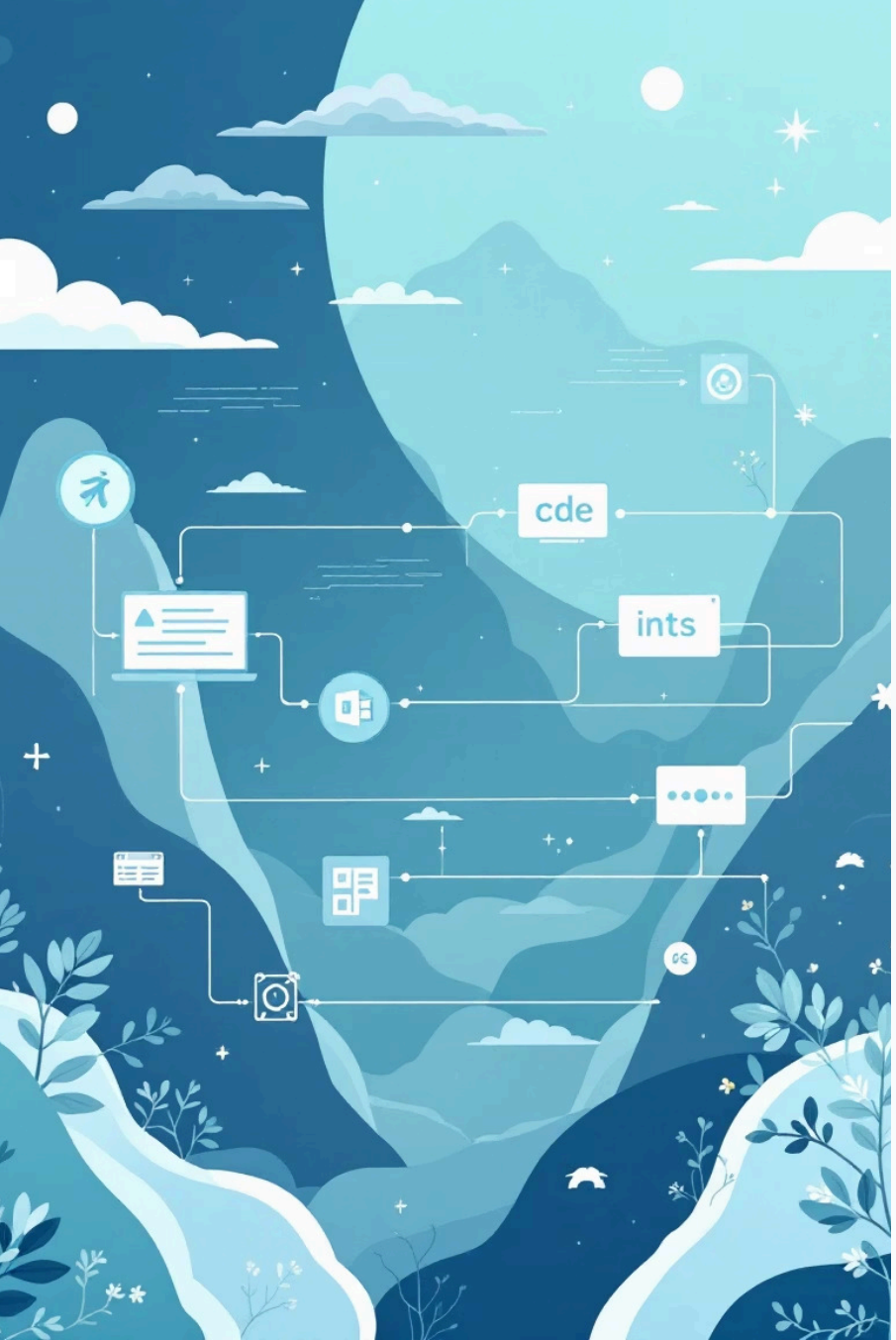


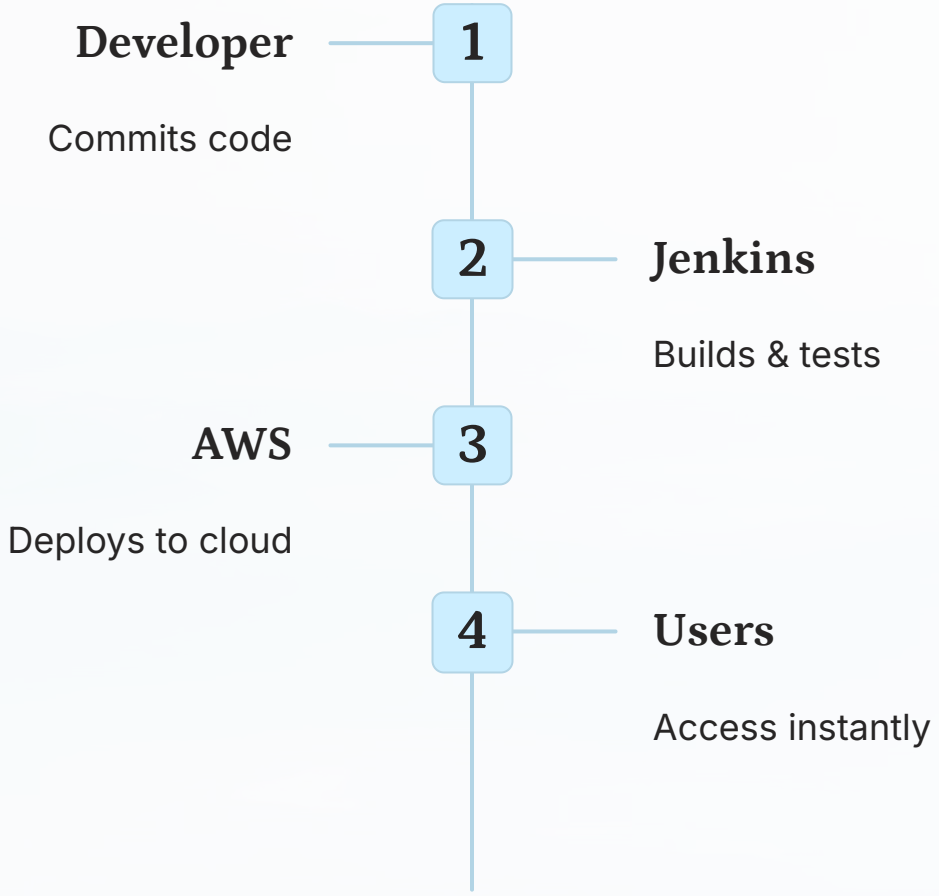

# Day 3 – Continuous Deployment & Automation

From "It works on my machine" to "It works in production — automatically."

# Why Deployment Automation Exists

Manual deployment is inherently inconsistent and error-prone, leading to unexpected failures and lengthy debugging sessions. CI/CD pipelines ensure every build is tested, versioned, and deployed identically across all environments.

**The goal:** Deliver smaller, faster, and safer changes to production with confidence and predictability.



# What Happens Without CD

01

### Developer pushes code

New feature committed to repository

02

### Operations team manually SSHs into EC2

Connects to production server

03

### Pulls repository, restarts application

Misses one critical configuration variable

04

### Application crashes, downtime begins

Users affected, revenue lost

**CD removes that fragile human link** — automating every step from commit to production deployment with consistency and reliability.

# Continuous Delivery vs Continuous Deployment

Understanding the distinction between these two approaches helps teams choose the right automation level for their deployment strategy.

| Stage | Who triggers? | When does it deploy? |
| --- | --- | --- |
| **Continuous Delivery** | Manual approval required | After tests pass successfully |
| **Continuous Deployment** | Fully automated process | On every successful build |

## Continuous Delivery

Human gatekeeper reviews and approves releases, providing control over timing and coordination with business needs.

## Continuous Deployment

Code flows automatically to production without human intervention, maximising speed and minimising delay between development and delivery.

# A Typical CD Workflow

### Code pushed to main

Developer commits changes to the main branch

### Jenkins triggers automatically

Webhook detects the commit and starts the pipeline

