Git workflows: branching, PRs, code reviews

In the world of software development, a **Git Workflow** is a recipe or a set of rules that governs how a team uses Git to manage their code. It ensures that the "main" version of the software stays stable while allowing multiple developers to work on new things simultaneously.

---

1. Branching Strategy

Branching is the core of Git. It allows you to diverge from the main line of development to work on a specific task without affecting the "production-ready" code.

- **Main/Master Branch:** This is the source of truth. It should always contain deployable, tested code.
- **Feature Branches:** For every new task (a bug fix or a new feature), a developer creates a new branch off of `main`.
  - *Naming convention:* `feature/login-page` or `bugfix/header-alignment`.
- **Isolation:** By working in a branch, you can experiment, break things, and commit frequently without worrying about breaking the project for others.

---

2. Pull Requests (PRs)

Once you've finished your work on a feature branch, you don't just "merge" it yourself. You open a **Pull Request**.

A PR is essentially a formal request to merge your code changes into the main branch. It acts as a staging area where:

- **The diff is visible:** Everyone can see exactly what lines were added, changed, or deleted.
- **Automated tests run:** Continuous Integration (CI) tools (like GitHub Actions or Jenkins) check if your code breaks the build.
- **Discussion happens:** Team members can ask questions about your implementation.

---

3. The Code Review Process

The code review is the "quality gate" of the workflow. It isn't just about finding bugs; it's about sharing knowledge and maintaining coding standards.

Best Practices for Reviewers:

- **Check Logic, Not Just Syntax:** Focus on edge cases and architectural choices. Let automated "linters" handle the formatting.

- **Be Kind:** Use phrases like "I suggest..." or "Have you considered..." rather than "This is wrong."
- **The Goal:** Ensure the code is readable and maintainable for whoever touches it next.

Best Practices for Authors:

- **Keep PRs Small:** It is much easier (and faster) to review 50 lines of code than 1,000.
- **Write Clear Descriptions:** Explain *why* you made certain decisions, not just *what* you changed.

Summary of the Standard Flow

Step	Action	Description
1	**Branch**	`git checkout -b feature/new-idea`
2	**Commit**	Save your progress with meaningful messages.
3	**Push**	Send your branch to the remote server (GitHub/GitLab).
4	**PR**	Open a Pull Request for the team to see.
5	**Review**	Peer review and automated testing.
6	**Merge**	The code is moved into the `main` branch and the feature branch is deleted.

CI/CD concepts and benefits

Building on our talk about Pull Requests, **CI/CD** is the engine that actually powers that "automated testing" and "merging" process we discussed. It transforms manual, risky software releases into a smooth, automated assembly line.

---

1. Core Concepts

CI/CD is actually composed of three distinct phases that work together to move code from a developer's laptop to the customer.

CI: Continuous Integration

This is the "safety net" for your code.

- **The Action:** Developers merge their code changes back to the main branch as often as possible (usually daily).
- **The Goal:** Every time code is pushed, an automated system **builds** the app and runs **unit tests**.

- **The Benefit:** It catches "integration bugs" (when my code breaks your code) within minutes instead of weeks.

CD: Continuous Delivery

This is about being **"ready to go"** at any moment.

- **The Action:** After passing CI, the code is automatically deployed to a **Staging** or **Testing** environment.
- **The Distinction:** The code is ready for production, but a **human** still has to click a "Deploy" button to push it to real users.

CD: Continuous Deployment

This is the "elite" level of automation.

- **The Action:** Every single change that passes the automated tests is **automatically** pushed to production.
- **The Risk/Reward:** There is no human gatekeeper. If the tests pass, the code is live. This requires extremely high-quality automated testing.

---

2. Key Benefits of the CI/CD Pipeline

Implementing this workflow changes the "vibe" of a development team from stressful to predictable.

Benefit	**How it happens**
Faster Time to Market	Small updates go live in minutes, not months.
Higher Quality	Automated tests (Unit, Integration, and UI) run on every single line of code.
Reduced Risk	Since you deploy small changes, if something breaks, it’s easy to find and fix.
No "Merge Hell"	Frequent small merges prevent the nightmare of trying to combine months of work at once.
Happier Developers	Engineers spend less time on "boring" manual deployments and more time building features.

