

# AI Agent Rules for Git Operations

This document provides specific rules for AI agents (like Cursor/Console AI) when performing git operations on this repository.

---

## Overview

**Primary Principle:** AI assists with git operations, but the user maintains final control over critical operations.

---

## What AI Can Do Directly

### [OK] Commit Message Generation

- Generate Angular-formatted commit messages following strict format
- Suggest appropriate commit types (feat, fix, docs, etc.)
- Format commit messages with proper structure (subject, blank line, body)
- Use multiple `-m` flags or single `-m` with escaped newlines for proper formatting

### [OK] Feature Branch Commits

- Create commits on feature branches
- Use informal messages during development ("wip", "refactor X")
- Commit code changes directly to feature branches

### [OK] Generate Commands

- Suggest git commands for workflows
- Generate merge commands with `--no-ff` flag
- Create command sequences for complex operations

### [OK] Automatic Operations

- Pre-commit hooks automatically update `version.py` git hash
  - No manual intervention needed for hash updates
- 

## What Requires User Review/Approval

### **WARNING: Merges to Main Branch**

#### **Process:**

1. AI generates merge command: `git merge --no-ff feat/branch-name`

2. AI generates merge commit message in Angular format
3. **User reviews** merge message in editor
4. **User saves/closes** editor to complete merge
5. **User pushes** when ready

**Why:** Main branch is sacred - requires human oversight

#### **WARNING: Creating Tags/Rleases**

**Process:**

1. AI suggests version number based on commit types
2. AI suggests update to `version.py`
3. **User confirms** version number
4. AI generates commands:  

```
# Update version.py __version__ to match tag
# Commit version change
git tag v1.0.0-beta.5
git push origin v1.0.0-beta.5
```
5. **User reviews** and executes commands
6. **User confirms** tag creation

**Why:** Releases are critical - require explicit approval

#### **WARNING: Pushing to Remote**

**Process:**

1. AI suggests push commands
2. **User executes** push commands manually
3. AI should NOT auto-push without explicit user request

**Why:** Security control - prevents accidental remote changes

#### **WARNING: Hotfix Operations**

**Process:**

1. AI generates hotfix workflow commands
2. **User reviews** each step
3. **User executes** step by step
4. User confirms each critical operation (tag, push)

**Why:** Hotfixes are high-risk - require careful manual review

---

## Character Encoding Rules

### 7-bit ASCII Requirement

AI agents **MUST** use 7-bit ASCII (characters 0-127 only) in all non-markdown files.

**Critical Rule:** When creating or modifying any code files, scripts, or configuration files, AI agents must:

- [OK] Use only 7-bit ASCII characters
- [OK] Replace any non-ASCII characters with ASCII equivalents
- Never use emoji, Unicode symbols, or multi-byte characters in code/scripts

### Files Affected:

- All source code files (.py, .js, .ts, .html, .css, etc.)
- All script files (.ps1, .bat, .sh, etc.)
- All configuration files (.json, .yaml, .ini, etc.)
- Documentation source files (.rst, .txt, etc.)

**Exception:** Markdown files (.md) may contain UTF-8 characters, BUT with PDF compatibility restrictions.

### PDF Compatibility Rule for Markdown:

- All markdown files **MUST** be PDF-compatible (no emoji or Unicode symbols that break LaTeX/PDF conversion)
- Use plain text alternatives instead of emoji:
  - [OK] (checkmark) → [OK], OK, or PASS
  - (cross mark) → [X], FAIL, or ERROR
  - WARNING: (warning) → WARNING: or WARN:
  - \* (bullet) → \* or -
  - Major Features: (target) → Major Features: or Features:
  - Enhancements: (sparkles) → Enhancements: or Improvements:
  - Bug Fixes: (bug) → Bug Fixes:
  - Documentation: (books) → Documentation:
  - Technical: (wrench) → Technical: or Technical Improvements:
  - Statistics: (chart) → Statistics: or Metrics:
  - Migration: (arrows) → Migration: or Migration Notes:
  - Deployment: (rocket) → Deployment: or Deployment Notes:
  - Notes: (memo) → Notes: or Commit History:
- **Rationale:** LaTeX fonts (e.g., lmroman10-regular) used for PDF conversion do not support emoji characters, causing warnings and formatting issues.

