SEP Project
Git Guidelines

CodeCatalyst UG33
Team Documentation

Version 1.1

September 6, 2025

Contents

1 Quick Reference

This section provides essential commands for daily development workflow.

1.1 Common Git Commands

Essential Git Commands

```
# Create and switch to new branch
git checkout -b UG33-<number>-<description>

# Stage and commit changes
git add .
git commit -m "feat: add new feature description"

# Push branch to remote
git push origin UG33-<number>-<description>

# Update branch with latest main / Rebasing main to keep commits ahead of
    main
git checkout main && git pull origin main
git checkout <your-branch> && git rebase main

# Clean up after merge
git checkout main && git pull origin main
git branch -d <merged-branch-name>
```

1.2 Essential Build Commands

Key Commands for Development Workflow

```
# Essential workflow commands
make test-all               # Build and test everything (use before PR)
make windows-package        # Create submission package

# See docs/Project Structure.pdf for complete Make reference
make help                   # Show all available targets
```

1.3 Quick PR Checklist

Before Creating a Pull Request

Essential checks before submitting a PR:

- Branch name: `UG33-<number>-<description>`
- Commits follow conventional format
- Jira ticket linked in PR description
- One reviewer assigned
- All acceptance criteria met
- `make test-all` passes completely
- Code follows project structure (`src/`, `include/`, `tests/`)
- No `using namespace std` in code
- Build artifacts only in `build/` directory

2 Repository Setup

This section outlines the initial repository configuration and access control for the SEP project.

2.1 Repository Configuration

Setting	Value
Repository name	`SEP-UG33-ProjectName`
Primary branch	`main` (protected)
Integration branch	`develop` (optional)
Access control	Team members only
Direct pushes to main	**Disabled**

Table 1: Repository Configuration Settings

2.2 Branch Protection Rules

The `main` branch must be configured with the following protection rules:

- **Require pull request reviews** – At least 1 approving review
- **Dismiss stale reviews** – When new commits are pushed
- **Require status checks** – All CI checks must pass
- **Restrict pushes** – Only allow pushes via pull requests
- **Include administrators** – Enforce restrictions for all users

3 Branching Strategy

This section defines the branching conventions and rules for the SEP project workflow.

3.1 Branch Naming Convention

All branches must follow this naming pattern:

```
UG33-<ticket-number>-<short-description>
```

3.2 Branch Types and Examples

Branch Type	Purpose	Example
Feature/Story	New functionality	`UG33-19-initiate-group-kick-off`
Bugfix	Small fixes	`UG33-25-fix-validation-error`
Hotfix	Urgent production fixes	`UG33-30-hotfix-memory-leak`

Table 2: Branch Types and Naming Examples

3.3 Branching Rules

- **Always branch from `main`** – Ensures clean history
- **One branch per ticket** – Keep changes focused and reviewable
- **Short-lived branches** – Merge frequently to avoid conflicts
- **Descriptive names** – Branch name should indicate the work being done

3.4 Branch Creation Examples

Good Example

Creating a feature branch:

```
git checkout main
git pull origin main
git checkout -b UG33-22-add-compression-optimization
```

Why this works:

- Starts from latest `main`
- Follows naming convention
- Clear, descriptive branch name

Poor Example

Poor branch naming:

```
git checkout -b fix-stuff
git checkout -b johns-work
git checkout -b temp-branch
```

Problems:

- No ticket reference
- Vague descriptions
- No team naming convention

4 Commit Conventions

This section defines the commit message format and standards for the SEP project.

4.1 Conventional Commit Format

All commits must follow this structure:

```
<type>: <short description>

[optional body]

[optional footer]
```

4.2 Commit Types

Type	Usage
feat	New feature or functionality
fix	Bug fix or correction
docs	Documentation changes only
chore	Maintenance tasks (build, dependencies, formatting)
refactor	Code changes without new features or fixes
test	Adding or modifying tests
style	Code formatting changes (no logic changes)

Table 3: Commit Types for SEP Project

4.3 Commit Message Examples

Good Example

Good commit messages:

```
feat: add Google login support
fix: correct block validation logic in compression
docs: update installation guide with new dependencies
chore: update React to v18
refactor: extract compression utilities to separate module
```

Why these work:

- Clear type prefix
- Concise but descriptive
- Present tense, imperative mood
- No period at end

Poor Example

Poor commit messages:

```
Fixed stuff
Updated files
Work in progress
asdf
Changed some things in the compression algorithm maybe
```

Problems:

- No type prefix
- Vague or meaningless descriptions
- Past tense or unclear language
- No indication of what was changed

5 Pull Request (PR) Process

This section outlines the complete pull request workflow from creation to merge.

