WHITEPAPER

Unlocking Data Productivity

A DataOps Guide for High-Performance Data Teams

Version 1.0 | 2025 - 03 - 31

matillion

Executive summary

Problem

In today's data-driven world, organisations are under pressure to deliver high-quality, actionable insights faster than ever. However, traditional methods of building and maintaining data pipelines are slow, error-prone and resource intensive. Misaligned teams, inconsistent processes and limited automation worsen these challenges and result in high operational costs, reduced trust in data outcomes and delayed time-to-market.

Context

The DataOps revolution has changed how businesses approach data pipeline design, testing and deployment. Inspired by the DevOps movement in software engineering, DataOps emphasises collaboration, automation and agility with a focus on delivering value to consumers as quickly and frequently as possible. However, many organisations face barriers to adopting these principles, including a lack of expertise, limited resources and the complexity of integrating disparate, inconsistent tools and workflows. These challenges have created a need for a unified, accessible platform that connects data teams to business goals, allowing them to create actionable insights to power organisational success.

Challenges

Existing solutions often fail to address the complexity of modern data workflows. They create siloed teams that rely on manual processes that are difficult to scale and result in inefficiencies and increased risks of pipeline failures. An option is to hire specialised DataOps experts, but this comes with a significant cost and time investment.

The cost of inaction is also substantial: delays in deploying data pipelines lead to missed opportunities, outdated insights and reduced competitiveness in fast-moving markets. Organisations need a solution that not only simplifies DataOps processes but also empowers existing teams to succeed without large upfront investment through additional personnel or infrastructure.

Solution overview

Matillion's Data Productivity Cloud provides an innovative approach to DataOps. It enables teams to design, test and deploy data pipelines efficiently no matter the scale. Combining powerful automation and seamless integration with familiar tools like Git providers and their built in CI/CD capabilities, the Data Productivity Cloud eliminates many of the traditional barriers to adopting DataOps practices. The intuitive interface and pre-built connection capabilities empower teams to develop rapidly & collaboratively, reduce errors and deploy pipelines with confidence - all without requiring a team of dedicated DataOps specialists.

Key benefits

Organisations adopting the Data Productivity Cloud can expect transformative outcomes for their Data capabilities, including:

- **Cost efficiency:** Simplified processes and automation reduce the need for specialized personnel and resource consumption.
- **Increased Reliability:** Automation and version control ensure pipelines are consistent, secure, and error-free.
- **Enhanced Collaboration:** Distributed Version Control tools improve team productivity and reduce friction.
- **Scalability:** A unified platform allows seamless growth of data operations with minimal overhead.
- **Improved Agility:** Faster iterations and deployments enable teams to adapt quickly to business needs.

Through the Data Productivity Cloud, organizations can focus on driving business value rather than wrestling with the complexities of traditional data management approaches.

Table of contents

Introduction 6

Overview of DataOps 7

- Safely collaborating with colleagues on pipelines 8
- Reviewing and approving work done by colleagues 8
- Ensuring data pipeline quality 8
- Managing pipelines across multiple environments 8
- Monitoring and performance 8
- Automating tools and processes to manage data workflows 8

The DataOps lifecycle 9

- Design phase 10
- Test phase 10
- Release phase 10

Key pillars of DataOps in action 10

Initial Data Productivity Cloud project setup 11

Quick-Start guide 12

Production-ready deployment 13

- Best practices for environment setup and naming 13
- Reference architecture 13

Recommended environment setup 15

Integrating with a Git provider 16

Inviting collaborators 16

Best practices for assigning roles 17

- Project access 17
- Environment access 17

Additional considerations 18

Final notes on project setup 18

Design phase 19

- Principles 20
- Tools and methodologies 22

Collaboration and version control 22

- Git actions available through designer 23

Table of contents

- Git core concepts 27
- Quick reference table 29
- Branching strategies 30
- Feature branching 30
- Trunk-based development 32
- Versioning of pipelines 34
- Documentation in DataOps 34
- Key practices 34
- Benefits of strong documentation 34
- Environment-specific configuration 35
- Motivation for environment-specific configuration 35
- Implementing environment-specific configuration in data pipelines 36
- Best practices and risk mitigation 36
- Dynamic pipeline configuration in practice 37
- Example use case: conditional execution in a data pipeline 37
- Step 1: Setting the project variable 37
- Step 2: Configuring the if component 38
- Step 3: Updating the configuration for promotion 38
- Step 4: Safe rollout and quick reversion 38
- Best practices for environment-specific configuration 39
- Summary of best practices for designing pipelines 40

Test Phase 41

- Principles 42
- Tools and methodologies 43
- Publishing from feature branches for testing 44
- Key benefits 44
- Best practice 44
- Example workflow 44
- Summary of best practices for testing pipelines 45

Table of contents

Release phase 46

Artifact management and deployment 47

Key features 47

Best practice 47

Example CI/CD pipeline: GitHub actions 48

Overview of the GitHub action 50

Step-by-step explanation of each job 51

Summary of best practices for release management 52

Conclusion 53

Introduction

This whitepaper outlines the foundational best practices and emerging standards for setting up and managing environments, components, and pipelines within Matillion's Data Productivity Cloud. As organisations increasingly rely on Matillion's Data Productivity Cloud to drive innovation and streamline data workflows, establishing robust and efficient processes is critical to ensuring reliability, scalability, and ease of use.

The guidance provided includes recommendations for environment configuration, component packaging, pipeline validation, and deployment strategies, offering actionable insights to simplify development while maintaining high operational standards.

As the platform matures and standards emerge, the practices outlined in this document are both expected and desired to evolve. The intention is to release updated versions of this document to reflect these advancements, ensuring it remains a relevant and valuable resource for practitioners.

This whitepaper will give readers a clear understanding of how to configure their environments, build and package components, and establish validation processes for pipelines. This document is best used during the initial stages of planning or improving data workflows within Matillion's Data Productivity Cloud, ensuring that teams can minimise friction, enhance collaboration, and build scalable, secure solutions that align with business needs.



01

Overview of DataOps

DataOps, short for Data Operations, is an agile methodology that focuses on improving the speed, quality, and collaboration involved in data management processes. Drawing from DevOps principles, it focuses on the entire data lifecycle, ensuring that data is reliable, secure, and actionable.

DataOps is not just about building data pipelines; it encompasses a wide range of activities that ensure the entire data lifecycle is managed efficiently and collaboratively. Below, we outline the key activities a Data Engineer might undertake within a DataOps framework using Matillion's Data Productivity Cloud:

Safely collaborating with colleagues on pipelines

Collaboration is at the heart of DataOps, with teamwork being essential in multiple areas. Within data teams Data Engineers need to work closely together to develop, modify, and maintain data pipelines, manage version control, and resolve pipeline conflicts. By leveraging tools that facilitate safe and efficient collaboration, teams can work in parallel without stepping on each other's toes, ensuring that the overall workflow remains smooth and uninterrupted.

Beyond the technical team, collaboration is necessary with business stakeholders to align on requirements, priorities and goals, ensuring that pipelines deliver meaningful insights and value. It is also essential to work with IT and Security teams to secure the necessary access and maintain compliance. This multidisciplinary collaboration model ensures a cohesive approach to DataOps and alignment across the organisation

Reviewing and approving work done by colleagues

To maintain high standards of quality and consistency, it's essential to review and approve the work of colleagues. This involves conducting code reviews, ensuring that changes align with best practices, and validating that new code or pipeline configurations do not introduce errors or degrade performance. In DataOps, Automated CI/CD pipelines play a crucial role in catching issues early, ensuring that all changes meet high standards before they are deployed



Ensuring data pipeline quality

Ensuring the overall quality of data pipelines is a fundamental responsibility within DataOps. This goes beyond just data quality and includes testing the consistency, functionality and performance of the entire pipeline. Quality assurance in DataOps involves setting up a variety of automated tests and validations as part of the development process to confirm that data is accurate, consistent, reliable and performant across all stages of the pipeline.

- **Data Quality Testing**
  Automated data quality checks confirm that the data processed by the pipeline adhere to the 6 dimensions of data quality: completeness, uniqueness, timeliness, validity accuracy and consistency. This helps prevent data issues from reaching production, ensuring that downstream processes and analyses are based on trustworthy data.

- **Functional and Integration Testing**
  Functional unit tests validate that individual components of the pipeline work as intended. Integration tests ensure that these components interact correctly with each other and external systems. These verify that the pipeline operates as expected across environments.

- **Regression Testing**
  Regression tests ensure that changes do not introduce bugs or issues into existing functionality. By rerunning tests after changes are made, teams can confirm that the pipeline continues to perform as expected.

- **Performance Testing**
  Performance tests evaluate how the pipeline handles different data loads and conditions. This testing ensures that the pipeline performs efficiently and can scale as data volumes grow, preventing performance bottlenecks in production.

Managing pipelines across multiple environments

Data pipelines often need to operate across multiple environments, such as development, testing, and production. DataOps practices like Versioning, Environment Segregation and Environment-Specific Configuration allow engineers to define and manage their environments in a safe, consistent and repeatable way, ensuring that pipelines behave predictably across environments.

Monitoring and performance

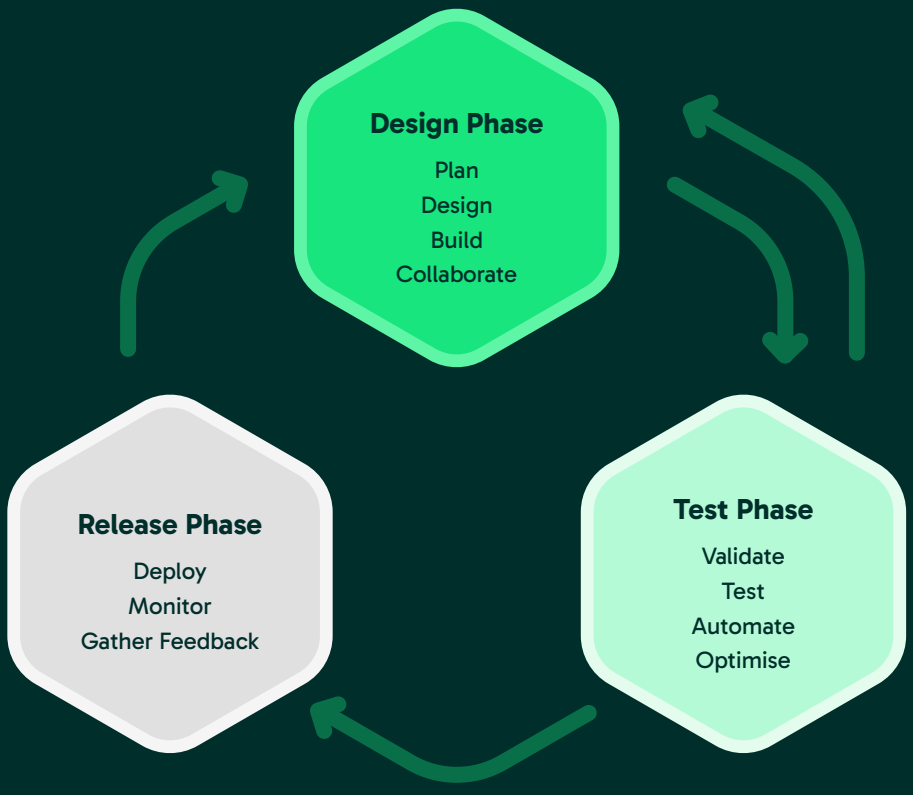
Once data pipelines are in production, monitoring of data availability and pipeline performance is essential. By continuously monitoring these metrics, Data Engineers can quickly identify and resolve issues, ensuring that data remains available and performant for end users and downstream systems.

Automating tools and processes to manage data workflows

Automation is a cornerstone of DataOps, allowing teams to manage complex data workflows with minimal manual intervention. By automating repetitive processes, Data Engineers can focus on higher-level tasks, such as optimising data flows and improving data quality.

The DataOps Lifecycle

The DataOps lifecycle is an iterative process that ensures data pipelines are designed, tested, and released in a controlled and efficient manner. This lifecycle is composed of three interconnected phases, each underpinned by the core pillars of DataOps: collaboration, automation, continuous integration/deployment (CI/CD), flexibility, and infrastructure as code (IaC).



Design phase

The Design Phase is where data pipelines are planned, crafted, and refined. This phase emphasises collaboration, ensuring that data engineers, data scientists, and other stakeholders work together effectively. Leveraging modern version control and collaboration tools, teams can contribute without disrupting each other's work, fostering communication and alignment.

