In this lecture, we're going to walk through how to implement a CI/CD workflow using **Databricks Asset Bundles (DABs)**. We'll cover how to plan a project, set up testing strategies, and deploy workloads efficiently across different environments.

Planning the Project

Requirements

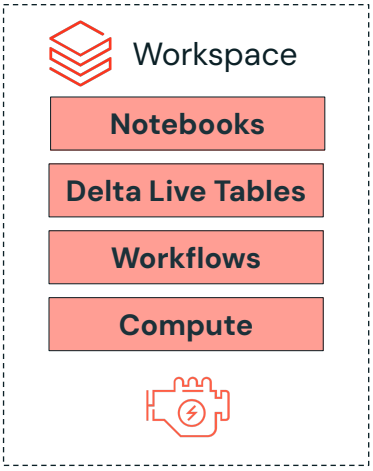
Deliverable

- Visualize Health Data

Tasks

- Ingest daily incremental CSV files to a bronze table
- Create a clean silver table
- Create gold tables to share with consumers

Databricks Assets



To start, let's discuss what **CI/CD** is and how **Databricks Asset Bundles (DABs)** streamline the process.

- **Continuous Integration (CI):** Automating code integration and testing.
- **Continuous Delivery/Deployment (CD):** Ensuring code is always in a deployable state.

Databricks Asset Bundles simplify this process by allowing us to **define, version, and deploy assets in a structured way using the Databricks CLI**.When planning a project, it's crucial to define:

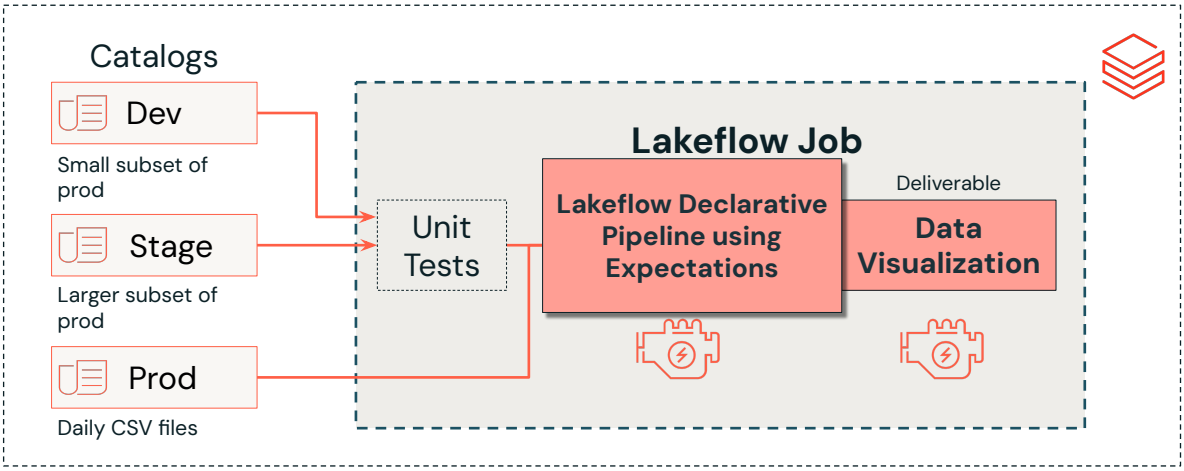
- **Deliverables** → What are we building?
- **Tasks** → What are the key steps to achieve this?
- **Databricks Assets** → What components are required?

In this case, we are **visualizing health data**. The key tasks include:

1. **Ingesting a daily incremental CSV file** into a **bronze table**
2. **Cleaning the data** and moving it to a **silver table**
3. **Creating a gold table** to share with consumers.

