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

2026-05-25 20:31 IST

Contents

CHAPTER 01

Stop GitHub Bot Spam with Git --author

What you'll build: You will learn to configure Git and implement CI/CD validation to prevent AI bot spam in GitHub repositories by enforcing correct commit author information. **Time needed:** ~60 minutes **Prerequisites:** Git installed, Basic Git CLI knowledge, GitHub account, Familiarity with CI/CD concepts (e.g., GitHub Actions) **Version used:** unknown

Understanding the AI Bot Spam Problem

Maintaining a healthy open-source project or even a private team repository on GitHub can be challenging. In recent times, a new breed of problem has emerged: AI bot spam. These bots often generate low-quality, irrelevant, or even malicious content, disguised as legitimate contributions. They might open numerous issues, submit nonsensical pull requests, or push commits with generic or fake author information.

Why this matters: This bot activity clutters your project, wastes maintainer time reviewing garbage, and can dilute the perception of your project's quality. It makes it harder to identify genuine contributions and can even deter real contributors. One common characteristic of these bot-generated commits is their often inconsistent or generic author information, which we can leverage to identify and prevent them.

Our goal is to implement a robust defense mechanism that ensures every commit to your repository has a verifiable and correct author identity, effectively trapping and blocking these automated nuisances before they become a problem.

Git Author Identity Fundamentals

Before we dive into blocking bot spam, let's ensure we understand how Git identifies authors. Every commit in Git has two important pieces of identity information: the **author** and the **committer**.

- **Author:** The person who originally wrote the work.
- **Committer:** The person who last applied the work (e.g., merged a pull request, or created the commit on behalf of the author).

For most of your daily work, the author and committer will be the same person. Git automatically picks up your author identity from your global or local configuration.

Let's start by checking your current global Git configuration for `user.name` and `user.email`. These are the default values Git uses when you make a commit.

First, open your terminal and check your global Git configuration:

```
git config --global user.name
git config --global user.email
```

You should see your name and email address printed. If you haven't set these up before, or if they are incorrect, you'll need to configure them. It's crucial that your `user.email` matches an email associated with your GitHub account, ideally your primary verified email.

To set or update your global `user.name` and `user.email`, use these commands. Replace `"Your Name"` and `"your.email@example.com"` with your actual name and email.

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

> ⚡ Note: The `--global` flag sets these values for all your Git repositories on your machine. You can also set them locally for a specific repository by omitting `--global` and running the command inside that repository. Local configurations override global ones.

After setting your configuration, let's verify that the changes have been applied correctly.

```
git config --list | grep user
```

You should now see your updated `user.name` and `user.email` listed. This ensures that any new commits you make will correctly attribute you as the author.

What we accomplished: We've reviewed the fundamental concepts of Git author identity and ensured your global Git configuration is correctly set up with your name and email. This forms the basis for proper commit attribution.

Using `--author` for Manual Commits

Sometimes, you might need to make a commit on behalf of someone else, or you might be working in an environment where your default Git configuration isn't the one you want to use for a specific commit. This is where the `git commit --author` flag comes in handy. It allows you to explicitly specify the author for a single commit, overriding your default Git configuration.

Why this exists: While it's typically used for legitimate reasons (e.g., applying a patch from someone who doesn't have direct commit access, or correcting an accidental misattribution), we can also use this flag to simulate how bots might bypass simple checks, or how we might correct a human error. Understanding its explicit usage is key to then enforcing its correct application.

Let's create a new temporary Git repository to experiment with this.

First, create a new directory, initialize a Git repository, and create a simple file:

```
mkdir bot-spam-test
cd bot-spam-test
git init
echo "Initial content." > README.md
git add README.md
```

Now, let's make a commit, but this time, we'll explicitly set the author using the `--author` flag. We'll use a generic "Bot User" email to simulate a potential bot commit.

```
git commit --author="Bot User <bot@example.com>" -m "Simulating a bot commit"
```

In this command:

- `--author="Bot User <bot@example.com>"` specifies the author's name and email address for this particular commit.
- `-m "Simulating a bot commit"` provides the commit message.