---

3. How it looks in the "Real World"

Imagine you are fixing a typo on a website:

1. **Push:** You push your fix to a branch.

2. **CI Trigger:** The CI server (like GitHub Actions) wakes up, compiles the code, and runs 500 tests.
3. **The PR Gate:** Your teammate sees "All checks passed" in your PR and approves it.
4. **Merge & Deploy:** You merge. The CD pipeline takes that code, packages it into a "container" (like Docker), and replaces the old code on the server without a single second of downtime.

In the context of data engineering, **GitHub Actions** acts as an orchestrator. While it's famously used for web apps, it is incredibly powerful for data pipelines—especially for **ELT/ETL** processes, data validation, and automated reporting.

Git Actions for data pipelines

1. How GitHub Actions Fits into Data Pipelines

GitHub Actions uses **YAML** files to define "Workflows." In a data context, these workflows are usually triggered by one of three things:

- **Events:** A new `.csv` file is pushed to the repo.
- **Schedules:** A "Cron job" that runs every morning at 8:00 AM to refresh a dashboard.
- **Manual:** A "Workflow Dispatch" where a human clicks a button to re-run a pipeline.

---

2. Common Data Use Cases

A. Data Validation (Great for "Data Contracts")

Before your data hits your warehouse (like Snowflake or BigQuery), you can use GitHub Actions to run **Great Expectations** or **pytest**.

- **The Flow:** A PR is opened $\rightarrow$ Actions runs a script to check if the new data has `null` values or wrong data types $\rightarrow$ If it fails, the PR cannot be merged.

B. Automated Scraping & Transformation

If you have a small-to-medium dataset, GitHub Actions can actually *be* your compute engine.

1. **Extract:** Python script scrapes an API.
2. **Transform:** Pandas or Polars cleans the data.
3. **Load:** The script pushes a clean `.parquet` file to an AWS S3 bucket.

C. dbt (data build tool) Integration

One of the most popular uses is running **dbt** commands.

- When code is merged to main, GitHub Actions runs dbt cloud run or dbt run (using a lightweight runner) to rebuild your SQL models in the warehouse.

---

3. A Simple Data Pipeline Example

Here is what a .github/workflows/data_pipeline.yml file looks like for a daily Python script:

YAML
```
name: Daily Data Refresh

on:
  schedule:
    - cron: '0 8 * * *' # Runs at 8:00 AM every day
  workflow_dispatch:      # Allows manual trigger

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.9'

      - name: Install dependencies
        run: pip install pandas sqlalchemy psycopg2

      - name: Run ETL Script
        env:
          DB_PASSWORD: ${{ secrets.DB_PASSWORD }} # Use GitHub Secrets for security
        run: python scripts/etl_process.py
```

---

4. Pros and Cons for Data Work

Pros	Cons
Version Controlled: Your pipeline logic lives right next to your code.	**Time Limits:** Jobs usually have a 6-hour timeout (bad for massive migrations).
Free Tier: Generous free minutes for public (and private) repos.	**Not a Full Orchestrator:** It lacks the complex "dependency graphs" of Airflow or Prefect.
Secrets Management: Securely handles API keys and DB passwords.	**Limited Compute:** Not meant for processing Terabytes of data in memory.

Automated testing in CI/CD:

Automated Testing is the "filter" that prevents bad code from reaching your users. Without it, CI/CD is just a fast way to deploy bugs.

To make the process efficient, developers use the **Testing Pyramid** strategy: running many fast, cheap tests first, and a few slow, expensive tests last.

---

1. The Testing Pyramid

The goal is to catch bugs as early as possible in the pipeline.

- **Unit Tests (The Base): * What:** Tests a single function or component in isolation.
    - **Speed:** Extremely fast (milliseconds).
    - **CI Role:** Runs immediately on every "Push." If a unit test fails, the build stops instantly.
- **Integration Tests (The Middle):**
    - **What:** Tests how two or more parts of the system work together (e.g., "Does my Python script successfully talk to the Database?").
    - **Speed:** Moderate.
- **End-to-End (E2E) Tests (The Top):**
    - **What:** Mimics a real user. A tool (like Playwright or Selenium) opens a browser, clicks buttons, and checks if the page loads.
    - **Speed:** Slow and "flaky" (can fail due to network lag).
    - **CI Role:** Usually runs only after the code is deployed to a "Staging" environment.

