Deploy Early & Often

Continuous Integration & Deployment

Enabling rapid feedback and reliable delivery

Why Deploy Early & Often?

> **"If it hurts, do it more often, and bring the pain forward"**

- **Faster Feedback** - Identify issues quickly
- **Reduced Risk** - Smaller, manageable changes
- **Faster Value Delivery** - Users see benefits sooner
- **Better Quality** - Continuous integration catches problems

The Problem

Traditional Deployment Issues:

- 🚢 Big bang releases with high risk
- 😰 Fear of deploying due to complexity
- 🐛 Integration issues discovered late
- ⏰ Long time between development and user feedback
- 🔧 Manual, error-prone deployment processes

Result: Slow, risky, unreliable software delivery

The Solution

CI/CD Pipeline Fundamentals:

1. **Continuous Integration (CI)**
2. **Continuous Deployment (CD)**
3. **Automated Testing**
4. **Infrastructure as Code**
5. **Monitoring & Rollback**

Continuous Integration (CI)

Automated Build & Test

Core Practices:

- Commit code frequently (daily minimum)
- Automated builds on every commit
- Fast feedback on build status
- Shared integration environment

Benefits:

- Early detection of integration issues
- Reduced merge conflicts
- Consistent build process

CI Pipeline Example

```yaml
# GitHub Actions CI Pipeline
name: CI Pipeline
on: [push, pull_request]

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 install
      - run: npm run lint
      - run: npm run test
      - run: npm run build
```

Continuous Deployment (CD)

Automated Release Pipeline

Deployment Stages:

1. **Build** - Compile and package code
2. **Test** - Run automated test suites
3. **Stage** - Deploy to staging environment
4. **Production** - Deploy to live environment

Key Principle: Every commit is a potential release

CD Pipeline Example

```yaml
# Production Deployment
deploy:
  needs: test
  runs-on: ubuntu-latest
  if: github.ref == 'refs/heads/main'
  steps:
    - uses: actions/checkout@v2
    - name: Deploy to Production
      run: |
        docker build -t myapp .
        docker push registry.com/myapp:latest
        kubectl set image deployment/myapp app=registry.com/myapp:latest
```

Branching Strategies

Git Workflows for CI/CD

GitFlow:

- `main` - Production releases
- `develop` - Integration branch
- `feature/*` - New features
- `release/*` - Release preparation
- `hotfix/*` - Critical fixes

GitHub Flow (Simpler):

- `main` - Always deployable
- `feature/*` - Pull request workflow

Feature Branches & Pull Requests

Workflow:

1. Create feature branch from `main`
2. Develop and commit changes
3. Open pull request
4. Automated tests run
5. Code review process
6. Merge to `main`
7. Automatic deployment

Benefits:

- Code review before integration
- Isolated feature development

Deployment Strategies

Minimizing Risk

Blue-Green Deployment:

- Two identical production environments
- Switch traffic between them
- Instant rollback capability

Canary Deployment:

- Deploy to small subset of users
- Monitor performance and errors
- Gradually increase traffic

Blue-Green Deployment

```
Production Traffic (100%)
          ↓
    Load Balancer
          ↓
    [Blue Environment]  ←  Current Production
    [Green Environment] ←  New Version (Testing)

# After validation, switch traffic:
Production Traffic (100%)
          ↓
    Load Balancer
          ↓
    [Blue Environment]  ←  Previous Version (Standby)
    [Green Environment] ←  New Production
```

Feature Flags

Controlled Feature Rollout

Benefits:

- Deploy code without activating features
- A/B testing capabilities
- Quick feature disabling
- Gradual user rollout

Example:

```
// Feature flag implementation
if (featureFlags.isEnabled('newCheckoutFlow', user)) {
  return <NewCheckoutComponent />;
} else {
```

Infrastructure as Code

Automated Environment Management

Tools:

- **Terraform** - Cloud infrastructure
- **Ansible** - Configuration management
- **Docker** - Containerization
- **Kubernetes** - Orchestration

Benefits:

- Consistent environments
- Version-controlled infrastructure
- Repeatable deployments

Docker Example

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

```yaml
# docker-compose.yml
version: '3.8'
services:
  app:
    build: .
    ports:
      - "3000:3000"
    environment:
      - NODE_ENV=production
```

Monitoring & Observability

Production Health Awareness

Essential Metrics:

- **Application Performance** - Response times, throughput
- **Error Rates** - Failed requests, exceptions
- **Infrastructure** - CPU, memory, disk usage
- **Business Metrics** - User engagement, conversions

