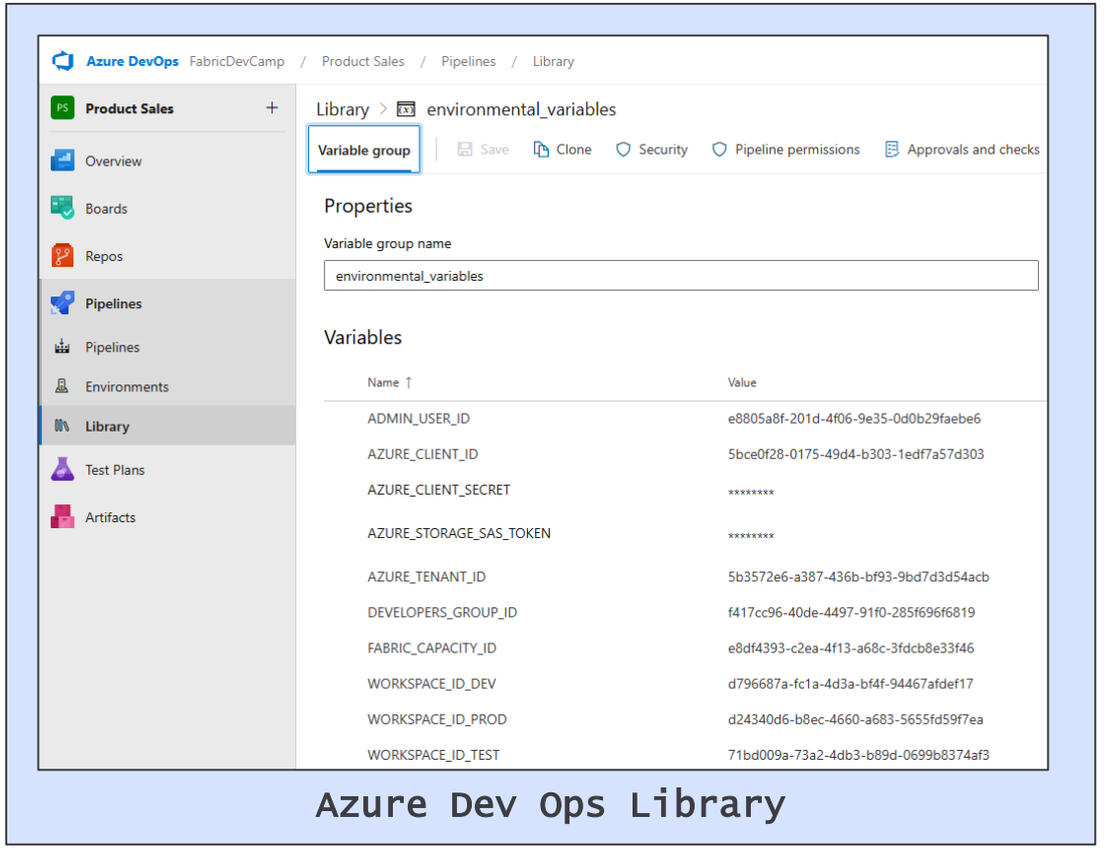
Azure Dev Ops Project and Repository



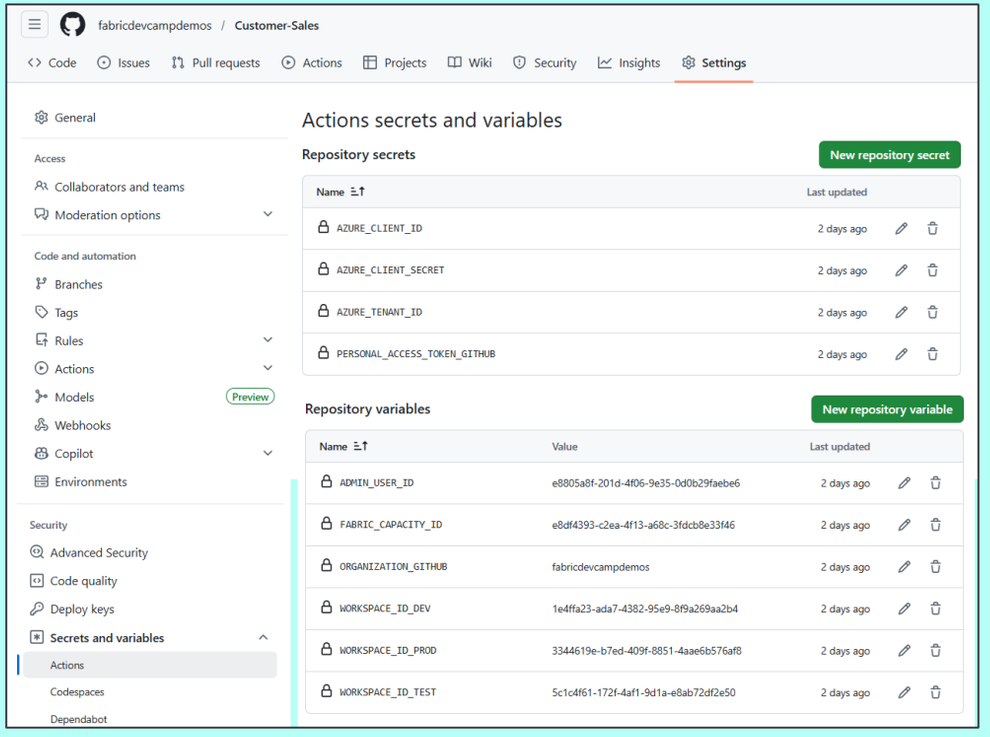
GitHub Repository

Configure GIT Repository with Secrets and Variables

- Add secrets and variables that will be used by workflow code
  - Add secrets for authentication credentials
  - Add variables for the Ids of workspaces, capacities, users and Entra Id groups



