# *Implementing Version Control and CI/CD Practices in Power BI*



PBIP DEFINITION:

A PBIP (POWER BI PROJECT) IS A FOLDER-BASED FORMAT THAT BREAKS DOWN A TRADITIONAL .PBIX FILE INTO MULTIPLE READABLE AND EDITABLE FILES.

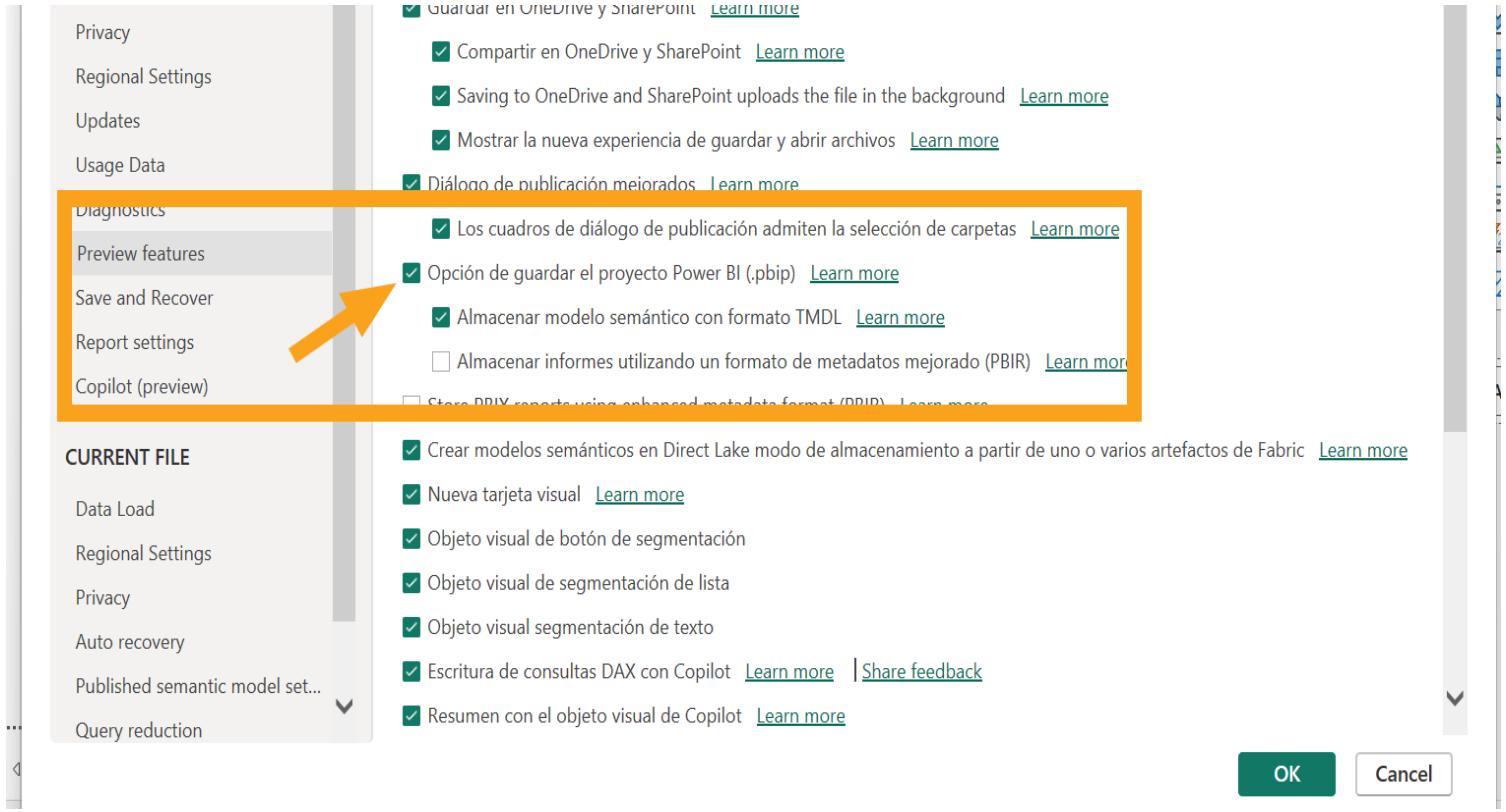
✅ **VERSION CONTROL**: MAKES IT EASIER TO TRACK CHANGES IN GIT REPOSITORIES.

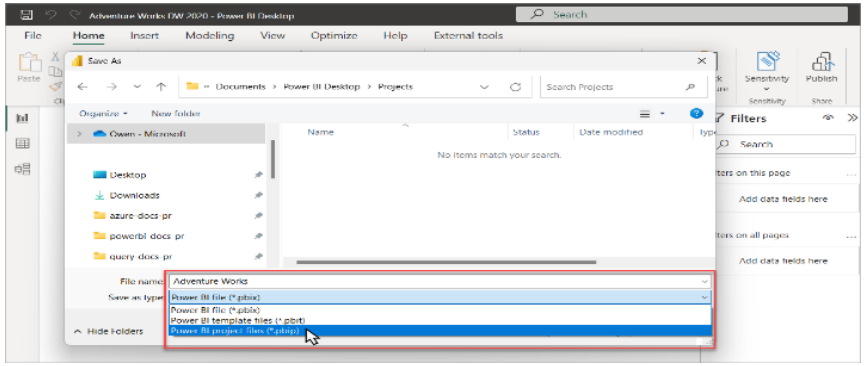
✅ **TEAM COLLABORATION**: MULTIPLE DEVELOPERS CAN WORK IN PARALLEL ON DIFFERENT PARTS OF THE PROJECT.

✅ **CI/CD INTEGRATION**: ENABLES DEVOPS PIPELINES AND AUTOMATED DEPLOYMENTS IN POWER BI.

# Using PBIP

Its use currently requires activation in Power BI settings.




Once this feature is enabled, when saving the file, we must select the .pbip extension.

