Engineering Standards & Contribution Guide

1. Core Philosophy

We are modernizing our version control workflow.

- **Git is the Source of Truth:** We do **not** use inline change logs, author names, or date stamps inside source files. The Git history is our official record.
- **The ”Main” Branch is Sacred:** The `main` branch must always represent a stable state (or a stable Beta state).
- **Release vs. Development:** We separate the act of ”saving work” (Commits) from the act of ”publishing software” (Tags).

---

2. Character Encoding Policy

7-bit ASCII Requirement

All non-markdown text files must use 7-bit ASCII encoding (characters 0-127 only).

This includes but is not limited to:

- Source code files (`.py`, `.js`, `.ts`, `.html`, `.css`, etc.)
- Script files (`.ps1`, `.bat`, `.sh`, etc.)
- Configuration files (`.json`, `.yaml`, `.ini`, etc.)
- Documentation source files (`.rst`, `.txt`, etc.)

Rationale: 7-bit ASCII ensures maximum compatibility across all tools and environments:

- PowerShell and batch files parse correctly without encoding issues
- Works with any encoding interpretation (ASCII, UTF-8, Windows-1252, etc.)
- No risk of multi-byte character sequences causing parsing errors
- Universally supported on all platforms and terminals

Exceptions:

- Markdown files (`.md`) are allowed to contain UTF-8 characters including emoji, as markdown renderers handle UTF-8 encoding correctly.

Replacement Guidelines for Common Characters:

- (checkmark) → `[OK]`, `OK`, or `PASS`
- (cross mark) → `[X]`, `FAIL`, or `ERROR`
- (warning) → `WARNING:` or `WARN:`
- • (bullet) → `*` or `-`
- © (copyright) → `(c)`, `Copyright`, or `(C)`
- ® (registered) → `(R)` or `Registered`

- Accented characters (é, ñ, ü, etc.) → Use ASCII equivalents (e, n, u) or spell out

---

3. File Header Standard

All Python (`.py`) files must begin with the following standard header block. **Note for AI Agents:** When creating new files, you **must** apply this template.

```
# {filename}.py
#
# Purpose: {A concise, one-sentence description of the module's primary responsibility.}
#
# Copyright (c) {Year} Mark Bailey, KD4D
# Contact: kd4d@kd4d.org
#
# License: Mozilla Public License, v. 2.0
#          ([https://www.mozilla.org/MPL/2.0/](https://www.mozilla.org/MPL/2.0/))
#
# This Source Code Form is subject to the terms of the Mozilla Public
# License, v. 2.0.
# If a copy of the MPL was not distributed with this
# file, You can obtain one at [http://mozilla.org/MPL/2.0/](http://mozilla.org/MPL/2.0/).
```

Important: We do **not** include Author, Date, or Version in file headers. Git is our source of truth for this information.

Note: This same principle applies to data files (JSON, etc.) and template content (HTML). Do **not** include version tags, version numbers, or date metadata in JSON data structures or hardcode version strings in HTML templates. The version information in `contest_tools/version.py` (derived from git tags) is the single source of truth for application versioning.

---

4. Versioning Strategy (SemVer)

We follow **Semantic Versioning 2.0.0**.

Version Source of Truth

- **Git Tags** are authoritative for releases (e.g., `v1.0.0`, `v1.0.0-beta.1`)
- **Runtime Version** is stored in `contest_tools/version.py` as `__version__`
- **Git Hash** is automatically updated by pre-commit hooks and included in version strings
- Version strings include git hash for traceability: `"1.0.0-beta.1 (commit abc1234)"`

The Beta Lifecycle

- **Format:** `1.0.0-beta.X` or `0.x.y-beta.z`
  - Correct: `1.0.0-beta.1`, `1.0.0-beta.2`, `0.160.0-beta.5`
  - Use dots, not dashes: `beta.1` not `Beta` or `beta1`
  - During beta phase, increment prerelease identifier or patch number
- **Untagged Commits:** Use most recent tag + distance (e.g., `1.0.0-beta.5-3-gabc1234`)

The Release Lifecycle

- **Alpha (optional):** `1.0.0-alpha.1` (Early development)
- **Beta:** `1.0.0-beta.1` (Public testing)
- **Release Candidate:** `1.0.0-rc.1` (Pre-release verification)
- **Gold Release:** `1.0.0` (No suffix, production-ready)

---

5. Commit Message Standard (Angular)

We use the strict **Angular Convention**. This structure is mandatory for the final commit that lands on `main`.

The Structure

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

                          Imperative, lowercase, < 50 chars
           Optional: (api), (auth), (db)
     See list below

<BLANK LINE>  <-- MANDATORY

<body>

     Detailed explanation. Bullet points allowed.
      Wrap text at 72 chars.