After running the command, let's verify the commit's author. We can use `git log` to see the commit history and inspect the author information.

```
git log --format=fuller -1
```

You should see output similar to this (dates and hashes will differ):

```
commit 1234567890abcdef1234567890abcdef1234567890
Author:     Bot User <bot@example.com>
AuthorDate: Thu May 25 10:00:00 2026 -0700
Commit:     Your Name <your.email@example.com>
CommitDate: Thu May 25 10:00:00 2026 -0700

    Simulating a bot commit
```

> ⚡ Note: Notice that the `Author:` field shows "Bot User bot@example.com", but the `Commit:` field still shows your globally configured `user.name` and `user.email`. This is because you are the one who executed the `git commit` command, making you the committer, even though you specified a different author for the content. This distinction is important for later validation.

Now, let's make a second commit using your default Git configuration, without the `--author` flag. This will represent a legitimate commit from you.

```
echo "Adding more content." >> README.md
git add README.md
git commit -m "A legitimate commit from me"
```

Verify this second commit's author:

```
git log --format=fuller -1
```

This time, both `Author:` and `Commit:` should show your configured name and email.

What we accomplished: You've learned how to explicitly set the author for a single commit using `git commit --author`, and observed the difference between the author and committer identities in Git history. This hands-on experience is crucial for understanding how author information can be manipulated and, consequently, how it can be validated.

Correcting Existing Commit Authors (History Rewriting)

Mistakes happen. Sometimes a commit is made with the wrong author information, perhaps due to a misconfigured Git client or a simple oversight. While we want to prevent future bot spam, it's also useful to know how to correct past mistakes. This involves rewriting Git history, which is a powerful operation and comes with significant caveats.

Why this exists: Correcting author information in existing commits ensures the integrity and accuracy of your project's history. For instance, if a new contributor accidentally uses a generic email, you might want to fix it for proper attribution and GitHub's contribution graph.

We'll cover how to change the author of the last commit using `git commit --amend`. For older commits or multiple commits, we'll discuss the tools, but emphasize the risks.

Let's assume you just made a commit, and realized the author email was wrong. We can amend it. First, make another commit with an intentionally incorrect author email, just to demonstrate correcting it.

```
echo "Some new feature." >> feature.txt
git add feature.txt
git commit --author="Accidental User <wrong@email.com>" -m
"Oops, wrong author email"
```

Verify the last commit's author:

```
git log --format=fuller -1
```

You should see "Accidental User wrong@email.com" as the author.

Now, let's correct it using `git commit --amend --author`. Replace `"Your Correct Name"` and `"your.correct.email@example.com"` with your actual correct details.

```
git commit --amend --no-edit --author="Your Correct Name
<your.correct.email@example.com>"
```

Here's what these flags mean:

- `--amend`: This command doesn't create a new commit; instead, it modifies the last commit.
- `--no-edit`: This keeps the original commit message. If you want to change the message as well, omit this flag, and Git will open your configured editor.
- `--author="Your Correct Name <your.correct.email@example.com>"`: This explicitly sets the new author for the amended commit.

After running the command, let's verify the last commit again.

```
git log --format=fuller -1
```

You should now see "Your Correct Name your.correct.email@example.com" as both the author and committer. The commit hash will have changed because you've rewritten history.

> ⚠️ Common mistake: Rewriting history, especially on commits that have already been pushed to a shared remote repository, is a **dangerous operation**. If others have based their work on the old history, force-pushing your rewritten history (`git push --force-with-lease` or `git push -f`) will cause conflicts and headaches for them. **Only amend or rewrite history on commits you haven't pushed yet, or if you are absolutely sure you understand the implications and have coordinated with your team.**

For correcting multiple older commits or an entire repository's history, tools like `git rebase -i` (interactive rebase) or `git filter-repo` are used. These are more advanced topics and outside the scope of a basic `--author` tutorial, but it's important to know they exist and involve significant risk. For instance, to change an author in an interactive rebase, you would mark commits with `edit` and then `git commit --amend --author="..."` for each.

