DevOps Complete Knowledge Base

Comprehensive Guide to Tools, Technologies, and Best Practices

---

Table of Contents

21. Kubernetes Architecture
22. Kubernetes Deployments
23. Kubernetes Services and Networking
24. Kubernetes Storage
25. Helm Charts
26. Kubernetes Security

Part VI: Cloud Platforms

27. AWS DevOps Services
28. Azure DevOps Platform
29. Google Cloud Platform
30. Multi-Cloud Strategies

Part VII: Infrastructure as Code

31. Terraform
32. Ansible
33. Puppet
34. Chef
35. CloudFormation
36. ARM Templates

Part VIII: Monitoring and Logging

37. Prometheus
38. Grafana
39. ELK Stack
40. Splunk
41. Datadog
42. New Relic

Part IX: Security and Compliance

43. DevSecOps
44. Security Scanning Tools
45. Compliance and Governance
46. Secret Management

Part X: Advanced Topics

47. Microservices Architecture
48. Service Mesh
49. Serverless Computing
50. Performance Optimization

Chapter 1: Introduction to DevOps

What is DevOps?

DevOps is a cultural and technical movement that emphasizes collaboration between software development (Dev) and IT operations (Ops) teams. It aims to shorten the development lifecycle while delivering features, fixes, and updates frequently in close alignment with business objectives.

Core Principles

Collaboration: Breaking down silos between development and operations teams to foster better communication and shared responsibility.

Automation: Implementing automated processes for testing, deployment, and infrastructure management to reduce manual errors and increase efficiency.

Continuous Integration: Regularly merging code changes into a central repository where automated builds and tests are run.

Continuous Deployment: Automatically deploying code changes to production after passing all tests and quality checks.

Monitoring and Feedback: Continuously monitoring applications and infrastructure to gather feedback and improve processes.

Benefits of DevOps

Faster Time to Market: Streamlined processes allow for quicker feature releases and bug fixes.

Improved Quality: Automated testing and continuous integration catch issues early in the development process.

Better Collaboration: Teams work together more effectively, leading to better solutions and reduced conflicts.

Increased Reliability: Automated deployments and monitoring reduce downtime and improve system stability.

Cost Efficiency: Automation reduces manual work, leading to cost savings and better resource utilization.

DevOps vs Traditional IT

Traditional IT operations often involve manual processes, longer release cycles, and separate teams for development and operations. DevOps transforms this by:

- Integrating development and operations workflows
- Implementing automation throughout the pipeline
- Enabling faster feedback loops
- Promoting shared responsibility for system reliability
- Encouraging experimentation and learning from failures

Chapter 2: DevOps Culture and Principles

The CALMS Framework

Culture

Creating a collaborative environment where teams share responsibility for the entire application lifecycle. This involves:

- Breaking down organizational silos
- Encouraging open communication
- Promoting shared goals and metrics
- Fostering a learning mindset
- Embracing failure as a learning opportunity

Automation

Implementing automated processes to reduce manual effort and human error:

- Automated testing and validation
- Continuous integration and deployment
- Infrastructure provisioning
- Monitoring and alerting
- Self-healing systems

Lean

Applying lean principles to eliminate waste and optimize value delivery:

- Minimizing work in progress
- Reducing batch sizes
- Eliminating non-value-added activities
- Continuous improvement
- Focus on flow efficiency

Measurement

Using metrics and data to drive decisions and improvements:

- Key performance indicators (KPIs)
- Mean time to recovery (MTTR)
- Deployment frequency
- Lead time for changes
- Change failure rate

Sharing

Promoting knowledge sharing and collective ownership:

- Documentation and knowledge bases
- Code reviews and pair programming
- Post-mortem analysis
- Cross-team collaboration
- Open source contributions

DevOps Transformation Strategies

Assessment Phase

- Current state analysis
- Identifying pain points
- Skill gap assessment
- Tool evaluation
- Cultural readiness

Planning Phase

- Defining transformation goals
- Creating roadmap
- Resource allocation

- Risk assessment
- Success metrics definition

Implementation Phase

- Pilot project selection
- Tool implementation
- Process automation
- Team training
- Change management

Optimization Phase

- Performance monitoring
- Continuous improvement
- Scaling successful practices
- Advanced tool adoption
- Culture reinforcement

Chapter 3: DevOps Lifecycle and Methodologies

The DevOps Lifecycle

Plan

Strategic planning and requirement gathering:

- Business requirements analysis
- Technical specifications
- Resource planning
- Timeline estimation
- Risk assessment

Code

Development and version control:

- Writing application code
- Code reviews
- Version control management
- Branching strategies

- Code quality checks

Build

Compilation and packaging:

- Source code compilation
- Dependency management
- Artifact creation
- Build automation
- Quality gates

Test

Automated testing and validation:

- Unit testing
- Integration testing
- Performance testing
- Security testing
- User acceptance testing

Release

Deployment preparation:

- Release planning
- Environment preparation
- Rollback strategies
- Deployment automation
- Release notes

Deploy

Production deployment:

- Automated deployment
- Blue-green deployment
- Canary releases
- Infrastructure provisioning
- Configuration management

Operate

Production management:

- System monitoring
- Performance optimization
- Incident response
- Capacity planning
- User support

Monitor

Continuous monitoring and feedback:

- Application monitoring
- Infrastructure monitoring
- Log analysis
- User behavior tracking
- Performance metrics

Agile and DevOps Integration

Scrum and DevOps

Combining Scrum methodology with DevOps practices:

- Sprint planning with DevOps considerations
- Continuous integration in sprint cycles
- Automated testing in definition of done
- DevOps tasks in sprint backlog
- Cross-functional team collaboration

Kanban and DevOps

Using Kanban boards for DevOps workflow management:

- Visualizing work in progress
- Limiting work in progress
- Continuous flow optimization
- Metrics-driven improvement
- Flexible prioritization

Lean Startup and DevOps

Applying lean startup principles:

- Build-measure-learn cycles
- Minimum viable product (MVP)
- A/B testing
- Rapid experimentation
- Customer feedback integration

Chapter 4: Setting Up DevOps Environment

Development Environment Setup

Local Development Environment

Setting up a consistent local development environment:

Prerequisites:

- Operating system requirements
- Hardware specifications
- Network configuration
- Security policies

Tools Installation:

- Code editors (VS Code, IntelliJ)
- Version control (Git)
- Programming languages and runtimes
- Package managers
- Development frameworks

Configuration Management:

- Environment variables
- Configuration files
- Secrets management
- Database connections
- API endpoints