## PBIP File Structure

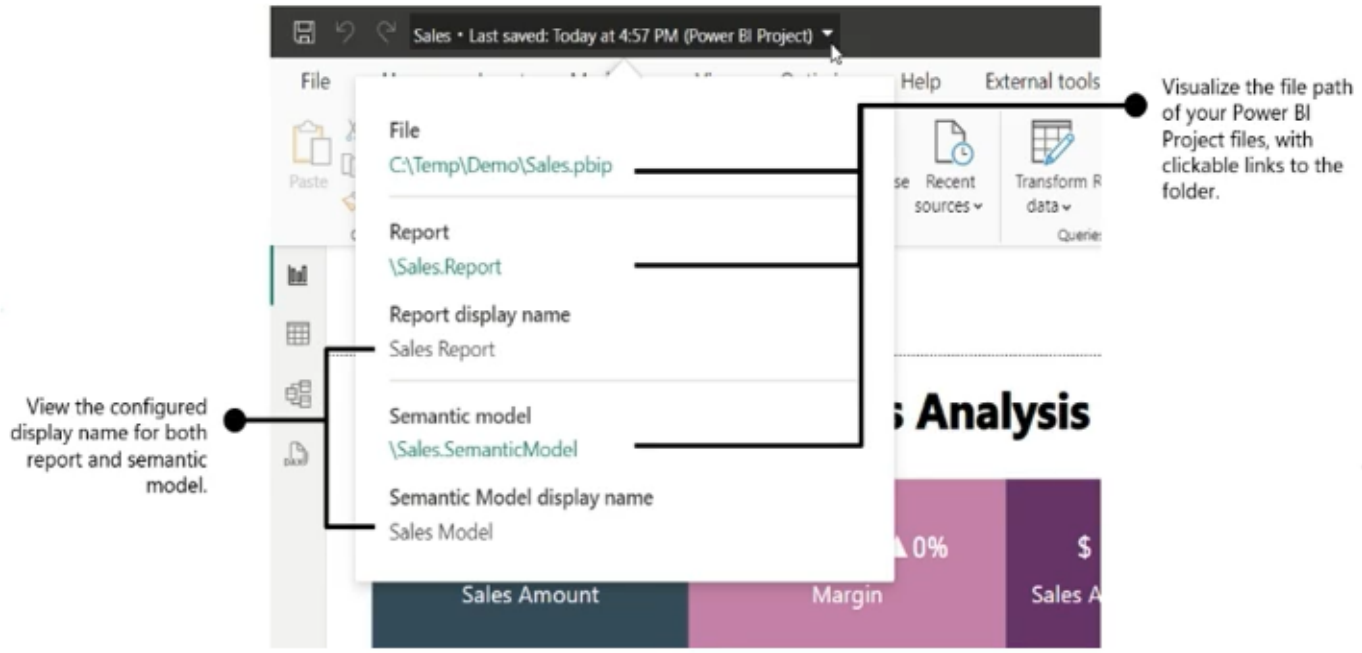
The PBIP file is structured as follows:

**<project>.PBIP** – Main file that acts as a pointer to the dataset and the report. Opening it loads the model and the report in Power BI. Replaces the old .pbix file.

**Dataset Folder** – Contains the data model metadata (tables, relationships, measures, security, etc.). Key file: model.bim.

**Report Folder** – Contains the report definition. Central file: report.json.

**.gitignore** – Defines which files to exclude from version control.



## TMSL

```
{
  "name": "Product",
  "annotations": [
    {
      "name": "PBI_ResultType",
      "value": "Table"
    }
  ],
  "columns": [
    {
      "name": "Product",
      "annotations": [
        {
          "name": "SummarizationSetBy",
          "value": "Automatic"
        }
      ],
      "dataType": "string",
      "isDefaultLabel": true,
      "lineageTag": "da435585-1f9a-44bd-ba2c-34c98f298cfc",
      "sourceColumn": "Product",
      "summarizeBy": "none"
    },
    {…
    },
    {…
    },
    {…
    }
  ],
  "description": "Product Catalog",
  "hierarchies": [
    {…
    }
  ],
  "lineageTag": "e9374b9a-faee-4f9e-b2e7-d9aafb9d6a91",
  "measures": [
    {
      "name": "# Products",
      "expression": "COUNTROWS('Product')",
      "formatString": "#,##0",
      "lineageTag": "1f8f1a2a-06b6-4989-8af7-212719cf3617"
    }
  ],
  "partitions": [
    {
      "name": "Product-171f48b3-e0ea-4ea3-b9a0-c8c673eb0648",
      "mode": "import",
      "source": {
        "expression": [
          "let",
          "    Source = #\"RAW-Product\",",
          "    #\"Renamed Columns\" = Table.RenameColumns(Source,{{\"Prod
          "    #\"Changed Type\" = Table.TransformColumnTypes(#\"Renamed
          {\"Product\", type text}, {\"Manufacturer\", type text}, {\"Bra
          {\"Weight\", type number}, {\"Unit Cost\", type number}, {\"Uni
          type text}, {\"Category Code\", type text}, {\"Category\", type
          "in",
          "    #\"Changed Type\""
        ],
        "type": "m"
      }
```

## TMDL

```
/// Product Catalog
table Product
    lineageTag: e9374b9a-faee-4f9e-b2e7-d9aafb9d6a91

    /// Count of products
    measure '# Products' =
            var v_countProducts = COUNTROWS('Product')
            return v_countProducts
        formatString: #,##0
        lineageTag: 1f8f1a2a-06b6-4989-8af7-212719cf3617

    column Product
        dataType: string
        lineageTag: da435585-1f9a-44bd-ba2c-34c98f298cfc
        isDefaultLabel
        summarizeBy: none
        sourceColumn: Product

    column ProductKey…

    column 'Product Code'…

    column Manufacturer…

    hierarchy 'Product Hierarchy'…

    partition Product-171f48b3-e0ea-4ea3-b9a0-c8c673eb0648 = m
        mode: import
        source =
                let
                    Source = #"RAW-Product",
                    #"Renamed Columns" = Table.RenameColumns(Source,{{"Product Name", "Product"}}),
                    #"Changed Type" = Table.TransformColumnTypes(#"Renamed Columns",{{"ProductKey", Int64.
                    Type}, {"Product Code", type text}, {"Product", type text}, {"Manufacturer", type text},
                    {"Brand", type text}, {"Color", type text}, {"Weight Unit Measure", type text}, {"Weight",
                    type number}, {"Unit Cost", type number}, {"Unit Price", type number}, {"Subcategory
                    Code", type text}, {"Subcategory", type text}, {"Category Code", type text}, {"Category",
                    type text}})
                in
                    #"Changed Type"
```

Concise representation, e.g., no need for name property.

Explicit syntax support for object descriptions

Multi-line expressions in clear text without using any escaping characters.

Use of Indentation to represent tabular object tree and properties.

Representation of child collections without the use of parent properties or delimiters.