---

2. Where Tests Live in the CI/CD Pipeline

The pipeline acts as a sequence of "gates." If a test fails at any gate, the code is rejected.

Stage	**Test Type**	**Purpose**
Commit/Push	**Linting & Static Analysis**	Checks for syntax errors and formatting (e.g., Flake8, ESLint).
Build	**Unit Tests**	Ensures the basic logic of the code is sound.
Staging/Pre-prod	**Integration & API Tests**	Ensures the new code doesn't break existing connections.
Post-Deployment	**Smoke Tests**	A quick check of major features in the live environment to ensure the "lights are on."

---

3. Automated Testing for Data Pipelines

Testing isn't just for software; for data engineering, we test the **data itself**:

- **Schema Validation:** Does the incoming data still have the same 10 columns? Did a column name change from user_id to ID?
- **Null Checks:** Does a critical field like email suddenly contain null values?
- **Volume Tests:** Did we expect 10,000 rows but only received 50?
- **Data Contracts:** Tools like **dbt tests** or **Great Expectations** allow you to write these checks as code and run them inside GitHub Actions.

---

4. Key Benefits of Automating These Tests

1. **Confidence:** You can merge code on a Friday afternoon because you know the 500 tests have your back.
2. **Documentation:** Tests act as documentation. If you want to know what a function is *supposed* to do, look at the test case.
3. **Speed:** You don't have to wait for a QA team to manually click through the app for two days.

Terraform basics: providers, resources, state

To understand **Terraform**, think of it as a way to write a "blueprint" for your digital infrastructure. Instead of clicking buttons in a cloud console (like AWS or Snowflake), you write code that describes what you want, and Terraform makes it happen.

For a data engineer, this is how you create your S3 buckets, Snowflake databases, or BigQuery datasets in a way that is repeatable and version-controlled.

---

1. Providers: The Translators

Terraform doesn't know how to talk to every cloud by default. It uses **Providers** as plugins to translate your code into specific API calls.

- **Examples:** AWS, Google Cloud, Azure, and even **Snowflake** or **dbt Cloud**.
- **Role:** You declare which provider you need, and Terraform downloads the logic to manage those specific services.

Terraform

```
# Example Provider Configuration
provider "snowflake" {
  account  = "xy12345.us-east-1"
  username = "terraform_user"
}
```

---

2. Resources: The Building Blocks

Resources are the actual "things" you want to build. Each resource has a **type** (what it is) and a **name** (what you call it internally).

Terraform
```
# Creating a Snowflake Database
resource "snowflake_database" "raw_data" {
  name    = "RAW_DB"
  comment = "Database for raw landing data"
}
```

- **Declarative Nature:** You tell Terraform "I want a database named RAW_DB." If the database already exists, Terraform does nothing. If it's missing, it creates it.

---

3. The State: Terraform's Memory

The **State file** (`terraform.tfstate`) is the most critical part of Terraform. It is a JSON file that maps your code to the real-world resources.

- **The Source of Truth:** Terraform uses this file to remember what it has already built.
- **The "Diff":** When you run `terraform plan`, Terraform compares your **Code** vs. the **State File** vs. the **Real World**.
- **Remote State:** In a team, you **never** keep this file on your laptop. You store it in a "Remote Backend" (like an S3 bucket) so everyone on the team shares the same memory.

---

4. The Standard Workflow

Every Terraform project follows these four steps:

Command	Purpose
terraform init	Downloads the providers (the plugins).
terraform plan	Shows you a "preview" of what will happen (e.g., "Plan: 1 to add, 0 to change, 0 to destroy").
terraform apply	Executes the changes and updates the state file.
terraform destroy	Deletes everything managed by that specific project.

---

5. Why Data Teams Love Terraform

1. **Environment Parity:** You can use the same code to spin up a "Dev" Snowflake account and a "Prod" Snowflake account, ensuring they are identical.
2. **Safety:** Before making a change, `terraform plan` tells you if you're about to accidentally delete a table with 5 years of data.
3. **Auditing:** Since the infrastructure is code, you can see exactly who changed a firewall rule or added a new database by looking at the Git history.

