**Madiha Aimon Tappal**

[madihaaimon@gmail.com](mailto:madihaaimon@gmail.com)

**Data Engineering Batch – 1**

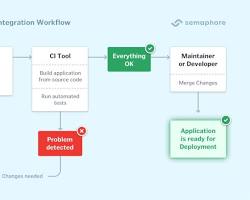
**Day – 25 Assignment**

**Azure Databricks**

**What is CI/CD?**

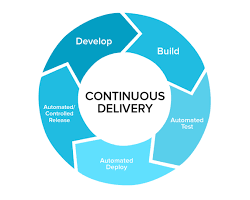
CI/CD stands for **Continuous Integration and Continuous Delivery/Deployment**. It's a set of practices that aim to automate the software development lifecycle, from code changes to deployment in production. Here's a breakdown of the two key components:

**Continuous Integration (CI)** involves frequently merging code changes from multiple developers into a shared repository. This happens automatically, often triggered by events like code commits. Each merge is then followed by an automated build and test process. This helps to catch bugs early on and ensure that the codebase remains stable and functional.

[w](https://semaphoreci.com/continuous-integration)

Continuous Integration

**Continuous Delivery (CD)** focuses on automating the delivery of code changes to different environments, such as testing, staging, and production. This allows for faster feedback and quicker releases. In some cases, CD can even involve **Continuous Deployment**, where every code change that passes the tests is automatically deployed to production.

[w](https://www.altexsoft.com/blog/continuous-delivery-and-integration-rapid-updates-by-automating-quality-assurance/)

Continuous Delivery

**Benefits of CI/CD:**

* **Faster feedback:** With CI/CD, developers can get feedback on their code changes quickly, which helps them identify and fix bugs early on.
* **Improved quality:** By automating the build and test process, CI/CD helps to ensure that code is always of high quality.
* **Reduced risk:** By automating deployments, CI/CD helps to reduce the risk of human error and ensures that deployments are consistent and reliable.
* **Faster releases:** By streamlining the development process, CI/CD can help teams release new features and updates more frequently.

Overall, CI/CD is a powerful set of practices that can help software development teams to be more efficient, productive, and deliver higher quality software.

A CI/CD pipeline is a concept central to software. It spans a whole field of processes, testing methods, and tooling, all facilitated by the Git code versioning process.

Since the terms “CI/CD Pipeline” and “Data Pipeline” are confusing, we will simply refer here to the CI/CD process.

Imagine you're building a toy train track (your data pipeline). Every time you add a new piece (a code change), you want to ensure it fits perfectly and doesn't derail the train (break the pipeline).

***Continuous integration***: Every time you add a new track piece, you immediately test it by running the toy train (data) through it. This ensures that your new addition didn't introduce any problems. If there's an issue, you know instantly and can fix it before it becomes a bigger problem.

***Continuous deployment***: Once you've confirmed that your new piece fits and the train runs smoothly, you don't wait to show it off. You immediately let everyone see and use the updated track. In other words, as soon as your changes are verified, they're made live and functional in the main track (production environment).

In technical terms, CI/CD automates this process:

1. CI checks and tests every new piece of code (or data transformation logic) you add to your data pipeline.
2. CD ensures that once tested and approved, this code gets added to the live system without manual intervention.

For a data engineer, this means faster, more reliable updates to data processes, ensuring high-quality data is delivered consistently. And if there's ever an issue, it's caught and fixed swiftly.

**CI/CD in data pipelines**

CI/CD, in the context of data pipeline deployment, focuses on automating data operations and transformations.

This merges development, testing, and operational workflows into a unified, automated process, ensuring that data assets are consistently high quality and that data infrastructure evolves smoothly, even at scale.

Using CI/CD for data pipeline automation has become more critical in ensuring the development velocity of processes such as training machine learning models, supporting a data science team, doing large-scale data analysis, business intelligence or data visualization, supporting the growth of unstructured data collection, and other business needs. For example, as organizations adopt a data mesh approach, more structured and trackable deployment becomes more vital.

