ENSF 381
Full Stack Web Development

Lecture 32: Introduction to CI/CD

Slides: Ahmad Abdellatif, PhD

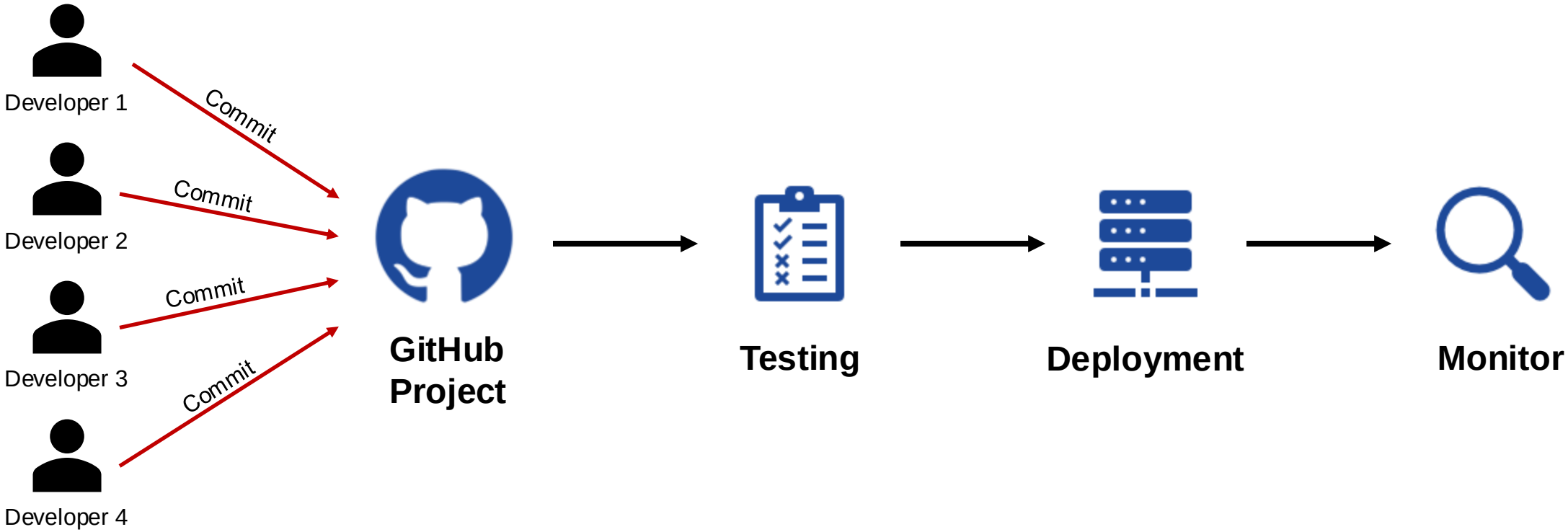
Instructor: Novarun Deb, PhD

UNIVERSITY OF CALGARY

Outline

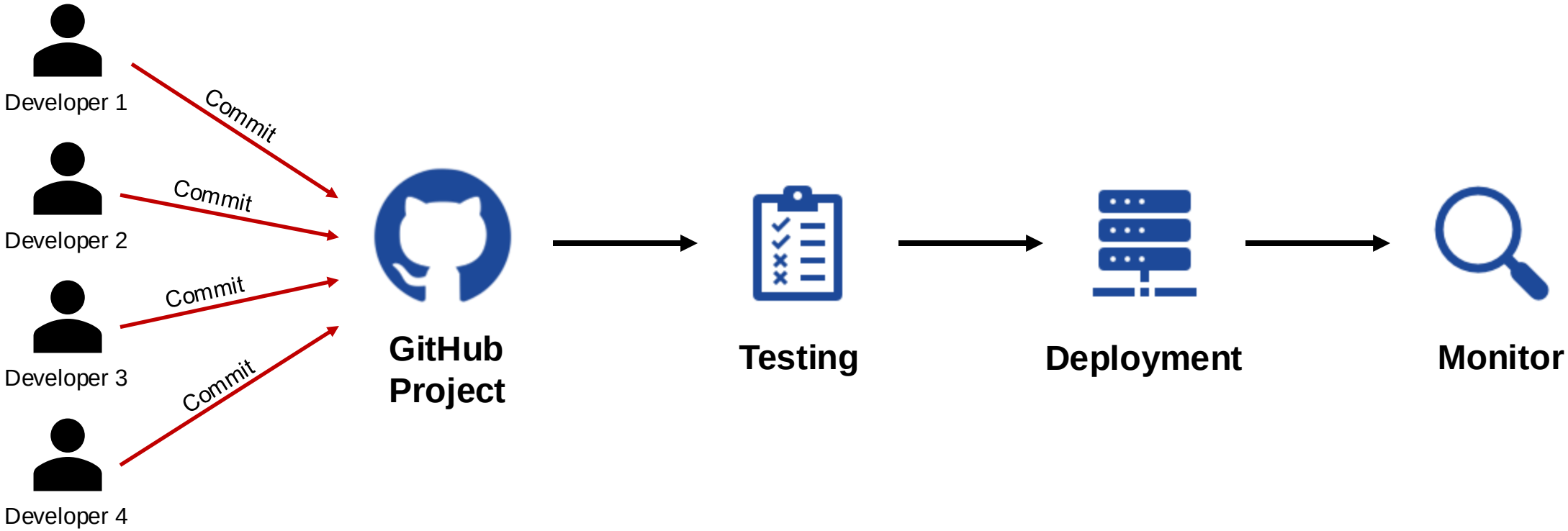
- Traditional Software Development.
- Overview of CI/CD.
- CI/CD Practices.
- CI/CD Tools.

Traditional software development



All pictures were taken from PowerPoint Stock Images.

Traditional software development



Bug Fixing

New Feature

Performance Improvement

We need to deliver these improvements to the users.

All pictures were taken from PowerPoint Stock Images.

Challenges in traditional software development

- **Lengthy development cycles**: traditional models often lead to extended development cycles, delaying the time-to-market for software products.

- **Manual errors and debugging**: manual processes in testing and deployment increase the likelihood of errors, leading to time-consuming debugging.

- **Lack of collaboration**: siloed development practices hinder collaboration, making it challenging to coordinate efforts among team members.

Continuous Integration and Delivery (CI/CD)

- A set of best practices and automated processes in software development.
- Emerged as a response to challenges in traditional software development.
- The growing adoption of agile methods and the increasing demand for faster and more reliable software delivery are accelerating the process.

CI/CD benefits

- **Reduced Manual Errors**: automated processes minimize human errors in tasks like testing and deployment.

- **Continuous Feedback Loop**: automated testing provides prompt feedback on code changes. Developers receive continuous insights, allowing for quick adjustments and improvements.