CloudFormation overview

While Terraform is a tool that works across many clouds (AWS, Azure, Snowflake), **AWS CloudFormation** is Amazon's native "Infrastructure as Code" service. It is designed specifically to model, provision, and manage AWS resources using a single source of truth.

If you are a data engineer working exclusively in the AWS ecosystem (using S3, Glue, Redshift, and Lambda), CloudFormation is the "built-in" way to manage your environment.

---

1. Core Concepts: The Template and The Stack

CloudFormation works by taking a **Template** and turning it into a **Stack**.

- **The Template (The Blueprint):** A JSON or YAML file where you describe the resources you want. You don't write *how* to build them; you describe *what* they should look like.
- **The Stack (The Reality):** When you "run" a template, CloudFormation creates a **Stack**. A stack is a single unit that contains all the resources defined in your template. If you delete the stack, every resource inside it (the database, the bucket, the permissions) is deleted automatically.

---

2. Anatomy of a Template

A CloudFormation template has several key sections. Only **Resources** is mandatory.

Section	Purpose	Example
Parameters	Custom values you provide at runtime (e.g., "EnvironmentName").	Dev, Prod
Mappings	Static lookup tables (e.g., mapping an AWS Region to a specific AMI ID).	us-east-1 $\rightarrow$ ami-123
Resources	**(Required)** The actual AWS objects you want to create.	AWS::S3::Bucket, AWS::Redshift::Cluster
Outputs	Values you want to see after the build is done or share with other stacks.	The URL of a new website or an S3 Bucket ARN.

Simple Example (YAML)

YAML
```
Resources:
  MyDataBucket:
    Type: 'AWS::S3::Bucket'
    Properties:
      BucketName: my-unique-data-landing-zone
```

---

3. Key Features for Data Engineering

CloudFormation offers a few specialized features that make managing data pipelines safer:

- **Change Sets:** Before you apply a change, CloudFormation can show you a "Preview" (similar to a `terraform plan`). It tells you if an update will **modify** a resource or **replace** (delete and recreate) it—which is vital when dealing with databases.
- **Drift Detection:** It can scan your resources to see if someone manually changed a setting in the AWS Console that contradicts your code.
- **Rollback on Failure:** If you try to create a 10-resource pipeline and the 9th resource fails, CloudFormation will automatically "roll back" and delete the first 8 resources so you aren't left with a messy, half-finished environment.

---

4. CloudFormation vs. Terraform

Since we just discussed Terraform, here is how they compare:

Feature	**AWS CloudFormation**	**HashiCorp Terraform**
Cloud Support	**AWS Only**	**Multi-Cloud** (AWS, Azure, Snowflake, etc.)
State Management	Managed automatically by AWS (Hidden).	Managed by you (State file in S3).
Language	YAML or JSON.	HCL (HashiCorp Configuration Language).
Best For...	AWS-only shops who want a managed service.	Teams using multiple clouds or third-party tools.

Environment management (dev/staging/prod)

In modern software and data engineering, **Environment Management** is the practice of maintaining separate "copies" of your system so that you can build, test, and break things without affecting your real users.

The gold standard is to have three distinct environments, often referred to as **Tiered Environments**.

---

1. The Standard Three-Tier Architecture

Environment	**Purpose**	**Who uses it?**	**Data Strategy**
Dev (Development)	The "Sandpit." Where engineers write code and experiment.	Developers	Synthetic/Fake data.

Environment	Purpose	Who uses it?	Data Strategy
Staging (UAT)	The "Pre-release." A near-exact clone of Production.	QA, Product Owners	Masked or sampled production data.
Prod (Production)	The "Live" version. This is where real customers live.	Real Users	Real, sensitive data.

---

2. Managing Environments with Code (IaC)

Using **Terraform** or **CloudFormation** (as we discussed) is the best way to manage these. You use **Variables** or **Parameters** to swap out the configurations for each environment using the same blueprint.