Virtual Development Environments

Vagrant Setup:

```bash
# Vagrantfile example
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/bionic64"
  config.vm.network "private_network", ip: "192.168.33.10"
  config.vm.provision "shell", inline: <<-SHELL
    apt-get update
    apt-get install -y docker.io
    usermod -aG docker vagrant
  SHELL
end
```

Docker Development Environment:

```dockerfile
# Development Dockerfile
FROM node:16-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

Docker Compose for Multi-Service Development:

```yaml
version: '3.8'
services:
  web:
    build: .
    ports:
      - "3000:3000"
    depends_on:
      - db
  db:
    image: postgres:13
    environment:
      POSTGRES_DB: devdb
      POSTGRES_USER: developer
      POSTGRES_PASSWORD: password
```

CI/CD Environment Setup

Jenkins Installation and Configuration

Jenkins Installation on Ubuntu:

```bash
# Install Java
sudo apt update
sudo apt install openjdk-11-jdk

# Add Jenkins repository
wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add -
sudo sh -c 'echo deb https://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'

# Install Jenkins
sudo apt update
sudo apt install jenkins

# Start Jenkins service
sudo systemctl start jenkins
sudo systemctl enable jenkins
```

Jenkins Configuration:

- Security configuration
- Plugin installation
- Global tool configuration
- Credential management
- Job templates

GitHub Actions Setup

Basic Workflow Configuration:

```yaml
```

```yaml
name: CI/CD Pipeline
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Setup Node.js
      uses: actions/setup-node@v2
      with:
        node-version: '16'
    - run: npm ci
    - run: npm test
```

GitLab CI/CD Setup

GitLab CI Configuration:

yaml

```yaml
stages:
  - test
  - build
  - deploy

variables:
  DOCKER_IMAGE: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA

test:
  stage: test
  script:
    - npm install
    - npm test
  only:
    - merge_requests
    - main

build:
  stage: build
  script:
    - docker build -t $DOCKER_IMAGE .
    - docker push $DOCKER_IMAGE
  only:
    - main
```

Monitoring Environment Setup

Prometheus Configuration

Prometheus Installation:

```bash
# Download Prometheus
wget https://github.com/prometheus/prometheus/releases/download/v2.30.0/prometheus-2.30.0.linux-amd64.tar
tar xvfz prometheus-2.30.0.linux-amd64.tar.gz
cd prometheus-2.30.0.linux-amd64
```

Prometheus Configuration:

```yaml
```

```
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']

  - job_name: 'node'
    static_configs:
      - targets: ['localhost:9100']
```

Grafana Setup

Grafana Installation:

```bash
# Install Grafana
sudo apt-get install -y software-properties-common
sudo add-apt-repository "deb https://packages.grafana.com/oss/deb stable main"
wget -q -O - https://packages.grafana.com/gpg.key | sudo apt-key add -
sudo apt-get update
sudo apt-get install grafana
```

Grafana Configuration:

- Data source configuration
- Dashboard creation
- Alert configuration
- User management
- Plugin installation

Chapter 5: Git Fundamentals

Git Basics

What is Git?

Git is a distributed version control system that tracks changes in source code during software development. It allows multiple developers to work on the same project efficiently while maintaining a complete history of all changes.

Key Concepts

Repository: A Git repository is a directory that contains your project files and the entire history of changes made to those files.

Working Directory: The current state of your project files on your local machine.

Staging Area: A intermediate area where changes are prepared before committing them to the repository.

Commit: A snapshot of your project at a specific point in time, containing a unique identifier and metadata.

Branch: A parallel line of development that allows you to work on different features independently.

Remote: A version of your repository hosted on a server, enabling collaboration with other developers.

Basic Git Commands

Repository Initialization:

```bash
# Initialize a new Git repository
git init

# Clone an existing repository
git clone https://github.com/username/repository.git

# Check repository status
git status
```

File Operations:

```bash
# Add files to staging area
git add filename.txt
git add .  # Add all files
git add *.js  # Add all JavaScript files

# Remove files from staging area
git reset filename.txt
git reset .  # Remove all files

# Commit changes
git commit -m "Add new feature"
git commit -am "Add and commit in one step"
```

Branch Operations:

```bash
# List branches
git branch
git branch -a  # Include remote branches

# Create new branch
git branch feature-branch
git checkout -b feature-branch  # Create and switch

# Switch branches
git checkout main
git switch feature-branch  # New syntax

# Merge branches
git checkout main
git merge feature-branch

# Delete branch
git branch -d feature-branch
```

Remote Operations:

```bash
# Add remote repository
git remote add origin https://github.com/username/repository.git

# Push changes
git push origin main
git push -u origin feature-branch

# Pull changes
git pull origin main
git fetch origin  # Fetch without merging

# List remotes
git remote -v
```

Git Configuration

Global Configuration:

```bash
```

```
# Set user information
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"

# Set default editor
git config --global core.editor "code --wait"

# Set default branch name
git config --global init.defaultBranch main

# View configuration
git config --list
```

Repository-Specific Configuration:

```
bash

# Configure for specific repository
git config user.name "Work Name"
git config user.email "work.email@company.com"
```

Advanced Git Features

Git Stash:

```
bash

# Stash current changes
git stash
git stash push -m "Work in progress"

# List stashes
git stash list

# Apply stash
git stash apply
git stash pop  # Apply and remove

# Drop stash
git stash drop stash@{0}
```

Git Log and History:

```
bash
```

```
# View commit history
git log
git log --oneline
git log --graph --decorate --all

# View specific file history
git log filename.txt
git log -p filename.txt  # Show changes

# Search commits
git log --grep="bug fix"
git log --author="John Doe"
```

Git Rebase:

```
bash

# Interactive rebase
git rebase -i HEAD~3

# Rebase onto another branch
git rebase main

# Continue rebase after conflicts
git rebase --continue

# Abort rebase
git rebase --abort
```

Git Workflows

Feature Branch Workflow:

1. Create feature branch from main
2. Make changes and commit
3. Push feature branch to remote
4. Create pull request
5. Review and merge
6. Delete feature branch

Gitflow Workflow:

- Main branch: Production-ready code
- Develop branch: Integration branch

- Feature branches: New features
- Release branches: Prepare releases
- Hotfix branches: Emergency fixes

GitHub Flow:

1. Create branch from main
2. Add commits
3. Open pull request
4. Review and discuss
5. Deploy and test
6. Merge to main

Chapter 6: GitHub/GitLab/Bitbucket

GitHub

GitHub Features

Repository Management:

- Public and private repositories
- Repository templates
- Repository insights and analytics
- Wiki documentation
- Issue tracking
- Project boards

Collaboration Features:

- Pull requests
- Code reviews
- Team management
- Organization settings
- Permission management
- Branch protection rules

Integration Capabilities:

- GitHub Actions (CI/CD)

- Third-party integrations
- API access
- Webhooks
- GitHub Apps
- Marketplace

GitHub Actions

Basic Workflow Structure:

```yaml
name: CI Pipeline
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout code
      uses: actions/checkout@v3

    - name: Setup Node.js
      uses: actions/setup-node@v3
      with:
        node-version: '18'
        cache: 'npm'

    - name: Install dependencies
      run: npm ci

    - name: Run tests
      run: npm test

    - name: Upload coverage
      uses: codecov/codecov-action@v3
```

Advanced Workflow Features:

```yaml
```

```yaml
name: Deploy to Production
on:
  release:
    types: [published]

jobs:
  deploy:
    runs-on: ubuntu-latest
    environment: production
    steps:
    - uses: actions/checkout@v3

    - name: Deploy to server
      uses: appleboy/ssh-action@v0.1.5
      with:
        host: ${{ secrets.HOST }}
        username: ${{ secrets.USERNAME }}
        key: ${{ secrets.KEY }}
        script: |
          cd /var/www/app
          git pull origin main
          npm install
          npm run build
          pm2 restart app
```

Matrix Builds:

```yaml
strategy:
  matrix:
    node-version: [16, 18, 20]
    os: [ubuntu-latest, windows-latest, macos-latest]

runs-on: ${{ matrix.os }}
steps:
- uses: actions/checkout@v3
- uses: actions/setup-node@v3
  with:
    node-version: ${{ matrix.node-version }}
```

GitHub Advanced Features

GitHub Pages:

- Static site hosting

- Custom domains
- Jekyll integration
- Automated deployment
- SSL certificates

GitHub Packages:

- Package registry
- Docker container registry
- npm registry
- Maven repository
- NuGet gallery

GitHub Security:

- Dependabot alerts
- Security advisories
- Code scanning
- Secret scanning
- Supply chain security

GitLab

GitLab Features

Repository Management:

- Git repository hosting
- Merge requests
- Issue tracking
- Wiki and documentation
- Code review tools
- Repository analytics

CI/CD Integration:

- Built-in CI/CD pipelines
- Auto DevOps
- Kubernetes integration
- Container registry

- Deployment management
- Pipeline schedules

Project Management:

- Issue boards
- Milestones
- Time tracking
- Roadmaps
- Requirements management
- Value stream analytics

GitLab CI/CD

Pipeline Configuration:

```yaml
```

```yaml
stages:
  - test
  - build
  - deploy

variables:
  DOCKER_DRIVER: overlay2
  DOCKER_TLS_CERTDIR: "/certs"

before_script:
  - echo "Starting pipeline"

test:
  stage: test
  image: node:16
  script:
    - npm install
    - npm test
  coverage: '/Coverage: \d+\.\d+%/'
  artifacts:
    reports:
      coverage_report:
        coverage_format: cobertura
        path: coverage/cobertura-coverage.xml

build:
  stage: build
  image: docker:latest
  services:
    - docker:dind
  script:
    - docker build -t $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA .
    - docker push $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
  only:
    - main
```

Advanced Pipeline Features:

```yaml
```

```yaml
deploy_staging:
  stage: deploy
  script:
    - kubectl apply -f k8s/staging/
  environment:
    name: staging
    url: https://staging.example.com
  only:
    - main

deploy_production:
  stage: deploy
  script:
    - kubectl apply -f k8s/production/
  environment:
    name: production
    url: https://example.com
  when: manual
  only:
    - main
```

Multi-Project Pipelines:

```yaml
trigger_downstream:
  stage: deploy
  trigger:
    project: group/downstream-project
    branch: main
  variables:
    UPSTREAM_BRANCH: $CI_COMMIT_REF_NAME
```

GitLab DevOps Platform

Planning:

- Issue management
- Epic tracking
- Requirements management
- Design management
- Portfolio management

Create:

- Source code management
- Web IDE
- Code quality
- Static site generator
- Snippet management

Verify:

- Continuous integration
- Code review
- Testing framework
- Accessibility testing
- Browser performance testing

Package:

- Container registry
- Package registry
- Dependency proxy
- Infrastructure registry

Secure:

- Static application security testing
- Dynamic application security testing
- Interactive application security testing
- Dependency scanning
- License compliance

Deploy:

- Continuous deployment
- Feature flags
- Release orchestration
- Auto deploy
- Environment management

Monitor:

- Application monitoring

- Incident management
- Error tracking
- Performance monitoring
- Product analytics

Bitbucket

Bitbucket Features

Repository Management:

- Git and Mercurial support
- Pull requests
- Branch permissions
- Code insights
- Repository access keys
- Smart mirroring

Integration with Atlassian:

- Jira integration
- Confluence integration
- Trello integration
- Bamboo CI/CD
- Crowd authentication

Bitbucket Pipelines:

- YAML-based configuration
- Docker container support
- Parallel steps
- Manual triggers
- Deployment environments

Bitbucket Pipelines

Basic Pipeline Configuration:

```yaml
```

```yaml
image: node:16

pipelines:
  default:
    - step:
        name: Test
        caches:
          - node
        script:
          - npm install
          - npm test
        artifacts:
          - test-results/**

    - step:
        name: Build
        script:
          - npm run build
        artifacts:
          - dist/**

  branches:
    main:
      - step:
          name: Deploy to Production
          deployment: production
          script:
            - npm run deploy
```

Advanced Pipeline Features:

```yaml
```

```yaml
pipelines:
  pull-requests:
    '**':
      - step:
          name: Test PR
          script:
            - npm install
            - npm test
          services:
            - postgres
            - redis

  branches:
    main:
      - parallel:
          - step:
              name: Test
              script:
                - npm test
          - step:
              name: Security Scan
              script:
                - npm audit
      - step:
          name: Deploy
          deployment: production
          script:
            - ./deploy.sh

definitions:
  services:
    postgres:
      image: postgres:13
      variables:
        POSTGRES_DB: testdb
        POSTGRES_USER: testuser
        POSTGRES_PASSWORD: testpass
    redis:
      image: redis:6
```

Custom Docker Images:

```yaml
```

```
image: mycompany/custom-build-image:latest

pipelines:
  default:
    - step:
        name: Custom Build
        script:
          - ./custom-build-script.sh
        services:
          - docker
```

Comparison Summary

Feature	GitHub	GitLab	Bitbucket
Hosting	Cloud, Enterprise	Cloud, Self-hosted	Cloud, Server
CI/CD	GitHub Actions	Built-in GitLab CI	Bitbucket Pipelines
Issue Tracking	Basic	Advanced	Basic
Project Management	Project boards	Comprehensive	Limited
Integration	Extensive marketplace	Built-in tools	Atlassian suite
Pricing	Free tier generous	Free tier good	Free tier limited
Enterprise Features	Advanced	Comprehensive	Good

Chapter 7: Branching Strategies

Git Flow

Git Flow is a branching model that defines strict branching rules and workflows designed around project releases. It's ideal for projects with scheduled releases and multiple versions in production.

Branch Types

Main Branches:

- `main` (or `master`): Production-ready code
- `develop`: Integration branch for features

Supporting Branches:

- `feature/*`: New features
- `release/*`: Prepare new releases
- `hotfix/*`: Emergency fixes

Git Flow Workflow

Feature Development:

```bash
# Start new feature
git flow feature start new-feature

# Work on feature
git add .
git commit -m "Implement new feature"

# Finish feature
git flow feature finish new-feature
```

Release Process:

```bash
# Start release
git flow release start 1.0.0

# Prepare release (version bumps, documentation)
git add .
git commit -m "Prepare release 1.0.0"

# Finish release
git flow release finish 1.0.0
```

Hotfix Process:

```bash
# Start hotfix
git flow hotfix start fix-critical-bug

# Fix the bug
git add .
git commit -m "Fix critical bug"

# Finish hotfix
git flow hotfix finish fix-critical-bug
```

Git Flow Implementation

Repository Setup:

```bash
# Initialize git flow
git flow init

# Configure branch names
git config gitflow.branch.main main
git config gitflow.branch.develop develop
git config gitflow.prefix.feature feature/
git config gitflow.prefix.release release/
git config gitflow.prefix.hotfix hotfix/
```

Automated Git Flow:

```bash
#!/bin/bash
# Automated feature workflow

FEATURE_NAME=$1

if [ -z "$FEATURE_NAME" ]; then
    echo "Please provide feature name"
    exit 1
fi

# Start feature
git flow feature start $FEATURE_NAME

# Create initial commit
echo "# $FEATURE_NAME" > README_$FEATURE_NAME.md
git add README_$FEATURE_NAME.md
git commit -m "Start feature: $FEATURE_NAME"

echo "Feature branch created: feature/$FEATURE_NAME"
```

GitHub Flow

GitHub Flow is a simpler branching strategy that focuses on continuous deployment and is ideal for web applications and services that deploy frequently.

GitHub Flow Process

1. **Create Branch**: Create a descriptive branch from main
2. **Add Commits**: Make changes and commit regularly

3. **Open Pull Request**: Start discussion and review
4. **Review**: Collaborate and iterate
5. **Deploy**: Deploy from branch for testing
6. **Merge**: Merge to main after approval

GitHub Flow Implementation

Branch Creation:

```bash
# Create and switch to new branch
git checkout -b feature/user-authentication

# Make changes
git add .
git commit -m "Add user authentication"

# Push branch
git push -u origin feature/user-authentication
```

Pull Request Process:

```bash
# Create pull request via GitHub CLI
gh pr create --title "Add user authentication" --body "Implements login and logout functionality"

# Review and approve
gh pr review --approve

# Merge pull request
gh pr merge --squash
```

Automated GitHub Flow:

```yaml
```

```yaml
# .github/workflows/github-flow.yml
name: GitHub Flow

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    - name: Run tests
      run: npm test

  deploy-staging:
    if: github.event_name == 'pull_request'
    runs-on: ubuntu-latest
    needs: test
    steps:
    - name: Deploy to staging
      run: echo "Deploy to staging environment"

  deploy-production:
    if: github.ref == 'refs/heads/main'
    runs-on: ubuntu-latest
    needs: test
    steps:
    - name: Deploy to production
      run: echo "Deploy to production environment"
```

GitLab Flow

GitLab Flow combines feature-driven development with issue tracking and provides additional flexibility for different release scenarios.

GitLab Flow Variations

Environment Branches:

- `main` → `staging` → `production`
- Each environment has its own branch
- Deployments happen through merges

Release Branches:

- main → 2.1-stable → 2.0-stable
- Stable branches for each release
- Backports to older versions

GitLab Flow Implementation

Environment Branch Strategy:

```yaml
# .gitlab-ci.yml
stages:
  - test
  - deploy-staging
  - deploy-production

test:
  stage: test
  script:
    - npm test

deploy-staging:
  stage: deploy-staging
  script:
    - deploy-to-staging.sh
  only:
    - staging

deploy-production:
  stage: deploy-production
  script:
    - deploy-to-production.sh
  only:
    - production
  when: manual
```

Release Branch Strategy:

```bash
```

```
# Create release branch
git checkout -b 2.1-stable

# Cherry-pick features
git cherry-pick feature-commit-hash

# Push release branch
git push origin 2.1-stable
```

Feature Branch Workflow

A simple branching strategy where each feature is developed in its own branch and merged back to main through pull requests.

Feature Branch Best Practices

Branch Naming Conventions:

- `feature/JIRA-123-user-authentication`
- `bugfix/fix-login-error`
- `hotfix/security-patch`
- `refactor/optimize-database-queries`

Branch Management:

```
bash

# Create feature branch
git checkout -b feature/payment-integration

# Regular commits
git add .
git commit -m "Add payment gateway integration"

# Keep branch updated
git fetch origin
git rebase origin/main

# Push changes
git push origin feature/payment-integration
```

Code Review Process:

```
bash
```

```
# Create pull request
gh pr create --title "Add payment integration" \
  --body "Implements Stripe payment gateway with error handling"

# Address review comments
git add .
git commit -m "Address code review comments"
git push origin feature/payment-integration

# Merge after approval
gh pr merge --squash --delete-branch
```

Trunk-Based Development

Trunk-based development is a source-control branching model where developers collaborate on code in a single branch (trunk/main) and avoid long-lived feature branches.

Trunk-Based Development Principles

Short-Lived Branches:

- Feature branches live for hours or days, not weeks
- Frequent integration with main branch
- Small, incremental changes

Continuous Integration:

- Automated testing on every commit
- Build and deployment automation
- Fast feedback loops

Feature Flags:

- Deploy incomplete features behind flags
- Gradual rollout of new features
- A/B testing capabilities

Trunk-Based Implementation

Daily Integration:

bash

```
# Start of day - sync with main
git checkout main
git pull origin main

# Create short-lived branch
git checkout -b quick-feature

# Make small changes
git add .
git commit -m "Small incremental change"

# Integrate quickly
git checkout main
git pull origin main
git merge quick-feature
git push origin main

# Clean up
git branch -d quick-feature
```

Feature Flags Implementation:

javascript

```
// Feature flag service
class FeatureFlag {
  static isEnabled(flagName, userId) {
    // Check feature flag status
    return this.flags[flagName] && this.isUserInRollout(userId);
  }

  static isUserInRollout(userId) {
    // Determine if user is in rollout percentage
    return (userId.hashCode() % 100) < this.rolloutPercentage;
  }
}