# Git and Platforms

- **Git**: Version control system to track and manage changes in files (mainly code).
  - Enables teamwork without overwriting each other's work.
  - Allows reverting to previous versions if something goes wrong.
  - Multiple users can work on the same Power BI file safely.
- **GitHub**: Free cloud platform to store and share projects using Git.
  - Simple, fast, and widely used.
- **Azure DevOps**: Paid platform by Microsoft.
  - Uses Git for project management.
  - Offers additional tools: task tracking, continuous integration, automated deployments, etc.

# Git Configuration

**git config**

**Commands to set up user identity:**

**Set username:**

```
git config --global user.name "Your-Full-Name"
```

**Set email:**

```
git config --global user.email "your-email-address"
```

# Creating a Git Repository

```
git init
```

- **Initializing a Git Repository**

The command git init <repository-name> creates a new repository with the name <repository-name>. It generates a folder named after the repository, which contains a hidden .git folder. This .git folder stores the database that tracks all changes in the repository.

# Clone a Repository

```
git clone
```

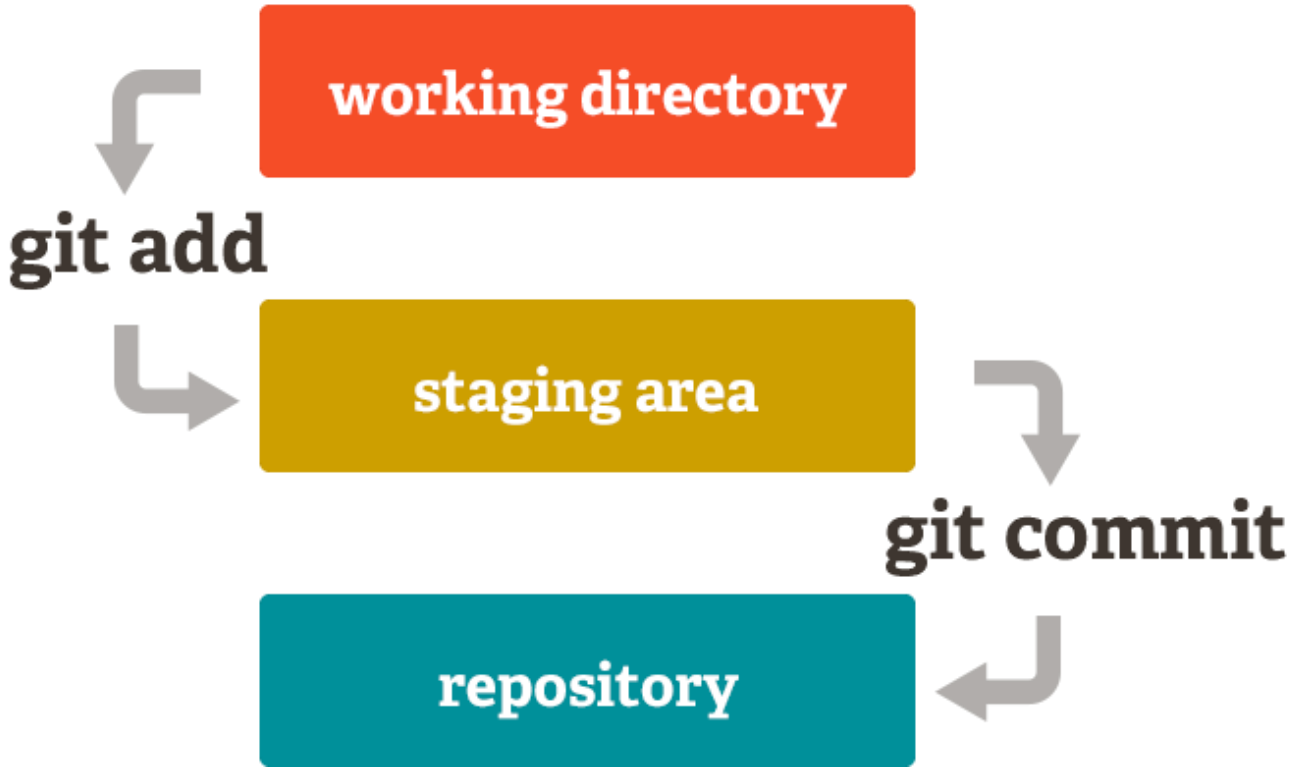
- **Cloning a Git Repository**

The command git clone <repository-URL> creates a local copy of the repository located at <repository-URL>. Once the copy is made, the two repositories—the original and the copy—are independent. Any changes made in one repository will not affect the other.

# Adding Changes to a Git Repository

With Git, any changes made to a project pass through **three stages** before being permanently saved in the repository:

**Working Directory**
The folder containing a copy of a specific version of the project you are working on.
Can include files that are not yet part of the repository.
**Staging Area**
A temporary area where changes from the working directory are placed before being committed.
**Repository**
The place where changes are finally saved (committed) from the staging area.



# Adding Changes to the Staging Area

```
git add
```

- **git add <file>** – Adds changes from the specified file in the working directory to the staging area.
- **git add <folder>** – Adds changes from all files in the specified folder to the staging area.
- **git add .** – Adds all changes from all untracked or modified files in the working directory to the staging area.

# Adding Changes to the repository

```
git commit
```

- **git commit -m "message"** – Confirms all changes in the staging area, adds them to the repository, and creates a new version of the project."message" is a brief description of the changes being committed.

- **git commit --amend -m "message"** – Replaces the message of the last commit with the new "message".

# Change Tracking in Git

Git uses a **three-level structure** to record changes in a repository:

- **Commit**

  Contains information about the author, timestamp, and a message describing the changes.

- **Tree**

  Each commit includes a tree that records the names and paths of the files in the repository at the time of the commit.