Continuous integration and continuous deployment both have a set of characteristics that we need to understand to design an effective process:

**Continuous Integration (CI) in data pipelines**

**Automated Testing**

Automated tests check the integrity and quality of data transformations, ensuring that data is processed as expected and any error is spotted early.

**Version Control**

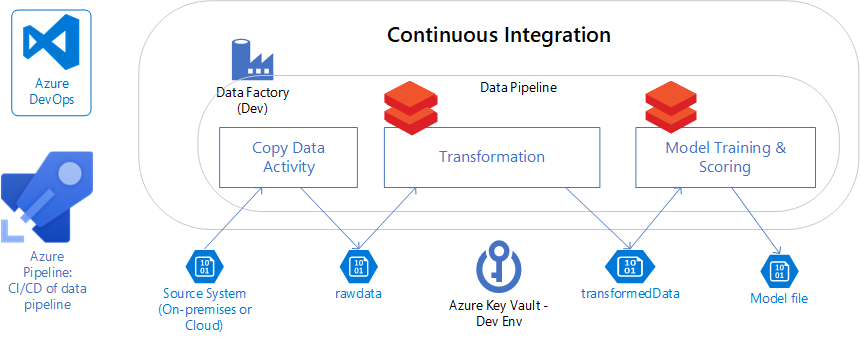
Data pipeline code (e.g., SQL scripts, Python transformations) is stored in repositories like Git, allowing tracking and managing changes.

**Consistent Environment**

CI tools can run tests in environments that mirror production, ensuring that differences in configuration or dependencies don't introduce errors.

**Data Quality Checks**

These might include checks for null values, data range violations, data type mismatches, or other custom quality rules.



**Continuous data pipeline deployment**

**Automated Deployment**

Once code changes pass all CI checks, CD tools can automate their deployment to production, ensuring seamless data flow.

**Monitoring and Alerts**

Once deployed, monitoring tools keep track of the data pipeline's performance, data quality, and any potential issues. Automated alerts can notify on discrepancies.

**Rollbacks**

In case an issue is identified post-deployment, CD processes allow for quick rollbacks to a previously stable state of the data pipeline.

**Infrastructure as Code (IaC)**

Many CD tools support IaCs. For example, cloud resources such as storage or compute can be provisioned automatically as part of the deployment process.

While CI/CD is a concept, there are various tools and frameworks developed to implement and support CI/CD practices, such as Jenkins, GitLab CI/CD, Travis CI, CircleCI, and many others.

How data engineers use Git

Now that we understand the value of CI/CD in the context of data pipeline deployments, let’s take a look at how to use Git to facilitate this process.

When most people think of Git, they think of version control—a way to track code changes, collaborate with others, and merge different code branches.

But pushing to Git can mean a lot more than just saving the latest version of a script. It can be synonymous with deployment, especially when integrated with tools like GitHub Actions.

