Click to prove you're human

Github best practices

AI AI AI AI AI AI AI AI You can’t perform that action at this time. We interviewed hundreds of software developers, and perfomed code scanning on thousands of GitHub repositories using our own product to produce this list.These best practices are still applicable even if you use something other than GitHub for version control or source control, because they’re all about ensuring source code security and writing good code.# 10 — Don’t just git commit directly to masterRegardless if you use Gitflow or any other git branching model, it is always a good idea to turn on git branch protection to prevent direct commits and ensure your main branch code is deployable at all times. All commits should be managed via pull requests.# 9 — Do git commit with the right email addressSometimes you commit code using the wrong email address, and as a result GitHub shows that your commit has an unrecognized author. Having commits with unrecognized authors makes it more difficult to track who wrote which part of the code.Ensure your Git client is configured with the correct email address and linked to your GitHub user. Check your pull requests during code reviews for unrecognized commits.# 8 — Define code owners for faster code reviewsWhen you’re dealing with dozens, hundreds, or more repositories and engineers, it’s nearly impossible to know who owns which parts of the codebase and need to review your changes.Even in smaller teams you’d still have code owners — for example, front-end code changes should be reviewed by the Front-End Engineer.Use Code Owners feature to define which teams and people are automatically selected as reviewers for the repository.# 7 — Don’t let secrets leak into source codeSecrets, or secret keys or secret credentials, include things like account passwords, API keys, private tokens, and SSH keys. You should not check them into your source code.Instead, we recommend you inject secrets as environment variables externally from a secure store. You can use tools like Hashicorp Vault or AWS Secrets Manager to do this.There are also tools for scanning secrets in repos and prevent them from getting into repos.Git-secrets can help you to identify passwords in your code.Git hooks can be used to build a pre-commit hook and check every pull request for secrets.Datree has a built-in policy rule for this.Read this tutorial or watch this video for a more detailed explanation on why you should manage secrets this way and how to do it right.# 6 — Don’t commit dependencies into source codePushing dependencies into your remote origin will increase repository size. Remove any projects dependencies included in your repositories and let your package manager download them in each build. if you are afraid of “dependencies availability” you should consider using a binary repository manager solution like Jfrog or Nexus Repository. Or check out GitHub’s Git-Sizer.# 5 — Don’t commit configuration files into source codeWe strongly recommend against committing your local config files to version control. Usually, those are private configuration files you don’t want to push to remote because they are holding secrets, personal preferences, history or general information that should stay only in your local environment.# 4 — Create a meaningful git ignore fileA .gitignore file is a must in each repository to ignore predefined files and directories. It will help you to prevent secret keys, dependencies and many other possible discrepancies in your code. You can choose a relevant template from Gitignore.io to get started quickly.# 3 — Archive dead repositoriesOver time, for various reasons, we find ourselves with unmaintained repositories. Sometimes developers create repos for an ad hoc use case, a POC, or some other reason. Sometimes they inherit repos with old and irrelevant code.In any case, these repos were left intact. No one is doing any development work in those repos anymore, so you want to clean them up and avoid the risk of other people using them. The best practice is to archive them, i.e. make them “read-only” to everyone.# 2 — Lock package versionYour manifest file contains information about all packages and dependencies in your project and their versions. The best practice is to specify a version or version range for every package and dependency listed in the manifest. Otherwise, you can’t be sure which version will get installed during the next build, and consequently your code may break.# 1 — Align packages versioningEven when everyone on your team are using the same packages, reusing code and tests across different projects can still be difficult if the packages are of different versions.If you have a package that is used in multiple projects, try at a minimum to use the same major version of the package.What’s next?All that’s left for you to do is check off each of the aforementioned best practices, on each of your repositories, one by one…Or, save your sanity and connect with Datree’s GitHub app to scan your repositories and generate your free status report to assess if your repositories align with the listed best practices. A demonstration animation of a code editor using GitHub Copilot Chat, where the user requests GitHub Copilot to refactor duplicated logic and extract it into a reusable function for a given code snippet.Build code quickly and more securely with GitHub Copilot embedded throughout your workflows.With GitHub Copilot embedded throughout the platform, you can simplify your toolchain, automate tasks, and improve the developer experience.A Copilot chat window with extensions enabled. The user inputs the @ symbol to reveal a list of five Copilot Extensions. @Sentry is selected from the list, which shifts the window to a chat directly with that extension. There are three sample prompts at the bottom of the chat window, allowing the user to Get incident information, Edit status on incident, or List the latest issues. The last one is activated to send the prompt: @Sentry List the