- **Blob (Binary File Object)**

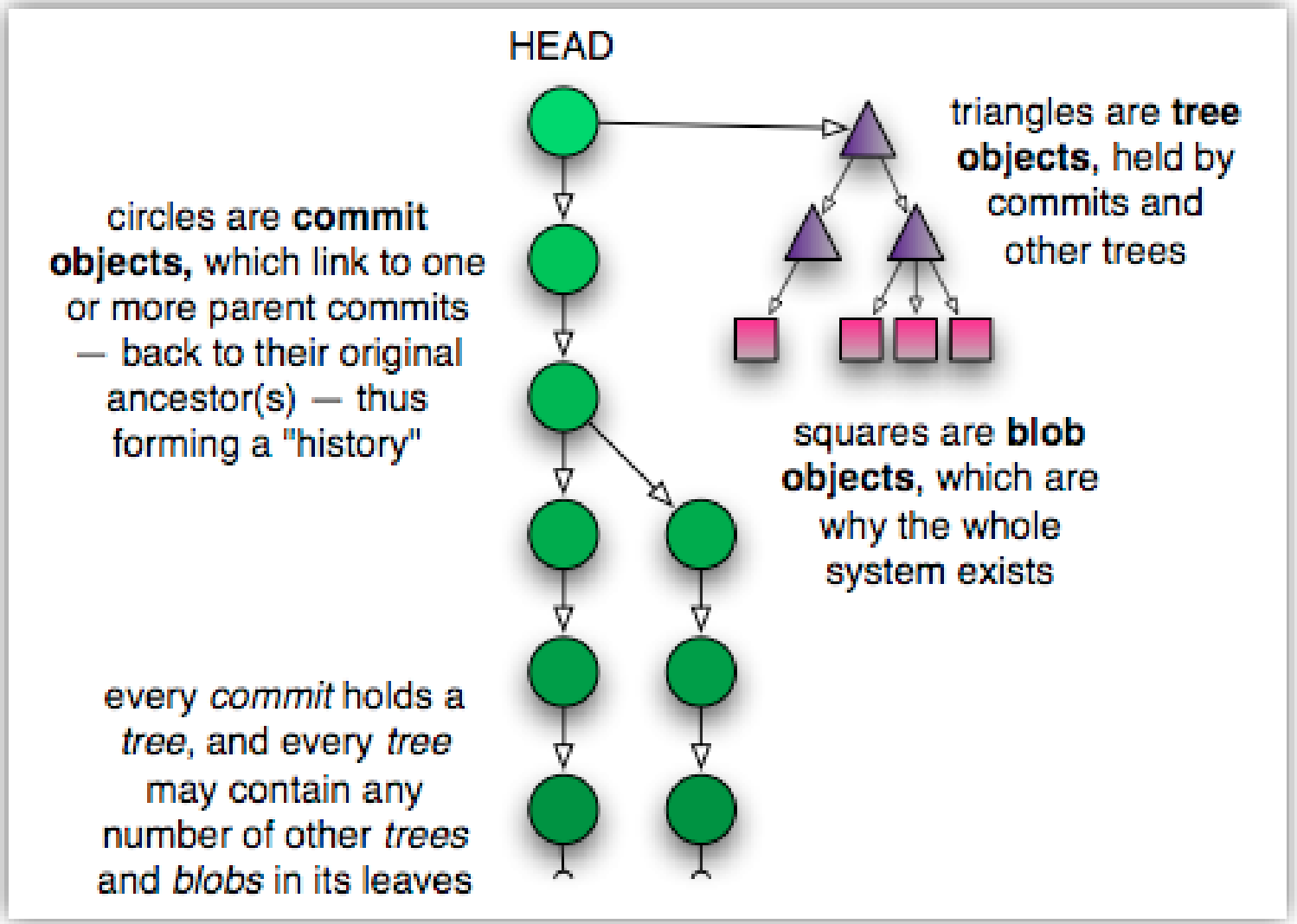
  For each file listed in the tree, there is a blob containing a compressed snapshot of the file's content at the time of the commit.

  If a file hasn't changed since the previous commit, the tree points to the blob from the last commit where the file was modified.

# Referencing a Commit

Each commit has a unique **40-character hexadecimal hash**. There are two ways to reference a commit:

- **Absolute Name**
  - Uses the commit's hash code.
  - Usually, the first 4–5 characters are enough to identify it.
- **Relative Name**
  - Uses the keyword HEAD to refer to the latest commit.
  - To refer to earlier commits:
    - HEAD~1 → previous commit
    - HEAD~2 → commit before that, and so on.



# Checking the Status of a Repository

```
git status
```

The command git status shows the current state of changes in the repository since the last saved version. Specifically, it displays:

- Files in the working directory that have been modified but **not yet added** to the staging area.
- Files in the staging area that have been **staged but not yet committed** to the repository.

# Viewing Commit History

```
git log
```

The command git log displays the commit history of a repository in chronological order. For each commit, it shows:

- Commit hash
- Author
- Date and time
- Associated commit message

**Common options:**

- --oneline – Shows each commit in a single line for a more compact view.
- --graph – Displays the commit history as a graph.

# Viewing Commit Details

```
git show
```

- **git show** – Displays the author, date, time, and message of the latest commit, along with the differences compared to the previous commit.

- **git show <commit>** – Displays the author, date, time, and message of the specified commit, along with the differences compared to the previous commit.

# Viewing Differences Between Versions

```
git diff
```

**Viewing Differences Between Versions**

- **git diff** – Shows the differences between the working directory and the staging area.
- **git diff --cached** – Shows the differences between the staging area and the last commit.
- **git diff HEAD** – Shows the differences between the working directory and the last commit.

# Undoing Changes

`git checkout`

**Revert changes in the working directory or restore a previous version**

**git checkout <commit> -- <file>** – Updates the file <file> to the version in the specified <commit>.

Common use: discard changes in a file that haven't been staged yet:

- git checkout HEAD -- <file> restores the file to the last committed version.

# Removing Changes from the Staging Area

```
git reset
```

**git reset <file>**

•Unstages the file <file>, meaning it removes it from the staging area.
•The changes in the working directory are **preserved** (the file remains modified locally).

**Completely discarding changes to a staged file:**

1.First, run git reset <file> to remove it from the staging area.
2.Then, run git checkout HEAD -- <file> to restore the file to its last committed version (deleting all local modifications).

# Branches in Git

- A **branch** is like a separate line of development in a project.

- Every new repository starts with a default branch, usually called **main** (or sometimes **master**).

- Branches let you work on **new features, fixes, or experiments** without affecting the main project.
- While you work in your branch, the main branch stays stable.

- When the work is ready, the branch can be **merged** into the main branch so the new changes become part of the project.