### Jenkins builds and tests

Compiles code and runs comprehensive test suite

### Creates Docker image

Packages application with all dependencies

### Pushes image to AWS ECR

Stores container in Elastic Container Registry

### Deploys to EC2/ECS

Launches containerised application in production

### Health checks verify deployment

Confirms application is running correctly

# Jenkins CD Pipeline – Overview

A Jenkins pipeline orchestrates the entire deployment process through distinct stages, each with a specific responsibility in the continuous deployment workflow.

## Essential Plugins

- **Pipeline** — Core pipeline functionality and Groovy DSL support
- **AWS Credentials** — Secure authentication with AWS services
- **Docker Pipeline** — Build and push container images seamlessly

| | |
|---|---|
| 1 | **Build**<br>Compile and package |
| 2 | **Test**<br>Run test suites |
| 3 | **Dockerise**<br>Create container |
| 4 | **Deploy**<br>Push to production |

# Pipeline Script (Simplified)

This Jenkinsfile defines a complete CD pipeline that automates everything from build to deployment — eliminating manual SSH access and ensuring consistent, repeatable deployments.

```
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        sh 'npm run build'
      }
    }
    stage('Dockerise') {
      steps {
        sh '''
          docker build -t myapp:latest .
          docker push /myapp:latest
        '''
      }
    }
    stage('Deploy') {
      steps {
        sh 'ssh ec2-user@ "docker pull ... && docker run -d ..."'
      }
    }
  }
}
```

💡 **Why this matters:** Automates everything post-commit — no SSHing ever again. Every deployment follows the exact same process, reducing errors and increasing confidence.

# Blue-Green Deployment

## The Strategy

Blue-green deployment maintains two identical production environments, allowing seamless transitions between versions with zero downtime and instant rollback capability.

