ChatGPT

# Day 3: Git Commit Discipline and Repository Hygiene

## Goals

- **Enforce Conventional Commit messages:** Implement a Git hook to automatically reject commit messages that don't follow the Conventional Commits format (e.g. `feat(scope): add feature` as per spec [1]).
- **Establish repository code ownership:** Define a CODEOWNERS file so that `@sir-mammut` is set as the default code owner for all files, ensuring they are automatically requested for review on pull requests.
- **Improve repository cleanliness:** Update the `.gitignore` to exclude common unwanted files and directories (like editor settings, OS artifacts, dependencies, and logs) as seen in the repository structure screenshot.
- **Configure GPG commit signing (macOS):** Set up GPG tools on macOS, generate a GPG key, configure Git to sign commits, and upload the public key to GitHub so that commits show as "Verified".
- **Demonstrate signed conventional commits and squash merging:** Make a commit on a feature branch (`day-03-git`) following the new rules (signed and properly formatted) and perform a squash merge into the main branch to maintain a clean history.

## Actions

### Enforcing Conventional Commits with a Git Hook

To ensure every commit message is standardized, I set up a **commit-msg** Git hook that validates the message format against the Conventional Commits specification. According to the spec, a commit message should have a type, optional scope, colon, and a description (e.g. `fix(parser): handle null inputs`) [1]. I created a dedicated directory for version-controlled hooks:

- **Created `.githooks/commit-msg`:** This script runs on each commit to check the message. It uses a regex to allow types like `feat, fix, docs, style, refactor, perf, test, chore, build, ci` (common Conventional Commit types) and an optional `(scope)`. If the message doesn't match, the hook exits with an error, preventing the commit.

```
# .githooks/commit-msg
#!/bin/sh
# commit-msg hook to enforce Conventional Commits format on the commit message
# Allowed types: build, chore, ci, docs, feat, fix, perf, refactor, style, test.
# Scope is optional. Format: type(scope): description
commit_msg=$(head -n1 "$1")  # Only check the first line (commit summary)
```

```
pattern='^(build|chore|ci|docs|feat|fix|perf|refactor|style|test)(\(.+\))?: .+'
if ! echo "$commit_msg" | grep -Eq "$pattern"; then
    echo "ERROR: Commit message does not follow Conventional Commits format."
>&2
    echo "Please use \"type(scope): description\" format. For example:
\"feat(ui): add search feature\"" >&2
    exit 1
fi
```

*(After creating the file, I made sure to mark it as executable with* `chmod +x .githooks/commit-msg`*.)* The hook works by reading the commit message from the file path passed as `$1`, checking the first line against the regex. If the format is wrong, it prints an error and aborts the commit.

- **Activated custom hooks path:** By default, Git only runs hooks in `.git/hooks`. To use our versioned hooks folder, I configured the repo to use `.githooks` as the hooks directory:

```
git config core.hooksPath .githooks
```

This command tells Git to load hooks from the `.githooks` directory in the repository [2]. (This setting is saved in the local repo's Git config, so teammates need to run the same command or it can be included in the project setup docs.) With this, our `commit-msg` hook will run on every commit.

- **Tested the hook:** To verify, I attempted a commit with a bad message (`"Bad commit"`). As expected, the hook rejected it, outputting an error message and instructions:

```
ERROR: Commit message does not follow Conventional Commits format.
Please use "type(scope): description" format. For example: "feat(ui): add search
feature"
```

Next, I tried a properly formatted message: `git commit -m "feat(repo): add commit hook and codeowners"`. The hook passed this and allowed the commit. This ensures all future commits will follow a consistent format, improving readability and enabling automated changelogs.

## Defining Code Ownership with CODEOWNERS

To improve code review workflow and accountability, I added a CODEOWNERS file. This file makes `@sir-mammut` **the default code owner** for the entire repository. According to GitHub's CODEOWNERS rules, using the pattern `*` will match all files in the repo, and listing a username assigns them as the code owner [3].