5.1 PR Creation Steps

1. **Push branch to repository**

   ```
   git push origin UG33-22-add-compression-optimization
   ```

2. **Create PR targeting `main` branch**
3. **Fill out PR template completely**
4. **Request review from one team member**
5. **Ensure all CI checks pass**

5.2 PR Checklist

Before submitting a PR, verify all items in this checklist:

Pull Request Checklist

Required Information:

- Jira ticket linked in PR description
- Branch name follows `UG33-<number>-<description>` convention
- PR title is descriptive and clear
- Description explains what was changed and why

Code Quality:

- Only relevant source files included (`.cpp`, `.h`)
- No unnecessary files (binaries, `.DS_Store`, IDE files)
- Code builds successfully locally
- All commits follow conventional commit format

Testing & Validation:

- Changes tested locally
- No regression in existing functionality
- Acceptance criteria from Jira ticket met

Assignment:

- One assignee (PR owner)
- One reviewer assigned (not PR owner)

6 Code Review Guidelines

This section establishes the standards and process for conducting thorough code reviews.

6.1 Reviewer Responsibilities

- **Review within 24-48 hours** of assignment
- **Provide constructive feedback** with specific suggestions
- **Test changes locally** when possible
- **Approve only when satisfied** with code quality and correctness

6.2 Review Criteria

Review Area	What to Check
Correctness	Logic accuracy, algorithm implementation, edge cases
Code Quality	Naming conventions, formatting, readability
Conventions	Branch naming, commit messages, PR format
Files	Only relevant source files, no binaries or temp files
Testing	Adequate testing, no regression, acceptance criteria met

Table 4: Code Review Focus Areas

6.3 Feedback Guidelines

Constructive Review Comments

Good feedback examples:
- "Consider using a const reference here to avoid unnecessary copying"
- "This logic could be simplified using the STL algorithm library"
- "Add error handling for the case when input file is empty"
- "Variable name 'temp' is unclear – consider 'compressedBlock'"

Why this works:
- Specific and actionable
- Educational and helpful
- Focuses on code improvement

Poor Review Comments

Poor feedback examples:
- "This is wrong"
- "Bad code"
- "I don't like this"
- "Change everything"

Problems:
- Not specific or actionable
- Doesn't explain the issue
- Not constructive or helpful

7 Merging Guidelines

This section defines when and how to merge pull requests into the main branch.

7.1 Merge Requirements

A PR can only be merged when **ALL** of the following conditions are met:

- **At least one approval** from a team member reviewer
- **All CI checks passing** (build, tests, linting)
- **No merge conflicts** with main branch
- **All review comments addressed** or resolved
- **Branch is up to date** with latest main

7.2 Post-Merge Cleanup

After successfully merging a PR:

Post-Merge Commands

```
# Delete the merged branch locally
git checkout main
git pull origin main
git branch -d UG33-22-add-compression-optimization

# Delete the remote branch (done automatically by GitHub)
```

8 Rebasing and Branch Management

This section covers advanced Git workflows for maintaining clean commit history and handling branch updates.

8.1 Understanding Rebasing

Rebasing is the process of moving or combining commits from one branch onto another. Unlike merging, rebasing creates a linear history by replaying commits on top of the target branch.

Operation	Purpose	Result
Rebase	Update branch with latest main	Linear, clean history
Merge	Combine branches	Preserves branch history
Squash	Combine multiple commits	Single commit

Table 5: Git Operations Comparison

8.2 When to Use Rebasing

- **Before creating a PR** – Ensure your branch is up-to-date
- **During development** – Keep your feature branch current
- **Cleaning up commits** – Combine related commits before review
- **Resolving conflicts** – Preferred method for conflict resolution

8.3 Interactive Rebasing

Interactive rebasing allows you to modify commits during the rebase process:

Interactive Rebase Commands

```
# Start interactive rebase for last 3 commits
git rebase -i HEAD~3

# Rebase against main branch interactively
git rebase -i origin/main
```

8.3.1 Interactive Rebase Actions

8.4 Standard Rebasing Workflow

Follow this process for routine branch updates:

Action	Description
pick	Use commit as-is (default)
reword	Change commit message
squash	Combine with previous commit, keep both messages
fixup	Combine with previous commit, discard this message
drop	Remove commit entirely

Table 6: Interactive Rebase Actions

Standard Rebase Process

```
# 1. Ensure you're on your feature branch
git checkout UG33-22-add-compression-optimization

# 2. Fetch latest changes from remote
git fetch origin

# 3. Rebase your branch onto latest main
git rebase origin/main

# 4. If conflicts occur, resolve them and continue
# (See conflict resolution section below)

# 5. Force push your updated branch
git push --force-with-lease origin UG33-22-add-compression-optimization
```