Tools: **DataDog, New Relic, Prometheus, Grafana**

Deployment Monitoring

Health Checks:

```
// Health check endpoint
app.get('/health', (req, res) => {
  const health = {
    status: 'healthy',
    timestamp: new Date().toISOString(),
    version: process.env.APP_VERSION,
    database: await checkDatabaseConnection(),
    memory: process.memoryUsage()
  };
  res.json(health);
});
```

Automated Alerts:

- Deployment success/failure
- Performance degradation

Rollback Strategies

Quick Recovery from Issues

Automated Rollback Triggers:

- Health check failures
- Error rate thresholds
- Performance degradation
- Manual intervention

Rollback Methods:

- Previous Docker image
- Blue-green environment switch
- Database migration rollback

Rollback Example

```
# Kubernetes rollback
kubectl rollout undo deployment/myapp

# Docker rollback
docker service update --image myapp:previous-version myapp

# Feature flag rollback
featureFlags.disable('problematicFeature');
```

Database Migrations

Schema Changes in CI/CD

Best Practices:

- **Backward Compatible** - Support old and new code
- **Incremental Changes** - Small, safe modifications
- **Rollback Scripts** - Prepared reversal procedures
- **Data Migration Testing** - Validate transformations

Migration Strategy:

1. Deploy code that supports old and new schema
2. Run migration scripts
3. Deploy code that uses new schema only

Security in CI/CD

Secure Deployment Pipeline

Security Practices:

- **Secret Management** - Environment variables, key vaults
- **Image Scanning** - Vulnerability detection
- **Access Controls** - Limited deployment permissions
- **Audit Logging** - Track all deployments

Tools: **HashiCorp Vault, AWS Secrets Manager, Snyk, SonarQube**

Secrets Management

```yaml
# GitHub Actions with secrets
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Deploy to AWS
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        run: |
          aws s3 sync ./build s3://my-bucket
```

Testing in Production

Verify Deployment Success

Production Testing:

- **Smoke Tests** - Basic functionality verification
- **Synthetic Monitoring** - Simulated user interactions
- **Real User Monitoring** - Actual user experience
- **Chaos Engineering** - Failure resilience testing

Tools: **Pingdom, Datadog Synthetics, Chaos Monkey**

Performance Considerations

Optimizing Deployment Speed

Build Optimization:

- **Caching** - Dependencies and build artifacts
- **Parallel Jobs** - Run tests and builds concurrently
- **Incremental Builds** - Only rebuild changed components
- **Artifact Reuse** - Share builds across environments

Deployment Speed:

- **Blue-green** - Instant traffic switching
- **Rolling updates** - Gradual instance replacement
- **CDN Updates** - Fast static asset deployment

Common Pitfalls

❌ Deployment Anti-Patterns:

- **Manual Steps** - Human error prone processes
- **Environment Drift** - Inconsistent configurations
- **No Rollback Plan** - Unable to recover quickly
- **Skipping Staging** - Deploying untested code
- **Large Batch Deployments** - High risk changes
- **No Monitoring** - Blind to deployment success

Implementation Roadmap

Getting Started with CI/CD:

Week 1-2: Basic CI Setup

- Automated builds on commits
- Unit test execution
- Build artifact creation

Week 3-4: Deployment Automation

- Staging environment deployment
- Environment configuration
- Basic monitoring

Week 5-6: Production Pipeline

- Production deployment automation
- Rollback procedures
- Advanced monitoring

Metrics & KPIs

Measuring CI/CD Success:

Deployment Metrics:

- **Deployment Frequency** - How often do you deploy?
- **Lead Time** - Time from commit to production
- **MTTR** - Mean time to recovery
- **Change Failure Rate** - Percentage of failed deployments

Target Goals:

- Deploy multiple times per day
- Lead time under 1 hour
- MTTR under 1 hour
- Change failure rate under 15%

Tools Comparison

Popular CI/CD Platforms:

Tool	Strengths	Best For
GitHub Actions	Git integration, free tier	Open source projects
GitLab CI	All-in-one platform	Enterprise teams
Jenkins	Highly customizable	Complex workflows
CircleCI	Fast builds, good caching	Medium-sized teams
Azure DevOps	Microsoft ecosystem	.NET applications

Key Takeaways

🎯 Remember:

1. **Start small** - Begin with basic CI, add CD gradually
2. **Automate everything** - Reduce manual intervention
3. **Monitor actively** - Know when deployments succeed/fail
4. **Plan for rollback** - Always have an escape route
5. **Deploy frequently** - Smaller changes are safer