AI VOID FIELD GUIDE

Step-by-Step Tutorials

Focused, practical tutorials that teach you how to accomplish specific tasks step by step. Learn by doing with clear instructions and working code examples.

AI VOID

2026-05-19 14:31 IST

Contents

CHAPTER 01

First Open Source Contribution GitHub

What you'll build: Successfully make your first open-source contribution to a GitHub project by following the standard workflow. **Time needed:** ~45 minutes
Prerequisites: GitHub account, Git installed, Basic command line familiarity
Version used: unknown

Introduction and Prerequisites

Welcome! Making your first open-source contribution can feel like a big step, but it's an incredibly rewarding experience. Open source is all about collaboration – people from around the world working together to build and improve software. By contributing, you'll not only help projects you care about but also gain valuable real-world development experience, learn best practices, and connect with a global community.

This tutorial will walk you through the standard GitHub workflow for contributing to an existing project, from finding a suitable task to getting your changes reviewed and merged. We'll focus on the practical steps, using the command line (CLI) and GitHub's web interface.

Before we dive in, let's make sure you have everything you need:

- **GitHub Account:** You'll need a free GitHub account to host your code, fork repositories, and open pull requests. If you don't have one, head over to github.com and sign up.
- **Git Installed:** Git is the version control system that GitHub uses. You'll use Git commands on your local machine to manage your code changes. You can download it from git-scm.com.
    - To verify Git is installed, open your terminal or command prompt and type:

```
git --version
```

```
You should see a version number (e.g., `git version 2.39.2`).
```

- **Basic Command Line Familiarity:** We'll be using basic commands like `cd` (change directory), `ls` (list files), `git clone`, `git add`, `git commit`, and `git push`. If you're new to the command line, don't worry, we'll guide you through each step.

By the end of this tutorial, you'll have successfully navigated the entire contribution process, ready to tackle more complex tasks in the future!

Finding a Good First Issue

The first step in contributing is often the hardest: finding a project and a task that's suitable for a beginner. Many open-source projects understand this and label specific issues as "good first issues" or "beginner-friendly." These are typically small, self-contained tasks that allow you to get familiar with the contribution workflow without needing deep project knowledge.

How to Locate Beginner-Friendly Tasks

There are several excellent ways to find these entry-level contributions:

1. **GitHub's `good-first-issue` Topic:** You can directly search GitHub for issues labeled `good first issue` across various repositories.
    - Go to github.com/topics/good-first-issue.
    - You can also refine your search on GitHub by adding `label:"good first issue"` to your search query, potentially combined with a language (e.g., `is:issue is:open label:"good first issue" language:python`).
2. **Dedicated Websites:**
    - goodfirstissue.dev curates easy-to-pick issues from popular open-source projects.
    - firstcontributions.github.io offers a guided tutorial using a dedicated repository, which is a fantastic place to practice the actual steps we'll cover here.

3. **Project-Specific Guidelines:** Once you find a project that interests you, always look for their `CONTRIBUTING.md` file (often in the root of the repository) or a "Contribution Guidelines" section in their `README.md`. These files often outline how to find issues, what kind of contributions are welcome, and the specific workflow they expect.

> ⚡ Note: For your very first contribution, aim for something small. Fixing a typo in documentation, improving an example, or clarifying an instruction in a `README.md` are perfect starting points. These types of changes build confidence and help you learn the mechanics without getting bogged down in complex code.

Once you've found an issue you'd like to work on, it's good practice to comment on the issue saying something like, "I'd like to work on this!" This lets maintainers know you're interested and helps avoid multiple people working on the same task simultaneously.

What you've accomplished: You've learned how to effectively find beginner-friendly issues on GitHub, setting the stage for your contribution.

Forking the Repository and Cloning Locally

Now that you've identified an issue, the next step is to get a copy of the project's code onto your own GitHub account and then onto your local machine. This is done through a process called "forking" and "cloning."

Understanding Forking

When you "fork" a repository, you're creating your own personal copy of that project on your GitHub account. This copy is completely separate from the original (the "upstream" repository). You can make any changes you want to your fork without affecting the original project. This is crucial for open-source contributions, as it allows you to experiment freely before proposing your changes back to the main project.

Step 1: Fork the Repository on GitHub

1. Navigate to the GitHub page of the project you wish to contribute to (e.g., `github.com/owner/repository-name`).
2. In the top-right corner of the page, you'll see a **"Fork"** button. Click it.