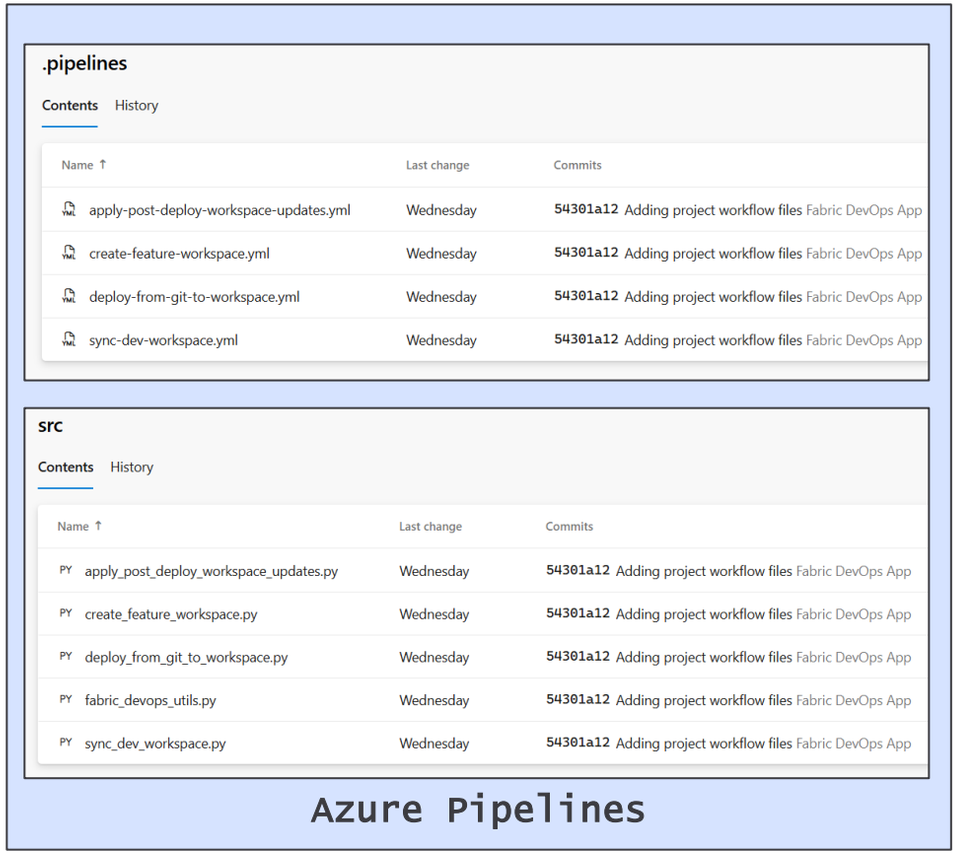
Azure Dev Ops Library

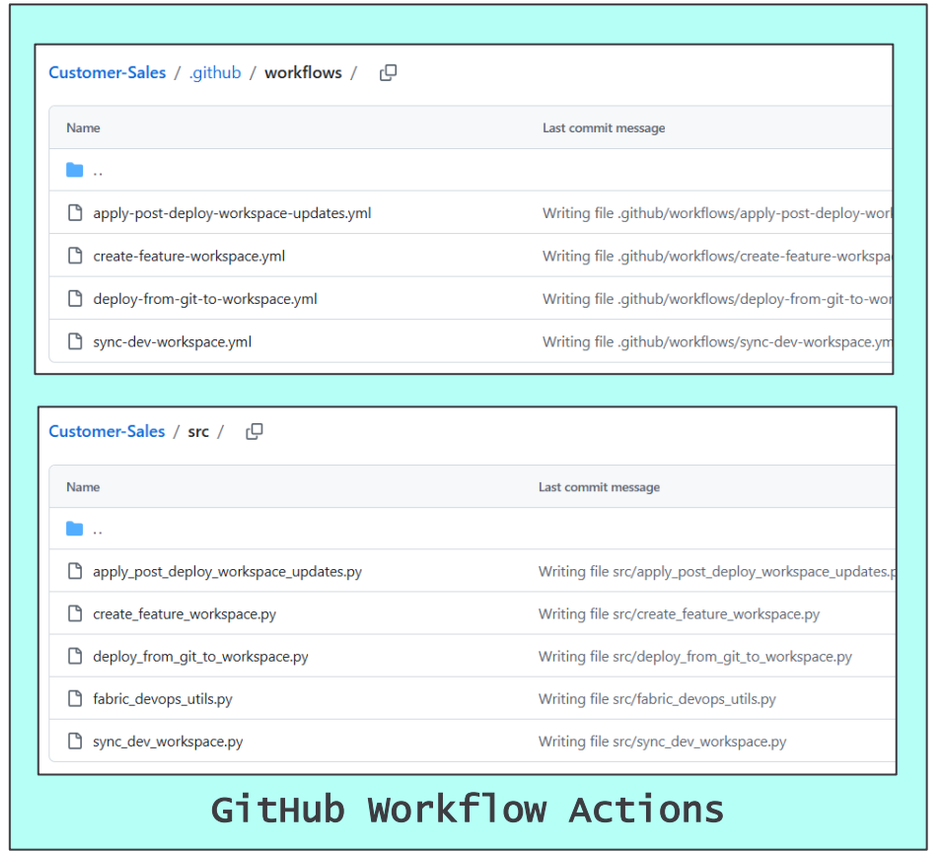


GitHub Repository secretes and variables

Extend the GIT Repository with Workflows

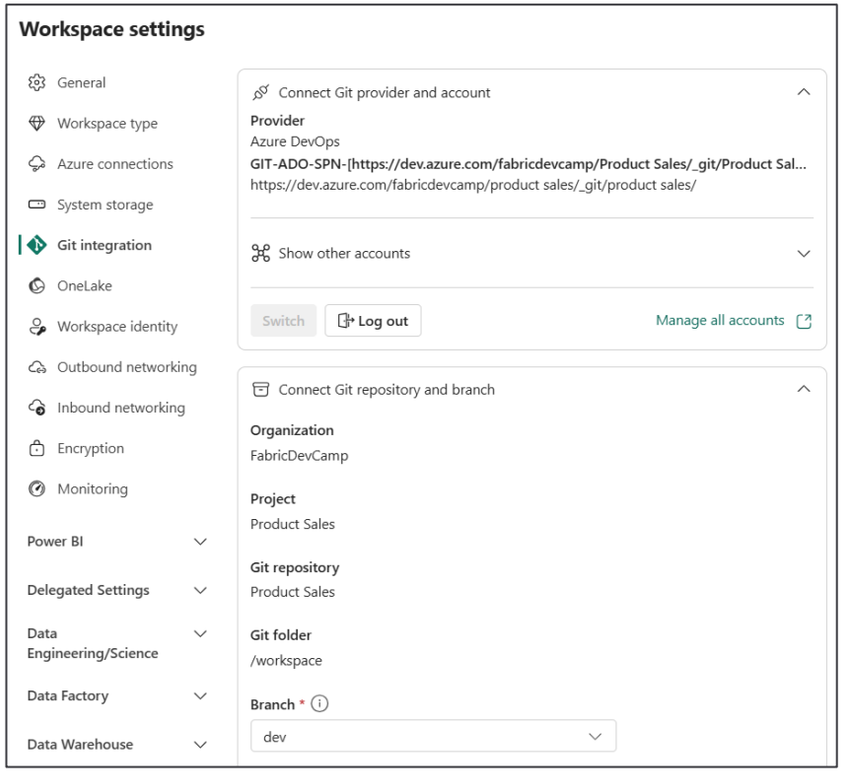
- Develop workflow logic to control Fabric CI/CD lifecycles
  - With Azure Dev Ops, develop workflows using Azure pipelines
  - With GitHub, develop workflows using GitHub Workflow Actions

