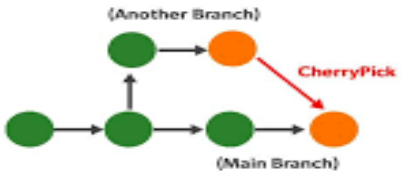
GIT

1. **Git Cherry-Pick:** Cherry-pick is the act of picking a commit from a branch and applying it to another. **Cherry-pick** is a way to take a specific change (commit) from one branch in Git and apply it to another branch. It's like picking just one fruit (the commit) from a tree (the branch) instead of taking the whole tree (merging the entire branch).

   git cherry-pick <commit-id>

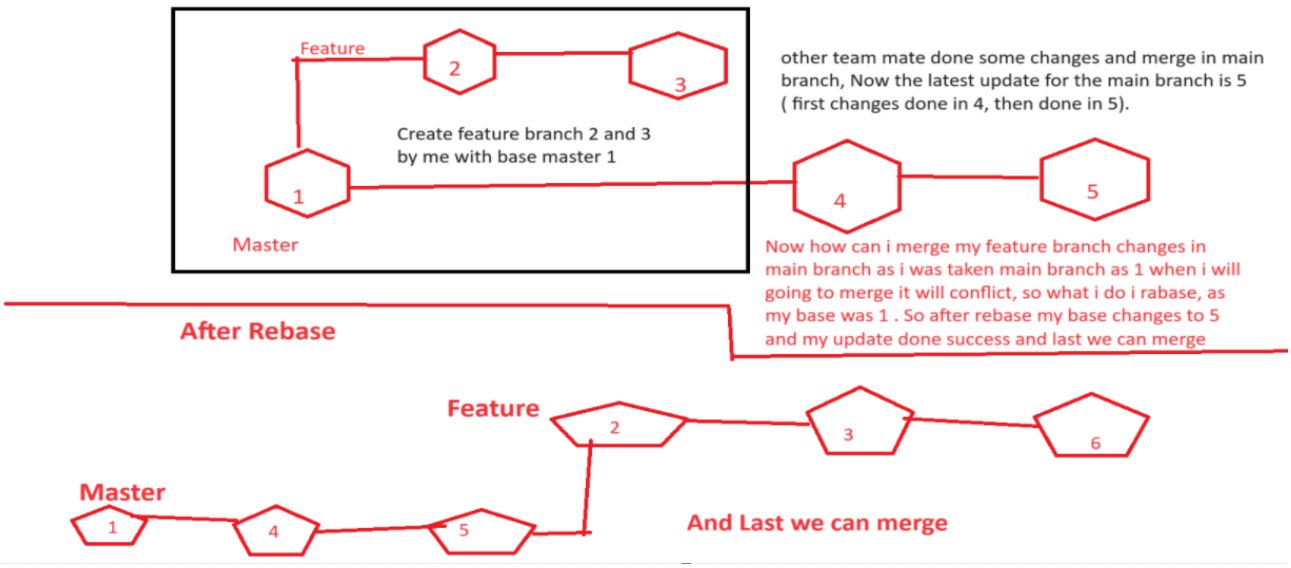


Cherry-Pick ek Git command hai jo aapko ek specific commit ko aapke current branch par apply karne ki suvidha deti hai. Iska use tab hota hai jab aap kisi aur branch se sirf ek ya kuch specific commits ko le kar aana chahte hain, bina poore branch ko merge kiye.

2. **Git Rebase:** Git rebase is a way to move or combine a sequence of commits to a new base commit. Think of it like taking a branch of a tree and reattaching it to a different spot on the main trunk. This helps keep your project's history clean and linear.

   git rebase <upstream>

   git rebase main



Git Rebase ek command hai jo aapko ek branch ke commits ko doosri branch ke upar "reapply" karne ki suvidha deti hai. Iska matlab hai ki aap apne changes ko ek naye base par laga sakte hain, jaise ki ek purani gaadi ko naye model par chadha dena.

3. Git Stash:

Git stash is a command that helps you temporarily save your changes (the work you've done but haven't committed yet) so that you can switch to another task or branch without losing your progress.

When Do You Use Git Stash?