In this phase, automation begins to play a role. DataOps encourages teams to automate repetitive tasks, such as infrastructure setup or environment configuration, using practices like Infrastructure as Code (IaC) and Configuration as Code (CaC). These practices ensure environments are consistent, scalable, and easy to reproduce.

The Design Phase is the creation of the pipeline - establishing its structure, logic, and flow, setting the stage for testing and validation in the next phase.

Test phase

Following the Design Phase, the Test Phase ensures that the pipeline functions as intended. This phase involves rigorous testing to validate the pipeline's performance, security, and data quality. Automated tests, integrated into CI/CD pipelines, play a critical role here, enabling continuous validation to catch issues early and minimize errors in downstream processes.

The principles of flexibility and automation are key during this phase. DataOps practices ensure that pipelines can be adapted to changing requirements without compromising quality. Automated testing frameworks and continuous feedback loops allow teams to iterate rapidly, feeding back into the Design Phase when changes are needed.

Release phase

The Release Phase focuses on deploying tested and validated pipelines into production. Teams rely on CI/CD pipelines to automate deployment, ensuring that changes move seamlessly from staging to production environments. This approach minimises manual intervention, reducing errors and accelerating the delivery of new features and updates.

Once in production, monitoring and alerting systems (often automated) ensure that pipelines perform reliably and securely. Feedback gathered from this phase informs the Design Phase, closing the loop in the iterative lifecycle.

Flexibility remains critical here, as pipelines may need updates to meet evolving business needs or accommodate new data sources. The use of IaC and CaC ensures that any changes to infrastructure or configurations can be applied consistently across environments, supporting a stable and scalable production environment.

Key pillars of DataOps in action

Throughout the lifecycle, the following principles guide the implementation of DataOps practices:

- **Collaboration**
  Fostered during the Design Phase through version control and teamwork tools, collaboration ensures alignment between team members and stakeholders.
- **Automation**
  Foundational to every phase, automation reduces manual effort and enforces consistency, from testing in the Test Phase to deployment in the Release Phase.
- **Continuous Integration/Deployment (CI/CD)**
  Integral to the Test and Release Phases, CI/CD pipelines ensure that changes are integrated and deployed rapidly with minimal risk.
- **Flexibility**
  Supports the iterative nature of the lifecycle, enabling teams to adapt pipelines to changing requirements across all phases.
- **Infrastructure as Code (IaC) and Configuration as Code (CaC)**
  Provides consistency and scalability, ensuring environments and infrastructure are managed as code throughout the lifecycle.

02

Initial Data Productivity Cloud Project Setup

Getting started with a new Data Productivity Cloud project is simple. This guide covers clear, best-practice steps to quickly set up your project to meet your organisation's needs. You'll learn h ow to name environments, establish collaboration guidelines, and integrate with your Git provider for seamless teamwork from the start.

Quick-Start guide: Getting familiar with the Data Productivity Cloud

The Data Productivity Cloud is a powerful platform for designing, testing, and managing data pipelines at scale.

If you are already familiar with the Data Productivity Cloud and want to get started with a production-ready deployment, jump ahead to the Production-Ready Deployment section below.

For those who are new to the Data Productivity Cloud it's essential to familiarise yourself with the platform's core features and workflows. Refer to our Project quickstart guide. Alternatively here are a few key tasks to help you get started:

Once you've completed these tasks, you'll have a strong foundation for understanding how the Data Productivity Cloud operates. From here, you'll be ready for more advanced configurations, such as integrating CI/CD workflows or managing multiple environments for large-scale deployments.



01 Set up your first project and environment

- Begin by creating a new project in the Data Productivity Cloud and linking it to a single environment (e.g. Development) - more details in our documentation
- Use this setup to explore how the Data Productivity Cloud organises resources, including pipelines, variables, and schedules.
- Take note of how environments are isolated, allowing you to safely test configurations without impacting other environments like Staging or Production.

02 Design and run your first pipeline

- Create a basic pipeline that integrates with a data source and writes to a data target.
- Use the drag-and-drop Designer to build your workflow and explore pipeline components such as transformations, assertions, and scheduling.
- Run the pipeline in your development environment and review the logs to understand how execution details are displayed.

03 Explore key features

Familiarise yourself with critical features, including:

- Version Control Integration
  Experiment with committing and pushing your pipeline to the Matillion Hosted Version control system or linking it to your Git provider.
- **Environment Variables and Secrets Management**
  Learn how to manage sensitive information and override variables for different environments.
- Pipeline Scheduling
  Use the scheduling feature to automate pipeline runs, testing how incremental loads are handled.

Production-ready deployment

See the Official Data Productivity Cloud Documentation for a streamlined guide to project setup.

Best Practices for Environment Setup and Naming in the Data Productivity Cloud

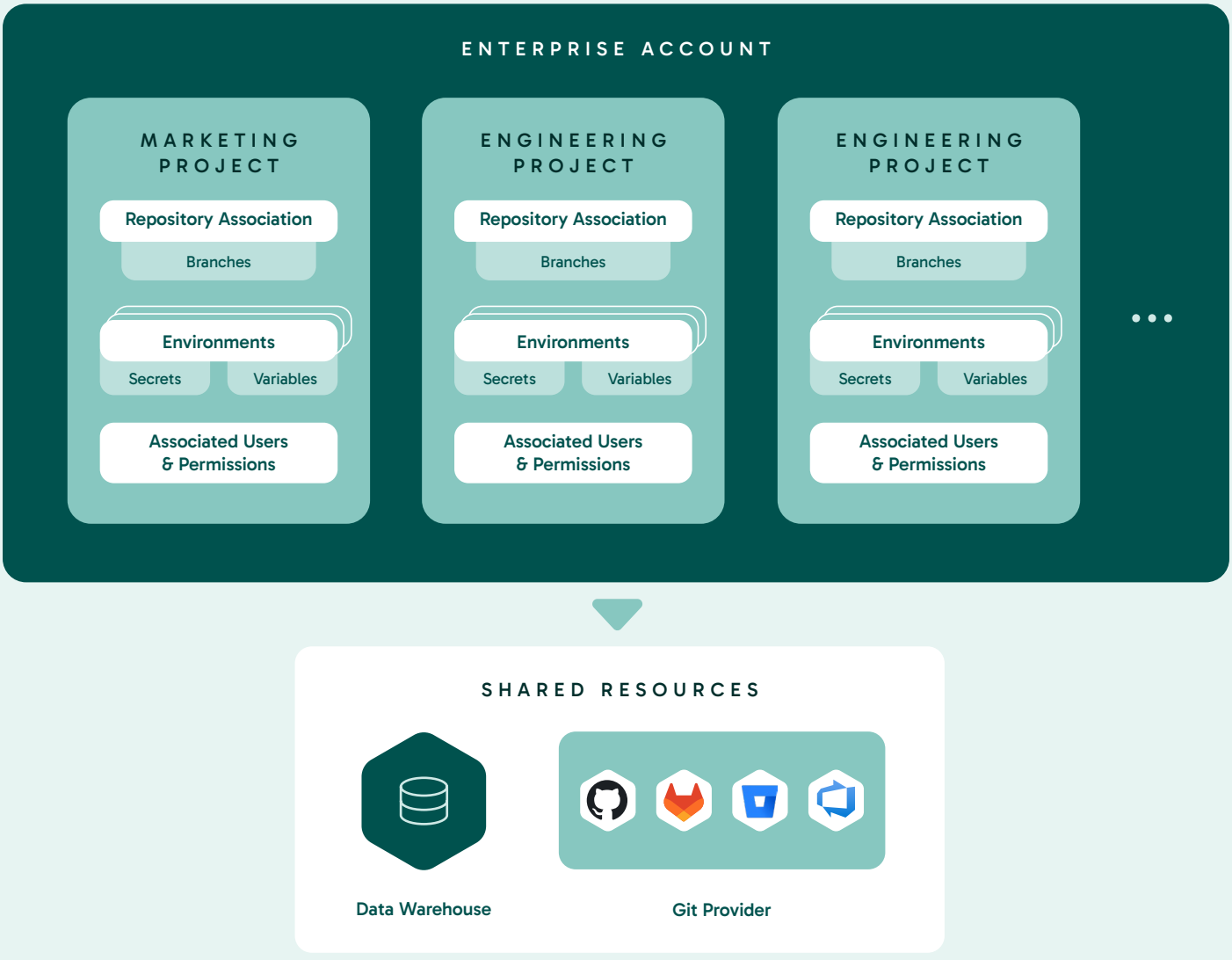
For organisations planning a production-ready deployment of the Data Productivity Cloud, it's important to establish a clear and consistent approach to environment setup and naming. These best practices ensure that your workflows are scalable, maintainable, and aligned with industry standards for managing data pipelines across multiple environments.

If you're new to the Data Productivity Cloud or are still in the experimentation phase, we recommend starting with the Quick-Start Guide to familiarize yourself with the platform. The guide walks you through a few foundational tasks, such as creating pipelines, managing variables, and scheduling runs, providing the context you need before embarking on a large-scale deployment.

For those already familiar with the Data Productivity Cloud, this section outlines a recommended setup to maximize efficiency and organisation in complex deployments.

Reference architecture

Enterprise-Level DataOps Architecture



In large organisations, effective DataOps often requires multiple teams and departments to operate independently while adhering to the same, shared enterprise patterns and standards. The Data Productivity Cloud supports this by enabling enterprises to manage multiple projects, each aligned to specific business units or functional areas. This structure ensures:

Scalability

Departments can design their own data pipelines to align with their goals while maintaining easily manageable and performant projects.

Decentralized Teams

Different departments manage independent projects but share the same data warehouse and adhere to version control standards.

Collaboration

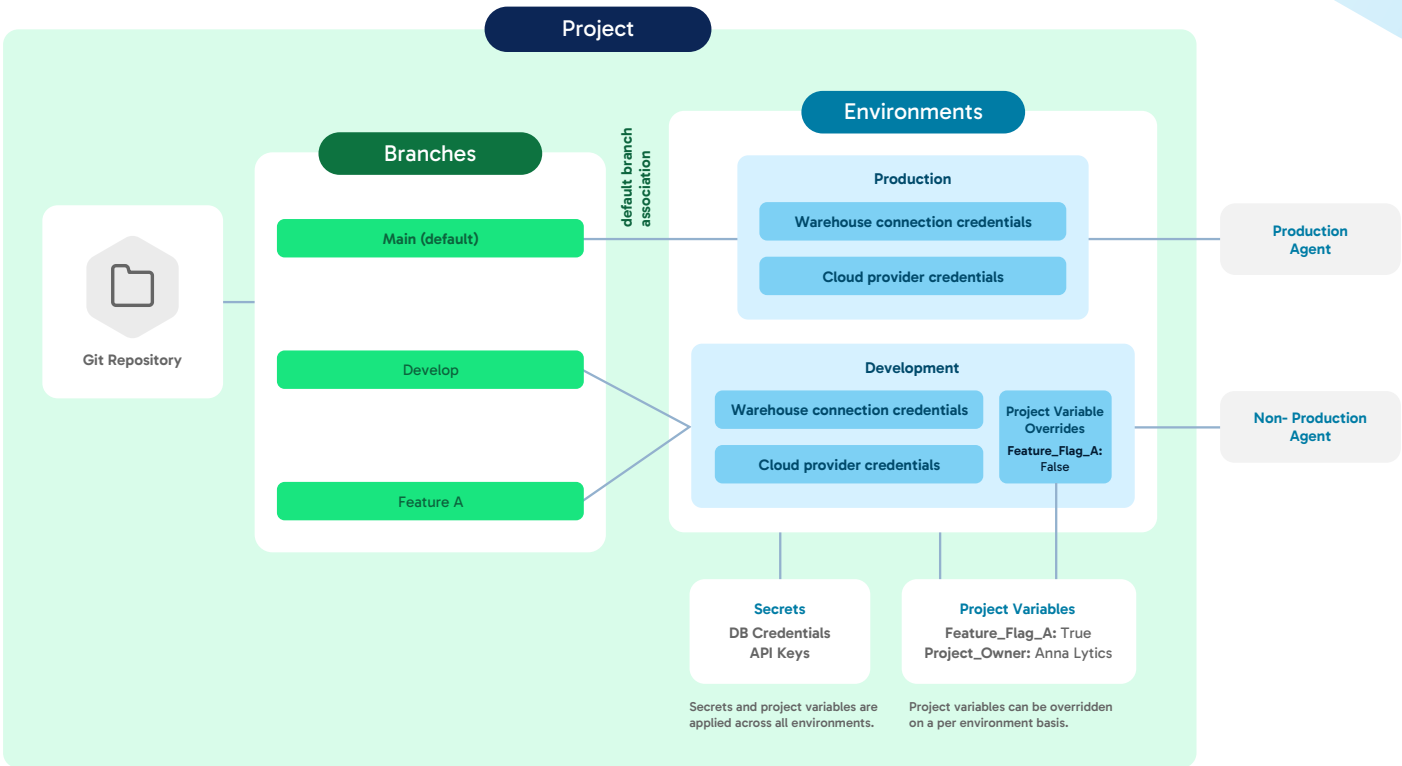
Teams operate independently but use shared resources to enable ease of cross-department learning.