What we accomplished: You've learned how to correct the author of the last commit using `git commit --amend --author`. We also discussed the critical warnings associated with rewriting Git history, especially in shared repositories.

Implementing CI/CD Author Validation

Now that we understand how Git author identity works and how it can be explicitly set, we can implement automated validation. The most effective place to do this is in your Continuous Integration/Continuous Deployment (CI/CD) pipeline. By adding an author validation step to your CI/CD workflow, you can automatically reject commits or pull requests that don't meet your author identity standards, effectively stopping bot spam before it pollutes your repository.

Why this exists: Automated validation provides a consistent and immediate feedback loop. It prevents human error and ensures that all contributions adhere to your project's rules without manual intervention. For bots, it creates a barrier they cannot easily bypass.

We'll use GitHub Actions for this, as it's tightly integrated with GitHub repositories and provides a straightforward way to run custom validation scripts. Our strategy will be to check the commit author's email against a pattern or a whitelist.

First, you'll need a GitHub repository. If you've been following along, push your `bot-spam-test` repository to GitHub.

```
# Assuming you have a remote named 'origin'
git remote add origin https://github.com/YOUR_USERNAME/bot-spam-test.git
git branch -M main
git push -u origin main
```

Now, let's create a GitHub Actions workflow. This workflow will run on every `push` event and check the author of the pushed commits.

Create a directory `.github/workflows` in your repository root, and then create a file named `validate-author.yml` inside it.

```
mkdir -p .github/workflows
touch .github/workflows/validate-author.yml
```

Open `.github/workflows/validate-author.yml` in your editor and add the following content:

```
name: Author Validation

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  validate_commit_authors:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v4
        with:
          fetch-depth: 0 # Needed to fetch all history for author checks

      - name: Get commit authors
        id: get_authors
        run: |
          # Get the list of commit authors from the push/PR event
          # For pushes: compare current branch head to previous head
          # For PRs: compare PR branch head to base branch head
          if [ "${{ github.event_name }}" == "push" ]; then
            COMMITS=$(git log --pretty=format:'%an <%ae>' $
{{ github.event.before }}..${{ github.sha }})
          elif [ "${{ github.event_name }}" == "pull_request" ]; then
            COMMITS=$(git log --pretty=format:'%an <%ae>' $
{{ github.event.pull_request.base.sha }}..${{ github.sha }})
          else
            echo "Unsupported event type: ${{ github.event_name }}"
```

```
            exit 1
          fi

          echo "Commits to validate:"
          echo "$COMMITS"
          echo "COMMITS_TO_VALIDATE<<EOF" >> $GITHUB_OUTPUT
          echo "$COMMITS" >> $GITHUB_OUTPUT
          echo "EOF" >> $GITHUB_OUTPUT

      - name: Validate authors
        run: |
          AUTHORS="${{ steps.get_authors.outputs.COMMITS_TO_VALIDATE }}"
          VALID_EMAIL_REGEX="^[a-zA-Z0-9._%+-]+@([a-zA-Z0-9-]+\.)+[a-zA-Z]
{2,}$" # Basic email regex
          ALLOWED_DOMAINS=("example.com" "github.com") # Whitelist specific
domains

          echo "Starting author validation..."
          echo "$AUTHORS" | while IFS= read -r author_line; do
            if [ -z "$author_line" ]; then
              continue # Skip empty lines
            fi

            AUTHOR_EMAIL=$(echo "$author_line" | grep -oP '(?<=<)[^>]+(?=>)')
            AUTHOR_NAME=$(echo "$author_line" | grep -oP '^[^<]+(?=\s<)')

            echo "Validating author: $AUTHOR_NAME, Email: $AUTHOR_EMAIL"

            # Check if email matches a basic regex pattern
            if ! [[ "$AUTHOR_EMAIL" =~ $VALID_EMAIL_REGEX ]]; then
              echo "❌ Error: Author email '$AUTHOR_EMAIL' does not appear to
be a valid email format."
              exit 1
            fi

            # Check if email domain is in the allowed list
            EMAIL_DOMAIN=$(echo "$AUTHOR_EMAIL" | awk -F@ '{print $2}')
            IS_ALLOWED_DOMAIN=false
            for domain in "${ALLOWED_DOMAINS[@]}"; do
              if [ "$EMAIL_DOMAIN" == "$domain" ]; then
                IS_ALLOWED_DOMAIN=true
                break
              fi
            done

            if ! $IS_ALLOWED_DOMAIN; then
              echo "❌ Error: Author email domain '$EMAIL_DOMAIN' is not in
the allowed list."
              exit 1
            fi

            echo "✅ Author '$AUTHOR_NAME <$AUTHOR_EMAIL>' is valid."
          done

          echo "All commit authors validated successfully."
```