# git branch

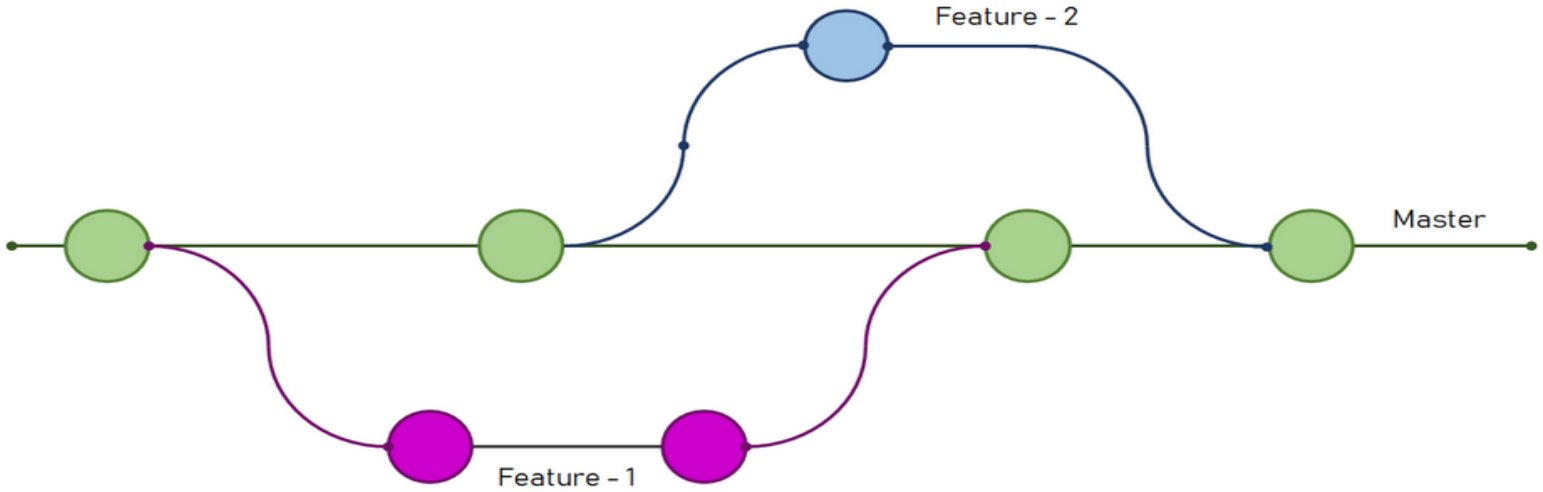
**git branch <branch-name>** This command **creates a new branch** called <branch-name> in the repository.

The new branch starts from the **latest commit**, i.e., the point where HEAD is currently pointing.

From that commit, the history of commits **splits into two lines**:

-The original branch continues its normal development.
-The new branch allows parallel development on a separate line.

This means you can work on **two versions of the project at the same time** without interfering with each other, and later merge them if needed.



# Listing Branches

```
git log
```

- **git branch** – Lists all the branches in the repository and marks the current active branch with an asterisk *.

- **git log --graph --oneline --all** – Displays the repository history as a graph (--graph) in a compact format (--oneline), including all branches (--all).

# Switching Branches

`git checkout`

- **git checkout <branch>** – Updates the files in the working directory to match the latest version of **<branch>** and makes it the active branch. In other words, HEAD now points to the latest commit of that branch.

- **git checkout -b <branch>** – Creates a new branch named **<branch>** **and** switches to it immediately.

Equivalent to running:

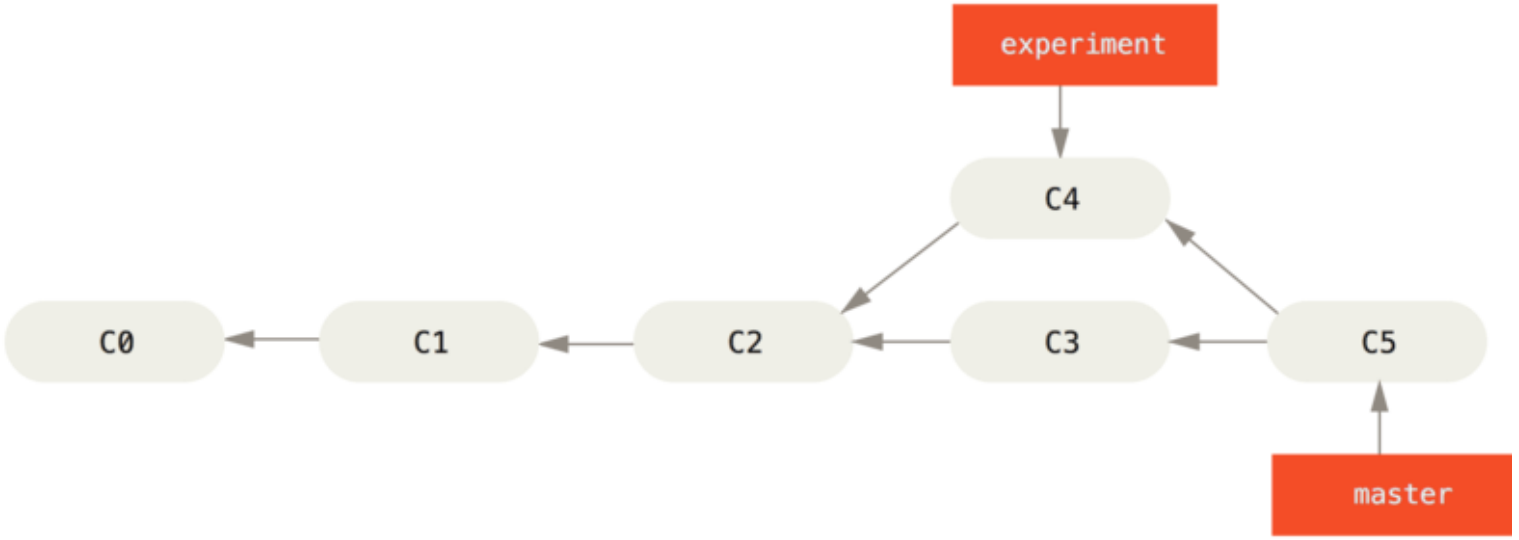
1.git branch <branch>

2.git checkout <branch>

# git merge

**Merging Branches:**
**git merge <branch>** – Integrates the changes from <branch> into the current branch (the one HEAD is pointing to).

This allows you to bring together the work done in different branches into a single unified history.



# Conflict Resolution

When merging two branches, Git requires that there are **no conflicts** between the changes.

A **conflict** occurs when both branches modify the same part of a file in different ways.

In that case, Git cannot decide which version to keep, so you must resolve it manually:

1-Review the conflicting changes.

2-Decide which code should remain (or combine both).

Tools like **KDiff3** or **Meld** can help simplify the conflict resolution process.