- **Added `.github/CODEOWNERS`:** I created a file at `.github/CODEOWNERS` with the following content:

```
* @sir-mammut
```

This single line means for any file (`*` wildcard) changes, GitHub will automatically request a review from `@sir-mammut` on any incoming pull request. In other words, `@sir-mammut` becomes the default owner for all code in the repo [3]. This is a great practice for a solo or mentored project – it ensures the mentor (sir-mammut) is looped into all changes. The CODEOWNERS file is now tracked in version control, and once pushed to the default branch, GitHub will enforce it on new PRs (for any branch based off that default).

*Side note:* In larger projects, CODEOWNERS can have multiple patterns and owners for different parts of the codebase (e.g., specific teams for certain directories). But here, we keep it simple with a global owner. Since `@sir-mammut` has write access, they will be automatically assigned to review all PRs [4]. If required, branch protection rules can even be set so that code owner approval is mandatory before merge.

### Updating .gitignore for Repository Hygiene

A clean repository avoids committing unnecessary or sensitive files. I reviewed the repository structure (and the screenshot provided) to identify patterns to ignore. Common culprits include editor settings, OS files, dependency directories, and logs. I updated the root **.gitignore** to include these:

```
# Ignore editor settings and OS artifacts
.vscode/
.DS_Store

# Ignore Node.js dependencies and build outputs
node_modules/

# Ignore log files and directories
logs/
*.log
```

**Explanation of entries:**

- `.vscode/` – Ignores the entire VS Code config folder, which often contains local settings and operational logs (e.g. `settings.json`, `*.log` files from tasks). These shouldn't be in version control as they are developer-specific.
- `.DS_Store` – Ignores macOS Finder's custom folder attribute file. This file is automatically created on macOS in folders and is not useful to share.
- `node_modules/` – Ignores Node.js dependencies. Since dependencies are tracked via `package.json` and `package-lock.json`, the actual `node_modules` directory (which can be thousands of files) should not be committed.
- `logs/` and `*.log` – Ignores any `logs` directory and any file with a `.log` extension. This covers log output directories (perhaps created by the app or tests) and individual log files (e.g. `debug.log`, `npm-debug.log`). These files are generally ephemeral and should remain local.

After editing `.gitignore`, I did a quick check by creating dummy files/folders (like a `.vscode/settings.json`, a `node_modules/test.js`, a `logs/app.log`, etc.). Running `git status` showed that none of these appeared as untracked files, confirming that the ignore rules work. This **repository hygiene** step prevents clutter and the accidental commit of large or private files.

## Configuring GPG Commit Signing on macOS

Another aspect of professional Git practice is **signing commits with GPG**, which verifies the author's identity. We want all commits to be signed so that they show as "Verified" on GitHub and ensure authenticity. Here's how I set this up on macOS:

1. **Install GPG tools:** If not already installed, I installed GnuPG via Homebrew: `brew install gnupg`. This provides the `gpg` command-line tool. (Homebrew may refer to it as `gpg2` but it's the same; it also installs `gpg-agent` for key management. Optionally, install `pinentry-mac` for GUI passphrase prompts.) [5]

2. **Generate a new GPG key pair:** I ran `gpg --full-generate-key` to create a new key. The `--full` flag gives a detailed prompt where I chose:

3. Key type: RSA and RSA (default)
4. Key size: 4096 bits (for strong security)
5. Expiration: chose "never expire" (or you can set an expiration as a policy)
6. Name and email: **Use the same name and email as my Git user** (this is important – the email must match the one used in Git commits and associated with my GitHub account). I entered my name and my GitHub-verified email here.
7. Passphrase: set a secure passphrase to protect the private key (when prompted by GPG). *(For simplicity in this demo, one could use no passphrase, but in a real setup a passphrase is recommended. GPG will integrate with macOS Keychain if configured, so you don't have to enter it every time.)*

GPG then generated the key pair (public and private key). It also printed the key fingerprint/ID. If not, I can list it with: `gpg --list-secret-keys --keyid-format=long "<your email>"`. The output looks like:

```
sec   rsa4096/ABCDEFGHIJ123456 2025-08-26 [SC] [expires: 2026-08-25]
      1234ABCD5678EFGH9012IJKLABCDEFGHIJ123456
uid                 [ultimate] Your Name <youremail@example.com>
```

The long string on the second line is the full fingerprint. The shorter `RSA key ID` (10 hex characters in this example, or sometimes 16) after the slash (`ABCDEFGHIJ123456` in example) is what we'll use as the key ID.