<BLANK LINE>  <-- MANDATORY

<footer>

     Optional. ONLY for "BREAKING CHANGE" or issue tracking.
```

Allowed Types

Type	Description	SemVer Impact
feat	A new feature for the user.	**MINOR**
fix	A bug fix for the user.	**PATCH**
docs	Documentation only changes.	None
style	Formatting (no code change).	None
refactor	Refactoring production code.	None
perf	A code change that improves performance.	**PATCH**
test	Adding/correcting tests.	None
build	Build system changes.	None
ci	CI configuration changes.	None
chore	Miscellaneous changes.	None

Handling BREAKING CHANGES

If a commit breaks backward compatibility, the Footer **must** be the very last section, preceded by a blank line.

```
feat(api): rename user response fields

BREAKING CHANGE: API consumers accessing `userName` will now fail.
```

---

5. Git Workflow Manual (Human & AI)

A. Feature Development (The Standard Path)

All work happens in branches. We do not commit to `main`.

1. **Branch:** `git checkout -b feat/user-dashboard`
2. **Work:** Commit frequently on feature branch.
3. **Merge (--no-ff):** We use `--no-ff` merges to preserve full history while maintaining clean views.
   ```
   # 1. Switch to main
   git checkout main
   git pull

   # 2. Merge with --no-ff (preserves branch history)
   git merge --no-ff feat/user-dashboard

   # 3. Write meaningful merge commit message (Angular format)
   # Editor opens: write "feat(dashboard): add user dashboard with widgets"
   # Body can describe the feature at a high level
   ```

Viewing Clean History

Use git aliases for clean, linear-looking history (see Setup section):

```
git mainlog      # Clean main branch view (recommended for daily use)
git recent       # Last 20 commits on main (clean view)
git fullhistory  # Full history with all branches
git branchdiff   # Show commits in current branch not in main
```

Why --no-ff instead of squash?

- Preserves full branch history for debugging
- Enables `git bisect` to isolate problematic commits
- Maintains individual contributor credit
- `--first-parent` view gives clean, linear appearance
- Can still revert specific intermediate commits if needed

B. Hotfix Workflow (Emergency)

Used when a critical bug is found in a released version (e.g., `v1.0.0`), but `main` already contains new, unstable features.

1. **Time Travel:** Checkout the specific release tag. `git checkout v1.0.0`
2. **Branch:** Create a hotfix branch. `git checkout -b hotfix/security-patch`
3. **Fix:** Fix the bug and commit. `git commit -m "fix(security): resolve auth bypass"`
4. **Tag:** Tag the hotfix immediately. `git tag v1.0.1`
5. **Push:** Push the tag to deploy. `git push origin v1.0.1`
6. **Backport (Critical):** Merge the fix back into `main` so it isn't lost.
   ```
   git checkout main
   git merge hotfix/security-patch
   # Resolve conflicts if `main` has changed significantly.
   ```

C. LTS / Support Workflow (Enterprise)

Used for customers stuck on old versions (e.g., v1.0) while we are working on v2.0.

- **Rule: Upstream First.** Never fix a bug in the support branch first.
- **Workflow:**
  1. Fix the bug on `main` and commit (Hash: `A1B2C3`).
  2. Checkout the support branch: `git checkout support/v1.0.x`.
  3. **Cherry Pick:** `git cherry-pick A1B2C3`.
  4. Tag the support release: `git tag v1.0.5`.

---

7. AI Agent Rules for Git Operations

Standard Cursor/Console Workflow

Is it standard for AI to handle git operations in Cursor? Yes, this is becoming increasingly common, but with important considerations:

Current Practice:

- Many Cursor users have AI handle commits for code it generated
- Users typically review proposed commit messages before execution
- Cursor shows commands before execution (allows review)
- Common pattern: AI suggests → User reviews → User executes

Our Hybrid Approach: We use a hybrid model where AI generates git operations, but the user reviews and executes critical operations:

- **AI Generates:** Commit messages, merge commands, suggestions
- **User Reviews:** Merge messages to main, tag creation, release operations
- **User Executes:** Final merges, tags, pushes (security/safety control)

This balances AI efficiency with human oversight for critical operations.

When User Requests Git Operations

AI Can Generate (User Reviews/Executes):

- Generate commit messages following Angular format
- Suggest merge commands with `--no-ff` flag
- Update version.py via pre-commit hooks (automatic)
- Create feature branch commits
- Suggest git commands for workflows

User Should Review/Execute (Critical Operations):

- Final merge to main (AI generates, user reviews merge message)
- Creating tags/releases (user must approve tag name)
- Pushing to remote (security control)
- Hotfix operations (safety)

Commit Message Generation Rules

When generating commit messages:

1. **Follow Angular Convention strictly:**
    - Format: `<type>(<scope>): <summary>`
    - Type must be: feat, fix, docs, style, refactor, perf, test, build, ci, chore
    - Subject: imperative, lowercase, < 50 chars
    - Body: Detailed explanation with bullet points (optional but recommended)
    - Footer: Only for BREAKING CHANGE or issue tracking
2. **For merge commits to main:**
    - Write a meaningful merge commit message summarizing the feature
    - Use Angular format: `feat(scope): feature summary`

    - Body should describe what the feature does, not implementation details

3. **For feature branch commits:**
    - Can use ”wip” or informal messages
    - Final merge commit should be properly formatted

Merge Workflow for AI