Compliance

Enterprise accounts provide a unified view of usage, governance and cost management across the organization.



Project-Level DataOps architecture



Recommended environment setup

Development (Dev) environment

- **Purpose**
  The Dev environment is used for building, testing, and iterating on pipelines during active development.
- **Naming Convention**
  dev-<project-name>
- **Example**
  dev-sales-data-project
- **Configuration**
  Enable detailed logging and verbose error messages for easier troubleshooting. Use lower-cost or lower-performance data warehouses and resources to minimise costs during development.
- **Agent Assignment**
  Use the Non-Production Agent for executing pipelines in the Dev environment. This agent is responsible for managing pipeline execution across all non-production environments, ensuring scalability without affecting production resources.

  - Non-Production Agents typically have no access to Production data and are often smaller or less expensive as a cost-saving measure.
- **Access Control**
  Development environments should have broader access for teams, enabling collaboration while protecting sensitive data. Use role-based access control to ensure that access aligns with the least-privilege principle.

Production (Prod) environment

- **Purpose**
  The Production environment is where pipelines are fully deployed and run on live data.
  Only stable and thoroughly tested pipelines should be deployed here.
- **Naming Convention**
  prod-<project-name>
- **Example**
  prod-sales-data-project
- **Configuration**
  Use high-performance, highly-available data warehouses. Enable strict monitoring and alerting to track pipeline performance and detect potential issues immediately.
- **Agent Assignment**
  Assign the **Production Agent** for managing pipeline execution in the Production environment. This agent is isolated from non-production agents to ensure maximum reliability, availability, and security for live data processing.
- **Access Control**
  Limit access to the Production environment strictly to admins, ensuring critical systems are secure. Non-admin users should have read-only access or no access if stricter security is required.

Intermediate environments (e.g. Staging, Perf, UAT, Pre-Prod):

- **Purpose**
  Intermediate environments are used for testing and validation of pipelines before deployment to Production. These environments can serve distinct purposes such as User Acceptance Testing, validating pipelines in a Production-like environment or performance testing.
- **Naming Convention**
  Use a name that reflects the purpose of the environment (e.g. Stage, UAT, Pre-Prod, etc.)
- **Configuration**
  Configure each intermediate environment to match relevant aspects of the Production environment as needed (e.g. Database schemas, data volume, integrations). This may include specific configuration for performance, integration or user acceptance testing.
- **Agent Assignment**
  Use a Non-Production Agent for executing pipelines in these environments. This ensures separation of workloads between Production and other environments.
- **Access Control**
  Intermediate environments typically have the same or similar access policies to Development environments. They will differ based on the purpose of each environment.

General best practices

Environment naming

Use a clear and consistent naming convention (as outlined above) to distinguish between different stages of your pipeline lifecycle.

Configuration consistency

Ensure that configurations between Staging and Production environments are as similar as possible to avoid unexpected behaviour during deployment.

Security

Use environment-specific secrets, user access and credentials for access to different databases or cloud providers, and avoid hardcoding sensitive information in pipeline configurations



Integrating with a Git provider

Integrating your project with a Git provider offers significant benefits that enhance collaboration, version control, and automation in the DataOps lifecycle. By linking the Data Productivity Cloud with a Git provider, you can leverage a robust ecosystem of tools and workflows that improve efficiency, traceability, and team alignment. This integration enables teams to track changes, collaborate on pipeline development, and implement continuous integration and deployment (CI/CD) processes.

Key benefits of third-party Git integration include:

- **Collaboration**
  Unlock features like Pull Requests for reviewing and approving changes, ensuring high-quality pipelines and fostering collaboration among team members.
- **Automation**
  Seamlessly integrate with CI/CD tools such as GitHub Actions or GitLab CI/CD, enabling automated testing, validation, and deployment of pipelines.

It is strongly recommended to integrate the Data Productivity Cloud with a Git Provider of your choice. Currently supported integrations include GitHub, GitLab, Azure DevOps, and Atlassian Bitbucket. By linking your project with one of these providers, you can fully leverage version control, collaboration, and CI/CD workflows to streamline your DataOps processes.

For more detailed instructions, see our documentation on how to connect your project to your Git provider of choice.

Inviting collaborators

Collaboration is crucial to DataOps, and inviting other team members to participate in a project allows for shared development, reviews, and deployments. See our documentation for a detailed guide to adding collaborators.

When inviting collaborators to a project, it's essential to assign the appropriate level of access based on each team member's role and responsibilities. Additionally, ensure that all collaborators have been added to the supporting Git repository within your Git Provider to enable seamless version control and collaboration.

Best practices for assigning roles

Project Access

There are three levels of access that can be granted within the project: **Owner, Contributor, and Viewer.** It's important to assign these roles carefully to maintain project security and efficiency while enabling collaboration. See our documentation for more detail on the specifics of Project Access

Environment Access

There are four access levels for Designer environments: **Owner, Viewer, Runner** and **None.** You can set one of these access levels as the default for a given environment. This access level is then granted to all existing and future users who join a project until an Project Owner edits a user's access level. See our documentation for more detail on the specifics of Environment Access.

Owner role

01

Who should have Owner-level access?

- Team leads, senior engineers, or project managers responsible for managing the overall project configuration.
- Team members responsible for managing environments.

Responsibilities

- Project Owners are responsible for managing the entire project. This includes user management, environment management, setting project defaults, configuring agents, and adjusting version control settings.
- Environment Owners are responsible for the management of specific environments. This includes managing team members' ability to validate, sample or run pipelines, and managing schedules and artifacts for a specific environment.

Best practices

- Restrict Owner access to a small number of trusted individuals to minimise risk of unintended changes.
- Ensure that Project Owners are also added as admins in the associated Git repository (if you are bringing your own Git Provider) to enable full integration between the project and the version control system.

Contributor role

02

Who should have Contributor-level access?

- Data engineers, developers, or analysts who are actively involved in developing, testing, or modifying data pipelines.

Responsibilities

- Contributors are responsible for developing, modifying, and testing pipelines within development and staging environments.
- Contributors are responsible for pushing changes to the associated Git repository, ensuring pipelines are reviewed before deployment.
- Contributors cannot invite collaborators or manage project-wide settings but have the ability to trigger pipeline execution and monitor performance.

Best practices

- Contributors should have their corresponding Git Provider user added to the associated repository (if you are bringing your own Git Provider), allowing them to commit, push, pull, and merge changes as part of the CI/CD workflow.
- Contributors should be encouraged to follow best practices for branching strategies, pull requests, and testing in non-production environments before merging to the Production associated branch.

Runner role (Environments only)

03

Who should have Runner-level access?

- Team members responsible for executing and monitoring data pipelines.
- Quality engineers or analysts who need to sample, validate or schedule pipelines but do not require the ability to edit pipelines.

Responsibilities

- Runners can trigger pipeline runs and monitor pipeline executions. This allows them to view if jobs run successfully and report any issues to Project Owners.
- Runners can also manage schedules but they are not able to Publish or Promote artifacts.

Best practices

- Limit Runner access to team members who need to execute pipelines but do not require full project configuration access.
- Ensure Runners understand pipeline dependencies and review execution logs to maintain pipeline reliability.

Viewer role

04

Who should have Viewer-level access?

- Stakeholders, data consumers, or junior team members who need visibility into pipeline execution and project status but do not need to modify pipelines.

Responsibilities

- Viewers can view, validate and sample pipelines, view branches, environments and secrets, but they cannot make changes to pipelines, configurations, or project settings.
- This role is suitable for team members who need visibility into the project's progress but are not directly involved in development or testing.

Best practices

- Viewers should have their access limited to project monitoring and reporting functions.
- You will also need to add Viewers to the associated Git repository (if you are bringing your own Git Provider) as a read-only user.

Additional considerations

Security and compliance
Always follow the principle of least privilege when assigning roles. Only grant Owner access to team members who are responsible for managing the project and environments.

Consistency across platforms
Ensure that roles in the project are aligned with roles in the Git repository to prevent access issues or miscommunication during development and deployment.

Periodic access reviews
Regularly review and update the roles of collaborators as project needs evolve, ensuring that team members have the appropriate level of access for their current responsibilities.

Final notes on project setup

Properly setting up your project with well-defined environments, consistent naming conventions, collaboration roles, and Git integration is critical to ensuring a smooth DataOps workflow. By following these best practices and guides, you can create a strong foundation for managing and scaling your data pipelines effectively.

End of Data Productivity Cloud Specific Section

03

Design Phase

The Design Phase is the foundation of the DataOps development lifecycle. This is where we build robust, scalable, and maintainable data pipelines. This phase involves planning, designing, and constructing the data pipelines. By implementing strong version control practices, teams can collaborate effectively, manage changes efficiently, and ensure the quality of their data pipelines throughout the development process.

Principles

In the Design Phase, several core principles guide the development of good data pipelines:

01 Version control as the backbone

Implementing version control is crucial for managing changes, enabling collaboration, and ensuring the integrity of pipelines. Every change should be tracked, reviewed, and versioned. This topic is explored in detail below.

02 Incremental and iterative development

Adopt an iterative approach to building data pipelines, allowing for continuous improvement. Start with a minimum viable pipeline and incrementally add features, optimising and refining the pipeline with each iteration.

Make changes small and atomic

- Whenever possible, break down changes into the smallest possible units and make them atomic. An atomic change is one that is self-contained, addressing a single aspect of the pipeline, such as a specific bug fix, a minor feature addition, or a change to a particular component.
- By making changes small and atomic, you reduce the complexity of each update, making it easier to test, review, and understand. This approach minimizes the risk of introducing bugs and makes it simpler to roll back a specific change if needed without affecting other parts of the pipeline.
- Additionally, small atomic changes improve the efficiency of your version control system, as each change can be tracked, documented, and merged independently, ensuring a clear and precise history of modifications.

03 Decoupling deployments and releases

Utilize feature flags to decouple deployments from releases. This approach allows teams to deploy new code into the production environment without immediately activating new features. Features can then be toggled on or off independently of the deployment process, enabling a controlled and safe release. This strategy reduces risk, enhances flexibility, and allows for gradual rollouts and quick reversions if necessary. Feature Flags are covered in-depth below.

04 Isolation and collaboration

- Design your workflows so that team members can work on isolated branches without interfering with each other's work. This promotes parallel development and reduces the risk of conflicts.
- A good way to guarantee this is to break up work in such a way that no two people are working on the same pipeline at the same time. This reduces the chances of engineers' changes conflicting with each other.
- If two or more engineers need to work on the same pipeline at the same time, consider pairing-up on the work so that only one set of changes is being committed at a time.



05 Consistency and standardisation

Standardise the structure and naming conventions of your pipelines and pipeline components. This ensures that all team members can easily understand and navigate the codebase, reducing errors and improving maintainability.

An example of a good naming convention for data pipelines

[System]_[Function]

- **System**
  The name of the system, table or application the pipeline is associated with. This could be the data source, like “CRM,”or “WebAnalytics“, or a table in your DataWarehouse like “User“ or “Account“.

- **Function**
  Describes the primary function or purpose of the pipeline. This could be something like “DeleteCustomerData”, “UpdateSalesReporting”, or “LoadUserActivity”.

Examples

- **SF_LoadSalesReporting**
  An orchestration pipeline that loads sales reporting data from the SalesForce system.

- **Snowflake-Account_UpdateCustomerData**
  A pipeline that transforms Customer data on the Account table in your Data Warehouse.

Explanation

- **Consistency**
  This naming convention ensures that every pipeline name follows a clear, logical structure, making it easier to identify the purpose and context of the pipeline at a glance.

- **Clarity**
  By including the system, function and environment, the naming convention provides all necessary information in a concise format, aiding in navigation, management, and troubleshooting.

- **Scalability**
  This structure scales well as more pipelines are added, ensuring that names remain unique and descriptive, even in large projects with numerous pipelines.