3. GitHub will ask where you want to fork the repository. Choose your own GitHub account.
4. After a moment, you'll be redirected to your new fork, which will look something like `github.com/your-username/repository-name`. Notice the small text below the repository name indicating it was forked from the original project.

You've now successfully created a personal copy of the project on your GitHub account!

Step 2: Clone Your Fork to Your Local Machine

Now, let's get that code onto your computer so you can start making changes. We'll use the `git clone` command.

1. On your forked repository's page on GitHub (e.g., `github.com/your-username/repository-name`), click the green **"Code"** button.
2. In the dropdown, select **HTTPS** (unless you've set up SSH keys, which is beyond this tutorial's scope) and copy the URL. It will look like `<https://github.com/your-username/repository-name.git>`.
3. Open your terminal or command prompt on your local machine.
4. Navigate to a directory where you want to store your project (e.g., `cd ~/Development` or `cd C:\Users\YourName\Projects`).
5. Now, run the `git clone` command, pasting the URL you copied:

```
    git clone https://github.com/your-username/repository-name.git
```

```
Replace `<https://github.com/your-username/repository-name.git>` with the
actual URL of *your fork*.

You'll see output indicating that Git is cloning the repository.
```

```
    Cloning into 'repository-name'...
    remote: Enumerating objects: 123, done.
    remote: Counting objects: 100% (123/123), done.
    # ... (more output)
```

1. Once cloning is complete, change into the newly created project directory:

```
cd repository-name
```

```
Replace `repository-name` with the actual name of the cloned directory.

To verify you are in the correct directory, you can list its contents:
```

```
ls # (on macOS/Linux)
dir # (on Windows)
```

```
You should see the project files.
```

Step 3: Add the Original Repository as an "Upstream" Remote

It's good practice to set up a remote pointing to the original repository (the one you forked from). This is called the "upstream" remote. It allows you to easily fetch updates from the original project later, keeping your fork in sync.

1. First, check the current remotes configured for your local repository:

```
git remote -v
```

```
You should see `origin` pointing to your fork's URL for both fetch and push
operations.
```

```
origin  https://github.com/your-username/repository-name.git (fetch)
origin  https://github.com/your-username/repository-name.git (push)
```

1. Now, add the original repository as `upstream`. You'll need the HTTPS URL of the original project (e.g., `<https://github.com/owner/repository-name.git>`).

```
git remote add upstream https://github.com/owner/repository-name.git
```

```
Replace `<https://github.com/owner/repository-name.git`> with the actual URL
of the *original* project.
```

1. Verify that the `upstream` remote has been added correctly:

```
git remote -v
```

```
You should now see both `origin` (your fork) and `upstream` (the original
project):
```

```
origin  https://github.com/your-username/repository-name.git (fetch)
origin  https://github.com/your-username/repository-name.git (push)
upstream        https://github.com/owner/repository-name.git (fetch)
upstream        https://github.com/owner/repository-name.git (push)
```

> ⚠️ Common mistake: Accidentally cloning the original repository directly instead of your fork. If you do this, you won't be able to push your changes later without special permissions. Always clone your fork.

What you've accomplished: You've successfully forked the project on GitHub and cloned your fork to your local machine, preparing your environment for making changes.

Creating a New Branch for Your Changes

Before you start making changes, it's crucial to create a new branch. This is a fundamental practice in Git and open-source contributions.

Why Branching is Essential

A "branch" in Git is essentially an independent line of development. When you create a new branch, you're creating a copy of the current state of the code. Any changes you make on this new branch won't affect the `main` (or `master`) branch of your local repository until you explicitly merge them.

This isolation is incredibly powerful:

- **Keeps `main` clean:** Your `main` branch remains a pristine copy of the original project.

- **Multiple features:** You can work on several features or bug fixes simultaneously, each on its own branch.
- **Easy collaboration:** When you submit your changes for review, you'll submit the branch, not your entire repository.
- **Reversibility:** If something goes wrong on your branch, you can discard it without impacting the `main` branch.

Step 1: Ensure You're on the `main` Branch

Before creating a new branch, it's always a good idea to ensure you're starting from the `main` branch and that it's up-to-date.