### Common Replacements:

- (checkmark) → [OK], OK, or PASS
- (cross mark) → [X], FAIL, or ERROR

- (warning) → WARNING: or WARN:
- • (bullet) → \* or -

**Rationale:** PowerShell, batch files, and many other tools can fail to parse files correctly when multi-byte UTF-8 characters are present, leading to cryptic parsing errors. Additionally, PDF conversion tools require plain text alternatives to emoji.

---

## Commit Message Standards

**Standard:** All commits MUST follow strict Angular Conventional Commits format, regardless of branch.

### Angular Format Structure (Mandatory)

```
<type>(<scope>): <short summary>

<body>

<footer> (optional, only for BREAKING CHANGE)
```

#### Requirements:

- **Type:** One of: feat, fix, docs, style, refactor, perf, test, build, ci, chore
- **Scope:** Optional, lowercase, in parentheses (e.g., (readme), (dashboard))
- **Subject:** Imperative, lowercase, < 50 characters
- **Blank line:** MANDATORY between subject and body
- **Body:** Detailed explanation, wrap at 72 characters, bullet points allowed
- **Footer:** Only for BREAKING CHANGE or issue tracking

### For Feature Branch Commits

- **Must follow Angular format strictly**
- Use appropriate type (feat, fix, docs, refactor, etc.)
- Include body with bullet points describing changes
- Example:

```
feat(dashboard): add user dashboard with widgets
```

Implements main dashboard view with widget framework.  
Adds real-time data updates and user preferences.

- Created dashboard view component
- Implemented widget system
- Added API integration for live data

## For Merge Commits to Main

- Must follow Angular format strictly:  
`feat(dashboard): add user dashboard with widgets`

`Implements main dashboard view with widget framework.`  
`Adds real-time data updates and user preferences.`

- Created dashboard view component
- Implemented widget system
- Added API integration for live data
- **Subject:** User-facing feature description
- **Body:** What the feature does (not implementation details)
- **Format:** Lowercase, imperative, < 50 chars for subject

## For Direct Commits to Main (Rare)

- Follow Angular format strictly
  - Use appropriate type (fix, docs, chore)
  - Include detailed body for significant changes
- 

# Version Management

## How Version.py Works

- `contest_tools/version.py` contains `__version__` and `__git_hash__`
- Pre-commit hook automatically updates `__git_hash__` before each commit
- `__version__` should match most recent git tag (manual update before releases)

## Release Workflow: Creating a New Version Tag

**Purpose:** Ensure version consistency across `version.py`, documentation, and git tags when creating a new release.

**When to Use:** Before creating any version tag (alpha, beta, or release).

## Step-by-Step Process 1. Pre-Release Checklist

- Verify current branch state (`git status`)
- Ensure all changes are committed
- Review recent commits to determine appropriate version increment
- Confirm target version number with user (e.g., `v1.0.0-alpha.3`)

## 2. Update Version References

AI should update all version references in this order:

a. Update `contest_tools/version.py`:

```
--version__ = "1.0.0-alpha.3" # Match target tag (without 'v' prefix)
```

b. Update `README.md`:

- Line 3: `**Version: 1.0.0-alpha.3**`
- Verify no other version references exist

c. Check other documentation files:

- Search for version strings: `grep -r "1\.\.0\.\.0" Docs/ README.md`
- Update any hardcoded version references found

### 3. Create Release Notes

a. Create release notes file:

- Location: `ReleaseNotes/RELEASE_NOTES_1.0.0-alpha.3.md`
- Format: Use previous release notes as template
- Content: Summary of commits since last tag

b. Generate commit summary:

```
git log --oneline <last-tag>..HEAD
```

- Organize by category (Features, Enhancements, Bug Fixes, Documentation)
- Include all significant changes