- Switching Tasks: When you're in the middle of working on something and suddenly need to fix a bug or work on a different feature.
- Cleaning Up: If you want to pull the latest changes from the main branch but have uncommitted changes that you don't want to commit yet.

How Does It Work?

- **Stash Your Changes:** When you're ready to save your work temporarily, you run:
  git stash
- **Switch Branches:** After stashing, you can switch to another branch or pull updates without any issues.
- **Bring Back Your Changes:** When you're ready to continue your work, you can restore your stashed changes with:
  git stash apply

Git Stash ek command hai jo aapko apne kaam ko temporarily save karne ki suvidha deti hai, bina unchanges ko commit kiye. Ye tab kaam aata hai jab aap kisi aur branch par jaana chahte hain, lekin aapke paas kuch uncommitted changes hain jo aap nahi chahte ki abhi commit karein.

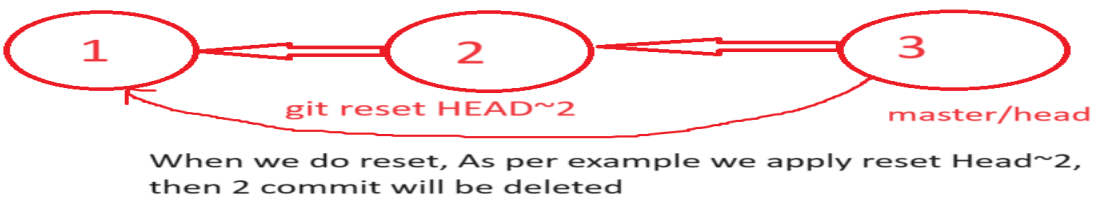
4. Git fetch vs Git Pull:

Feature	git fetch	git pull
Definition	Updates your local repository with changes from the remote without merging.	Fetches changes from the remote and merges them into your current branch.
Action	Only downloads the updates.	Downloads updates and applies them to your current branch.
Working Files	Your local working files remain unchanged.	Your local working files may change after the merge.
Use Case	When you want to see what's new without making any changes to your branch.	When you want to update your branch with the latest changes from the remote.
Command Example	git fetch	git pull
Best For	Checking for updates and reviewing them before merging.	Quickly updating your branch with the latest changes.
Conflicts	No conflicts will occur since it doesn't merge.	Conflicts may arise if there are conflicting changes during the merge.

5. Git Reset:

Reset means moving the position of head pointer to the previous commit.

git reset HEAD~2



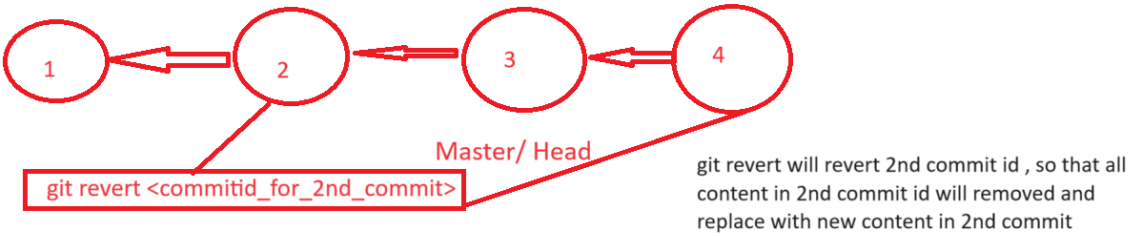
When we do reset, As per example we apply reset Head~2, then 2 commit will be deleted

Git reset ek command hai jo aapko Git repository mein changes ko undo karne ki suvidha deti hai. Iska matlab hai ki aap apne current branch pointer ko kisi purane commit par le ja sakte hain, jisse aapka working directory aur staging area uss point par "reset" ho jata hai.