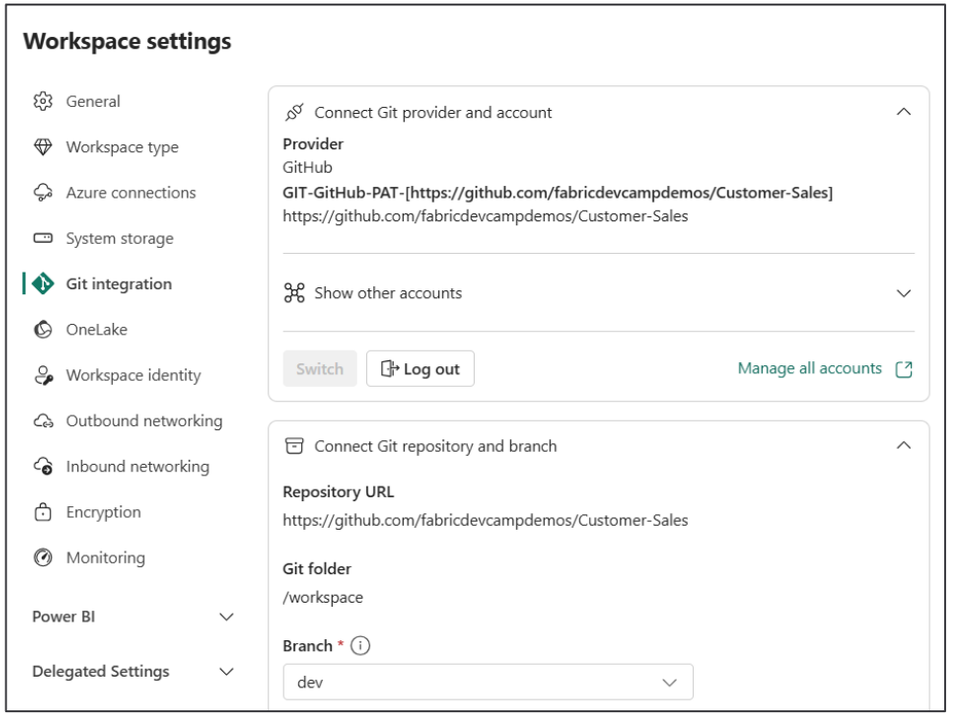




Connect the dev workspace to integration branch

- Fabric GIT integration enabled at workspace scope
  - Connect dev workspace to integration branch using GIT source control connection
  - Configure **GIT folder** setting so item definition files are not added to root folder





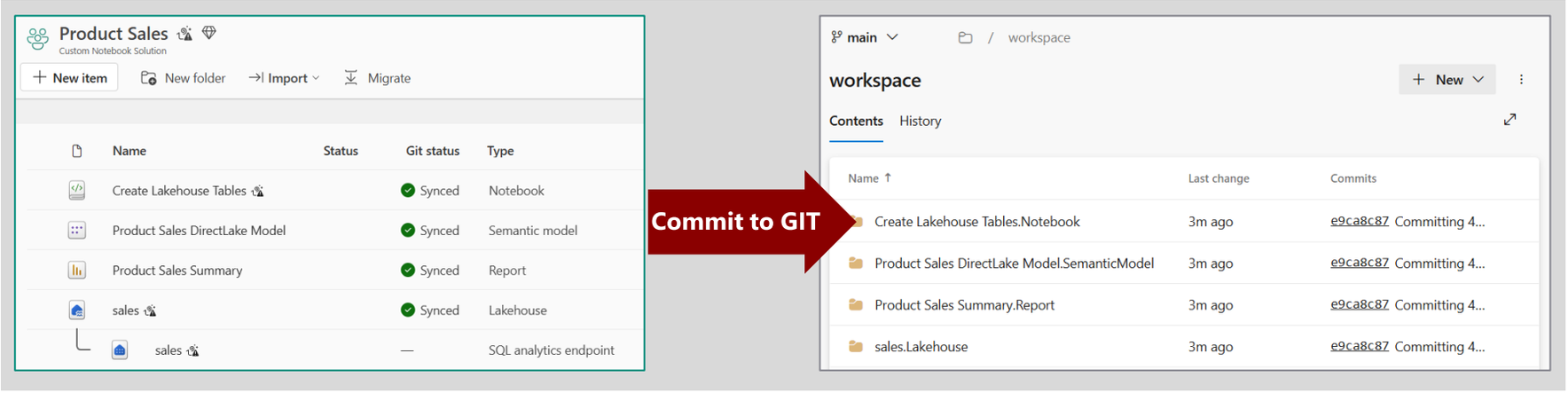
GIT Source Control Connections

- Fabric provides GIT source control connectors for creating GIT connections
  - Azure Dev Ops source control connection can be created using service principal credentials
  - GitHub source control connection must be configured using a personal access token (PAT)

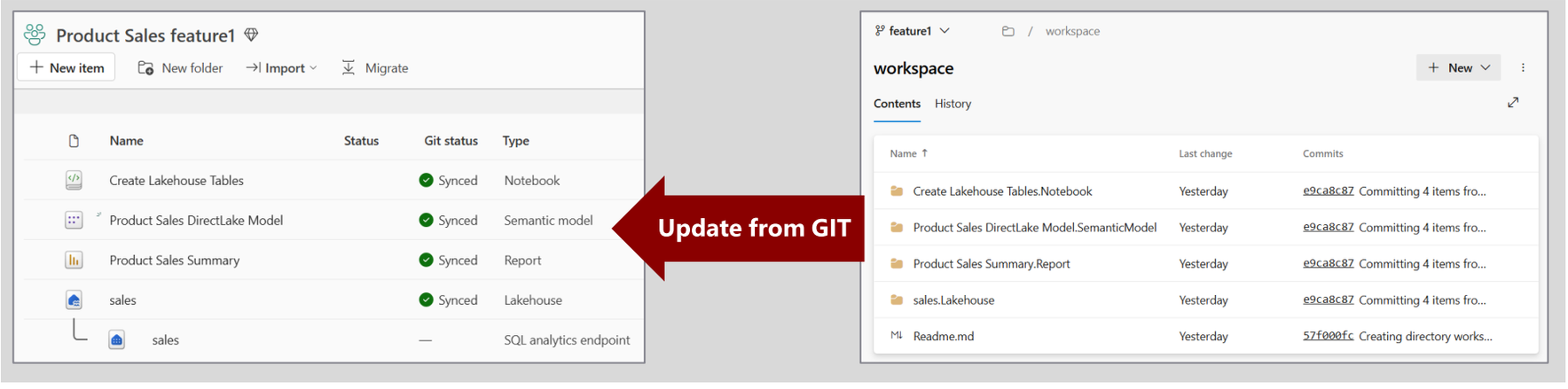
- Source control connection has path to repository - not to a specific branch
  - You can reuse single source control connection to connect N branches to N workspaces