latest issues. The extension then lists several new issues and their metadata.Optimize your process with simple and secured CI/CD.A list of workflows displays a heading ‘45,167 workflow runs’ at the top. Below are five rows of completed workflows accompanied by their completion time and their duration formatted in minutes and seconds.Discover GitHub ActionsStart building instantly with a comprehensive dev environment in the cloud.A GitHub Codespaces setup for the landing page of a game called OctoInvaders. On the left is a code editor with some HTML and Javascript files open. On the right is a live render of the page. In front of this split editor window is a screenshot of two active GitHub Codespaces environments with their branch names and a button to ‘Create codespace on main.’Check out GitHub CodespacesManage projects and chat with GitHub Copilot from anywhere.Two smartphone screens side by side. The left screen shows a Notification inbox, listing issues and pull requests from different repositories like TensorFlow and GitHub’s OctoArcade octoinvaders. The right screen shows a new conversation in GitHub Copilot chat.Download GitHub MobileSync with 17,000+ integrations and a growing library of Copilot Extensions.A grid of fifty app tiles displays logos for integrations and extensions for companies like Stripe, Slack, and Docker. The tiles extend beyond the bounds of the image to indicate a wide array of apps. Visit GitHub MarketplaceCollaborate with your teams, use management tools that sync with your projects, and code from anywhere—all on a single, integrated platform.It helps us onboard new software engineers and get them productive right away. We have all our source code, issues, and pull requests in one place... GitHub is a complete platform that frees us from menial tasks and enables us to do our best work.Create issues and manage projects with tools that adapt to your code.Display of task tracking within an issue, showing the status of related sub-issues and their connection to the main issue.Explore GitHub IssuesCreate space for open-ended conversations alongside your project.A GitHub Discussions thread where a GitHub user suggests a power-up idea involving Hubot revealing a path and protecting Mona. The post has received 5 upvotes and several reactions. Below, three other users add to the discussion, suggesting Hubot could provide different power-ups depending on levels and appreciating the collaboration idea.Discover GitHub DiscussionsCreate review processes that improve code quality and fit neatly into your workflow.Two code review approvals by helios-ackmore and amanda-knox, which are followed by three successful checks for ‘Build,’ ‘Test,’ and ‘Publish.’Learn about code reviewBecome an open source partner and support the tools and libraries that power your work.A GitHub Sponsors popup displays ‘$15,000 a month’ with a progress bar showing 87% towards a $15,000 goal.Dive into GitHub SponsorsWhether you’re scaling your development process or just learning how to code, GitHub is where you belong. Join the world’s most widely adopted AI-powered developer platform to build the technologies that redefine what’s possible. You can’t perform that action at this time. As a DevOps engineer, managing GitHub repositories is as crucial as the code they contain. A well-maintained Github repo sets the stage for effective collaboration, code quality, and streamlined workflows. In this blog, we'll discuss and look at my top 10 tips for best practices in managing GitHub repositories effectively. Tip 1: Use a Clear Repository Naming Convention A clear repository naming convention in GitHub is a vital as it helps with organisation and clarity, which are crucial in a collaborative environment. A clear repository naming convention makes it easier to: Identify the purpose and content of a repository at a glance. Search and retrieve repositories more effectively. Adopt a standardised approach across teams and projects. Implement automation to work more effectively by predicting the structure and naming of repositories. For example, CI/CD workflows can deploy versions based on naming conventions. Lets look at some examples: Prefix by Project or Team: If your organisation has several projects or teams, you could start with a prefix that identifies them e.g. teamalpha_authentication_service or teambravo_data_pipeline. Use Descriptive Names: Repositories should have descriptive and specific names that tell you what's inside e.g. customer_support_ticketing_system or machine_learning_model_trainer. Include the Technology Stack: It can be useful, particularly for microservices architectures, to include the primary technology stack in the name e.g. image_processor_python or frontend_react_app. Versioning or Status Tags: If you maintain different versions of a tool or library, or when a repository holds something at a specific stage of development, indicate this within the name e.g. payment_gateway_v2 or inventory_management_deprecated. Avoid Special Characters: Stick to simple alphanumeric characters and hyphens/underscores to maintain URL compatibility and avoid confusion e.g. invoice-generator or invoice_generator. Use Case: Sometimes indicating whether the repository is a library, service, demo, or documentation can be helpful e.g. authentication_lib, payment_api_service, demo_inventory_app, api_documentation. By adhering to a clear and standardised repository naming convention, you ensure that everyone on the team can navigate repositories more efficiently, anticipate the nature and content of each repository before delving into it, and work cohesively with an intuitive structure guiding them. This ultimately leads to better collaboration, time-saving, and fewer mistakes, allowing teams to focus on building and deploying rather than being bogged down with organisational confusion. Tip 2: Classify Repositories with Topics GitHub allows you to classify repositories with topics. Topics are labels that can be added to repositories to help categorise