6. Git Revert:

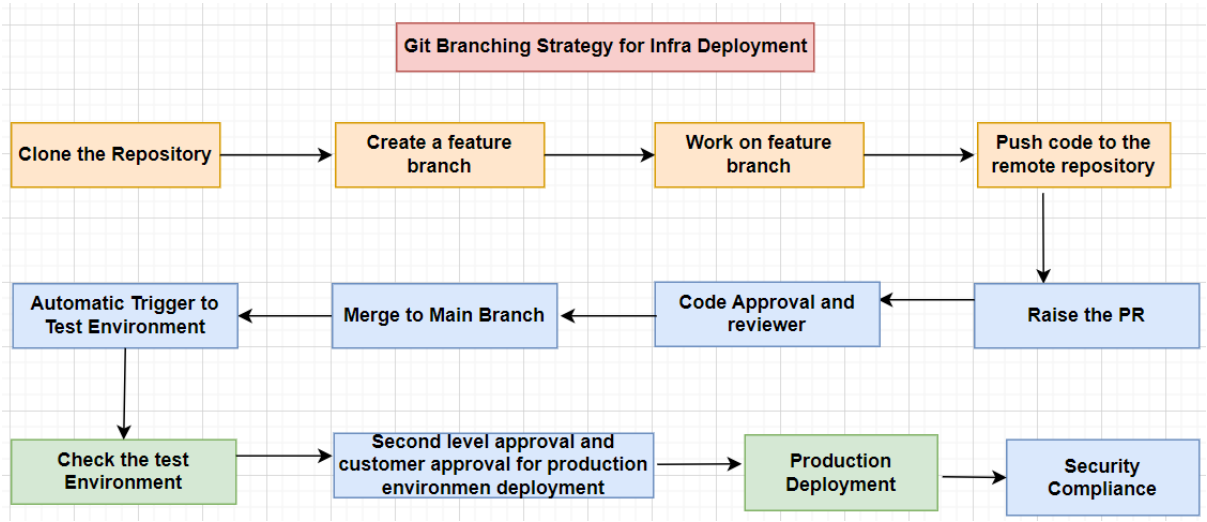
In case of Revert, any commit not get deleted, it just create the new commit which revert the commit which we want to revert.

Git revert commitid



7. Git Branching Strategy:

a. Terraform Deployment Process



1. Clone the Repository

- First, we make a copy of our Terraform code from GitHub or Azure Repos to our computer. This ensures we have the latest version to work on.

2. Create a Feature Branch

- We then create a new branch specifically for the feature or change we want to work on. This keeps our changes separate from the main code.

3. Work on the Feature

- Now, we write the code for the new feature in our feature branch. We can also test our changes locally by using terraform plan to see what will happen.

4. Push Code to the Remote Repository

- After finishing our work, we send our changes back to GitHub or Azure Repo.

5. Raise a Pull Request (PR)

- Next, we create a Pull Request to ask our technical lead to review our code. This is where they can check our changes and give feedback.

6. Code Review and Approval

- Our technical lead and other reviewers look at the code. They might ask for changes or approve it if everything looks good.

7. Merge to Main Branch

- Once approved, we combine our feature branch with the main branch. This is where our new feature becomes part of the main codebase.

8. Automatic Trigger for Test Environment Deployment

- Merging the code automatically starts the CI/CD pipeline. This means:
    - Tests run to check if everything works.
    - The new infrastructure gets deployed to a test environment using Terraform.

9. Check the Test Environment

- After deployment, we check the test environment to ensure everything is set up correctly. We look for any issues and fix them if needed.

10. Second Level Approval and Customer Approval

- Once testing is complete, we ask for a second round of approval from stakeholders or the customer.
- This could involve sending notifications or having a quick meeting.