- **Faster Time-to-Market**: streamlines development workflows, enabling quicker releases, which ensures products reach the market ahead of competitors.

- **Improved Collaboration**: CI/CD fosters collaboration among developers through continuous integration. Early issues resolution swift identification and resolution of integration issues strengthen teamwork.

CI/CD practices

CI/CD comprises the integrated practices of:

1. Continuous Integration (CI).

2. Continuous Delivery (CD).

3. Continuous Deployment.

Continuous Integration (CI)

- The practice of regularly integrating code changes into a shared version control repository (e.g., GitHub).
- Automatically compiling and validating code changes upon integration.
- Integration issues are identified and resolved early in the development cycle.

Continuous Integration (CI) – Workflow overview

- **Code Commit**: commit small code changes to the version control system (e.g., Git).
- **Automated Build**: CI triggers an automated build process, compiling and validating the new code.
- **Automated Testing**: executes automated tests to guarantee the integrity and functionality of the code.
- **Feedback Loop**: immediate feedback is provided to developers, highlighting any issues that need attention.

Continuous Delivery (CD)

- The practice of ensuring code is always in a deployable state, not necessarily in production.
- Code is kept in a state of readiness, enabling organizations to deploy updates swiftly when needed.
- The key objective of continuous delivery is to minimize the time and effort required to deliver new features, improvements, or bug fixes to users while maintaining high-quality standards.