Serialization of Workspace Items to Item Definitions

- When you connect workspace to GIT branch…
  - Fabric GIT integration serializes workspace item definitions and writes definition files to branch



  - Fabric GIT integration can create/update workspace items from item definitions in GIT branch

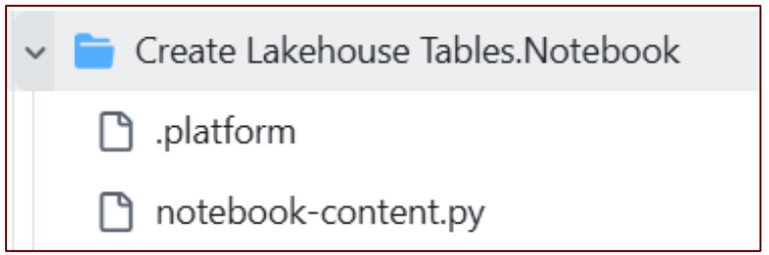


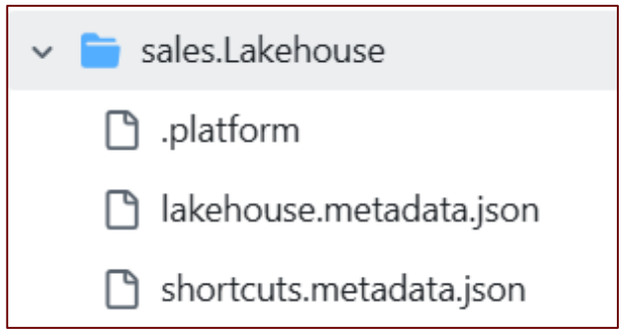
Item Definition Files

- item definition is set of files that contain the metadata for workspace item instance
  - All item definitions contain common platform file name **.platform**

```
{
  "$schema": "https://developer.microsoft.com/json-schemas/fabric/gitIntegration/platformProperties/2.0.0/schema.json",
  "metadata": {
    "type": "Notebook",
    "displayName": "Create Lakehouse Tables"
  },
  "config": {
    "version": "2.0",
    "logicalId": "ddd40c6a-7e76-a701-4221-8af9259bd045"
  }
}
```

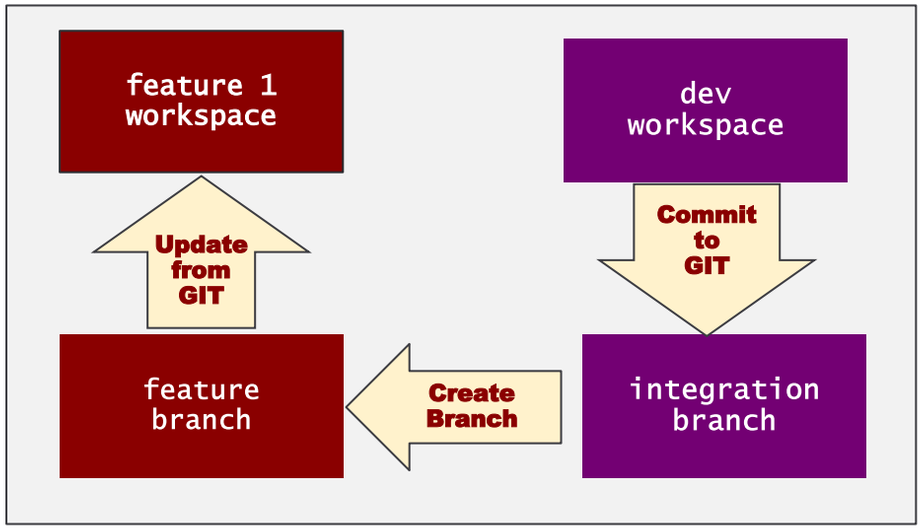
- Files required and/or allowed in item definition determined by workspace item type





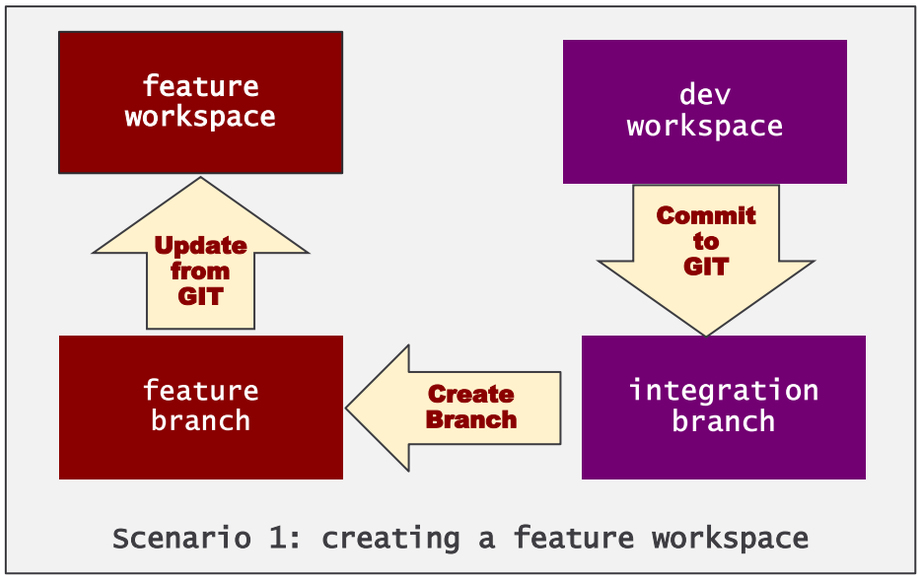
Create Feature Branches and Feature Workspaces