11. Production Deployment

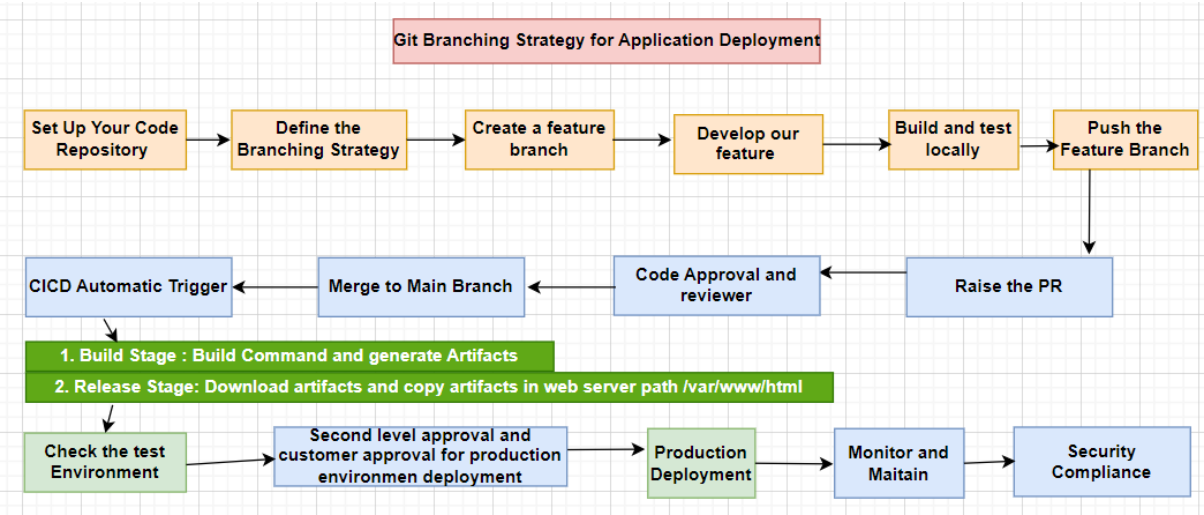
- After all approvals are in place, the pipeline will deploy the changes to the production environment, which is where the live application runs.

12. Security Compliance

- Throughout the process, we follow security rules. Any sensitive information, like passwords, is stored safely in Azure Key Vault, which keeps it secure and hidden from the code.

b. Monolithic Application Deployment Process

Deploying a Java-based application using a CI/CD pipeline with a Git branching strategy involves several organized steps to ensure efficient collaboration, code quality, and seamless deployment. Here's a detailed guide that integrates Git branching practices into the CI/CD process.



1. Set Up Your Code Repository (Developer put java code in git/azure repo)

- Version Control: Ensure your Java application is stored in a version control system like Git (GitHub, GitLab, Bitbucket, etc.).
- Repository Structure: Organize your project with standard Java project structure (e.g., src, lib, resources, etc.).

2. Define the Branching Strategy

Choose a branching strategy that fits your team's workflow. Common strategies include:

- Feature Branching: Each new feature is developed in its own branch, named feature/your-feature-name.
- Git Flow: This involves multiple branches: main, develop, and feature/release branches.
- GitHub Flow: Simple and lightweight, focusing on a main branch and feature branches.

3. Create a Feature Branch

1. Clone the Repository:
    - Clone your repository to your local machine.
2. Create and Checkout a Feature Branch:
    - Create a new branch for your feature or bug fix.

4. Develop Your Feature

- Work on your Java application in this feature branch.
- Implement the required changes and ensure that you follow coding standards.

5. Build and Test Locally

- Before pushing your changes, build and test your application locally.
- Use Maven or Gradle to build the project and run tests.
    - Maven:

      mvn clean install
    - Gradle:

      ./gradlew build

6. Push the Feature Branch

- Once your changes are complete and tested locally, push your feature branch to the remote repository.

7. Create a Pull Request (PR)

- After pushing, create a Pull Request (PR) to merge your feature branch into the develop or main branch.
- Include details in the PR description about what changes were made and why.

8. Code Review and Approval

- The PR is reviewed by your team members or technical lead.
- They can provide feedback, request changes, or approve the PR.
- Once approved, the PR can be merged into the develop or main branch.

9. Trigger CI/CD Pipeline

- The merge triggers the CI/CD pipeline. This typically involves:
    - Build Stage: Compile the code and run tests automatically.
    - Deployment Stage: Deploy the application to a test or staging environment.

Example CICD Pipeline Steps:

1. Build Stage:
    - Compile the application using Maven or Gradle.
    - Run unit tests.
2. Deployment Stage:
    - Use scripts or tools to deploy the Java application to your existing infrastructure (e.g., a Tomcat server).

10. Check the Test Environment

- Validate the deployment in the test environment.

- Perform functional tests to ensure the application works as expected.

11. Approval for Production Deployment

- Once testing is successful, you may require additional approvals for production deployment.
- Follow the same PR process for merging into the main branch (if using Git Flow).