### 4. Commit Version Updates

a. Stage all version-related changes:

```
git add contest_tools/version.py README.md ReleaseNotes/RELEASE_NOTES_1.0.0-alpha.3.md
```

b. Create commit:

```
git commit -m "chore(release): bump version to 1.0.0-alpha.3 and add release notes"
```

- Use `chore(release)`: type for version bumps
- Include "bump version" in message
- Mention release notes if created

### 5. Create and Push Tag

a. Create annotated tag:

```
git tag -a v1.0.0-alpha.3 -m "Release v1.0.0-alpha.3
```

[Summary of major changes]

[Key features/enhancements]

"

- Use `-a` for annotated tag (recommended)

- Include meaningful tag message
- Tag name must match version.py (with 'v' prefix)

b. Verify tag:

```
git tag -l "v1.0.0-alpha.3"
git show v1.0.0-alpha.3
```

c. Push tag (user executes):

```
git push origin v1.0.0-alpha.3
git push origin <branch-name> # Push commits too
```

## 6. Post-Release Verification

a. Verify version consistency:

```
# Check version.py matches tag
grep "__version__" contest_tools/version.py
git describe --tags
```

```
# Check README matches
grep "Version:" README.md
```

b. Verify release notes exist:

```
ls ReleaseNotes/RELEASE_NOTES_1.0.0-alpha.3.md
```

## AI Agent Responsibilities

• AI Can Do:

- Suggest version number based on commit history
- Update version.py and README.md
- Generate release notes from commit history
- Create commit message for version bump
- Generate tag creation command
- Verify version consistency

• User Must:

- Confirm target version number
- Review release notes content
- Execute tag push command
- Verify final state

## Version Number Guidelines

- **Alpha versions:** 1.0.0-alpha.X (e.g., 1.0.0-alpha.3)
- **Beta versions:** 1.0.0-beta.X (e.g., 1.0.0-beta.5)
- **Release candidates:** 1.0.0-rc.X (e.g., 1.0.0-rc.1)
- **Releases:** 1.0.0, 1.1.0, 2.0.0 (SemVer)

## Quick Reference Commands

```
# Find last tag
git describe --tags --abbrev=0

# List commits since last tag
git log --oneline $(git describe --tags --abbrev=0)..HEAD

# Verify version consistency
grep -r "1\.\.0-alpha" contest_tools/version.py README.md ReleaseNotes/

# Create and push tag (after version updates committed)
git tag -a v1.0.0-alpha.3 -m "Release v1.0.0-alpha.3"
git push origin v1.0.0-alpha.3
```

## Legacy: Simple Tag Creation (Deprecated)

For quick tags without full release process, see "Before Creating a Tag" section below. However, **full release workflow is recommended** for all version tags.

### Before Creating a Tag (Legacy/Quick Method)

1. User specifies target version (e.g., v1.0.0-beta.5)
2. AI suggests updating `version.py`:  
`--version__ = "1.0.0-beta.5"`
3. User confirms version number
4. AI generates commit message:  
`chore: bump version to 1.0.0-beta.5`
5. User commits version change
6. User creates tag: `git tag v1.0.0-beta.5`
7. User pushes tag: `git push origin v1.0.0-beta.5`

**Note:** This method does NOT update `README.md` or create release notes. Use full Release Workflow for proper releases.

## Untagged Commits

- Version string will be: "1.0.0-beta.5-3-gabc1234" (most recent tag + distance + hash)
  - Git hash always included: "1.0.0-beta.5 (commit abc1234)"
  - This works for bug reports - version + hash = exact code state
- 

## Workflow Examples

### Example 1: Feature Development

User: "Commit these changes with a message about adding the dashboard"

AI:

1. Generates commit message following strict Angular format:

feat(dashboard): add user dashboard with widgets

Implements main dashboard view with widget framework. Adds real-time data updates and user preferences.