and discover projects. They are a great way to organise and group repositories based on their purpose, technology stack, or any other relevant criteria. Topics can be added to a repository by navigating to the repository's About settings to edit repository details and selecting the Topics tab. You can then add topics that are relevant to the repository. It is useful to add topics to repositories for several reasons, including: Discoverability: Make it easier for others to find your repository. When someone searches for a topic, repositories with that topic will be included in the search results. Organisation: Help you organise your repositories. You can group repositories based on their purpose, technology stack, or any other relevant criteria. Community: Help you connect with others who are interested in the same topics. When someone views a repository with a topic, they can see other repositories with the same topic. Insights: Provide insights into the technologies and tools that are popular in your organisation. You can use this information to identify trends and make informed decisions about the technologies and tools you use. Standardisation: Help you standardise the way you categorise repositories. You can use the same topics across all your repositories to ensure consistency. When adding topics to a repository, it's important to choose topics that are relevant and meaningful. You should choose topics that accurately describe the purpose, technology stack, or other relevant criteria of the repository. You can get more information on topics and how to use them effectively from GitHub repo topics documentation. Tip 3: Use README.md to Document the Repository A well-documented repository is a treasure trove for developers, contributors, and maintainers. The README.md file is the first thing a visitor sees when they land on your repository. It's a great place to provide a quick overview of the repository, its purpose, and how to get started with it. It could include useful information such as: Project description Setup instructions Usage examples Contribution guidelines License information A well-written README.md file can help you: Attract Contributors: Attract contributors to your project. It provides them with the information they need to understand the project and get started with it. Onboarding: Help new team members get up to speed with the project. It provides them with the information they need to understand the project and start contributing to it. Documentation: Serve as documentation for the project. It provides users with the information they need to use the project. Promotion: Help promote your project. It provides potential users with the information they need to understand the project and decide whether to use it. Standardisation: Help standardise the way you document your projects. It provides a consistent structure for documenting your projects. When writing a README.md file, it's important to keep it concise and to the point. You should include the most important information at the top of the file, and provide links to more detailed information where necessary. You should also use formatting to make the file easy to read, and include images and other media where appropriate. You can get more information on how to write a good README.md file in the GitHub repo readme documentation. Tip 4: Embrace a consistent branching strategy A consistent branching strategy is crucial for effective collaboration and code management. It provides a clear structure for how code changes are managed and integrated into the codebase. It also helps to maintain a clean and stable codebase, and reduces the risk of conflicts and errors. There are several branching strategies that you can adopt, such as: Gitflow: A popular branching strategy that uses two main branches, master and develop, and a variety of supporting branches to aid parallel development and release management. Feature Branching: A strategy where each feature or task is developed in a dedicated branch, and then merged into the main branch once complete. Trunk-Based Development: A strategy where all changes are made directly to the main branch, and feature toggles or other techniques are used to manage incomplete features. GitHub Flow: A lightweight branching strategy that uses a single main branch, and feature branches are created for each new feature or bug fix. GitLab Flow: A strategy similar to GitHub Flow, but with the addition of environments and release branches for managing the release process. Release Branching: A strategy where a release branch is created from the main branch to prepare for a new release, and then merged back into the main branch once the release is complete. Environment Branching: A strategy where branches are used to manage different environments, such as development, staging and production. When choosing a branching strategy, it's important to consider the needs of your team and project. You should choose a strategy that is simple, flexible, and scalable, and that supports the way your team works. You should also document the branching strategy and make sure that everyone on the team understands how it works and follows it consistently. You can get more information on branching and how to use branches by checking the official documentation: GitHub repo branch documentation. Tip 5: secure your repository with branch protection rules Branch protection rules are a powerful feature of GitHub that allow you to enforce certain restrictions and requirements on branches. They can help you maintain a clean and stable codebase. They can also help you prevent mistakes and errors, and improve the quality and security of your code. To name a few, you can use branch protection rules to: Require pull request reviews: Require that a certain number of reviewers approve a pull request before it can be merged. Require status checks: Require that certain status checks, such as CI/CD checks, pass before a pull request can be merged. Require conversation resolution