Planning the Project

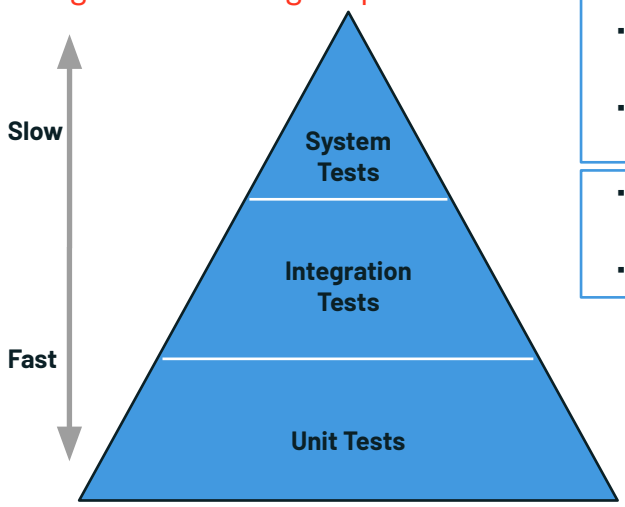
Course Project Architecture



Now that we have an understanding of our project, our data, and how to isolate environments, let's dive into the project setup.
In this project we will create a catalog for each environment: dev, stage, and prod.
The dev catalog will contain a small, static subset of our production data, used for development and initial testing.
The stage catalog will hold a larger subset of production data, enabling more comprehensive testing as we move through our CI/CD process.
Finally, the prod catalog will contain the live production data that we rely on for final operations.
Our workflow starts by developing on the dev data, running unit and integration tests as we progress through the pipeline. The pipeline is set up with Databricks Workflows, which execute the unit tests, Lakeflow Declarative pipeline, and final visualizations.
As we test the pipeline through each stage, we will ensure everything is functioning correctly before deploying to production.

Review the Role of CI/CD Testing

High-level Testing Steps



- Test the entire application, ensuring that all parts function together in a real-world scenario
- **Ex:** End to end data pipeline in a Workflow

- Test the interaction between different components or systems
- **Ex:** Notebooks / Declarative Pipelines/ Jobs

- Test **individual** functions or methods in isolation
- Fast, low cost, high coverage, and automated
- **Ex:** Custom pyspark functions

Within CI/CD there are testing steps you should following within what's called the testing pyramid. The testing pyramid categorises different tests, unit tests, integration tests and system tests. Testing your code is extremely important.

- The base of the pyramid are unit tests which test individual functions or methods in isolation. Since they small individual functions, they typically can run quickly, frequently and automatically, ensuring that the functions work as expected. Unit tests form the foundation because they are inexpensive and provide the broadest coverage. For example, testing if a pyspark method works as expected.
- Next is integration tests test the interaction between different components or systems. These are typically slower and more costly than unit tests, but provide greater assurance that components work together correctly. Within Databricks these typically will revolve around using Notebooks, Lakeflow Declarative Pipelines and or Lakeflow Jobs. For example, testing whether a pyspark method and pipeline work correctly.
- Lastly system tests test the entire application, ensuring that all parts function together in a real-world scenario. These are typically slow, costly, and often run in a production-like environment. For example, for our end to end data pipeline, testing whether the data data pipeline works as expected within a Workflow, creating our desired results.

Review of Unit Testing with pytest

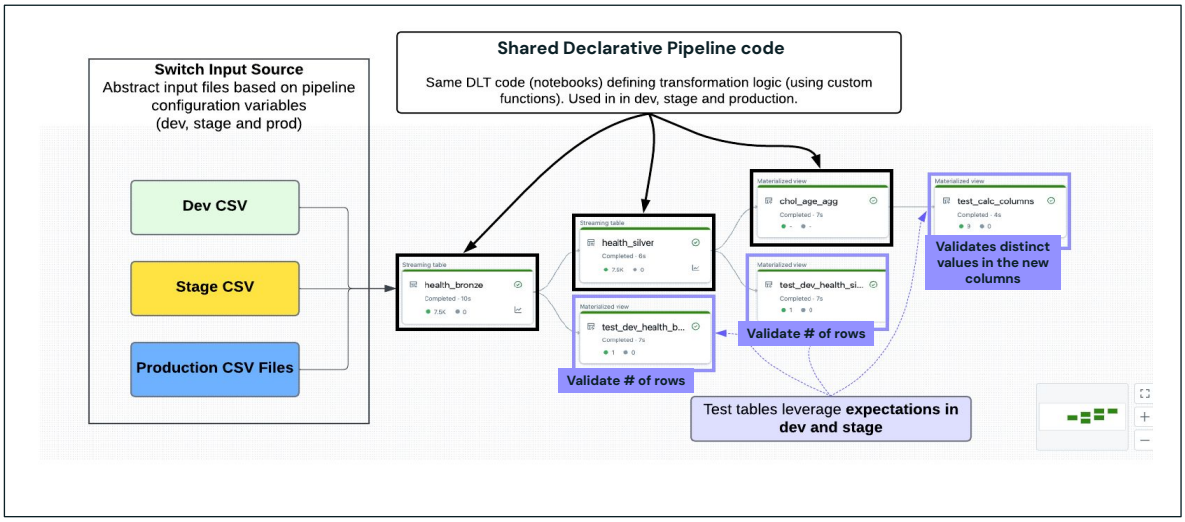
Pytest -is popular testing framework for Python that makes it easy to write simple and scalable test cases.

Uses Simple Syntax	Provides Assertions	Automatic Discovery	Rich Ecosystem
Minimal syntax, just define functions starting with **test_**	Use **assert** statements to provide detailed error messages on failure	Finds and runs all tests **automatically** with a simple configuration	**Extend** functionality with plugins for coverage, parallel tests, and more

This course provides a simple introduction to **pytest**. There are many testing frameworks available, select the one that best meets your organization's needs.