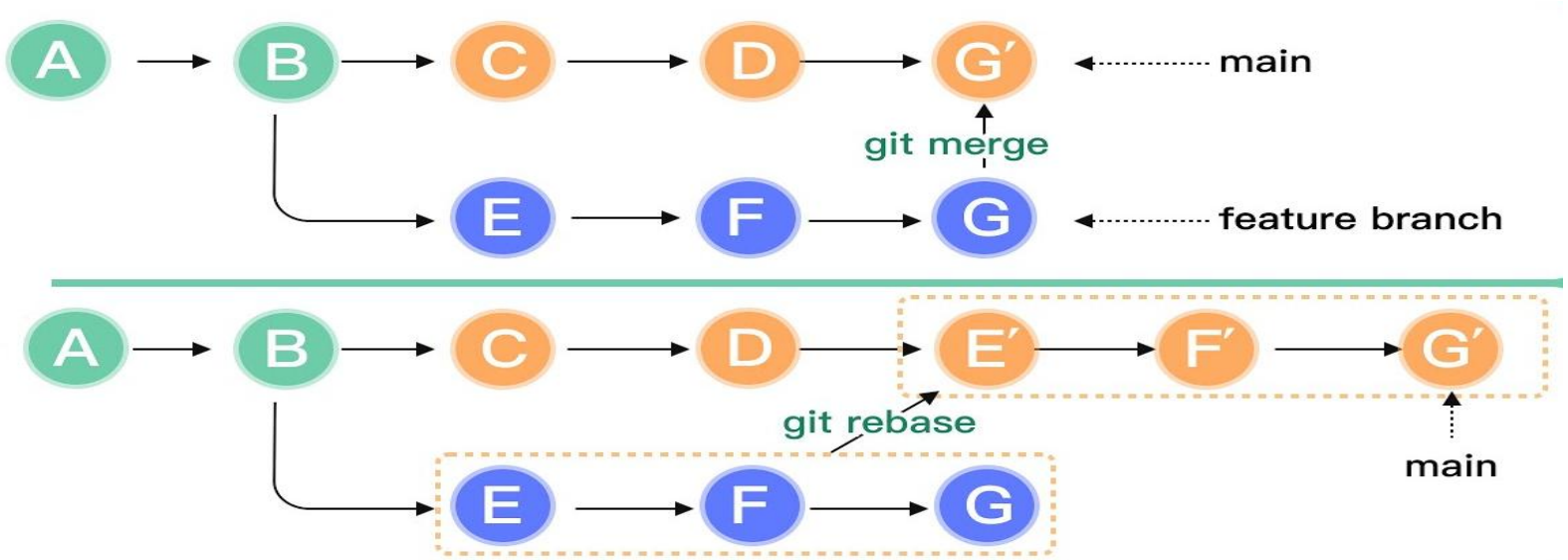
# git rebase

**What it does:**

-Applies changes from <branch-2> onto <branch-1>
-Starts from the common ancestor of both branches

**Result:**

-Similar to merging branches
- <branch-2> fork disappears
-Commits from <branch-2> now appear in <branch-1>



# Listing Branches

```
git branch -d
```

**git branch -d <branch>** deletes the branch named <branch> **only if it has been previously merged**.

**git branch -D <branch>** deletes the branch named <branch> **even if it hasn’t been merged**. Any unmerged changes on the branch will be lost.

# Remote Repositories & Platforms

- **Remote Repository**

  A **remote repository** is a version of your Git repository hosted on an online server.

  It **enables multiple collaborators** to work on the project simultaneously.

  Serves as a **central backup**, ensuring your work is safe and accessible from anywhere.

- **GitHub**

  **GitHub** is a widely-used platform for hosting Git repositories.

  Provides **collaboration tools** such as pull requests, code reviews, and issue tracking.

  Facilitates **project management** with built-in features like milestones and project boards.

- **Azure DevOps**

  **Azure DevOps** is Microsoft's integrated development service that hosts Git repositories.

  Includes **CI/CD pipelines**, boards, and artifact management for end-to-end DevOps.

  Designed for **team collaboration**, offering tools for planning, tracking, and deploying software.


Azure **DevOps**

# Adding a Remote Repository

```
git remote add
```

The command **git remote add <remote-name> <url>** creates a connection between your local repository and a remote repository hosted online. Once added, Git tracks changes in the remote repository, allowing you to **pull** updates from it and **push** your own changes back. This keeps your local and remote repositories synchronized and enables smooth collaboration with other developers.

List remotes:

**git remote** ,show all the remote connections defined in your local repository.

List remotes with URL'S:

**git remote –v** ,display the URLs for each remote, helping you verify where your repository is connected.

# Downloading Changes from a Remote Repository

```
git pull
```

- **git pull <remote> <branch>**

Downloads changes from the <branch> of the remote repository <remote> and integrates them directly into your local repository's current branch (HEAD).

- **git fetch <remote>**

Downloads changes from the remote repository <remote> but does not integrate them into your local branch. This allows you to review changes before merging.

# Pushing Changes to a Remote Repository

```
git push
```

The command **git push <remote> <branch>** is used to upload the changes from your local branch to a remote repository. This updates the remote repository with your latest commits, making your work available to other collaborators. Pushing ensures that the remote repository stays synchronized with your local development and allows teams to share progress efficiently.


# Azure DevOps

Azure DevOps is a set of cloud-based services from Microsoft designed to support the entire software development lifecycle. It combines Git repositories, planning tools, automation, and deployment in a single collaborative environment.

**Main Uses:**

- **Git Repositories:** securely store and manage source code.
- **Boards:** plan and track tasks, sprints, and agile projects.
- **Pipelines (CI/CD):** automate building, testing, and deploying applications.
- **Artifacts:** manage packages and project dependencies.
- **Test Plans:** organize and run manual or automated tests.

