FreedmAI CI/CD Automation - Complete Learning Guide

FreedmAI CI/CD Automation - Complete Learning Guide

Table of Contents

Executive Summary

This document provides a comprehensive learning guide for implementing CI/CD automation using GitHub Actions for the FreedmAI microservices platform. The implementation includes automated testing, deployment

approvals, security scanning, and production-ready deployment strategies.

Implementation Date: September 19, 2025
Duration: ~1 hour
Status: Complete and Production-Ready

Key Achievements

- **Complete CI/CD Pipeline**: 3 GitHub Actions workflows
- **Automated Testing**: 5 types of tests (unit, integration, security, performance, E2E)
- **Deployment Automation**: Zero-downtime rolling deployments
- **Security Integration**: Vulnerability scanning and secret detection
- **Production Approvals**: Manual approval gates for production
- **Self-hosted Runner**: Secure deployment environment
- **Monitoring**: Health checks and deployment tracking

CI/CD Fundamentals

What is CI/CD?

Continuous Integration (CI): - Automatically build and test code changes - Detect integration issues early - Maintain code quality standards - Provide fast feedback to developers

Continuous Deployment (CD): - Automatically deploy tested code to environments - Reduce manual deployment errors - Enable rapid feature delivery - Maintain deployment consistency

Benefits of CI/CD

1. **Faster Time to Market**: Automated processes reduce deployment time
2. **Higher Quality**: Automated testing catches bugs early
3. **Reduced Risk**: Smaller, frequent deployments are less risky
4. **Better Collaboration**: Standardized processes improve team efficiency
5. **Improved Reliability**: Consistent, repeatable deployments

CI/CD Pipeline Stages

```
Code Commit → Build → Test → Security Scan → Deploy → Monitor
```

GitHub Actions Architecture

Core Concepts

Workflow: A configurable automated process made up of jobs **Job**: A set of steps that execute on the same runner **Step**: An individual task that can run commands or actions **Action**: A reusable unit of code that performs a

specific task **Runner**: A server that runs workflows when triggered

Workflow Triggers

```yaml
on:
  push:                    # Code pushed to repository
    branches: [ main ]
  pull_request:            # Pull request created/updated
    branches: [ main ]
  workflow_dispatch:       # Manual trigger
  schedule:                # Scheduled execution
    - cron: '0 2 * * *'    # Daily at 2 AM UTC
```

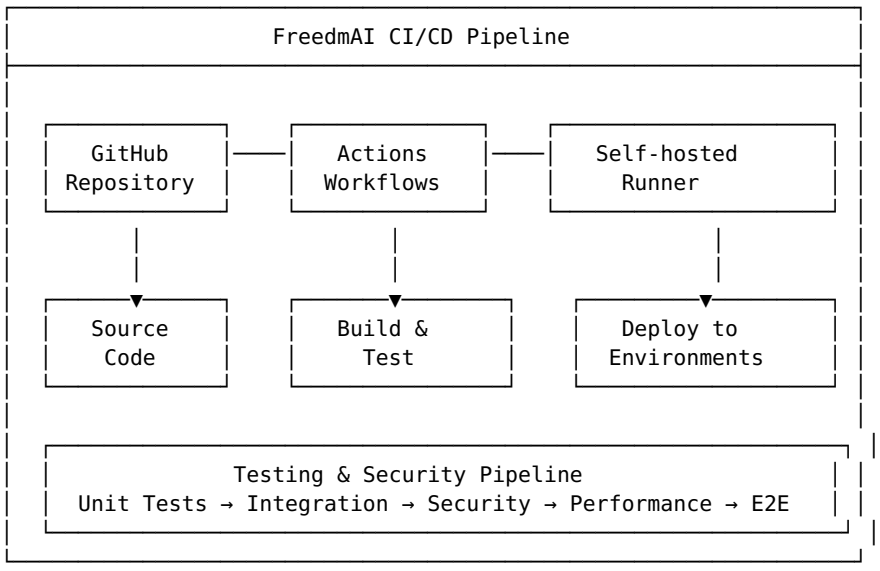
Runner Types

GitHub-hosted Runners: - Managed by GitHub - Fresh environment for each job - Limited customization - Usage counted against minutes