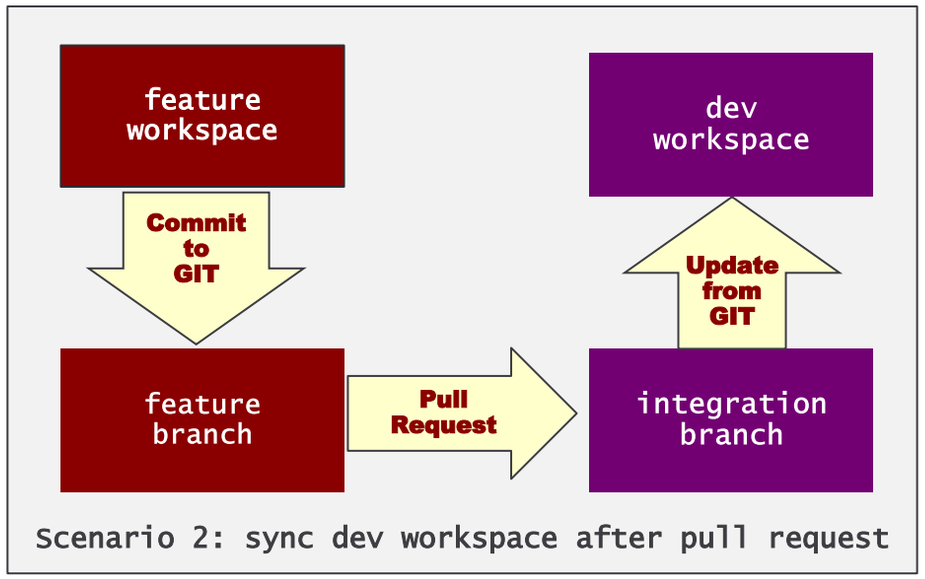
- Development process based on feature branches and feature workspaces
  - New feature branch can be created from release branch
  - New feature workspace can be initialized from branch using **Update from GIT**
  - GIT synchronization creates matching set of workspace items
- Workspace items might need modification before workspace is ready
  - You might need to run pipeline/notebook to populate lakehouse with data
  - You might need to to create connection to bind semantic model to datasource



Automation Required After Update-from-GIT Operation

- Automation often required after Update from GIT operation completes
  - Workspace items need modification before workspace is ready for use
  - **Post-deploy jobs** run only once after an empty workspace is first initialized
  - **Post-sync jobs** run each time after an Update from GIT operation completes on a workspace
  - Updates required for workspace items that do not support auto-binding
  - Updates required for workspace items that do not support variable libraries

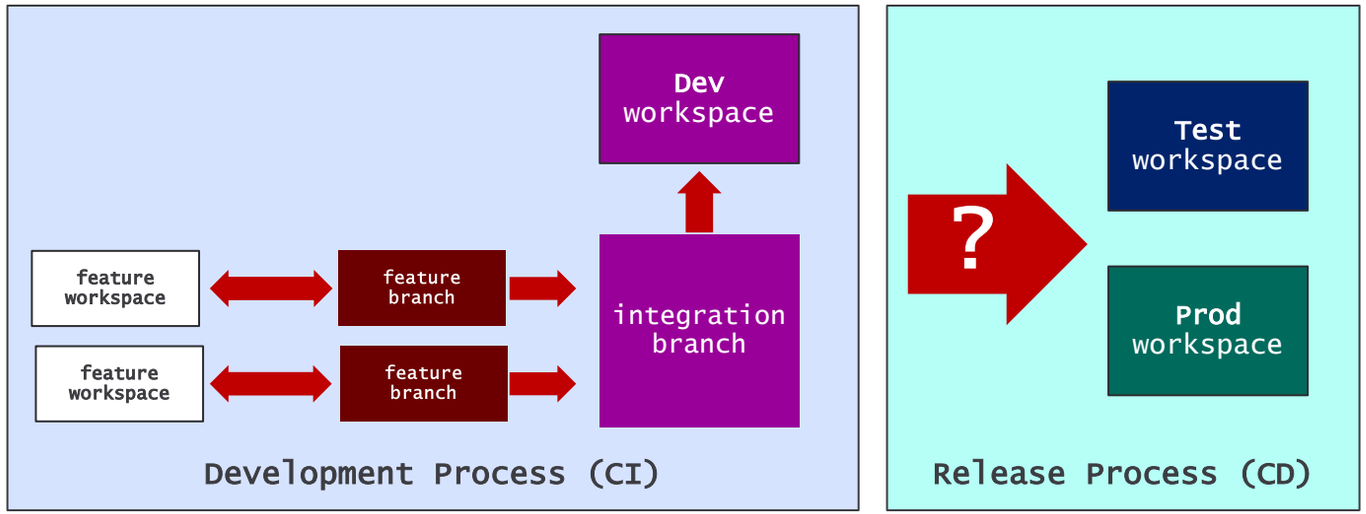




Building a Release Process

Building th Release Process for Continuous Intergation

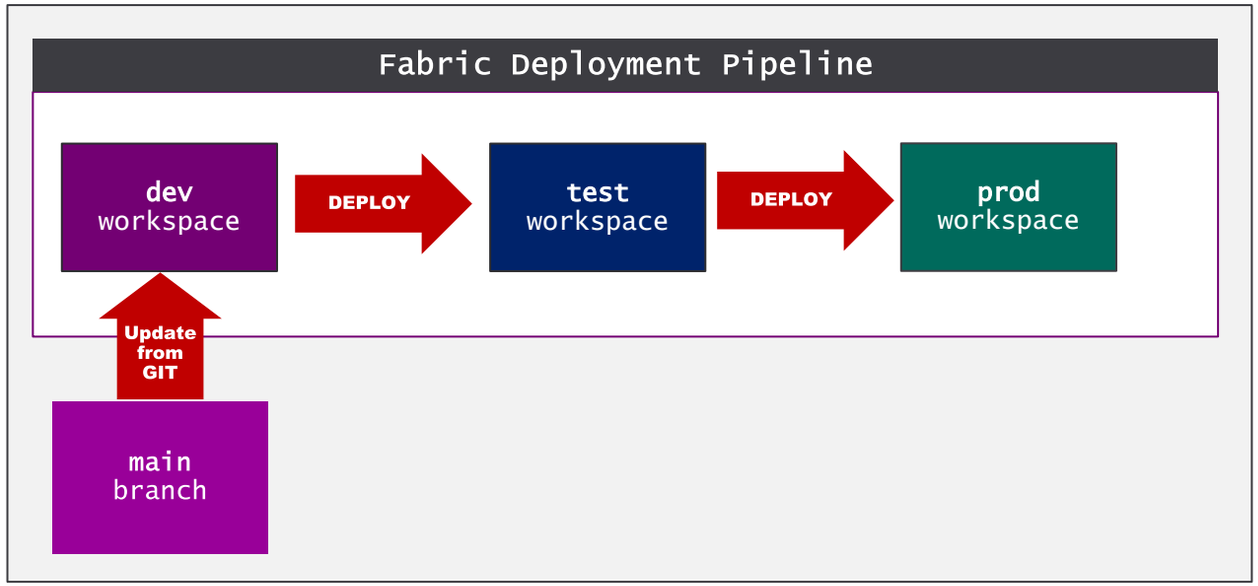
- Release process built to deploy updates from integration branch to **test** and **prod**
  - Deploy to **test** workspace for testing and, once approved, deploy to **prod** workspace