12. Production Deployment

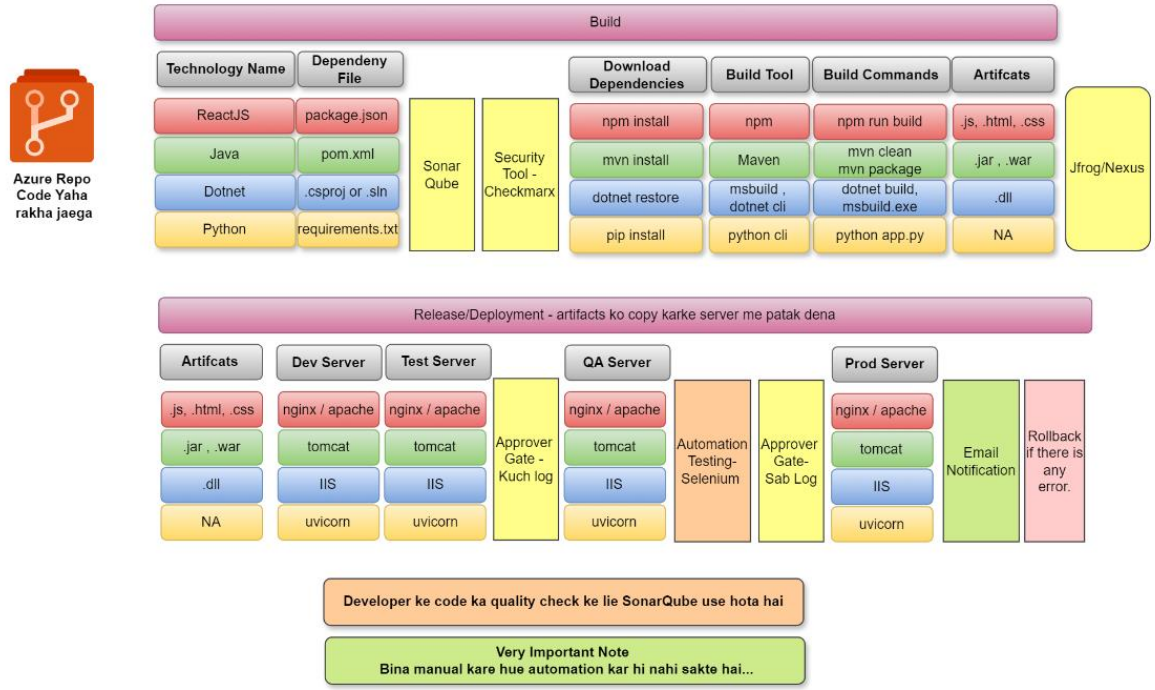
- Once all approvals are in place, deploy the application to the production environment.
- Ensure that you have rollback procedures in case of issues.

13. Monitor and Maintain

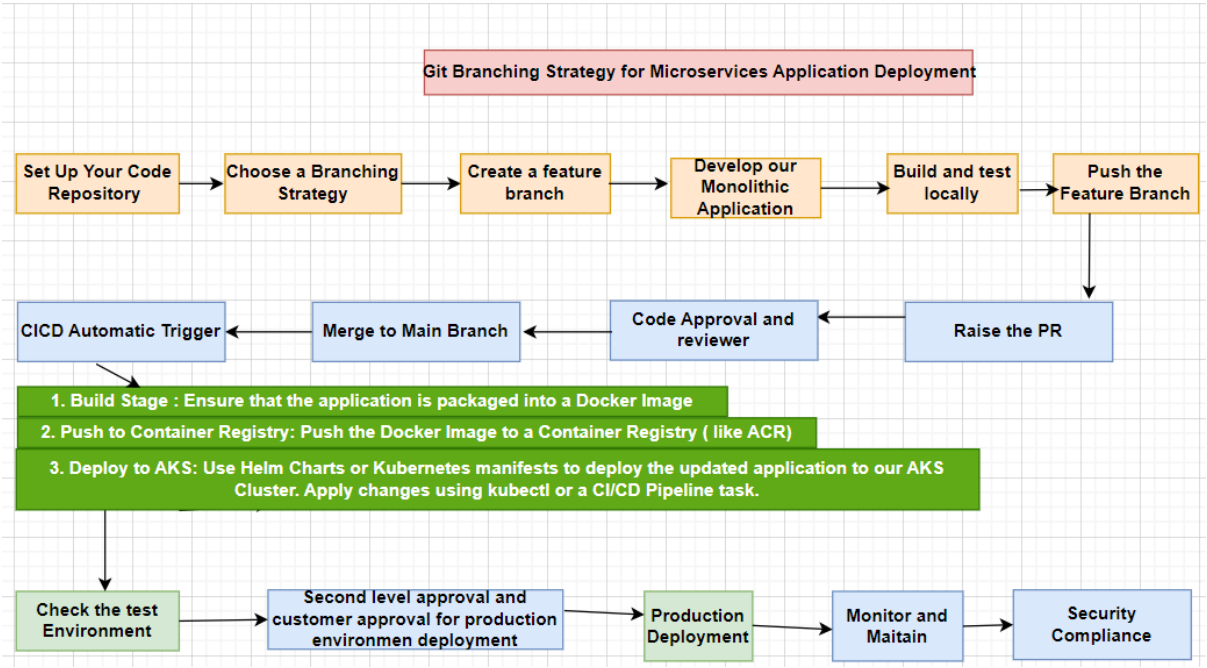
- After deployment, monitor the application's performance using monitoring tools (like Prometheus, Grafana).
- Address any issues that arise and plan for the next set of features or fixes.