A note on Folders
This naming convention can also be extended to folders, allowing for further organisation and standardisation of Data Productivity Cloud Projects. Consistent folder naming ensures that related pipelines are grouped logically, enhancing project structure and clarity.

06 Automation and continuous integration

Automate as much of the design and build process as possible, including testing, code quality checks, and deployments. Automation reduces manual errors, speeds up the development process, and ensures that consistent practices are followed across the team.

Continuous Integration (CI) should be integrated from the beginning to catch issues early in the development cycle. By continuously integrating code into a shared repository, teams can detect and address errors more quickly, leading to more reliable and stable pipelines.

More on this in the “Test Phase”
As you move into the Test Phase, automation and CI will play an even more critical role. We’ll look more closely at how automated testing and continuous integration contribute to maintaining the quality and performance of your data pipelines.

Tools and methodologies

To effectively design and build data pipelines within a DataOps framework, the following tools and methodologies are essential:

01 Version Control Systems (VCS)

Git is the backbone of version control in the Data Productivity Cloud. It enables teams to manage changes, collaborate across branches, and maintain a history of all modifications. With Git, every change to the pipeline can be tracked, reviewed, and rolled back if necessary.

02 Branching strategies

Feature branching
This strategy allows developers to work on individual features in isolation from the main branch. Each feature is developed in its own branch, ensuring that changes do not affect the stability of the main branch until they are fully tested and ready to be merged.

Trunk-based development
In contrast to Feature Branching, Trunk-Based Development encourages frequent integration of small, incremental changes into the main branch (often referred to as "trunk"). This approach helps to reduce merge conflicts and ensures that the main branch is always in a deployable state. Trunk-Based Development also provides a fixed approach for managing releases and hotfixes, which we'll explore further in the section titled "Release Phase".

More on this in the "Collaboration and Version Control" section below.

03 A pipeline design tool

Matillion Data Productivity Cloud
The Data Productivity Cloud's integrated environment supports the design and construction of data pipelines, with version control tightly integrated. It provides a visual interface that simplifies the creation of complex data workflows while ensuring they are stored and managed within Git.

04 Feature flagging

Feature flagging
This allows teams to control the activation of features independently from deployment. By using Project Variables and If Components, teams can safely introduce, test, and manage new features in their data pipelines.

More on this in the "Feature Flagging" section below.

05 Git providers

GitHub, Azure DevOps, GitLab & Bitbucket
These platforms not only host your Git repositories but also provide features like pull requests (PRs), issue tracking, and continuous integration pipelines, which are integral to the collaborative aspect of the Design Phase.

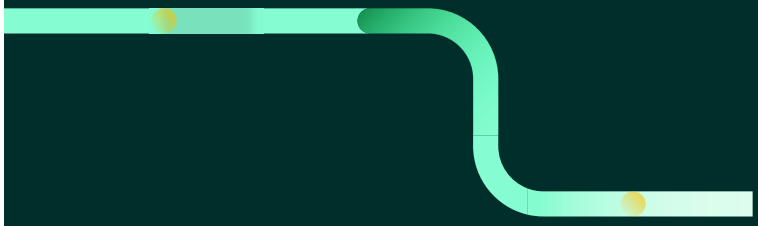
06 Infrastructure as Code (IaC)

AWS CloudFormation Templates
Use CloudFormation to define AWS resources for your data pipelines, including compute instances, storage, and networking configurations. CloudFormation templates are declarative and integrate seamlessly with AWS services, enabling robust and scalable infrastructure management. **AWS CloudFormation Templates for agents can be found here.**

Azure Resource Manager (ARM) Templates
For Azure-based workflows, ARM templates provide a declarative syntax to define and deploy resources, such as storage accounts, virtual networks, and database instances, that support your data pipelines. **ARM Templates for agents can be found here.**

Collaboration and version control

In the Design Phase, effective collaboration and version control are crucial for maintaining the integrity and quality of your data pipelines. By versioning all artifacts (pipelines, scripts, configuration files) together, teams can maintain a history of all changes, roll back to previous versions if necessary, and guarantee that all files within a version are consistent.

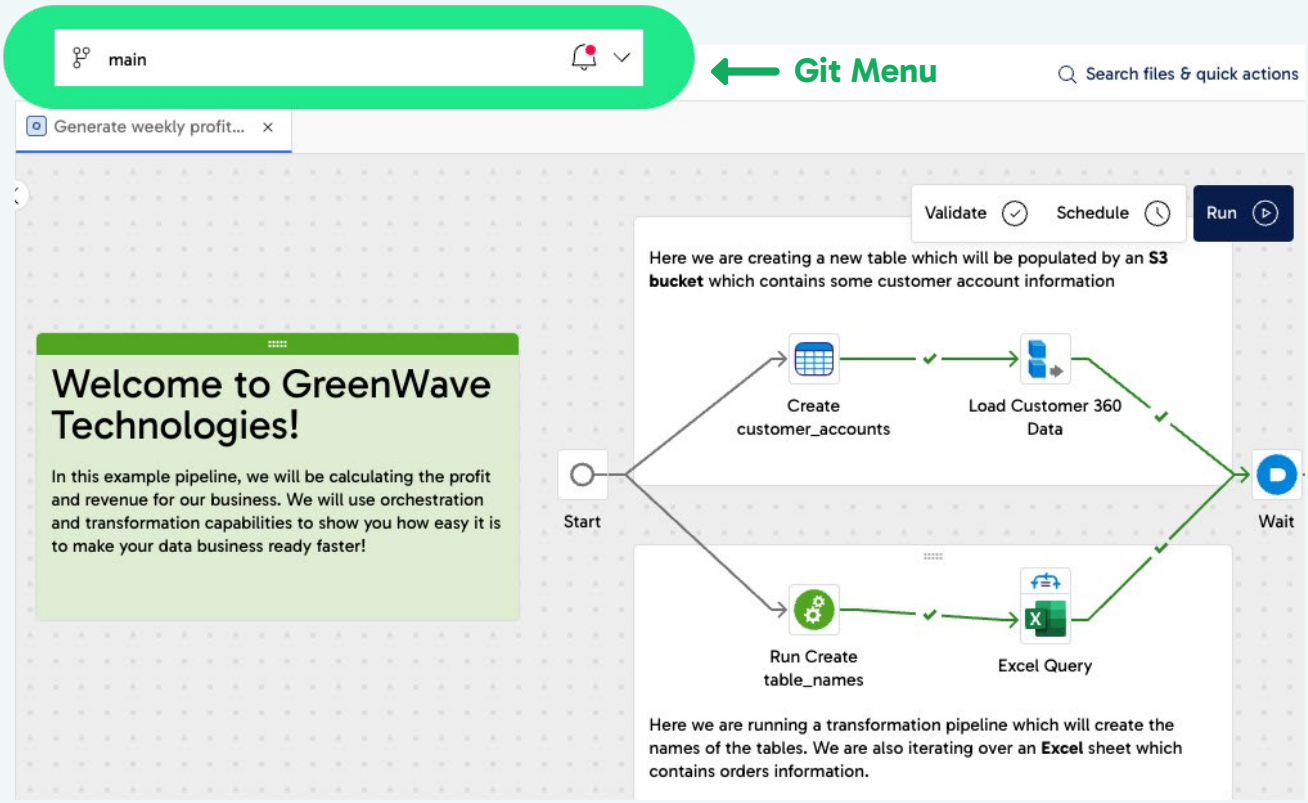


Git actions available through designer

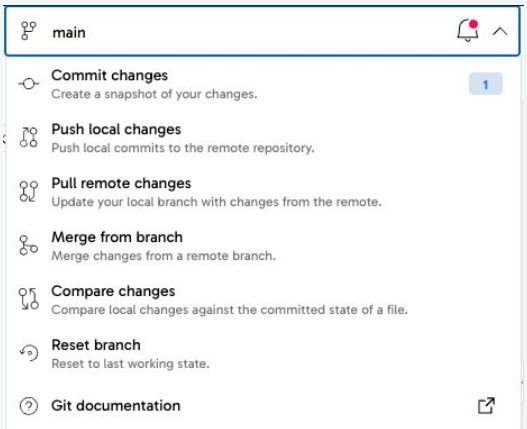
In order to facilitate seamless collaboration and version control within the DataOps lifecycle, the product offers an integrated Git menu that allows Data Engineers to manage their data pipelines efficiently. By using the Git menu, users can commit changes, push local updates, pull changes from remote repositories, merge branches, and reset branches directly from within the product.

The Git menu is easily accessible within the Data Productivity Cloud Designer, providing an intuitive interface for managing version control actions without needing to switch to external Git tools. Below is an overview of the available Git actions and how they can be used.

Git menu in the context of the Data Productivity Cloud Designer



Opened Git Menu



Commit changes

The **Commit Changes** action allows users to save their progress by creating a snapshot of their current work. When committing changes, the user provides a descriptive message explaining what has been modified, making it easier to track the evolution of the pipeline. Commits are stored in the local repository and can be reverted to later if needed.

Best practice

Commit frequently with clear messages, ensuring that each commit represents a small, logical unit of work. This makes it easier to track progress and roll back if necessary.

Recommended use

01 After making modifications to the pipeline, select **Commit Changes** from the Git menu.

02 Try generating a commit message using Copilot or enter your own message to describe the changes made.

03 Click “Commit” to save the changes locally.

Push local changes

The **Push Local Changes** action sends all committed changes from the local repository to the remote repository, making them available to other collaborators. Pushing ensures that the team has access to the most up-to-date version of the pipeline.

Best practice

Push regularly to keep the remote repository up-to-date and avoid conflicts. Always pull the latest changes from the remote before pushing to ensure your changes are based on the most current version.

How to use

01 After committing changes, select Push Local Changes from the Git menu

02 The system will push all local commits to the remote repository, synchronizing your work with the rest of the team.

Pull remote changes

The **Pull Remote Changes** action updates your local repository with the latest changes from the remote repository. This ensures that you are working with the most recent version of the project, including any changes made by other team members.

Best practice

Pull remote changes regularly, especially before starting new work, to avoid conflicts and ensure you are working with the most current version of the pipeline.

How to use

01 Select Pull Remote Changes from the Git menu

02 The system will fetch the latest changes from the remote repository and merge them into your local branch.

Merge from branch

(Merging changes from a remote branch)

The **Merge from Branch** action allows you to integrate changes from a different remote branch into your current working branch. This is typically used when you want to bring updates from another branch, such as a feature branch or the main branch, into your current branch.

Note: You will only merge changes that have been pushed to remote on the branch you choose to merge from so make sure you, or your colleagues, have pushed any changes on the relevant branch before attempting a merge.reverted to later if needed.

Best practice

Before merging, ensure that your current branch is up-to-date by pulling any remote changes. Use this action to integrate changes from feature or development branches into your main branch once they have been reviewed and tested.

How to use

01 Select Merge from Branch from the Git menu.

02 Choose the branch you want to merge from.

03 The system will merge the changes from the selected branch into your current branch.

Reset branch

The **Reset branch** action allows you to go back to a previous working commit. This is useful when you need to undo recent changes that have not yet been committed or pushed, effectively returning the branch to a previous “clean” state. There are two available options:

Reset to latest local commit: This will discard all your uncommitted changes in your local branch. Any local commits that have not been pushed to remote will not be affected.

Reset to latest remote commit: This will discard all your uncommitted changes and local commits that have not been pushed to remote. It will also pull the latest commits from remote.

Best practice

Use reset cautiously, as it permanently removes uncommitted changes or unpushed commits. It’s best to use this action when you are sure that the current changes are not needed or when recovering from an error in the working directory.

How to use

01 Select Reset branch from the Git menu.

02 Confirm that you want to reset the branch, as this will discard any uncommitted changes.

03 The system will revert your branch to the state of the last commit.

Compare changes

The **Compare changes** action provides a side-by-side view of differences between files in the working directory and their counterparts in the latest Git commit. This feature is particularly useful for reviewing changes before committing, ensuring that modifications are intentional and well-documented. Differences are displayed on a per-file basis, highlighting added, modified, or removed lines of code.

Best practice

Use Compare changes before committing to verify your edits and ensure that only intentional modifications are staged. This helps maintain a clean and traceable commit history, making it easier to identify and resolve issues during code reviews or testing.

How to use

01 Select **Compare changes** from the Git menu.

02 Review the list of files with changes. Clicking on a file will display a detailed comparison of its current state in the **Working Tree** versus its state in the latest Git commit.