1. **Export the public key:** To use this key on GitHub, I exported the public portion in ASCII-armored format with:

   ```
   gpg --armor --export <YourKeyID>
   ```

   This prints out a block starting with `-----BEGIN PGP PUBLIC KEY BLOCK-----`. I copied that entire block.

2. **Add the GPG key to GitHub:** In my GitHub account settings, under "SSH and GPG keys," I clicked "New GPG key" and pasted the armored public key block. Once added, GitHub will show this key as linked to my account. Now any commits signed by this key will show as verified on GitHub.

3. **Tell Git to use the key:** I configured Git with my new key ID and to sign commits by default:

   ```
   git config --global user.signingkey <YourKeyID>
   git config --global commit.gpgSign true
   ```

   The first line sets the signing key in your global config. The second line ensures all commits are GPG-signed by default [6] (so I don't have to remember to add `-S` every time). I also confirmed my `user.name` and `user.email` in Git config match the GPG uid.

4. **Ensure passphrase caching (optional):** On macOS, to avoid frequent passphrase prompts when signing, I made sure the GPG agent is properly configured. One common step is adding `export GPG_TTY=$(tty)` to my shell profile (`~/.zshrc` or `~/.bash_profile`) and sourcing it [7], so that the terminal can prompt for the passphrase. Additionally, using the GPG Suite on Mac can integrate with the Keychain [8] to remember the passphrase. In my setup, since I set the key with no passphrase for testing, this wasn't an issue; but in a real scenario, this step prevents signing from hanging if GPG can't prompt for the password.

With these steps completed, my Git is now set to sign commits. I verified the configuration by running `git config --list` and checking that `user.signingkey` and `commit.gpgSign=true` are present.

## Making a Signed Conventional Commit (Feature Branch Workflow)

With all the pieces in place (hook, codeowners, .gitignore, GPG), I proceeded to make a commit implementing today's changes. Following best practices, I did this work on a **separate branch** and will use a **squash merge** to integrate it.

- **Created a feature branch:** I started a new branch for Day 3 work:

  ```
  git checkout -b day-03-git
  ```

  This isolates my changes until they are reviewed and merged, keeping the main branch history clean.

- **Staged all changes:** The new files created (`.githooks/commit-msg`, `.github/CODEOWNERS`, updated `.gitignore`, and the `journal/day03.md` log file itself) were added via `git add .`. I double-checked `git status` to ensure all intended changes are staged and nothing stray is included.

- **Commit with Conventional format and GPG signature:** I then committed on this branch with a message following the Conventional Commits style and signed by GPG. Because I set `commit.gpgSign=true`, the commit was automatically signed. The command and message used:

```
git commit -m "feat(repo): enforce commit conventions and add codeowners [Day
3]"
```

*(The* `feat(repo): ...` *message summarizes the changes: it's a new feature/improvement in repository configuration. I included a bracket note* `[Day 3]` *just for clarity in the log; that's optional.)*

Git invoked the commit-msg hook first, which checked the message format. The message started with `feat(repo):` so it passed the regex. Then Git created the commit and GPG signed it with my key. Since my key had no passphrase (or if the passphrase is cached), the signing happened non-interactively.