Example: Using Terraform Workspaces

Instead of writing new code for "Prod," you tell Terraform to use a different variable file:

- **dev.tfvars**: instance_size = "t3.micro" (Cheap and small)
- **prod.tfvars**: instance_size = "m5.large" (Powerful and expensive)

---

3. Data Engineering Specifics: "The Data Problem"

Managing environments is harder for data engineers than for software engineers because of the **Data**. You can't just copy a 50TB database to "Dev" every morning.

Strategies for Data Environments:

1. **Zero-Copy Cloning (Snowflake):** This is a "cheat code" for data engineers. It allows you to create a clone of your Production database for Dev in seconds without actually copying the data or paying for double storage.
2. **Data Masking:** If you do move data to Staging for testing, you must use scripts to "scramble" sensitive info like PII (Personally Identifiable Information) so developers don't see real customer emails or credit cards.
3. **Static Sampling:** Keeping a small, representative sample (e.g., 1% of users) in the Dev environment to keep costs low while still having "real-looking" data to test logic.

---

4. Promotion Logic: How Code Moves

Code should only move "up" the chain via the Git workflows we talked about earlier:

1. **Commit to Feature Branch:** Code runs in a **Sandbox**.
2. **Merge to Develop Branch:** Code is automatically deployed by CI/CD to **Dev**.
3. **Merge to Main/Release Branch:** Code is deployed to **Staging** for final testing.
4. **Tag a Release:** Code is pushed to **Production**.

Governance-as-Code (GaC) is the shift from managing data rules and security via manual tickets and spreadsheets to managing them through **version-controlled code**.

By treating governance artifacts like software, you gain auditability, consistency, and the ability to automate "gatekeeping" within your CI/CD pipelines.

---

1. Data Quality Rules (Great Expectations)

Instead of a data steward manually checking for nulls, you define "Expectations" in YAML.

- **Version Control:** If you change a threshold (e.g., from 95% to 99% accuracy), the Git history shows *who* changed it and *why*.
- **CI/CD Integration:** During a PR, GitHub Actions runs these tests. If the data quality falls below the threshold, the pipeline "fails," preventing bad data from entering production.

---

2. Match/Merge Logic

In Master Data Management (MDM), you often have "fuzzy matching" (e.g., deciding that "Jon Doe" and "John Doe" are the same person).

- **The Artifact:** Store your logic (Levenshtein distance, Jaro-Winkler thresholds) in a config file (YAML/JSON).
- **The Benefit:** If the "Merge" logic is too aggressive and creates duplicate records, you can **revert** the Git commit to a previous version of the logic instantly.

---

3. Access Policies (IAM & Bucket Policies)

This is the "Security-as-Code" pillar. You define who can see what using Terraform or CloudFormation.

- **The Artifact:** A JSON policy document.
- **Example:**

  JSON

  ```
  {
  ```

```
  "Effect": "Allow",
  "Action": "s3:GetObject",
  "Resource": "arn:aws:s3:::prod-financial-data/*",
  "Condition": { "Bool": { "aws:MultiFactorAuthPresent": "true" } }
}
```

- **Governance Benefit:** No one should ever manually click "Make Public" on a bucket. The CI/CD pipeline scans these JSON files for security holes before they are applied.

---

4. Metadata Schemas (JSON Schema)

Standardizing how data is described (e.g., "Every table must have a created_at and owner tag") is managed through schemas.

- **The Artifact:** A .json schema file that defines required fields and data types.
- **Governance Benefit:** You can run a script in your CI/CD pipeline that checks your data warehouse metadata. If a new table is missing an "Owner" tag, the build fails.

---

5. Approval Workflows

Governance often requires a "human in the loop" for sensitive changes.

- **The Artifact:** environments.yml or GitHub Environment settings.
- **The Benefit:** You can code a rule that says: *"To deploy to Production, you need a 'Review' approval from the Security team AND the Data Lead."* This replaces the need for a "Change Approval Board" (CAB) meeting.

Hands-on:

• Create Terraform templates for S3, Lambda, RDS

• Deploy infrastructure through CI/CD

• Create version-controlled governance configs:

• Data quality rules file (YAML)

• Access policy templates (JSON)

• Metadata schema definitions

• Deployment pipeline with approval gates for production
This hands-on guide connects your infrastructure (Terraform), your automation (GitHub Actions), and your governance (YAML/JSON) into one unified system.