**What is Git?**

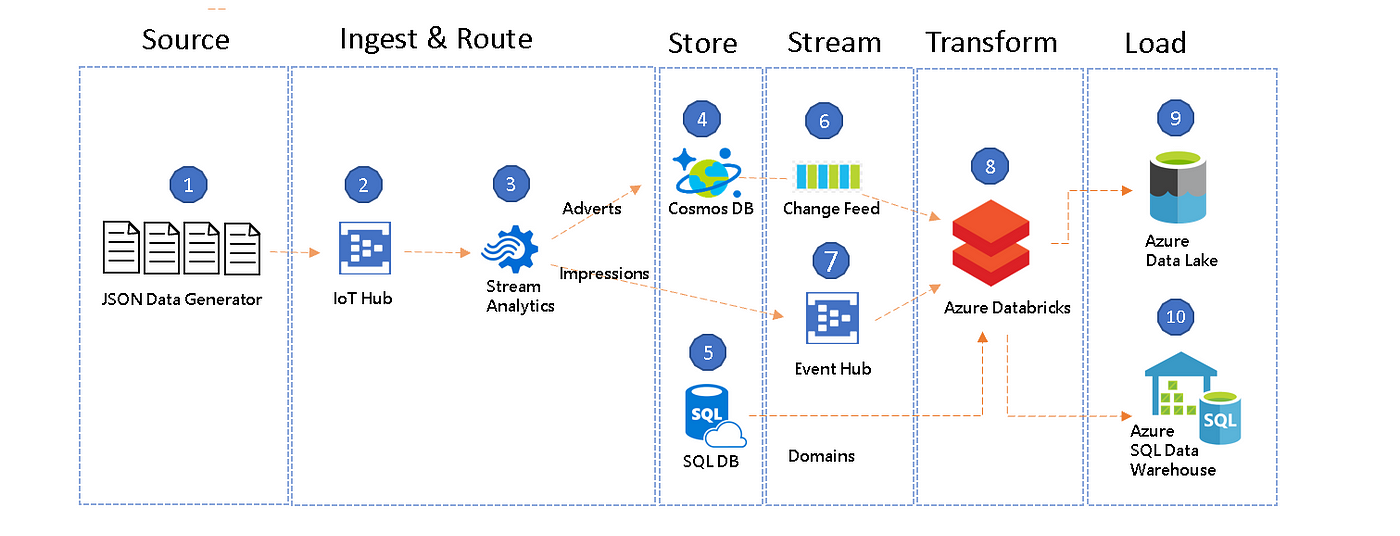
Git is a distributed version control system that facilitates collaborative software development by tracking changes across multiple contributors. For data engineers, it's indispensable for managing complex data projects, offering the ability to maintain multiple versions of data processing scripts and pipelines.

Git can be paired with data orchestration tools and integrated into CI/CD workflows, providing the benefits of streamlined deployment and consistency in data engineering tasks.

**ETL pipelines**

ETL (Extract, Transform, Load) pipelines are at the heart of data engineering. They're the processes that pull data from sources (databases, APIs, etc.), transform it into a usable format, and then load it into a destination, like databases or a data warehouse. When you deploy an ETL script to Git, you're not just saving the code—you could be triggering a series of events:

1. **Testing**: Automated tests are first run to ensure the new code doesn't break anything
2. **Deployment**: Once tests pass, the ETL process can be automatically deployed to a staging or production environment
3. **Notifications**: If any part of the process fails, or if it's successfully completed, notifications can be sent out



**Data pipeline deployment**

While Git's primary function is version control, its integration with CI/CD solutions like GitHub Actions makes it a powerful deployment tool. By setting up specific "actions" or "workflows", data engineers can automate the deployment of their pipelines. This means that when we push code to a Git repository, it can automatically be deployed to a production environment, provided it passes all the set criteria.

This approach brings production-level engineering to data operations. It ensures that data pipelines are robust, reliable, and continuously monitored. It also means that data engineers can focus on writing and optimizing their code, knowing that the deployment process is automated and in safe hands.

**CI/CD, Git, and data processes**

Data professionals integrate Git and CI/CD into their workflows to automate repetitive tasks, ensure data quality, and focus on optimizing data pipelines. Here are some common workflows that you may have encountered:

Data validation

Every data engineer knows about “garbage in, garbage out.” Incoming data always reintroduces potential anomalies or inconsistencies. CI/CD tools can execute scripts that meticulously validate the integrity and quality of new data. This proactive approach minimizes the risk of downstream issues and maintains the trustworthiness of the data ecosystem.

Scheduled data jobs

Certain analytical tasks, like aggregating metrics or updating summary tables, don't need to be executed on-the-fly. Instead, they can be scheduled to run at specific intervals, optimizing resource usage. With the scheduling features of CI/CD tools, data engineers can seamlessly integrate these tasks into their Git repositories.