Let's break down this GitHub Actions workflow:

- **`on: push: branches: main` and `on: pull_request: branches: main`**: This configures the workflow to run whenever a commit is pushed to the `main` branch or when a pull request targeting `main` is opened or updated.
- **`actions/checkout@v4`**: This action checks out your repository's code. `fetch-depth: 0` is crucial because it fetches the entire history, allowing `git log` to correctly compare against previous commits or PR base branches.
- **`Get commit authors` step**: This script identifies the commits that are part of the current push or pull request.
    - For a `push` event, it compares the current `HEAD` (`${{ github.sha }}`) with the commit before the push (`${{ github.event.before }}`).
    - For a `pull_request` event, it compares the PR's head commit (`${{ github.sha }}`) with the base branch's head (`${{ github.event.pull_request.base.sha }}`).
    - It then extracts the author name and email for each relevant commit using `git log --pretty=format:'%an <%ae>'`.
    - The output `COMMITS_TO_VALIDATE` is then passed to the next step.
- **`Validate authors` step**: This is where the core logic resides.
    - It iterates through each author line extracted from the previous step.
    - It extracts the `AUTHOR_EMAIL` and `AUTHOR_NAME`.
    - It checks if the `AUTHOR_EMAIL` matches a basic regular expression (`VALID_EMAIL_REGEX`) for email format.
    - It then checks if the email's domain (`EMAIL_DOMAIN`) is present in a hardcoded `ALLOWED_DOMAINS` list.
    - If any validation fails, the script `exit 1`, which causes the GitHub Action job to fail, preventing the merge or indicating a problem with the pushed commits.

> ⚡ Real-world insight: In a production environment, `ALLOWED_DOMAINS` might be dynamically loaded from a configuration file or environment variables. You might also integrate with an identity provider to verify if the email belongs to an active team member. For open-source projects, you might primarily rely on a robust email regex and ensure the email is not a known spam pattern.

What we accomplished: You've set up a GitHub Actions workflow that automatically validates the author email of every commit pushed to or part of a pull request targeting your `main` branch. This provides a powerful, automated defense against bot spam.

Practical Examples and Testing

Now that our CI/CD author validation is in place, let's test it with both a "bad" commit (simulating a bot) and a "good" commit (a legitimate contribution). This will demonstrate how the workflow catches invalid authors and allows valid ones.

First, ensure your `validate-author.yml` workflow file is added and committed to your repository.

```
git add .github/workflows/validate-author.yml
git commit -m "Add GitHub Actions workflow for author validation"
git push origin main
```

Wait for this initial push to complete and the GitHub Action to run. It should pass, as your commit author is legitimate. You can check the "Actions" tab on your GitHub repository page.