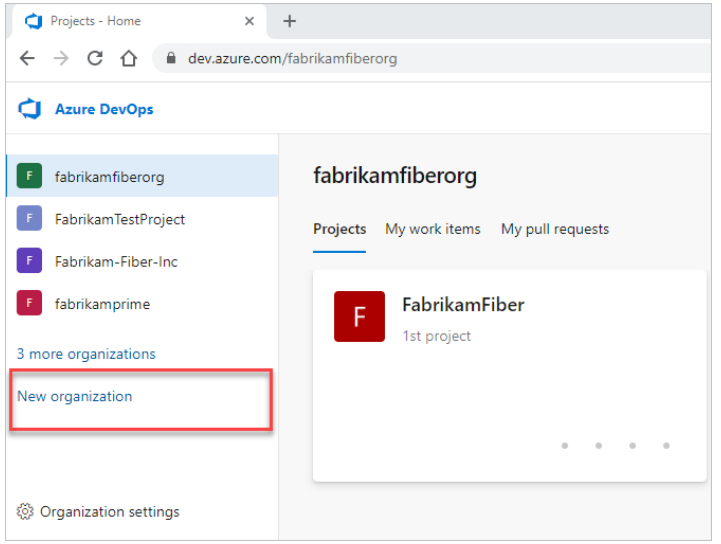
# Azure DevOps Organizations

An **organization in Azure DevOps** is the top-level container that groups together projects, users, and resources. It provides a central space where teams can collaborate, manage permissions, and access all Azure DevOps services.

**Main Uses:**

- Centralize **projects** under one structure.
- Manage **users and permissions** across teams.
- Provide access to **repositories, pipelines, boards, and artifacts**.
- Enable collaboration and resource sharing across multiple projects.

# Creating an Organization in Azure DevOps




1.Sing in to Azure DevOps.

2.Select **New Organization**.

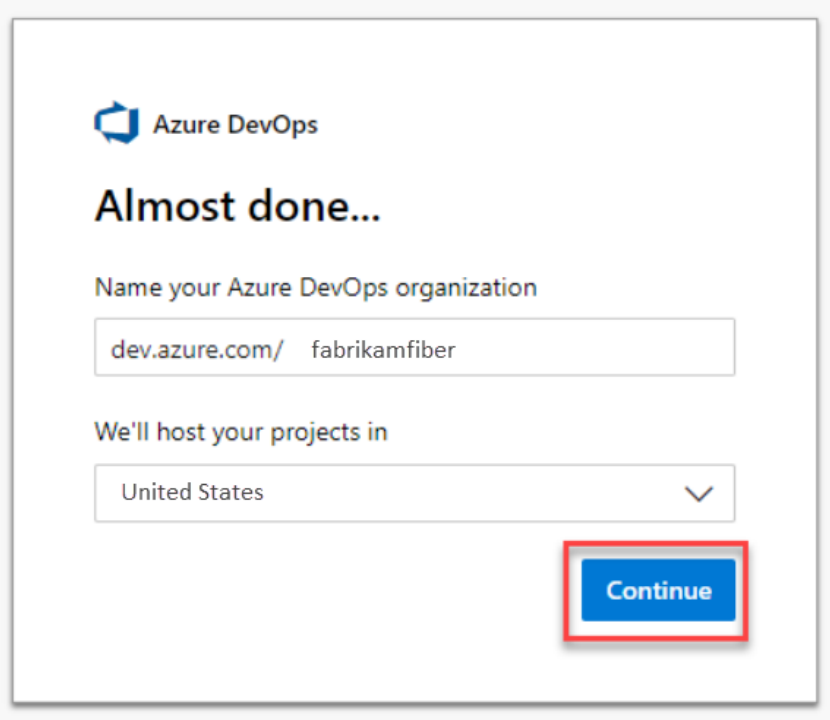
**3.Enter the organization name**, choose the hosting region, then click **Continue**.

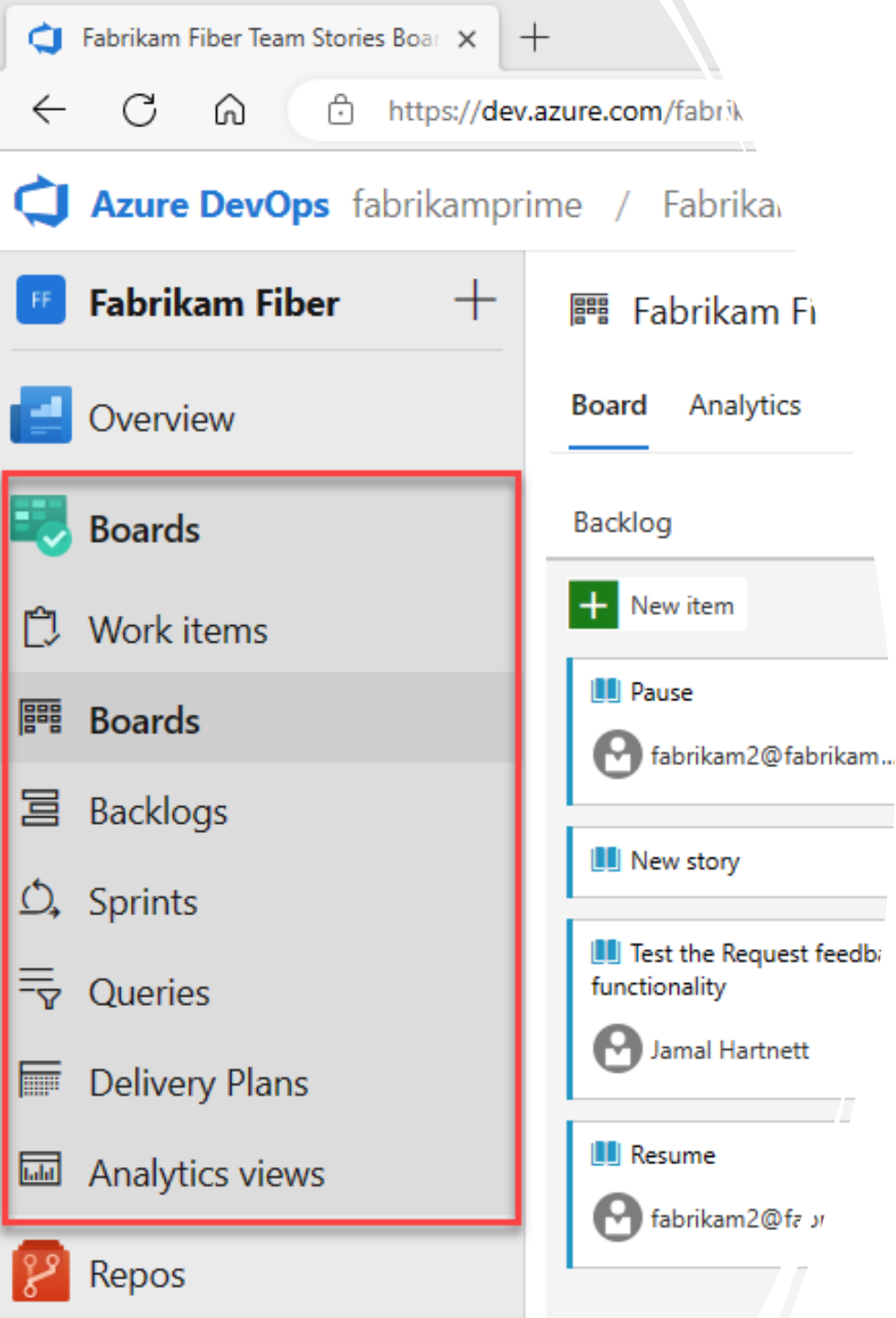
- Naming rules: only English letters/numbers, max 50 characters, must start and end with a letter or number.