// Usage in application
if (FeatureFlag.isEnabled('new-payment-system', user.id)) {
  // Use new payment system
  return newPaymentService.processPayment(amount);
} else {
  // Use old payment system
  return oldPaymentService.processPayment(amount);
}
```

Automated Trunk-Based Workflow:

```yaml
# .github/workflows/trunk-based.yml
name: Trunk-Based Development

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    - name: Run tests
      run: |
        npm install
        npm test
        npm run integration-test

  deploy:
    if: github.ref == 'refs/heads/main'
    needs: test
    runs-on: ubuntu-latest
    steps:
    - name: Deploy to production
      run: |
        # Deploy with feature flags
        kubectl apply -f k8s/
        kubectl set env deployment/app FEATURE_FLAGS="${{ secrets.FEATURE_FLAGS }}"
```

Branch Protection and Policies

GitHub Branch Protection

Protection Rules Configuration:

```bash
```

```
# Using GitHub CLI
gh api repos/:owner/:repo/branches/main/protection \
  --method PUT \
  --field required_status_checks='{"strict":true,"contexts":["ci/test"]}' \
  --field enforce_admins=true \
  --field required_pull_request_reviews='{"required_approving_review_count":2}' \
  --field restrictions=null
```

Branch Protection Settings:

- Require pull request reviews
- Require status checks to pass
- Require branches to be up to date
- Require signed commits
- Restrict pushes to matching branches
- Allow force pushes
- Allow deletions

GitLab Push Rules

Push Rule Configuration:

```yaml
```

```yaml
# .gitlab-ci.yml
include:
  - template: Security/SAST.gitlab-ci.yml
  - template: Security/Dependency-Scanning.gitlab-ci.yml

variables:
  SAST_EXCLUDED_ANALYZERS: "spotbugs"

stages:
  - test
  - security
  - deploy

test:
  stage: test
  script:
    - npm test
  rules:
    - if: '$CI_MERGE_REQUEST_ID'
    - if: '$CI_COMMIT_BRANCH == "main"'

security:
  stage: security
  dependencies: []
  rules:
    - if: '$CI_MERGE_REQUEST_ID'
    - if: '$CI_COMMIT_BRANCH == "main"'
```

Merge Request Approvals:

- Required approvers
- Code owner approvals
- Security team approval
- Approval rules by file changes

Bitbucket Branch Permissions

Branch Permission Configuration:

```
json
```

```
{
  "type": "restrict",
  "pattern": "main",
  "users": [],
  "groups": ["developers"],
  "accessKeys": [],
  "kind": "push",
  "value": "allow"
}
```

Merge Checks:

- Minimum approvals required
- Reset approvals on source branch changes
- Dismiss stale approvals
- Require tasks to be resolved
- Check for merge conflicts

Branching Strategy Selection Guide

Project Characteristics Assessment

Team Size Considerations:

- Small teams (1-5): GitHub Flow or Feature Branch
- Medium teams (6-15): GitLab Flow or Git Flow
- Large teams (16+): Trunk-based with feature flags

Release Frequency:

- Continuous deployment: GitHub Flow or Trunk-based
- Weekly releases: GitLab Flow
- Monthly/quarterly: Git Flow

Product Type:

- Web applications: GitHub Flow
- Mobile apps: Git Flow
- Enterprise software: GitLab Flow
- Open source: Feature Branch

Decision Matrix

Factor	Git Flow	GitHub Flow	GitLab Flow	Trunk-based
Complexity	High	Low	Medium	Medium
Learning Curve	Steep	Gentle	Moderate	Moderate
Release Frequency	Scheduled	Continuous	Flexible	Continuous
Team Size	Large	Small-Medium	Any	Large
Deployment Risk	Low	Medium	Low	High
Feature Flags	Optional	Optional	Optional	Required

Implementation Checklist

Pre-Implementation:

- [ ] Assess team size and experience
- [ ] Evaluate release requirements
- [ ] Consider deployment infrastructure
- [ ] Review compliance requirements
- [ ] Plan training and documentation

During Implementation:

- [ ] Configure branch protection rules
- [ ] Set up CI/CD pipelines
- [ ] Implement code review process
- [ ] Create branch naming conventions
- [ ] Establish merge policies

Post-Implementation:

- [ ] Monitor branch metrics
- [ ] Gather team feedback
- [ ] Optimize workflows
- [ ] Update documentation
- [ ] Continuous improvement

Chapter 8: Code Review Best Practices

Code Review Fundamentals

Purpose of Code Reviews

Quality Assurance:

- Catch bugs and defects early

- Ensure code meets standards
- Verify functionality requirements
- Maintain architectural consistency

Knowledge Sharing:

- Spread domain knowledge
- Share coding techniques
- Align on conventions
- Mentor junior developers

Collaboration:

- Foster team communication
- Build collective ownership
- Improve team dynamics
- Establish trust

Code Review Process

Pre-Review Preparation:

1. Complete feature development
2. Write comprehensive tests
3. Update documentation
4. Self-review changes
5. Ensure CI passes

Review Execution:

1. Understand the context
2. Review systematically
3. Focus on important aspects
4. Provide constructive feedback
5. Approve or request changes

Post-Review Actions:

1. Address feedback
2. Update code as needed
3. Respond to comments

4. Merge when approved
5. Follow up on action items

Code Review Guidelines

What to Review

Functionality:

- Code correctness
- Logic implementation
- Edge case handling
- Error management
- Performance implications

Design and Architecture:

- Code structure
- Design patterns
- SOLID principles
- Maintainability
- Scalability

Style and Standards:

- Coding conventions
- Naming conventions
- Documentation
- Test coverage
- Security practices

Code Review Checklist

General:

- [ ] Code compiles without warnings
- [ ] All tests pass
- [ ] Code follows style guidelines
- [ ] Documentation is updated
- [ ] No sensitive information exposed

Functionality:

- [ ] Requirements are met
- [ ] Edge cases are handled
- [ ] Error conditions are managed
- [ ] Performance is acceptable
- [ ] Security is considered

Design:

- [ ] Code is well-structured
- [ ] Appropriate abstractions
- [ ] No code duplication
- [ ] Proper separation of concerns
- [ ] Follows established patterns

Testing:

- [ ] Adequate test coverage
- [ ] Tests are meaningful
- [ ] Tests are maintainable
- [ ] Integration tests included
- [ ] Performance tests if needed

Review Comment Guidelines

Constructive Feedback:

```
// Good: Specific and actionable
"Consider using a constant for this magic number (42)
to improve readability and maintainability."