Next like talk about the pytest unit testing framework.
Pytest -is popular testing framework for Python that makes it easy to write simple and scalable test cases. It provides a variety of benefits for executing your unit tests.
First up, pytest has a very simple syntax. You don't need to worry about complex setup. Just write test functions that start with test_, and pytest will automatically pick them up and run them.
Next, pytest leverages Python's built-in assert statements (or you can use other assert statements like in pyspark). These are simple to use but provide detailed error messages when a test fails, making your function debugging much easier and faster by understanding what went wrong.
Pytest also automatically discovers and runs all your tests. There's no need to manually configure which tests to run. Just name your test files and functions with the test_ prefix, and pytest will take care of the rest.
Lastly, pytest has a rich ecosystem of plugins to extend its functionality. Whether you need test coverage reports, parallel test execution, or integration with other tools, there's a plugin for almost anything, allowing you to customize pytest to fit your needs.
This course provides a simple introduction to pytest. There are many testing frameworks available, select the one that best meets your organization's needs.

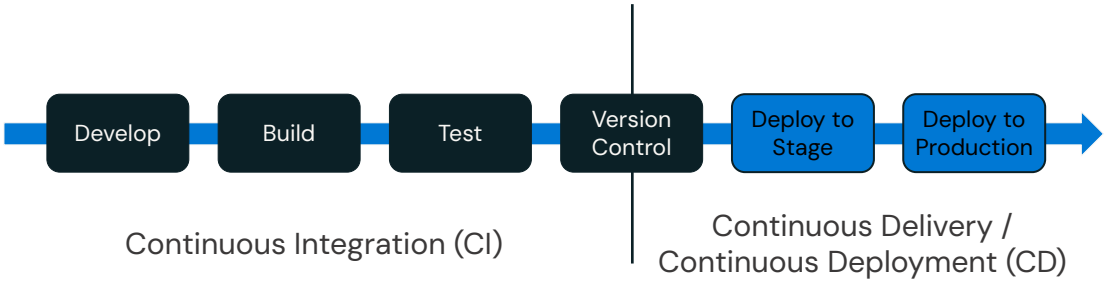
Review of Pipeline Expectations for Integration Tests



Lakeflow Declarative Pipelines expectations are data quality constraints designed to validate data as it flows through ETL pipelines. By defining these expectations, you can ensure that your data meets specific criteria before moving on to further processing. They provide an automated way to verify that transformations and pipeline logic deliver the intended outcomes. Incorporating expectations into Lakeflow Declarative Pipelines helps detect and handle data quality issues early, resulting in more robust and reliable data processing workflows.

The Role of CI/CD in DevOps

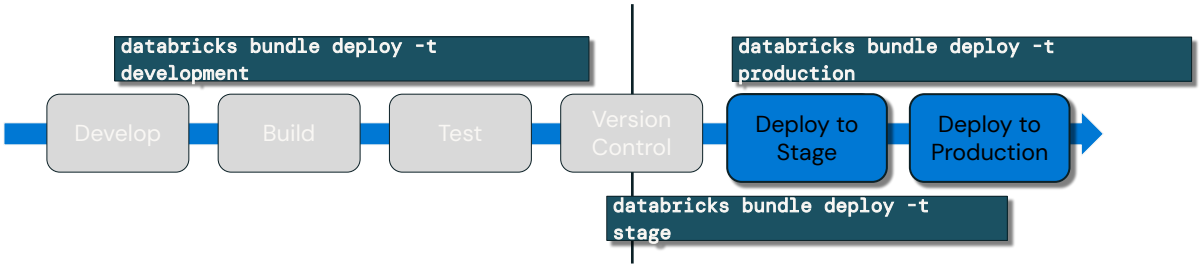
High-Level CI/CD Workflow Overview



Here is a high-level view of CI/CD and DevOps. Continuous Integration (CI) involves develop, build, test, and version control. Continuous Delivery/Deployment (CD) focuses on deploying to staging and production environments in an automated way.

CI/CD with DABs

High-Level CI/CD Workflow Overview



The idea with DABs is to automate the deployment process in each environment, whether development, staging, production, or QA. DABs provide an infrastructure-as-code approach to managing Databricks resources, making it easier to version, deploy, and collaborate on data and AI projects. They support defining multiple deployment targets to manage different environments and their specific configurations. For example, deploying to development uses `databricks bundle deploy -t development`, to staging `databricks bundle deploy -t stage`, and similarly for production. By leveraging DABs for CI/CD, data teams can adopt software engineering best practices to improve collaboration and streamline deploying and managing Databricks projects across environments.

Thank you for completing this lesson and continuing your journey to develop your skills with us.