8.5 Rebasing Best Practices

Good Rebasing Practices

Safe rebasing guidelines:

- Always use `--force-with-lease` instead of `--force`
- Rebase frequently to avoid large conflicts
- Test your code after rebasing
- Communicate with team when rebasing shared branches
- Use interactive rebase to clean up commit history

Why this works:

- `--force-with-lease` prevents overwriting others' work
- Frequent rebasing keeps conflicts small and manageable
- Clean commit history improves code review quality

Rebasing Pitfalls to Avoid

Dangerous practices:
- Rebasing shared/published branches
- Using `git push --force` without `--force-with-lease`
- Rebasing without understanding the changes
- Ignoring conflicts during rebase

Problems:
- Can rewrite history others depend on
- Risk of losing work or overwriting changes
- Creates confusion and broken workflows

9 Conflict Resolution

This section covers handling merge conflicts during rebasing and maintaining branch synchronization.

9.1 Preventing Conflicts

- **Keep branches short-lived** – Merge frequently
- **Rebase regularly** – Stay current with main branch
- **Coordinate with team** – Communicate when working on same files
- **Small, focused changes** – Easier to review and merge

9.2 Resolving Conflicts During Rebase

When conflicts occur during rebasing, follow this detailed process:

Conflict Resolution Process

```
# 1. Start rebase (conflicts may occur here)
git checkout UG33-22-add-compression-optimization
git fetch origin
git rebase origin/main

# 2. If conflicts occur, Git will pause and show conflicted files
git status  # Shows files with conflicts

# 3. Open conflicted files and resolve conflicts manually
# Look for conflict markers: <<<<<<<, =======, >>>>>>>

# 4. After resolving conflicts, stage the resolved files
git add <resolved-file1> <resolved-file2>

# 5. Continue the rebase
git rebase --continue

# 6. Repeat steps 3-5 for each conflicted commit

# 7. Once rebase completes, force push the updated branch
git push --force-with-lease origin UG33-22-add-compression-optimization
```

9.3 Conflict Resolution Example

Resolving a Typical Conflict

Conflict in source file:

```
<<<<<<< HEAD (your changes)
std::cout << "New compression algorithm" << std::endl;
=======
std::cout << "Updated compression logic" << std::endl;
>>>>>>> main (incoming changes)
```

Resolution steps:

1. Understand both changes
2. Decide which version to keep or combine them
3. Remove conflict markers
4. Test the resolved code
5. Stage and continue rebase

Resolved version:

```
std::cout << "Enhanced compression algorithm" << std::endl;
```

9.4 Rebase Abort and Recovery

If rebase becomes too complex, you can abort and try a different approach:

Rebase Recovery Commands

```
# Abort current rebase and return to original state
git rebase --abort

# Alternative: Merge instead of rebase
git merge origin/main

# Check rebase status
git status

# View rebase progress
git rebase --show-current-patch
```

10 Related Documentation

This guide focuses on Git workflow and team collaboration. For other aspects of the project:

- **Build & Code Quality** – See `docs/Project Structure.pdf` for Make targets, testing, clang-format
- **Project Planning** – See `docs/Jira Workflow.pdf` for ticket management and sprint planning
- **Quick Daily Reference** – See `README.md` for essential commands
- **Live Build Help** – Run `make help` for current build targets

11 Documentation Organization

The SEP project documentation is organized across multiple specialized documents:

11.1 Document Structure

Document	Purpose
docs/Git & Github Workflow.pdf	Git workflow, branching, rebasing, PR process
docs/Project Structure.pdf	Build system, testing, code quality, Make targets
docs/Jira Workflow.pdf	Jira ticket management and project planning
README.md	Quick daily reference for developers

Table 7: Complete Documentation Organization

11.2 Usage Guidelines

- **Git workflow** – This document (`docs/Git & Github Workflow.pdf`) for branching, rebasing, PRs
- **Technical setup** – `docs/Project Structure.pdf` for build system and code quality
- **Project management** – `docs/Jira Workflow.pdf` for ticket management and planning
- **Daily reference** – `README.md` for quick commands and immediate help
- **Live help** – `make help` for current build targets

11.3 Maintenance

- Update documents as processes evolve or new tools are adopted
- Ensure all team members can access and reference these guidelines
- Review and update during retrospectives or process improvements
- Keep README.md synchronized with LaTeX documentation
- Maintain consistency between workflow and structure documents