03 Examine the highlighted differences to ensure your changes are accurate and complete.

04 Close the comparison view once your review is complete.



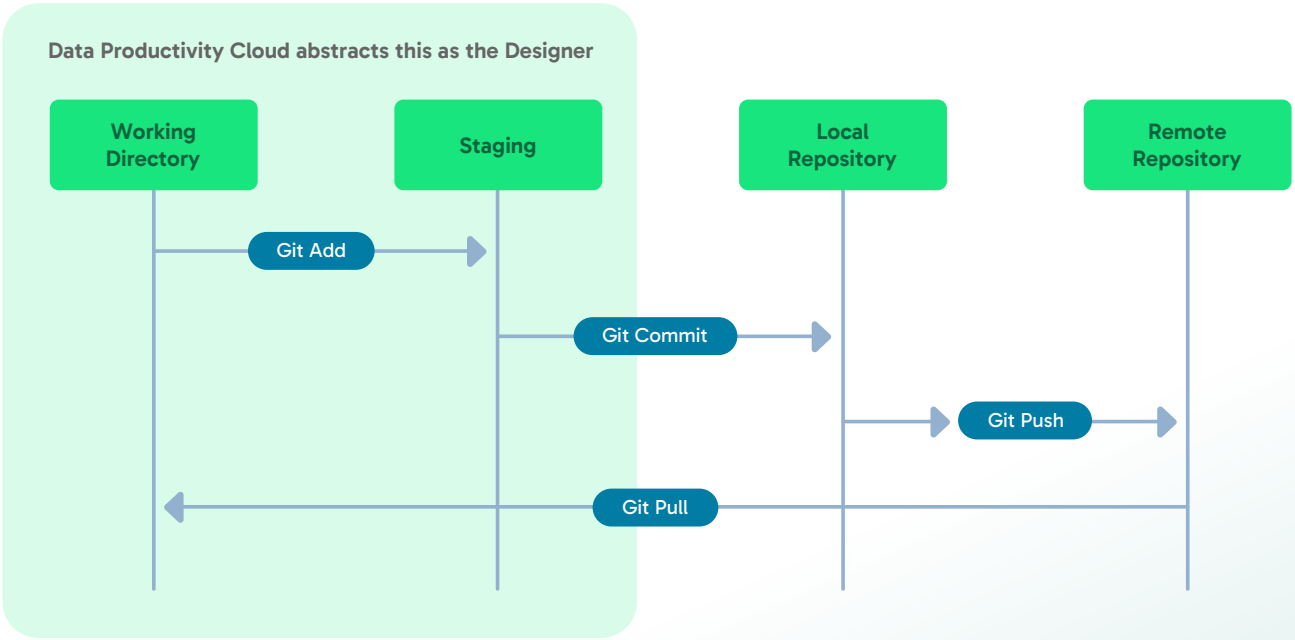
End of Data Productivity Cloud Specific Section

Git core concepts

What follows is a reworked version of the below blog post

Read here

What follows is a description of the concepts that will help a user better understand how the Git version control works in the Data Productivity Cloud.





Working directory

The **Working Directory** serves as a personal workspace where users can craft and refine their data pipelines independently of Git. In the Data Productivity Cloud, this corresponds to the Designer session within the web browser. Changes made in this environment are only visible to the individual user, allowing them to experiment and develop pipelines without affecting the shared codebase.

Staging and Git add

The **Staging** area plays a crucial role in organising and preparing changes before they are committed. When modifications are made in the Working Directory, Git allows users to selectively stage these changes by **adding** them to (or removing them from) the staging area. This step ensures that only the desired changes are included in the next commit to the local repository.

Note: In the Data Productivity Cloud, the "add" action is managed automatically. All changes made in the Working Directory are automatically staged, so manual use of the 'add' functionality is not required. Future iterations will include the ability to selectively stage files before committing, offering more control over which changes are included in each commit.

Local repository and Git commit

Once the desired changes have been staged, they can be committed to the **Local Repository. Committing** changes in Git is akin to saving a snapshot of your progress, with a commit message for easy reference. This commit is stored locally, allowing the user to track the evolution of the project and revert to previous states if necessary. The Local Repository is a valuable tool for maintaining a personal record of changes before they are shared with the wider team.

Remote repository and Git push

When a user is ready to share their changes with the team, these changes need to be pushed to the **Remote Repository. Pushing** commits to the remote repository makes them accessible to all team members, ensuring that everyone is working with the latest version of the project. This step is essential for collaboration, as it synchronises the work of individual users with the broader development effort.

Git pull

Pulling changes from the remote repository updates the local copy of the project, synchronising it with the latest version from the **remote repository**. This process is crucial for collaboration, as it ensures that all team members are working with the most up-to-date version of the codebase. In cases where changes made by others conflict with a user's local modifications, a **merge** will be required to reconcile these differences.

Merging

Merging in Git involves integrating changes from one branch into another, such as bringing updates from a feature branch into the main branch, reintegrating a release branch back into development, or incorporating changes from a collaborator's branch. **Merging** is essential for effective collaboration in Git, enabling teams to work on separate tasks independently and then combine their efforts into a unified project. This process ensures that the work done by different team members is seamlessly integrated, maintaining the cohesion of the overall codebase. More on merging in the **"Branching Strategies"** section below.

Quick reference table

Git Action	Description	Best Practices
Commit	A commit captures the current state of the working directory, saving changes to the local repository. Commits should be logical units of work that represent a complete, self-contained change.	• Write clear and concise commit messages that describe the changes made. • Commit frequently to capture your work incrementally, which allows you to easily roll back to a previous "good" state if needed. • Keeping commits small and focused makes it easier to manage and understand the history.
Push	Push updates the remote repository with commits made in the local repository, making them available to other collaborators.	• Push frequently to ensure that your changes are backed up and shared with the team. • Regular pushes also ensure that your work is safely stored in the remote repository, allowing you to restore your work if something goes wrong locally. • Before pushing, always pull the latest changes from the remote to avoid conflicts.
Pull	Pull updates the local repository with changes from the remote repository, incorporating any updates made by other collaborators.	• Regularly pull changes before starting new work to stay up-to-date with the latest developments. • Resolve conflicts immediately if they arise to keep your local repository in sync with the remote.
Merge	Merge combines changes from different branches, integrating them into a single branch. Merging is commonly used to bring feature branches into the main branch.	• Always merge changes from the main branch into your feature branch before merging your feature branch back into the main branch. • This practice helps to resolve conflicts early and ensures a smoother integration process. • Use pull requests (available through Git Providers) for merging to ensure that changes are reviewed before being integrated.
Reset	Reset undoes changes by moving the current branch to a previous state. This action can be used to remove commits from the history or to return to the most recent committed state.	• Always ensure that any reset is communicated with your team to avoid confusion. • Frequent commits can make it easier to reset to a "good" state when necessary.
Compare Changes	Compare changes allows you to compare the local changes in a file with the previous commit.	• Always use Compare Changes before using Commit to ensure the commit you create contains changes you expect it to. • Always use Compare Changes before using Reset to ensure you don't erase any work you want to keep.

For more advanced git interactions, we recommend connecting your Designer Projects with your own Git Provider. See our Documentation on **Installing your own Git Provider** for more information on this.

Branching strategies

There are two primary approaches to collaboration and version control we recommend in a DataOps environment: **Feature Branching** and **Trunk-Based Development.**

Feature branching

Feature Branching is a strategy that allows developers to work on individual features in isolation from the main branch. This approach is particularly useful in environments where multiple developers are working on different aspects of a project simultaneously. It's recommended to adopt this approach in teams where 3 or more engineers are working on the same project

Advantages of the feature branching workflow

- **Parallel development**
  This workflow enables parallel development, where multiple features can be developed at the same time without disrupting the main codebase or other ongoing work.
- **Maintaining branch integrity**
  By regularly pulling changes from the main branch into active feature branches, developers ensure that their work remains compatible with the latest codebase and minimizes merge conflicts.
- **Collaboration, code quality and review**
  Before merging any feature branch into the main branch, it is essential to perform code reviews via Pull Requests (PRs). This process ensures that all changes are thoroughly vetted and meet the team's standards.
- **Flexibility**
  Feature branching offers flexibility by allowing teams to experiment, iterate, and refine features in isolation, without risking the stability of the production code.
- **Isolation**
  Developers can work on features independently without worrying about the stability of the main branch. This reduces the risk of conflicts and makes it easier to manage large projects.

By following this workflow, teams can manage complex projects efficiently, reduce the risk of integration issues, and maintain a clean, stable main branch that is always ready for deployment.

Key processes in the feature branching workflow

01 Branching and merging

Each new feature, bug fix, or enhancement is developed in its own branch. This ensures that changes do not affect the stability of the main branch until they are fully tested and ready to be merged.

Developers can work independently on their branches, and once their work is complete, they can merge their changes back into the main branch. This process typically involves a code review to ensure the quality and integrity of the changes.

02 Pull requests for code review

Once a feature is complete and ready to be merged, a Pull Request (PR) is created. This serves as a formal request to merge the feature branch into the main branch and provides an opportunity for other team members to review the changes.

Note that in the Data Productivity Cloud, Pull Requests for code review are currently only available to those who have integrated their own Git provider with the Data Productivity Cloud. Teams using GitHub, Azure DevOps, or other supported providers should perform Pull Requests using their provider's UI.

The PR process helps catch bugs, enforce coding standards, and share knowledge among team members. It's a critical step in ensuring that only high-quality, vetted code makes its way into the main branch.

03 Commit practices

When working on a feature branch, developers should commit frequently and with clear, descriptive commit messages. Each commit should represent a logical unit of work, making it easier to track changes and identify issues.

Smaller, atomic commits are preferable as they allow for easier debugging and more granular tracking of changes.

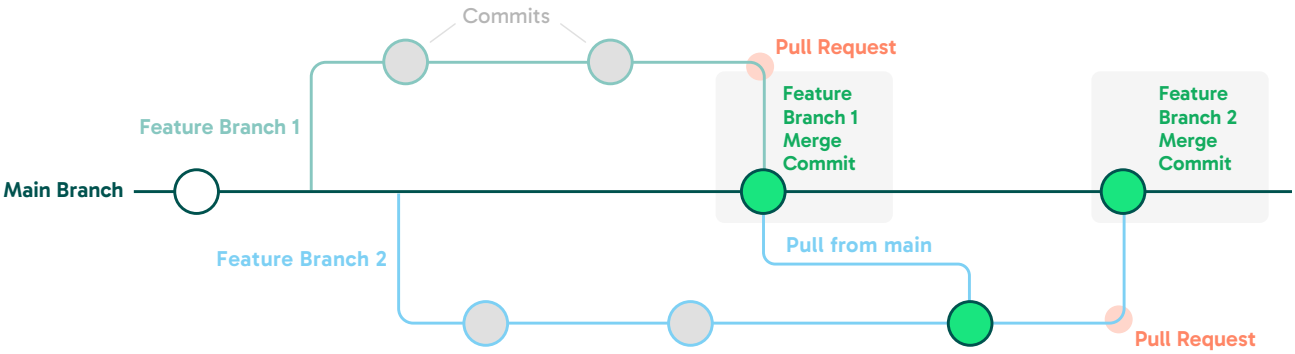
04 Keeping branches up-to-date

Regularly merge changes from the main branch into your feature branch to keep it up to date with the latest developments. This practice helps to avoid large merge conflicts when it's time to integrate your feature back into the main branch.

Developers should perform these updates frequently, especially in fast-moving projects where the main branch is regularly updated.

Example

This diagram illustrates the key steps and interactions within the Feature Branching workflow, including the management of multiple feature branches.



Explanation of the diagram

Main branch

The main horizontal line in the diagram represents the Main Branch, which is the central codebase where stable, production-ready code resides. All feature branches are eventually merged back into this branch once they are completed and thoroughly tested.

Feature branch 1

The first feature branch (Feature Branch 1) is created from the main branch and is represented by the first green line branching upwards. Developers work on this branch to implement a new feature, making regular commits to track their progress. Once the feature is complete a pull request is raised and, once the engineer has approvals, Feature Branch 1 is merged back into the main branch. This merge is marked by a green merge commit on the main branch and Feature Branch 1 is closed.

Feature branch 2

While Feature Branch 1 is still in progress, a second feature branch (Feature Branch 2) is created from the main branch. This branch allows another developer (or the same developer) to work on a separate feature simultaneously. Like Feature Branch 1, this branch has its own commits, and development proceeds independently of the first branch.

Merging and keeping up-to-date

Once Feature Branch 1 is merged into the main branch, the codebase in the main branch is updated with the new feature. However, Feature Branch 2 does not automatically receive these changes. To keep Feature Branch 2 up to date with the latest codebase, it is necessary to pull in the changes from the main branch.