- Choose between several common options for building a release process
  - **Option 1**: Build the release process using **deployment pipelines**
  - **Option 2**: Build the release process using **GIT synchronization**
  - **Option 3**: Build the release process using **fabric-cicd** and **GitFlow** branching
  - **Option 4**: Build the release process using **fabric-cicd** and **release flow** branching

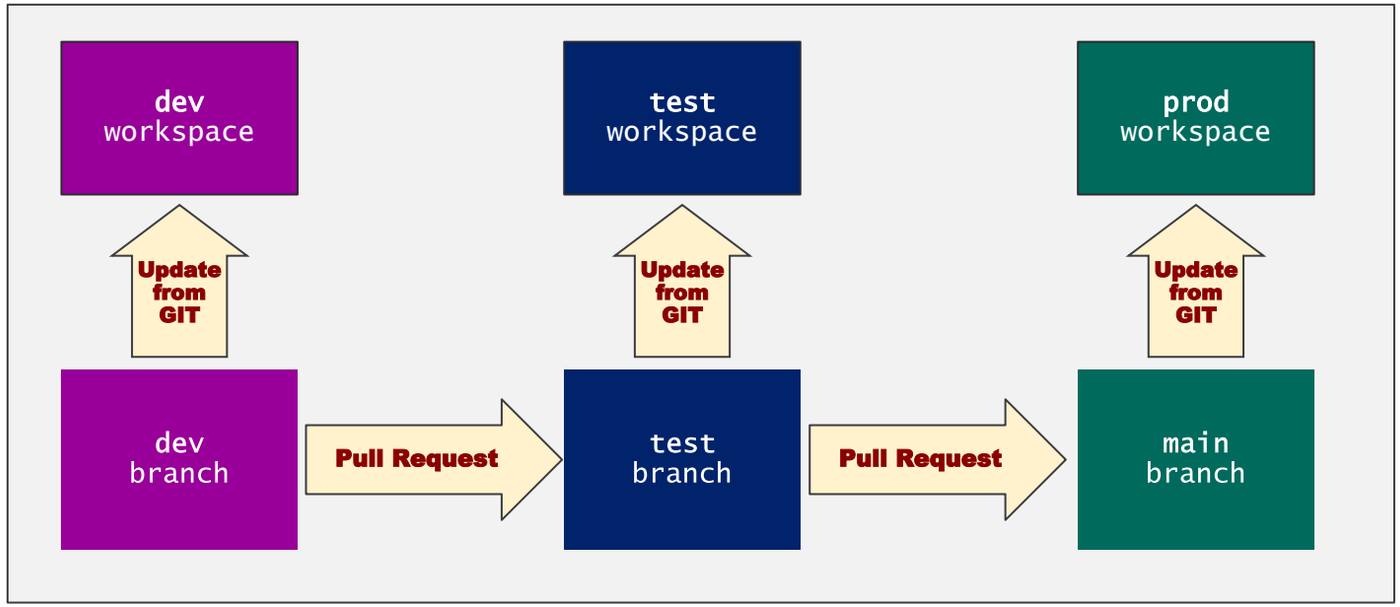
Option 1: Release Process using Deployment Pipelines

- Use GIT integration for CI and deployment pipeline for CD
  - **main** branch serves as integration branch
  - **dev** workspace kept in sync with **main** branch using automated **Update from GIT** action
  - Workspace item changes moves across workspaces using deployment pipeline **Deploy** action



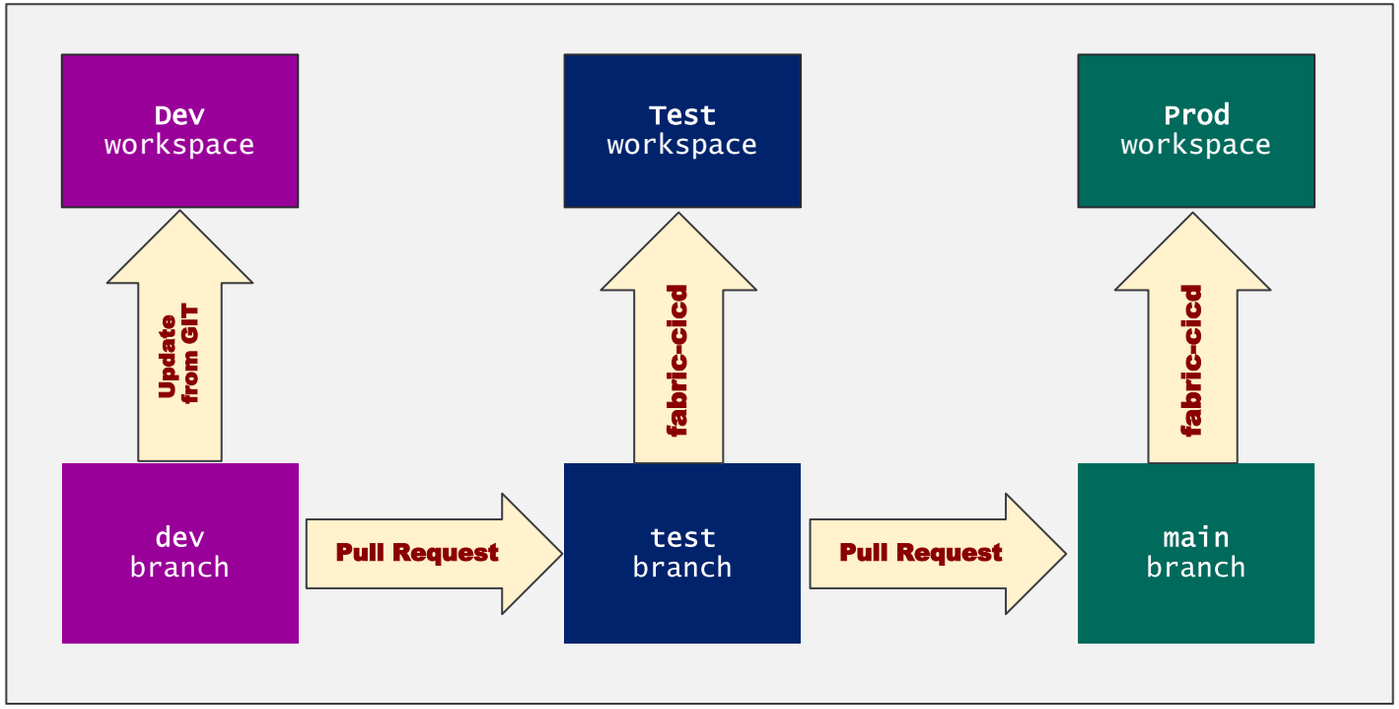
Option 2: Release Process using GIT Sync

