GitHub Workflow Guide

Document ID: IT-SW-003 **Last Updated:** March 2024 **Owner:** Engineering
Applies To: All Engineering Teams

Overview

This guide outlines NovaTech's GitHub workflow, including branching strategies, code review process, and CI/CD integration with DevPipeline.

GitHub Organization

Organization Structure

- **Organization:** github.com/novatech-solutions
- **Access:** Via Okta SSO
- **Teams:** Aligned with engineering teams

Repositories

Repository Type	Naming Convention	Example
Product	product-name	cloudforge, devpipeline
Libraries	lib-name	lib-auth, lib-common
Infrastructure	infra-name	infra-terraform, infra-k8s
Documentation	docs-name	docs-api, docs-internal

Branching Strategy

Branch Types

Branch	Purpose	Naming	Lifetime
main	Production code	`main`	Permanent

Branch	Purpose	Naming	Lifetime
develop	Integration branch	`develop`	Permanent
feature	New features	`feature/TICKET-description`	Temporary
bugfix	Bug fixes	`bugfix/TICKET-description`	Temporary
hotfix	Production fixes	`hotfix/TICKET-description`	Temporary
release	Release preparation	`release/vX.Y.Z`	Temporary

Branch Flow

```
main

                                                    (merge)

develop


     feature/A                          release/v1.0

     feature/B

     bugfix/C
```

Branch Protection

main branch: - Require pull request reviews (2 approvals) - Require status checks to pass - Require signed commits - No force pushes - No deletions

develop branch: - Require pull request reviews (1 approval) - Require status checks to pass - No force pushes

---

Workflow Process

Feature Development

1. **Create branch from develop:**

```
git checkout develop
git pull origin develop
git checkout -b feature/ENG-123-add-user-auth
```

2. **Make changes and commit:**

```
git add .
git commit -m "feat(auth): add user authentication

- Implement JWT token generation
- Add login endpoint
- Add logout endpoint

Fixes ENG-123"
```

3. **Push and create PR:**

```
git push -u origin feature/ENG-123-add-user-auth
```

4. **Create pull request** targeting develop
5. **Address review feedback**
6. **Merge after approval**

Bug Fixes

1. **Branch from develop** (or main for hotfix)
2. **Fix the issue**
3. **Create PR with bug details**
4. **Merge after review**

Hotfixes

1. **Branch from main:**

```
git checkout main
git checkout -b hotfix/ENG-456-fix-login-crash
```

2. **Fix critical issue**
3. **Create PR to main**
4. **After merge, backport to develop:**

```
git checkout develop
git merge main
```

Commit Conventions

Commit Message Format

```
<type>(<scope>): <subject>

<body>

<footer>
```

Types

Type	Description
feat	New feature
fix	Bug fix
docs	Documentation
style	Formatting (no code change)
refactor	Code restructuring
perf	Performance improvement
test	Adding tests
chore	Maintenance
ci	CI/CD changes

Examples

```
feat(api): add user profile endpoint

- GET /api/v1/users/:id/profile
- Returns user profile data
- Includes avatar URL

Fixes ENG-123

---

fix(auth): resolve token expiration issue

Tokens were not being refreshed correctly when the refresh
token was close to expiration.

Fixes ENG-456

---
```

```
docs(readme): update installation instructions
```

Rules

- Subject line max 72 characters
- Use imperative mood ("add" not "added")
- No period at end of subject
- Body explains what and why
- Reference issue numbers

Pull Request Process

Creating a PR

1. Use the PR template
2. Link related issues
3. Add appropriate labels
4. Assign reviewers
5. Add to project board

PR Template

```
## Description
Brief description of changes

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

## Related Issues
Fixes #123

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

## Checklist
```

```
- [ ] Code follows style guidelines
- [ ] Self-review completed
- [ ] Documentation updated
- [ ] No new warnings
```