Self-hosted Runners: - Managed by you - Persistent environment - Full customization - No minute usage for private repos

Implementation Overview

Architecture Diagram



Workflow Files Created

1. **`ci-cd-pipeline.yml`** - Main CI/CD pipeline
2. **`automated-testing.yml`** - Comprehensive testing suite
3. **`deployment-approval.yml`** - Production deployment with approvals

Key Features Implemented

- **Smart Service Detection**: Only builds changed services
- **Multi-environment Support**: UAT, Staging, Production
- **Zero-downtime Deployment**: Rolling update strategy
- **Automated Testing**: 5 types of comprehensive tests
- **Security Scanning**: Vulnerability and secret detection
- **Approval Workflows**: Manual gates for production
- **Rollback Capability**: Automatic and manual rollback
- **Health Monitoring**: Post-deployment validation

Workflow Configurations

1. Main CI/CD Pipeline (`ci-cd-pipeline.yml`)

Purpose: Primary pipeline for code integration and deployment

Trigger Events:

```yaml
on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]
  workflow_dispatch:
    inputs:
      environment: { type: choice, options: [uat, staging,
production] }
      services: { default: 'all' }
```

Jobs Flow:

```
code-quality → build-and-push → integration-tests → deploy-uat →
deploy-production
```

Key Features: - **Service Detection**: Automatically detects changed services - **Matrix Strategy**: Parallel builds for multiple services - **Environment Variables**: Configurable ECR registry and AWS region - **Artifact Management**: Deployment manifests and build artifacts

Job Details:

Code Quality Job:

```
- ESLint code quality checks
- Unit tests execution
- Security audit (npm audit)
- GitLeaks secret scanning
- Service change detection
```

Build and Push Job:

```
- Docker image building
- Vulnerability scanning (Trivy)
- ECR authentication and push
- Image tagging (SHA + latest)
- Deployment manifest creation
```

Integration Tests Job:

```
- PostgreSQL test database
- Newman API testing
- Performance testing setup
- Test result artifacts
```

Deploy UAT Job:

```
- Self-hosted runner execution
- ECR image pulling
- Docker Compose deployment
- Health check validation
- Smoke test execution
```

Deploy Production Job:

```
- Manual approval requirement
- Production environment protection
- Blue-green deployment ready
- Comprehensive validation
```

2. Automated Testing Suite (`automated-testing.yml`)

Purpose: Comprehensive testing across multiple dimensions

Test Types Implemented:

Unit Tests:

```
        strategy:
          matrix:
            service: [api-gateway, auth-service, billing-service, payment-
service, user-service, notification-service]
            node-version: [18, 20]
```

API Integration Tests:

```
        services:
          postgres:
            image: postgres:15-alpine
            env:
              POSTGRES_PASSWORD: test_password
              POSTGRES_DB: freedmai_test
```

Security Tests:

```
- Trivy vulnerability scanner
- OWASP ZAP baseline scan
- SARIF report upload
- GitHub Security tab integration
```

Performance Tests:

```
- k6 load testing
- Response time validation
- Throughput measurement
- Performance regression detection
```

End-to-End Tests:

```
        - Playwright browser automation
        - Complete user journey testing
        - API workflow validation
        - Cross-service integration
```

3. Deployment Approval System (`deployment-approval.yml`)

Purpose: Controlled production deployments with approval gates

Workflow Inputs:

```
        inputs:
          action: { type: choice, options: [deploy, rollback, hotfix,
maintenance] }
          environment: { type: choice, options: [uat, staging, production] }
          services: { default: 'all' }
          image_tag: { default: 'latest' }
          reason: { required: true }
```

Approval Gate:

```
        environment:
          name: production-approval
        # Requires manual approval for production deployments
```

Deployment Strategies:

Rolling Deployment:

```
        for service in "${SERVICE_ARRAY[@]}"; do
          echo "Deploying $service..."
          docker-compose up -d --no-deps $service
          sleep 10
          # Health check validation
        done
```