---

1. Terraform Templates (Infrastructure)

Create a file named main.tf. This defines your "Data Lake" landing zone (S3), your processing engine (Lambda), and your structured storage (RDS).

```
Terraform
# 1. S3 Bucket for Raw Data
resource "aws_s3_bucket" "data_lake" {
  bucket = "my-company-data-lake-${var.env}"
}

# 2. RDS Instance (PostgreSQL)
resource "aws_db_instance" "warehouse" {
  allocated_storage    = 20
  engine               = "postgres"
  instance_class       = "db.t3.micro"
  db_name              = "warehouse_${var.env}"
  username             = "admin"
  password             = var.db_password
  skip_final_snapshot  = true
}

# 3. Lambda for ETL
resource "aws_lambda_function" "etl_processor" {
  filename      = "lambda_payload.zip"
  function_name = "etl_processor_${var.env}"
  role          = aws_iam_role.lambda_role.arn
  handler       = "index.handler"
  runtime       = "python3.9"
}
```

---

2. Governance Artifacts (The "Code" in GaC)

Instead of burying these in a UI, we store them in a /governance folder in your Git repo.

Data Quality Rules (/governance/quality_rules.yml)

Using the **Great Expectations** style:

```
YAML
expectation_suite_name: raw_sales_data
expectations:
  - expectation_type: expect_column_values_to_not_be_null
    kwargs:
      column: order_id
  - expectation_type: expect_column_values_to_be_between
    kwargs:
      column: discount_percent
      min_value: 0
      max_value: 100
```

Access Policy Templates (/governance/iam_policy.json)

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["s3:GetObject", "s3:ListBucket"],
      "Resource": ["${bucket_arn}", "${bucket_arn}/*"]
    }
  ]
}
```

Metadata Schema (/governance/schema_definition.json)

```json
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "UserEvent",
  "type": "object",
  "properties": {
    "user_id": { "type": "string" },
    "event_time": { "type": "string", "format": "date-time" }
  },
  "required": ["user_id", "event_time"]
}
```

---

3. Deployment Pipeline with Approval Gates

We use **GitHub Actions** to tie it all together. We will create two environments in GitHub: `staging` and `production`.

Create `.github/workflows/deploy.yml`:

```yaml
name: Infrastructure & Governance Deployment

on:
  push:
    branches: [ main ]

jobs:
  # STEP 1: Validate Governance & Code
  validate:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Run Data Quality Check Simulation
        run: |
          echo "Validating YAML/JSON Governance artifacts..."
          # Insert script here to check JSON syntax or run Great Expectations

  # STEP 2: Deploy to Staging
  deploy-staging:
    needs: validate
    environment: staging
    runs-on: ubuntu-latest
    steps:
      - name: Terraform Apply (Staging)
        run: terraform apply -var="env=staging" -auto-approve

  # STEP 3: Deploy to Production (WITH APPROVAL GATE)
  deploy-production:
    needs: deploy-staging
    environment: production # This environment must have "Required Reviewers" set in GitHub settings
    runs-on: ubuntu-latest
    steps:
      - name: Terraform Apply (Production)
        run: terraform apply -var="env=prod" -auto-approve
```

---

4. How the "Gate" works in practice

1. **Push to Main:** The `validate` and `deploy-staging` jobs run automatically.
2. **The Wait:** The pipeline **pauses** at `deploy-production`.
3. **The Approval:** GitHub sends a notification to the "Data Lead" or "Security Team." They must review the logs and click **"Approve and Deploy."**
4. **The Result:** Only after human approval does the infrastructure and governance rules hit the live production environment.

Summary of the "Hands-on" Flow

1. **Develop:** You modify the `quality_rules.yml` to be stricter.
2. **PR:** You open a Pull Request.
3. **Automated Test:** GitHub Actions runs `pytest` or `Great Expectations` against your new rules using sample data.
4. **Merge:** You merge to `main`.
5. **Staging:** The new rules are deployed to the staging data lake.
6. **Approval:** A Senior Engineer checks the staging results and approves the production deployment.