1. Switch to the `main` branch (if you're not already there):

```
    git checkout main
```

```
You'll see output like `Switched to branch 'main'` or `Already on 'main'`.
```

1. Pull the latest changes from the original (upstream) repository to ensure your local `main` is current:

```
    git pull upstream main
```

```
This command fetches any new commits from the original project's `main` branch
and merges them into your local `main`. This keeps your local copy from
getting too far behind the main project.
```

Step 2: Create and Switch to Your New Branch

Now, let's create a new branch specifically for your contribution. It's good practice to give your branch a descriptive name related to the issue you're fixing (e.g., `fix-typo-in-readme`, `add-new-feature-x`).

1. Create a new branch and switch to it in one command:

```
    git checkout -b fix-typo-in-readme
```

```
Replace `fix-typo-in-readme` with a meaningful name for your branch.

You'll see output like `Switched to a new branch 'fix-typo-in-readme'`.
```

1. Verify that you are on the new branch:

```
git branch
```

```
The branch you are currently on will be highlighted (often with an asterisk
`*`).
```

```
* fix-typo-in-readme
  main
```

> ⚡ Note: If you're contributing to a project that uses `master` instead of `main` as its primary branch name, adjust the commands accordingly (e.g., `git checkout master`, `git pull upstream master`).

What you've accomplished: You've created a dedicated branch for your changes, ensuring your work is isolated and ready for development.

Making and Committing Your Changes

This is where you actually implement the fix or feature you identified. For a first contribution, a simple change like fixing a typo in a `README.md` file is ideal.

Step 1: Make Your Changes

1. Open the relevant file(s) in your preferred code editor (VS Code, Sublime Text, Notepad++, etc.). For this example, let's assume you're fixing a typo in `README.md`.

```
# Open README.md in a text editor (example for VS Code)
code README.md
```

```
Or simply open your editor and navigate to the file.
```

1. Locate the typo or the section you need to modify and make your change. Save the file.

Step 2: Check Your Changes with `git status`

After saving your changes, Git needs to know about them. The `git status` command is your best friend for understanding the current state of your repository.

1. Run `git status`:

```
    git status
```

```
You'll see output similar to this, indicating which files have been modified:
```

```
    On branch fix-typo-in-readme
    Your branch is up to date with 'origin/fix-typo-in-readme'.

    Changes not staged for commit:
      (use "git add <file>..." to update what will be committed)
      (use "git restore <file>..." to discard changes in working directory)
            modified:   README.md

    no changes added to commit (use "git add" and/or "git commit -a")
```

```
The "Changes not staged for commit" section tells you that Git sees changes in
`README.md`, but they haven't been prepared for a commit yet.
```

Step 3: Stage Your Changes with `git add`

"Staging" changes means telling Git exactly which modifications you want to include in your next commit. You can stage individual files or all changes.

1. Stage the `README.md` file:

```
    git add README.md
```

```
If you had multiple files and wanted to stage all modified files, you could
use `git add .` (be careful with this, as it stages *all* changes, including
potentially unwanted ones).
```

1. Run `git status` again to see the effect:

```
    git status
```

```
Now, `README.md` should be listed under "Changes to be committed":
```

```
    On branch fix-typo-in-readme
    Your branch is up to date with 'origin/fix-typo-in-readme'.

    Changes to be committed:
      (use "git restore --staged <file>..." to unstage)
            modified:   README.md
```

Step 4: Commit Your Changes with `git commit`

A "commit" is a snapshot of your staged changes. Each commit should represent a single logical change and have a clear, concise commit message. Good commit messages are vital for project history and review.

1. Commit your staged changes:

```
    git commit -m "Fix: Corrected typo in README.md"
```

```
The `-m` flag allows you to provide a commit message directly.

> ⚡ Note: For more complex changes, you might omit `-m`, which will open your
default text editor (like Vim or Nano) to write a more detailed commit
message. The first line should be a concise summary (max 50-72 chars),
followed by a blank line, then a more detailed explanation if needed.

You'll see output confirming the commit:
```

```
    [fix-typo-in-readme 1a2b3c4] Fix: Corrected typo in README.md
     1 file changed, 1 insertion(+), 1 deletion(-) # (or similar)
```

1. Verify your commit history:

```
git log --oneline
```

```
This shows a compact view of your commits, with your latest commit at the top.
```

```
1a2b3c4 (HEAD -> fix-typo-in-readme) Fix: Corrected typo in README.md
d5e6f78 (upstream/main, origin/main, main) Initial commit
# ... (earlier commits)
```

> ⚠️ Common mistake: Writing vague commit messages like "Update" or "Fixes." Be specific! What did you change? Why? This helps maintainers understand your contribution.