This step is crucial to avoid conflicts and ensure that Feature Branch 2 is compatible with the latest version of the main branch. The diagram shows a line from the main branch back to Feature Branch 2, indicating this pull operation.

Final merge of feature branch 2

After updating Feature Branch 2 with the latest changes from the main branch and completing development, this branch is also merged back into the main branch with an approved Pull Request. The final merge commit is shown on the main branch, completing the integration of both features and Feature Branch 2 is closed.

Trunk-based development

Trunk-based development is a streamlined approach to version control that emphasises simplicity and speed by having all developers work directly on a single branch, often referred to as “trunk” or “main.” This workflow is particularly suited to smaller teams focused on continuous delivery and rapid iteration, where the goal is to minimise the complexity of branch management and accelerate the deployment of new features.

Advantages of the trunk-based workflow

Continuous integration and rapid delivery

Trunk-based development aligns closely with Continuous Integration (CI) and Continuous Delivery (CD) practices. By keeping all development on a single branch, teams can continuously integrate small, incremental changes, leading to more frequent and reliable releases.

Reduced merge conflicts

By minimising or eliminating long-lived feature branches, Trunk-based development reduces the likelihood of complex merge conflicts that can arise when integrating changes from multiple branches.

Simplicity and focus

The simplicity of working on a single branch reduces the overhead associated with managing multiple branches. This allows teams to focus more on writing and delivering quality code rather than on complex branching strategies.

Shared responsibility

With everyone working on the same branch, there is a shared responsibility among team members to ensure the t runk remains stable and deployable. This encourages a culture of discipline and collaboration.

Key processes in the trunk-based workflow

01 Frequent small commits

In Trunk-Based Development, developers make frequent, small commits directly to the trunk. These commits represent small, incremental changes that are easier to test, review, and integrate.

02 Feature flags

Should we provide a description of how to implement feature flags in a different section, using project variables and if components

- To manage features that are not yet ready for release, developers use feature flags. This allows them to commit incomplete or experimental features to the trunk without exposing them to end users.
- Feature Flags can be toggled on or off using project variables, enabling safe deployment of code that is still under development
- More on this in the Feature Flag section below.

03 Continuous integration

Every commit triggers a build and automated tests through a CI pipeline (read more about this in the Test Phase below). This ensures that any issues are caught early, maintaining the stability of the trunk. The goal is to always keep the trunk in a deployable state.

04 Short-lived branches (optional)

While Trunk-based development generally discourages long-lived branches, short-lived branches may still be used for tasks like hotfixes or experiments. These branches are merged back into the trunk as quickly as possible, often within a few hours or days.

05 Code reviews

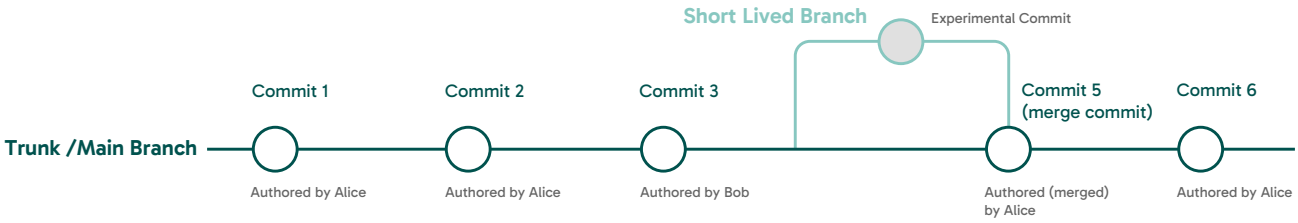
Code reviews are still an essential part of the process. Developers may use pre-commit reviews, where peers review changes before they are committed to the trunk, or post-commit reviews, where reviews happen after the commit but before deployment.

06 Automated deployment

With the trunk always in a deployable state, automated deployment processes can be triggered directly from the trunk (see Release Phase). This supports rapid, frequent releases, which is a key advantage of this workflow.

Example

This diagram illustrates the key steps and interactions within the Trunk-Based Development workflow.



Explanation of the diagram

This diagram reads left-to-right, with each circle representing a commit and each line representing a git branch.

Trunk/main branch

The central horizontal line in the diagram represents the Trunk/Main Branch, which serves as the single, continuous codebase where all development work is integrated. This branch is kept in a deployable state at all times, ensuring that the latest changes are always ready for production.

Frequent small commits

Along the trunk line, you'll see multiple circles representing frequent commits. In Trunk-Based Development, developers commit their changes directly to the trunk multiple times a day. These commits are small, incremental updates that are easier to integrate and test.

Each commit is a self-contained change, such as a bug fix, a small feature addition, or a refactor, which minimises the risk of introducing conflicts or errors.

Short-lived branches (optional)

The diagram also shows short-lived branches that are quickly merged back into the trunk. These branches are used sparingly, usually for urgent fixes or experimental changes that need to be isolated temporarily.

Once the work on a short-lived branch is complete, it is promptly merged back into the trunk to maintain the simplicity and flow of the Trunk-Based Development process.

Note that in our diagram, no commits happen between creating the Short-lived branch and merging it to main but this is not necessary. In-fact the creation of a short-lived branch is often to allow other work to continue while experimental changes are made safely on the isolated branch.

Merging and integration

The diagram illustrates that changes are pushed directly into the trunk without the need for long-lived feature branches. This continuous integration helps ensure that the trunk remains stable and free of integration issues, as changes are integrated and tested regularly.

Versioning of pipelines

Versioning of Pipelines ensures that all artifacts—including pipelines, Python files, and SQL scripts—are versioned together in a commit. By versioning all these elements simultaneously, and giving their versions descriptive names, users can maintain an accurate history of changes, revert to earlier versions if needed, and schedule pipelines with confidence. When a pipeline is selected for execution, all its supporting files (Python, SQL) are pulled from that specific version, ensuring consistency and reliability.

Versioning enables rollback to previous states and ensures that all artifacts are tightly coupled to their pipeline, eliminating the risk of discrepancies between code and supporting files.

Best Practice
Regularly commit changes, publish & version your pipelines after meaningful changes, and always include detailed commit messages that describe the context of the updates as these will be visible when selecting your pipeline version for scheduling.

Documentation in DataOps

Effective documentation is a cornerstone of successful DataOps practices, ensuring teams work collaboratively, onboard new members quickly and maintain long-term project stability. Documentation encompasses not just the pipelines themselves but also the design decisions, architecture and rationale behind decisions so that approaches can be replicated, creating a consistent engineering experience. By embedding documentation into everyday workflows, teams can create a comprehensive record that evolves alongside their pipelines.

Key practices

READMEs
Every project should contain a README file outlining its purpose, structure, key components and any setup instructions required to get team members up and running. READMEs provide a high-level overview and serve as the starting point for anyone unfamiliar with the project.

Notes
The Data Productivity Cloud Designer allows users to annotate their pipelines directly on the canvas. Notes can be used to document key design decisions, explain complex logic or provide context for other engineers coming to work on that area of your project.

The Data Productivity Cloud also has an AI companion which will generate notes for you based on selected components.

Commit messages
Each commit message is an opportunity to provide a clear and concise message to other engineers describing what changes were made and why. They act as a historical log of decisions and enable teams to trace the evolution of their pipelines.

Project management tools
Use project management tools to track tasks, feature requests and bug reports related to Data Pipelines. This practice helps document the rationale behind changes, aligns technical efforts with business objectives and creates a centralised audit trail to support compliance and promote accountability.

Benefits of strong documentation

Comprehensive, clear and concise documentation ensures pipelines are understandable, maintainable and the project in its entirety is scalable. Teams can onboard new members faster, minimise dependencies on tribal knowledge and reduce risk of errors stemming from misunderstandings. Proper documentation also ensures compliance with organisational standards and external regulations.

Best practice
Encourage team members to document as they work. Use the Notes feature in the Data Productivity Cloud Designer to annotate pipelines, include detailed commit messages for every change and maintain READMEs at the project level. Regularly review and update existing documentation to reflect the current state of the project, ensuring it remains accurate and valuable.

Environment-Specific Configuration

Environment-Specific Configuration allows teams to define and adjust pipeline behavior dynamically based on the environment in which the pipeline runs. This provides flexibility and control over how pipelines operate across Development, Staging, and Production environments. Using Project Variables and If Components, teams can implement environment-specific logic while maintaining a clear separation of concerns during publishing and deployment.



Motivation for Environment-Specific Configuration

01 Incremental rollouts

Teams can test features in lower environments (e.g. Development or Staging) before deploying them to Production. This ensures that new pipeline behavior is stable and reliable before being used with production data.

02 Controlled deployments

Dynamic configuration allows teams to adjust pipeline behavior in each environment without modifying or redeploying pipeline code. However, teams should exercise caution when using Project Variables to manage significant feature changes.

03 Minimised risk

By testing behaviour changes in isolated environments before releasing to Production, teams can reduce the risk of untested features affecting live workflows.

04 Complementing publishing and deployment

Environment-specific configurations should not replace the publishing and deployment workflow. Instead, they complement it by enabling minor adjustments to pipeline behavior in each environment. Major feature changes should always follow the publish and deploy process to ensure consistency and accountability.

05 Faster feedback loops

With Environment-Specific Configuration, new features can be tested in production with limited exposure, enabling teams to gather real-time feedback and iterate quickly based on real-world data and outcomes.

Implementing Environment-Specific Configuration in data pipelines

Environment-specific pipeline behavior is managed using **If Components** and **Project Variables**. However, it's crucial to follow best practices to minimize risks, especially when controlling critical functionality in Production

01 Using If components

If components are conditional logic blocks that dynamically adjust pipeline execution based on environment-specific criteria. These components can check Project Variables to determine which steps or logic to execute.

Example use case

If a new transformation is under development, an If Component can determine whether to run the new logic or the existing stable logic. For example:

- In Development, the pipeline runs the new transformation logic.
- In Production, it defaults to the stable transformation logic.

02 Controlling with Project Variables

Project Variables are key-value pairs defined at the project level, with values that can differ across environments (e.g. "new" in Development and "stable" in Production). However, Project Variables should be used with caution, particularly for Production pipelines.

Best practices and risk mitigation

01 Access control

Currently all users can manage Project Variables, therefore it's up to users to manage the modification of environment-specific settings. Changes to variables used to manage which components run should be discussed and the results of changing them monitored closely to prevent unintended behaviour.

02 Auditability

Currently, changes to Project Variables are not tracked or logged. For critical pipelines, avoid using Environment-Specific Project Variables to control flow of components unless you have an external process to document and approve changes. This ensures accountability and reduces the risk of unintentional modifications

03 Approvals

Significant pipeline behavior changes should follow a formal approval process, such as pull requests or manual reviews, before being activated in Production. Avoid bypassing the publishing and deployment workflow for major changes.

04 Complementing publishing and deployment

Use Project Variables to fine-tune existing pipelines for different environments (e.g. toggling non-critical debugging features in Development). For major changes, follow the publish and deploy process to maintain consistency across environments.

05 Documentation

Maintain clear documentation of how Project Variables are used in your pipelines, including the logic controlled by each variable and the potential impact of value changes. Ensure all team members understand their purpose and limitations.

Dynamic pipeline configuration in practice

By leveraging **Project Variables** and **If Components**, pipelines can adapt dynamically across environments without requiring redeployment. However, this capability should complement—not replace—the **publish and deploy** workflow. For critical changes, rely on publishing and promoting artifacts to Production to ensure traceability and accountability.

When used responsibly, Environment-Specific Configuration empowers teams to maintain flexible and controlled pipelines while minimising risks to Production.