14. Security Compliance

- Ensure sensitive information is securely managed (e.g., using Azure Key Vault or AWS Secrets Manager).
- Follow best practices for code security and deployment.



c. Microservices Application Deployment Process using AKS



1. Set Up Your Code Repository

- Use a version control system like Git to manage your monolithic application's code. Structure the repository clearly, including directories for source code, resources, and configuration files.

2. Choose a Branching Strategy

For a monolithic application, a common branching strategy includes:

- **Main Branch**: The stable version of your application that is ready for production.
- **Develop Branch**: Where ongoing development occurs. This branch contains the latest features that are tested and ready for integration.
- **Feature Branches**: Created for new features or bug fixes. Each feature or fix is developed in its own branch, named something like feature/your-feature-name.

3. Create a Feature Branch

When starting work on a new feature or bug fix, create a new feature branch:

4. Develop Your Monolithic Application

- Make the necessary changes to your code within the feature branch.
- Ensure that your application maintains its modularity to facilitate easier future updates.

5. Build and Test Locally

- Before pushing your changes, build and test your application locally to ensure everything works correctly. You might use a build tool like Maven, Gradle, or npm, depending on your stack.

6. Push the Feature Branch

- Once your changes are complete and tested, push your feature branch to the remote repository:

7. Create a Pull Request (PR)

- After pushing, create a Pull Request to merge your feature branch into the develop branch.
- Include details in the PR description about what changes were made and why.

8. Code Review and Approval

- The PR is reviewed by team members or technical leads. They may provide feedback or approve the changes.
- Once approved, merge the feature branch into the develop branch.

9. Trigger CI/CD Pipeline

Merging the PR triggers the CI/CD pipeline. You can use Azure DevOps, Jenkins, or another CI/CD tool for this process.

CI/CD Pipeline Steps:

1. **Build Stage**:
    - Compile the application and run automated tests.
    - Ensure that the application is packaged into a Docker image.
2. **Push to Container Registry**:
    - Push the Docker image to a container registry (like Azure Container Registry).
3. **Deploy to AKS**:
    - Use Helm charts or Kubernetes manifests to deploy the updated application to your AKS cluster.
    - Apply changes using kubectl or a CI/CD pipeline task.

10. Check the Test Environment

- Validate the deployment in the AKS test environment. Perform functional tests to ensure the application is running correctly.

11. Approval for Production Deployment

- After successful testing, seek approval from stakeholders for deploying the application to the production environment.

12. Production Deployment

- Once approved, deploy the application to the production environment in AKS.
- Monitor the deployment process to ensure everything goes smoothly.

13. Monitor and Maintain

- Use monitoring tools (like Azure Monitor, Prometheus, Grafana) to keep an eye on the application's performance and health.
- Address any issues that arise and plan for future features or fixes.

14. Security and Compliance

- Ensure sensitive information (e.g., API keys, database passwords) is managed securely, possibly using Azure Key Vault.
- Follow best practices for security and compliance in your application deployment.