- Use GIT synchronization to publish changes to **test** and **prod**
  - **dev** branch serves as integration branch
  - All workspace kept in sync with underlying branch using automated **Update from GIT** action
  - Workspace item changes moved across branches using pull request



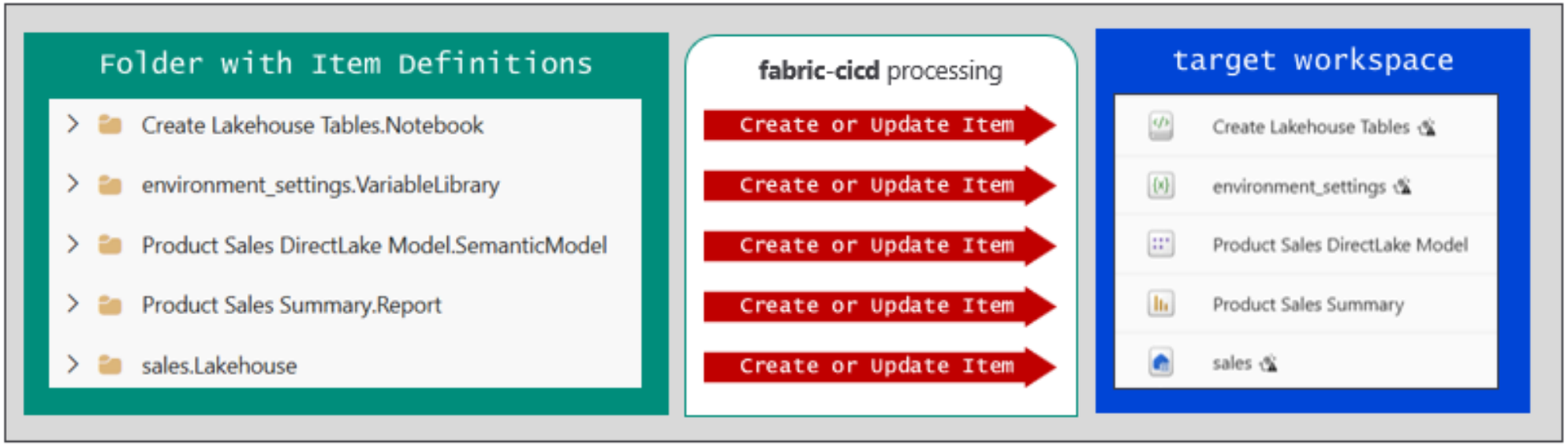
Option 3: Release Process using fabric-cicd and GitFlow

- Use **fabric-cicd** library to deploy changes to **test** and **prod**
  - **dev** branch serves as integration branch
  - **dev** workspace kept in sync with underlying branch using automated **Update from GIT** action
  - **test** and **prod** workspaces kept in sync with underlying branch using **fabric-cicd** library



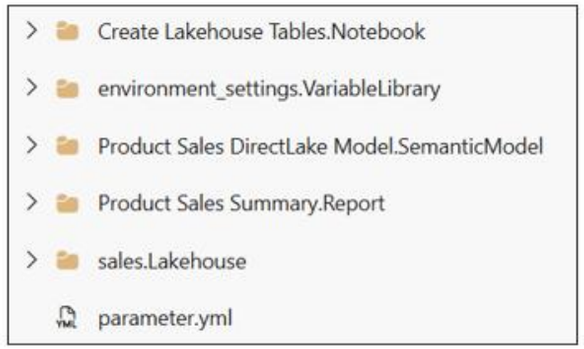
The fabric-cicd Python Library

- **fabric-cicd** library provides ease-of-use wrapper around Fabric REST APIs
  - Provides reusable library of code for implementing a Fabric release process
  - Code-first designed allows for deployment from item definitions folder to target workspace
  - Provides parameterization options for environment-specific setting persisted to GIT branch



Parameterization with fabric-cicd

- Parameterization allows updates to item definition files on the fly
  - Parameterization required to deal with environment-specific settings committed to GIT
  - Parameterization enabled by adding **parameter.yml file** to root folder



  - Parameterization used to update item definition files using substitution patterns (e.g. find-replace)

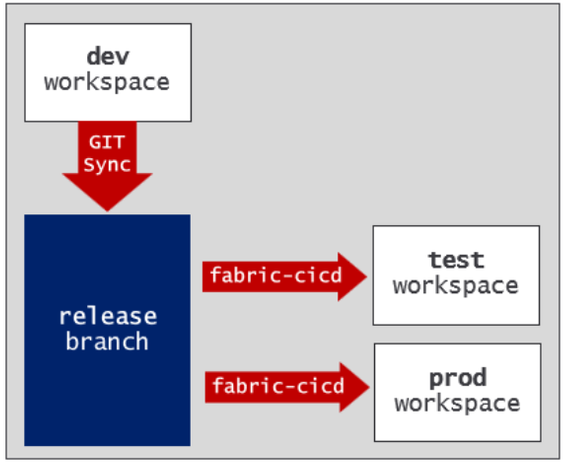
```yaml
find_replace:
- find_value: "https://onelake.dfs.fabric.microsoft.com/{dev-workspace-id}-{dev-lakehouse-id}" # [dev]
  replace_value:
    TEST: "https://onelake.dfs.fabric.microsoft.com/{test-workspace-id}-{test-lakehouse-id}"   # [test]
    PROD: "https://onelake.dfs.fabric.microsoft.com/{prod-workspace-id}-{prod-lakehouse-id}"   # [prod]
  Item_type: "SemanticModel"
  file_path:
    - "/definition/expressions.tmdl"
```

Python Code for fabric-cicd Deployment

- Create FabricWorkspace object and call publish_all_items - that's it!