Test 1: Simulating a Bot Spam Commit (Expected to Fail)

Let's create a commit with an email address that does not match our `ALLOWED_DOMAINS` list (`example.com` and `github.com`). We'll use `spam@malicious.xyz`.

```
echo "Bot-generated content." >> bot-file.txt
git add bot-file.txt
git commit --author="Spam Bot <spam@malicious.xyz>" -m "Adding spam content"
```

Now, push this commit to your `main` branch.

```
git push origin main
```

After pushing, navigate to the "Actions" tab in your GitHub repository. You should see a workflow run in progress or recently completed. Click on it.

You will see that the `validate_commit_authors` job should **fail**. Expand the "Validate authors" step, and you should see an error message similar to:

```
❌ Error: Author email domain 'malicious.xyz' is not in the allowed list.
```

This demonstrates that your CI/CD pipeline successfully identified and rejected a commit with an unauthorized author email domain.

Test 2: Making a Legitimate Commit (Expected to Pass)

Now, let's make a legitimate commit using your correctly configured Git identity (which should use an `example.com` or `github.com` email if you followed the setup).

```
echo "Legitimate contribution." >> contributor-file.txt
git add contributor-file.txt
git commit -m "Adding a legitimate contribution"
```

Push this commit to your `main` branch.

```
git push origin main
```

Go back to the "Actions" tab on GitHub. You should see a new workflow run. This time, the `validate_commit_authors` job should **pass**. Expand the "Validate authors" step, and you should see:

```
✅ Author 'Your Name <your.email@example.com>' is valid.
All commit authors validated successfully.
```

This confirms that your CI/CD validation allows legitimate contributions through while effectively blocking bot spam.

What we accomplished: You've successfully tested your CI/CD author validation workflow by simulating both a bot-generated commit (which failed validation) and a legitimate commit (which passed). This practical demonstration confirms your defense mechanism is working as intended.

Common Pitfalls and Best Practices

Implementing author validation is a powerful step, but it's important to be aware of common pitfalls and follow best practices to ensure a smooth workflow for legitimate contributors while effectively blocking spam.

Common Pitfalls:

- **Forgetting Global/Local Git Config:** If a developer's `user.name` and `user.email` are not correctly set, their legitimate commits might fail your CI/CD validation. This is a common source of frustration for new contributors.
    - `> ⚠️ Common mistake:` Your CI fails, but you don't understand why, only to find out your team member's Git config was generic like `user@hostname`.
- **Overly Strict or Inflexible Validation Rules:** If your `ALLOWED_DOMAINS` list is too narrow, or your regex is too strict, you might inadvertently block legitimate contributions from people using personal email addresses or corporate emails not on your initial whitelist.
    - `> ⚠️ Common mistake:` Blocking a valuable contribution because the author used `gmail.com` and it wasn't on your whitelist, and they didn't realize it.
- **Rewriting Shared History:** As discussed, using `git commit --amend --author` or `git rebase -i` on commits that have already been pushed to a shared branch can cause significant problems for collaborators.
    - `> ⚠️ Common mistake:` Force-pushing a rewritten branch, causing others to have to `git pull --rebase` or reset their local branches.
- **Ignoring Legitimate Bot Accounts:** Many projects use automated tools like Dependabot, Renovate, or other custom bots for dependency updates, linting, or documentation generation. These bots will also make commits, and their author emails (e.g., `dependabot[bot]@users.noreply.github.com`) must be explicitly accounted for in your validation.

Best Practices:

- **Clear Contribution Guidelines:** Document your author validation requirements clearly in your `CONTRIBUTING.md` file. Explain why it's in place and what contributors need to do to ensure their commits pass. This includes instructions on setting `user.name` and `user.email` correctly.
- **Educate Your Team/Contributors:** Proactively communicate these changes to your team members and regular contributors. Provide them with the `git config` commands they need.