- Created dashboard view component
- Implemented widget system
- Added API integration for live data

2. Executes: git commit with properly formatted message (using multiple -m flags or single -m)

3. Confirms: "Committed to feat/dashboard branch"

### Example 2: Merge to Main

User: "Merge feat/dashboard into main"

AI:

1. Generates: git checkout main && git pull  
2. Generates: git merge --no-ff feat/dashboard  
3. Opens editor with merge message:  
"feat(dashboard): add user dashboard with widgets"

Implements main dashboard view with widget framework.  
4. Waits for user to review/edit and save  
5. Confirms: "Merge complete. Ready to push?"  
6. User: "Push it"  
7. AI suggests: git push origin main  
8. User executes manually

### Example 3: Creating a Release Tag

User: "Tag the next beta release"

AI:

1. Checks recent commits for version bump  
2. Suggests: "Based on recent commits, suggest v1.0.0-beta.5"  
3. User: "Yes"  
4. AI updates version.py \_\_version\_\_ = "1.0.0-beta.5"  
5. AI generates: git commit -m "chore: bump version to 1.0.0-beta.5"  
6. AI suggests commands:  
git tag v1.0.0-beta.5  
git push origin v1.0.0-beta.5  
7. User reviews and executes

---

## Error Handling

### If Pre-commit Hook Fails

- AI should explain the error
- Do NOT skip hooks without user permission
- Fix the issue (e.g., update version.py manually) before proceeding

### If Merge Conflicts

- AI should detect conflicts
- AI should suggest resolution steps
- **User must resolve** conflicts manually
- AI can help review conflict resolution

### If Version Mismatch

- AI should warn if `version.py` doesn't match recent tag
  - AI should suggest updating version before tagging
  - User must confirm version number
- 

## Safety Checks

Before any critical operation, AI should:

1. [OK] Confirm current branch (`git branch`)
  2. [OK] Show recent commits (`git log --oneline -5`)
  3. [OK] Warn if on main branch for direct commits
  4. [OK] Suggest creating feature branch if not on one
  5. [OK] Verify pre-commit hooks are installed
- 

## Session Context & Workflow State

### HANOVER.md Pattern (Optional)

For complex multi-session workflows, a `HANOVER.md` file may exist at the repository root to maintain workflow state between agent sessions.

#### When HANOVER.md Exists:

- AI agent **must** check for `HANOVER.md` at session start
- AI agent **must** read and incorporate workflow state from `HANOVER.md`
- AI agent **should** update `HANOVER.md` as work progresses
- AI agent **should** overwrite `HANOVER.md` when switching sessions

#### When HANOVER.md Does NOT Exist:

- AI agent proceeds with standard context (git history, documentation, code, user request)
- AI agent does NOT create `HANOVER.md` unless explicitly requested
- Standard workflow: User request → Agent analyzes code/docs → Agent performs task

#### **When to Use `HANOVER.md`:**

- [OK] Complex multi-session workflows (e.g., version cleanup, major refactoring)
- [OK] Multi-step processes spanning multiple sessions (test → patch → tag → cleanup)
- [OK] Context that spans multiple sessions (current state, next steps, decisions)

#### **When NOT to Use `HANOVER.md`:**

- Simple bug fixes (agent can analyze code directly)
- Single-session tasks (new features, quick fixes)
- Tasks with sufficient context in code/docs/git history

#### **Agent Behavior:**

1. **Session Start:** Check for `HANOVER.md` (non-blocking)
2. **If Found:** Read workflow state, incorporate into context
3. **If Not Found:** Proceed with standard context gathering
4. **During Work:** Update `HANOVER.md` if it exists and workflow state changes
5. **Session End:** Overwrite `HANOVER.md` with current state if it exists

**File Location:** Repository root (`HANOVER.md`)

**Git Treatment:** Committed but overwritten (allows backup in git history while maintaining current state)

---

## **Summary**

**AI's Role:** Generate, suggest, assist, format **User's Role:** Review, approve, execute, control **Balance:** AI efficiency + Human oversight = Safe, consistent workflow