**Trunk-Based Development** (TBD)

Trunk-based development is a software development strategy where all developers commit code changes to a single shared branch, known as the “trunk” or “main,” rather than working in separate branches for extended periods. This approach minimizes the complexity of merging code and aims to accelerate development by encouraging small, frequent commits to the main codebase.

**Key Concepts of Trunk-Based Development:**

1. **Main (Trunk) Branch**:
   * The **trunk** (or main branch) is the central branch where all the developers' changes are integrated.
   * The idea is to ensure that the trunk is always in a **deployable state**.
   * All developers work on a single trunk (main) branch.
   * The trunk is continuously integrated, meaning that developers must merge small changes frequently (multiple times a day).
2. **Short-Lived Branches**:
   * Developers might create short-lived feature branches, typically lasting a few hours to a day, before merging back into the main branch.
3. **Continuous Integration**:
   * Each commit on the trunk triggers automated builds, tests, and deployments to ensure code quality and stability.
   * Developers ensure that the trunk is always in a deployable state.
4. **Feature Flags**:
   * Instead of long-lived branches, unfinished or experimental features are hidden behind **feature flags**. This allows the feature to be merged into the main branch without being fully functional or exposed to users.
     1. **Handling Unfinished Features**: In TBD, developers often push code for incomplete or experimental features into the main branch. While this is part of the process, it can make your application unstable if not managed carefully.  
        **Solution**: Feature flags let us add unfinished features to the main branch without making them visible to users. This allows you to keep building and testing features in real environments without affecting the end user.
     2. **Managing Complex Feature Interactions**: When many features are developed and merged at the same time, it can lead to unexpected bugs or issues caused by how the features interact with each other.  
        **Solution**: Feature flags allow us to turn features on or off individually, making it easier to focus on specific features and troubleshoot any problems. This helps you quickly identify and fix issues caused by feature interactions.
     3. **Quickly Disabling Problematic Features**: Sometimes new features cause problems after they are deployed, even if they passed testing. Rolling back changes can be slow and risky.  
        **Solution**: Feature flags give us an easy way to turn off problematic features without rolling back the whole deployment. This lets you fix the issue quickly, without disrupting other parts of the application.

**What if we have multiple working environment?**

Since only one branch is responsible for deployment i.e., trunk/main branch. So how in real world it will be deployed to DEV, then QA, then Production. if we are using the same branch for everything environment and any change happen with the branch then it will trigger the QA and Prod also. So how it actually works?

* Robust CI/CD Pipelines for Each Environment
* Feature Flags setup for each and every env
* Environment-Specific Configurations for build, deploy and testing and pipeline trigger mechanism (Manual/conditional deployment)
* Production Deployment strategy –
  + Blue-Green Deployments
  + Canary Releases

An example flow:

**Step 1: Developers Commit to Trunk**

* Developers commit small, incremental changes to the **trunk** branch.
* **CI Pipeline** runs for each commit:
  + Unit tests are executed.
  + Some static code analyses are performed and some basic tests executed.

**Step 2: Automatic Deployment to Dev**

* Once a commit is merged into the trunk, the **CI/CD pipeline** automatically deploys the latest code to the **Dev environment**.
  + This allows for early-stage testing and validation in a realistic environment.
  + Developers can run integration tests and manually test new features in Dev.

**Step 3: Promoting to QA**

* After testing in Dev, the same trunk code is **promoted** to the **QA environment**, but it doesn’t happen automatically after every commit.
* Some common strategies for promoting code to QA:
  + **Manual Trigger**: Team decides when the code is stable enough and promotes it to QA.
  + **Tagging a Build**: Once a build in Dev passes certain tests, it is tagged (e.g., v1.2.3-qa) and that specific tagged version is deployed to QA for further testing.
  + **Scheduled Deployment**: Deployments to QA may occur at scheduled intervals (e.g., nightly builds), allowing for regular testing cycles.

**Step 4: Deployment to Production**

* **Code in Production** should be stable and fully tested. Therefore, deployments to production are often gated:
  + **Manual Approval**: After the code passes all tests in QA, a manual approval is required before promoting the same code to Production.
  + **Tagged Releases**: Code is deployed to production only when it is tagged with a **release version** (e.g., v1.2.3-prod).
  + **Feature Flags**: Even if the code reaches Production, certain features might be behind feature flags and not visible to users until fully tested or approved.

**Some benefits of this strategy**:

* **Faster Integration and Delivery**
* **Improved Code Quality and Stability**
* Feature Flexibility with Feature Flags