- **Verified the commit locally:** I ran `git log -1 --show-signature` to see details of the last commit. The output confirmed that the commit was signed and valid:

```
gpg: Signature made Tue Aug 26 14:36:47 2025 UTC
gpg:                using RSA key 1234567890ABCDEFG
gpg: Good signature from "Your Name <youremail@example.com>" [ultimate]
commit 1a2b3c4 (HEAD -> day-03-git)
Author: Your Name <youremail@example.com>
Date:   Tue Aug 26 14:36:47 2025 +0000

    feat(repo): enforce commit conventions and add codeowners [Day 3]
```

The key lines are **"Good signature from …"** which means Git was able to verify the commit against my public key. On GitHub, this commit now shows a "Verified" badge, indicating it's signed with a key that GitHub trusts for my account. (If the signature was invalid or the key wasn't added to GitHub, it would show as unverified.)

- **Pushed the branch and opened a PR:** Next, I pushed `day-03-git` to the remote repository and opened a pull request. Because of the CODEOWNERS file, GitHub automatically added `@sir-mammut` as a required reviewer for the PR. This is exactly the intended outcome – the mentor/owner is notified to review the changes.

## Squash Merging the Changes

After review (and any approvals), the final step is to merge the Day 3 branch back into the main line. To keep the history tidy, I used a **squash merge** strategy. Squash merging condenses all my intermediate commits on the feature branch into a single commit on the target branch. This maintains a linear history and one commit per feature/issue, which is easier to read.

- **Squash merge via GitHub:** In the GitHub PR, I chose "Squash and merge". I edited the commit message to a concise summary of Day 3 changes (retaining the Conventional Commit format, e.g.

`feat(repo): implement Day 3 git discipline improvements`). Upon merging, GitHub combined the branch's commits and added one new commit to `main`. That commit was, of course, GPG-signed (since my local commits were signed and GitHub will mark the merged commit as verified if the signature carries over or if using the "Sign commits" option in the repo settings – though squash via the UI typically uses GitHub's key to sign if enabled).

*Alternatively,* I could do this locally: check out the main branch, run `git merge --squash day-03-git`, then commit the squash. This would stage all changes from the feature branch without creating a merge commit. I tried this approach in testing – after `git merge --squash`, all changes were staged, and I ran `git commit -m "feat(repo): implement Day 3 changes (squashed)"`. The result was one new commit on main containing all the modifications. The commit was GPG-signed (since my config signs commits by default). A `git log --oneline` of `main` then showed a new single commit from Day 3 on top of previous work.

- **Verified history and cleanup:** Finally, I confirmed that the main branch has the intended single commit for Day 3's work. The commit message is in Conventional format and marked verified. The intermediate branch commits are not in main, keeping history succinct. I also deleted the `day-03-git` branch from the remote to tidy up, since its work is merged.

## Gotchas and Lessons Learned

- **Making the hook available to all contributors:** The `core.hooksPath` setting is local and isn't automatically set for others. This means if someone else clones the repo, they must run `git config core.hooksPath .githooks` to activate the shared hooks [2]. A way to enforce this is to document it (e.g., in the README or in a setup script). Alternatively, one can include a `.gitconfig` in the repo that sets hooksPath and instruct users to include it [9]. In a team setting, using a tool like Husky or a repository template can also automate this. For now, I noted this requirement in the project docs.

- **Hook script execution:** It's important that the hook script has execute permissions (`chmod +x`). A common pitfall is forgetting that, which causes Git to silently skip the hook. I ensured the file mode was 755 in the commit (`100755` for `commit-msg` as seen in git's output).

- **Regex coverage:** The regex in the hook is somewhat strict. It requires a lowercase type and at least one character after the colon. While it catches most bad messages, it might need refinement in the future. For example, it doesn't explicitly forbid capitalized summaries or long lines. If needed, we could extend it or use a more robust solution like **commitlint**, but the current approach is a simple, dependency-free guard.

- **GPG key email mismatch:** I learned that the GPG key's email **must match** the email used in the Git commit for GitHub to verify it. If you generate a key with a different email (or add an alias), you'll need to add that email to your GitHub account or regenerate the key. I made sure to use my primary GitHub email for the GPG uid. If a commit shows "Unverified" despite being signed, this is the first thing to check.