```
# AI generates these commands, user reviews:
git checkout main
git pull
git merge --no-ff feat/branch-name
# Editor opens: AI-generated merge message, user reviews/edits
```

Tag Creation Workflow

```
# AI suggests, user confirms:
# 1. Update version.py __version__ to match target tag
# 2. Commit version change
# 3. Create tag: git tag v1.0.0-beta.5
# 4. Push tag: git push origin v1.0.0-beta.5
```

Workflow State Management (HANDOVER.md)

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

When to Use HANDOVER.md:

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

When NOT to Use HANDOVER.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:

- If `HANDOVER.md` exists: Agent reads workflow state and incorporates it
- If `HANDOVER.md` does NOT exist: Agent proceeds with standard context (git history, docs, code)
- Agent does NOT create `HANDOVER.md` unless explicitly requested

For detailed rules on HANDOVER.md usage, see `Docs/AI_AGENT_RULES.md` (Section ”Session Context & Workflow State”).

8. Setup Instructions

Quick Setup (Recommended)

From Command Prompt (cmd.exe):

```
# One command sets up everything (environment + git workflow)
setup.bat --setup-git
```

Or just git workflow tools:

```
tools\setup_git_workflow.bat
```

Note: You can run these from regular Command Prompt - the scripts will automatically invoke PowerShell if needed. No need to open PowerShell manually.

What Gets Set Up

1. **Pre-commit Hooks**
    - Automatically updates `__git_hash__` in `contest_tools/version.py` before each commit
    - Requires: `pip install pre-commit` (if not already installed)
2. **Git Aliases** (for clean history viewing)
    - `git mainlog` - Clean main branch view (recommended for daily use)
    - `git recent` - Last 20 commits on main
    - `git fullhistory` - Full history with all branches
    - `git featurelog` - See everything with branch structure
    - `git branchdiff` - Show commits in current branch not in main

Manual Setup (Alternative)

If you prefer to set up manually:

Install Pre-commit Hooks:

```
pip install pre-commit
pre-commit install
```

Setup Git Aliases Manually:

```
git config alias.mainlog 'log --first-parent --oneline --graph --decorate'
git config alias.recent 'log --first-parent --oneline --graph -20'
git config alias.fullhistory 'log --all --graph --oneline --decorate'
git config alias.featurelog 'log --oneline --graph --all --decorate'
git config alias.branchdiff 'log --oneline --graph main..'
```

Verification

After setup, verify everything works:

```
# Check git aliases
git mainlog

# Check pre-commit hooks
pre-commit --version

# Check version.py
python -c "from contest_tools.version import get_version_string; print(get_version_string()
```

Viewing All Files in Git Status

By default, `git status` groups untracked files in directories together. To see all files individually:

```
git status --untracked-files=all
```

This is useful when you want to see exactly which files are untracked, rather than seeing them grouped by directory (e.g., `tools/` instead of individual files).

For more detailed information, tutorials, and troubleshooting, see `tools/README.md` and `Docs/GitWorkflowSetup.md`.

---

9. AI Agent Prompt Library

Copy these prompts to instruct your AI agent on the correct protocols.

Prompt: Commit Message Generation

> ”Write a git commit message following the strict Angular Conventional Commits standard. Use one of these types: feat, fix, docs, style, refactor, perf, test, build, ci, chore. The subject must be lowercase and imperative. If the commit touches multiple files, list the changes in the body using bullet points. If there is a breaking change, use the 'BREAKING CHANGE:' footer format with a mandatory blank line before it.”

Prompt: File Creation

> ”When generating the new Python file, please ensure you include the standard project header block with the Copyright and License (Mozilla Public License 2.0). Do NOT include Author, Date, or Version fields as Git is our source of truth.”

Prompt: Merge to Main

”Generate a merge commit command using --no-ff. Write a merge commit message in Angular format that summarizes the feature being merged. The message should describe what the feature does from the user's perspective.”

Prompt: Hotfix Guidance

”I need to perform a Hotfix on version `v1.0.0`. Please give me the CLI commands to checkout that tag, create a branch named `hotfix/my-fix`, and after I commit, show me how to tag it as `v1.0.1` and merge it back into `main`.”