Continuous Delivery (CD) – Workflow overview

- **Automated Testing**: tests are often run again as part of the deployment process to validate the behavior of the application in an environment that closely mirrors the production setup.

- **Staging Deployment**: code changes are automatically deployed to a staging environment, mirroring the production setup.

- **Manual Approval**: stakeholders manually review and approve the changes (e.g., product managers, QA team) in the staging environment before deployment to production.

Continuous Deployment

- The practice of automatically deploying code changes to production without manual intervention.
- Continuous deployment is often paired with robust monitoring systems to quickly detect and address any issues in the live environment.
- Accelerates the feedback loop from users, facilitating quick adjustments and improvements.

Challenges in using CI/CD

- **Integration Issues**: achieving smooth integration among various tools, environments, and dependencies can be complex.
- **Complex Configurations**: managing and configuring CI/CD pipelines for complex projects can be challenging.
- **Automated Testing**: designing and maintaining comprehensive automated test suites can be time-consuming.
- **Version Control**: managing versions across multiple branches and repositories can lead to confusion.

Semantic Versioning (SemVer)

A standardized way for developers and systems to understand the nature of changes between different versions of a software library or application.

The version number is structured as three numbers separated by dots:

Major Version: indicates significant and backward-incompatible changes that may require modifications in the way software interacts with the new release.

Minor version: introduce new features or enhancements while maintaining compatibility with previous versions.

Patch Version: Indicates of issues or bugs resolution without introducing new features or breaking changes.

SemVer

- Example Progression:
  - 1.0.0 (Initial stable release)
  - 1.1.0 (Minor feature additions)
  - 1.1.1 (Patch for bug fix)
  - 2.0.0 (Major release with breaking changes)

- Pre-release Versions:
  - Used for versions in development or testing.
  - Indicated by a hyphen, e.g., 1.0.0-alpha.

- Build Metadata:
  - Identifies specific builds for internal purposes.
  - Indicated by a plus sign, e.g., 1.0.0+build123.

SemVer benefits

- **Clear Communication:** developers easily understand the nature of changes between versions.
- **Predictable Releases:** enables users to anticipate the impact of updates on their systems.
- **Dependency Management:** enables the handling and tracking of these external components to ensure that the software works as intended.

Tools for CI/CD Implementation

- **Version Control**: tracks changes, facilitates collaboration, and integrates with CI/CD pipelines.
  - Example: Git, SVN.
- **Build Automation**: orchestrates the CI/CD process, automates builds, and deployments.
  - Example: GitHub Actions, Jenkins, Travis CI, CircleCI.
- **Automated Testing**: automates testing processes, ensuring code quality and reliability.
  - Example: JUnit, Selenium, PHPUnit.
- **Containerization**: encapsulates applications and dependencies, ensuring consistency across different environments.
  - Example: Docker.
- **Orchestration**: manages containerized applications, automates deployment, scaling, and operations.
  - Example: Kubernetes.
- **Configuration Management**: automates infrastructure setup and configuration for consistency.
  - Example: Ansible, Puppet, Chef.

GitHub Actions

```yaml
name: Example on React Application Deployment

on:
  push:
    branches:
      - main
jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout Repository
      uses: actions/checkout@v2

    - name: Set up Node.js
      uses: actions/setup-node@v2
      with:
        node-version: 18

    - name: Install required dependencies
      run: npm install

    - name: Test React App
      run: npm run test

    - name: Build React App
      run: npm run build

    - name: Deploy to GitHub Pages
      uses: peaceiris/actions-gh-pages@v3
      with:
        github_token: ${{ secrets.GITHUB_TOKEN }}
        publish_dir: ./build
```

The workflow will be triggered only when there's a push to the 'main' branch.

Specify the type of environment for the job.

Fetches the latest commit of your repository.

Configures the GitHub Actions runner (server) with Node.js.

Installs the Node.js dependencies using npm.

Runs the automated tests.

Runs the build script to generate the production-ready build of your React application.

Deploy the built application to GitHub Pages

Questions