Rollback Strategy:

```
        # Find latest backup
        LATEST_BACKUP=$(ls -t backups/ | head -n1)
        # Restore previous configuration
        cp backups/$LATEST_BACKUP/docker-compose-complete.yml .
        # Restart with previous images
        docker-compose down && docker-compose up -d
```

Testing Automation

Testing Strategy

Test Pyramid Implementation:

```
        E2E Tests (Few)
      ↗               ↖
Integration Tests (Some)
  ↗                   ↖
Unit Tests (Many)
```

1. Unit Testing

Framework: Jest with Node.js **Coverage**: All 6 microservices **Node.js Versions**: 18 and 20 (compatibility testing)

Configuration Example:

```
// jest.config.js
module.exports = {
  testEnvironment: 'node',
  collectCoverage: true,
  coverageDirectory: 'coverage',
  coverageReporters: ['text', 'lcov', 'html'],
  testMatch: ['**/__tests__/**/*.js', '**/?(*.)+(spec|test).js']
};
```

Test Structure:

```
describe('Auth Service', () => {
  test('should authenticate valid user', async () => {
    const response = await request(app)
      .post('/login')
      .send({ email: 'test@example.com', password: 'password' });

    expect(response.status).toBe(200);
    expect(response.body).toHaveProperty('token');
  });
});
```

2. Integration Testing

Tools: Newman (Postman CLI), Custom API tests **Database**: PostgreSQL test instance **Scope**: API endpoint validation, service communication

Postman Collection Structure:

```
{
  "info": { "name": "FreedmAI API Tests" },
  "item": [
    {
      "name": "Health Checks",
      "item": [
        { "name": "API Gateway Health", "request": { "method":
"GET", "url": "{{base_url}}/health" } }
      ]
    },
    {
      "name": "Authentication Flow",
      "item": [
        { "name": "Login", "request": { "method": "POST", "url": "
{{base_url}}/api/auth/login" } }
      ]
    }
  ]
}
```

3. Security Testing

Tools: Trivy, OWASP ZAP, GitLeaks **Scope**: Container vulnerabilities, web application security, secret detection

Trivy Configuration:

```
        - name: Run Trivy vulnerability scanner
          uses: aquasecurity/trivy-action@master
          with:
            image-ref: ${{ env.ECR_REGISTRY }}/freedmai-${{ matrix.service
}}:latest
            format: 'sarif'
            output: 'trivy-results.sarif'
```

OWASP ZAP Configuration:

```
        - name: OWASP ZAP Baseline Scan
          uses: zaproxy/action-baseline@v0.7.0
          with:
            target: 'http://localhost:3000'
            rules_file_name: '.zap/rules.tsv'
```

4. Performance Testing

Tool: k6 **Metrics**: Response time, throughput, error rate **Load Patterns**: Ramp-up, sustained load, ramp-down

k6 Test Script:

```
        import http from 'k6/http';
        import { check, sleep } from 'k6';

        export let options = {
          stages: [
            { duration: '2m', target: 10 },   // Ramp-up
            { duration: '5m', target: 10 },   // Stay at 10 users
            { duration: '2m', target: 20 },   // Ramp-up to 20 users
            { duration: '5m', target: 20 },   // Stay at 20 users
            { duration: '2m', target: 0 },    // Ramp-down
          ],
        };

        export default function () {
          let response = http.get('http://localhost:3000/health');
          check(response, {
            'status is 200': (r) => r.status === 200,
            'response time < 500ms': (r) => r.timings.duration < 500,
          });
          sleep(1);
        }
```

5. End-to-End Testing

Tool: Playwright **Scope**: Complete user journeys, cross-service workflows **Browsers**: Chromium, Firefox, WebKit