before merging: Require that all conversations on a pull request are resolved before it can be merged. Require signed commits: Require that commits are signed with a verified signature before they can be merged. Require linear history: Require that the commit history of a pull request is linear before it can be merged. Require merge queue: Require that pull requests are merged using a merge queue, such as GitHub Actions or a third-party service to run required checks on pull requests in a merge queue. Require deployments to succeed before merging: Require that deployments to certain environments, such as production, succeed before a pull request can be merged. You can get more information on branch protection rules and how to use them at GitHub repo branch protection documentation. When using branch protection rules, it's important to strike a balance between enforcing restrictions and requirements, and allowing your team to work effectively. You should consider the needs of your team and project, and choose rules that support the way your team works. You should also document the rules and make sure that everyone on the team understands how they work and follows them consistently. Tip 6: Maintain a Clean Commit History A clean commit history is crucial for effective collaboration and code management. It provides a clear record of the changes that have been made to the codebase, and helps to maintain a clean and stable codebase. It also makes it easier to understand the history of the codebase, and reduces the risk of conflicts and errors. There are several best practices that you can adopt to maintain a clean commit history, such as: Write descriptive commit messages: Write clear and descriptive commit messages that explain the purpose and context of the changes that have been made. Use atomic commits: Make small, focused commits that contain a single logical change. This makes it easier to understand the history of the codebase, and reduces the risk of conflicts and errors. Use meaningful commit titles: Use meaningful commit titles that summarise the purpose of the changes that have been made. Use consistent formatting: Use consistent formatting for your commit messages, such as using the imperative mood and keeping the first line to 50 characters or less. Use signed commits: Use signed commits to verify the authenticity of your commits and protect against tampering. For example, a good commit message looks like this:: git commit -m "Add user authentication mechanism to the inventory management system" It's bad practice to have vague messages such as: git commit -m "Fixed stuff" When maintaining a clean commit history, it's important to consider the needs of your team and project. You should choose practices that are simple, flexible, and scalable, and that support the way your team works. You should also document the practices and make sure that everyone on the team understands how they work and follows them consistently. Tip 7: Utilise .gitignore The .gitignore file is a simple and effective way to manage the files and directories that you want to exclude from version control. It allows you to specify patterns that match files and directories that you want to ignore, and prevents them from being added to the repository. To name a few, the .gitignore file is particularly useful for: Ignoring build artifacts: Ignore files and directories that are generated during the build process, such as log files, temporary files, and build artifacts. Ignoring sensitive information: Ignore files and directories that contain sensitive information, such as passwords, API keys, and configuration files. Ignoring user-specific files: Ignore files and directories that are specific to individual users, such as editor settings, local configuration, and temporary files. Ignoring large files: Ignore files and directories that are large and not necessary for version control, such as media files, binary files, and archives. Ignoring logs and caches: Ignore files and directories that are created as part of the logging and caching process, such as log files, cache files, and temporary files. Ignoring test files: You can use .gitignore to ignore files and directories that are created as part of the testing process, such as test results, test logs, and test artifacts. When using .gitignore, it's important to consider the needs of your team and project. You should choose patterns that are simple, flexible, and scalable, and that support the way your team works. You should also document the patterns and make sure that everyone on the team understands how they work and follows them consistently. You can get more information on .gitignore and how to use it effectively in the GitHub repo .gitignore documentation. Tip 8: Use GitHub Actions for CI/CD GitHub Actions is a a powerful feature of GitHub that allows you to automate you tasks through workflows. It provides a flexible and scalable way to build, test, and deploy your code, and helps you to maintain a clean and stable codebase. GitHub Actions is a big topic on it's own but to touch on few topics, you can use GitHub Actions to: Automate build processes: Build your code automatically whenever a change is made to the repository. Automate tests: Run your tests automatically whenever a change is made to the repository. Automate deployment processes: Deploy your code automatically whenever a change is made to the repository. Automate releases: Create releases automatically whenever a change is made to the repository. Automate documentation: Use GitHub Actions to generate documentation automatically whenever a change is made to the repository. Automate IaC: Automate infrastructure as code (IaC) tasks such as provisioning, configuring, and deploying infrastructure. Automate security checks: Automate security checks such as vulnerability scanning, dependency analysis, and code analysis. The list goes on, but the point is that GitHub Actions is a powerful tool that can help you automate many of the tasks that are involved