- **Deploy new version to Green**

  Idle environment receives update

- **Tests pass successfully**

  Verify functionality and performance

- **Route 53 switches traffic**

  Users redirected to Green instantly

- **Blue kept for rollback**

  Previous version remains available

### Blue Environment

**Current production version**

Serving all user traffic until switch

### Green Environment

**New version deployment**

Tested and ready to receive traffic

# What Blue-Green Solves

### No Partial Updates

The entire environment switches atomically — users never experience a partially updated application state.

### Immediate Rollback

If issues arise, simply redirect traffic back to the previous environment within seconds without redeployment.

### Zero Downtime Switching

Traffic switches instantly at the DNS or load balancer level — users don't notice any interruption in service.

# Canary Deployment

Canary deployment gradually rolls out changes to a small subset of users first, allowing teams to detect issues before they affect the entire user base.

1

### Deploy to 10% of traffic

New version serves a small, controlled user segment

2

### Observe metrics closely

Monitor error rates, performance, and user behaviour

3

### Gradually increase to 100%

Expand rollout as confidence grows: 25%, 50%, 75%, then full

4

### Rollback if failure detected

Immediately revert traffic if anomalies appear

## Benefits

- Minimises blast radius of bugs
- Real user feedback before full rollout
- Data-driven deployment decisions

# Pipeline Folder Setup
# Organize your Jenkins workspace cleanly:

```
cd /var/lib/jenkins/workspace
mkdir myapp-cd
cd myapp-cd
touch Jenkinsfile
```

✅ Keep your app code and Jenkinsfile in the same repo.
Jenkins automatically detects pipeline stages from the file.

# Minimal Jenkinsfile for CD

```
pipeline {
  agent any
  stages {
    stage('Build') {
      steps { sh 'npm run build' }
    }
    stage('Dockerize') {
      steps {
        sh '''
        docker build -t myapp:latest .
        docker tag myapp:latest <ECR_URL>/myapp:latest
        docker push <ECR_URL>/myapp:latest
        '''
      }
    }
    stage('Deploy') {
      steps {
        sh 'ssh ec2-user@<EC2_IP> "docker pull <ECR_URL>/myapp:latest && docker run -d -p 80:80 <ECR_URL>/myapp:latest"'
      }
    }
  }
}
```

🧩 Core logic: Build → Dockerize → Push → Deploy

# Jenkins → AWS Workflow (Visual)

This visual flow illustrates how Jenkins orchestrates the deployment of containerised applications to AWS, from code commit to live production.

### Code Commit

Developer pushes new code to the repository, triggering the pipeline.

### Jenkins Build

Jenkins automatically builds the Docker image from the source code.

### Push to ECR

The built Docker image is pushed to AWS Elastic Container Registry.

### Deploy to EC2

Jenkins securely connects to the EC2 instance and deploys the container.

### Live Application

The application is now running on AWS and serving user traffic.

# AWS Credentials Setup

1. Create IAM user: `jenkins-deployer`
2. Permissions: `AmazonEC2FullAccess`, `AmazonECRFullAccess`
3. In Jenkins:
   - Go to **Manage Jenkins → Credentials → Global → Add**
   - Add AWS Access Key & Secret
4. Use environment variables:

```
environment {
  AWS_ACCESS_KEY_ID = credentials('aws-key')
  AWS_SECRET_ACCESS_KEY = credentials('aws-secret')
}
```

# ECR Authentication in Jenkins
# Before pushing or pulling images:

```
aws ecr get-login-password --region ap-south-1 | \
docker login --username AWS --password-stdin <AWS_ACCOUNT_ID>.dkr.ecr.ap-south-1.amazonaws.com
```

Add this step before Docker build.

Prevents “unauthorized” errors during image push.

# EC2 Deployment Script

```
#!/bin/bash
APP_NAME="myapp"
ECR_URL="<AWS_ACCOUNT_ID>.dkr.ecr.ap-south-1.amazonaws.com/$APP_NAME:latest"

echo "Deploying latest build..."
docker pull $ECR_URL
docker stop $APP_NAME || true
docker rm $APP_NAME || true
docker run -d -p 80:80 --name $APP_NAME $ECR_URL
```

✅ Save as `deploy.sh` in your repo.
✅ Make executable: `chmod +x deploy.sh`

# Putting It All Together

### Code Commit to GitHub

Developer pushes new code to the repository.

### Jenkins Trigger & Build

Pipeline automatically starts, building the application.

### Dockerise & Push to ECR

Application is containerised and pushed to AWS ECR.

### Deploy to EC2 via SSH

Jenkins connects to EC2 and executes deployment script.

### Application Live

The new version is active and accessible to users.

# Verification After Deployment

```
curl http://<EC2_PUBLIC_IP>:80
docker ps
docker logs myapp --tail 10
```

If Jenkins logs show `exit 0` and you can curl the IP → deployment success.
If not → check Docker logs first.

# Day 3 Recap – Key Takeaways

1

**Understood the core difference between CI and CD**, laying the foundation for automated deployments.

2

**Built a complete Build → Dockerize → Push → Deploy pipeline** from scratch.

3

**Successfully integrated Jenkins with AWS and Docker** for seamless application delivery.

4

**Achieved fully automated, one-click deployment**, streamlining the release process.