// Bad: Vague and unhelpful
"This is wrong."
```

Asking Questions:

```
// Good: Encouraging discussion
"What's the reasoning behind this approach?
Would using a factory pattern be beneficial here?"

// Bad: Accusatory
"Why did you do it this way?"
```

Suggesting Improvements:

```
// Good: Providing alternatives
"This could be simplified using array destructuring:
const [first, second] = items;"

// Bad: Just pointing out problems
"This code is too complex."
```

Code Review Tools

GitHub Pull Request Reviews:

```yaml
# .github/pull_request_template.md
## Description
Brief description of changes

## Type of Change
- [ ] Bug fix
- [ ] New feature
- [ ] Breaking change
- [ ] Documentation update

## Testing
- [ ] Unit tests added/updated
- [ ] Integration tests added/updated
- [ ] Manual testing performed

## Checklist
- [ ] Code follows style guidelines
- [ ] Self-review performed
- [ ] Documentation updated
- [ ] No console.log statements
```

GitLab Merge Request Reviews:

```yaml
```

```
# .gitlab/merge_request_templates/Default.md
## What does this MR do?
Describe the changes in detail

## Related issues
Closes #issue-number

## Author's checklist
- [ ] Follow the style guide
- [ ] Add tests for new functionality
- [ ] Update documentation
- [ ] Check for security issues

## Review checklist
- [ ] Code quality is maintained
- [ ] Tests are comprehensive
- [ ] Documentation is clear
- [ ] No performance regressions
```

Automated Code Review

Static Code Analysis

ESLint Configuration:

```json
{
  "extends": [
    "eslint:recommended",
    "@typescript-eslint/recommended",
    "prettier"
  ],
  "rules": {
    "no-console": "error",
    "prefer-const": "error",
    "no-unused-vars": "error",
    "complexity": ["error", 10],
    "max-depth": ["error", 4]
  }
}
```

SonarQube Integration:

```yaml
```

```yaml
# .github/workflows/sonarqube.yml
name: SonarQube Analysis

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  sonarqube:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
      with:
        fetch-depth: 0

    - name: SonarQube Scan
      uses: sonarqube-quality-gate-action@master
      env:
        SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
```

CodeClimate Configuration:

```yaml
```

```yaml
# .codeclimate.yml
version: "2"
checks:
  argument-count:
    enabled: true
    config:
      threshold: 4
  complex-logic:
    enabled: true
    config:
      threshold: 4
  method-complexity:
    enabled: true
    config:
      threshold: 5
  method-count:
    enabled: true
    config:
      threshold: 20
  method-lines:
    enabled: true
    config:
      threshold: 25
```

Security Code Review

Security Scanning Tools:

```yaml
```

```yaml
# .github/workflows/security.yml
name: Security Scan

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  security:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3

    - name: SAST Scan
      uses: github/codeql-action/init@v2
      with:
        languages: javascript, typescript

    - name: Build
      run: npm run build

    - name: CodeQL Analysis
      uses: github/codeql-action/analyze@v2

    - name: Dependency Check
      run: npm audit --audit-level moderate
```

Security Review Checklist:

- [ ] Input validation implemented
- [ ] SQL injection prevention
- [ ] XSS protection
- [ ] Authentication/authorization
- [ ] Sensitive data handling
- [ ] Error handling doesn't expose info
- [ ] Dependencies are up to date
- [ ] No hardcoded secrets

Performance Code Review

Performance Analysis:

```javascript
```

```
// Performance testing example
const { performance } = require('perf_hooks');

function measurePerformance(fn, iterations = 1000) {
  const start = performance.now();

  for (let i = 0; i < iterations; i++) {
    fn();
  }

  const end = performance.now();
  return end - start;
}

// Usage in tests
describe('Performance Tests', () => {
  it('should execute within acceptable time', () => {
    const executionTime = measurePerformance(() => {
      // Function to test
      processLargeDataSet(testData);
    });

    expect(executionTime).toBeLessThan(100); // 100ms threshold
  });
});
```

Performance Review Guidelines:

- [ ] Algorithm efficiency
- [ ] Memory usage optimization
- [ ] Database query optimization
- [ ] Network request minimization
- [ ] Caching implementation
- [ ] Resource cleanup
- [ ] Async/await usage

Review Workflow Automation

Automated Review Assignment

GitHub CODEOWNERS:

```
# .github/CODEOWNERS
# Global owners
* @dev-team

# Frontend code
/frontend/ @frontend-team

# Backend code
/backend/ @backend-team

# Database migrations
/migrations/ @database-team @backend-team

# CI/CD configuration
/.github/ @devops-team
/docker/ @devops-team
```

GitLab Code Owners:

```
# .gitlab/CODEOWNERS
# Default owners
* @maintainers

# Documentation
/docs/ @tech-writers @maintainers

# Security-related files
/security/ @security-team
```

Review Automation Rules

GitHub Actions for Reviews:

```yaml
```

```yaml
name: Auto Review

on:
  pull_request:
    types: [opened, synchronize]

jobs:
  auto-review:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3

    - name: Auto-assign reviewers
      uses: kentaro-m/auto-assign-action@v1.2.1
      with:
        configuration-path: '.github/auto-assign.yml'

    - name: Label PR
      uses: actions/labeler@v4
      with:
        repo-token: ${{ secrets.GITHUB_TOKEN }}
        configuration-path: '.github/labeler.yml'
```

Auto-assign Configuration:

```yaml
# .github/auto-assign.yml
addReviewers: true
addAssignees: false
reviewers:
  - senior-dev-1
  - senior-dev-2
numberOfReviewers: 2
```

Quality Gates

Merge Requirements:

```yaml
```

```yaml
# .github/workflows/quality-gate.yml
name: Quality Gate

on:
  pull_request:
    branches: [ main ]

jobs:
  quality-check:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3

    - name: Test Coverage
      run: |
        npm test -- --coverage
        npx nyc check-coverage --lines 80 --functions 80 --branches 80

    - name: Code Quality
      run: |
        npm run lint
        npm run type-check

    - name: Security Check
      run: npm audit --audit-level high

    - name: Performance Check
      run: npm run performance-test
```

Merge Protection Rules:

```
json
```

```
{
  "required_status_checks": {
    "strict": true,
    "contexts": [
      "ci/test",
      "ci/lint",
      "ci/security",
      "ci/performance"
    ]
  },
  "required_pull_request_reviews": {
    "required_approving_review_count": 2,
    "dismiss_stale_reviews": true,
    "require_code_owner_reviews": true
  },
  "enforce_admins": true,
  "restrictions": null
}
```

Code Review Metrics

Key Metrics to Track

Review Efficiency:

- Time to first review
- Review cycle time
- Review participation rate
- Review coverage percentage

Code Quality:

- Defect escape rate
- Code coverage
- Technical debt ratio
- Security vulnerabilities

Team Collaboration:

- Review comment sentiment
- Knowledge sharing index
- Mentor-mentee interactions
- Cross-team reviews

Metrics Collection

GitHub Metrics:

javascript