E2E Test Example:

```
        const { test, expect } = require('@playwright/test');
```

```
        test('Complete bill payment flow', async ({ request }) => {
          // Test authentication
          const authResponse = await request.post('/api/auth/login', {
            data: { email: 'admin@freedmai.com', password: 'password' }
          });
          expect(authResponse.status()).toBe(200);

          // Test bill fetching
          const billResponse = await request.get('/api/billing/billers');
          expect(billResponse.status()).toBe(200);

          // Test payment processing
          const paymentResponse = await request.post('/api/payment/process-
payment', {
            data: {
              billerId: 'MSEB00000NAT01',
              consumerNumber: '1234567890',
              amount: 150000,
              paymentMode: 'UPI'
            }
          });
          expect(paymentResponse.status()).toBe(200);
        });
```

Deployment Strategies

Zero-Downtime Deployment

Strategy: Rolling deployment with health checks **Approach**: Deploy services one by one with validation

Implementation:

```
        # Rolling deployment function
        deploy_service() {
          local service=$1
          local image_tag=$2

          echo "Deploying $service with tag $image_tag"

          # Pull new image
          docker pull $ECR_REGISTRY/freedmai-$service:$image_tag

          # Update service
          docker-compose up -d --no-deps $service

          # Wait for service to be ready
          sleep 10

          # Health check
          for i in {1..30}; do
            if curl -f http://localhost:$PORT/health; then
              echo "□ $service is healthy"
              return 0
            fi
            echo "Waiting for $service to be healthy... ($i/30)"
            sleep 2
```

```
  done

  echo "❌ $service health check failed"
  return 1
}
```

Blue-Green Deployment (Ready)

Concept: Two identical production environments **Benefits**: Instant rollback, zero downtime **Implementation**: Infrastructure ready, can be activated

```
# Blue-Green deployment preparation
prepare_blue_green() {
  # Create blue environment (current production)
  docker-compose -f docker-compose-blue.yml up -d

  # Deploy to green environment (new version)
  docker-compose -f docker-compose-green.yml up -d

  # Validate green environment
  validate_environment "green"

  # Switch traffic to green
  switch_traffic_to_green

  # Keep blue as rollback option
}
```

Canary Deployment (Future)

Concept: Gradual traffic shift to new version **Benefits**: Risk mitigation, performance validation **Implementation**: Can be added with load balancer configuration

Rollback Strategies

Automatic Rollback:

```
- name: Automatic rollback on failure
  if: failure()
  run: |
    echo "Deployment failed, initiating rollback..."
    LATEST_BACKUP=$(ls -t backups/ | head -n1)
    cp backups/$LATEST_BACKUP/docker-compose-complete.yml .
    docker-compose down && docker-compose up -d
```

Manual Rollback:

```
# Manual rollback command
gh workflow run deployment-approval.yml \
  -f action=rollback \
  -f environment=production \
  -f reason="Critical bug fix required"
```

Security Implementation

Pipeline Security

OIDC Authentication:

```yaml
permissions:
  id-token: write
  contents: read

- name: Configure AWS credentials
  uses: aws-actions/configure-aws-credentials@v4
  with:
    role-to-assume: ${{ secrets.AWS_ROLE_ARN }}
    aws-region: us-east-1
```

Benefits: - No long-lived access keys - Temporary credentials - Audit trail - Least privilege access

Secret Management

GitHub Secrets:

```
# Required secrets
AWS_ROLE_ARN=arn:aws:iam::339713159370:role/GitHubActionsRole-
FreedmAI
ECR_REGISTRY=339713159370.dkr.ecr.us-east-1.amazonaws.com
JWT_SECRET=uat-jwt-secret-key-2025
DATABASE_URL=postgresql://user:pass@localhost:5432/freedmai
```

Secret Scanning:

```yaml
- name: GitLeaks Secret Scan
  uses: gitleaks/gitleaks-action@v2
  env:
    GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

Vulnerability Scanning

Container Scanning:

```yaml
- name: Run Trivy vulnerability scanner
  uses: aquasecurity/trivy-action@master
  with:
    scan-type: 'fs'
    scan-ref: '.'
    format: 'sarif'
    output: 'trivy-results.sarif'
```

Dependency Scanning:

```
# npm audit for each service
cd $service && npm audit --audit-level=high
```

Access Control

Environment Protection:

```yaml
environment:
  name: production
  url: https://api.freedmai.com
protection_rules:
  - type: required_reviewers
    required_reviewers: 2
  - type: wait_timer
    wait_timer: 5
```

Monitoring and Observability

Health Monitoring

Service Health Checks:

```bash
# Automated health validation
services="api-gateway auth-service billing-service payment-service user-service notification-service"
for service in $services; do
  if curl -f http://localhost:$port/health; then
    echo "✅ $service health check passed"
  else
    echo "❌ $service health check failed"
    exit 1
  fi
done
```

Deployment Tracking

Deployment Records:

```json
{
  "deployment_id": "deploy-20250919-143022-a1b2c3d4",
  "action": "deploy",
  "environment": "production",
  "services": ["api-gateway", "auth-service"],
  "image_tag": "a1b2c3d4",
  "reason": "Feature release v1.2.0",
  "requested_by": "developer",
  "timestamp": "2025-09-19T14:30:22Z",
  "status": "success",
  "completed_at": "2025-09-19T14:35:45Z"
}
```

Performance Monitoring

Metrics Collection: - Deployment duration - Service startup times - Error rates - Response times - Throughput

Alerting (Future Enhancement): - Slack/Teams notifications - Email alerts - PagerDuty integration - Custom webhooks

Log Aggregation

Centralized Logging:

```yaml
- name: Collect deployment logs
  run: |
    # Collect logs from all services
    docker-compose logs > deployment-logs.txt

    # Upload to artifacts
    echo "Logs collected at $(date)"
```

Setup and Configuration

Prerequisites

Required Tools: - GitHub CLI (`gh`) - Docker and Docker Compose - AWS CLI - Node.js 18+ - Git

AWS Requirements: - ECR repositories - IAM role for OIDC - CloudWatch log groups - Appropriate permissions

Step-by-Step Setup

Step 1: GitHub Repository Setup

Run the automated setup script:

```
./setup-github-repo.sh
```

What it does: 1. Creates GitHub repository in `freedmai` organization 2. Initializes git repository with all files 3. Sets up GitHub Actions workflows 4. Configures environments (UAT, Production) 5. Sets up GitHub secrets 6. Creates initial commit and pushes to GitHub

Manual verification:

```
# Check repository creation
gh repo view freedmai/freedmai-microservices

# Check workflows
gh workflow list

# Check environments
gh api repos/freedmai/freedmai-microservices/environments
```

Step 2: Self-Hosted Runner Configuration

Run the runner setup script:

```
./setup-github-runner.sh
```

What it does: 1. Downloads GitHub Actions runner 2. Configures runner with repository 3. Creates systemd service 4. Sets up monitoring and health checks 5. Creates management scripts

Verification:

```
# Check runner status
github-runner-manage status

# View runner logs
github-runner-manage logs

# Run health check
github-runner-health
```

Step 3: AWS Infrastructure

Terraform deployment:

```
cd terraform/
terraform init
terraform apply -var="environment=uat"
```

Verify resources:

```
# Check ECR repositories
aws ecr describe-repositories

# Check CloudWatch log groups
aws logs describe-log-groups --log-group-name-prefix "/freedmai"
```

Step 4: First Deployment

Trigger UAT deployment:

```
gh workflow run ci-cd-pipeline.yml \
  -f environment=uat \
  -f services=all
```

Monitor deployment:

```
# Check workflow status
gh run list

# View specific run
gh run view [run-id]

# Check deployment logs
gh run view [run-id] --log
```

Best Practices

Workflow Design

1. Keep Workflows Simple - Single responsibility per workflow - Clear job dependencies - Meaningful job and step names - Proper error handling

2. Use Matrix Strategies

```
strategy:
```

```
          matrix:
            service: [api-gateway, auth-service, billing-service]
            node-version: [18, 20]
```

3. Implement Proper Caching

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

Security Best Practices

1. Use OIDC Instead of Long-lived Keys