in managing a codebase. It's important to consider the needs of your team and project. You should choose workflows that are simple, flexible, and scalable, and that support the way your team works. You should also document the workflows and make sure that everyone on the team understands how they work and follows them consistently. You can get more information on GitHub Actions and how to use them effectively from the official GitHub Actions documentation. Tip 9: Leverage Issue Tracking and Projects GitHub provides a powerful issue tracking system that allows you to manage and track issues, bugs, and feature requests. It also provides project status boards that allow you to manage and track the progress of your work. Github Projects can also help you to manage your work more effectively, and improve the collaboration and communication within your team. Issue tracking and Projects are useful for several reasons, including: Track issues and bugs: Track issues and bugs, and manage the process of fixing them. Track feature requests: Track feature requests, and manage the process of implementing them. Plan and prioritise work: Plan and prioritise your work, and manage the process of completing it. Manage releases: Manage releases, and track the progress of your work through milestones. Collaborate and communicate: Collaborate and communicate with your team, and improve the quality and security of your code. Labelling: Use labels to categorise issues, and make it easier to manage and track them. (e.g. bug, enhancement, help wanted). You can get more information on issue tracking and project boards and how to use them effectively from the official GitHub issue tracking documentation and GitHub projects documentation. Tip 10: Make use of GitHub security features GitHub provides a range of security features that can help you to improve the security of your codebase. These features can help you to identify and fix security vulnerabilities, and proactively protect your code from security threats and leaks. To name a few, you can use GitHub security features to: Security Alerts for Vulnerable Dependencies: Get alerts when your repository has a vulnerable dependency. Code and Secret Scanning: Scan your code for security vulnerabilities, secrets committed in code and coding errors. Dependabot Security/Dependency Updates: Automatically update your dependencies to the latest secure version using Github Dependabot. Security Policies and Advisories: Create and manage security policies and advisories for your repository. Dependency Insights: Get insights into the security/dependencies of your codebase, and identify areas for improvement using dependency graph. For more information and also to take a deeper dive into some of the security features and tooling available in GitHub natively I recommend checking an earlier blog post of this blog Series: Securing Your Code with GitHub Conclusion In this blog, we only touched on a few topics, and we discussed a few best practices for managing GitHub repositories effectively. But for more valuable information follow this link to get additional guidelines on how to set up your project for healthy contributions. I hope you have enjoyed this post and have learned something new. ❤ Author Like, share, follow me on: GitHub | X/Twitter | LinkedIn Learn to automate dependency management using GitHub Copilot, GitHub Actions, and Dependabot to eliminate manual checks, improve security, and save time for what really matters. Andrea Griffiths · March 5, 2025 Explore GitHub’s top blogs of 2024, featuring new tools, AI breakthroughs, and tips to level up your developer game. Laura Lindeman · December 30, 2024 As part of the GitHub for Beginners guide, learn how to improve the security of your profile and create a profile README. This will let you give your GitHub account a little more personality. Kedasha Kerr · September 9, 2024 Using Git in the CLI can improve your development speed and power. Here are our top eight commands for using GitHub via your command line. Michelle Duke · August 15, 2024 As part of the GitHub for Beginners guide, learn how to create pull requests. This will enable you to suggest changes to existing repositories. Kedasha Kerr · August 12, 2024 Take the next step in our GitHub for Beginners series and add code to your repository. Learn how to create branches and upload changes into a pull request. Kedasha Kerr · July 29, 2024 GitHub Staff Engineer Sarah Vessels discusses her philosophy of code review, what separates good code review from bad, her strategy for finding and reviewing code, and how to get the most from reviews of her own code. Sarah Vessels · July 23, 2024 The next step in our GitHub for Beginners series is learning how to add files and folders to your GitHub repository. Kedasha Kerr · July 8, 2024 Git started on your first repository in the third installment of GitHub for Beginners. Discover the essential features and settings to manage your projects effectively. Kedasha Kerr · June 24, 2024 The latest installment of GitHub for Beginners, where we cover the essential Git commands to get you Git-literate. Kedasha Kerr · June 10, 2024 GitHub Copilot increases efficiency for our engineers by allowing us to automate repetitive tasks, stay focused, and more. Holger Staudacher · April 9, 2024 GitHub Copilot is a powerful AI assistant. Learn practical strategies to get the most out of GitHub Copilot to generate the most relevant and useful code suggestions in your editor. Kedasha Kerr · March 25, 2024 Unlock the secret to organization and collaboration magic with our GitHub Projects tips and tricks roundup. Sara Verdi · March 21, 2024 Learn what GitHub Copilot can help your business achieve in this expert-guided GitHub Learning Pathway, featuring insights from tech leaders at top organizations. Ryan J. Salva · March 4, 2024 Learn how we’re managing feature releases and establishing best practices within and across teams at GitHub using GitHub Projects. Riley Broughten · February 28, 2024