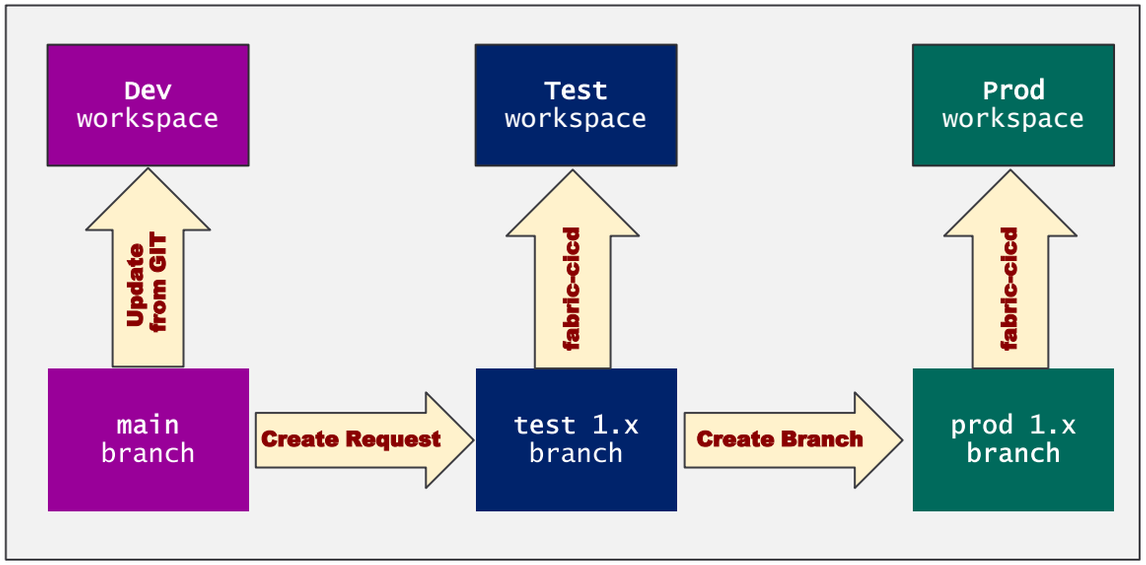
```
test_workspace = FabricWorkspace(
  environment = "TEST",
  repository_directory = "{PATH_TO_FOLDER_WITH_ITEM_DEFINITION}",
  workspace_id = "{TEST_WORKSPACE_ID}",
  item_type_in_scope = ["Lakehouse", "Notebook", "SementicModel", "Report", "VariableLibrary"]
)

publish_all_items(test_workspace)

prod_workspace = FabricWorkspace(
  environment = "PROD",
  repository_directory = "{PATH_TO_FOLDER_WITH_ITEM_DEFINITION}",
  workspace_id = "{PROD_WORKSPACE_ID}",
  item_type_in_scope = ["Lakehouse", "Notebook", "SementicModel", "Report", "VariableLibrary"]
)

publish_all_items(prod_workspace)
```

- **fabric-cicd** provides flexibility of one-to-many between GIT branch and workspaces



Option 4: Release Process using fabric-cicd and release flow

- Option 4 is built on trunk-based development approach
  - **main** branch serves as integration branch with **dev** workspace kept in sync using **Update from GIT**
  - Release process does not include log-lived branches for **test** and **prod**
  - Release process involves creating new test branch for each release
  - Upon creation, **test** branch deployed to **test** workspace using **fabric-cicd**
  - Once **test** branch approved, **prod** branch created from **test** branch and deployed with **fabric-cicd**
  - Creating of **test 1.x** and **prod 1.x** branches creates valuable release history



Session Summary

- Essential Concepts and Capabilities
- Planning the Fabric CI/CD Project Lifecycle
- Creating Project Infrastructure using Terraform
- Building a Development Process
- Building a Release Process

Sound off.
The mic is all yours.
Influence the product roadmap.

Join the Fabric User Panel

Share your feedback directly with our Fabric product group and researchers.

https://aka.ms/JoinFabricUserPanel

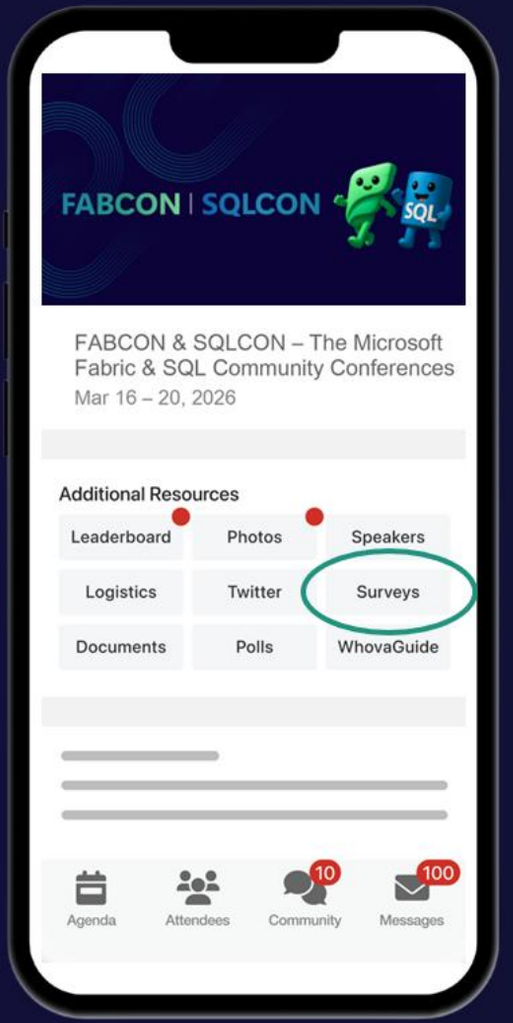
Join the SQL User Panel

Influence our SQL roadmap and ensure it meets your real-life needs

https://aka.ms/JoinSQLUserPanel

How was the session?

Complete Session Surveys in Whova for your chance to WIN PRIZES!



Get Two Fabric Certifications for FREE

Attendees of FABCON can take the Fabric Analytics Engineer or Fabric Data Engineer exam for free. Be part of the 2 fastest growing role-based certifications in Microsoft history.

Request your voucher by March 23, 2026.

https://aka.ms/fabcon/cert100