What you've accomplished: You've successfully made your desired changes, staged them, and committed them to your new branch with a clear commit message.

Opening a Pull Request on GitHub

Now that your changes are committed to your local branch, it's time to propose them to the original project. This is done by "pushing" your branch to your fork on GitHub and then opening a "Pull Request" (PR).

Understanding Pull Requests

A Pull Request is essentially a formal proposal to merge your changes from your branch into another branch (usually the `main` branch of the original project). It's a central feature of GitHub that facilitates code review, discussion, and automated checks before changes are integrated.

Step 1: Push Your Branch to Your Fork

Your changes are currently only on your local machine. You need to push your new branch to your fork on GitHub.

1. Push your `fix-typo-in-readme` branch to your `origin` (your fork):

```
git push origin fix-typo-in-readme
```

```
Replace `fix-typo-in-readme` with the name of your branch.

The first time you push a new branch, Git will often give you a hint to set an
"upstream" tracking branch:
```

```
    Enumerating objects: 5, done.
    Counting objects: 100% (5/5), done.
    Delta compression using up to 8 threads
    Compressing objects: 100% (3/3), done.
    Writing objects: 100% (3/3), 329 bytes | 329.00 KB/s, done.
    Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
    remote:
    remote: Create a pull request for 'fix-typo-in-readme' on GitHub by
visiting:
    remote:      https://github.com/your-username/repository-name/pull/new/
fix-typo-in-readme
    remote:
    To https://github.com/your-username/repository-name.git
     * [new branch]      fix-typo-in-readme -> fix-typo-in-readme
```

```
Notice that GitHub even provides a direct link to create the pull request!
```

Step 2: Open the Pull Request on GitHub

1. Go to your forked repository on GitHub (e.g., `github.com/your-username/repository-name`).
2. GitHub is smart! It will often detect that you've just pushed a new branch and display a prominent banner with a **"Compare & pull request"** button. Click it.
    - If you don't see the banner, go to the "Pull requests" tab and click the **"New pull request"** button.
3. On the "Open a pull request" page:
    - **Base repository:** This should be the original project's repository (e.g., `owner/repository-name`).
    - **Base branch:** This is usually `main` (or `master`) of the original project.
    - **Head repository:** This should be your forked repository (e.g., `your-username/repository-name`).
    - **Compare branch:** This should be the branch you just pushed (e.g., `fix-typo-in-readme`).

4. **Fill out the Pull Request form:**
    - **Title:** GitHub often pre-fills this with your latest commit message. Make sure it's clear and concise (e.g., "Fix: Correct typo in README").
    - **Description:** This is crucial. Explain what changes you made and why.
        - Reference the issue number you're addressing (e.g., `Closes #123` or `Fixes #123`). This often automatically closes the issue when your PR is merged.
        - Explain the problem your change solves.
        - If applicable, describe how you tested your change.
    - Many projects have a **PR template** that automatically populates the description field. Fill this out thoroughly; it helps maintainers review your work.
5. Review the "Files changed" tab to ensure only your intended changes are present.
6. Click the green **"Create pull request"** button.

You've now successfully opened your first pull request!

> ⚠️ Common mistake: Opening a PR from your `main` branch directly. Always use a dedicated feature branch. If you push directly from `main`, any subsequent changes you make to your local `main` and push will automatically be added to that existing PR, which can get messy.

What you've accomplished: You've pushed your local branch to your GitHub fork and created a formal Pull Request to propose your changes to the original project.

Navigating the Review Process

Opening a pull request is a significant achievement, but the journey isn't over yet! Your changes will now be reviewed by the project's maintainers and potentially other community members. This review process is a vital part of open source, ensuring code quality, consistency, and alignment with the project's goals.

What to Expect During Review

1. **Automated Checks (CI/CD):** Many projects have automated tests, linters, and build processes (Continuous Integration/Continuous Delivery, or CI/CD) that run automatically when a PR is opened. You'll see status checks (green checkmark for success, red 'X' for failure) at the bottom of your PR. If any fail, you'll need to investigate and fix them.
2. **Feedback and Comments:** Maintainers will review your code and may leave comments, ask questions, or suggest improvements directly on your PR. This is a normal and healthy part of the process.
3. **Requested Changes:** It's common for maintainers to request changes. Don't take this personally! They're helping you improve your contribution and ensure it meets project standards.