- **Whitelisting Legitimate Bot Emails:** Update your CI/CD validation script to explicitly allow the email addresses of any legitimate bots you use. GitHub's official bot emails often end with `[bot]@users.noreply.github.com`.
    - `> 🔥 Optimization / Pro tip:` Create a separate list or regex for known bot emails that you trust.
- **Use Descriptive Error Messages:** Make sure your CI/CD pipeline's failure messages are clear and actionable. Instead of just "Validation failed," provide specific details like "Author email domain 'xyz.com' is not allowed. Please use an email from example.com or github.com."
- **Consider a Grace Period/Warning:** For new projects or teams, you might start with a warning in CI for invalid authors before enforcing a hard failure. This allows contributors to adapt without immediately blocking their work.
- **Test Your Validation Thoroughly:** Just as we did in the previous section, test your validation with various valid and invalid author scenarios to ensure it behaves as expected.

By being mindful of these pitfalls and adopting these best practices, you can create a robust and developer-friendly author validation system that effectively combats bot spam without hindering legitimate contributions.

What we accomplished: We've identified common issues that can arise when implementing author validation and outlined best practices to ensure a smooth and effective process for both maintainers and contributors.

Maintaining Code Quality and Contributions

Implementing author validation is more than just blocking spam; it's a foundational step towards maintaining high code quality and fostering a healthy contribution environment. When every commit is correctly attributed to a known and legitimate source, several positive outcomes emerge:

- **Clearer Accountability:** With accurate author information, it's easy to track who contributed what, which is vital for code reviews, debugging, and understanding the history of changes. This transparency discourages anonymous or low-quality contributions.
- **Improved Code Ownership:** Proper attribution reinforces a sense of ownership among contributors, encouraging them to maintain the quality of their code.

- **Accurate GitHub Contribution Graphs:** For open-source projects, a clean commit history with correct author emails ensures that contributors receive proper credit on their GitHub profiles, which is a significant motivator.
- **Reduced Noise:** By filtering out bot spam, maintainers can focus their attention on genuine issues, valuable pull requests, and meaningful discussions, significantly improving productivity and reducing burnout.
- **Enhanced Security:** While not a primary security measure, enforcing known author identities can make it harder for malicious actors to inject code anonymously or disguise their commits as legitimate.

The `git commit --author` flag, when used responsibly and coupled with strong CI/CD validation, transforms from a simple Git command into a powerful tool for project governance. It empowers you to define and enforce the identity standards for your repository, ensuring that every line of code can be traced back to a verifiable source.

This approach strikes a balance: it leverages Git's inherent capabilities for attribution, integrates seamlessly with modern CI/CD practices, and ultimately protects your project from the growing threat of automated spam, allowing your community to thrive on quality contributions.

What to Build Next

You've successfully implemented a system to combat GitHub bot spam. Here are three concrete ideas to extend your project and further enhance your repository's security and contribution experience:

1. **Implement a Whitelist/Blacklist Management System:** Instead of hardcoding `ALLOWED_DOMAINS` in your GitHub Action, create a separate configuration file (e.g., `allowed_authors.json` or a `.yml` file) in your repository. Update your GitHub Action script to read from this file. This makes it easier to manage allowed domains and specific author emails without modifying the workflow YAML directly. You could even implement a "blacklist" for known spam patterns or generic names.
2. **Integrate with a Contributor License Agreement (CLA) Bot:** For open-source projects, ensuring contributors sign a CLA is crucial. You could extend your CI/CD pipeline to check if the commit author's email is associated with a signed CLA. If not, the CI could fail and provide instructions on how to sign the CLA, ensuring legal compliance alongside author verification.

3. **Add Automated Issue/PR Closing for Spam:** While your current setup blocks spam at the commit/PR merge stage, bots might still open issues or pull requests with invalid authors. Enhance your GitHub Actions with a separate workflow that triggers on `issues.opened` or `pull_request.opened` events. If the issue/PR creator's email (obtainable via GitHub API) or username doesn't meet your criteria, automatically close the issue/PR and add a comment explaining why, directing them to your contribution guidelines.