Catching anomalies and failures

The complexity of data pipelines means that even with the best precautions, things can go awry. Using CI/CD tools, data engineers can set up workflows that automatically send notifications to platforms like Slack or email whenever specific events occur. This integration ensures that any disruptions in the data flow are quickly communicated, allowing teams to swiftly address and maintain the integrity of the data pipeline.

A novel workflow

In traditional software development, the use of branch deployments has long been a staple to ensure that new features, bug fixes, or code refactors are developed and tested in isolation before merging them into the main codebase.

By contrast, data engineering has typically involved separate stages of development, testing, and deployment, often with manual interventions or handoffs between stages.

However, traditional software development practices don't always neatly translate to data workflows. Enigma has discussed how the conventional 'dev-stage-prod' pattern may not be the optimal approach for data pipelines.

Instead, branch deployments can be seen as a branch of data platforms – you can preview, validate, and iterate on changes without impacting the production environment or overwriting existing testing setups.

This shift in perspective highlights the need for tools and practices tailored specifically for data engineering. Enter modern CI/CD tools such as the ones mentioned earlier.

Instead of rigidly adhering to the 'dev-stage-prod' paradigm, data engineers can leverage these CI/CD solutions to create dynamic, ephemeral environments on demand, ensuring that each data transformation or pipeline change is tested in an environment that closely mirrors production.

But how does this actually work in practice?

CI/CD and ephemeral environments

When an engineer creates a new feature branch in Git, CI/CD tools can be set up to listen for this specific event (i.e., the creation of the branch). Through defined workflows, it can communicate with the APIs or SDKs of platforms like AWS, Azure, or GCP.

This means that if your data engineering workflow requires resources such as an Amazon Redshift cluster, a Microsoft Azure Data Lake, or a Google Cloud Dataflow job, these CI/CD tools can automate their provisioning.

Upon detecting the branch creation event, the CI/CD system can trigger a predefined workflow that automates the process of setting up an *ephemeral* environment.

What are ephemeral environments?

Ephemeral dev/test environments are temporary environments that are spun up for the purpose of development, testing, or experimentation and are torn down after. They are used in data engineering workflows to test data pipelines, transformations, and integrations in a safe, isolated manner before deploying to production.

The entire process, from setting up the necessary configurations and seeding data, to ensuring the right permissions, can be automated, ensuring that the environment is ready for testing in a matter of minutes.

“Ephemeral” means short-lived, and once testing is completed, it’s crucial to tear down the resources to avoid incurring costs or leaving unused resources running. CI/CD can be set up to automatically de-provision resources too.

Thus, CI/CD acts as the automation bridge between the act of branching in Git and the provisioning of resources required for the ephemeral environment.

Git best practices

As data engineers, adopting best practices in Git not only ensures the integrity of our data pipelines but also fosters collaboration and efficiency. We present 7 habits for Git that every data engineer should adopt:

1. Handling large data files: While some teams historically used Git Large File Storage (LFS) to manage and version large datasets, there's a growing consensus that data should be kept separate from code repositories. Modern practices often recommend versioning cloud storage buckets or using dedicated data versioning tools.
2. Pull requests: The heart of collaboration. Use pull requests to propose changes, solicit feedback, and ensure code quality before merging.
3. Code reviews: Foster a culture of reviewing code. It's not just about catching errors but also about sharing knowledge and ensuring consistent coding standards.
4. Commit often: It's easier to merge smaller, frequent changes than large, infrequent ones. Aim for atomic commits, where each commit represents a single logical change
5. Commit with clear messages: Write clear, concise messages that explain the "why" behind your changes, not just the "what"
6. Branch deployments: They automatically create staging or temporary environments based on the code in a specific branch of a git repository. This allows data professionals (like data scientists) to test, preview, and validate the changes made in that branch in an isolated environment before merging them into the main or production branch