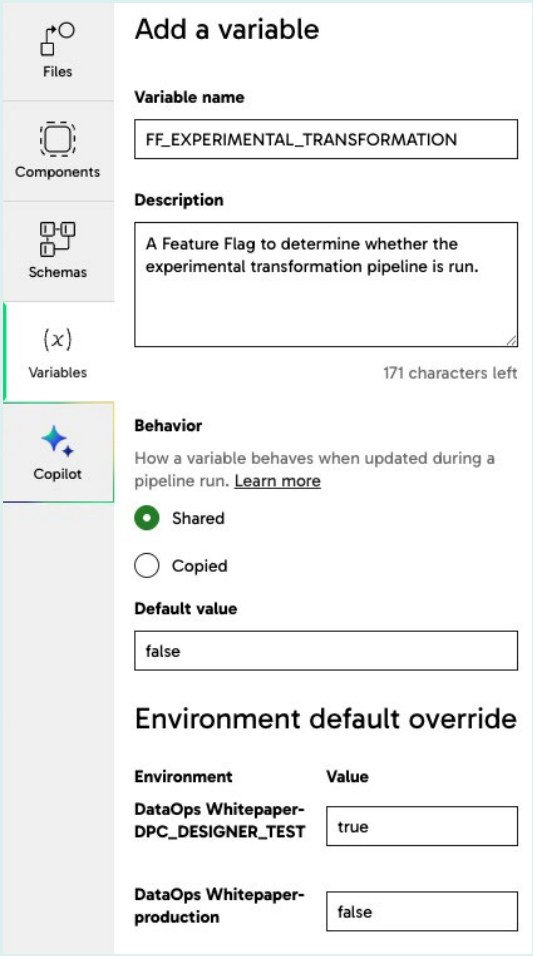
Example use case: conditional execution in a data pipeline

Environment-Specific Configuration allows teams to dynamically adjust pipeline behaviour based on the environment. Consider a scenario where a team is developing a new method for cleaning customer data in a pipeline. This method is still under development and testing, so it should only be enabled in the Development environment, while the Production environment continues to use a stable, well-tested method. Here's how this can be implemented using **Project Variables** and **If Components:**

Step 1

Setting the Project Variable

The team creates a Project Variable named TRANSFORMATION_MODE and sets it to "true" in the Development environment and "false" in the Production environment (alternatively these values could be set to something such as "experimental" and "stable" if this standard helps with clarity of purpose):



This variable acts as the control point, enabling the pipeline to dynamically determine which transformation logic to execute based on the environment.

Step 2

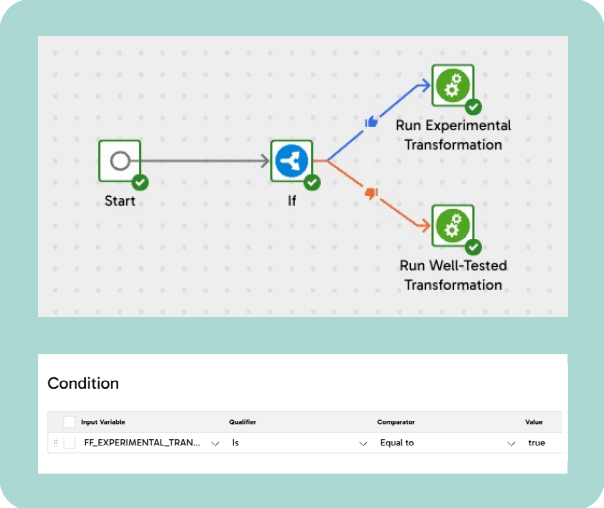
Configuring the If Component

An **If Component** named Transformation Logic Selector is added to the pipeline at the point where the transformation logic is executed. This component checks the value of the TRANSFORMATION_MODE Project Variable.

- If the variable is “true”, the pipeline executes the **Experimental Transformation** logic.
- If the variable is “false”, the pipeline defaults to the **Well-Tested Transformation** logic.

For example

- In the **Development environment**, the pipeline runs the Experimental Transformation for testing and validation.
- In the **Production environment**, the pipeline uses the Well-Tested Transformation to ensure reliability.



Step 3

Updating the configuration for promotion

Once the Experimental Transformation has been thoroughly tested and validated in the Development and Staging environments, the team updates the Project Variable in the Production environment to “true”. This activates the new transformation logic in Production without requiring any changes to the pipeline code or redeployment.

Step 4

Safe rollout and quick reversion

If issues are detected in Production after the new transformation logic is activated, the team can quickly update the TRANSFORMATION_MODE variable in Production back to “false”. This reverts the pipeline to the Well-Tested Transformation with minimal disruption, avoiding the need for a rollback or redeployment.



Best Practices for Environment-Specific Configuration

01 Limit temporary configurations

Environment-Specific Configurations that control significant pipeline behaviour should be treated as temporary where possible. Once a new feature or transformation is fully rolled out and stable, any associated conditional logic or configuration should be cleaned up to maintain a tidy and maintainable pipeline.

02 Clearly document configurations

Maintain clear documentation for all environment-specific configurations, including their purpose, associated Project Variables, and expected lifecycle. Use the Project Variable Description field or pipeline Notes to record this information. Proper documentation helps the team manage and review configurations effectively, ensuring alignment and avoiding misconfiguration.

03 Test all scenarios

Ensure that all possible paths controlled by environment-specific configurations (e.g. conditions in If Components) are thoroughly tested. This includes testing scenarios across Development, Staging, and Production environments to prevent unexpected behavior when promoting changes.

04 Use for controlled rollouts

Environment-specific configurations are ideal for testing new or experimental features in lower environments before rolling them out to Production. They allow teams to validate behavior, gather feedback, and make adjustments before committing to Production deployment.

05 Promotion between environments

Pipelines should be built and tested in lower environments (e.g. Development and Staging) before deployment to Production. Following the “build once, deploy anywhere” principle ensures that the same version of the pipeline is used consistently across all environments. Once a pipeline and its artifacts are published, they are immutable—guaranteeing that no changes can occur between environments, and that Production execution will match what was tested in Staging.

06 Best practice

Handle environment-specific configurations (e.g. variables, secrets, credentials) using Project Variables or environment overrides. This approach maintains the immutability of the pipeline code while allowing for dynamic behavior tailored to each environment.



End of Data Productivity Cloud Specific Section

Summary of best practices for designing pipelines

To ensure the Design Phase is executed effectively, adhere to the following best practices:

01 Adopt an appropriate branch strategy

For teams of 3 or more, encourage a multi-branch strategy where each team member works on a separate branch. Regularly merge changes from these branches into the main branch using PRs, and review them for quality and consistency.

- Keep the main branch up to date byfrequently merging feature branches after they pass code reviews

For smaller teams, consider adopting a trunk-based development strategy where frequent integration of changes into the main branch keeps the main branch stable and deployable.

02 Regularly review and refactor

Schedule regular code reviews to assess the design and implementation of the pipelines. Use PRs as a mechanism to review and discuss code changes before they are integrated into the main branch.

Establish a practice of refactoring as part of the review process, ensuring that the codebase remains clean, maintainable, and optimised. Refactoring should focus on improving performance, reducing technical debt, and enhancing readability.

03 Automate testing and deployment

Integrate automated tests that run with every commit to ensure that changes do not introduce errors or degrade performance. Automate deployment to ensure that changes can be quickly and reliably rolled out to production.

Use CI/CD pipelines to automate the entire build, test, and deployment process, reducing manual intervention and the risk of errors.

For an example of this, see **Example CI/CD Pipeline: GitHub Actions** in the "**Test Phase**" below.

04 Maintain a clear source of truth

Designate the main branch as the single source of truth for production-ready pipelines. Ensure that only thoroughly reviewed and tested code is merged into this branch.

Implement role-based access control to restrict who can merge into the main branch, further ensuring the integrity of the production codebase.

05 Encourage documentation

Promote the habit of documenting not only the code itself but also the design decisions, architecture, and rationale behind certain approaches. This can be done both using commit messages and Notes in the Data Productivity Cloud Designer. This documentation should be updated as the pipelines evolve, providing a valuable resource for current and future team members.

Users can leverage the Data Productivity Cloud's generative AI to annotate pipelines using notes. **See the documentation** for more details.

06 Risk reduction when making changes

Implement Environment-Specific Configuration to safely manage the rollout of new features. This approach enables quick rollbacks and minimises disruption, ensuring that pipeline changes can be activated or deactivated with minimal risk.

04

Test Phase

The Test Phase is a critical component of the DataOps development lifecycle, ensuring that data pipelines are reliable, secure, and performant before they are released into production. This phase involves rigorous testing across various dimensions, including functionality, performance, security, and compatibility, to validate that the pipeline meets all necessary requirements and can operate seamlessly in the target environment.

Principles

The Test Phase is guided by several core principles that ensure thorough validation and a smooth transition from development to production:

01 Early and continuous testing

Testing should begin as early as possible in the development lifecycle, often referred to as "shift-left" testing. This approach helps identify and resolve issues sooner, reducing the cost and complexity of fixes later in the process. Continuous testing should be integrated into every stage of development, with automated checks and tests running in CI pipelines to catch issues with every code change where possible.

- Note that currently CI pipelines are only available to those customers who integrate with a third-party git provider such as GitHub, Azure DevOps or GitLab.

02 Comprehensive coverage

Ensure that all aspects of the data pipeline are covered by test cases, including functional correctness, performance under various conditions, security compliance, and compatibility with existing systems. Test cases should be designed to validate both the typical use cases and edge cases that might expose vulnerabilities or inefficiencies.

03 Realistic lower environments

The lower environments such as Dev, and optionally UAT, Perf, Pre-Prod, etc. should mirror the production environment where appropriate in terms of data volume, structure, and external dependencies. This ensures that the tests provide meaningful insights into how the pipeline will perform in real-world scenarios.

- There are many ways to do this including taking a direct copy of the Product DB and loading it into your lower environments or generating new data using scripts.

- Generating new data would be preferable to taking a copy of Production in the case where you need to inspect the data directly but the Production source contains PI data.

Lower environments can be clones of Production, as long as the data is obfuscated in an appropriate way to protect Engineers from GDPR and other liability.

- Note that 2 logical environments can be on the same physical hardware for cost-saving purposes.

04 Tight feedback loop

Maintain a close feedback loop between the testing and design phases. Quick identification and resolution of issues during testing allow for iterative improvements and ensure that the pipeline is robust and reliable before deployment.

05 Risk-based testing

Prioritise testing efforts based on the potential impact of failure. Critical components and high-risk areas of the pipeline should receive the most rigorous testing to minimise the risk of costly issues in production.

Tools and methodologies

The Test Phase relies on a variety of tools and methodologies to ensure comprehensive testing and efficient issue resolution:

01 Pipeline validation tools

Use pipeline-specific validation tools that can verify the correctness of declarative pipelines such as yamlint. This is an area of development at Matillion and we expect a more robust set of tools to emerge in future versions of this paper.

02 Data quality assurance

Ensuring data quality is critical to the success of any data pipeline. Use pipeline components to validate that the data processed by the pipeline is in the expected shape and volume. By incorporating checks directly into your pipeline, you can automatically verify that the data meets your quality standards before it proceeds to the next stage. This proactive approach helps catch issues early and ensures that only clean, reliable data is used in downstream processes.

03 Performance testing

Performance testing is crucial to ensure that your pipelines can handle production-scale data volumes efficiently. It's important to set up test environments with datastores that contain high volumes of data, closely mirroring the conditions your pipelines will face in production. By testing with large datasets, you can assess how the pipeline performs under load, identify potential bottlenecks, and make adjustments to improve throughput and scalability. This approach helps ensure that your pipelines remain performant as data volumes grow over time.

04 Security testing

Incorporate security testing into the pipeline to validate that data is handled securely and that access controls are enforced. Custom scripts can be used to ensure compliance with data privacy regulations. This testing ensures that sensitive data is protected throughout the pipeline and that the pipeline meets all relevant security standards and regulatory requirements.



Publishing from feature branches for testing

Experimentation, validation and iteration are aspects that are key to the Test Phase, giving teams confidence before merging changes to the main branch. The ability to publish and schedule artifacts from any branch unlocks this flexibility.

Key benefits

Isolated testing and early issue detection:

- Teams can publish and schedule pipeline artifacts for execution in test environments directly from feature branches. This allows for targeted validation over time without affecting production pipelines.
- Running feature branch artifacts in lower environments enables earlier identification of performance bottlenecks, data quality issues, and security vulnerabilities before merging into the main branch.

Parallel development

- Multiple feature branch artifacts can exist simultaneously, making it easier for teams to test different versions of a pipeline concurrently without interference.

Best practice

Leverage feature branch publishing to test incremental changes in isolated environments. Once validated these artifacts can be promoted to higher environments for further testing before merging into the main branch.

Example workflow

- Developer creates a feature branch.
- Changes are committed and published to a test environment.
- Automated schedules run to validate functionality.
- Once tests pass, the branch is merged into main.



Summary of best practices for testing pipelines

01 Integrate testing early and continuously

Begin testing as early as possible in the development lifecycle and continue testing throughout. Incorporate automated testing into every stage using CI/CD pipelines to catch issues early, reduce the complexity of fixes, and ensure that all changes are thoroughly validated before deployment.