```javascript
// GitHub API script for metrics
const { Octokit } = require("@octokit/rest");

const octokit = new Octokit({
  auth: process.env.GITHUB_TOKEN
});

async function getReviewMetrics(owner, repo) {
  const pulls = await octokit.pulls.list({
    owner,
    repo,
    state: 'closed',
    per_page: 100
  });

  let totalReviewTime = 0;
  let reviewCount = 0;

  for (const pull of pulls.data) {
    const reviews = await octokit.pulls.listReviews({
      owner,
      repo,
      pull_number: pull.number
    });

    if (reviews.data.length > 0) {
      const createdAt = new Date(pull.created_at);
      const firstReview = new Date(reviews.data[0].submitted_at);
      const reviewTime = firstReview - createdAt;

      totalReviewTime += reviewTime;
      reviewCount++;
    }
  }

  return {
    averageReviewTime: totalReviewTime / reviewCount,
    totalReviews: reviewCount
  };
}
```

Dashboard Example:

```javascript
// Simple metrics dashboard
function createReviewDashboard(metrics) {
  return {
    overview: {
      totalPullRequests: metrics.totalPRs,
      averageReviewTime: `${metrics.avgReviewTime}h`,
      reviewParticipation: `${metrics.participation}%`
    },
    quality: {
      defectEscapeRate: `${metrics.defectRate}%`,
      coverageIncrease: `${metrics.coverageIncrease}%`,
      securityIssues: metrics.securityIssues
    },
    team: {
      activeReviewers: metrics.activeReviewers,
      knowledgeSharing: metrics.knowledgeSharing,
      crossTeamReviews: metrics.crossTeamReviews
    }
  };
}
```

Chapter 9: CI/CD Principles

Continuous Integration Fundamentals

What is Continuous Integration?

Continuous Integration (CI) is a development practice where developers integrate code into a shared repository frequently, preferably several times a day. Each integration is verified by an automated build and testing process to detect integration errors as quickly as possible.

Core CI Principles

Frequent Integration:

- Developers commit code multiple times daily
- Early detection of integration issues
- Smaller, manageable changesets
- Reduced merge conflicts

Automated Testing:

- Comprehensive test suite execution
- Fast feedback on code changes
- Consistent testing environment
- Multiple testing levels

Build Automation:

- Automated compilation and packaging
- Dependency management
- Environment configuration
- Artifact generation

Fast Feedback:

- Quick build and test execution
- Immediate notification of failures
- Easy access to build results
- Clear failure diagnostics

CI Implementation Strategy

Repository Setup:

```
bash

# Project structure for CI
project/
├── src/
│   ├── main/
│   └── test/
├── docs/
├── scripts/
│   ├── build.sh
│   ├── test.sh
│   └── deploy.sh
├── ci/
│   ├── Dockerfile
│   └── docker-compose.yml
├── .github/
│   └── workflows/
├── Jenkinsfile
└── README.md
```

Build Script Example:

```bash
#!/bin/bash
# build.sh

set -e  # Exit on any error

echo "Starting build process..."

# Clean previous builds
rm -rf dist/ build/

# Install dependencies
npm ci

# Run linting
npm run lint

# Run tests
npm test

# Build application
npm run build

# Run integration tests
npm run test:integration

echo "Build completed successfully!"
```

Continuous Deployment Fundamentals

What is Continuous Deployment?

Continuous Deployment (CD) is a software release process that uses automated testing to validate if changes to a codebase are correct and stable for immediate autonomous deployment to production.

CD vs Continuous Delivery

Continuous Delivery:

- Automated deployment to staging
- Manual approval for production
- Release-ready code at all times
- Human decision for release timing

Continuous Deployment:

- Fully automated deployment pipeline
- Automatic production deployment
- No manual intervention required
- Immediate release of validated changes

Deployment Strategies

Blue-Green Deployment:

bash

```bash
#!/bin/bash
# Blue-green deployment script

CURRENT_ENV=$(kubectl get service app-service -o jsonpath='{.spec.selector.version}')
NEW_ENV=$([ "$CURRENT_ENV" = "blue" ] && echo "green" || echo "blue")

echo "Current environment: $CURRENT_ENV"
echo "Deploying to: $NEW_ENV"

# Deploy to new environment
kubectl apply -f k8s/deployment-$NEW_ENV.yml

# Wait for deployment to be ready
kubectl rollout status deployment/app-$NEW_ENV

# Run health checks
if curl -f http://app-$NEW_ENV.internal/health; then
  echo "Health check passed, switching traffic"
  kubectl patch service app-service -p '{"spec":{"selector":{"version":"'$NEW_ENV'"}}}'
  echo "Traffic switched to $NEW_ENV"
else
  echo "Health check failed, rolling back"
  kubectl delete deployment app-$NEW_ENV
  exit 1
fi
```

Canary Deployment:

yaml

```yaml
# Canary deployment configuration
apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: app-rollout
spec:
  replicas: 10
  strategy:
    canary:
      steps:
      - setWeight: 10
      - pause: {duration: 60s}
      - setWeight: 50
      - pause: {duration: 60s}
      - setWeight: 100
      analysis:
        templates:
        - templateName: success-rate
        args:
        - name: service-name
          value: app-service
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp
        image: myapp:latest
```

Rolling Deployment:

```yaml
```

```yaml
# Rolling deployment configuration
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-deployment
spec:
  replicas: 5
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 2
      maxUnavailable: 1
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp
        image: myapp:latest
        readinessProbe:
          httpGet:
            path: /health
            port: 8080
          initialDelaySeconds: 30
          periodSeconds: 5
```

Pipeline Design Principles

Pipeline Architecture

Linear Pipeline:

```
Code → Build → Test → Deploy → Monitor
```

Parallel Pipeline:

```
Code → Build → [Unit Tests, Integration Tests, Security Scans] → Deploy → Monitor
```

Fan-in/Fan-out Pipeline:

```
Code → Build → [Frontend Tests, Backend Tests, E2E Tests] → Merge → Deploy → Monitor
```

Stage Design

Build Stage:

- Source code compilation
- Dependency resolution
- Asset optimization
- Artifact creation
- Version tagging

Test Stage:

- Unit testing
- Integration testing
- Performance testing
- Security testing
- Code quality checks

Deploy Stage:

- Environment preparation
- Application deployment
- Configuration updates
- Health checks
- Rollback capabilities

Pipeline Configuration

GitHub Actions Pipeline:

```yaml
```

```yaml
name: CI/CD Pipeline

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]

env:
  NODE_VERSION: '18'
  DOCKER_REGISTRY: ghcr.io
  IMAGE_NAME: ${{ github.repository }}

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3

    - name: Setup Node.js
      uses: actions/setup-node@v3
      with:
        node-version: ${{ env.NODE_VERSION }}
        cache: 'npm'

    - name: Install dependencies
      run: npm ci

    - name: Run linter
      run: npm run lint

    - name: Run tests
      run: npm test -- --coverage

    - name: Upload coverage
      uses: codecov/codecov-action@v3

  build:
    needs: test
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3

    - name: Setup Docker Buildx
      uses: docker/setup-buildx-action@v2
```

```yaml
    - name: Login to Container Registry
      uses: docker/login-action@v2
      with:
        registry: ${{ env.DOCKER_REGISTRY }}
        username: ${{ github.actor }}
        password: ${{ secrets.GITHUB_TOKEN }}

    - name: Build and push
      uses: docker/build-push-action@v4
      with:
        context: .
        push: true
        tags: ${{ env.DOCKER_REGISTRY }}/${{ env.IMAGE_NAME }}:${{ github.sha }}
        cache-from: type=gha
        cache-to: type=gha,mode=max

  deploy:
    needs: [test, build]
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'
    environment: production
    steps:
    - uses: actions/checkout@v3

    - name: Deploy to Kubernetes
      run: |
        echo "${{ secrets.KUBECONFIG }}" | base64 -d > kubeconfig
        export KUBECONFIG=kubeconfig

        sed -i "s/IMAGE_TAG/${{ github.sha }}/g" k8s/deployment.yml
        kubectl apply -f k8s/
        kubectl rollout status deployment/app-deployment
```

GitLab CI Pipeline:

```yaml
```

```yaml
stages:
  - test
  - build
  - deploy

variables:
  DOCKER_DRIVER: overlay2
  DOCKER_TLS_CERTDIR: "/certs"

.base_job: &base_job
  image: node:18-alpine
  cache:
    key: ${CI_COMMIT_REF_SLUG}
    paths:
      - node_modules/
      - .npm/

test:
  <<: *base_job
  stage: test
  script:
    - npm ci --cache .npm --prefer-offline
    - npm run lint
    - npm test -- --coverage
    - npm run test:integration
  artifacts:
    reports:
      coverage_report:
        coverage_format: cobertura
        path: coverage/cobertura-coverage.xml
    expire_in: 1 hour
  rules:
    - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
    - if: '$CI_COMMIT_BRANCH == "main"'

build:
  stage: build
  image: docker:20.10.16
  services:
    - docker:20.10.16-dind
  before_script:
    - echo $CI_REGISTRY_PASSWORD | docker login -u $CI_REGISTRY_USER --password-stdin $CI_REGISTRY
  script:
    - docker build -t $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA .
    - docker push $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
  needs: [test]
```

```yaml
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'

deploy:
  stage: deploy
  image: bitnami/kubectl:latest
  script:
    - echo "$KUBECONFIG" | base64 -d > kubeconfig
    - export KUBECONFIG=kubeconfig
    - sed -i "s/IMAGE_TAG/$CI_COMMIT_SHA/g" k8s/deployment.yml
    - kubectl apply -f k8s/
    - kubectl rollout status deployment/app-deployment
  environment:
    name: production
    url: https://app.example.com
  needs: [build]
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
      when: manual
```

Pipeline Optimization

Performance Optimization

Parallel Execution:

```yaml
jobs:
  test:
    strategy:
      matrix:
        node-version: [16, 18, 20]
        os: [ubuntu-latest, windows-latest, macos-latest]
    runs-on: ${{ matrix.os }}
    steps:
    - uses: actions/checkout@v3
    - uses: actions/setup-node@v3
      with:
        node-version: ${{ matrix.node-version }}
    - run: npm ci
    - run: npm test
```

Caching Strategy:

```yaml
```

```yaml
- name: Cache node modules
  uses: actions/cache@v3
  with:
    path: ~/.npm
    key: ${{ runner.os }}-node-${{ hashFiles('**/package-lock.json') }}
    restore-keys: |
      ${{ runner.os }}-node-

- name: Cache Docker layers
  uses: actions/cache@v3
  with:
    path: /tmp/.buildx-cache
    key: ${{ runner.os }}-buildx-${{ github.sha }}
    restore-keys: |
      ${{ runner.os }}-buildx-
```

Build Optimization:

dockerfile

```dockerfile
# Multi-stage build for optimization
FROM node:18-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production

FROM node:18-alpine AS runtime
WORKDIR /app
COPY --from=builder /app/node_modules ./node_modules
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

Quality Gates

Test Coverage Gates:

yaml

```yaml
- name: Check coverage
  run: |
    npm test -- --coverage
    npx nyc check-coverage --lines 80 --functions 80 --branches 80 --statements 80
```

Security Gates:

```yaml
- name: Security audit
  run: |
    npm audit --audit-level high
    npx snyk test --severity-threshold=high
```

Performance Gates:

```yaml
- name: Performance test
  run: |
    npm run build
    npm run test:performance
    npx lighthouse-ci --assert --preset=ci
```

Monitoring and Observability

Pipeline Monitoring

Metrics Collection:

```yaml
- name: Collect metrics
  run: |
    echo "BUILD_DURATION=$(($(date +%s) - $BUILD_START_TIME))" >> $GITHUB_ENV
    echo "TEST_RESULTS=$(cat test-results.json)" >> $GITHUB_ENV
    echo "COVERAGE_PERCENTAGE=$(grep -o 'Lines.*[0-9]*\.[0-9]*%' coverage/text-summary.txt | grep -o '[0-9
```

Notification Setup:

```yaml
- name: Notify on failure
  if: failure()
  uses: 8398a7/action-slack@v3
  with:
    status: failure
    channel: '#ci-alerts'
    webhook_url: ${{ secrets.SLACK_WEBHOOK }}
```

Application Monitoring

Health Checks:

javascript

```javascript
// Health check endpoint
app.get('/health', (req, res) => {
  const healthCheck = {
    uptime: process.uptime(),
    message: 'OK',
    timestamp: Date.now(),
    checks: {
      database: checkDatabase(),
      redis: checkRedis(),
      externalAPI: checkExternalAPI()
    }
  };

  const allChecksPass = Object.values(healthCheck.checks).every(check => check.status === 'OK');

  res.status(allChecksPass ? 200 : 503
```