How to Address Feedback and Make Further Changes

If changes are requested, here's how you incorporate them:

1. **Go back to your local branch:** On your local machine, ensure you're on the same branch where you made your original changes (e.g., `fix-typo-in-readme`).

```
git checkout fix-typo-in-readme
```

1. **Make the requested modifications:** Open your code editor and implement the changes suggested by the reviewers. Save the files.
2. **Stage and Commit the new changes:** Just like before, stage and commit your new modifications. You can create multiple commits on the same branch.

```
git add . # Or git add specific_file.py
git commit -m "Refactor: Add requested changes for better readability"
```

```
It's good practice to write a commit message that reflects the feedback you're
addressing.
```

1. **Push the updated branch:** Push your local branch to your fork again. Since you're pushing to the same branch you already pushed, Git will simply update it.

```
    git push origin fix-typo-in-readme
```

```
GitHub will automatically detect these new commits on your branch and add them
to your existing pull request. The PR view will update to show the new commits
and the updated "Files changed."
```

1. **Respond to comments:** Once you've made the changes, go back to your PR on GitHub and respond to the maintainers' comments. You can mark discussions as "resolved" if you believe you've addressed the feedback.

Patience and Politeness

The review process can sometimes take time, especially for larger projects or during busy periods.

- **Be patient:** Avoid repeatedly asking for updates.
- **Be polite:** Always interact respectfully. Open source thrives on positive collaboration.
- **Ask questions:** If you don't understand a piece of feedback, don't hesitate to ask for clarification.

Once the maintainers are satisfied with your changes, they will merge your pull request into the main project. Congratulations, you've made your first open-source contribution!

What you've accomplished: You understand the pull request review process, how to address feedback, and how to update your contribution.

Common Mistakes and Next Steps

Congratulations on making your first open-source contribution! It's a significant milestone. As you continue your open-source journey, being aware of common pitfalls can help you avoid frustration and contribute more effectively.

Common Mistakes to Avoid

1. **Not Reading Contribution Guidelines:** Every project is different. Always check for `CONTRIBUTING.md` or similar documentation. It outlines coding standards, commit message formats, testing procedures, and the preferred PR process. Ignoring these can lead to unnecessary rework.

2. **Large, Unfocused Pull Requests:** Trying to fix many things at once (e.g., a bug fix, a new feature, and a refactor) in a single PR makes it hard for reviewers. Keep PRs small, focused, and addressing one specific issue.
3. **Vague Commit Messages or PR Descriptions:** "Fixes bug" isn't helpful. "Fix: Prevent crash when input is empty by adding validation check" is much better. Explain what you did and why.
4. **Not Syncing Your Fork Regularly:** If you work on a branch for a long time without pulling updates from the original project's `main` branch, your fork can fall behind. This can lead to merge conflicts when you open your PR. Remember `git pull upstream main` on your local `main` branch before creating new feature branches.
5. **Opening a PR from Your `main` Branch:** As mentioned, always create a dedicated branch for each contribution. This keeps your `main` clean and prevents unrelated changes from being bundled into a single PR.
6. **Ignoring Automated Checks:** If CI/CD tests fail on your PR, don't ignore them. These failures usually indicate a problem with your code that needs to be fixed before it can be merged.
7. **Becoming Discouraged by Feedback:** Code review is about improving the code, not criticizing the person. Embrace feedback as a learning opportunity.

What to Build Next

Now that you've successfully navigated the open-source contribution workflow, here are some ideas to continue your journey and deepen your skills:

1. **Tackle a Slightly More Complex "Good First Issue":** Look for another `good first issue` in the same project or a different one, perhaps one that involves a small code change rather than just documentation. This will help you understand the project's codebase a bit more.
2. **Improve an Existing Feature or Documentation:** Think about a project you use. Is there a part of its documentation that could be clearer? Is there a small UI tweak or a minor bug you've noticed? These "quality of life" improvements are often highly valued.
3. **Explore a New Project or Technology:** Branch out and find a project built with a technology you want to learn. Contributing is a fantastic way to get hands-on experience with new languages, frameworks, or tools in a real-world context.

Every contribution, no matter how small, makes a difference. Keep learning, keep collaborating, and enjoy being a part of the open-source community!