02 Comprehensive validation

Ensure that your testing covers all critical aspects of the data pipeline, including functionality, performance, security, and data quality. Utilise components like Assert Table and Assert Scalar Variables to automate these validations directly within the pipeline, ensuring that data integrity is maintained at each stage. Always ensure that all artifacts—pipelines, Python, and SQL scripts—are versioned together, and that tests are run on a specific version. By testing a fixed version, teams can be confident that the exact same artifacts will be deployed across environments, preventing discrepancies between environments.

03 Leverage CI/CD pipelines

Use CI/CD pipelines to automate the validation, testing, and deployment of your data pipelines. Implement linting for YAML, Python, and SQL files, validate pipeline structure, check for vulnerabilities, and perform dry-runs to preview pipeline behaviour. Trigger these pipelines both pre- and post-merge to ensure that your main branch remains functional and ready for deployment.

04 Run tests in lower environments

Use the If Component in conjunction with Project Variables to restrict testing to lower environments, such as Development or Staging. This prevents performance-critical production environments from being affected by testing activities while still validating data quality and pipeline behaviour in a controlled setting.

05 Focus on data quality assurance

Implement robust data quality checks using test components. These checks should be automated and embedded throughout the pipeline to ensure that data is accurate, consistent, and reliable before it moves downstream.

06 Document and review regularly

Keep detailed documentation of all testing processes, assertions, and validations within your pipelines. Regularly review and update these practices as your data and business requirements evolve, ensuring that your testing strategy remains effective and relevant.

07 Maintain flexibility

Refine and optimise your CI/CD pipelines as you gain more experience and as your needs change. Continuously iterate on your testing processes, incorporating feedback from real-world usage to develop a robust system that ensures the ongoing quality and reliability of your data pipelines.

05

Release Phase

The Release Phase is critical to ensuring that data pipelines execute correctly in production, unlocking reliable insights for your organisation. While DataOps practices are an area of ongoing research and development, the Data Productivity Cloud provides a robust foundation for managing deployments confidently.

By integrating with well-established and industry-leading tools, such as GitHub actions, teams can automate the deployment process to ensure consistency, reliability and speed. As DataOps matures, our commitment to innovation means that this phase will grow more robust, with best practices refined and adapted through real-world application and continuous feedback.

Artifact management and deployment

Artifact management streamlines the deployment process by providing full control over which versions of pipelines are deployed to production. This capability ensures that the deployment process is structured, repeatable, and auditable.

Key features

- **Assigning useful version names**
  Teams can assign meaningful version names (e.g. v1.0-release, v2.1-hotfix), making it easy to track which versions are being tested and deployed.

- **Selecting pipeline versions for scheduling**
  When creating schedules, teams can choose a specific artifact version or opt to “Always use latest,” providing flexibility depending on the stability and deployment needs of the pipeline.

- **Cross-environment deployment**
  Pipelines can be deployed from one environment to another, allowing seamless promotion of well-tested artifacts from Staging to Production, ensuring consistency and minimizing deployment risks.

- **Comprehensive version tracking**
  The ability to view all published artifacts across environments provides an audit trail, helping teams maintain governance, roll back when necessary, and ensure regulatory compliance.

Best practice

Utilise artifact management to enforce deployment consistency across environments. Always test a pipeline in a lower environment before deploying it to production to avoid unexpected issues.

Example workflow

- Developer merges feature branch into main.
- The GitHub Action publishes an artifact for the main branch and deploys it to the production environment.
- The team monitors performance and iterates if needed.

The following GitHub Action represents a foundational approach to deploying pipelines effectively, allowing teams to achieve reliable and scalable operations:

Example CI/CD pipeline: GitHub Actions

GitHub Action File

This main.yml file contains a Github Action which uses the Data Productivity Cloud Public API to Upload all Pipeline files in a project and Publish them as an Artifact to a given environment.

Github Action file: main.yaml

Note: For projects that use custom connectors, there is a separate set of files which can be incorporated into your repository. Find them with the associated README here.

Setup in GitHub

01 Repository Setup

Ensure this YAML workflow file (main.yml) is added to .github/workflows/ in the root of your repository.

02 Secrets Configuration

Add the following secrets to the repository under Settings > Secrets and variables > Actions:

- `MATILLION_PUBLIC_API_CLIENT_ID:`
  Client ID for Matillion API access.
- `MATILLION_PUBLIC_API_CLIENT_SECRET:`
  Client Secret for Matillion API access.

03 Repository Variables:

Add the following repository variables under Settings > Secrets and variables > Actions:

- `MATILLION_PROJECT_ID`: The ID of the Matillion project you are working with.

Pipeline setup

The workflow includes several hard-coded environment variables that need to be configured appropriately for your team and project. Below is an explanation of each variable, its purpose, and how to configure it

Versioning

```
VERSION_PREFIX: "v"
```

Purpose: Sets the prefix for semantic versioning (e.g. v1.0.0).

Default: `"v"`

Customization: Update this if your versioning scheme requires a different prefix (e.g., `"release-"` for `release-1.0.0`).

Authentication

```
CLIENT_ID: ${{
secrets.MATILLION_PUBLIC_API_CLIENT_ID }}
```

Purpose: Matillion API Client ID for authentication.

Source: Stored securely in GitHub Secrets.

Setup:

- Go to Settings > Secrets and variables > Actions.
- Add `MATILLION_PUBLIC_API_CLIENT_ID` with the Client ID from your Matillion account.

```
CLIENT_SECRET: ${{
secrets.MATILLION_PUBLIC_API_CLIENT_
SECRET }}
```

Purpose: Matillion API Client Secret for authentication.

Source: Stored securely in GitHub Secrets.

Setup:

- Go to Settings > Secrets and variables > Actions.
- Add `MATILLION_PUBLIC_API_CLIENT_SECRET` with the Client Secret from your Matillion account.

API Endpoints

`TOKEN_URL: "https://id.core.matillion.com/oauth/dpc/token"`

Purpose: Matillion OAuth2 token endpoint.

Default: This URL is specific to Matillion's authentication system and should not need modification.

```
PROJECTS_URL: "https://eu1.api.matillion.com/dpc/v1/projects"
```

Purpose: Base URL for Matillion API requests.

Default: `"https://eu1.api.matillion.com/dpc/v1/projects"`.

Customization:

- If your Matillion account operates in a different region, update this URL (e.g. `https://us1.api.matillion.com/dpc/v1/projects` for US regions).

Project and Environment

```
PROJECT_ID: ${{ vars.MATILLION_PROJECT_ID }}
```

Purpose: The ID of the Matillion project being managed by this workflow.

Source: Stored in GitHub Variables.

Setup:

Go to **Settings > Secrets and variables > Actions > Repository Variables.**

- Add `MATILLION_PROJECT_ID` with your project's ID from Matillion.

```
ENVIRONMENT_NAME: "DataOps Whitepaper-production"
```

Purpose: Human-readable name of the environment where pipelines are deployed.

Customization:

Replace "DataOps Whitepaper-production" with your target environment's name.

```
ENVIRONMENT_NAME_URL_ENCODED: "DataOps%20Whitepaper-production"
```

Purpose: URL-encoded version of the `ENVIRONMENT_NAME` for API calls.

Customization:

Replace `DataOps%20Whitepaper-production` with the URL-encoded version of your environment name.

- Use an online tool to encode spaces and special characters (e.g. "My Environment" → `"My%20Environment"`).

Test Pipelines

```
TEST_PIPELINE_NAMES: "dataops-orchestration-pipeline"
```

Purpose: List of pipelines to verify and execute during the testing phase of the pipeline.

Customization:

Provide a comma-separated list of pipeline names (e.g. `"pipeline1,pipeline2"`).

Execution Timing

```
CHECK_INTERVAL: 15
```

Purpose: Time (in seconds) to wait before checking pipeline statuses.

Default: `15`

Customization:

- Adjust this based on the typical runtime of your pipelines to balance efficiency and API rate limits.

Overview of the GitHub Action

This GitHub Action automates the process of validating, deploying, and testing Matillion pipelines on the main branch.

It runs whenever a commit is pushed or merged to the main branch (and can also be triggered manually).

Upon merge to main, the pipeline publishes the artifact to production. Deployments to production use versioned artifacts, which are generated as part of the pipeline, rather than latest code commits, ensuring reproducibility.

Note: Scheduling relies on specific versions, reducing the risk of unintended changes.

It consists of the following jobs:



Validate pipelines

- Checks out the repository.
- Validates YAML files using yamllint.

Execute pipelines

- Runs only after the validate-pipelines job succeeds
- Performs several tasks:

01 Calculate next version

Determines the next semantic version by incrementing the patch version.

02 Generate access token

Authenticates with Matillion and generates a token for subsequent API calls.

03 Upload artifacts and publish to default environment

Uploads .orch.yaml and .tran.yaml files to Matillion as part of the artifact deployment.

04 Tag main branch

Tags the main branch with the new version using the calculated semantic version.

05 Verify test pipelines

Ensures the specified test pipelines are published in the environment.

06 Execute test pipelines

Executes the pipelines and retrieves execution IDs.

07 Wait for pipeline completion

Monitors the status of executed pipelines, ensuring all complete successfully.

08 All pipelines completed successfully

Outputs success if all pipelines execute as expected.

Step-by-step explanation of each Job

01 Validate Pipelines

Purpose: Ensures the syntax of YAML files in the repository is valid.

Command: Uses yamllint to validate all YAML files.

02 Calculate Next Version

Purpose: Calculates the next semantic version by inspecting existing tags and incrementing the patch version.

Logic:

- Fetches all tags.
- Identifies the latest tag (e.g. v1.0.0).
- Increments the patch version to e.g. v1.0.1.

03 Generate Access Token

Purpose: Authenticates with Matillion and generates a bearer token for API calls.

04 Upload Artifacts and Publish to Default Environment

Purpose: Uploads files (.orch.yaml and .tran.yaml) to Matillion.

Implementation:

- Finds all relevant files in the repository.
- Constructs a single POST request with a multipart payload containing all files.
- Includes headers for versionName, environmentName, and commitHash.

05 Tag Main Branch

Purpose: Tags the main branch with the calculated version.

Implementation:

- Uses the GitHub API via actions/github-script to create the tag.

06 Verify Test Pipelines

Purpose: Ensures the specified test pipelines are published in the environment.

API: Calls the Matillion published-pipelines API and checks if all expected pipelines exist.

07 Execute Test Pipelines

Purpose: Executes the specified pipelines in the given environment.

API: Calls the Matillion pipeline-executions API to initiate execution for each pipeline.

Outputs: Collects pipelineExecutionId for each executed pipeline.

08 Wait for Pipeline Completion

Purpose: Monitors the status of executed pipelines.

API: Waits the defined amount and then calls the Matillion pipeline-executions status API to check execution status.

09 All Pipelines Completed Successfully

Purpose: Confirms all pipelines executed successfully.

Output: Displays a success message if all pipelines succeed.

How to Run

01 Trigger Automatically

- Push or merge changes to the main branch to automatically trigger the workflow.

02 Trigger Manually

- Go to the **Actions** tab in your repository.
- Select the workflow and click **Run workflow.**



End of Data Productivity Cloud Specific Section

Summary of best practices for release management

01 Use version naming conventions

Adopt consistent naming conventions to differentiate between development, testing, and production versions (e.g. `v1.0-dev`, `v1.0-staging`, `v1.0-prod`).

02 Schedule known versions in production

Avoid using the “Always use latest” option in production environments to ensure predictable and controlled deployments.

03 Implement deployment pipelines

Leverage automation to promote tested artifacts from lower environments (e.g. Development to Staging, Staging to Production) in a structured and documented manner.



Conclusion

The journey to effective DataOps begins with a well-structured approach to designing, testing, and deploying data pipelines. By embracing the principles and practices outlined in this whitepaper, including collaboration, automation, CI/CD workflows, and environment-specific configurations, teams can build scalable and resilient pipelines that support their business goals.

Adopting a **DataOps approach** fosters continuous improvement and ensures that pipelines remain reliable and adaptable in the face of evolving data and business requirements. Leveraging best practices such as dynamic configurations, robust testing, and immutable publishing, teams can confidently promote their pipelines across environments, minimizing risk and ensuring consistent behavior.

As you implement these practices, it's worth remembering that DataOps is not just a methodology—it's a mindset. By integrating these principles into your workflows, you'll unlock greater agility, efficiency, and collaboration within your organization, empowering your teams to deliver high-quality data solutions faster than ever before.

We encourage you to experiment, iterate, and refine your approach to DataOps. Start small, leverage the **Quick-Start Guide** to familiarize yourself with the platform, and scale your practices as you gain confidence. With the right tools and processes in place, you'll be well-positioned to meet the challenges of modern data operations and drive success in your data-driven initiatives.