DevOps Curriculum Module 1 & 2

Module 1: Introduction to DevOps

1.1 What is DevOps?

DevOps is a set of cultural philosophies, practices, and tools that increases an organization's ability to deliver applications and services at high velocity. By evolving and improving products at a faster pace than organizations using traditional software development and infrastructure management processes, DevOps enables businesses to serve their customers more effectively and compete in the market more aggressively.

1.2 Key Principles of DevOps

1. Collaboration: DevOps emphasizes close collaboration between development (Dev) and operations (Ops) teams. This breaks down silos and improves communication and shared responsibility for the outcomes.
2. Automation: Automating repetitive tasks, such as code integration, testing, and deployment, increases efficiency and reduces human error.
3. Continuous Integration and Continuous Deployment (CI/CD): Integrates and delivers code changes frequently and reliably, ensuring that software can be released at any time.
4. Monitoring and Logging: Continuous monitoring and logging help detect issues and ensure system reliability and performance.
5. Infrastructure as Code (IaC): Managing infrastructure using code allows for version control, automation, and consistency across environments.

1.3 Benefits of DevOps

- Faster Time to Market: Accelerates the development and deployment cycle.
- Improved Deployment Frequency: Allows for frequent, reliable releases.
- Lower Failure Rate of New Releases: Automated testing and deployment reduce errors.
- Shorter Lead Time Between Fixes: Faster detection and resolution of issues.

- Improved Mean Time to Recovery (MTTR): Rapid recovery from failures through automated rollback and recovery mechanisms.

1.4 DevOps Lifecycle

1. Plan: Involves project planning and defining requirements.
2. Code: Writing and reviewing code collaboratively.
3. Build: Compiling code into executable artifacts.
4. Test: Automated testing to ensure code quality.
5. Release: Deploying code to production environments.
6. Deploy: Ensuring that code runs smoothly in production.
7. Operate: Monitoring applications in production.
8. Monitor: Continuous monitoring and feedback loop.

1.5 DevOps Tools

- Version Control: Git
- CI/CD: Jenkins, GitLab CI, Aws codebuild, Azure pipeline
- Configuration Management: Ansible, Puppet, Chef
- Containerization: Docker
- Orchestration: Kubernetes, Docker-Compose, Openshift
- IaC: Terraform, cloudformation
- Monitoring: Prometheus, Grafana, elk stack, appdynamic, splunk

Module 2: Version Control with Git

2.1 Introduction to Git

Git is a distributed version control system (VCS) that allows multiple developers to work on a project simultaneously without overriding each other's changes. Created by Linus Torvalds in 2005, Git is widely used for source code management in software development.

2.2 Installing Git

To install Git on your system:

- Windows: Download from git-scm.com and follow the installation instructions.
- macOS: Use Homebrew with the command `brew install git`.
- Linux: Use your distribution's package manager (e.g., `sudo apt install git` for Debian-based systems).

2.3 Basic Git Operations

2.3.1 Initializing a Repository

To start a new Git repository:

```
git init
```

This command creates a new subdirectory named `.git` that contains all the necessary repository files.

2.3.2 Cloning a Repository

To create a copy of an existing repository:

```
git clone <repository_url>
```

2.3.3 Staging and Committing Changes

To add changes to the staging area:

```
git add <file_or_directory>
```

To commit the staged changes with a message:

```
git commit -m "Your commit message"
```

2.3.4 Pushing and Pulling Changes

To push committed changes to a remote repository:

```
git push origin <branch_name>
```

To fetch and merge changes from a remote repository:

```
git pull origin <branch_name>
```

2.4 Branching and Merging

2.4.1 Creating and Managing Branches

Branches allow you to develop features, fix bugs, or experiment independently from the main codebase.
To create a new branch:

```
git branch <new_branch>
```

To switch to a branch:

```
git checkout <branch_name>
```

To create and switch to a new branch in one command:

```
git checkout -b <new_branch>
```

2.4.2 Merging Branches

To merge changes from one branch into another:

```
git checkout <target_branch>
git merge <source_branch>
```

2.4.3 Resolving Conflicts

If there are conflicts during a merge, Git will mark the conflicts in the files. You need to edit the files to resolve the conflicts and then stage and commit the changes.

2.5 Git Workflows

2.5.1 Centralized Workflow

A simple workflow where all changes are committed to a single central repository. Useful for small teams.

2.5.2 Feature Branching

Each feature is developed in its own branch, which is merged into the main branch when complete. This allows multiple features to be developed simultaneously without interfering with each other.

2.5.3 GitFlow

A robust workflow involving multiple branches for development, testing, and production. Main branches include `master` for production-ready code and `develop` for the latest delivered development changes. Feature branches are created from `develop` and merged back when complete.

2.6 Practical Lab Exercise: Version Control with Git

1. Set Up a Local Repository:
    - Initialize a Git repository and create a README file.
    - Commit the README file to the repository.

```
git init
echo "# My Project" > README.md
git add README.md
git commit -m "Initial commit"
```

2. Basic Git Operations:
    - Clone a remote repository.
    - Make changes to a file and commit the changes.
    - Push the changes to the remote repository.

```
git clone <repository_url>
echo "New line" >> README.md
git add README.md
git commit -m "Added a new line to README"
git push origin main
```

3. Branching and Merging:
    - Create a new branch, make changes, and merge back to the main branch.

```
git checkout -b new-feature
echo "Feature code" > feature.txt
git add feature.txt
git commit -m "Added feature"
git checkout main
git merge new-feature
```

4. Conflict Resolution:
    - Simulate a merge conflict by modifying the same line in different branches.
    - Resolve the conflict and complete the merge.

```
git checkout -b conflicting-branch
echo "Conflicting change" > conflict.txt
git add conflict.txt
git commit -m "Conflicting change"
git checkout main
echo "Main branch change" > conflict.txt
git add conflict.txt
git commit -m "Main branch change"
git merge conflicting-branch
# Resolve conflict in conflict.txt manually
git add conflict.txt
git commit -m "Resolved conflict"
```