- **GPG passphrase issues on macOS:** Initially, my signed commits failed because GPG couldn't get the passphrase (the commit would hang or error). Setting the `GPG_TTY` environment variable fixed the issue [7], as it allows the pinentry to prompt in the terminal. Alternatively, using the GPG Suite provides a macOS prompt which is more seamless. This is an example of a platform-specific hiccup – once configured, signing became painless.

- **GitHub's squash commit signature:** When using the GitHub UI to squash and merge, the commit is authored by the user who merged (or the original author) and signed with GitHub's key if the repository is configured to "Sign commits you merge". In our case, since we want our own GPG signature, another approach is to squash locally as I did in testing. In a real project, enabling "Allow squash merging" and "Require signed commits" in branch protection can ensure even the merge commits are signed (GitHub will sign them on merge). For now, all commits we pushed were signed, and that suffices.

## Verification

At the end of Day 3, I verified that all objectives were met:

- **Conventional Commit enforcement:** Attempting a commit with a non-conforming message was blocked by the hook with an error, while a proper message succeeded. The commit history now shows Day 3's commit with a neatly formatted message (e.g., `feat(repo): ...`). This will make it easier to generate changelogs and understand history going forward [1].

- **CODEOWNERS in effect:** The CODEOWNERS file is present in `.github/`. On the open PR for Day 3, I indeed saw `@sir-mammut` automatically added as a reviewer. This confirms that `* @sir-mammut` in CODEOWNERS works – any future PRs will ping the mentor by default [3].

- **Clean working tree:** Unwanted files are now ignored. I noticed that my local `.vscode` folder and `node_modules` (if any) do not show up in `git status`. The `.gitignore` entries are effective, preventing those files from ever polluting commits.

- **Signed commit visible on GitHub:** The commit I pushed shows a "Verified" label on GitHub. By clicking it, it shows the signature is from a known key of mine. Locally, `git log --show-signature` indicates a good signature from my GPG key, confirming that my setup is correct and secure [10].

- **Squashed merge result:** The main branch history now contains one new commit for Day 3's changes (instead of multiple). I checked `git log` on `main` and saw a single commit message encapsulating everything. The commit message retained the Conventional Commit style and carried the GPG signature. This approach keeps the repository history linear and clean, which will pay off in readability as the project grows.

Overall, Day 3's tasks have strengthened the repository's professionalism: every commit is now standardized and verified, the repository has an owner oversight mechanism, and we've eliminated the chance of editor clutter or secrets slipping into version control. These are small but crucial steps toward enterprise-grade development practices.

With this foundation in place, future days can build on a clean, well-governed Git workflow. The discipline enforced today will ensure the project remains maintainable and trustworthy as it evolves.

**Sources:**

- Conventional Commits format – *Conventional Commits v1.0.0 spec* [1]
- Configuring a custom Git hooks directory (`core.hooksPath`) – *Atlassian Bitbucket Knowledge Base* [2]
- CODEOWNERS default owner example – *Graphite Dev Guide on CODEOWNERS* [3]
- Enabling GPG commit signing – *GitHub Docs: Signing commits* [6] [10]

---

[1] Conventional Commits
https://www.conventionalcommits.org/en/v1.0.0/

[2] [9] Standardize git hooks across a repository | Bitbucket Cloud | Atlassian Support
https://support.atlassian.com/bitbucket-cloud/kb/standardize-git-hooks-across-a-repository/

[3] Understanding GitHub CODEOWNERS
https://graphite.dev/guides/in-depth-guide-github-codeowners

[4] About code owners - GitHub Docs
https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-code-owners

[5] [7] Signing your Git Commits on MacOS · GitHub
https://gist.github.com/troyfontaine/18c9146295168ee9ca2b30c00bd1b41e

[6] [8] [10] Signing commits - GitHub Docs
https://docs.github.com/en/authentication/managing-commit-signature-verification/signing-commits