Labels

Label	Description
`bug`	Bug fix
`feature`	New feature
`breaking`	Breaking change
`needs-review`	Awaiting review
`approved`	Approved, ready to merge
`blocked`	Cannot proceed
`wip`	Work in progress

Code Review

Reviewer Guidelines

What to review: - Code correctness - Test coverage - Security considerations - Performance implications - Documentation - Coding standards

Review comments: - Be constructive and respectful - Explain the "why" - Suggest alternatives - Distinguish required vs. optional changes

Comment Prefixes

Prefix	Meaning
`blocking:`	Must fix before merge
`suggestion:`	Consider this approach
`question:`	Need clarification
`nit:`	Minor, optional
`praise:`	Great work!

Review Expectations

- Respond to PR within 24 hours

- Complete review within 48 hours
- Use "Request changes" sparingly
- Approve when ready (don't delay)

CI/CD Integration

DevPipeline Integration

Every PR triggers:

```yaml
# .devpipeline/pr.yaml
on:
  pull_request:
    branches: [develop, main]

jobs:
  test:
    steps:
      - checkout
      - run: npm install
      - run: npm test
      - run: npm run lint

  security:
    steps:
      - checkout
      - run: devpipeline security scan

  build:
    needs: [test, security]
    steps:
      - checkout
      - run: npm run build
```

Required Checks

Check	Required For
Unit tests	All PRs
Lint	All PRs
Security scan	All PRs

Check	Required For
Build	All PRs
Integration tests	PRs to main
Code coverage	PRs to main

Merge Requirements

- All checks passing
- Required approvals received
- No merge conflicts
- Branch up to date with target

Release Process

Creating a Release

1. **Create release branch:**

```
git checkout develop
git checkout -b release/v1.2.0
```

2. **Update version numbers**
3. **Final testing**
4. **Create PR to main**
5. **After merge, tag release:**

```
git checkout main
git tag -a v1.2.0 -m "Release v1.2.0"
git push origin v1.2.0
```

6. **Create GitHub release** with changelog

Semantic Versioning

Format: `MAJOR.MINOR.PATCH`

Change	Increment
Breaking changes	MAJOR
New features	MINOR
Bug fixes	PATCH

GitHub Actions

Workflow Examples

Auto-labeling:

```yaml
# .github/workflows/labeler.yml
name: Labeler
on: [pull_request]
jobs:
  label:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/labeler@v4
        with:
          repo-token: ${{ secrets.GITHUB_TOKEN }}
```

Stale PR cleanup:

```yaml
# .github/workflows/stale.yml
name: Stale
on:
  schedule:
    - cron: '0 0 * * *'
jobs:
  stale:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/stale@v8
        with:
          stale-pr-message: 'This PR is stale.'
          days-before-stale: 14
          days-before-close: 7
```

Best Practices

Do

- Keep PRs small and focused
- Write descriptive commit messages
- Update documentation with code changes
- Respond to reviews promptly
- Keep branches up to date
- Delete merged branches

Don't

- Commit directly to main or develop
- Force push to shared branches
- Leave PRs open for weeks
- Merge without required approvals
- Ignore failing checks

Troubleshooting

Merge Conflicts

```
# Update your branch
git checkout feature/my-branch
git fetch origin
git rebase origin/develop

# Resolve conflicts
# Edit conflicting files
git add .
git rebase --continue

# Push updated branch
git push --force-with-lease
```

Failed Checks

1. Read the error message
2. Check the CI logs
3. Fix locally and push
4. Re-run checks if flaky

Accidental Commit to Main

Contact Engineering Manager immediately. Do not attempt to fix without guidance.

Resources

- **GitHub Docs:** docs.github.com
- **Internal Wiki:** wiki.novatech.com/github
- **DevPipeline Docs:** docs.novatech.com/devpipeline
- **Help:** #engineering-help on Slack

Related Documents: DevPipeline Getting Started (PRD-DP-001), Code Review Guidelines (ENG-PROC-005)