4.Once created, your organization will be available at: https://dev.azure.com/{yourorganization}

- **Invite users** to the organization so they can join projects and collaborate.

Note: Organizations can only be created manually through the web portal.






# Azure Boards

Azure Boards is a **web-based service** for planning, tracking, and analyzing work throughout the software development lifecycle. It supports agile methodologies and provides customizable tools for managing work items, collaborating across teams, and streamlining workflows.

**Key Features:**

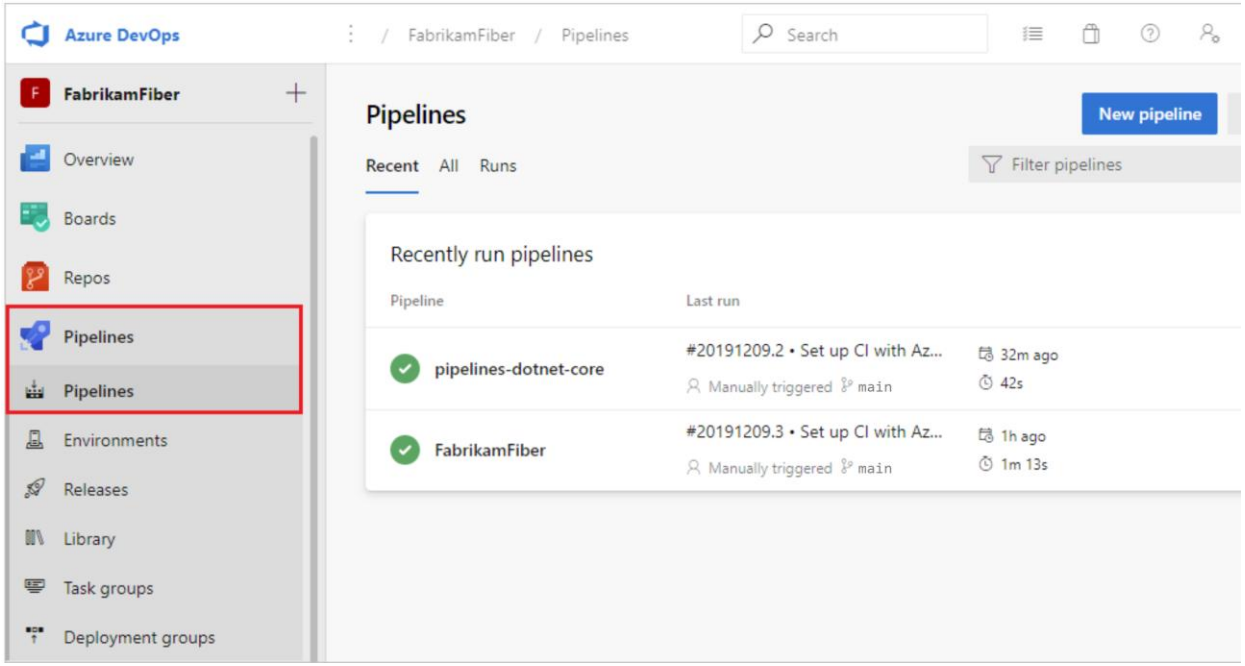
- **Work Items:** Track tasks, bugs, and user stories.
- **Boards:** Visualize work as cards and update status using Kanban-style boards.
- **Backlogs:** Plan, prioritize, and organize work items for projects or portfolios.
- **Sprints:** Manage work items for specific iterations or timeframes.
- **Queries & Analytics:** Filter and report on work items, integrate with Power BI.

# Azure Pipelines

Azure Pipelines is the part of Azure DevOps that enables **continuous integration (CI), continuous testing, and continuous delivery (CD)**. It automatically builds, tests, and deploys code to any target, supporting all major languages and platforms, whether applications are on-premises or in the cloud.

**Key Features and Benefits:**

- **Continuous Integration (CI):** Automatically combines and tests code to detect errors early.
- **Continuous Testing:** Run automated tests in various frameworks to ensure code quality.
- **Continuous Delivery (CD):** Build, test, and deploy applications to multiple environments, including cloud, on-premises, and mobile app stores.
- **Multi-platform Support:** Works with Windows, Linux, and macOS, and supports languages like Node.js, Python, Java, C#, C++, Go, .NET, Android, iOS, and more.
- **Integration with Repositories:** Supports GitHub, Azure Repos, and other version control systems.
- **Package Management:** Publish build artifacts and packages (NuGet, npm, Maven, Python) to internal or external repositories.
- **Scalable & Automated:** Run parallel jobs, monitor builds, and generate actionable reports.



# Azure Pipelines Agents – Overview & Types

An **agent** is a computing resource that executes jobs in Azure Pipelines. Each job runs on one agent at a time. Agents are required to build, test, and deploy code.

**Types of Agents:**

- **Microsoft-hosted Agents:** Managed by Microsoft, automatically updated, disposable virtual machines.
- **Self-hosted Agents:** Installed and managed by you on Windows, Linux, macOS, or in Docker containers; provide more control and caching.
- **Azure VM Scale Set Agents:** Self-hosted agents that automatically scale to meet demand.
- **Managed DevOps Pools:** Fully managed pools that simplify scaling, security, and maintenance.

# Key Concepts & Agent Usage

**Parallel Jobs:** Number of jobs that can run simultaneously; more jobs require more agents.

**Capabilities & Demands:** Define the required software or features for an agent to run specific jobs.

**Communication:** Agents communicate with Azure Pipelines via HTTP/HTTPS, download jobs, execute them, and report status.

**Interactive vs. Service Mode:** Run agents interactively for testing or as a service for production.

**Directory Structure:**

- **Installation Directory:** Where the agent software resides.
- **Working Directory:** Stores source code, binaries, and artifacts during job execution.