```
        permissions:
          id-token: write
          contents: read
```

2. Scan for Vulnerabilities - Container images - Dependencies - Source code - Secrets

3. Implement Least Privilege - Minimal permissions for each job - Environment-specific access - Time-limited credentials

Testing Best Practices

1. Test Pyramid - Many unit tests - Some integration tests - Few end-to-end tests

2. Fail Fast - Run fastest tests first - Stop on first failure - Provide clear error messages

3. Test in Production-like Environment - Same container images - Similar data - Realistic load

Deployment Best Practices

1. Zero-Downtime Deployments - Rolling updates - Health checks - Graceful shutdowns

2. Rollback Strategy - Automated rollback on failure - Manual rollback capability - Version tracking

3. Environment Parity - Same deployment process - Same container images - Configuration differences only

Troubleshooting Guide

Common Issues

Issue 1: Workflow Not Triggering

Symptoms: - Push to main branch doesn't trigger workflow - Manual dispatch not available

Solutions:

```
# Check workflow syntax
gh workflow list

# Validate YAML syntax
yamllint .github/workflows/ci-cd-pipeline.yml

# Check repository permissions
gh api repos/freedmai/freedmai-microservices/actions/permissions
```

Issue 2: Self-Hosted Runner Not Connecting

Symptoms: - Runner shows as offline - Jobs queued but not executing

Solutions:

```
# Check runner service
github-runner-manage status

# Restart runner service
github-runner-manage restart

# Check runner logs
github-runner-manage logs

# Verify network connectivity
curl -I https://api.github.com
```

Issue 3: Docker Build Failures

Symptoms: - Docker build step fails - Image push to ECR fails

Solutions:

```
# Check Docker daemon
sudo systemctl status docker

# Verify ECR authentication
aws ecr get-login-password --region us-east-1 | docker login --
username AWS --password-stdin 339713159370.dkr.ecr.us-east-
1.amazonaws.com

# Check Dockerfile syntax
docker build -t test-image .

# Verify ECR repository exists
aws ecr describe-repositories --repository-names freedmai-api-
gateway
```

Issue 4: Deployment Health Checks Failing

Symptoms: - Services deployed but health checks fail - Rollback triggered automatically

Solutions:

```
# Check service logs
docker-compose logs [service-name]

# Test health endpoint directly
curl http://localhost:3000/health

# Check service dependencies
docker-compose ps

# Verify environment variables
docker exec [container-name] env
```

Issue 5: AWS Authentication Failures

Symptoms: - OIDC authentication fails - ECR push permission denied

Solutions:

```
# Verify IAM role exists
aws iam get-role --role-name GitHubActionsRole-FreedmAI

# Check trust relationship
aws iam get-role --role-name GitHubActionsRole-FreedmAI --query
'Role.AssumeRolePolicyDocument'

# Verify GitHub secrets
gh secret list

# Test AWS CLI access
aws sts get-caller-identity
```

Debugging Commands

Workflow Debugging:

```
# List all workflows
gh workflow list

# View workflow runs
gh run list --workflow=ci-cd-pipeline.yml

# View specific run details
gh run view [run-id]

# Download run logs
gh run download [run-id]
```

Runner Debugging:

```
# Check runner health
github-runner-health

# View runner logs
```

```
sudo journalctl -u github-runner -f

# Check runner configuration
cat /opt/github-runner/.runner

# Test runner connectivity
sudo -u github-runner /opt/github-runner/run.sh --check
```

Docker Debugging:

```
# Check container status
docker-compose ps

# View container logs
docker-compose logs -f [service-name]

# Execute commands in container
docker exec -it [container-name] /bin/sh

# Check container resources
docker stats
```

Performance Optimization

Workflow Performance: - Use caching for dependencies - Parallel job execution - Optimize Docker builds - Use appropriate runner sizes

Deployment Performance: - Optimize container images - Use health check timeouts - Implement proper resource limits - Monitor deployment metrics

---

Document Version: 1.0
Implementation Date: September 19, 2025
Status: Complete CI/CD Implementation
Learning Objective: Comprehensive CI/CD mastery with GitHub Actions